Sujeevan
Vijayakumaran

DevOps

Wie IT-Projekte mit einem modernen Toolset und der richtigen Kultur gelingen

- Die DevOps-Kultur kennenlernen und umsetzen
- Effizientes und effektives Teamwork mit schlanken Prozessen
- Quellcode-Management, Continuous Integration, Deployment & Delivery, Observability, Sicherheit und Compliance

Rheinwerk
Computing

Liebe Leserin, lieber Leser,

Sie kennen sicherlich das Klischee des nerdigen Programmierers, der einsam vor seinem Bildschirm sitzt und nach jahrelanger stiller Tüftelei seine (mehr oder weniger) geniale Lösung präsentiert.

Dieses Bild entspricht jedoch nur selten der Realität moderner Softwareentwicklung. Ohne Teamarbeit, schnelles Feedback und den kontinuierlichen Austausch mit Kollegen, selbst über Abteilungsgrenzen hinweg, funktioniert inzwischen nicht mehr viel.

Dieses kooperative Vorgehen macht zweifellos die Softwareentwicklung einfacher, sie ringt jedoch zusätzliche Komplexität in ohnehin schon anspruchsvolle Aufgaben. Wer heute den (Quer-)Einstieg in die IT wagt, muss weit mehr beherrschen als nur das Programmieren. Man sollte nicht nur die Theorie von Netzwerken und Betriebssystemen kennen und wissen, wie man mit Cloud-Infrastrukturen arbeitet, sondern am besten beherrscht man auch noch die Automatisierung von Tests und kann eine CI-Pipeline verwalten. Es ist alles andere als trivial, dabei den Überblick zu behalten, geschweige denn alle wichtigen Tools in diesen Bereichen zu kennen.

In diesem Praxisbuch begleitet Sie Sujeevan Vijayakumaran auf dieser Reise durch sämtliche Phasen der Softwareentwicklung. In seiner Karriere als Softwareentwickler und Senior Solutions Architect bei GitLab hat er den gesamten Software-Entwicklungslebenszyklus kennengelernt und ist mit dem Arbeitsalltag großer IT-Projekte bestens vertraut. Während seiner Projektarbeit hat er die entscheidende Bedeutung einer soliden DevOps-Kultur für den Erfolg von Projekten hautnah erlebt und kann Ihnen aus erster Hand schildern, welche Vorteile sie im alltäglichen Chaos eines IT-Projekts bietet.

Ein Wort zum Abschluss: Dieses Buch wurde mit größter Sorgfalt geschrieben und hergestellt. Sollten Sie dennoch Fragen, Kritik oder inhaltliche Anregungen haben, freue ich mich, wenn Sie mit mir in Kontakt treten.

Ihr Christoph Meister
Lektorat Rheinwerk Computing

christoph.meister@rheinwerk-verlag.de
www.rheinwerk-verlag.de
Rheinwerk Verlag · Rheinwerkallee 4 · 53227 Bonn

Hinweise zur Benutzung

Dieses E-Book ist **urheberrechtlich geschützt.** Mit dem Erwerb des E-Books haben Sie sich verpflichtet, die Urheberrechte anzuerkennen und einzuhalten. Sie sind berechtigt, dieses E-Book für persönliche Zwecke zu nutzen. Sie dürfen es auch ausdrucken und kopieren, aber auch dies nur für den persönlichen Gebrauch. Die Weitergabe einer elektronischen oder gedruckten Kopie an Dritte ist dagegen nicht erlaubt, weder ganz noch in Teilen. Und auch nicht eine Veröffentlichung im Internet oder in einem Firmennetzwerk.

Die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen lesen Sie im Abschnitt *Rechtliche Hinweise*.

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf:

Exemplar Nr. fwnb-per9-cz6s-k4ax
zum persönlichen Gebrauch für
Thomas Bartholomäus,
minduxde,
thomas.bartholomaeus@mindux.de

Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

Lektorat Christoph Meister
Gutachten Dirk Deimeke
Korrektorat Friederike Daenecke, Zülpich
Herstellung E-Book Stefanie Meyer
Covergestaltung Mai Loan Nguyen Duy
Satz E-Book III-satz, Kiel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit und lesen Sie weiter auf den *Serviceseiten*.

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

ISBN 978-3-8362-9100-2 (E-Book)
ISBN 978-3-8362-9102-6 (Bundle)

1. Auflage 2024

© Rheinwerk Verlag GmbH, Bonn 2024
www.rheinwerk-verlag.de

Inhalt

Kapitel 1
Einleitung

»Nichts ist so beständig wie der Wandel.«

Dieses Zitat ist nun schon fast 2000 Jahre alt, aber relevanter denn je: Die Entwicklungszyklen von Produkten müssen heute schneller sein, um den sich ständig verändernden Anforderungen gerecht zu werden, auf die wir oft keinen direkten Einfluss haben. Ein Beispiel für diese Notwendigkeit war die COVID-19-Pandemie, die verdeutlichte, wie entscheidend Flexibilität ist.

Sie führte nicht nur dazu, dass man schnell vom Bürojob zum verteilten Arbeiten von überall aus wechselte, sondern betraf auch die Frage, wie die Arbeit überhaupt organisiert wurde. Plötzlich mussten gewohnte Arbeitsweisen und -prozesse von einem Tag auf den anderen angepasst werden. Diese Herausforderung betraf nicht nur reine IT-Unternehmen, sondern auch andere Organisationen, Behörden, Schulen und Vereine.

Unternehmen, die nicht nur ihre internen Arbeitsabläufe, sondern auch ihre Geschäftsmodelle schnell an veränderte Bedingungen anpassen konnten, erhielten einen erheblichen Wettbewerbsvorteil. Die Pandemie zeigte deutlich, wie sich Anforderungen rasch ändern können und wie wichtig es ist, sich agil anzupassen.

In dieser sich ständig wandelnden Umgebung spielt die DevOps-Kultur eine entscheidende Rolle. DevOps fördert die Zusammenarbeit zwischen *Entwicklung* (Dev) und *Betrieb* (Ops), um Prozesse zu beschleunigen, Qualität zu verbessern und gleichzeitig die Flexibilität zu erhöhen. Dieses Konzept ermöglicht es Unternehmen, sich effektiv an veränderte Anforderungen anzupassen und somit erfolgreich in einer dynamischen Welt zu agieren.

Sie suchen konkrete Beispiele? Eine bemerkenswerte gesetzliche Änderung, die wahrscheinlich alle Firmen in Deutschland traf, war der Beschluss vom 3. Juni 2020, die Mehrwertsteuer zum 1. Juli 2020 für ein halbes Jahr von 19 % auf 16 % zu senken. Zwischen diesem Entschluss und der Umstellung waren also nur knappe vier Wochen Zeit vorhanden, um nötige Änderungen zu implementieren, zu testen und überall dort auszurollen, wo es nötig war.

Auch zwei Jahre später, im Jahr 2022, gab es mit der Einführung des 9-Euro-Tickets für die Monate Juni, Juli und August wieder ein schönes Beispiel dafür, wie schnell doch einige Änderungen entschieden und umgesetzt werden können.

Persönliches Exemplar für Thomas Bartholomäus

So hat die jüngste Entwicklung gezeigt, wie anpassungsfähig Unternehmen und Organisationen sein können, wenn die Notwendigkeit besteht. Anstatt nur in Ausnahmesituationen schnell Anpassungen vorzunehmen, wäre es jedoch weitaus effizienter, wenn dies zur normalen Vorgehensweise gehören würde. Das hätte nicht nur Vorteile für die Unternehmen selbst, sondern vor allem auch für die Nutzer, die von diesen Diensten profitieren.

Und das gilt nicht nur für hippe IT-Startups, die an Apps und Webseiten arbeiten. Vielmehr ist es so, dass viele Unternehmen – und sogar Behörden – bereits heute im Kern IT-Organisationen sind, auch wenn sie nicht unbedingt reine Softwareprodukte anbieten. Zum Beispiel basiert das Hauptgeschäft einer Bank weitgehend auf der Software, die zur Verwaltung von Bankkonten und -diensten benötigt wird. In gewisser Weise ist eine Bank also eher ein Unternehmen der Software-Entwicklung mit einer zusätzlichen Banklizenz. Es ist daher nicht überraschend, dass einige Banken wie die ING sich selbst genau so beschreiben.

Wenn Sie sich bereits ein wenig in der IT auskennen, wird Ihnen diese Forderung bekannt vorkommen: Das ist doch der Grundgedanke der agilen Software-Entwicklung. Soll es hier nicht um DevOps gehen?

Genau bei diesem Punkt stellt sich bereits die erste gute Frage: Was ist Agilität, was ist DevOps, wie hängen beide miteinander zusammen und wo vielleicht auch nicht?

Aber treten wir noch einen Schritt zurück, denn die vielleicht wichtigste Frage ist eigentlich: Warum das Ganze eigentlich? Viel zu oft läuft man Trends jeglicher Art hinterher und macht Dinge, weil es ja schließlich alle so machen. Viel sinnvoller ist jedoch, die Frage nach dem *Warum* zu stellen und darauf basierend dann Änderungen einzuführen.

Tatsächlich gibt es noch sehr viel mehr Fragen, die es zu beantworten gilt, nicht umsonst haben Sie sich wohl dieses Buch angeschafft.

Dieses Buch teilt sich im Grunde in zwei Bereiche auf: Kultur und Technik. Wir betrachten zunächst die *Kultur*: Es werden Konzepte, Ideen, Vor- und auch Nachteile behandelt, ohne zu tief auf einzelne technische Tools eingehen zu müssen. Ganz ausblenden können wir die IT-Tool-Landschaft aber nicht. Da sie im stetigen Wandel begriffen ist, finden Sie einen Überblick, wie der Stand der Technik ist, mit Fokus darauf, was es zu beachten gibt und welche Konzepte zugrunde liegen.

Damit das Buch möglichst wenig trocken und theoretisch wird, betrachten wir das ganze Thema anhand einer fiktionalen Firma, die in Kapitel 3 vorgestellt wird. Diese Firma setzte bislang nicht auf DevOps-Prinzipien. Kapitel für Kapitel werden immer mehr DevOps-Prinzipien in die Firma eingeführt, um zu zeigen, wie eine DevOps-Transformation aussehen kann. Denn die wenigsten werden die DevOps-Kultur auf der grünen Wiese starten können.

© Rheinwerk Verlag, Bonn 2024

1.1 Kultur

Um eines gleich zu Beginn klarzustellen: Viel wichtiger als die letztendlichen Tools ist das Verständnis der DevOps-Kultur. Denn DevOps ist eine Kultur, also eine Art und Weise, wie im Team zusammengearbeitet wird. Viele haben ein falsches Verständnis von DevOps, weshalb es wichtig ist, dass Sie die DevOps-Kultur richtig verstehen, bevor wir uns in diesem Buch um technische Tools kümmern.

Der Begriff *DevOps* selbst ist ein Kofferwort aus »Development« und »Operations«. Es geht darum, die Wände zwischen dem Entwicklungsteam (»Development«) und dem Betriebsteam (»Operations«) einzureißen und zusammen am gemeinsamen Ziel zu arbeiten: an der kontinuierlichen Entwicklung und Bereitstellung der softwarebasierten Lösung.

In diesem Buch möchte ich beschreiben, was die DevOps-Kultur ausmacht. Ich beschreibe, was sie ist, und auch, was sie nicht ist oder wo sie sich mit anderen Arbeitsweisen überschneidet. Am wichtigsten ist allerdings: DevOps muss in der ganzen Organisation gelebt werden, und zwar sowohl ganz oben als auch unten in der Hierarchie.

Die größte Herausforderung für Sie und insbesondere für Ihre Organisation dürfte sein, dass der Schalter im Kopf umgelegt werden muss. Und das ist nicht einfach, denn es gibt bekanntlich viele Wege, die nach Rom führen. Dazu gehört auch, dass alle sich auf andere Aufgaben und Blickwinkel einlassen müssen: Entwickler müssen mal Bereitschaftsdienste leisten, Admins sollten wissen, was ein Pull Request ist, und für die Sicherheit und Zuverlässigkeit der Anwendung sind letztendlich alle verantwortlich.

1.2 Technik

Auch wenn DevOps im Wesentlichen eine Kultur ist und ihre Beschreibung allein schon das Buch relativ gut füllen würde, gehe ich trotzdem auf einige Tools aus der DevOps-Landschaft ein, damit Sie den heutigen Stand der Technik besser nachvollziehen können.

Allerdings geht es mir nicht darum, jedes einzelne Tool bis ins letzte Detail zu besprechen. Ganz im Gegenteil: Sinn und Zweck ist vielmehr, dass Sie die Konzepte der Tools verstehen, damit Ihnen klar wird, warum gerade diese Werkzeuge in der aktuellen DevOps-Tool-Landschaft genutzt werden. Zudem sollte Ihnen anschließend klar sein, warum einige Tools seltener verwendet werden oder auch wo und in welcher Kombination sie gerade bei ordentlicher Umsetzung von DevOps nützlich sein können.

Persönliches Exemplar für Thomas Bartholomäus

Wichtig ist zu verstehen, dass Technik lediglich ein Hilfsmittel zur Implementierung der Kultur ist. Die besten Tools helfen Ihnen nicht, wenn die Prozesse und die Kultur nicht gelebt werden. Gute Tools unterstützen so die Kultur, allerdings helfen gute Tools nicht bei schlechter Arbeitskultur.

1.3 Mein Weg zu DevOps und zu diesem Buch

Viele Freunde, Bekannte und Arbeitskollegen fragten mich, warum ich denn nun ein DevOps-Buch schreibe, schließlich, so der Tenor, ist das Thema doch schon ein alter Hut und ja bereits »fast durch«. Inzwischen behaupten ja sowohl kleinere als auch größere Firmen, dass sie DevOps machen. In der Realität sieht das alles allerdings ein wenig anders aus.

Meine Motivation für dieses Projekt hat ihre Wurzeln zu Beginn meiner IT-Karriere. Im Jahr 2011 startete ich ein duales IT-Studium bei einem kleinen mittelständischen Unternehmen und wollte Software-Entwickler werden. Zum Programmieren bin ich damals allerdings nie so wirklich gekommen, denn es gab viel mehr zu tun, um die Entwicklungsproduktivität zu verbessern.

In diesem Unternehmen gab es fast gar keine Automatisierungen, vieles wurde händisch zusammengefrickelt. Daran ging viel Arbeitszeit verloren, sodass weder die verschiedenen Teams gut zusammengearbeitet haben noch die Tools für das Bauen und Testen effizient benutzt wurden. Und hier schreibe ich nur von dem Entwickler-Workflow: Was dann mit der Software passierte, wenn sie ausgerollt wurde, erreichte das Entwicklungsteam kaum, denn das fertige Paket wurde quasi »über die Mauer« zu einem anderen Team eines Dienstleisters geworfen, das dann den Betrieb übernahm. Was dort geschah? Keine Ahnung, denn Feedback gab es nie.

Hinzu kam, dass es keine Umgebungen gab, in denen man die Software produktionsnah testen konnte. Ob eine Änderung funktionierte, war immer eine große Frage, die niemand so recht beantworten konnte. Als Notlösung zogen die Mitarbeitenden viele kleine Testumgebungen von Hand hoch, und es ging viel Zeit verloren, da diese Testsysteme immer wieder manuell aufgeräumt und neu installiert werden mussten. Richtig sauber waren die Testsysteme nie.

Für kleine oder mittelständische Unternehmen sind wahrscheinlich auch heute noch viele dieser Unzulänglichkeiten typisch – vor allem dann, wenn die Firmen schon etwas älter sind. Die Aussage »Aber das haben wir schon immer so gemacht!« hört man oft genug, und durch den fehlenden Willen, stetig besser zu werden und dazuzulernen, gibt es in solchen Firmen praktisch keine Fortbildungsmaßnahmen. Nach einigen Jahrzehnten Stillstand merkt man dann leider, dass das Leben der Entwicklerinnen und Entwickler unnötig schwer gemacht wurde, wodurch die Produktivität stark gelitten hat.

© Rheinwerk Verlag, Bonn 2024

Im Laufe der Zeit führte ich damals Automatisierungen ein, vereinfachte das Test-Setup für das Dev-Team, sorgte für automatisierte Test-Umgebungen und machte so den Entwicklungsalltag effizienter und angenehmer.

Ich stellte allerdings schnell fest: Dass es eine Automatisierung gibt, ist eine Sache. Die Nutzung und ihre Akzeptanz von weiter oben in der Hierarchie ist aber eine ganz andere. Denn auch wenn die meisten Automatisierungen mit wenig Aufwand hätten genutzt werden können, war es doch unumgänglich, dass sich die Leute damit auseinandersetzen. Das klappt nur, wenn Zeit dafür da ist und Prozessverbesserungen auch erwünscht sind.

Auch wenn ich DevOps als Begriff und Kultur damals nicht kannte – beides steckte schließlich selbst noch in den Kinderschuhen –, war mir schnell klar: Diese Kultur muss in dem Unternehmen fest verankert sein. Es muss der Wille herrschen, sich stets zu verbessern, komplexe Prozesse zu hinterfragen und Unnützes wegzuschneiden. Auch die Barrieren zwischen den Teams und den Aufgabengebieten müssen von der ganzen Organisation aktiv abgebaut werden, damit effizient gearbeitet werden kann.

Zur gleichen Zeit besuchte ich nacheinander zwei verschiedene Hochschulen: erst eine Fachhochschule und dann eine Universität, an der ich meinen Master absolviert habe. Dort gab es das eine oder andere Gruppenprojekt, in dem etwas gemeinsam in einem Team entwickelt und deployt werden musste. Dabei habe ich festgestellt, dass es bei den verschiedenen Teammitgliedern zwar ordentliche Programmierkenntnisse gab, allerdings hatte noch kaum jemand in einem »echten« Software-Projekt gearbeitet.

Das hieß also, dass zwar grundlegende Entwicklungskenntnisse in der Theorie vorhanden waren, allerdings kaum jemand Ahnung und praktische Erfahrung darin hatte, wie moderne Software-Entwicklung heutzutage aussehen sollte. Das fing mit grundlegenden Werkzeugen wie Git und CI-Tooling an und erstreckte sich weiter auf die richtige Verwendung von Reviews, Tests und Deployments.

Viele dieser Themen wurden zwar an einigen Stellen im Studium erwähnt, allerdings fehlte die praktische Erfahrung, um die vielen Einzelpunkte zusammenzuführen.

Eigentlich wäre ein solches Uni-Projekt eine gute Ausgangslage, um DevOps-Techniken auszuprobieren, denn man hat mit ihm eine leere Spielwiese, auf der man neue Prozesse und Tools ausprobieren kann. In der Realität ist man in Firmen eher mit Projekten konfrontiert, die älter sind als die meisten Studierenden selbst. Ein Sprung zu neuen DevOps-Techniken ist dann wesentlich schwieriger, denn ausgetretene Pfade kann man nur schwer verlassen. Wenn Sie sich gerade in der Ausbildung oder einem Studium befinden, möchte ich Ihnen daher mit diesem Buch etwas an die Hand geben, mit dem Sie schon frühzeitig die Grundlagen der modernen Software-Entwicklung nach DevOps-Prinzipien verstehen können.

Persönliches Exemplar für Thomas Bartholomäus

Nach meinem Studium hatte ich als Consultant die Möglichkeit, in weitere Unternehmen hineinzuschauen. Dort erkannte ich oft ein ähnliches Muster: festgefahrene, starre Strukturen und Prozesse, in denen Aufgaben und Probleme zwischen verschiedenen Teams hin und her geworfen werden, anstatt kollaborativ daran zusammenzuarbeiten.

Stets hatte dies mit einer Unternehmenskultur zu tun, die eine gute Zusammenarbeit gar nicht ermöglichte, sondern teilweise sogar aktiv verhinderte. Das ging fast immer zulasten der Endnutzer: Nützliche Funktionen und dringende Fehlerkorrekturen erreichten sie erst verspätet. Die Nutzer waren verständlicherweise mit den Produkten unzufrieden, und alle waren genervt. In Privatgesprächen unter Freunden und Bekannten in der IT-Szene hörte ich häufiger ähnliche Storys.

Seit Frühjahr 2020 bin ich als *Solutions Architect* bei GitLab tätig. Dort berate ich Konzerne aus Deutschland, Österreich und der Schweiz und helfe ihnen, die DevOps-Idee mit GitLab umzusetzen. Mit diesem Job wurde mir klar, dass DevOps zwar *irgendwie* fast überall angekommen ist, also auch bei den großen Firmen mit Tausenden von Mitarbeitenden, allerdings wird noch immer fast nur auf die Tool-Landschaft und damit auf die Technik geschaut. Die Kultur wird nur eher schwergängig angepasst, wenn überhaupt.

Alle diese Erfahrungen haben mir deutlich gemacht, dass in der praktischen Umsetzung von DevOps noch ein langer Weg zu gehen ist. In zahlreichen Kundengesprächen wird häufig ein falsches Bild von DevOps gemalt oder es viel zu sehr vereinfacht. Mit diesem Buch möchte ich den Ein- und Umstieg in die DevOps-Welt vereinfachen und so zu Verbesserungen an der Arbeitskultur, der Mitarbeiterzufriedenheit und der generellen Produktivität beitragen.

1.4 Zielgruppe

Das Buch richtet sich grundsätzlich an alle, die eine Rolle im Software-Delivery-Lifecycle innehaben. Dazu gehört explizit auch der Betrieb der Anwendungen, und mit ihm sind nicht nur die Techniker angesprochen, sondern auch die Business-Teams und Führungskräfte, die die Richtung vorgeben. Das Verständnis von DevOps ist für alle relevant, und alle müssen sich gemeinsam an den DevOps-Prinzipien ausrichten, nur dann kann die Umsetzung auch erfolgreich sein.

Das Buch richtet sich also an zwei überschneidende Gruppen. Zum einen soll zunächst erst ein generelles, klares Verständnis vermittelt werden, was DevOps ist und wie eine DevOps-Transformation gelingen kann. Dieser Teil des Buches ist für alle wichtig, die sich näher mit DevOps beschäftigen wollen.

Das sind nicht aber nur die Personen aus einem cross-funktionalen Team mit Fokus auf Entwicklung und Betrieb, sondern auch Personen im Management mit Entschei-

© Rheinwerk Verlag, Bonn 2024

dungsbefugnissen. Denn ohne Unterstützung von oben lässt sich DevOps nicht realistisch umsetzen. Und dafür wird ein tiefes Verständnis von den Prozessen und Dynamiken im kompletten Software-Delivery-Lifecycle benötigt, von der Planung über das Programmieren und die Auslieferung.

Die Ausführungen zu Tools und Techniken sind hauptsächlich für diejenigen interessant, die täglich in einem DevOps-Team arbeiten. Aber auch Entscheider können aus diesem Teil etwas an Erkenntnissen ziehen, da sie dann einige Tools besser einordnen können. Immer wieder habe ich gesehen, wie Firmen und deren Mitarbeiter nach Kubernetes rufen, ohne sich mit den dazugehörigen Fragen beschäftigt zu haben. Bevor Sie sich also aus dem Bauch heraus für einzelne Tools entscheiden, sollten Sie die Konzepte der einzelnen Tools und Techniken verstanden haben.

1.5 Die Struktur des Buches

Zu Beginn erfolgt im nächsten Kapitel eine Einordnung dessen, was DevOps eigentlich ist. Damit das Ganze auch anschaulich ist, wird in Kapitel 3 die *Schick Gekleidet GmbH* vorgestellt. Anhand dieser Firma möchte ich Ihnen zeigen, welche Tücken es in der »alten« Welt gibt und wie Sie es in der »neuen« DevOps-Welt besser machen.

In Kapitel 3 bis Kapitel 11 werden die einzelnen Stages des DevOps-Lifecycles behandelt, dabei wird immer wieder die Schick Gekleidet GmbH als Beispiel genommen. Die einzelnen Kapitel beinhalten sowohl einen kulturellen sowie einen technischen Teil. Mir ist es wichtig, dass die DevOps-Prinzipien anhand von anschaulichen Beispielen verstanden werden, damit Sie anschließend auch Tools und Techniken kennenlernen, die für die jeweilige Stage jeweils relevant ist.

Am Ende jedes Abschnitts folgt eine Zusammenfassung zur »Reflexion«. So möchte ich Ihnen die wichtigsten Aspekte in Erinnerung rufen, ein paar Einblicke aus der Praxis mit Ihnen teilen und idealerweise Ideen liefern, mit denen Sie über Ihre eigene Organisation und Technik nachdenken. Hoffentlich finden Sie direkt ein paar Ansatzpunkte, die Sie anpassen können.

1.6 Feedback

In diesem Buch geht es nicht nur um die Technik. Ganz im Gegenteil: Es geht mir um die Arbeitskultur. Und das ist meine größte Herausforderung. Während es bei technischen Problemen meist eine klare Lösung gibt, ist das bei diesem Thema nicht der Fall.

Wenn es um Kultur geht, gibt es sehr viele Wahrheiten, und bei Änderungen der Arbeitskultur können Sie vieles richtig, aber noch viel mehr falsch machen. Es gibt mannigfache Fallstricke und zahlreiche Fehler, die für Frust sorgen können. Und leider gibt es fast nie eine allgemeingültige Lösung, die für alle Firmen, alle Teams, alle

Persönliches Exemplar für Thomas Bartholomäus

Menschen passt. Letztlich hängen viele Faktoren in diesem Buch von diversen anderen Faktoren ab: von der Branche, der Firmengröße und der bisherigen Firmenkultur.

Und genau deshalb bin ich an Ihrem Feedback interessiert. Wenn Sie Punkte haben, von denen Sie denken, dass sie hier fehlen oder tiefer behandelt werden sollten, dann schreiben Sie mir doch einfach eine E-Mail an *mail@svij.org*, alternativ kurz gerne auch über Mastodon unter *@svij@ruhr.social* oder in LinkedIn unter *https://www.linkedin.com/in/svijee/*. Ich bin sehr daran interessiert, was bei Ihnen gut oder auch schlecht funktioniert hat und was Sie daraus gelernt haben.

1.7 Danke!

Ein solches Buch lässt sich ohne tatkräftige Unterstützung auf verschiedenen Ebenen – fachlich und nicht fachlich – nicht umsetzen. Entsprechend muss ich mich bei etlichen Leuten bedanken und dabei hoffen, dass ich niemanden vergessen habe.

In erster Linie bedanke ich mich bei Dirk Deimeke als Mentor und Ratschlaggeber, der mir immer ausgezeichnetes Feedback gegeben hat, damit das Buch so wird, wie es ist. Und das ging Hand in Hand mit der Betreuung meines Lektors Christoph Meister, der sehr viel Input geliefert hat, um das Buch immer besser zu machen.

Ebenso gilt mein Dank meinen Eltern, meiner Schwester Shrani, meinen Schwager Max und meiner Nichte Mira, die zuletzt leider viel zu wenig von mir hörten, während ich mit dem Schreiben beschäftigt war.

Mein Dank gilt weiterhin aktuellen und ehemaligen Arbeitskollegen und -kolleginnen von GitLab, die nicht nur wesentlich zu meiner DevOps-Karriere beigetragen haben, sondern von denen ich auch sehr viel Feedback zu einzelnen kleineren und größeren Punkten für dieses Buch erhalten habe.

Mein Dank gilt daher an Ralf Gronkowski, Vlad Budica, Kristof Goossens, Alexander Dess, Timo Schuit, Manuel Kraft, Martin Brümmer, Jörn Schneeweisz, Jörg Heilig, Sarina Kraft, Andrea Obermeier, Michael Friedrich, Sander Brienen, Ted Gieschen, Julia Gätjens, Christoph Parschau, Christoph Caspar, Simon Mansfield, Idir Ouhab, Christopher Allenfort, Dominique Top, Stefania Chaplin, James Moverley und Stephen Walters.

Viel gelernt habe ich in den vergangenen 15 Jahren in der deutschen Linux- und Open-Source-Community, maßgeblich von den Kolleginnen und Kollegen von ubuntuusers.de: Torsten Franz, Stefan Betz, Christian Klotz, Jörg Kastning, Sarah Blume, Philipp Schmidt, Christoph Volkert und Dominik Wagenführ.

Last but not least auch noch ein Dank an ehemalige Arbeitskollegen meiner ersten Arbeitsstelle, der otris software AG, namentlich Dr. Veit Jahns und Thomas Schmidt.

Ach ja, und natürlich den Hörerinnen und Hörern meines Podcasts TILpod, die jeden Monat hören und mitfühlen konnten, wie es um das Buch steht.

© Rheinwerk Verlag, Bonn 2024

Kapitel 2
Was ist DevOps?

DevOps ist schon lange kein Hype mehr. Viele Personen machen »irgendetwas mit DevOps« und haben ihre Meinungen und Einschätzungen dazu. Fragt man sie, was genau hinter diesem Wort steckt, hört man zwar oft zunächst eine Erklärung, was sich hinter diesem Begriff verbirgt und wofür DevOps gut ist, aber genaueres Nachfragen offenbart dann eher eine Mischung aus gefährlichem Halbwissen, einer losen Aufzählung von expliziten Technologien und Buzzwords.

Gut, dass Sie dieses Buch lesen: Sie scheinen Interesse am Thema zu haben, dürften sicherlich schon etwas darüber gehört haben und wollen sich jetzt ein genaueres Bild machen, ob DevOps auch was für Sie etwas ist. Gute Idee!

Falls Sie vielleicht etwas skeptischer sind, dann ist das natürlich auch gut. Ich bin kein großer Fan von Buzzwords und Superlativen. Und genauso halte ich es auch hier: Natürlich ist nicht alles super und schön, sobald man DevOps-Prinzipien umsetzt. Natürlich ist nicht alles schlecht, sobald mal keine DevOps-Prinzipien umsetzt. Natürlich ist auch weiterhin nicht alles DevOps, nur weil man DevOps dranschreibt.

Häufig kann ich es den Leuten gar nicht übelnehmen, wenn sie keine klare Vorstellung von der Idee hinter DevOps haben und dementsprechend skeptisch sind. Mittlerweile schreiben viele Firmen »DevOps« an ihre Produkte oder an die Job-Titel ihrer Mitarbeitenden, weil es gleich viel moderner klingt. Was früher »Agile« und »Scrum« war, ist heute »DevOps«. Hauptsache, es wirkt so, als ob man mit der Zeit geht.

Ob in solchen Firmen die DevOps-Umsetzung wirklich beispielhaft gelungen ist, kann man teilweise von außen erkennen, teilweise aber auch nicht. Mein Lieblingsbeispiel ist der »DevOps-Engineer«, den gefühlt jede Firma sucht, die sich moderner aufstellen will. Praktisch für diejenigen, die sich auf diese Stellen bewerben: Diese Jobs werden meistens anständiger bezahlt, als wenn kein DevOps dranstehen würde. Bitte verstehen Sie mich nicht falsch: Es gibt einige triftige Gründe, einen Job mit »DevOps-Engineer« zu betiteln! Leider steckt nur meist nicht das dahinter, was man erwartet. Dieses Thema wird in Abschnitt 14.3 näher betrachtet.

Steigen wir jetzt also in das Thema direkt ein, und schauen wir uns an, was *DevOps* überhaupt ist.

Persönliches Exemplar für Thomas Bartholomäus

2.1 DevOps: Das große Ganze

DevOps als Ganzes ist im Wesentlichen eine *Arbeitskultur*. Es ist keine Technik, kein Tool, keine Job-Bezeichnung und, ja, es ist im Grunde auch keine Team-Bezeichnung, obwohl es sich von den beiden Begriffen *Development* und *Operations* ableitet, mit denen klassischerweise die Entwicklung und der Betrieb bezeichnet werden. Um erfolgreich nach den DevOps-Prinzipien arbeiten zu können, müssen die Prinzipien dieser Kultur möglichst umfassend in der Firma gelebt werden.

Das heißt also: Nicht nur die einzelnen Teams sollten nach DevOps-Prinzipien arbeiten, sondern die ganze Firma, denn ansonsten sind ihre Erfolgschancen gering. Daher ist es nicht sinnvoll, wenn einzelne Teams als »DevOps-Teams« bezeichnet werden und andere Teile wiederum gar nicht existieren. Schließlich geht es um das große Ganze und nicht um einzelne Teams.

Außerdem ist DevOps unabhängig von den Tools, die verwendet werden. Wer sich wundert, warum ich das hier abermals erwähne: Das ist essenziell und wird – obwohl es bekannt ist – immer wieder ignoriert. Es ist nämlich andersherum: *Die Tools unterstützen die Prozesse, was wiederum die Menschen unterstützt.* Es wäre also ein falscher Ansatz, die Prozesse umzustellen, nur weil man überzeugt ist, dass man endlich Kubernetes in der Firma einführen sollte.

Während die agile Software-Entwicklung eine gute Basis darstellt, geht es bei DevOps darum, den ganzen *Software-Development-Lifecycle* (SDLC) zu betrachten. Dieser beginnt mit der Projektplanung, geht mit dem Programmieren und dem Ausrollen an die Endnutzer weiter, um anschließend das Feedback aus dem Betrieb für die weiteren Entwicklungsarbeiten zu nutzen. Eigentlich könnte man auch gleich vom *Software-Delivery-Lifecycle* sprechen, schließlich ist ein Development ohne Delivery reichlich witzlos.

Grundsätzlich geht es bei DevOps darum, den agilen Software-Entwicklungsprozess deutlich auszuweiten. Die agile Software-Entwicklung zielt darauf ab, dass nicht alles im Voraus haargenau geplant wird, sondern dass in kürzeren Iterationen und in Inkrementen gearbeitet wird, um flexibler mit den sich ändernden Anforderungen zurechtzukommen. Ein etwas tieferer Einstieg in die agile Software-Entwicklung erfolgt gleich in Abschnitt 4.1.

Obwohl DevOps eigentlich eine gelebte Kultur sein soll, kann man sich ihr natürlich theoretisch nähern und versuchen, sie durch Definitionen zu fassen. Es muss Ihnen aber klar sein, dass diese Zusammenfassungen immer nur Ideale und Abstraktionen darstellen, die Sie konkret an Ihre Umgebung anpassen müssen.

Dennoch ist es sinnvoll, diese Prinzipien zu kennen, da sie ein gutes Gerüst bilden, an dem Sie sich bei der Umsetzung orientieren können. In der Praxis haben sich zwei Methodiken und Definitionen etabliert. Das sind einmal die Grundwerte von DevOps, nämlich das *CA(L)MS-Modell* sowie die sogenannten *Drei Wege*, wobei der Begriff meistens im englischen Original genutzt wird: *The Three Ways*.

© Rheinwerk Verlag, Bonn 2024

Beide Aspekte werden Sie im Laufe des Buches immer wieder erkennen, und ich werde mich wiederholt auf diese Punkte zurückbesinnen, damit bei den jeweiligen Aspekten klar ist, wozu ein jeder gehört. Lassen Sie uns nun zunächst erst mal in die Prinzipien einsteigen, damit klar ist, worum es geht, ohne uns allzu sehr in den Details zu verlieren. Sowohl CA(L)MS als auch The Three Ways nähern sich DevOps auf unterschiedlichen Arten.

2.1.1 CA(L)MS

Das CA(L)MS-Modell enthält die Leitprinzipien von DevOps. Aber jetzt erst einmal einen Schritt zurück: Wofür steht überhaupt CA(L)MS?

CA(L)MS ist eine Abkürzung von vier oder fünf Werten:

- C für *Culture*
- A für *Automation*
- L für *Lean*
- M für *Measurement*
- S für *Sharing*

Die Thematik rund um »Lean« wird oft als optional betrachtet, weswegen die Grundwerte als CAMS und die Erweiterung als CALMS zu finden ist.

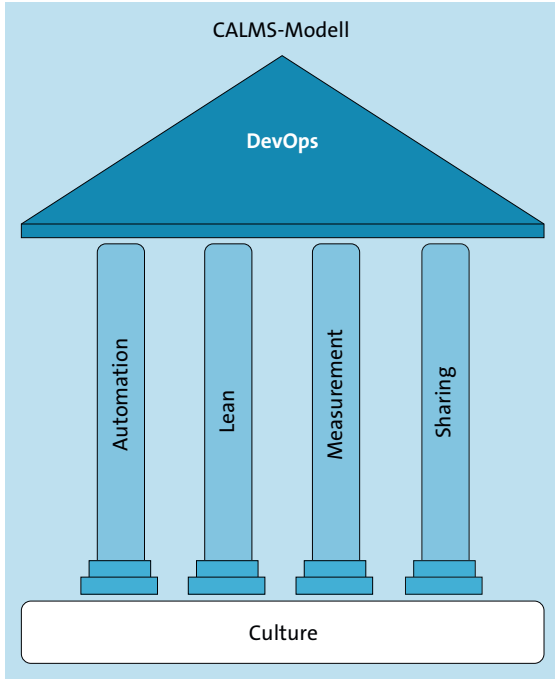

Abbildung 2.1 Das CALMS-Modell

Bei Ihrer DevOps-Transformation hilft Ihnen das CALMS-Modell, da es betont, dass alle Aspekte gleichmäßig aufgebaut werden und somit gemeinsam wachsen müssen. Sie stehen auf dem Fundament der Kultur, denn ohne diese geht es nicht.

Auf jeden dieser Punkte gehe ich hier zunächst einmal relativ oberflächlich ein. In den folgenden Kapiteln werde ich allerdings immer wieder auf die einzelnen Bestandteile des CALMS-Modells verweisen, damit klarer ist, was die einzelnen Aspekte jetzt mit DevOps zu tun haben und wo sie sich in dem CALMS-Modell wiederfinden.

C für Culture

Das **C** steht für *Culture*, also Kultur. Hier ist vordergründig die Arbeitskultur gemeint: Wie erfolgt die Zusammenarbeit im Team und innerhalb der Organisation, ganz unabhängig von den verschiedenen Rollen und Aufgabengebieten im Team?

Ein Team, das nach den DevOps-Prinzipien arbeitet, besteht stark vereinfacht gesprochen im Wesentlichen aus *Entwicklung* (Dev) und *Betrieb* (Ops), die zusammenarbeiten und nicht, wie so häufig, gegeneinander antreten. Anstatt dass beide Teams getrennt voneinander in Silos hocken und mit großen, hohen Wänden voneinander abgeschottet sind, setzen sich beide Gruppen bei ihren Aufgaben für ein gemeinsames Ziel ein.

Aber hier kommt es schon zum ersten Problem: Die Entwickler müssen neue Features entwickeln, die möglichst wenig Fehler aufweisen. Das Hauptaugenmerk des Betriebsteams liegt auf dem sicheren und stabilen Einsatz der Software, die vom Entwicklungsteam geschrieben wurde.

Die Kultur des Teams ist entscheidend, da alle Menschen aus dem Team nicht nur organisatorisch, sondern auch hinsichtlich der Ziele zusammengeführt werden müssen. Für beide Gruppen ist es wichtig, beide Aufgaben zu erfüllen.

Ein klassisches Entwicklungsteam kann die neuen Features nicht ausrollen, weil das Deployment zu kompliziert und fehleranfällig ist. Hier ist Teamwork gefragt, nicht »Works for me«.

Ein wesentlicher Aspekt ist das häufige Ausrollen von Releases, um die Zeit von der Fehlerkorrektur bis zum Ausrollen möglichst kurz zu halten. Für diese Aufgabe ist Automatisierung notwendig. Und diese hilft auch nur dann, wenn man sich auch wirklich traut, die Software auszurollen. Hier spielen noch weitere Aspekte hinein, wie ausführliche Tests, um sicher und regelmäßig Änderungen veröffentlichen zu können.

Und genau deshalb ist es wichtig, dass nicht nur diese beiden Gruppen an der Arbeitskultur arbeiten, sondern auch die Vorgesetzten, die Entscheider und das Management. Alle müssen sich darüber einig sein, dass auch Fehler akzeptiert werden, die auf dem Weg auftauchen werden. Diese Änderung lässt sich nicht von heute auf morgen erreichen – und das ist völlig in Ordnung.

© Rheinwerk Verlag, Bonn 2024

A für Automation

Weiter geht es mit dem **A** aus C**A**LMS: der *Automatisierung*. Wenig überraschend geht es hier darum, möglichst viele Aufgaben zu automatisieren.

Grundsätzlich besteht das Ziel nämlich darin, möglichst alles zu automatisieren, um keine oder nur sehr wenige manuelle Schritte selbst ausführen zu müssen, damit sich der »Speed« erhöht. Dazu gehören automatisierte Tests auf allen Ebenen, das Bauen der Artefakte und ihr Deployment sowie auch das Monitoring, nachdem das Deployment durchgeführt wurde.

Wichtig ist nämlich, dass eben nicht blind ausgerollt wird, sondern dass die Software gut getestet und mit einem gewissen Selbstvertrauen und auch Selbstbewusstsein automatisiert ausgerollt wird.

Problematisch ist hingegen, dass viele Firmen DevOps mit Automatisierung gleichsetzen. Ein großer Fehler, denn: Wer sich nur um die Automatisierung und weniger um die anderen Aspekte kümmert, automatisiert im Wesentlichen die Fehler komplett durch. So hört man häufig: »Wir machen doch DevOps! Wir haben eine CI/CD-Pipeline!«

Das ist schon mal ein guter Schritt in die richtige Richtung. Denn eine CI/CD-Pipeline ist auf der technischen Ebene ein elementarer Bestandteil von DevOps. Wichtig ist allerdings zu betonen, dass DevOps eben nicht nur aus einer CI/CD-Pipeline besteht. Und nur eine solche Pipeline zu haben, heißt noch lange nicht, dass man diese auch regelmäßig, sprich täglich mehrfach, nutzt.

L für Lean

Das Prinzip *Lean* stammt zwar ursprünglich aus der Produktion von Waren, wird aber mittlerweile auch an vielen anderen Orten eingesetzt, so auch in der Software-Entwicklung. Das berühmteste Lean-Modell kennen wir aus der Industrieproduktion von Toyota. Lean ist der Punkt, der in den DevOps-Prinzipien als optional betrachtet wird. Der Vollständigkeit halber behandele ich Lean hier im Buch dennoch, da es viele valide Punkte mit sich bringt, die im DevOps-Alltag nicht fehlen sollten.

Was ist Lean in diesem Kontext überhaupt? Grundsätzlich spricht man bei Lean von der Eliminierung von »Waste«. All das, was Müll oder Abfall ist, kann und soll weg. Damit ist konkret gemeint, dass all das, was keinen Mehrwert mit sich bringt, entfernt werden soll.

Eine weitere wichtige Komponente von Lean ist die kontinuierliche Verbesserung, die Erhöhung des Nutzens für die Kunden und die Fokussierung auf langfristige Ziele. Weitere Bestandteile von Lean sind die Menschen, die kontinuierlich Verbesserungen bewirken können und sollen, indem man sie dazu ermutigt.

In der Praxis sieht das etwa so aus, dass man so viel Vertrauen in das Team und die Software hat, dass man auch gut getestete Experimente auf Produktivsystemen fah-

ren kann. Damit dürfte man viel schneller ans Ziel kommen, wenn man weiß, was funktioniert und was nicht, ohne sehr lange diskutieren zu müssen.

Auch wenn Lean ursprünglich gar nicht aus der IT kommt: Es passt sehr wohl auch zur IT und infolgedessen zu den DevOps-Prinzipien.

M für Measurement

Der nächste Punkt ist **M** für *Measurement*. Auf Deutsch reden wir vom »Messen« von Metriken. Im Grunde geht es hier nun darum, dass man sein Urteil, was funktioniert und was eben nicht funktioniert, anhand von Metriken trifft. Das können größere, aber auch kleine Dinge sein. Wichtig ist aber vor allem, dass es quantifizierbare Daten als Metriken gibt, die entsprechend ausgewertet werden können.

Ein klassisches Beispiel ist etwa die Frage, wie lange es dauert, bis eine Forderung von Kunden (etwa eine Fehlerkorrektur) sowohl implementiert als auch auf dem Produktivsystem ausgerollt wird und somit live geht. Diese Zeit wird *Lead Time* genannt und ist eine übliche Metrik.

Teil der Lead Time ist die *Processing Time*, also die Zeit vom Commit der Entwicklerin, bis diese Änderung auf den Produktivsystemen ausgerollt wird. Wenn dies gemessen wird, können Sie eine bessere Aussage darüber treffen, wie gut die Prozesse rund um die CI/CD-Pipeline sind, um Änderungen nicht nur im Notfall, sondern auch im Normalfall rasch zu den Endnutzern ausrollen zu können.

Grundsätzlich wird auch noch unterschieden zwischen Prozessmetriken, Betriebsmetriken oder auch Business-Metriken, sodass idealerweise viele relevante Aspekte im Blick behalten werden und darauf basierend bewertet wird.

Im Laufe des Buches werden Sie noch sehr viele weitere Metriken kennenlernen, die in der DevOps-Welt wichtig sind, etwa die DORA-Metriken aus Kapitel 12.

S für Sharing

Der letzte Buchstabe von CALMS ist das **S** für *Sharing*. Hier geht es primär darum, dass alle voneinander lernen können. Dabei ist keineswegs gemeint, dass neue Technologien in der Organisation verbreitet werden sollen und alle plötzlich Experten für alles sind. Sondern es geht darum, dass Probleme und Lernerffekte dokumentiert und innerhalb und außerhalb der Organisation geteilt werden, damit möglichst niemand diese Fehler noch einmal begeht.

Dazu gehören Kleinigkeiten genauso wie größere Probleme, die zu einer langwierigen Downtime geführt hätten. Dabei kommt es darauf an, dass die Existenz von Fehlern grundsätzlich akzeptiert wird und man aus diesen lernt. Und das über Teamgrenzen hinweg. Das wirkt sich dann wieder auf die Kultur aus: Nur bei einem gesunden Umgang mit Fehlern lassen sich kleinere und größere Probleme ohne Angst mit dem eigenen Team oder auch mit der großen weiten Welt teilen.

© Rheinwerk Verlag, Bonn 2024

Fazit zum CALMS-Modell

Das CALMS-Modell ist eine gute Grundlage, um die Kernmerkmale von DevOps nachzuvollziehen und einordnen zu können. Sie sind eine gute Richtlinie, um einige kulturelle Änderungen in der Organisation einzuführen. Das ist, wie auch viele der noch im Buch folgenden Punkte, kein Modell, das für alle 1:1 passt.

Das CALMS-Modell kann allerdings nicht genutzt werden, um den Erfolg von DevOps zu messen, denn dafür existieren die DORA-Metriken, die in Kapitel 12 näher behandelt werden.

2.1.2 The Three Ways

Die weiteren Prinzipien sind »The Three Ways«. Im Gegensatz zu CALMS liegt ihnen der Gedanke zugrunde, dass alle DevOps-Prinzipien von drei Grundideen abgeleitet werden können. Sie werden gleich sehen, dass sich sehr viel mit dem CALMS-Prinzip überschneidet und sich manches wiederholt, allerdings mit einem etwas anderen Blickwinkel und somit mit einem anderen Fokus.

Grundsätzlich geht es bei den Drei Wegen darum, den Flow vom Entwicklungsteam zum Ops-Team – und somit vom Business zum Kunden – zu verbessern. Dazu gehören die Feedback-Schleife zurück vom Kunden zum Business, aber auch regelmäßige Feedback-Schleifen im ganzen Prozess.

In den folgenden Kapiteln werde ich, wie auch beim CALMS-Modell, immer wieder auf die Three Ways referenzieren, damit auch hier klar ist, wie die einzelnen Aspekte, Methoden und Arbeitsweisen mit DevOps zusammenhängen.

Die Drei Wege wurden maßgeblich von Gene Kim entwickelt, der sie im August 2012, also schon vor über 10 Jahren, in einem Blogpost präsentierte:

https://itrevolution.com/articles/the-three-ways-principles-underpinning-devops/

Dieser Post diente ihm als Grundlage für sein »The DevOps Handbook«.

The First Way: Systems Thinking

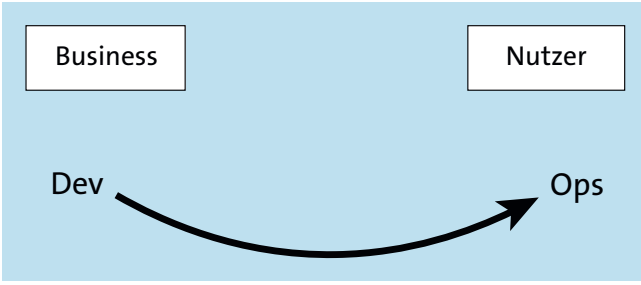

Abbildung 2.2 The First Way

Beim ersten Weg handelt es sich um das Verständnis des gesamten Entwicklungsprozesses als *System* und als *Flow*. Systeme sind in der Regel komplex und bestehen aus vielen verschiedenen Komponenten, die, je nach Organisation, von vielen verschiedenen Teams entwickelt und bereitgestellt werden müssen.

Wichtig ist daher, dass man sowohl das ganze System versteht als auch den Flow zwischen den einzelnen Komponenten des Systems. So ist ein Kriterium, dass die Änderungen von der Entwicklung (Dev) zum Betrieb (Ops) möglichst reibungslos fließen sollen. In der traditionellen Arbeitsweise läuft das in der Regel gar nicht so geschmeidig: Viel zu oft gibt es Hürden, die zu Verzögerungen führen und manuell überwunden werden müssen.

Letztlich kann man hier die Arbeit des Entwicklungsteams mit dem Business gleichsetzen: Dieses erbringt schließlich den Mehrwert, den die Kunden des Systems nutzen wollen. Und das geht nur, wenn die Anwendung auch entsprechend bereitgestellt wird. Das wiederum geht einher mit dem zügigen und geschmeidigen Weg von Dev zu Ops. Konkret heißt das also, dass Änderungen möglichst zeitnah ausgeliefert werden sollen, damit der Mehrwert zügig genutzt werden kann.

Damit das Ganze einfacher geht, muss die Größe der Änderungen angepasst werden. Wenn man riesige Änderungen aus einem halben Jahr Arbeit an die Kunden bringen will, hat man direkt mehrere Probleme an der Backe: Viele Änderungen wurden nicht richtig getestet und sind vielleicht auch noch nicht korrekt implementiert. Und es kommt auch viel zu oft vor, dass bereits auf der Basis dieser Änderungen weitere Verbesserungen umgesetzt wurden, was das Deployment verkompliziert. Deswegen hilft hier: Die Änderungen möglichst klein halten, um den Flow von Dev zu Ops und somit vom Business zum Kunden zu vereinfachen.

Zusätzlich müssen die Arbeiten, die erledigt werden, für alle sichtbar gemacht werden. Viel zu häufig sieht man innerhalb und außerhalb des Teams nämlich genau – nichts! Das bedeutet: Häufig werden die Fortschritte versteckt. Das Team versucht in so einem Fall, langsamen Fortschritt zu verschleiern, und hat Angst, dass mögliche Fehler von anderen entdeckt werden.

Vor allem zwischen den Teams, die voneinander abhängig sind, ist das problematisch. Das Qualitätssicherungsteam sollte sehen, wie der Fortschritt ist, um Qualitätsmängel schon frühzeitig beheben zu können. Genauso sollte das Betriebsteam schon frühzeitig sehen können, was relevant für den Betrieb ist, damit Fragen zur Infrastruktur nicht erst mit der angenommenen Fertigstellung nochmals aufgerollt werden müssen.

Dies betrifft nicht nur die eigentlichen Implementierungsarbeiten, sondern auch den generellen Fortschritt im Hinblick auf die Planung. Hier geht es also maßgeblich um den Status einzelner Aufgaben: Wird gerade daran gearbeitet, oder nicht? Wartet man gerade auf die Fertigstellung einer anderen Abhängigkeit?

© Rheinwerk Verlag, Bonn 2024

Nützlich ist hierfür ein Planungsboard. Ein *Issue-Board* ist ein klassisches Beispiel, häufig auch nach der Kanban-Methode implementiert. Ein einfaches Kanban-Board enthält drei Spalten mit Arbeitspaketen, die entweder »To Do«, »In Bearbeitung« oder »Erledigt« zugeordnet sind (siehe Abbildung 2.3). So erreicht man eine gute und einfache Sichtbarkeit. In der Praxis werden meist mehrere verschiedene Stufen genutzt, um auch Fälle wie »in Review«, »Wartend« oder »Blockiert« schnell sichtbar zu machen.

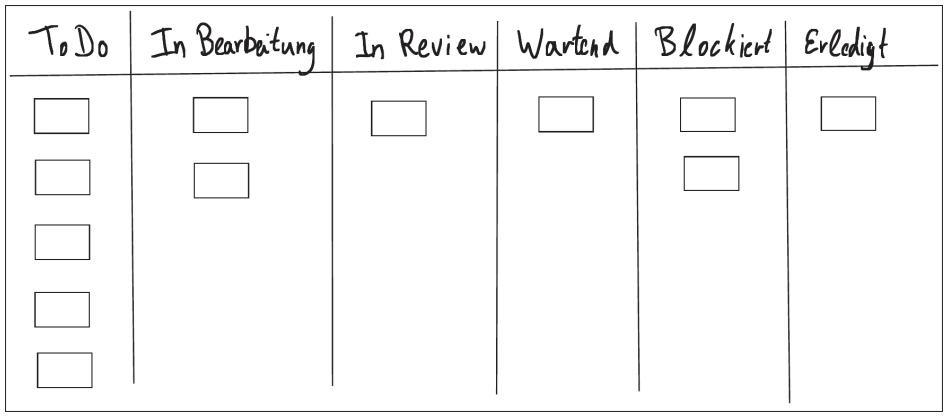

Abbildung 2.3 Dieses beispielhafte und vereinfachte Kanban-Board zeigt übersichtlich die Arbeitspakete in ihren verschiedenen Status an.

Grundsätzlich gilt daher: Immer schön die Arbeit sichtbar halten, denn nur so werden auch spätere Probleme frühzeitig erkannt! Das trifft nicht nur auf das eigene Team zu, sondern ist auch ein wichtiger Faktor zwischen verschiedenen Teams.

Das Ganze hilft vor allem dabei zu sehen, ob nicht doch gerade zu viel gleichzeitig entwickelt wird. Denn bei langen Entwicklungszeiten im traditionellen Ansatz gibt es sehr viele Aufgaben, die über einen sehr langen Zeitraum umgesetzt werden müssen. Es ist schlicht nicht möglich, alles gleichzeitig zu machen. Und da Ihre Arbeits- und Zeitressourcen im Team nun einmal begrenzt sind, sollte nicht an zu vielen Baustellen gleichzeitig gearbeitet werden.

Es muss also stets sichtbar sein, wie viele Arbeiten gerade parallel passieren. Wenn etwa zu viele Arbeitspakete im Status »Doing« sind, vielleicht sogar mehrere Aufgaben von einer Person erledigt werden müssen, dann ist dies ein guter Indikator dafür, dass es Probleme gibt: Entwickler sind überlastet, haben schlicht zu viele Aufgaben gleichzeitig oder es findet keine richtige Priorisierung statt.

Auch hierbei hilft das Board, indem es visualisiert, wie viele Tasks gerade in einem Status bzw. bei einer Person hängen. Diese Analyse geht also Hand in Hand mit der Sichtbarmachung der Arbeitsstände. Damit können Sie auf den zweiten Aspekt achten: auf die Limitierung der Arbeit, die gerade durchgeführt wird, um den Flow zu gewährleisten.

Persönliches Exemplar für Thomas Bartholomäus

Das Ganze wird fast nie problemlos funktionieren. Deswegen ist es immer wichtig, unnötige Prozesse zu entfernen und die Arbeit zu entschlacken. Und das in der ganzen Wertschöpfungskette. Dazu gehört die Eliminierung von Übergaben und Genehmigungen, die nicht gebraucht werden.

Viel wichtiger als die etwaigen Prozesse ist es allerdings, dass die Systeme so gebaut werden, dass man möglichst das ganze System versteht – wenn auch nicht alles bis ins letzte Detail. Genau deshalb heißt dieser erste Weg auch »Systems Thinking«, da an das gesamte System gedacht werden soll. Das Hauptaugenmerk liegt darauf, sowohl die Komplexität zu reduzieren als auch das System nachvollziehbar zu machen. Beides geht Hand in Hand.

Nur wenn der Weg des Produkts in den unterschiedlichen Schritten als ganzes System verstanden wird, können Probleme, die sonst im Verborgenen geblieben wären, schon frühzeitig von den Entwicklungsteams einzelner Module erkannt werden. Ansonsten würden die Erfahrungen, die das Betriebsteam beim Betrieb des Dienstes sammelt, nicht zu den Entwicklern vordringen, die ihrerseits nicht von diesen *Learnings* profitieren können.

Der Betrieb ist natürlich nur ein Beispiel: Es geht beim systemischen Denken darum, dass alle Bestandteile mit ihren diversen Abhängigkeiten beachtet, verstanden und überblickt werden müssen. Das geht nur, wenn das große Ganze als System grundsätzlich verstanden wird und man sich nicht in Insellösungen aus unterschiedlichen Kompetenzen verheddert.

Ein üblicher Praxisfall ist folgender: Das Business-Team stellt die Anforderungen auf, die an das Architekturteam gehen, das dann nach getaner Arbeit die Aufgabenpakete an die Entwicklungsteams gibt. Nach Abschluss der Arbeiten geht das Ergebnis an das Qualitätssicherungsteam und anschließend an das Betriebsteam über, das dann die Software in Betrieb nimmt.

Zwischen jeden dieser Teams gibt es Übergänge, die koordiniert werden müssen und an denen Abnahmen erfolgen. Das kostet unfassbar viel Zeit, aber viele Kenntnisse werden frühzeitig schon gebraucht und nicht erst zum späteren Zeitpunkt. Das ist ein ein großes Problem, das auch im Laufe des Buches immer wieder angesprochen wird.

Diese Übergänge müssen auf das Nötige reduziert werden. Dazu gehört auch, dass Entscheidungen, die nicht benötigt werden, ebenfalls entfernt werden. Häufig werden Genehmigungen benötigt, die von weiter oben in der Organisationsstruktur kommen, obwohl diese Entscheider kaum eine fundierte Aussage darüber treffen können, ob die Aufgaben wie gewünscht erledigt worden sind. Diese Information kommt ohnehin meist von den Teams selbst. Also warum nicht gleich den Teams diese Aufgabe übergeben?

Zur Reduzierung der Übergaben gehört auch, dass Automatisierung genutzt werden sollte, um wiederkehrende Aufgaben, die sonst nur zu weiteren Wartezeiten führen,

© Rheinwerk Verlag, Bonn 2024

einfacher und effizienter zu gestalten. Das offensichtlichste Beispiel sind die Deployments, die nicht mehr händisch gemacht werden. Das spart nicht nur Arbeit, sondern beugt auch möglichen Fehlern vor.

Es geht aber auch um viele kleinere Dinge, etwa um die Bereitstellung von Zugangsdaten für Systeme, wenn neue Mitarbeitende anfangen. Häufig sieht man, dass das Betriebsteam manuell tätig werden muss, um Accounts zu erstellen und Zugänge freizuschalten. Das ist ein klassischer Fall für eine Automatisierung!

Das geht auch Hand in Hand mit dem nächsten Punkt einher: Im Laufe der Entwicklung und des Deployments gibt es und wird es immer wieder Flaschenhälse geben, die aufgedeckt und beseitigt werden sollten, um den Flow zwischen Dev und Ops und somit vom Business zum Kunden geschmeidig zu halten. Alles, was hilft, sollte eingeführt werden, um den Fluss zu erleichtern. Das Ziel ist, dass wiederkehrende Probleme und Hürden erkannt und aus dem Weg geräumt werden.

The Second Way: Feedback

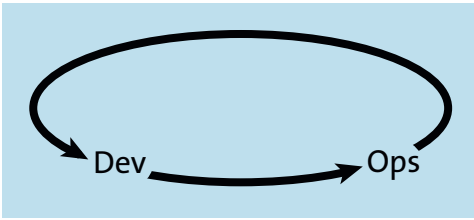

Abbildung 2.4 The Second Way

Beim ersten Weg geht vor allem darum, das ganze System zu verstehen, was dazu führt, dass man den Flow von Dev zu Ops und somit vom Business zum Kunden besser und zügiger handhabt. Beim zweiten Weg geht es genau um die umgekehrte Richtung: Die Wünsche und Probleme von Ops und den Kunden sollen zu Dev und somit zu den Business-Entscheidungen fließen.

Es geht also vordergründig darum, das hereinkommende Feedback sowohl konstant zu halten als auch zügig anzunehmen. Das steht im direkten Zusammenhang mit dem ersten Weg, denn zügiges und konstantes Feedback geht nur dann, wenn den Nutzern häufig Änderungen zur Verfügung gestellt werden. Das ist nur dann zuverlässig machbar, wenn diese Schritte klein sind.

Eine Voraussetzung dafür ist, dass man möglichst sicher mit komplexen Systemen arbeiten kann. Das ist schon prinzipiell keine einfache Aufgabe, denn es ist wenig überraschend schwierig, komplexe Systeme zu überblicken. Die Lösung besteht darin, ein interdisziplinäres Team aufzustellen, sodass möglichst jede Person einen weiten Blick auf das System erlangt – aber mit einzelnen Spezialisierungen.

Persönliches Exemplar für Thomas Bartholomäus

Die Aufgabe besteht also darin, dass Probleme, die zwangsläufig in Produktivsystemen auftreten werden, als Basis genommen werden, um Verbesserungen anzustoßen, damit es in der nächsten Iteration geschmeidiger läuft. Dieses Wissen muss innerhalb des Teams geteilt werden, sodass es allen hilft, das komplexe System zu verstehen und mit Fehlern umgehen zu können.

Da in einem komplexen System Module von verschiedenen Teams zusammenarbeiten, müssen Probleme sichtbar werden, damit man zusammen an den Problemen arbeiten kann: Nichts soll versteckt oder ignoriert werden.

Das automatisierte Testen ist ein zentraler Aspekt, um sicher mit den Systemen arbeiten zu können. Es gilt nicht nur, die zentralen Abläufe zu testen, sondern auch Punkte zu beachten, die man bisher nur vage angenommen hat – man spricht hierbei von *Edge Cases*. Auch hier hilft regelmäßiges Feedback, was aber nicht funktionieren kann, wenn nur in sehr seltenen Fällen ein Deployment durchgeführt wird. So würden dann Probleme, die etwa für das Software-Design relevant sind, erst verspätet entdeckt werden, was eine Korrektur zeitaufwendiger und somit sehr viel teurer macht.

Die Feedback-Schleife von ausgerolltem Code ist letztlich die langsamste Variante von Feedback. Es ist daher wichtig, dass es automatisierte Tests gibt, die Fehlverhalten möglichst zeitnah identifizieren: Fehler, die bereits während dieser Tests auffallen, führen nicht zu Problemen bei den Kunden und müssen nicht über Bug Reports oder Issue Tracker gesammelt und nachgestellt werden.

Automatisierte Tests gibt es auf verschiedenen Ebenen (darauf gehe ich in Kapitel 7 genauer ein):

- **Unit-Tests** – Diese Tests werden lokal und unmittelbar nach und während des Programmierens ausgeführt und liefern schnelles und einfaches Feedback.
- **Integrationstests** – Mit ihnen werden mehrere Komponenten im Zusammenspiel getestet.
- **Systemtests** – Sie prüfen das gesamte System und laufen entsprechend langsamer.
- **Akzeptanztests** – hierbei wird überprüft, ob das Ergebnis den Anforderungen entspricht

Wichtig ist wie immer: Automatisieren Sie möglichst viel und halten Sie Tests so einfach wie möglich.

Frühes Feedback zu den getätigten Änderungen ist essenziell, damit die Probleme nicht weitergeschoben werden, sondern das Übel an der Wurzel behandelt wird. Es geht um die Vermeidung und den Abbau von sogenannten *technischen Schulden*, die meist mit dem englischen Begriff *Technical Debt* beschrieben werden.

© Rheinwerk Verlag, Bonn 2024

Schnelles Feedback setzt sich nicht nur aus den verschiedenen Testtypen und häufigen Deployments zusammen, sondern besteht auch aus Reviews von Kollegen aus dem gleichen oder aus anderen Teams. Das Ziel ist, nicht nur Verbesserungen am Code vorzunehmen, sondern auch Kenntnisse aus der Umgebung, in der der Code laufen wird, von dem Kollegium abzugreifen.

The Third Way: Kontinuierliches Lernen

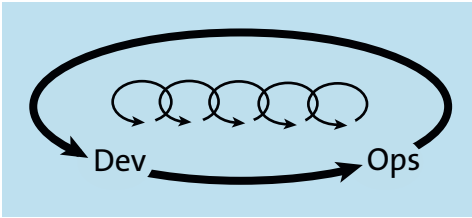

Abbildung 2.5 The Third Way

Der dritte und letzte Weg dreht sich um das kontinuierliche Lernen. Hier liegt Ihr Hauptaugenmerk darauf, stets Neues zu lernen und offen gegenüber anderen Techniken, Prozessen, Tools usw. zu sein. Das ist aber natürlich kein Selbstzweck, sondern zielt immer auf eine Problemlösung. Sie sollten sich also stets fragen, welche Probleme aktuell bestehen und welche Fehler immer wieder auftreten. Diese Stolpersteine sollten Sie dann nicht einfach akzeptieren oder ignorieren, sondern eben jene Stolpersteine müssen als Chance zur Verbesserung begriffen werden. Dieser Ansatz muss tief in der Kultur der Organisation verankert sein.

Ich beobachte vor allem bei älteren deutschen Firmen eine ungesunde Fehlerkultur. Wie wird in der Organisation damit umgegangen, wenn Fehler passieren? Dürfen überhaupt Fehler passieren? Wie sieht es mit Experimenten auf Live-Systemen aus, aus denen man sehr viel über das System lernen kann?

Firmen mit schlechter Fehlerkultur erkennt man an ihren vielen Silos. Dort denken Teams bzw. Abteilungen immer nur an ihre eigene Arbeitsweise und ihre eigenen Probleme. Ungereimtheiten in der Zusammenarbeit mit anderen Teams werden nicht direkt an der Quelle korrigiert, sondern es werden Workarounds gebaut.

Das hängt zum einen damit zusammen, dass man nicht weiß, wie die anderen arbeiten, oder damit, dass die Teams komplett voneinander abgeschottet sind und überhaupt keine Kommunikation stattfindet. Zum Schluss kann es sein, dass wichtige Entscheidungen und Hintergründe nicht geteilt und wie Privatgeheimnisse behandelt werden.

Das führt zu großen Problemen, wenn Fehler auftreten, etwa bei einer größeren Downtime. Schnell wird eine schuldige Person oder ein schuldiges Team gesucht, anstatt sich um das eigentliche Problem zu kümmern und eine Analyse der Grund-

Persönliches Exemplar für Thomas Bartholomäus

ursache anzustrengen, mit der die aktuelle Ursache korrigiert werden könnte. Dieses *Fingerpointing* sollte vermieden werden. Stattdessen sollte alles darangesetzt werden, das Problem gemeinsam zu verstehen und Lösungen zu erarbeiten, wie so etwas in Zukunft vermieden werden kann.

Fehler sollten demnach nicht vertuscht werden, sondern aktiv geteilt werden, damit alle etwas aus ihnen lernen können. So sollte auch niemand Angst haben, Fehler zu begehen – was aber auch nicht heißen soll, was man unvorsichtig sein sollte. Die richtige Balance zu finden, ist hier die große Kunst.

Wichtig ist vor allem auch, wie mit dem Melden von Fehlern umgegangen wird. Das ist sogar noch ein Schritt vor der eigentlichen Fehlerbehandlung. Werden Fehler ernst genommen oder werden sie lieber ignoriert? Oder wird gar das Aufdecken des Fehlers kritisiert?

Ein Aspekt, den ich in den letzten Abschnitten schon zwischen den Zeilen angerissen habe, ist die Organisationsstruktur in Firmen und die Art und Weise, wie Informationen gehandhabt werden. Häufig sieht man, dass sehr viele Organisationen ihre Teams komplett voneinander abschotten, sodass weder Code noch Projektmanagement eingesehen werden können. Informationen werden also selbst innerhalb der Organisation streng geheim behandelt, sodass niemand etwas sehen soll.

Dabei gibt es häufig gar keinen Grund dafür! Dieses Vorgehen verstärkt in der Regel den Silo-Gedanken, allein durch die Organisationsstruktur von Informationen. Transparenz schafft allgemein betrachtet viel mehr Vorteile, als sie Nachteile mit sich bringt. Ausnahmen gibt es hier sicherlich. Ein wichtiger Punkt, auf den ich im Laufe des Buches immer wieder eingehen werde, ist die Transparenz in allen Belangen des Software-Development-Lifecycles, denn ohne sie ist eine enge Zusammenarbeit weitestgehend unmöglich.

Das Lernen ist aber nicht nur ein organisatorischer Aspekt. So lässt sich auch am »lebenden Objekt« lernen: Durch das regelmäßige Ausrollen der eigenen Änderungen, auch wenn sie sehr klein sind, kann man hervorragend Teilaspekte testen und die Erkenntnisse daraus anschließend in die weitere Entwicklung einfließen lassen.

Aber auch das ist nicht nur eine technische Umsetzung, sondern muss vom Führungspersonal auch entsprechend gedeckt und unterstützt werden: Kein normales Entwicklungsteam würde wohl Experimente auf Live-Systemen ausführen (auch wenn die Entwicklungen gut getestet und somit ungefährlich sind), wenn diese Experimente von weiter oben weder abgesegnet und noch gewollt sind.

Dazu gehört auch, dass neue Erkenntnisse und Informationen, wie einzelne Teams kleinere und größere Fehler behoben haben, immer wieder auch in der ganzen Organisation geteilt werden sollten. Das kann etwa durch Blogposts oder Vorträge geschehen. Solche Aktivitäten können dabei helfen, zukünftige Fehler oder Ineffizienzen zu

© Rheinwerk Verlag, Bonn 2024

vermeiden, da sich schon andere Teams mit dem Problem befasst haben und diese Informationen wiederverwendet werden können.

2.1.3 The Three Ways und CALMS

Und das sind dann auch schon »The Three Ways« und CALMS. Wie Sie hoffentlich erkannt haben, ähneln sich beide sehr stark, setzen bei der Erläuterung aber einen anderen Fokus. Die Prinzipien beider Ansätze werden sich durch das ganze Buch ziehen. Ich habe bewusst auf viele Beispiele verzichtet, um den Einstieg möglichst kurz zu halten. Im Laufe des Buches werden Sie also immer wieder auf die hier vorgestellten Aspekte und Prinzipien stoßen und sie wiedererkennen.

Reflexion

In diesem Unterkapitel haben Sie gelernt, was das CALMS-Modell ist, und auch, was »The Three Ways« eigentlich genau ist. Eines der Hauptprobleme beim CALMS-Modell ist, dass viele Organisationen sich rein auf die Automatisierung verlassen und wenig bis gar nicht in die anderen Aspekte investieren.

Mein Rat lautet daher: Machen Sie sich Gedanken, wo Sie mit Ihrem Team und Ihrer Organisation stehen. Fällt Ihnen direkt auf, welche Aspekte besser gehandhabt werden können und wo es Optimierungsbedarf gibt? Die nächsten Kapitel gehen tiefer auf die relevanten Details ein.

2.2 Missverständnisse rund um DevOps

Da Sie nun die Grundideen hinter DevOps kennen, werden Sie mir sicherlich zustimmen, dass sich dies alles perfekt anhört: Kürzere Wege, einfachere Prozesse, mehr Kommunikation, besserer Umgang mit Fehlern – selbstverständlich wollen das alle, niemand würde hier widersprechen.

Mit dieser Auflistung von Tugenden und guten Ideen ist es aber nicht getan: Die Theorie muss in die Praxis umgesetzt werden. Und »umgesetzt« bedeutet hier: Diese Tugenden und Ideen müssen gelebt werden und in die DNA Ihres Teams und der Organisation übergehen. Tatsächlich ist das viel einfacher gesagt als getan, denn natürlich läuft beim Umsetzen der DevOps-Idee häufig einiges falsch.

Das fängt schon damit an, dass man DevOps als Ganzes nicht versteht oder nur Teilaspekte einbaut, ohne zu begreifen, worum es dabei geht: Nein, ein gemeinsamer Slack-Channel des Dev- und des Ops-Teams reicht nicht aus, um jetzt »DevOps« zu machen.

Persönliches Exemplar für Thomas Bartholomäus

In diesem Abschnitt möchte ich schon einmal thematisieren, was häufige Fehler und Missverständnisse sind und was man beachten sollte, damit Sie sich dann im nächsten Kapitel anschauen können, wie die Praxis in der fiktiven Beispielfirma des Buches ausschaut.

2.2.1 Zu starker Fokus auf die Automatisierung

Immer wieder sehe ich Folgendes: Firmen und Teams konzentrieren sich beim Umstieg in die DevOps-Welt auf die Automatisierung, ohne die anderen Aspekte anzuschauen, oder sie setzen einen viel zu geringen Fokus auf diese. Am Ende haben diese Firmen von CALMS dann häufig nur das **A** erreicht, ohne die passende Kultur, aber mit ganz viel *Waste*. Es fehlen Metriken, und *Sharing* ist weiterhin ein Fremdwort. Abbildung 2.6 zeigt, dass sie auf diese Weise kein solides Gebäude errichtet haben.

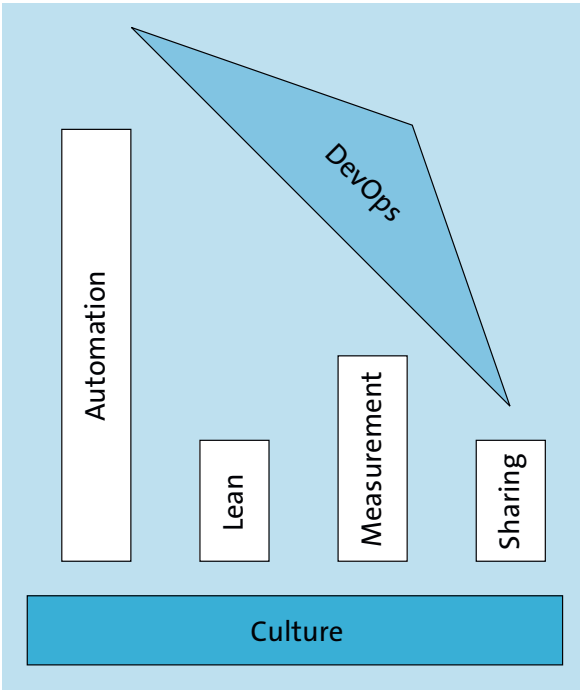

Abbildung 2.6 Eine starke Automatisierung in DevOps ist zwar wichtig – aber definitiv nicht das Einzige, denn sonst führt das Ungleichgewicht zu weiteren Problemen.

In der Praxis sieht man hauptsächlich eines: Es werden *DevOps-Engineers* eingestellt. Diese schreiben und warten eine CI/CD-Pipeline – und das war es dann auch schon. Zwar ist eine CI/CD-Pipeline ein elementarer Bestandteil des technischen Teils von DevOps, allerdings ändert sie wenig an den anderen Aspekten: Man arbeitet genauso weiter, wie man es auch schon zuvor getan hat.

© Rheinwerk Verlag, Bonn 2024

Nicht selten sehe ich CI/CD-Pipelines, die zwar in der Theorie ausgeführt werden können, aber nur, nachdem ein Ober-ober-Chef das Deployment abgesegnet hat. Und das, obwohl dieser Chef weder an der technischen Umsetzung beteiligt war noch die Qualität des Projekts im Ganzen selbst bewerten kann. Damit fehlt ein wichtiger Aspekt, nämlich die Übernahme von Verantwortung durch das Team.

Damit hat man das Schlechteste beider Welten erreicht: schwerfällige manuelle Prozesse und zusätzlich noch technischer Overhead durch die Werkzeuge einer CI/CD-Pipeline, die nun zusätzlich zum eigentlichen Produkt gewartet werden muss.

2.2.2 Mit DevOps, aber ohne Tests!

Ein Vorurteil, das man häufig hört, besagt, dass man mit DevOps viel schlechteren Code produziert, da die Änderungen ja viel zu schnell ausgerollt werden. Und wenn etwas schnell ist, dann muss es ja zwangsläufig schlecht und von niedriger Qualität sein. Vertreter dieser Ansicht gehen davon aus, dass DevOps eine schlechtere Qualitätssicherung bedeutet, da neuer Code einfach direkt auf die Produktivumgebungen geworfen wird, nach dem Motto: Die User können doch testen, ob alles funktioniert.

Typische Bananen-Software also: Das Programm reift beim Kunden.

Das stimmt so natürlich nicht. Wenn es richtig gemacht wird, ist das Gegenteil der Fall, nämlich, dass in Teams, die nach DevOps-Prinzipien arbeiten, viel strukturierter und gezielter getestet wird, da sehr viel automatisiert ist.

Am Ende eines Entwicklungsstrangs sind viel mehr Tests ausgeführt worden als in traditionellen Entwicklungszyklen. Hier geht es also darum, dass alle Hebel in Gang gesetzt werden, um einen sicheren Fallschirm zu haben. Dann traut man sich auch, zu springen bzw. die Änderungen auszurollen.

2.2.3 Falsches Verständnis der Teamstrukturierung

Der Klassiker unter den Missverständnissen von DevOps betrifft die Teamstrukturierung. So heißt es meistens:

> *DevOps heißt doch einfach nur, die Development- und Operations-Teams zusammenzuwerfen!*

Wobei, eigentlich ist das ja noch nahezu richtig, wenn auch mit diversen Einschränkungen. Häufiger hört man eher:

> *Das Entwicklungsteam übernimmt jetzt auch die Ops-Tätigkeiten. Admins brauchen wir nicht mehr!*

Nein, auch das ist kein DevOps. Admins benötigt man auch zukünftig. Die Aufgaben und Zuständigkeiten von allen beteiligten Personen werden sich im DevOps-Umfeld

allerdings verändern. Sie arbeiten dann eben nicht mehr so, wie es im klassischen Sinne bekannt ist (siehe Abbildung 2.7).

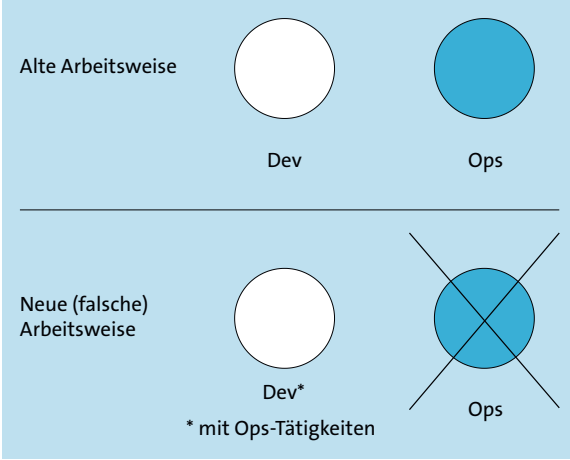

Abbildung 2.7 DevOps heißt eben nicht, dass die Entwickler nun die Betriebstätigkeiten übernehmen und somit Admins völlig unnötig werden.

Weiterhin gibt es vereinzelt auch separate *DevOps-Teams*, die zwischen Dev und Ops sitzen. Und zwar dauerhaft. Während der Übergangsphase ist das eine mögliche, vielleicht sogar gute Idee. Aber bestimmt nicht auf Dauer, denn das ist nämlich gerade nicht Sinn und Zweck der Sache!

Es gibt noch zahlreiche weitere Fehlinterpretationen, was die Teamstrukturierungen angeht, die ich im Laufe des Buches ansprechen werde.

Im Kern geht es ja grundsätzlich darum, das Entwicklungsteam und das Betriebsteam zusammenzuführen und die Stärken und Schwächen der jeweiligen Rollen zu verbessern bzw. auszumerzen. Aber die Frage, die sich hier stellt, lautet: Wie? Und: Wer noch?

Was ist mit dem Qualitätssicherungsteam? Was ist mit dem Security-Team? Was ist mit dem Business-Team, also den Leuten, die letztlich die Business-Entscheidungen treffen? Ja, auch die Finanzen spielen mit hinein: In welchen Bereich stecken wir dieses Team also wie mit rein?

2.2.4 Nicht alle Wände niederreißen

Nach DevOps-Prinzipien zu arbeiten, heißt nicht, dass *alle* alten Prozesse und Gewohnheiten über den Haufen geworfen werden müssen. Genau das ist das nächste Problem, das häufig sichtbar wird. Im DevOps-Kontext spricht man häufig von »Break down the wall«, »Tear down the fences« oder »Remove the silos«.

© Rheinwerk Verlag, Bonn 2024

Es geht also um den Abbau der Hindernisse zwischen den verschiedenen Teams. Ein typischer Fehler ist, dass wie in Abbildung 2.8 zwar die Silowände zwischen Dev und Ops heruntergerissen werden, aber nicht oder nur unzureichend zwischen den anderen Teams außerhalb der Technik.

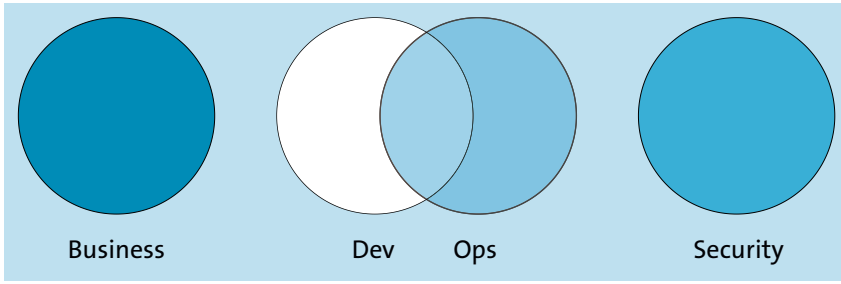

Abbildung 2.8 Die Säulen zwischen Dev und Ops wurden zwar niedergerissen, aber was ist mit den übrigen Teams?

Nur gemeinsam lässt sich ein Kulturwandel schaffen, und das betrifft alle Rollen und Aufgabengebiete in der Organisation. So sollten Führungskräfte die Business-Entscheidungen auch anhand des Feedbacks der Entwicklungs- und Betriebsteams umsetzen, da diese am besten beurteilen können, was geht oder was Probleme bereitet.

Oder denken Sie an die Finanzen: Je geringer die Barriere zwischen der Entscheidungsebene, die über die Budgets befindet, und der Technik ist, desto besser können fundierte Entscheidungen getroffen werden. Ein konkretes Beispiel ist etwa eine Migration des Rechenzentrums in die Public Cloud. Ein solches Vorhaben ist nicht nur für die Techniker und die Führungskräfte relevant, sondern sorgt auch für ein anderes Abrechnungsmodell, dessen Vor- und Nachteile diskutiert werden müssen. Keine Abteilung kann diese Entscheidung allein treffen.

Daher gilt: Nicht nur die technischen Rollen müssen zusammengeführt werden, sondern auch die anderen Aufgabenbereiche. Eine DevOps-Kultur, die nur im Maschinenraum gelebt wird, ist zum Scheitern verurteilt.

2.2.5 Tools über Prozesse über Menschen

Das nächste Problem ist, dass häufig als Erstes Tools angeschafft werden, um »DevOps« zu starten und umzusetzen. Danach werden auch noch Prozesse angepasst, und erst danach wird damit begonnen, die Menschen abzuholen. Zwar benötigt man alles – Tools, Technik, Prozesse –, aber die Umsetzung muss genau umgekehrt angegangen werden.

Der Grundsatz heißt immer »People over Processes over Tools«. Das bedeutet, die Leute kommen vor den Prozessen und diese wiederum vor den Tools, wie schon zu

Persönliches Exemplar für Thomas Bartholomäus

Beginn dieses Kapitels erwähnt. Dieser Leitsatz kommt aus Scrum und hat eine wichtige Bedeutung in DevOps (siehe Abbildung 2.9).

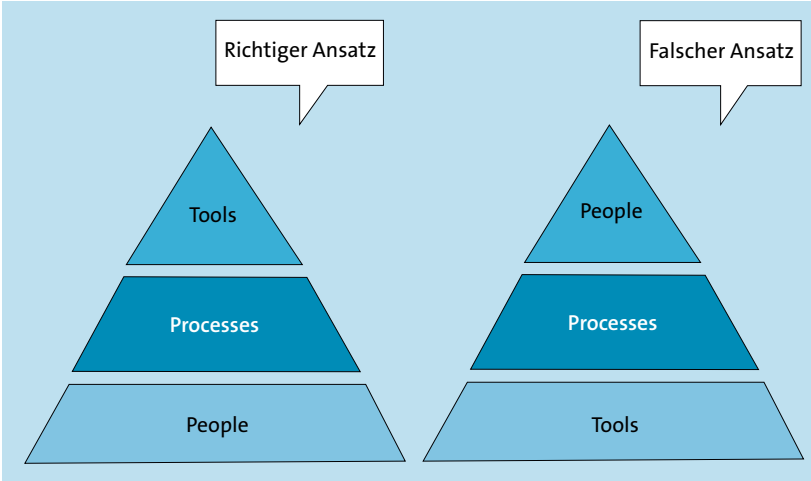

Abbildung 2.9 Die Menschen im Team bilden die wichtigste Grundlage. Erst danach kommen die Prozesse und die Tools – und eben nicht umgekehrt!

Die eingesetzten Tools sind zwar wichtig und helfen, DevOps-Ideen umzusetzen, aber sie bringen nichts, wenn die Prozesse schlecht sind. Und auch wenn Ihre Prozesse weltweit führend sind, helfen sie nicht, wenn die Mentalität der Personen im Team bzw. in der Organisation noch immer »falsch« ist.

Im ersten Schritt müssen Sie also die Menschen in den Teams Ihrer Organisation abholen. Erst dann sollten Sie sich um die Prozesse und anschließend um die Tools kümmern. So ist es zwar schön, wenn eine Firma eine CI/CD-Pipeline hat, um Änderungen integrieren und auch ausrollen zu können; diese bringt aber nicht sonderlich viel, wenn jeder Zwischenschritt von etlichen Vorgesetzten abgenickt werden muss. Etliche Approvals für ein Deployment einzuholen, dauert nicht nur ziemlich lange, es zeigt auch, dass man dem Team und somit den Menschen dahinter nicht vertraut. Und das, obwohl diese ja die Hauptarbeit erledigen und den Fortschritt des Projekts sowie seine Qualität am besten einschätzen können. Eigenverantwortliche Teams sind jedoch eine Kerneigenschaft von DevOps!

Aber auch dieses Vertrauen muss langsam aufgebaut werden. Und das ist weder super einfach noch geht es von heute auf morgen. Vertrauen lässt auch nur mit der Unterstützung von ganz oben entwickeln. Wenn dort das Verständnis und das Vertrauen fehlen, kann man sich eine gut funktionierende Organisation, die nach DevOps-Prinzipien arbeitet, schon fast abschminken.

Wie man Menschen abholt, wird in den nächsten Kapiteln zwar immer wieder erwähnt, aber ein tieferer Blick folgt in Kapitel 12.

© Rheinwerk Verlag, Bonn 2024

2.2.6 1:1-Kopien von Arbeitsweisen anderer Firmen

Viele Firmen und Organisationen teilen – ganz im Sinne von DevOps – ihre Erfahrungen rund um die Adaption von DevOps auf Konferenzen und in Blogbeiträgen. Das ist gut und auch wichtig, schließlich ist das gegenseitige Lernen nicht umsonst eines der Kernprinzipien von DevOps.

Was Sie allerdings nicht machen sollte, ist das Kopieren von Arbeitsweisen, Strukturen oder Tools anderer Firmen ohne jegliche Anpassung. Jede Firma und jede Organisation ist unterschiedlich. Jede hat ihre eigenen Herausforderungen, die es zu meistern gilt. Viele haben ihre eigenen Compliance-Richtlinien, die teilweise gesetzlich vorgegeben sind.

Sie werden daher in diesem Buch keine einfache Checkliste finden, die Ihnen in ein paar kurzen Schritten vermittelt, wie man »DevOps« einführt. Es wäre schön, wenn das so einfach ginge, aber dies ist ein großer Fehlschluss.

Es geht nämlich gerade nicht um ein paar allgemeingültige Punkte, die Sie auf Ihrem Zettel mit einem Häkchen versehen können, und schon haben Sie eine zertifizierte DevOps-Umgebung.

Ich möchte Ihnen vielmehr vermitteln, was zu einer gelebten DevOps-Kultur gehört, und kann Ihnen daher keine Abkürzungen liefern. Zu Beginn von Kapitel 12 gebe ich Ihnen trotzdem einen Einblick, wie die ersten Schritte einer DevOps-Transformation aussehen können. Aber wie bereits erwähnt: Verstehen Sie dies keineswegs als Checkliste, sondern fassen Sie es als Anregung Ihrer Gedanken auf!

Wichtig ist daher, dass Sie viele Aspekte von anderen Firmen lernen, ohne deren Vorgehen 1:1 zu kopieren. Häufig höre ich in Gesprächen, dass man ja dies oder jenes genau so tut, weil es ja schließlich Google oder Amazon oder Netflix oder irgendein amerikanisches Start-up auch so machen.

Grundsätzlich ist es löblich, sich die Arbeitsweisen von großen Firmen wie Google anzuschauen und zu adaptieren. Allerdings sind die wenigsten Firmen, vor allem in Deutschland, irgendwie mit Big-Tech-Unternehmen aus dem Silicon Valley wie Apple, Amazon, Alphabet (Google) oder Meta (Facebook) vergleichbar.

Viele Faktoren gehören dazu: Firmengröße, Firmenalter und somit das Alter der Code-Basis, die Bereitschaft zur Veränderung, aber vor allem die unterschiedliche Industrie. Eine Firma wie Google muss und kann anders arbeiten als ein deutscher Autohersteller oder eine schweizer Bank, und das liegt nicht nur an den gesetzlichen Regularien. Hier prallen gleich mehrere Industrien mit ihren eigenen Herausforderungen aufeinander.

Verstehen Sie mich nicht falsch: Von den Big-Tech-Unternehmen kann und sollte man hier in Deutschland sehr viel mehr lernen! Allerdings sollte man nicht versuchen, etwas zu kopieren, sondern sollte es stattdessen klug mit den eigenen Einschränkungen adaptieren.

Persönliches Exemplar für Thomas Bartholomäus

Viele der genannten Punkte, die bei der Umsetzung von DevOps falsch laufen, dürften Ihnen noch nicht sonderlich klar sein, da konkrete Beispiele fehlen. Diese Punkte werden immer wieder im Laufe des Buches vorkommen und sollten das Ganze dann wesentlich klarer machen.

Reflexion

Sie dürften in diesem Abschnitt sicherlich den einen oder anderen Aspekt entdeckt haben, den Sie entweder selbst bei Ihnen im Unternehmen finden oder schon mal von jemand anderem gehört haben.

Wenn Sie aktiv die DevOps-Transformation an- und weitertreiben wollen, dann kann ich Ihnen nur dazu raten, sich die eine oder andere Fehlannahme, die Sie hören werden, aufzuschreiben. Das hilft insbesondere bei der Kommunikation der gesamten DevOps-Vision, die in Kapitel 11 vertieft wird. Denn dann hätten Sie direkt ein paar Fehlannahmen aus erster Hand, die Sie wiederverwenden und aus dem Weg räumen können.

Eine erste Meinung, die es abzubauen gilt, ist die Arbeit nach dem Motto: »Never touch a running system!« Änderungen sind immer nötig, und sie sollten ohne Angst, aber auch nicht blindlings durchgeführt werden. Eine offene Fehlerkultur ist dafür essenziell, und die gilt es stetig aufzubauen.

Hier helfen Metriken, um Entscheidungen faktenbasiert zu treffen. Davor müssen Metriken natürlich erst einmal ermittelt werden. Sie können sich ja überlegen, welche Informationen Sie gerne hätten, um eine bessere Einsicht in und ein besseres Gefühl für die Anwendung zu bekommen.

2.3 Der DevOps-Lifecycle

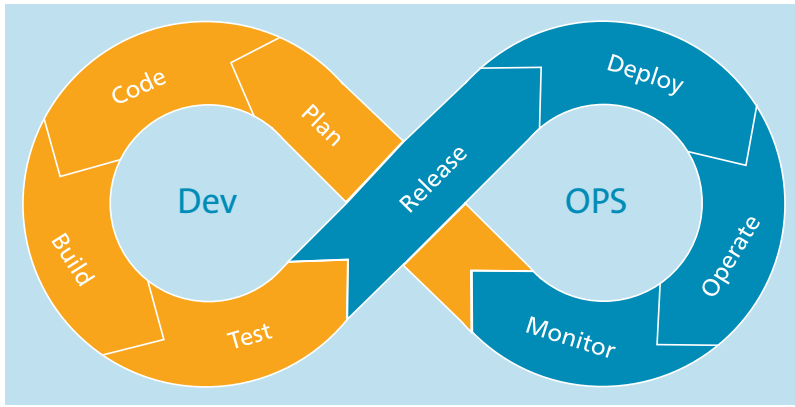

Abbildung 2.10 DevOps ist ein unendlicher Vorgang.

© Rheinwerk Verlag, Bonn 2024

Der *Software-Development-Lifecycle*, der mit SDLC abgekürzt wird, wird wie in Abbildung 2.10 als Unendlichkeitszeichen, also als liegende Acht, dargestellt. Damit wird angedeutet, dass es sich um einen kontinuierlichen Prozess handelt, der nie abgeschlossen ist, solange die Software genutzt wird. Er besteht aus verschiedenen Komponenten, die Hand in Hand gehen. Viele Aspekte überlappen sich und lassen sich nicht klar eingrenzen, was Sie im Laufe des Buches auch erkennen werden.

Wichtig hervorzuheben ist, dass dieser Lebenszyklus möglichst kurz gehalten werden soll, um die DevOps-Prinzipien auch mit all ihren Vorteilen umzusetzen.

Der erste Teil dieses Buches konzentriert sich auf einzelnen Aspekte des DevOps-Lifecycles. Dabei zeige ich anhand der Beispielfirma, die Sie im nächsten Abschnitt kennenlernen, auf, wie für jeden Teilschritt DevOps-Prinzipien eingeführt werden können und welche Vorteile sich daraus ergeben.

Beachten Sie, dass der Lifecycle einen Flow von Informationen und Tätigkeiten darstellt und dass er mit möglichst kurzen Iterationen immer wieder von Neuem beginnt. Das ist der Gegensatz zu einem wasserfallartigen Entwicklungsmodell, in dem jede Stufe stetig abgearbeitet wird und in dem man am Ende »fertig« ist.

- **Plan**

  Projekte beginnen in der Regel mit einer Planung. Wenn diese sehr ausführlich, umfassend und unflexibel ist, dann ähneln solche Ansätze häufig dem Wasserfall-Modell, in dem alles im Vorhinein geplant wird und man versucht, keine Änderungen nachträglich mehr hinzufügen zu müssen.

  Das passt so gar nicht mit der DevOps-Welt zusammen. Konzepte der agilen Software-Entwicklung wie Scrum oder Kanban sind mittlerweile schon sehr weit verbreitet und fester Bestandteil des Alltags, wenn man nach DevOps-Prinzipien arbeitet. Planungsschritte nach der agilen Arbeitsweise sind ein Grundprinzip in der DevOps-Welt. Ein kurzer Einblick in die agile Software-Entwicklung im Rahmen von DevOps erfolgt in Kapitel 4.

- **Code**

  Beim Programmieren des Projekts wird Quellcode geschrieben. Auf den ersten Blick mag man sich vielleicht denken: »Okay, aber das mache ich doch jetzt auch schon, was macht DevOps denn da überhaupt anders?«

  Und genau hier gibt es schon sehr viel zu beachten, was man falsch machen kann. Denn nicht nur bei der Planung der Projekte können viele Fehler passieren, auch beim Programmieren und bei den Prozessen drumherum können und müssen die Grundsteine so gelegt werden, dass der Ablauf gut zu den weiteren Komponenten des DevOps-Lifecycles passt. Es geht um die Kollaboration innerhalb des Entwicklerteams, aber auch darüber hinaus mit anderen Abteilungen.

  Das wird in Kapitel 5 näher betrachtet.

Persönliches Exemplar für Thomas Bartholomäus

- **Build**

  Quellcode ist nur dann nützlich, wenn er auch zu Artefakten gebaut werden kann, die sich problemlos veröffentlichen lassen. Daher geht es in diesem Schritt um die Bedeutung von Build-Tools und -Prozessen. Das Ziel sind reproduzierbare automatische Builds, die ohne manuellen Eingriff aus den Quellen gebaut werden können. Darunter fallen auch das Management von Abhängigkeiten mit dem Build-Tool der Wahl und die Optimierung der Build-Zeit, um die Entwicklungsproduktivität hochzuhalten.

  Die Details hierzu folgen in Kapitel 6.

- **Testing**

  Während in klassischen Firmen häufig noch komplett getrennte Qualitätssicherungsteams zu finden sind, liegt nach den DevOps-Prinzipien die Verantwortung für Unit- und Integrationstests direkt bei den cross-funktionalen Teams.

  Ziel ist es, die Tests im Entwicklungsprozess so weit wie möglich nach vorne zu rücken und so in die Pipeline einzubauen, dass es frühes Feedback gibt, ob sie fehlschlagen. Hier spielen die Automatisierung und die Zusammenarbeit der ehemals reinen Entwicklungs- und Qualitätssicherungsteams eine große Rolle.

  Sie ahnen es schon: Das ist der Schwerpunkt von Kapitel 7.

- **Release**

  Am Ende der reinen Entwicklung steht das fertige Artefakt. Für die Veröffentlichung der Artefakte gibt es viele verschiedene kreative Lösungen, die nicht sonderlich zielführend sein müssen. Sinn und Zweck ist es, ein standardisiertes Paket zu bauen, das auch reproduzierbar ist. Das ist die Grundlage für die nächsten Schritte, um ein geschmeidiges Deployment im Release-Prozess zu ermöglichen.

- **Deploy**

  Wenn ein Software-Paket gebaut wurde, geht es um das Veröffentlichen der zuvor geschriebenen und gebauten Artefakte. Beim Release-Management, das häufig einfach unter dem Sammelbegriff *Deployment* zu finden ist, geht es darum, wie die Anwendung an die Endnutzer ausgerollt wird. Wenn Sie es richtig machen, funktioniert dieser Prozess so gut, dass Sie ohne große Risiken Experimente fahren können und Veröffentlichungen nicht scheuen müssen.

  Da *Release* und *Deploy* eng miteinander gekoppelt sind, werden beide Themen gemeinsam in Kapitel 8 behandelt.

- **Operate**

  Nachdem das Projekt geschrieben, gebaut und deployt wurde, muss es noch betrieben werden. In diesem Stage geht es darum, was zu tun ist, wenn Dienste teilweise oder vollständig ausfallen, und wie mit den Fehlern umgegangen werden soll.

  In Kapitel 9 geht es um den Betrieb der Anwendung.

© Rheinwerk Verlag, Bonn 2024

- **Monitoring**

  Nah verwandt mit dem Betrieb und mit einem fließenden Übergang dazu ist das *Monitoring*, das benötigt wird, um möglichst schnell über Probleme informiert zu werden. Da alle Personen mit den verschiedenen Rollen im selben Team arbeiten, lassen sich hier sehr viele Synergien schaffen, sodass Release und Monitoring quasi gleichzeitig erfolgen können.

  Dazu gehört auch, dass man diverse Tests und Experimente auf dem Produktivumgebung mit echten Nutzern laufen lassen kann, um aus diesen Erkenntnissen zu lernen, was dann den nächsten Entwicklungszyklus beeinflussen kann.

  In Kapitel 10 beschreibe ich die Kernelemente des Monitorings.

- **Security**

  Der letzte Aspekt dieser Liste, der aber nicht der letzte Aspekt im DevOps-Lifecycle ist, ist die *Security*. Wie man anhand von Abbildung 2.10 erkennen kann, muss sich das Thema Sicherheit durch den ganzen Lebenszyklus ziehen, da Security-Standards und -Prinzipien überall eingebunden werden sollten und nicht (wie so häufig) zum Schluss ergänzt werden können. Denn warum sollte man das Ops-Team einbeziehen, aber das Security-Team außen vor lassen?

  Wichtig ist, dass Sicherheits- und Datenschutzfragen von Beginn an in allen Teilen des Entwicklungs- und Deployment-Prozesses mitgedacht werden. *Sicherheit* ist kein Aspekt, der an einer bestimmten Stelle ergänzt werden kann, sondern muss gelebter Teil des gesamten Systems sein.

  Das ist Teil von Kapitel 11, bevor es in Kapitel 12 darum geht, wie Sie die Ideen der DevOps-Kultur in Ihrer eigenen Umgebung umsetzen können. Helfen können dabei die DevOps-Plattformen, die ich in Kapitel 13 kurz vorstelle.

Reflexion

Die einzelnen Teile des Software-Development-Lifecycles sind wichtig. Dabei darf man nicht vergessen, dass diese untrennbar zusammengehören. Sie mögen sich vielleicht denken: »Ja, gut, Code, Build und Test mache ich doch heute schon, dann habe ich ja bereits die Hälfte!« Das ist gut, richtig und wichtig, aber auch nur die halbe Wahrheit.

Eine Fokussierung auf Code, Build und Test ist häufig zu finden. In der Praxis zeigt sich, dass der Ops-Teil oft vernachlässigt oder gar in andere Teams ausgelagert wird. Hohes Augenmerk muss zudem der Security gelten.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 3
Die Beispielfirma

An dieser Stelle ist es nun Zeit, zu der bereits angesprochenen Firma zu kommen, die uns über das ganze Buch hinweg als Beispiel dienen wird. Zunächst sehen wir uns an, wie dort die Arbeit ohne DevOps aussieht. In den folgenden Kapiteln zeige ich dann, wie die einzelnen Aspekte aus jeder Stage des DevOps-Lifecycles in dieser Firma angewandt werden und welche Vorteile das mit sich bringt.

Anhand dieses praktischen und nachvollziehbaren Beispiels wird hoffentlich klarer, wie sich die verschiedenen DevOps-Prinzipien auf die Produktivität, die Zusammenarbeit und die Wartung auswirken.

Am einfachsten lassen sich DevOps-Prinzipien sicherlich einführen, wenn »auf der grünen Wiese« gestartet wird. So muss gar keine große Umstellung oder Migration durchgeführt werden, sondern alle Aspekte können sofort implementiert werden.

Die wenigsten werden allerdings mit einem weißen Blatt Papier starten können, sondern müssen mit altem Code, gewachsenen Prozessen und antiquierten Tools einer bestehenden Firma arbeiten. Dort ist nicht nur der Code ordentlich abgehangen, sondern auch die Denkweise der Menschen entspricht eher dem Motto: »Das haben wir doch schon immer so gemacht und es hat funktioniert!«

Und genau das ist die größte Herausforderung! Es gibt und gab in jedem Unternehmen gute Gründe, warum so gearbeitet wurde, wie gearbeitet wird. Auf einem anderen Blatt steht jedoch, ob auch zukünftig so weitergearbeitet werden kann. Organisationen sind schwergängig, was nicht nur den Menschen geschuldet ist, sondern auch dem Code, der häufig mit hohen technischen Schulden daherkommt.

Die *Schick Gekleidet GmbH* soll nur als Beispiel dienen. Sie werden in ihr aber sicherlich einige Aspekte auch von Ihrer Organisation entdecken und einige Probleme wiedererkennen. Die Beispiele und Lösungsansätze, die in diesem Buch erwähnt werden, können und sollen aber nicht 1:1 kopiert werden.

Denn wie bereits angesprochen, ist das DevOps-Modell ja gerade keine Musterlösung, die Sie in wenigen Schritten über Ihre Firma stülpen können. Es geht stattdessen darum, Ihnen einige Ideen zu präsentieren, übliche Probleme darzustellen und Antworten zu diskutieren. Ihre Aufgabe ist es dann, diese Ansätze an die Realität Ihrer Umgebungen anzupassen. Wichtig ist immer, dass Sie das große Ganze betrachten oder – wie man es gemäß den Drei Wegen ausdrücken würde – immer an das ganze System denken.

Persönliches Exemplar für Thomas Bartholomäus

3.1 schick-gekleidet.de

Unsere *Schick Gekleidet GmbH* ist eine Firma, die Ende der 90er-Jahre entstanden ist und somit über 20 Jahre alt ist. Sie verdient ihr Geld damit, Kleidungsstücke sowohl online über den eigenen Webshop auf *schick-gekleidet.de* als auch offline in Ladenlokalen zu verkaufen. Der dazugehörige Webshop ist eine komplette Eigenentwicklung, und es existiert noch einiges an Sourcecode, der aus den Anfangstagen stammt. Der Quellcode ist an einigen Stellen über 20 Jahre alt.

Wenig überraschend ist also, dass der Code historisch gewachsen ist und somit strukturelle »Untiefen« mit sich bringt, die man bei einem kompletten Rewrite des Shops nach dem heutigen Stand der Technik sicherlich vermeiden würde. In dieser ganzen Zeit hat sich ja die Art und Weise der Software-Entwicklung, aber haben sich auch die Technologien, die eingesetzt werden, immer wieder verändert.

Grundsätzlich funktioniert der Code aber, und der Online-Shop erwirtschaftet gute, jedoch keine riesigen Gewinne. Bisher wurden allerdings noch genügend Umsätze eingefahren, sodass die Geschäftsleitung so weit zufrieden war.

Mit der Zeit schrumpften allerdings die Umsätze und somit die Gewinne immer weiter, sodass ein stärkerer Fokus auf den Online-Shop hermusste. Ladenlokale zu betreiben, ist in der aktuellen Wirtschaftssituation kein Kinderspiel, aber auch Online-Shops sind großer Konkurrenz ausgesetzt.

Obwohl der Online-Shop so weit grundsätzlich funktioniert und immer wieder Änderungen implementiert und ausgerollt werden, gibt es gelegentlich einige größere Probleme. Das wird schnell klar, wenn man die Mitarbeitenden fragt, was für Probleme sie gerade in der Entwicklung sehen. In solchen Gesprächen kommen viele Ärgernisse und Einschränkungen zum Vorschein, die zu verbreiteter Unzufriedenheit führen.

3.2 Das Entwicklungsmodell

Das erste Problem, das schon viele, aber bisher nicht alle in der Firma erkannt haben, besteht darin, dass vor einigen Jahren zwar eine agile Transformation stattgefunden hat, diese allerdings trotzdem sehr stark an das Wasserfall-Modell angelehnt ist (siehe Abbildung 3.1).

In früheren Zeiten wurde viel mit dem Wasserfall-Modell gearbeitet. Das Hauptproblem für das Entwicklungsteam ist dabei, dass schon sehr früh alles detailliert geplant und dann designt wird. Dabei vergehen meistens einige Wochen oder gar Monate. Erst dann geht die Arbeit für das Entwicklungsteam los, das nun Änderungen über mehrere Monate implementieren muss. Es gibt keine Abstimmung oder gar eine Feedback-Schleife zwischen der Umsetzung und dem Design-Prozess.

© Rheinwerk Verlag, Bonn 2024

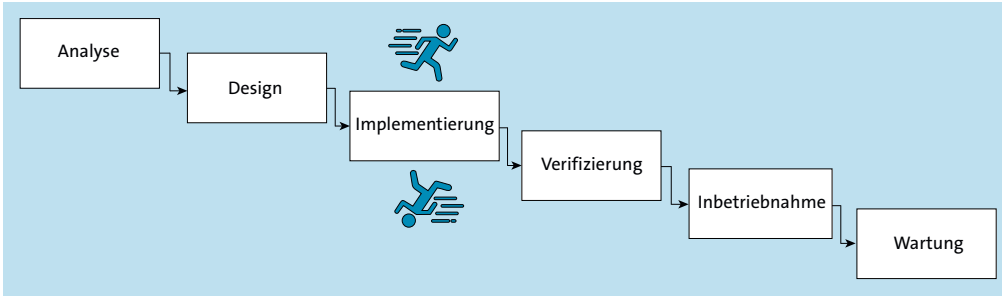

Abbildung 3.1 Das agilisierte Wasserfall-Modell mit seinen Stufen – nur die Implementierung folgt den agilen Methoden und nutzt Sprints.

Viele Ideen stellten sich dabei als unrealistisch heraus und konnten zeitlich gar nicht umgesetzt werden; bei anderen Anforderungen musste schon etliche Monate im Voraus abgestimmt, geplant und dann entwickelt werden. Wenn sich in der Zwischenzeit etwas an den Vorgaben oder den Rahmenbedingungen änderte, war die Arbeit umsonst.

Offiziell arbeitet *schick-gekleidet.de* zwar nicht nach dem Wasserfall-Modell, doch wurde im Wesentlichen nur die Implementierungsphase in die agile Arbeitsweise überführt. Der Rest erfolgte weiterhin streng getrennt voneinander, sodass sich die Vorteile der agilen Software-Entwicklung nicht entfalten konnten.

Eigentlich sollte am Ende des Sprints die Software so weit sein, dass man diese auch ausrollen kann. In der Realität sieht es allerdings eher so aus, dass nach einigen Sprints die Implementierungsarbeiten abgeschlossen sind, die Arbeit aber anschließend an das Qualitätssicherungsteam übergeht, das hauptsächlich manuell Tests durchführt. Sobald die Abnahme stattgefunden hat, findet ein Deployment statt. Erst dann sind die Arbeiten abgeschlossen und es geht in die Wartung des Systems über.

Zwischen der Anforderungsphase und dem Live-Betrieb der umgesetzten Änderungen liegen ungefähr sechs Monate. Jede Stufe des Entwicklungsmodells liegt in der Verantwortung eines eigenen Teams, und die Teams kommunizieren nur wenig miteinander. Die Mitglieder der jeweiligen Teams wissen weder so genau, was die nachfolgenden Teams machen, noch wissen sie, welche Auswirkungen ihre Arbeitsschritte auf die Tätigkeiten der anderen Teams haben.

3.3 Das Business-Team – Anforderungsanalyse

Das Business-Team von *schick-gekleidet.de* entscheidet zu Beginn der Entwicklungsphase, was in den folgenden Monaten entwickelt werden soll. Die Mitglieder dieses Teams führen daher die Anforderungsanalyse durch und stellen dann den Plan auf, welche neuen Features benötigt werden und wo Optimierungsbedarf besteht.

Persönliches Exemplar für Thomas Bartholomäus

Sie sind auch dafür zuständig, zu entscheiden, was eventuell entfernt werden kann. Sie kooperieren eng mit dem Marketing-Team, da einige Funktionen zusammen mit diversen Kampagnen gestartet werden sollen.

Das Kernziel des Businessteams ist es, die Anforderungen zu definieren, die helfen sollen, mehr Kunden in den Shop zu leiten und somit den Verkauf dort anzukurbeln.

Die Anforderungsanalyse geschieht allerdings im Wesentlichen ohne das Wissen um die technischen Zusammenhänge und den Stand der Codebasis. Daher können sie den Aufwand und die Wechselwirkungen von geplanten Änderungen kaum absehen, was noch dadurch verschlimmert wird, dass sie nur selten mit dem Architekturteam und den Entwicklungsteams direkt sprechen.

Ebenfalls gab es auch kaum Metriken, die detailliert und flexibel die Auswirkungen von Änderungen messen. Es werden lediglich die Verkaufszahlen einmal nach Abschluss des Quartals generiert, sodass erst spät ersichtlich wird, wie erfolgreich die vergangenen Monate waren.

3.4 Das Architekturteam – Design der Anwendung

Wenn das Businessteam eine Wunschliste mit den zu entwickelnden Features entworfen hat, wird der Plan dem Team für die Software-Architektur übergeben, das das Design der Anwendung entwirft. Die Architekten überlegen im Anschluss, wie und was alles angepasst werden muss, um den neuen Anforderungen gerecht zu werden.

Obwohl hier eine enge Zusammenarbeit mit den Entwicklern unabdingbar wäre, da deren Erfahrungswerte entscheidend für die erfolgreiche Umsetzung sind, passiert genau dies nicht: Die Programmierer sind nämlich noch mit den Feinarbeiten des letzten Auftrags beschäftigt, der nun dringend in die Live-Umgebung deployt werden muss. Zeit, um Input zu den anstehenden Änderungen zu liefern, haben sie also nicht.

3.5 Die Entwicklungsteams

Nachdem das Architekturteam die Architektur für das neue Release (vermeintlich) fertiggestellt hat, reicht es die Aufgabe an die verschiedenen Software-Entwicklungsteams weiter. Diese werden in den folgenden Monaten die gewünschten Features entwickeln und Fehler korrigieren. Es gibt mehrere Teams, die an verschiedenen Teilen der Software arbeiten. So gibt es ein Team für den Großteil des Shops, eines für das Billing-Backend, damit die Abrechnungen ordentlich funktionieren, sowie Teams für die Apps der verschiedenen mobilen Betriebssysteme.

Zwischen diesen verschiedenen Teams gibt es keine direkte Zusammenarbeit. Die Teams arbeiten in ihrer eigenen Kammer für sich und reden nur bei Bedarf mit

© Rheinwerk Verlag, Bonn 2024

Leuten aus den anderen Teams. Die Teams erhalten also immer nur die vermeintlich fertige Arbeit von dem jeweiligen Vorgängerteam.

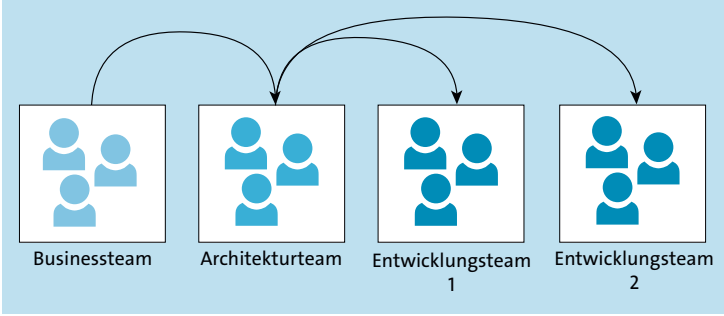

Abbildung 3.2 Die Informationen werden über die große Mauer zum nächsten Team weitergeworfen, ohne direkt zu kommunizieren.

3.5.1 Der Entwicklungsprozess

Bevor mit den eigentlichen Implementierungsarbeiten begonnen wird, schnappt sich die jeweilige Person erst einmal folgendes Ticket: »Für das Weihnachtsgeschäft sollen pünktlich zum ersten Advent die ersten Angebote stehen, und zu jedem weiteren Advent soll es weitere Angebote geben, die auf der Startseite dargestellt werden.«

Da die ganze Entwicklungsarbeit im ersten halben Jahr abgeschlossen werden soll und es zum Start der Arbeit an dem Ticket noch Januar ist, hat der Entwickler bzw. die Entwicklerin also noch reichlich Zeit. Das Ticket enthält noch viele weitere Informationen, etwa wann welche Angebote dargestellt werden sollen und wann diese jeweils wieder verschwinden sollen.

Eine Entwicklerin startet also die Arbeit an dem Ticket für die Angebote zum ersten Advent. Der Sourcecode, der von ihrem Team entwickelt wird, wird in einem Subversion-Repository verwaltet. Die Entwicklerin arbeitet, genauso wie ihre anderen Kollegen, lokal mit einer Arbeitskopie, so wie es bei Subversion nun einmal vorgegeben ist.

Das Feature für das Angebot wird über den Zeitraum von mehreren Wochen implementiert. In dieser Zeit erfolgt kein weiterer Check-out des Repositorys; die Änderungen, die von den anderen Mitgliedern des Entwicklungsteams geschrieben werden, werden also nicht noch mal heruntergeladen.

Zwischenzeitlich kommen immer wieder Fragen auf, wie etwas genau gedacht war, da die Dokumentation der Anforderungsanalyse und der Architekturphase weitere Fragen aufwirft. Diese Frage geht dann also an das Fachpersonal zurück, mit der Bitte um Konkretisierung. Nach einer gewissen Wartezeit erhalten die Entwickler leider keine konkreten Antworten, denn so ganz genau wissen die Entscheider auch nicht mehr, was sie sich dabei gedacht hatten. Deren Planungsphase ist schon lange vorüber, somit fehlt der Kontext, in dem diese Features diskutiert wurden. Also muss hier abermals nachgearbeitet werden, bevor die Implementierung weitergehen kann.

Persönliches Exemplar für Thomas Bartholomäus

Während das Weihnachtsangebots-Feature trotzdem weiter über mehrere Wochen implementiert wird, arbeiten andere Kollegen fleißig an anderen Teilen des Online-Shops. Während unsere Entwicklerin also das Advents-Feature implementiert, arbeitet jemand anderes im Team an einem größeren Refactoring des Codes, mit dem einige Altlasten entfernt werden können. Auch diese Arbeiten erstrecken sich über einen längeren Zeitraum. Sie werden aber vor der Fertigstellung des Weihnachtsangebots-Features fertig implementiert, integriert und in das Repository eingecheckt.

3.5.2 Integrationen mit Hindernissen

Alle im Team haben nun ein Problem, wenn sie ihren Code integrieren möchten. Dadurch, dass alle einen Branch im Repository verwenden, holen sie sich die Änderungen von den anderen Leuten grundsätzlich nicht während der Implementierungsarbeit ab. Ein Branch ist im Sourcecode-Management eine Entwicklungslinie, in der der Quellcode entsprechend enthalten ist.

Sie testen dabei die Features manuell bei der Entwicklung und können ihren Teil der Anwendung relativ problemlos durchchecken, wenn auch manuell. Am Ende ihrer eigenen Entwicklungstätigkeit müssen sie jetzt aber zwangsläufig die Änderungen herunterladen, die in der Zwischenzeit aufgelaufen sind. Dabei kommt es leider fast immer zu Konflikten im Code.

Die Entwicklerin, die das Weihnachtsangebots-Feature implementiert hat, muss jetzt mit zahlreichen Konflikten kämpfen, da das Refactoring auch den Code betrifft, den sie angefasst hat. Sie muss also jetzt die Änderungen abermals grundlegend anpassen, was noch einmal einige Tage in Anspruch nimmt.

Das ganze »Spielchen« muss sie dann wiederholen, schließlich sind noch mal Änderungen eingeflossen, die zum Glück nicht ganz so groß waren. Das ging aber auch nur, weil sie intern im Team abgesprochen hat, dass sie ihre Arbeit integrieren muss. Währenddessen haben ihre übrigen Kollegen also etwas gewartet, damit sie nicht immer wieder das gleiche Schicksal trifft.

Nachdem dann endlich die Integration erfolgreich war, stößt das Team direkt auf das nächste Problem: Einzelne Entwickler melden, dass der Webshop bei ihnen nicht mehr erfolgreich gebaut werden kann. Die Arbeit des übrigen Teams steht jetzt still, denn wenn nicht mehr sicher ist, dass das Projekt gebaut werden kann, kann auch nicht mehr sinnvoll weiterentwickelt werden.

Zum Glück war es kein großes Problem: Es wurde eine neue Abhängigkeit eingeführt, die nicht dokumentiert worden war. Nachdem sich alle Teammitglieder diese Abhängigkeit heruntergeladen haben, ist das Problem auch gelöst. Das muss nur noch an alle kommuniziert werden, was wieder ein wenig Zeit in Anspruch nimmt.

Einige Monate später soll das Deployment auf die Produktionsumgebung erfolgen. Das Deployment wird grundsätzlich einmal im Quartal durchgeführt, jeweils zum

© Rheinwerk Verlag, Bonn 2024

Quartalswechsel. Bevor wir uns dieses Thema anschauen, müssen wir noch einen Blick darauf werfen, was passiert, wenn das Entwicklungsteam und seine Subteams die Arbeit an den neuen Implementierungen abgeschlossen haben.

3.6 Das Qualitätssicherungsteam

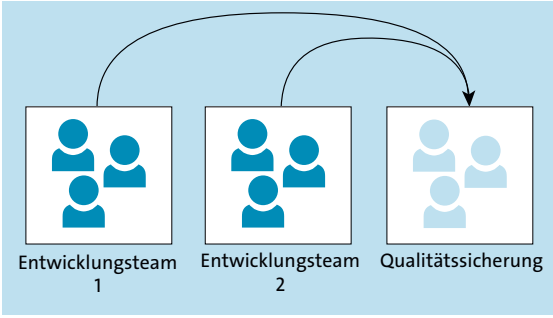

Abbildung 3.3 Die Projekte landen am Ende der Entwicklungsphase bei der Qualitätssicherung.

Bevor das Deployment durchgeführt wird, müssen die neuesten Entwicklungen durch das Qualitätssicherungsteam (*Quality Assurance Team*, kurz: QA-Team) geprüft werden. Auch bei diesem Schritt kommt es immer wieder zu Konflikten, die diesmal nicht nur technischer, sondern auch menschlicher Natur sind.

Das Ziel und die Motivation des Entwicklungsteams sind nämlich grundsätzlich, neue Features zu entwickeln und notwendige Fehlerkorrekturen durchzuführen. Die Devs interessiert es nicht sonderlich, was das QA-Team tut. Dieses hat allerdings das Ziel, möglichst viele Fehler zu finden, welche die Entwicklungsteams wieder korrigieren müssen.

Die Ziele der Teams sind also konträr: Je mehr Fehler das QA-Team findet, desto mehr Arbeit bedeutet das für das Entwicklungsteam. Entsprechend schlecht ist das Entwicklungsteam auf das QA-Team zu sprechen. Was das QA-Team nämlich genau tut, um die Fehler zu finden, ist dem Entwicklungsteam nicht bekannt und eigentlich auch egal: Sie sollen doch Probleme korrigieren und Features entwickeln und nicht Fehler finden.

Das QA-Team führt verschiedene Arten von Tests durch: Teilweise testet es sehr viel manuell, teilweise automatisiert es auch einige Teile. Die automatisierten Skripte liegen häufig auf den Rechnern von einzelnen Teammitgliedern verteilt, ohne dass sie in das Sourcecode-Repository neben dem eigentlichen Webshop-Projekt abgelegt werden. Die einzelnen QA-Team-Mitglieder finden also Fehler im Webshop und melden diese dem Entwicklungsteam und sind dann zunächst aus dem Schneider.

Persönliches Exemplar für Thomas Bartholomäus

Hier kommen wieder mehrere Probleme zutage: Zunächst einmal sind die Teams komplett getrennt voneinander. Sowohl organisatorisch als auch fachlich. Es findet keine direkte Kollaboration statt, da die Fehler immer nur jeweils »über den Zaun geworfen« werden, und zwar von beiden Teams zum jeweils anderen. Das führt letztlich dazu, dass grundlegende Probleme, die bei einer engeren Kollaboration eher angesprochen und behoben werden könnten, gar nicht erst erwähnt werden.

Ein weiteres Problem ist, dass die QA-Phase, in der hauptsächlich das QA-Team arbeitet, komplett nach der Implementierungsphase stattfindet. Die einzelnen Personen aus dem Entwicklungsteam bekommen also erst einige Monate nach der Implementierung für das jeweilige Feature Feedback.

So wird viel hin- und herkorrigiert und mit Problemen gehadert, die schon viel früher aufgetaucht sind und gelöst worden wären, wenn der Software-Entwicklungszyklus schneller gewesen wäre. Weiterhin liegt der Quellcode für die Tests in einem komplett getrennten Repository, was dazu führt, dass das QA-Team alle drei Monate – also jedes Mal, wenn es wieder an dem Projekt aktiv wird – die Tests korrigieren und den Änderungen der Anwendung entsprechend anpassen muss. Die Tests laufen zudem auf einem bestimmten Rechner, auf den nur bestimmte Leute Zugriff haben. Auf diesem Rechner wurden die ganzen Scripts und das Tooling drumherum einmal manuell installiert, sie sind aber kaum dokumentiert worden.

3.7 Das Betriebsteam – Das Ops aus DevOps

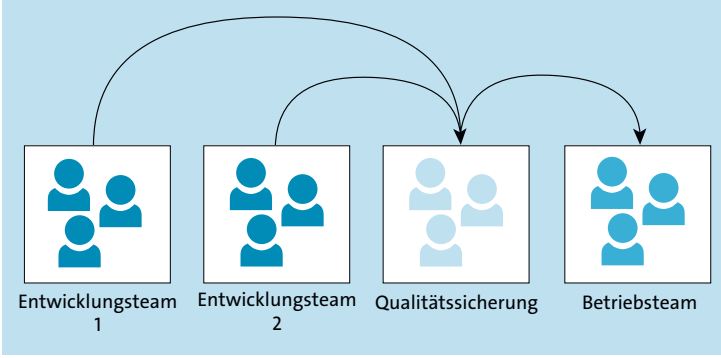

Abbildung 3.4 Nach der Freigabe durch die Qualitätssicherung landen die Artefakte beim Betriebsteam.

Neben dem Entwicklungsteam und dem QA-Team existiert auch noch das Betriebsteam, das die Anwendung in Betrieb nimmt und betreut. Hier geht es also um das *Ops* aus Dev*Ops*. Das Hauptaugenmerk des Teams liegt darauf, dass die Anwendung immer läuft und für die Kunden verfügbar ist. Denn ohne laufenden Online-Shop gibt es keine Einnahmen für die Firma.

© Rheinwerk Verlag, Bonn 2024

Admins und der Betrieb

Bisher schrieb ich immer vom »Betriebsteam«. Die einen fragen sich vielleicht schon, wo die bekannte Systemadministration hin ist. Das ist eine berechtigte Frage! Viele verwenden die Begriffe *Betriebsteam* und *Adminteam* äquivalent. Das ist allerdings nicht so ganz richtig, auch wenn sich viele Aspekte aus den Rollen überlappen.

Wenn ich hier über das Betriebsteam schreibe, dann ist es das Team, das die Anwendung betreibt. »Betriebsteam« kommt daher auch vom englischen Wort »Operations«, was nun mal nicht »Administration« ist.

Administratoren planen und verwalten die Infrastruktur, auf der die Anwendungen laufen. Dazu gehören unter anderem Server, Netzwerk und Rechenzentren. Häufig gehören zu den Admin-Aufgaben auch die Installation des Betriebssystems, die Verwaltung der Virtualisierungslösung und das Einspielen von Aktualisierungen von verschiedenen Systemen. Es ist also auch eine strategische Rolle und Aufgabe. Je nach Firma sind die Aufgabengebiete fließend. Mal gehört noch Security dazu, mal gibt es eigenständige Netzwerkadministratoren.

Ich fokussiere mich hier hauptsächlich auf den Begriff des Betriebsteams, denn dieses betreibt die Anwendung, die vom Entwicklungsteam entwickelt wird. Diejenigen, die die Hardware verwalten, nenne ich hier das »Infrastrukturteam«. Des Weiteren gibt es auch noch das »Plattformteam«, das das Bindeglied zwischen Infrastrukturteam und Betriebsteam ist, damit eine Anwendung darauf laufen kann.

Je kleiner die Firma ist, desto eher überlappen sich diese Aufgaben. Daher folge ich diesen genaueren Begriffen. Die Rolle des Systemadministrators ist also etwas größer. Also nicht wundern: Es wird sicherlich einige Systemadministratoren geben, die die Anwendungen betreiben, aber sich auch um das darunterliegende System kümmern.

Außer für den reinen Betrieb des Online-Shops ist das Betriebsteam auch noch dafür zuständig, neue Deployments einzuspielen. Das Betriebsteam kommt also genau dann ins Spiel, wenn jedes Halbjahr die Entwicklungs- und die Qualitätssicherungsphase erfolgreich durchlaufen wurde.

Das Betriebsteam betreut die Anwendungen von verschiedenen Entwicklungsteams. Je mehr Änderungen vorgenommen werden, desto mehr Arbeit bedeutet das für das Betriebsteam. Das Entwicklungsteam macht direkt für das Betriebsteam zwei Dinge:

- Es baut die Anwendung als Paket, damit die Entwickler es so dem Betriebsteam zur Verfügung stellen können.
- Zudem aktualisieren die Entwickler jedes Mal die Dokumentation, damit das Betriebsteam bestimmte applikationsabhängige Gegebenheiten anpassen kann.

Aber wie auch beim Entwicklungs- und QA-Team gilt, dass auch zwischen Entwicklungsteam und Betriebsteam hohe Mauern stehen.

Persönliches Exemplar für Thomas Bartholomäus

3.7.1 Manuelles Bauen des Projektes

Nachdem die Qualitätssicherung also das Release freigegeben hat, geht nun das Entwicklungsteam wieder an die Arbeit und baut das Paket. Hierfür wird genau ein Server verwendet, auf dem das Projekt bereits vorliegt und alle lokal benötigten Abhängigkeiten vorinstalliert sind. Dieser Server wird allerdings nur beim Release verwendet, also in der Regel nur alle sechs Monate.

Bevor das Release gebaut wird, muss viel manuell gemacht werden: Es muss noch mal überprüft werden, welche Abhängigkeiten neu hinzugekommen sind, und diese müssen dann nachinstalliert werden. Anschließend werden alle Komponenten des Webshops einzeln gebaut. Die jeweiligen Pakete, die aus den einzelnen Komponenten gebaut werden, müssen für das finale Bauen des Projekts einmal von Hand zusammenkopiert werden.

Häufiges Problem hierbei: Beim Bau werden manchmal einzelne Komponenten bewusst weggelassen, da nicht bei jedem Release Änderungen in jede Komponente eingeflossen sind. Das Betriebsteam benötigt allerdings alle Komponenten einmal neu für jedes Deployment.

3.7.2 Deployment mit Hindernissen

Das Betriebsteam erhält nach einer gewissen Wartezeit das fertige Paket. Zum Deployment-Termin zu Beginn des zweiten Quartals soll das Deployment eingespielt werden. In ihm ist auch das Feature für das Adventsangebot enthalten. Als das Deployment im definierten Wartungsfenster eingespielt wird, kommen allerdings einige Fehler ans Tageslicht.

So schlägt das Deployment zunächst fehl, weil etwas nicht gefunden werden konnte. Nach ein wenig Nachprüfen stellt sich heraus: Die neue Abhängigkeit, die eingeführt wurde, war nicht für das Betriebsteam dokumentiert worden. Da das Deployment nachts eingespielt wird, um bei möglichen Ausfällen möglichst geringe Umsatzeinbußen zu haben, ist der Shop zunächst einmal offline.

Da keine Person des Entwicklungsteams verfügbar ist, tappt das Betriebsteam erst mal eine Weile im Dunkeln, bis es das Problem findet und korrigiert – und zum Glück für *schick-gekleidet.de* lässt sich dieses Problem korrigieren.

3.7.3 Das Monitoring schlägt Alarm

Einige Stunden nach dem Deployment meldet das Monitoring, dass die Anwendung offline ist. Nach ein wenig Recherche stellt sich heraus, dass die neue Version des Webshops das Dateisystem vollschreibt und den Webshop damit zum Absturz bringt. Da das Betriebsteam kaum Kenntnisse von der Anwendung hat, muss es erst

© Rheinwerk Verlag, Bonn 2024

mal schauen, wo das Problem liegen könnte und wie man damit umgeht, schließlich ist es noch tief in der Nacht und kein Entwickler ist online.

Glücklicherweise stellt sich heraus, dass es eine einfache Lösung gibt: Wir löschen einfach ständig die Logdatei, die das Dateisystem vollschreibt. Am nächsten Tag geht die Problembeschreibung an die Entwickler und diese schauen, wie man das Problem beheben kann. Da in der Zwischenzeit die Entwicklung jedoch weiter vorangeschritten ist, braucht es ein neues Deployment, was dummerweise auch neue Features freischalten würde, die eigentlich noch gar nicht reif für das Publikum sind.

Zudem ist der Prozess mit seinen ganzen Übergaben und Stolpersteinen so umständlich, dass niemand Lust auf ein weiteres Deployment hat. Man weiß ja, dass dabei auch wieder neue Probleme mit neuen Fehlern warten, was alle Seiten vermeiden wollen. Stattdessen wird entschieden, dass der temporäre Quickfix bis zum nächsten Release drinbleiben soll.

Anstatt ein kleines Problem also zügig zu korrigieren und die Änderung auszurollen, wird hier wieder aufgeschoben. Man baut eine weitere Abhängigkeit ein (nämlich das Skript des Betriebsteams, das die Logdatei löscht), anstatt Fehler zu korrigieren und das System einfacher zu machen. Es ist nur eine Frage der Zeit, bis jemand die Logdatei für eine Systemanalyse vermisst und niemand mehr so genau weiß, warum diese Informationen nicht mehr gesammelt werden ...

3.8 Das Infrastrukturteam

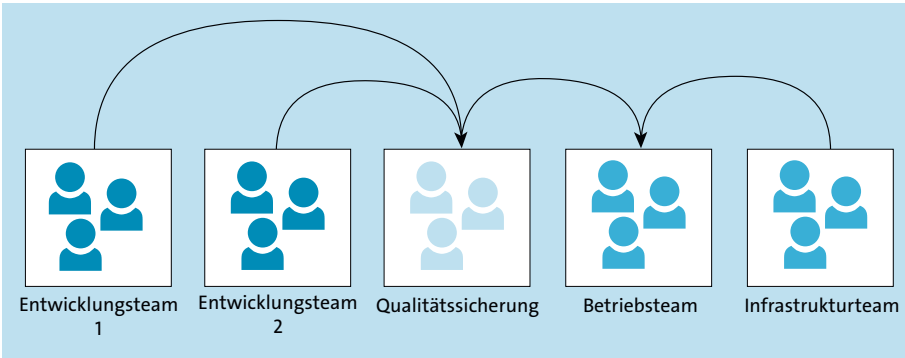

Abbildung 3.5 Das Infrastrukturteam stellt die Hardware für das Betriebsteam bereit.

Neben den vorgestellten Entwicklungs-, Qualitätssicherungs- und Betriebssteams gibt es noch zwei weitere Teams in der Firma: das Security-Team sowie das Infrastrukturteam. Die Aufgabe des Infrastrukturteams ist zwar schnell erklärt, aber ebenso wichtig: Es stellt die Infrastruktur für das Betriebsteam bereit. Konkret geht

Persönliches Exemplar für Thomas Bartholomäus

es also um den Betrieb des Rechenzentrums. Darunter fallen Anschaffung, Bereitstellung und Reparatur von Server-Hardware.

Das Infrastrukturteam sorgt also für die solide Basis, auf der das Basis-Betriebssystem installiert wird, damit das Betriebsteam dann damit arbeiten kann. Auch hier existieren Mauern zwischen dem Betriebsteam und dem Infrastrukturteam, da häufig einige Probleme und Herausforderungen abgesprochen werden müssen: Ein Beispiel: Wenn Hardware ausgetauscht werden muss, darf kein Deployment stattfinden – was dann das Entwicklungsteam natürlich auch wissen muss.

3.9 Das Security-Team

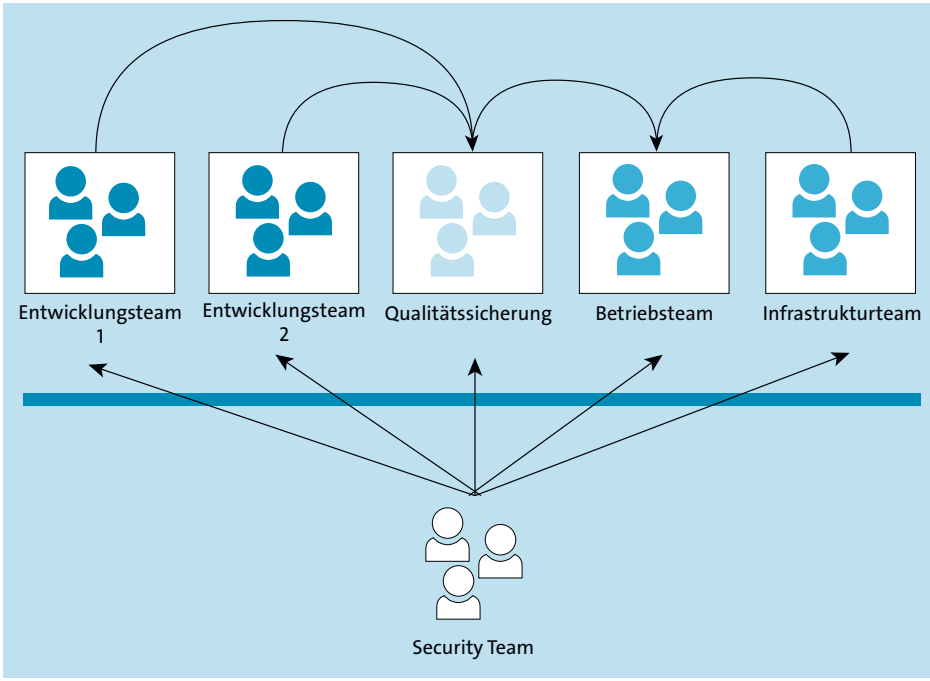

Abbildung 3.6 Das Security-Team arbeitet hinter seiner eigenen Wand mit allen Teams zusammen.

Schließlich existiert noch das Security-Team. Das Aufgabe dieses Teams ist, dass die Anwendung sicher ist und möglichst keine bekannten Sicherheitslücken enthält. Das bedeutet, dass sich das Team sowohl um die Sicherheit des Webshops als auch um die Infrastruktur kümmern muss, auf der er läuft.

Auf das Security-Team ist keines der anderen Teams gut zu sprechen, denn häufig blockieren Sicherheitsbedenken Prozesse und die Entwicklung neuer Features. So

© Rheinwerk Verlag, Bonn 2024

bemängelt das Security-Team etwa veraltete Abhängigkeiten, die mit großen Sicherheitslücken verbunden sind, oder gravierende Lücken in der Anwendung, die SQL-Injections ermöglichen, wodurch Angreifer potenziell Kundendaten abgreifen können. Diese Rügen bedeuten für die zuständigen Teams aber einen Heidenaufwand und lassen deren Todo-Listen wachsen.

Das Security-Team kämpft ebenfalls mit einigen Herausforderungen: Jede Abhängigkeit jeder Anwendung zu prüfen, ist ein Ding der Unmöglichkeit, vor allem dann, wenn keine oder nur wenig Automatisierung eingesetzt wird. Kaum anders sieht es mit der Code-Qualität und möglichen Sicherheitsproblemen aus. Selbst etwas Simples wie SQL-Injections kann man nicht restlos ausschließen, wenn man jede Ecke des Webshops eigentlich per Hand prüfen müsste.

Stattdessen schaut das Security-Team häufig nur stichprobenartig auf die Anwendung. Es gibt zwar einige Tools, mit denen diverse Analysen gefahren werden, aber diese liegen nicht in den richtigen Händen. Sie werden ausschließlich vom Security-Team selbst genutzt, das die Informationen dann an das Betriebsteam und das Entwicklungsteam weitergibt. Oder im Klartext: Das Security-Team schreibt lange Listen mit entdeckten Problemen, ohne den genauen Kontext überhaupt zu kennen. Es verteilt Aufgaben an die anderen Teams, meckert an Lösungen herum und kann mit dem Totschlagargument »Sicherheit« ein Veto gegen fast alle Entscheidungen einlegen, womit es die Arbeit von vielen Teams ausbremsen kann.

Aus Security-Sicht gibt es viele Punkte, um die Handbremse an vielen Stellen zu ziehen, etwa kurz vor oder auch nach dem Deployment, wenn gravierende Sicherheitslücken gefunden und bekannt werden. Dadurch wird aber die tägliche Arbeit von mehreren Teams ständig gestört.

3.10 Fazit

Sie haben nun einen Überblick über die Arbeitsweisen, den Aufbau der Teams (siehe Abbildung 3.7) und die Probleme erhalten und haben nun eine Idee, wie es in dieser Firma so läuft. Für die eine oder andere Person mag diese Beschreibung ein wenig übertrieben sein. Tatsächlich ist es allerdings gar nicht so unwahrscheinlich, dass solche Probleme immer wieder auftreten, da es an der Kollaboration zwischen den Teams durch die organisatorische Struktur und die Kultur im Allgemeinen mangelt.

In die Beschreibung der Firma hinter *schick-gekleidet.de* sind sehr viele Erfahrungen eingeflossen, die ich im Laufe meiner Karriere in verschiedenen Unternehmen gemacht beziehungsweise in vielen Gesprächen mitbekommen habe.

Dieses Kapitel hat also schon einige gängige Probleme angerissen. Ab jetzt geht es also langsam los: Schauen wir uns die einzelnen Aspekte genauer an, um herauszufinden, wo und wie DevOps helfen kann.

Persönliches Exemplar für Thomas Bartholomäus

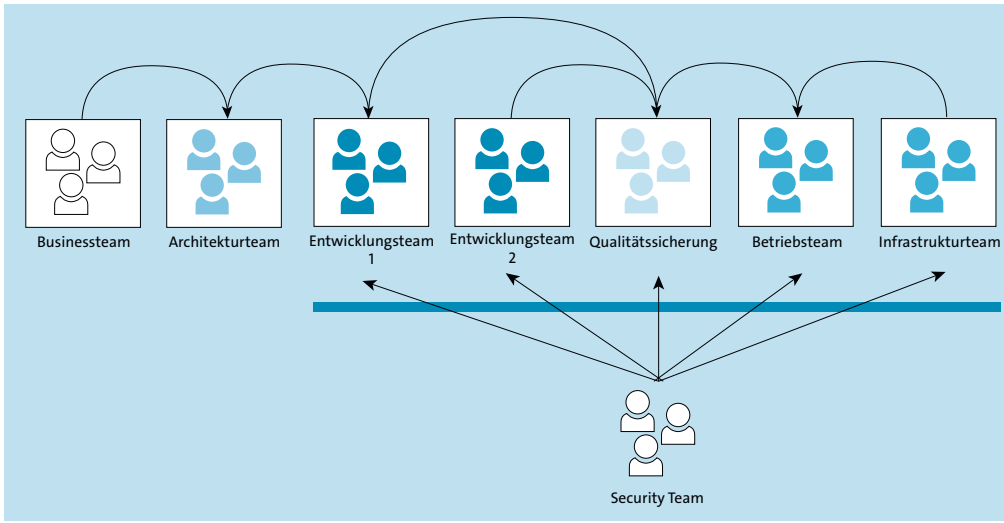

Abbildung 3.7 Die gesamte Teamstruktur

Auch hier dürften Sie sicherlich das eine oder andere Problem aus Ihrer Organisation wiedererkannt haben. Wenn Sie in der Technik arbeiten, dann sind Anpassungen an der Kultur nur in einem gewissen Rahmen möglich. Je höher in der Hierarchie Sie aber angesiedelt sind, desto eher haben Sie die Möglichkeit, aktiv die Kultur mitzugestalten.

© Rheinwerk Verlag, Bonn 2024

Kapitel 4
Projektmanagement und Planung

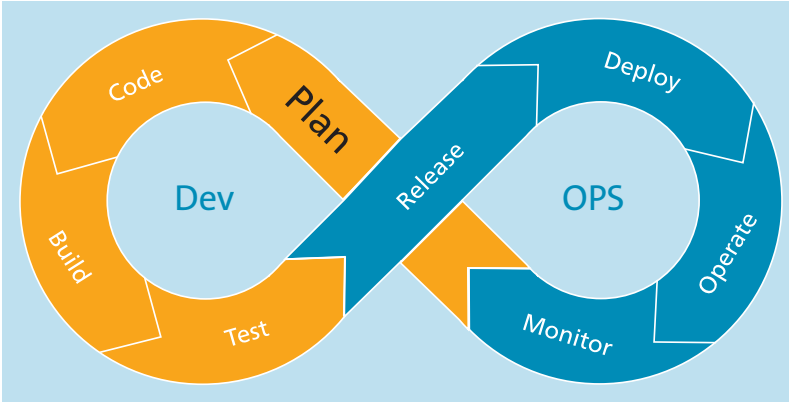

Abbildung 4.1 Plan: In dieser Phase werden Anforderungen gesammelt, Ziele definiert und Aufgaben für die Entwicklung erstellt.

Im ersten Schritt lohnt es sich, zunächst genauer hinzuschauen, was allein bei der Projektplanung schon alles optimiert und verbessert werden kann.

4.1 Der erste Schritt: Das agile Mindset

Viele Firmen arbeiten heutzutage bereits agil. Das ist gut und wichtig und die erste Grundlage, die geschaffen werden muss. Agiles Arbeiten ist ein großer und wichtiger Schritt weg vom früher üblichen Wasserfall-Modell, das heute aus gutem Grund nicht mehr gängig ist.

Im Wasserfall-Modell wurden langwierige Planungen gemacht, die dann über den Zeitraum von sechs Monaten von den verschiedenen Teams implementiert und getestet wurden. Anforderungen, die sich im Laufe der Zeit änderten, wurden gekonnt ignoriert und konnten schlicht nicht berücksichtigt werden, da das Wasserfall-Modell kaum flexibel ist.

Problematisch ist, dass die agile Transformation in vielen Fällen nur halbgar durchgeführt worden ist. Häufig wurde nur die reine Entwicklungsarbeit agilisiert. Was fehlte, war der Blick auf den gesamten *Value Stream*, denn dieser liefert schließlich den Wert für die Endnutzer. So sieht man leider oftmals einen *agilen Wasserfall*: Es

wurde zwar in Sprints gearbeitet, das Delivery aber gar nicht für Endnutzer zur Verfügung gestellt.

Ein solcher Prozess begann mit der Anforderungsanalyse und -spezifikation, gefolgt von der Designphase und dann der Implementierungsphase. Erst danach wurde getestet und anschließend veröffentlicht. Gravierende Fehler, die etwa in der Qualitätssicherungsphase auftraten, müssten eigentlich in der Design- oder Implementierungsphase korrigiert werden, die nun allerdings längst vorüber war. Bei jeder Stufe wurde die Software über die Mauer zum nächsten Team geworfen, und die Teams kommunizierten unzureichend miteinander.

Der benötigte Zeitraum von sechs Monaten zwischen der Anforderungsplanung und dem Fertigstellen der Anforderungen erwies sich schon länger als zu lang. Nicht nur ändern sich die Anforderungen stetig, sondern auch das Deployment auf die Zielplattform erfährt stellenweise gravierende Änderungen, da auch diese weiterentwickelt wurde, sei es durch Austausch der Hardware oder durch ein Aktualisieren der Betriebssystemversion.

Der Zeitraum zwischen Planung und Deployment liegt hier, wie schon erwähnt, bei etwa sechs Monaten. Wenn man sich ein Arbeitspaket anschaut, dann können also zwischen dem Anlegen der Aufgabe und dem Ausrollen der Änderung bis zu sechs Monate vergehen.

Diese Zeit wird *Lead Time* genannt. Daneben gibt es noch die *Cycle Time* (siehe Abbildung 4.2). Sie entspricht der Zeit, die zwischen dem Beginn der Arbeit an einer Aufgabe und der Fertigstellung inklusive Deployment vergeht. Beides sind typische Metriken aus der DevOps-Welt, die genutzt werden, um bewerten zu können, wie gut DevOps umgesetzt wird.

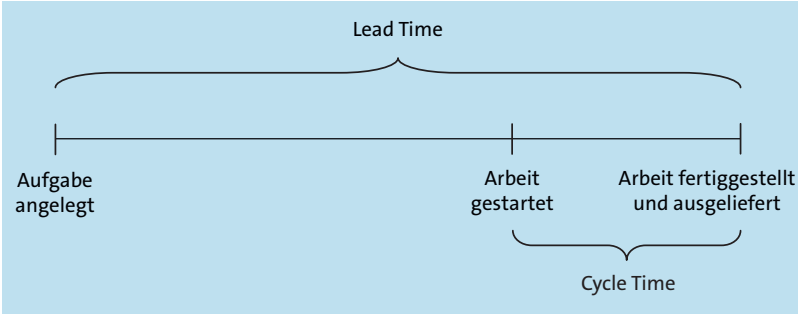

Abbildung 4.2 Die »Lead Time« ist die Zeit, die zwischen dem Anlegen einer Aufgabe und der Fertigstellung liegt, während die »Cycle Time« die Zeit zwischen dem Start der Arbeit und der Fertigstellung bezeichnet.

Grundsätzlich gehört eine möglichst kurze Lead Time zu einem der Hauptpfeiler von DevOps. Bei einer kurzen Lead Time lassen sich Änderungen möglichst schnell ausrollen, was an vielen Stellen sehr hilfreich sein kann, etwa um schnell auf sich

© Rheinwerk Verlag, Bonn 2024

ändernde Marktbedingungen reagieren zu können, Sicherheitslücken schließen zu können oder auch Fehlerkorrekturen schnell ausführen zu können. Genauso wie es im ersten Weg von den Drei Wegen aufgezeigt wird.

Bevor man das allerdings umsetzen kann, muss man die Projektplanung anschauen. Das Wasserfall-Modell ist heutzutage ja in vielen Unternehmen aus guten Gründen durch agile Methoden ersetzt worden. Das Hauptaugenmerk liegt hierbei auf der schnelleren und effizienteren Entwicklung mit einer höheren Qualität der Software.

Die agile Software-Entwicklung begann zu Beginn der 2000er-Jahre mit der Veröffentlichung des *Agile Manifesto*, das Sie unter *https://agilemanifesto.org/* finden. In diesem werden die zwölf Grundprinzipien der agilen Software-Entwicklung definiert.

Die Prinzipien der agilen Software-Entwicklung als auch von DevOps ähneln und überschneiden sich. So geht es bei beiden unter anderem darum, Änderungen grundsätzlich gut zu finden, häufig Releases zu veröffentlichen, die Zusammenarbeit im Team zu stärken, sowie auch um die Reflexion und die Anpassung von Software und Mensch.

Die 12 Grundprinzipien der agilen Software-Entwicklung

Die zwölf Grundprinzpien der agilen Software-Entwicklung nach dem *Agile Manifesto* lauten:

1. Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufriedenzustellen.
2. Heiße Anforderungsänderungen selbst spät in der Entwicklung willkommen. Agile Prozesse nutzen Veränderungen zum Wettbewerbsvorteil des Kunden.
3. Liefere funktionierende Software regelmäßig innerhalb weniger Wochen oder Monate, und bevorzuge dabei die kürzere Zeitspanne.
4. Fachexperten und Entwickler müssen während des Projekts täglich zusammenarbeiten.
5. Errichte Projekte rund um motivierte Individuen. Gib ihnen das Umfeld und die Unterstützung, die sie benötigen, und vertraue darauf, dass sie die Aufgabe erledigen.
6. Die effizienteste und effektivste Methode, um Informationen an und innerhalb eines Entwicklungsteams zu übermitteln, ist im Gespräch von Angesicht zu Angesicht.
7. Funktionierende Software ist das wichtigste Fortschrittsmaß.
8. Agile Prozesse fördern nachhaltige Entwicklung. Die Auftraggeber, Entwickler und Benutzer sollten ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können.
9. Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität.
10. Einfachheit – die Kunst, die Menge nicht getaner Arbeit zu maximieren – ist essenziell.

Persönliches Exemplar für Thomas Bartholomäus

11. Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams.
12. In regelmäßigen Abständen reflektiert das Team, wie es effektiver werden kann, und passt sein Verhalten entsprechend an.

Agile Software-Entwicklung ist heutzutage in vielen Firmen verbreitet und auch bekannt, weswegen ich hier nicht allzu tief auf sie eingehen werde. Obwohl das Konzept schon über 20 Jahre alt ist, hat die Adaption von Agilität erst langsam Fahrt aufgenommen, sodass man heute bereits sagen kann, dass sie sich etabliert hat, anders als eben DevOps.

In der agilen Software-Entwicklung wird bekanntlich in Sprints gearbeitet, die möglichst kurz gehalten werden. Anstatt wie bisher einem sechsmonatigen Entwicklungszyklus zu folgen, wird in der Regel eher mit zwei Wochen gearbeitet, während denen geplante Entwicklungen umgesetzt werden sollen.

Aber was ist nun der Unterschied zwischen agiler Software-Entwicklung und DevOps? Grundsätzlich kann man sagen, dass DevOps eine Ergänzung von agiler Software-Entwicklung ist. Wie zuvor schon kurz angerissen, wird bei Letzterer allerdings überwiegend die Entwicklung betrachtet und alles, was danach kommt, wird nahezu außer Acht gelassen. Es wird zwar von häufigen Releases und Deployments gesprochen, aber wie das genau passiert, wird nicht weiter erwähnt.

Dabei ist genau das ja das Spannende. Was können wir von dem deployten Release lernen? Wie nutzen es die Endanwender und Endanwenderinnen? Mit diversen Metriken können hier noch viel mehr Informationen gewonnen werden, was für die nächste Iteration nützlich sein kann.

Aber zurück zum Projektmanagement: Der agile Wasserfall musste als Erstes weg, um den Weg zu DevOps zu ermöglichen. Eine Lead Time von sechs Monaten ist definitiv zu lang. Stattdessen soll in schnelleren Zyklen entwickelt werden, um schneller auf sich ändernde Anforderungen reagieren zu können.

Am Ende des Sprints soll die Software so weit sein, dass man diese nicht nur theoretisch, sondern auch praktisch ausliefern kann. Als Grundlage hierfür wird eine engere Kollaboration geschaffen, indem alle relevanten Stakeholder involviert werden. Diese sitzen gemeinsam an einem Tisch, statt ihr Arbeitsergebnis wie bisher über die Wand zum nächsten Team zu schmeißen.

Scrum

Häufig wird agiles Projektmanagement mit Scrum gleichgesetzt. Das ist so nicht richtig. Scrum ist *eine* Möglichkeit. Daneben gibt es noch weitere, wie etwa Kanban oder auch Scaled Agile Framework.

© Rheinwerk Verlag, Bonn 2024

Dies ist allerdings nur der erste und auch nur ein kleiner Teil von DevOps, der im weiteren Vorgehen weiter angepasst werden muss.

Reflexion

Agilität ist der erste Schritt in die DevOps-Welt. Wenn Sie bisher nicht in einer agilen Arbeitsweise Software entwickeln, dann wird eine Adaption von DevOps-Prinzipien kaum möglich sein. Zum Glück ist die Wahrscheinlichkeit, schon hier zu scheitern, relativ gering, da heute viele mit agilen Methoden arbeiten.

4.2 Projektmanagement für alle?

Die erste Stage im Software-Development-Lifecycle ist die Planungsphase. Schauen wir uns also an, wie das Projektmanagement aus dem DevOps-Blickwinkel funktioniert.

Auf dem Markt gibt es zahlreiche kommerzielle und nicht kommerzielle Tools für das Projektmanagement. Platzhirsch und weitestgehend unangefochtene Nummer 1 im agilen Projektmanagement ist *Jira* von der Firma Atlassian. Fast alle, die dieses Buch lesen, dürften schon einmal mit Jira in Kontakt gekommen sein.

Jira ist als Branchenstandard für diese Aufgaben kaum wegzudenken. Viel zu tief ist es in sehr vielen Prozessen von Organisationen implementiert – und das mit allen Vor- und Nachteilen. Je nachdem, wen Sie fragen, ist das ein Fluch oder ein Segen. *Projektmanager*, *Product Owner*, *Agile Coaches* oder *Scrum Master* schwärmen von Jira und kennen häufig nicht viel anderes. Das führt häufig so weit, dass Scrum mit Jira gleichgesetzt wird.

Wenn man Techniker fragt, wie sie zu Jira stehen, dann hört man eher skeptische Stimmen. Für die einen ist es ein notwendiges Übel, das zum Projektmanagement halt dazu gehört. Andere hassen es schlicht und sind froh, wenn sie es nicht anfassen müssen. Einige Auswüchse von Letzterem kann man auch auf *https://ifuckinghatejira.com* nachlesen. Die wichtigsten Kritikpunkte sind, dass Jira ein Management-Werkzeug ist und in den Händen einer schlechten Projektleitung für mehr Aufwand und Ärger als Nutzen sorgen kann. Es hat die Tendenz, die Entwicklungsarbeit zu bürokratisieren und zu verkomplizieren.

Aber bleiben wir ganz nüchtern bei den Fakten: Jira lässt sich vielfältig einsetzen. Als vollumfängliches Projektmanagement-Tool kann es als Issue-Tracker oder auch als Workflow-Tool eingesetzt werden. Sie können damit Aufgaben in Form von Tickets sammeln und sie an Mitglieder Ihres Teams zuweisen. Mein Fokus hier ist primär das Issue-Tracking für das agile Projektmanagement.

Persönliches Exemplar für Thomas Bartholomäus

4.2.1 Jira kann (fast) alles

Eine große Stärke von Jira ist, dass Sie die Software bis ins kleinste Detail anpassen können und dass es zahlreichen Plugins gibt, mit denen Sie den Funktionsumfang erweitern können. Das ist grundsätzlich – unabhängig vom genutzten Tool – Fluch und Segen zugleich.

Der Fluch, nicht nur bei Jira, ist, dass jede Installation mit Hausmitteln und mit Plugins stark verändert werden kann. Für einen Engineer, der sich schlicht die Issues anschauen möchte und seinen Fortschritt tracken will, sieht dann jedes Projekt anders aus, weil alles individuell angepasst wurde. Das mag zunächst erstmal wie ein Vorteil wirken, ich betrachte es mittlerweile allerdings größtenteils als Nachteil. Wenn es zu viele Möglichkeiten gibt, zu einer Lösung zu kommen, muss man sich immer wieder neu einfinden.

Nicht selten habe ich zudem schon erlebt, dass nicht nur eine, sondern gleich mehrere Jira-Instanzen in einer Firma existieren, die alle von unterschiedlichen Teams gepflegt werden. Jede Instanz arbeitet in jedem Projekt durch den Einsatz von Plugins und hart verdrahteten Workflows ganz unterschiedlich. Für Personen, die nicht in diesen Projekten mitarbeiten, ist es komplett undurchsichtig, wie etwas funktioniert. Wer zwischen Projekten hin- und herspringt, muss ständig umdenken und sich an andere Workflows gewöhnen.

Initiativen wie *Inner Sourcing* funktionieren auch nicht sinnvoll, wenn es zu viele Jira-Instanzen gibt: Zu viele Zugänge mit den richtigen Berechtigungen müssen erstellt und verwaltet werden, zu hoch ist der Aufwand, sich woanders zurechtzufinden. Unabhängig vom konkreten Projektmanagement-Tool ist das ein Anti-Pattern – insbesondere im Hinblick auf die DevOps-Kultur, da das Anlegen von Tickets möglichst *einfach* gestaltet werden sollte. Ziel sollte immer sein, dass das Tool die Prozesse und somit die Menschen unterstützt. Getreu dem Motto: *People over Processes over Tools*.

Wenn Sie daher eine Software für das Projektmanagment einsetzen, egal ob es Jira ist oder ein anderes Tool: Achten Sie darauf, dass die Anpassungen und Individualisierungen nicht überhandnehmen. Sonst haben Sie am Ende mehr Management-Aufwand als zuvor, was gerade vermieden werden sollte. Je einfacher und einheitlicher die Lösung gestaltet ist, desto besser. (Ausnahmen bestätigen natürlich die Regel, müssen aber begründet sein.)

4.2.2 Menschen sind wichtiger als Prozesse

Jira ist ein mächtiges Tool. Aber wie wir spätestens seit Spiderman wissen: *Mit großer Macht kommt große Verantwortung*. Nutzen Sie es also mit Bedacht.

© Rheinwerk Verlag, Bonn 2024

Jira macht es sehr einfach, komplexe Prozesse einzuführen. Diese Prozesse sind dann schnell bürokratisch, was den Entwicklungsflow verlangsamt. Schnell lassen sich in Jira Regeln und Bedingungen für den Projektfortschritt formulieren, was dazu führt, dass die Entwicklerteams nur noch damit beschäftigt sind, Tasks abzuhaken, und kaum noch dazu kommen, Code zu schreiben und wirklich zu arbeiten. Aus Management-Sicht sehen die Regeln notwendig und sinnvoll aus, sie machen die Prozesse allerdings umständlicher und fehleranfälliger. Die große Kunst des Projektmanagements besteht darin, allen Overhead, der nicht notwendig ist, zu kürzen, damit die wirklich wichtigen Regeln umgesetzt werden können.

Aber nicht nur das: Im Projektmanagement spiegeln sich die Arbeitsschritte der Mitarbeiter Ihres Teams. Wenn es zu stark formalisiert wird, entstehen eben jene starren Strukturen, die durch ein flexibles und agiles Projektmanagement gerade abgeschafft werden sollten.

Ein häufiger Workflow ist etwa, dass ein Ticket stets von einem vordefinierten Status zum nächsten wechselt, etwa von »Planning« zu »Backlog« über »Ready for Development«, »In Development«, »In Test«, »In QA« und so weiter, bis es irgendwann im Status »Released to Production« landet, bevor es geschlossen wird.

Fällt Ihnen dabei etwas auf?

Mit diesem Prozess haben Sie genau das Silo-Denken verstärkt, das wir eigentlich abschaffen wollten: Wenn etwas in Planung ist, dann haben die Entwickler häufig nichts zu sagen. Wenn der Arbeitsschritt in die Entwicklung übergeht, dann wartet das QA-Team. Schließlich liegt die Verantwortung bei dem Ops-Team, das sich um das Deployment kümmern muss.

Das ist nicht nur unter reinen DevOps-Gesichtspunkten ungünstig, sondern sorgt häufig auch für komplizierte Workflows, da nicht einfach ersichtlich ist, nach welchem Status welche Aufgaben folgen und was zu tun ist, wenn etwas im Prozess zurück laufen muss.

Wenn Sie also Prozesse durch eine Projektmanagement-Software in starre Pflichtenhefte gießen, laufen Sie Gefahr, dass sich diese Strukturen auch im ganzen System und schließlich auch im Produkt selbst abbilden, das Sie entwickeln. Und das ist im Allgemeinen nicht der beste Weg.

Conway's Law

Diese Erkenntnis ist nicht neu, und wir werden sie in Abschnitt 5.1 genauer betrachten. Kurz gesagt geht es um einen Gedanken des amerikanischen Informatikers Melvin Conway: Die Systeme, die eine Organisation baut, sind immer ein Abbild ihrer Kommunikationsstruktur. Wenn die Gruppe, die an einer Software arbeitet, in drei Teams aufgeteilt ist, wird das daraus resultierende Programm auch aus drei Teilen bestehen. Die Einzelteile der Software entsprechen den Teilen der Gruppe.

Persönliches Exemplar für Thomas Bartholomäus

Achten Sie daher darauf, dass Ihr Projektmanagement-Werkzeug – sei es Jira oder irgendein anderes Tool – die Menschen im Team nicht zu einem bestimmten komplexen Workflow zwingt. Natürlich ist es stellenweise nötig, dass Prozesse vereinheitlicht werden und ihre Einhaltung überwacht wird, aber wichtig ist, dass man die richtige Balance zwischen einem flexiblen, vielleicht sogar kreativen Einsatz der Tools und möglichst standardisierten Abläufen findet. Mein Tipp: Schauen Sie sich an, ob wirklich alle komplexen Prozesse im Projektmanagement genau so gebraucht werden.

Je einfacher die Prozesse sind, desto eher kommen Informationen und wichtige Insights zutage, die sonst vielleicht verborgen werden. Schließlich sitzen da am Ende immer noch Menschen, und diese wollen mit den Tools arbeiten – und nicht dagegen.

Wenn das Projektmanagement gar nicht läuft und man das Gefühl hat, dass man nur mit den Tools kämpft, entsteht schnell der Wunsch nach einem Wechsel: »Wenn wir statt Jira irgendwas anderes einsetzen würden, hätten wir die Schmerzen nicht.« Meiner Erfahrung nach stimmt das nur in den allerwenigsten Fällen. Die genannten Fehler und Unzulänglichkeiten von Jira lassen sich auch bei anderen Tools finden, und schlechte Prozesse lassen sich mit allen Tools umsetzen.

Hinzu kommt, dass die Migration zu einem anderen Tool ziemlich aufwendig ist. Nachvollziehbarerweise lassen sich die wenigsten Organisationen auf eine solche Migration ein. Überraschend ist das nicht, schließlich ist ein solches Tool tief in die Firma integriert. Anstatt also alles umzuwerfen, sollten Sie sich lieber anschauen, wie Sie Ihr Projektmanagement mit Jira (oder einem anderen Tool) organisieren können.

4.2.3 Gutes Projektmanagement nicht nur mit Jira

Beim Projektmanagement, sowohl mit als auch ohne Jira, lässt sich Wildwuchs durch einige Handgriffe und Strukturen vermeiden, sodass möglichst effizient mit den Tools gearbeitet wird:

1. **Sichtbarkeit schaffen und sicherstellen**

   Eines der wichtigsten DevOps-Prinzipien ist die Schaffung und das Sicherstellen der Sichtbarkeit von Informationen. Wichtig ist, dass die verschiedenen Teams möglichst nicht nur in die Tickets von anderen Teams hineinschauen können, sondern sie auch kommentieren können. Das gilt im selben Maße auch für das Erstellen von Tickets. Auf technischer Ebene kann so verhindert werden, dass sich Silos zwischen verschiedenen Teams bilden.

   Dazu gehört ebenfalls die Möglichkeit, auf weitere Informationen von Dritt-Tools zugreifen zu können. Dazu gehört etwa die Darstellung von Pull Requests oder Merge Requests, um den Arbeitsfortschritt sichtbar zu machen, auch wenn keine Programmiererfahrung vorhanden ist.

© Rheinwerk Verlag, Bonn 2024

2. **Templates verwenden**

   Templates sind hilfreich, um sicherzustellen, dass die wichtigsten Informationen in einem Ticket erfasst werden. So kann durch ein Template die nötige Struktur vorgegeben werden, damit diejenigen Personen, die Tickets anlegen, nicht die notwendigen Informationen vergessen. Bei einem Bug-Report zum Beispiel ist es hilfreich, wenn die Nachfragen »Was ist der beobachtete Fehler?«, »Wie kann man ihn reproduzieren?«, und »Was ist das erwartete Verhalten?« existieren, damit diese Informationen nur noch eingetragen werden müssen. Dadurch wird der Anteil von informationsarmen Tickets à la »Funktioniert nicht« deutlich niedriger.

3. **Nah am Standard bleiben**

   Erfinden Sie das Rad nicht neu. Je näher Sie am Standard des eingesetzten Projektmanagement-Tools bleiben, desto besser. Das verringert nicht nur die Komplexität bei der Verwaltung der Tickets, sondern hilft auch, mögliche Probleme bei Upgrades zu verringern. Zudem ist die Hürde beim Einstieg in neue oder andere Projekte für die jeweilige Person deutlich geringer.

4. **Labels zur Organisation verwenden**

   Jira bietet die Möglichkeit, eigene Ticket-Typen anzulegen. So können unter anderem Tasks, Bugs oder Storys angelegt werden. Achten Sie darauf, dass Sie möglichst wenige Typen anlegen und dass Sie, so weit es praktikabel ist, keine oder weniger Custom Fields verwenden. Der Grundgedanke ist, dass hier die Nutzung für die Mitarbeitenden möglichst effizient, mit geringer Komplexität erfolgt.

   Helfen können dafür Labels. Diese lassen sich auch recht schnell ändern und einfügen. Aber auch das bedarf einer gründlichen Überlegung, wie diese definiert und eingesetzt werden können. Hierbei sind die zuvor angesprochenen Templates nützlich, damit vordefinierte Labels ausgewählt werden können. Damit können die Tickets ebenfalls kategorisiert werden und lassen sich auf diese Weise auch nach dem Anlegen schnell anpassen.

5. **Fokus auf einfache Workflows**

   Die Unbeliebtheit von Jira kommt nicht von ungefähr: Häufiger Kritikpunkt sind hausgemachte komplexe Workflows, in denen viele Genehmigungsschritte und sonstige Übergänge von dem einen in den anderen Status vorgesehen sind. Wenn diese unintuitiv und unübersichtlich sind, wird die Nutzung mühsam. Sehen Sie zu, dass Sie unnötig komplexe Workflows vereinfachen, um die Benutzerfreundlichkeit zu erhöhen.

6. **Den Umfang von Tickets möglichst klein halten**

   Achten Sie darauf, dass der Umfang von Tickets möglichst klein gehalten wird. Das ist unabhängig vom konkreten Tool, allerdings können Sie so schneller die DevOps-Prinzipien umsetzen: Viele kleinere Iterationen lassen sich idealerweise

Persönliches Exemplar für Thomas Bartholomäus

zügiger implementieren und bedürfen weniger einer Klarstellung, da es schon zu Beginn der Arbeit klar ist, was alles getan werden muss. Brechen Sie Aufgaben und Feedback in möglichst kleine Teile herunter, die sich gut schrittweise bearbeiten lassen.

7. **Metriken**

   Das **M** im CALMS-Modell steht für Measurement. Und das geht mit Metriken. Achten Sie darauf, welche Metriken zur Verfügung gestellt werden, und schauen Sie regelmäßig, wie die Tendenz der Metriken ist. Das Projektmanagement-Tool bietet viele Daten, mit denen Produktivität gemessen werden kann. Wenn Sie merken, dass sich die Werte verschlechtern, untersuchen Sie, woran das liegen kann. Die Tendenz ist hier allgemein betrachtet wichtiger als die konkrete Zahl, denn jede Firma startet von einem anderen Punkt aus.

   Aber auch hier gilt: Verwechseln Sie Metriken nicht mit Zielen. Wenn die Anzahl der Bug-Reports sinkt und es weniger Tickets mit dem Status *Critical* gibt, ist das natürlich ein gutes Zeichen. Aber geben Sie keinesfalls als Ziel vor, dass es weniger Tickets geben soll: Das führt nur dazu, dass Probleme nicht mehr dokumentiert werden oder zehn unterschiedliche Fehler in ein Ticket gezwängt werden.

4.2.4 Mehr als nur ein reines Projektmanagement-Tool

Wenn Sie sich tief in die Geheimnisse des Projektmanagements einarbeiten wollen, finden Sie im Internet und in anderen Büchern eine Vielzahl von Tipps und Ratschlägen. Was wirklich für Sie und Ihre Situation funktioniert, werden nur Sie selbst sagen können. Auch konkret zu Jira gibt es eine Menge Hinweise – von denen sich die Hälfte widersprechen. Worauf manche schwören, gilt bei anderen als Irrweg. Gut, dass das nicht das Thema dieses Buches ist! Denn aus DevOps-Sicht ist unabhängig vom konkret verwendeten Tool eigentlich nur wichtig, dass die Arbeit im Team und der Fortschritt des Projekts sichtbar sind, idealerweise für alle. Arbeit muss sichtbar sein! Dieses Mantra habe ich schon oft wiederholt.

Aus DevOps-Sicht ist es daher völlig unerheblich, ob etwa nach Scrum oder Kanban gearbeitet wird. Viel wichtiger ist, um die volle Visibilität zu haben, dass die Integration in die übrigen Tools und den gesamten Software-Development-Lifecycle möglich ist und alle mitmachen können.

Ganz unabhängig davon, welche verschiedenen Tools konkret im Einsatz sind, ist es, gelinde gesagt, immer wieder ein Krampf, da nicht alle Informationen einfach zwischen den Tools geteilt werden können und sich nicht einfach aus anderen Quellen ein Ticket im Projektmanagement-Tool anlegen lässt. Zudem fehlt vielen Personen häufig der Zugang zu Systemen, da Informationen abgeschottet sind. So wird Kollaboration schwer gemacht!

© Rheinwerk Verlag, Bonn 2024

Klarer wird es, wenn man sich konkrete Beispiele anschaut. Ein gelöstes Problem ist die Frage, ob an einem Ticket gerade aktiv gearbeitet wird. Es geht also darum, ob es schon Commits gibt, also ob bereits aktiv Code geschrieben wurde. In Jira wird dazu entweder direkt in der Historie angezeigt, dass Arbeiten in Form von Commits am Code stattfinden, oder es gibt zusätzlich Links zum entsprechenden Feature-Branch und Pull bzw. Merge Request. Mit wenigen Klicks erreicht man die aktuellen Entwicklungsarbeiten.

Aber das ist auch nur die halbe Miete. Und beantwortet nur das Offensichtlichste. Wie sieht es aber mit Security-Vulnerabilities aus? Wenn Ihnen auf dem Dashboard eine gefundene Sicherheitslücke angezeigt wird, möchten Sie basierend darauf sogleich ein Ticket anlegen, wollen aber auch direkt sehen, ob nicht vielleicht schon ein Ticket existiert oder ob dieser Fall bereits bearbeitet wurde. Diese Information betrifft gleich mehrere Personen, denn sowohl Führungskräfte als auch die Entwickler als auch das Security-Team möchte wissen und überwachen, wie der Stand ist. Und das geht nur sinnvoll, wenn alle auf die gleichen Informationen Zugriff haben. Ihr Projektmanagement muss also gut in die Security-Scanner integriert sein, obwohl diese Teile auf den ersten Blick nichts miteinander zu tun haben.

Ein anderes Beispiel ist etwa die Handhabung von Feature Flags. In Abschnitt 8.4 geht es um das Beispiel der Firma Knight Capital, die (unter anderem) durch ein versehentliches Wiederbenutzen eines veralteten Feature Flags in den Ruin getrieben wurde.

Wenn Feature Flags einsetzt werden, ist es sinnvoll, dass sofort sichtbar ist, an welchem Ticket jedes einzelne Feature Flag hängt, um sowohl die Implementierung als auch das nachfolgende Aufräumen zu tracken. Die Sichtbarkeit darf sich also nicht auf das Ticket beschränken, das das Feature angefordert hat, sondern muss auch das Tool integrieren, das die Feature Flags überwacht. Das ist nicht unbedingt eine Funktion, die in der heutigen Tool-Landschaft üblich ist, aber Sie sehen hoffentlich, worauf ich hinaus möchte.

Diese Beispiele lassen sich nahezu unendlich fortführen. Was ich damit aussagen will: Es ist wichtig, nicht nur auf das eigentliche Projektmanagement-Tool zu schauen, sondern auch darauf, wie es sich in den gesamten DevOps-Lifecycle integrieren lässt, sodass die volle Visibilität auf allen Ebenen für alle möglich ist, dass also alles nachvollziehbar ist. Ich kenne dafür keine perfekte Lösung, weil immer wieder etwas fehlt.

Mein Ratschlag: Anstatt diese Features in Jira (oder in das Projektmanagement-Tool Ihrer Wahl) zu integrieren, damit dort alles sichtbar ist, kann es gerade für kleinere Projekte sinnvoll sein, die Projektmanagement-Fähigkeiten von GitLab oder GitHub zu nutzen. Sie integrieren also nicht die technischen Aspekte in das Projektmanagement, sondern nutzen die Features der Coding-Tools, um den Fortschritt der Projekte darzustellen und ihn zu verwalten.

Persönliches Exemplar für Thomas Bartholomäus

Beide Plattformen bieten seit geraumer Zeit dazu Funktionen an, und besonders bei den kostenpflichtigen Varianten hat sich in den letzten Jahren einiges getan. Der Fokus lag vor allem auf Software-Entwicklungsprojekten. Während es bei Jira, Asana oder ähnlichen Tools um allgemeines Projektmanagement geht, ganz unabhängig von der Software-Entwicklung, setzen die beiden Plattformen auf den Background in der Verwaltung von Code. Allein die Möglichkeit, flexibel Labels zu setzen, ohne zwangsläufig Workflows zu definieren, hilft, das Projektmanagement einfach zu halten. Für Teams, die nach DevOps-Prinzipien arbeiten, ist ein Projektmanagement-Tool, das flexibel mit Dingen wie Labels und Milestones arbeitet, viel einfacher zu nutzen, da es speziell auf Software-Entwicklungsprozesse ausgerichtet ist und einfach nach einzelnen Tickets gefiltert werden kann.

Diese vereinfachen dann den Entwicklungsprozess. Der Nachteil ist, dass die meisten Projektmanager erst einmal ein anderes Tool lernen müssen. Hier kann es also, wie so häufig, mögliches Konfliktpotenzial geben; wie immer müssen alle Beteiligten abgeholt werden. Die Verantwortlichen müssen diese Probleme erkennen und entsprechende Entscheidungen treffen. Auch mit einem einfachen Projektmanagement-Tool direkt am Sourcecode werden nicht alle Probleme gelöst, da sie manchmal auch viel zu stark vereinfacht sind.

Dieser Abschnitt zeigt letztlich allerdings auch wieder: Auch bei der Auswahl des richtigen Tools ist die Kultur wichtig, und das richtige Tool mit den richtigen Einstellungen unterstützt die Kultur und die Prozesse – oder auch nicht, wenn man es nicht richtig umsetzt.

4.2.5 Projektmanagement bei schick-gekleidet.de

Auch für *schick-gekleidet.de* hieß es, dass eine Modernisierung und Vereinfachung der verwendeten Technologien umgesetzt werden sollte. In der Vergangenheit wurde viel mit Jira verwaltet. Es gab zahlreiche verschiedene Installationen; so gab es zwar für die Entwicklungsteams eine gemeinsame Jira-Instanz, aber die QA-Teams und die Betriebsteams hatten abermals separate Installationen. Dazu gesellten sich noch ein paar weitere, die sich um die Verwaltung der Firma selbst drehten. Jede Installation hatte einen unterschiedlichen Satz an Plugins, sodass die Nutzung und Wartung umständlich waren.

Im ersten Schritt sollte das Projektmanagement für die einzelnen Teams cross-funktional aufgebaut werden, sodass alle Abteilungen Zugriff darauf bekommen. Das Ziel ist, dass durch die Transparenz die Zusammenarbeit erleichtert wird. Für die Verwaltung der Firma selbst wurde im ersten Schritt nichts geändert, denn bestehende Prozesse sollten beibehalten werden. Diese lassen sich in einem klassischen Projektmanagement-Tool wie Jira gut abbilden.

© Rheinwerk Verlag, Bonn 2024

Ein wichtiges Kriterium für die Auswahl des neuen Tools war, dass nicht nur die Features für das Projektmanagement angeschaut werden. Stattdessen soll die Anzahl der Tools reduziert werden, und die Werkzeuge sollen nah am Code und den CI/CD-Pipelines liegen. Dementsprechend lag es nahe, sich GitHub und GitLab näher anzuschauen.

Als problematisch wurde eher angesehen, dass beide stark auf »Labels« setzen, was zwar ziemlich flexibel ist, aber auch nicht eine einfache Lösung bietet, wie die Struktur in den Projekten aufgebaut werden sollte.

Reflexion

Das Projektmanagement-Tool zu ersetzen oder einzuführen, ist keine einfache Aufgabe. Wie Sie vielleicht unschwer erkennen konnten, bin ich kein großer Fan von komplexen Setups – zumindest dann nicht, wenn es um die Entwicklung und den Betrieb von Software geht.

In vielen Organisationen ist das Ersetzen des zentralen Projektmanagement-Tools nicht einfach möglich, und das auch aus guten Gründen. Aus DevOps-Sicht sind zwei Dinge wichtig: die Visibilität für alle relevanten Personen sowie die einfache Nutzbarkeit.

Die Visibilität ist nicht auf das Zugänglichmachen des Ticketsystems beschränkt. Hier geht es mehr darum, dass auch die relevanten Informationen ebenfalls Teil der Entwicklungs- und Betriebsprozesse werden, sodass man schnell und zügig die relevanten Informationen zur Hand hat. Schauen Sie, was Sie integrieren und optimieren können, damit das Tool hilfreich im Alltag ist und nicht im Weg steht.

4.3 Fazit

Dieses Kapitel behandelte den Planungsteil. Lange vor der Etablierung von DevOps startet die Transformation schon mit der agilen Softwareentwicklung. Ohne Agilität gibt es quasi kein DevOps, denn es ist der logische nächste Schritt.

Effizientes und transparentes Projektmanagement ist der Grundstein für einen erfolgreichen Entwicklungs- und Betriebsprozess. Wichtig ist, dass dabei fortlaufend das Feedback der vorherigen Entwicklungszyklen und die Informationen aus dem Betrieb stetig und in möglichst kleinen Iterationen eingeplant werden.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 5
Kollaboration beim Coden

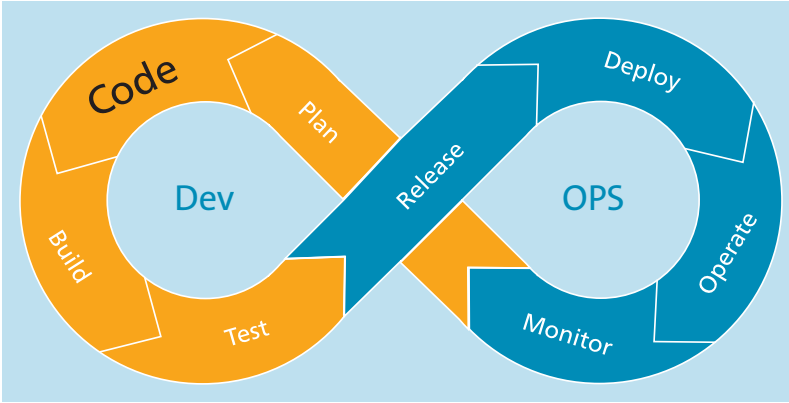

Abbildung 5.1 Code: In der Entwicklungsphase wird Code geschrieben und ins Repository eingecheckt.

Nachdem Sie in Kapitel 2 einen Einstieg in die Idee hinter DevOps erhalten haben und in Kapitel 4 die Planung des DevOps-Lifecycles thematisiert wurde, ist es nun an der Zeit für den nächsten Schritt des DevOps-Lifecycles: den Code.

Wenn Sie sich den DevOps-Lifecycle anschauen, dann erscheint der Code auf den ersten Blick eher als ein sehr technischer Teil des Lifecycles. Zu einem gewissen Grad stimmt das natürlich, tatsächlich geht es hier aber um viel mehr:

- Wie wird die Arbeit am Code organisiert?
- Wie wird im Team kollaborativ gearbeitet?

Diese Fragen haben zunächst wenig mit der Technik oder den verwendeten Programmiersprachen, Frameworks und Werkzeugen zu tun. Sie bilden den Grundpfeiler für das effektive und effiziente Zusammenarbeiten im Projekt, da alle anderen Stages auf den Code aufbauen.

Ziel dieses Kapitels ist es daher, die verschiedenen DevOps-Prinzipien hauptsächlich aus dem Blickwinkel des Programmierens zu betrachten.

Persönliches Exemplar für Thomas Bartholomäus

5.1 Die typischen Probleme bei der Verwaltung des Sourcecodes

Werfen wir zunächst einen Blick auf den Quellcode und den Coding-Workflow bei *schick-gekleidet.de*. Dort finden sich rasch einige Probleme und Herausforderungen, die sich im Wesentlichen in die drei folgenden Punkte aufteilen lassen:

- Organisation rund um den Code
- Entwicklungs-Workflow à la Kraut und Rüben
- erschwerte Einarbeitung durch Technical Debt

In dieser frühen Phase des Entwicklungsprozesses sorgten diese Probleme dafür, dass die Zusammenarbeit zwischen den Teams nicht gut funktionierte. Zudem führten die Organisationsschwächen zu vielen vermeidbaren Schwächen am eigentlichen Produkt.

Bugs und Programmierfehler sind natürlich nie ganz auszuschließen, aber durch die Mängel in der Sourcecode-Verwaltung vermehrten sie sich besonders stark und durchzogen ganze Projekte.

Im Folgenden gehe ich tiefer auf diese Probleme und Herausforderungen von *schick-gekleidet.de* ein. Der Fokus liegt insbesondere auf den Folgen, die sich durch diese schlechte Arbeitsweise ergeben.

5.1.1 Organisation rund um den Code

Neben der Architektur der Software hat auch die Organisation des Quellcodes starke Auswirkungen darauf, wie die Software entwickelt wird. Der strukturelle Aufbau beeinflusst so die Zusammenarbeit der Teams. Bislang wurde der Quellcode der Anwendungen, die bei *schick-gekleidet.de* entwickelt wurden, in einem Subversion-Repository versioniert. Welches Tool exakt zum Einsatz kommt, spielt dabei gar keine so große Rolle; viel wichtiger ist die Frage, wie der Quellcode verwaltet wurde und wie die Repositorys eingerichtet wurden.

Abbildung 5.2 zeigt schon das grundsätzliche Problem, das es in vielen Firmen gibt: Die verschiedenen Teams mit ihren unterschiedlichen Aufgabenbereichen arbeiten abgeschottet in ihren eigenen Repositorys. Ein Einblick in den Code oder gar ein Mitarbeiten am gleichen Quellcode ist so nicht möglich.

Sie sehen schon, dass die Kollaboration der einzelnen Teams so nicht wirklich möglich war, denn die einzelnen Projekte und Teams haben nur in ihren Silos gearbeitet.

Jedes Team kann auch nur Dateien im eigenen Repository ändern; die Teammitglieder hatten keinerlei Einblicke in die Arbeit der anderen Teams. Fehler oder Probleme wurden also eher durch Workarounds im eigenen Team und im eigenen Quellcode behoben, anstatt die Probleme an der echten Quelle zu korrigieren.

© Rheinwerk Verlag, Bonn 2024

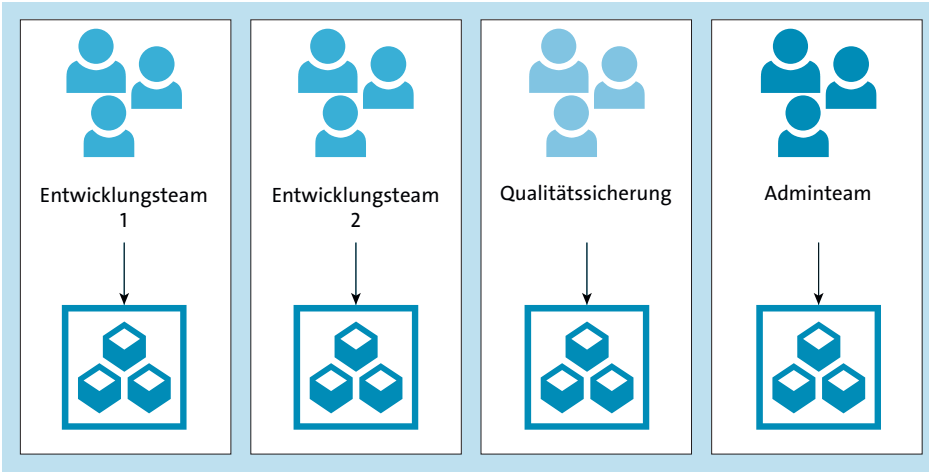

Abbildung 5.2 Die Wände zwischen den Teams wurden dadurch verstärkt, dass die Repositorys der jeweiligen Teams immer nur für das eigene Team sichtbar sind.

Conway's Law

Die Struktur der Teams ist nicht nur nettes Beiwerk, sondern hatte große Auswirkungen auf das ganze Produkt. Diese Erkenntnis wird *Conway's Law* genannt. Vereinfacht ausgedrückt besagt dieses »Gesetz«, dass die Software-Architektur von der Struktur der Organisation abhängt. Und das ist nicht positiv gemeint, sondern als Problem zu betrachten.

Conway's Law stammt von Dr. Melvin Conway, der es bereits 1968 formulierte. Seine Aussage lässt sich auch etwas anders ausdrücken: Die Architektur der Software ist ein Abbild der Organisationsstruktur der Firma.

Wenn in Ihrer Firma vier Software-Entwicklungsteams existieren, dann wird Ihr Produkt auch aus vier einzelnen Projekten bestehen, die unabhängig voneinander entwickelt werden, obwohl sie eigentlich zusammengehören. Wie gut diese vier Teile zusammenpassen, hängt davon ab, ob diese Teams gut und eng zusammenarbeiten. Wenn es in der Kommunikation zwischen zwei Teams hakt, wird es auch im Produkt an genau dieser Stelle Probleme geben. Das ist auch nicht verwunderlich, da es innerhalb eines Teams immer am einfachsten ist, zu kommunizieren und zusammenzuarbeiten.

Es ist sehr schwer, Conway's Law zu entkommen. Es bedarf eines sehr großen Management-Einsatzes, um mehrere Teams so zu koordinieren, dass die Kommunikationsstrukturen keine Auswirkungen auf die Arbeit haben – vielleicht ist das auch schlicht unmöglich, denn wenn man es wirklich versuchen würde, wäre der Overhead unglaublich groß, denn man wäre mehr mit der Koordination als mit der eigentlichen Arbeit beschäftigt.

Persönliches Exemplar für Thomas Bartholomäus

Wir können Conway's Law nicht nur als Warnung auffassen, sondern eine Lehre daraus ableiten: Je einfacher und unkomplizierter die Kommunikationsstrukturen beim Entwurf und der Umsetzung eines Projekts sind, desto besser wird das Gesamtergebnis sein.

Die Frage, die sich hier nun stellt, lautet also: Wie kann man die Zusammenarbeit der Teams so organisieren und strukturieren, dass sie vorteilhaft für das Softwareprodukt ist? Das Niederreißen der Mauern zwischen den Abteilungen, damit cross-funktionale, flexible Teams entstehen, ist ein Schritt in die richtige Richtung.

5.1.2 Abschottung aus vermeintlichen Sicherheitsgründen

Schauen wir uns an, wozu die Abschottung der Teams bei *schick-gekleidet.de* genau führte. Aber welche Teams sind überhaupt gemeint?

Zum einen die Entwicklungsteams selbst, die ihre eigenen Teile des Online-Shops von *schick-gekleidet.de* hinter verschlossenen Türen in ihren Kämmerlein entwickeln. Neben den Entwicklungsteams gibt es aber noch sehr viel mehr Teams, etwa das Qualitätssicherungsteam, das den Quellcode seiner Automatisierungen ebenfalls abgeschottet von den eigentlichen Projekten versioniert hat. Besonders schwer wiegt aber, dass das QA-Team keinen Einblick in die Entwicklungsarbeiten der Entwicklungsteams hatte.

Diese Arbeitsweise hatte sich nicht zufällig entwickelt, denn auf den ersten Blick sieht es gar nicht so unlogisch aus: Hier wird nach dem *Need-to-know*-Prinzip gearbeitet. Es dürfen also nur diejenigen den Sourcecode sehen, die auch mit ihm arbeiten müssen. Man hatte dieses Pattern also etabliert, weil bei einem etwaigen Einbruch über ein Benutzerkonto des QA- oder Dev-Teams auf diese Weise nicht der komplette Sourcecode der ganzen Firma abhandenkommen kann.

So wollte man vermeiden, dass durch den gestohlenen Quellcode Lücken in der eigentlichen Shop-Software gefunden werden, die etwa Zugang zu Kundendaten ermöglicht hätten. Das sollte natürlich nicht passieren.

Aber nicht nur an Angriffe von außen wurde gedacht. Ein weiterer Grund für die Abschottung war, dass man es Mitarbeitenden mit bösen Absichten schwerer machen wollte, an die Daten zu kommen. Der Hintergrund war also ein tiefes Misstrauen gegen die eigenen Mitarbeitenden, besonders gegen das Kollegium außerhalb des eigenen Teams.

Diese Sicherheitsbedenken sind nicht vollkommen von der Hand zu weisen, aber es sind häufig nur Alibi-Argumente, die mit ein wenig Konfigurationsarbeit im Sourcecode-Management zu beheben gewesen wären. Was für diverse Teamleitungen ein viel wichtigerer Grund war, der aber nicht offen kommuniziert wurde: Sie wollten eigentlich überhaupt nicht, dass andere Leute mögliche Sicherheitslücken, mindere Code-Qualität und andere Probleme im Code entdecken.

© Rheinwerk Verlag, Bonn 2024

Zum einen empfanden sie so etwas als sehr peinlich, und unter Umständen hätte es vielleicht auch Ärger mit den Vorgesetzten bedeutet. Weiterhin wollte man mögliche zwischenmenschliche Angriffspunkte von anderen Teams vermeiden.

Das führte zu einer gedrückten und angespannten Stimmung, in der weder offen gearbeitet noch offen kommuniziert wurde. Es gab keine echte Transparenz, und wenn dann natürlich doch mal etwas schiefging, übten sich alle in viel Fingerzeigen, weil ja niemand von Problemen beliebiger Art gewusst haben wollte.

Sicherheit ist wichtig

Dass der Status der Projekte sichtbar sein sollte, heißt im Umkehrschluss nicht, dass jeder ohne jegliche Einschränkungen Änderungen tätigen kann! In Kapitel 11 werden wir uns die Sicherheit ansehen.

5.1.3 Lange Entwicklungszeiten ermöglichen kaum schnelle Sicherheitsfixes

Kommen wir aber noch mal auf das Need-to-know-Prinzip und die Geheimhaltung des Sourcecodes zurück. Meiner Erfahrung nach wurde in vielen Projekten normal weitergearbeitet, auch wenn die Entwickler bereits wussten, dass es viele Sicherheitslücken gibt. Diese wurden häufig jedoch nicht behoben, da das Ausrollen der Änderungen sowieso erst in spätestens sechs Monaten erfolgen sollte. Bis dahin war ja schließlich noch viel Zeit, also konnte man sich auch später darum kümmern. Zwischen den Releases war es hingegen gewissermaßen unmöglich, sicherheitskritische Bugs schnell und flexibel zu beheben.

Manchmal musste hier natürlich schon ein Fix für eine Sicherheitslücke ausgerollt werden, wofür viele wichtige Punkte, wie die QA, einfach übersprungen wurden. Zudem wurde mit einer komplexen Branching-Strategie gearbeitet, um die Änderungen in die Produktivumgebung zu bekommen. Den Durchblick bezüglich dieser Strategie hatten aber nur einige wenige langjährige Mitarbeiter.

Häufig versuchte man stattdessen über Umwege, mit irgendwelchen unschönen Lösungen zumindest die schlimmsten Probleme möglichst unschädlich zu machen. Oder man musste noch schnell vor dem nächsten Release einen hässlichen Bug beheben, was aber nur möglich war, indem man eine IP-Adresse »hart« in Code hinterlegte.

In einem anderen Fall wurden gar die Zugangsdaten zu anderen Systemen fest verdrahtet, damit noch alles funktionierte. Das sollte dann natürlich niemand sehen, besonders nicht die anderen Teams. Also sorgte man lieber dafür, dass der Zugang zu Repository gesperrt ist, dann wird's schon niemandem auffallen ...

Dass auch das Qualitätssicherungsteam mit seine Testskripten komplett getrennt vom eigentlichen Entwicklungsteam arbeitet, machte die Kollaboration ebenfalls

sehr viel schwieriger. Obwohl der Code von beiden Teams zusammengehört und sogar voneinander abhängig ist, wurden auch hier die Silos zwischen den Teams aufrechterhalten. Das Qualitätssicherungsteam konnte so nie in den Quellcode der einzelnen Projekte des Online-Shops schauen, weshalb viele Fehler erst sehr viel später gefunden wurden. Der Prozess und die Probleme werden in Kapitel 7 vertieft.

5.1.4 Entwicklungs-Workflow à la Kraut und Rüben

Neben den Problemen mit dem Sourcecode-Management machte sich bei *schick-gekleidet.de* auch das Fehlen eines richtigen Entwicklungs-Workflows bemerkbar: Jede Person und auch jedes Team integrierte ihren bzw. seinen Code nach Gutdünken. Woran das lag, habe ich bereits in Kapitel 3 erläutert, als es um die Firmenstruktur von *schick-gekleidet.de* ging.

Das Hauptproblem war, dass die Entwicklungsarbeiten für viele neue Funktionen des neuen Online-Shops immer sehr lange gedauert haben. Auslöser dafür war, dass in der Planungsphase riesige Aufgabenpakete definiert wurden, die in der Sprint-Planung nur geringfügig verfeinert wurden. Die Änderungen, die alle Personen im Entwicklungsteam vorgenommen hatten, wurden daher immer nur am Ende der Entwicklungsarbeit in das Repository integriert, davon einige über mehrere Sprints hinweg.

5.1.5 Big-Bang-Integrationen

Daraus ergab sich, dass zwangsläufig alle Änderungen *Big-Bang-Integrationen* waren. Das heißt, anstatt kleiner, überschaubarer Häppchen mussten die Ergebnisse von wochen- oder monatelangen Arbeitsphasen in den Hauptzweig des Projekts integriert werden. Diese Änderungen waren sehr umfangreich, da viele Zeilen angefasst und verändert wurden, was die Integration in das Hauptprojekt merklich erschwerte.

Und da immer wieder viele große Änderungen gleichzeitig integriert werden mussten, gab es viele Wartezeiten, da mit jeder fertigen Integration alle Mitglieder des Teams den Code aus dem Repository erneut herunterladen mussten. Das bedeutet natürlich auch, dass die eigenen Änderungen wieder daran angepasst werden mussten.

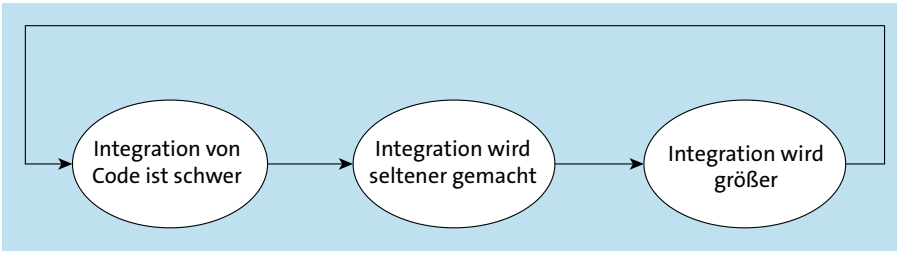

Abbildung 5.3 Durch Big-Bang-Integrationen entsteht ein Teufelskreis.

© Rheinwerk Verlag, Bonn 2024

Das erhöhte letztlich jedes Mal auch den Frust unter den Entwicklern und Entwicklerinnen, da das Mergen der restlichen Änderungen sehr mühselig und fehleranfällig war. Das führte dazu, dass viele Arbeiten weitestgehend von vorn begonnen werden mussten, da größere Refactoring-Schritte nicht abgesprochen waren, aber trotzdem durchgeführt wurden.

Nicht nur der Frust war hier ein großes Problem, auch verzögerten sich die Entwicklungsarbeiten ziemlich stark, da es immer wieder zu Wartezeiten durch die nötigen lokalen Korrekturen kam. Und das hatte letztlich Auswirkungen auf die anderen Teams, die auf die rechtzeitige Übergabe angewiesen waren, um ebenfalls ihren Teil passend abzuliefern.

Der Verzug, der dabei entstand, sorgte immer wieder für Ärger für das Qualitätssicherungsteam. Das hatte zur Folge, dass viele Fehler unentdeckt blieben, die wiederum dem Entwicklungsteam angelastet wurden, da die Verzögerungen schließlich dort ihren Ursprung hatten und der Grund waren, weswegen das Qualitätssicherungsteam nicht genügend Zeit für eine ordentliche QA-Phase hatte. Die Entwicklungsteams selbst haben hingegen nur stellenweise erkannt, dass die großen und häufig ungenauen und unspezifischen Aufgabenpakete von den Teams davor ebenfalls ein Problem darstellten.

Die Entwickler versuchten die Probleme gekonnt zu umschiffen, damit ihr Quellcode möglichst zeitnah integriert werden konnte. Eine Auswirkung davon war, dass immer wieder vorhandener Code kopiert wurde, der angepasst wurde, damit die Integration zügiger ablief. So konnte man umgehen, dass jemand anderes ebenfalls die gleichen Zeilen anpasste. Der doppelte Code wurde allerdings selten aufgeräumt, sodass weitere Fehler entstanden.

Bisher haben wir allerdings nur die Integration innerhalb *eines* Projekts und somit eines Teams betrachtet. Die Probleme aus den einzelnen Teams kaskadierten, als man die einzelnen Projekte am Ende integrieren musste, um dann den gesamten Online-Shop zum Laufen zu bekommen.

5.1.6 Code-Reviews konnten ein wenig helfen

Vereinzelt kam jemand auf die Idee, Code-Reviews einzuführen. Bisher wurde nämlich immer der Code entwickelt und integriert, ohne dass jemand über die Änderungen schaute. Der Sinn von Code-Reviews lag auf der Hand: Durch die gegenseitige Prüfung konnten schließlich einige Fehler vermieden werden und die Einarbeitung von neuen Kollegen im Team beschleunigte sich.

In den Teams stieß diese Neuerung allerdings auf große Ablehnung: Es werde sowieso schon mehr als genug Zeit für die Entwicklungsarbeit und für die Integration benötigt; es fehle schlicht die Zeit, wenn nun auch noch jemand neben einem sitzen

Persönliches Exemplar für Thomas Bartholomäus

müsse und die Änderungen nachvollziehen solle, die über Wochen oder Monate implementiert wurden. Und außerdem sei da ja noch das Qualitätssicherungsteam, dessen Aufgabe ja ohnehin die Sicherung der Qualität sei. Dann sollten die doch dafür sorgen, dass die Änderungen keinen Schaden anrichten …

All diese Probleme führten zu noch mehr technischen Schulden, da viel Sourcecode mit geringer Qualität entwickelt wurde. Weiterhin traten hier die Probleme auf, die in der Firmenvorstellung zu Beginn von Kapitel 3 schon erwähnt wurden: Das Projekt ließ sich immer wieder nicht bauen, da die Integration nicht einfach möglich war, vorwiegend nach den langen Entwicklungszeiten einzelner Features.

5.1.7 Erschwerte Einarbeitung durch Technical Debt

Der Begriff *Technical Debt* (auf Deutsch ganz einfach »technische Schulden«) beschreibt den Preis, den man zahlt, wenn Abkürzungen genommen werden und man sich auf Kompromisse einlässt, um schnell Code zu liefern, anstatt langfristig optimale Lösungen zu implementieren. Kaum jemand kann sich davon freisprechen: Die Frist naht, der (Arbeits-)Tag war lang, man möchte fertig werden.

Anstatt eine Änderung oder ein neues Feature gründlich zu durchdenken und entsprechend zu testen, ändert man den Code nur so weit, dass es irgendwie funktioniert. Integrationen in andere Teile des Projekts, nachvollziehbare Dokumentation, langfristige Wartbarkeit: Das sind alles Probleme für *Future Me*. Oder für das nächste Teammitglied, das später diese Codezeilen wieder anfassen muss.

Technische Schulden weisen viele Gemeinsamkeit mit finanziellen Schulden auf: Sie können sich auftürmen und später so unbeherrschbar werden, dass das Projekt nicht weitergeführt werden kann. Andererseits sind Schulden (oder Kredite) nicht per se schlecht. Bisweilen kann es notwendig oder sogar sinnvoll sein, Aufwand in die Zukunft zu verlagern, um *jetzt* besser handlungsfähig zu sein und schnell auf Probleme zu reagieren. Sie müssen sich aber immer im Klaren darüber sein, dass diese Schulden später beglichen werden müssen.

Wichtig ist daher, dass man Workarounds, die man einbaut und die zu technischen Schulden führen, dokumentiert. Dabei sollte beschrieben werden, wie hoch diese Schulden sind und was erledigt werden muss, um diese Schulden wieder abzubauen. Nur so lassen sich technische Schulden im Alltag nutzen, ohne dass diese einen zu starken negativen Effekt haben.

Eine häufige Ursache für technische Schulden sind Code-Teile, die mit Copy-and-paste dupliziert wurden. Anstatt eine mehrfach gebrauchte Funktionalität richtig in eine eigene Klasse oder Methode zu kapseln – wozu ja Zugriff auf den Code der Kollegen und damit auf unterschiedliche Repositorys notwendig gewesen wäre –, wurde der gleiche Code einfach als Kopie an unterschiedlichen Stellen genutzt.

© Rheinwerk Verlag, Bonn 2024

Dieser vervielfältigte Code mit nur wenigen Anpassungen hatte starken Einfluss auf die Qualität des Projekts. Der Code wurde nämlich nicht nur verdoppelt, um Integrationsschwierigkeiten möglichst zu umgehen, sondern auch, um neue Implementierungen zu testen und die alte erst einmal beizubehalten. Hintergedanke war da vor allem, dass der Code dann ja direkt schon vorhanden ist und nicht wiederhergestellt werden musste. Ignoriert wurde dabei allerdings, dass Versionsverwaltungsprogramme ja genau dafür existieren, auch ältere Stände wiederherstellen zu können.

Die duplizierten Code-Teile machten auf Dauer nicht nur die Arbeit am Projekt zur Hölle, durch sie wurde auch das Onboarding neuer Teammitglieder sehr aufwendig und somit teuer. Es dauerte einige Monate, bis neue Teammitglieder startklar waren und einen groben Überblick über die Codebasis hatten. Das weitere Problem war, dass die Zugänge zu dem Sourcecode nur manuell vergeben wurden.

5.1.8 Hohe Lernkurve ohne Dokumentation

Ebenfalls gab es kaum Dokumentation in Hinblick auf die Einrichtung der Entwicklungsumgebung. Das wird auch noch näher im nächsten Kapitel beleuchtet. Zu diesem Zeitpunkt geht es zunächst erst einmal um die Einarbeitung in den Code und darum, ihn zu verstehen. Durch die hohen Hürden war es für Neulinge schwierig, schnell einzusteigen, da auch die erfahreneren Teamkollegen wenig Zeit hatten. Das hing hauptsächlich mit den zahlreichen Verzögerungen zusammen, die auftraten.

Verstärkt wurde die Thematik auch noch durch die hohen Entwicklungszeiten der einzelnen Features: Für Neulinge waren somit die Lernkurve und die Hürde, erste Beiträge zum Projekt beizutragen, sehr steil und hoch. Die Integrationsprobleme, die ständig auftraten, machten das Ganze noch schwieriger, da es auch lokal nicht einfach möglich war, loszulegen.

Hier es war dann immer erforderlich, dass eine erfahrenere Person aus dem Team Neulinge in das Projekt einführt, wofür aber häufig kaum Zeit bestand, da die alten Hasen an den eigenen Features arbeiten mussten. Dementsprechend dauerte die Einarbeitung recht lange, da keine guten und einfachen Strukturen zum gegenseitigen Voneinander-Lernen vorhanden waren.

Generell verstärkt wurde diese Arbeitsweise durch das verwendete Tooling. Wie zuvor erwähnt, wurde Subversion eingesetzt. Damit lassen sich zwar Branches verwenden, um parallel Entwicklungsarbeiten zu leisten und diese bei Abschluss zu mergen und somit zu integrieren, allerdings war das immer sehr mühsam, weswegen auch nie jemand Branches verwendet hat.

Aber auch, wenn es nicht mühsam wäre: Die übrigen Probleme, die mehr kulturell und organisatorisch geprägt waren, traten unabhängig davon ohnehin auf. Das ist letztlich auch wieder nur ein Beispiel dafür, dass es erst um die Menschen, dann um die Prozesse und dann um das Tool geht und nicht andersherum.

Persönliches Exemplar für Thomas Bartholomäus

Reflexion

In diesem Abschnitt haben wir uns hauptsächlich auf die Aspekte konzentriert, die sich um die Verwaltung des Sourcecodes drehen. Gerade große Änderungen, die ohne Code-Review integriert werden sollen, macht kurze und schnelle Releases unmöglich. Und das hat direkte Auswirkungen auf Sicherheitslücken, die potenziell ausgenutzt werden könnten.

Sie sollten sich erneut anschauen, welche Punkte auch bei Ihnen auftauchen und welche vielleicht auch nicht. Gerade das Fehlen von effizientem Code-Review und die Arbeit mit großen Integrationen sind typische Fehler, die ich im Austausch mit Kunden sehe.

5.2 Die Organisation des Codes verbessern

Das Programmieren ist der zweite Schritt im DevOps-Lifecycle, der direkt nach der Planung kommt. Für die vorherigen Abschnitte sind daher im Wesentlichen das **C** für Culture und das **S** für Sharing des CALMS-Modells relevant.

Die Kultur, wie sie bei *schick-gekleidet.de* war, ist die Wurzel des Übels. Die Probleme hatten nichts damit zu tun, dass die Entwickler nicht kompetent programmieren konnten, dass die falschen Sprachen genutzt wurden oder der Tech-Stack nicht den modernsten Standards entsprach.

Stattdessen ging es damit los, dass viel versteckt, viel verheimlicht und vieles einfach ignoriert wurde. Dies führte zu einigen Problemen, die ich in den vorherigen Abschnitten hoffentlich anschaulich beschrieben habe. Vielleicht erkennen Sie ja zumindest Teile davon in Projekten wieder, in denen Sie schon einmal gearbeitet haben.

All diese Probleme führten dazu, dass niemand das ganze System verstand – schließlich wird schon der Quellcode hinter verschlossenen Türen entwickelt, ohne dass die Kollegen einen Einblick bekommen. Sie verhinderten zusätzlich, dass schnellere und häufigere Änderungen an die Kunden gebracht werden konnten, so wie es *der erste Weg* besagt.

Damit sowohl die Kultur als auch das Teilen der Information innerhalb der Organisation besser werden, müssen diverse Aspekte optimiert werden. Zunächst einmal müssen Sie sich anschauen, wie das Abspeichern und das Verwalten des Sourcecodes verbessert werden kann.

Der erste Schritt ist eher technischer Natur: Der Sourcecode soll von Subversion (SVN) zu Git überführt werden. Während SVN eine zentrale Versionsverwaltung ist, setzt Git auf einen dezentralen und verteilten Ansatz, sodass die gesamte Historie nicht nur auf einem zentralen Server liegt, sondern lokal auf jedem Klon, also auf den

© Rheinwerk Verlag, Bonn 2024

Geräten des Entwicklerteams. Neben Git gibt es noch weitere verteile Versionsverwaltungsprogramme wie Mercurial oder Bazaar, diese haben allerdings heutzutage keine große Relevanz.

Der Austausch des Versionskontrollsystems bringt auf den ersten Blick aber erst mal keine essenzielle Verbesserung der Ausgangslage. Denn auch mit Git lassen sich die Fehler aus der Verwaltung des Sourcecodes in Subversion wiederholen. Daher lohnt sich der Umstieg nur dann, wenn gleichzeitig ein Blick in die Struktur und die Freigaben des Sourcecodes innerhalb der Firma geworfen wird. Sie müssen sich gut überlegen, wie die Kollaboration grundlegend verbessert werden kann!

In unserem Beispiel sollen die einzelnen Projekte der Firma jeweils in ein Git-Repository überführt werden. Diese Projekte sollen nicht mehr wie bisher komplett verschlossen sein, sondern intern öffentlich werden, um *Inner Sourcing* zu ermöglichen. Die Grundidee ist, dass innerhalb der Firma nicht mehr alles hinter verschlossenen Türen entwickelt werden soll, damit bei Problemen besser zusammengearbeitet werden kann, indem auch Leute aus anderen Teams zum Projekt beitragen können. Ausführlicher wird das Thema Inner Sourcing in Abschnitt 5.5 behandelt. Das Ziel dabei ist, dass grundsätzlich jeder in der Organisation mitmachen kann.

Im Rahmen der Umstellung sollen dann auch die verschiedenen Repositorys der verschiedenen Teams, die aber zu einem Gesamtprojekt gehören, zusammengeführt werden. Das heißt also, der Quellcode des Online-Shops von *schick-gekleidet.de* wird mitsamt den Tests, die vom QA-Team kommen, an einem gemeinsamen Ort gespeichert (siehe Abbildung 5.4).

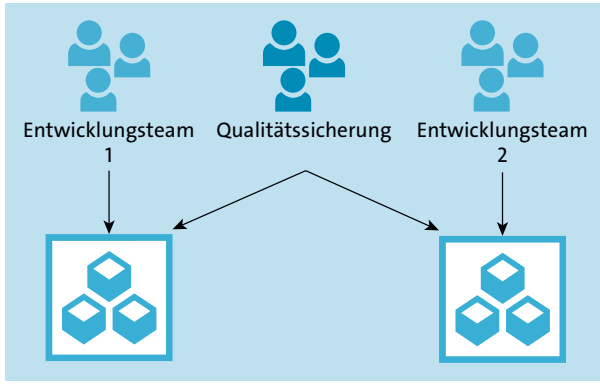

Abbildung 5.4 Das Qualitätssicherungsteam arbeitet in den gleichen Repositorys wie die Entwicklungsteams.

Das Ziel des Ganzen ist schließlich ein *Single Source of Truth* (kurz: SSOT): Es soll genau eine einzelne Quelle geben, in der alle relevanten Dateien für das Projekt liegen. Dazu gehören der Quellcode des Projekts selbst, die Tests und später dann auch noch Aspekte aus dem Betrieb der Anwendung.

Persönliches Exemplar für Thomas Bartholomäus

Der erste Vorteil wird direkt deutlich: Beide Teams haben Einblicke in die Arbeit des jeweils anderen Teams. Tests, die geschrieben werden, können direkt mit dem zurzeit in Entwicklung befindlichen Code zusammen geschrieben werden. Bisher sah es so aus, dass die Tests komplett getrennt voneinander entwickelt wurden, da kein Einblick möglich war. Das ist nun anders. Das vorherige Katz-und-Maus-Spielchen entfällt auf diese Weise mit einfachen Mitteln. Das wird auch tiefer in Kapitel 7 behandelt, wenn es um die Zusammenarbeit zwischen den Entwicklungsteams und dem Qualitätssicherungsteam geht.

Reflexion

Die Organisation ist das A und O für alle nachfolgenden Aspekte im DevOps-Software-Delivery-Lifecycle. Auch hier gilt, dass Visibilität essenziell ist, um Kollaboration zu fördern.

Ich kann nur dazu raten, die Transparenz zu verbessern. Der nachfolgende Schritt passiert idealerweise ziemlich natürlich: Die Teams erkennen die Synergien, wenn man die Projekte zusammenführt, um gemeinsam am selben Ziel zu arbeiten.

Der Knackpunkt ist auch hier – und das sah ich in der Praxis häufiger, als mir lieb ist –, dass es aus unbekannten Gründen eine Richtlinie gibt, die dagegenspricht, die Projekte zu öffnen. Je größer die Firma ist, desto schwieriger wird es, diese Richtlinie zu ändern. Je kleiner jedoch die Organisation ist, desto einfacher sollte es sein, die richtigen Personen zu überzeugen.

5.3 An Git führt nichts vorbei

Sie werden festgestellt haben, dass ich die Nutzung von Git als Versionsverwaltungsprogramm für (fast) alternativlos halte. Git ist ein modernes, verteiltes Versionsverwaltungsprogramm, das sich seit Jahren auf dem Markt durchgesetzt hat. Quasi alle, die in der IT arbeiten, sollten Git nicht nur kennen, sondern auch anwenden können. Je näher Sie an Dev- und Ops-Tätigkeiten dran sind, desto mehr müssen Sie mit dem Umgang vertraut sein.

Wenn es um die Vorteile von Git geht, wird oft das dahinterliegende, eher komplizierte Objektmodell hervorgehoben, das Git auf technischer Ebene im Vergleich zu älteren Versionsverwaltungsprogrammen effizient macht. Leider wird so erst mal die Lösung präsentiert, bevor überhaupt zunächst klar ist, welches Problem mit ihr gelöst werden soll. Denn sicherlich ist das effiziente Objektmodell von Git wichtig, aber es ist viel wichtiger zu betonen, dass Git für mehr Produktivität, Visibilität und Effizienz sorgt. Als verteiltes Versionsverwaltungssystem ermöglicht es das regelmäßige Erzeugen und Wieder-Zusammenführen von neuen Code-Branches. Das macht einen der wesentlichen Punkte aus Abschnitt 5.4, nämlich das Code-Review, viel einfacher.

© Rheinwerk Verlag, Bonn 2024

5.3.1 Git-Lösungen im Überblick

Wesentliche Plattformen für Sourcecode-Management sind GitHub, GitLab und mit abnehmender Wichtigkeit auch noch Bitbucket. Während hinter GitHub Microsoft steht, ist GitLab eine eigenständige Firma. Hinter Bitbucket steht, wie auch hinter Jira, der Konzern Atlassian. Während GitHub und Bitbucket Closed Source sind, ist GitLab Open Core mit einer großen Open-Source-Basis. Ergänzende Features, die vor allem für Firmen relevant sind, stehen unter einer propritären Lizenz.

Die komfortable Plattform GitHub dürfte einer der Hauptgründe sein, warum sich Git letztlich durchgesetzt hat. Zwar ermöglichen alle Git-Lösungen das schnelle, einfache und effiziente Anlegen von Branches, um parallele Entwicklungen voranzutreiben, aber erst mit GitHubs Pull Requests wurde dieser Workflow so komfortabel und einfach nutzbar, dass sich GitHub primär in Open-Source-Communitys durchgesetzt hat. Code-Reviews arbeiten mit *Merge Requests* bzw. *Pull Requests*. Die Namen sind zwar unterschiedlich, in der Praxis beschreiben beide Begriffe aber den gleichen Vorgang.

Was diese Funktion angeht, folgten GitLab und Bitbucket über die Jahre. Seitdem sind Pull Requests und Code-Review weitestgehend gesetzt und überall vorhanden. Hier wird ein konkretes Problem gelöst. Wie die genaue Implementierung aussieht, ist vernachlässigbar.

Weitere Tools

Abseits großer Projekte wird oftmals *Gitea* genutzt, was ein sehr leichtgewichtiges Sourcecode-Management-(SCM-)Tool ist. Es ist eine Abspaltung von *Gogs*. Von Gitea gibt es inzwischen eine weitere Abspaltung namens *Forgejo*.

Natürlich gibt es noch einige weitere Tools zum zentralen Sourcecode-Management. Allerdings sind diese weitaus kleiner und bringen wenig bis kaum Integration in alle anderen Stages des DevOps-Software-Development-Lifecycles. Zudem sind sie zumeist Open-Source-Projekte, die keinen professionellen Support anbieten können, was sie für Firmenkunden oftmals ausschließt.

Ja, es gibt auch noch andere Sourcecode-Management-Tools abseits von Git. *Subversion* findet man stellenweise immer noch, aber meistens eher aus historischen Gründen. *Mercurial* findet sich vereinzelt noch, ist aber quasi ausgestorben, seitdem sich das Ökosystem rund um Git so gut entwickelt hat.

Der Fokus dieses Abschnitts liegt also zwangsläufig auf GitHub, GitLab und am Rande auch auf Bitbucket. Viele Organisationen migrieren allerdings von Bitbucket weg. Ihr Ziel ist meistens GitLab oder GitHub, da Bitbucket wenig Weiterentwicklung erfährt und die Integration in den DevOps-Lifecycle bei den anderen beiden Plattformen besser und tiefer gestaltet ist.

Persönliches Exemplar für Thomas Bartholomäus

5.3.2 Entwicklungs-Workflows mit Git

Die Auswahl der richtigen DevOps-Plattform sollte Sie nicht auf leichte Schulter nehmen. In den nächsten Abschnitten werden Sie einige Aspekte kennenlernen, die Sie beachten sollten.

Grundsätzlich sollten Sie einen gescheiten Entwicklungs-Workflow implementieren. Das vorherige »Kraut und Rüben«-Development konnte und sollte nicht länger fortgeführt werden: zu viele Eigenbrötlereien, zu viele Verzögerungen, zu viele technische Schulden, die sich durch fehlende Prozesse aufgehäuft haben.

Grundsätzlich gibt es sehr viele verschiedene Entwicklungs-Workflows. Einige sind einfacher gehalten, andere sind komplex. Welcher Workflow die richtige Wahl ist, hängt davon ab, wie viele unterschiedliche Versionen und Varianten der Software gepflegt werden müssen. Für einen typische Webshop wie etwa *schick-gekleidet.de*, der in der Produktion nur aus einer Version besteht, die innerhalb der Firma betrieben wird, sind die Anforderungen recht gering. Hier bietet sich ein möglichst einfacher Git-Workflow an.

Komplexer wird es hingegen, wenn mehrere Versionen unterstützt werden müssen, was gar nicht so selten vorkommt. Und wenn dann noch die Software direkt beim Kunden vor Ort installiert wird, hat das noch viel mehr Auswirkungen auf den gesamten Entwicklungsprozess, insbesondere in Hinblick auf die Auslieferung neuer Versionen.

Workflows

Ein wichtiger Aspekt für gute Arbeit in der Software-Entwicklung ist der passende Git-Workflow. Dabei geht es darum, wie Änderungen am Code verwaltet werden. Die Frage ist immer, wie mit unterschiedlichen Versionen umgegangen wird: Welcher Code wird gerade produktiv genutzt, was wird gerade getestet, an was wird gerade herumgeschraubt? Diese verschiedenen Ebenen sollten nie vermischt werden – es ist ziemlich schlecht, wenn nicht getesteter Code, an dem noch gearbeitet wird, in die Produktion gelangt. Es ist nicht immer einfach, dort den Überblick zu bewahren; und die Verwaltung des Codes macht spätestens dann Probleme, wenn mehrere Leute gleichzeitig an ihm arbeiten.

Lösungen dafür gibt es zahlreiche! Einige (wie *Git Flow*) sind komplexer, andere (wie *GitHub Flow* oder *GitLab Flow*) sind deutlich einfacher gehalten. Über das Thema lässt sich sehr viel schreiben, aber beginnen wir mit einer kurzen Einführung, damit Sie nachvollziehen können, welcher Workflow mit welchen Vor- und Nachteilen im Alltag daherkommt.

Welche Workflows überhaupt relevant sind, hängt stark von den Projekten ab. Müssen beispielsweise viele Entwicklungslinien parallel gepflegt werden? Gibt es viele alte Versionen, die über eine lange Zeit aktualisiert werden und verfügbar sein müs-

© Rheinwerk Verlag, Bonn 2024

sen? Oder fließt schnell alles auf einen großen Hauptstrang zurück? Ein einfaches Release-Management spricht dafür, auch die Verwaltung des Quellcodes möglichst einfach zu lassen.

Das Gleiche gilt für das Deployment: Nicht jedes Projekt kann »echtes« CI/CD machen, da nicht jede Firma Anwendungen entwickelt, bei denen sowohl die Entwicklungsleistung als auch das Deployment komplett unter der eigenen Kontrolle steht. Wenn stattdessen »statischer« gearbeitet werden muss, also ein Paket gebaut wird, die Kunden es aber selbst bei sich ausrollen müssen, sind viele Git-Kunststücke weniger relevant. Insbesondere in Deutschland gibt es sehr viele Firmen, die Software an die Kunden ausliefern, die sie in eigenen Rechenzentren oder eben direkt auf Embedded Hardware, wie in Autos, ausrollen müssen.

Im Folgenden gehe ich auf die Workflows *GitHub Flow*, *GitLab Flow* und *Git Flow* ein, die in vielen Projekten im Einsatz sind. Vor allem Git Flow diente lange als »Best Practice«-Workflow, wenn es um die Einführung und Nutzung von Git ging. Ich bin davon aus diversen Gründen kein großer Fan und nutze ihn nur, wenn es sich nicht vermeiden lässt, aber dazu gleich mehr.

Wichtig ist, dass die Workflows für alle im Team einfach nachzuvollziehen und somit einfach anzuwenden sind. Die heutige IT-Welt ist schon komplex genug, sodass man auch hier nicht unnötige Komplexität aufbauen sollte. Konkret heißt das, dass man mit so wenigen langlebigen Branches wie möglich arbeiten und die Feature-Branches so kurz und klein wie möglich gestalten sollte. Denn nur so lassen sich die DevOps-Prinzipien für einen schnellen Flow von den Business-Anforderungen zum Kunden ermöglichen.

Trunk-based Development

Schauen wir uns zunächst den einfachsten Workflow an: das *Trunk-based Development*. Hierbei arbeitet das ganze Team direkt auf dem Hauptentwicklungsbranch. Die Idee ist, dass nicht mit Branches gearbeitet wird, sondern dass alle Änderungen direkt in den *Trunk* integriert werden – das ist die Hauptline, von der es keine Abweichungen gibt. Um sicherzustellen, dass das Projekt gebaut werden kann, wird vorher ein Pre-Integration-Build ausgeführt, damit nur baubare Ergänzungen integriert werden.

In größeren Projekten mit vielen Beteiligten skaliert das allerdings nicht, daher ist dieser Ansatz keine Alternative, wenn es ständig zu Merge-Konflikten kommt und Änderungen nicht integriert werden können.

Der Unterschied zum alten Wasserfall-Modell mit den Big-Bang-Integrationen ist im Wesentlichen, dass die Implementierung nicht vollständig allein läuft, sondern dass in der Regel im Rahmen von *Pair Programming* gearbeitet wird. Das heißt, mehrere Personen arbeiten gemeinsam an einer Implementierung, wodurch direkt ein Review stattfindet. Dabei wird auch lokal ein Build ausgeführt, sodass klar ist, dass das Projekt baut und mit den anderen geteilt werden kann.

Persönliches Exemplar für Thomas Bartholomäus

Obwohl es beim Trunk-based Development eigentlich keine Abzweigungen gibt, wird oft mit kurzlebigen Branches gearbeitet, um den Bau auch auf dem CI-Server zu verifizieren. Dies ist besonders dann hilfreich, wenn kein Pair Programming durchgeführt werden kann. Genauere Details zum Trunk-based Development finden sich auf der Webseite *trunkbaseddevelopment.com*.

GitHub Flow

Trunk-based Development eignet sich nicht mehr, wenn mehrere Leute gleichzeitig an unterschiedlichen Features arbeiten und diese in den Main-Branch integrieren müssen. Dazu werden Feature-Branches benötigt, wie sie GitHub Flow bietet. Es gibt dann lediglich einen langlebigen Branch `main` (früher: `master`). Wenn neue Features entwickelt oder Bugs gefixt werden, dann wird auf Basis dieses Branches ein kurzlebiger Feature-Branch erstellt, der nach einem Code-Review direkt in den `main` Branch zurückfließt (siehe Abbildung 5.5). Anschließend sollen die Änderungen mithilfe der konfigurierten Pipeline direkt auf die Produktivsysteme ausgerollt werden. Es werden also immer kurzlebige Branches erzeugt, die möglichst zeitnah in den Hauptentwicklungsbranch gemergt werden. Das bedeutet, dass die Änderungen möglichst klein gehalten werden sollten.

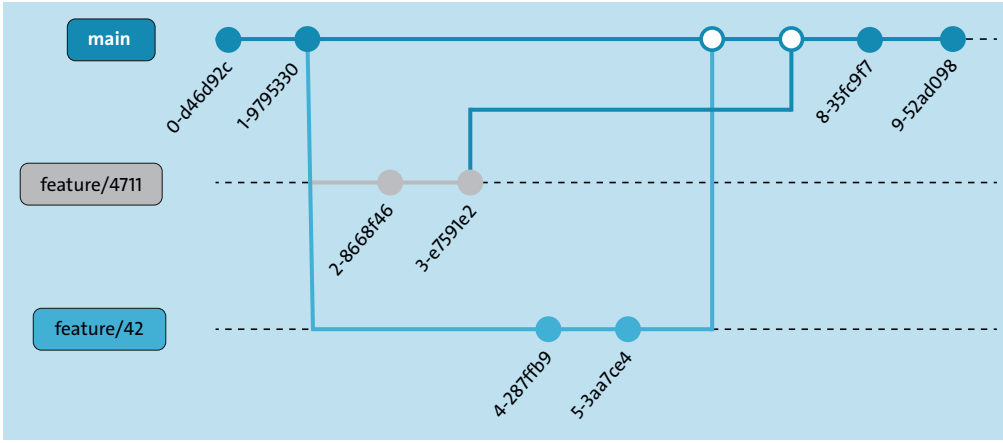

Abbildung 5.5 Idealerweise gibt es nur einen langlebigen Entwicklungsbranch (hier: »main«) und kurzlebige Feature-Branches, um Integrationsschmerzen zu verringern.

Die genauen Anweisungen zur tatsächlichen Umsetzung des Workflows finden Sie in der Dokumentation von GitHub unter *https://docs.github.com/en/get-started/quickstart/github-flow*.

Wie einfach der Workflow ist, zeigt sich schon daran, dass er mit nur wenigen Befehlen umgesetzt werden kann.

© Rheinwerk Verlag, Bonn 2024

Zunächst muss sichergestellt werden, dass der Branch `main` auf dem aktuellen Stand ist. Dafür wechseln Sie zunächst auf den Branch, um dann mit einem Pull die Änderung lokal verfügbar zu machen:

```
$ git switch main
$ git pull
```

Anschließend muss nur noch der neue Branch angelegt und müssen die Änderungen durchgeführt werden. Hier erstellen wir ein neues Feature mit dem Namen `foobar`:

```
$ git switch -c feature/foobar
```

Zum Schluss muss nur noch der Branch gepusht werden, sodass ein Code-Review angestoßen werden kann. Zu Code-Reviews folgt in Kürze ein tieferer Einblick.

```
$ git push origin feature/foobar
```

Und das war auch schon der Workflow! Das funktioniert zwar gut für kleine Webanwendungen, die man selbst betreut, aber für alles, was darüber hinaus geht, ist er *zu* einfach. So gibt es nicht mal eine Staging-Umgebung, in die Testversionen deployt werden können.

GitLab Flow

Diese Einschränkungen geht GitLab Flow an und macht da weiter, wo GitHub Flow aufgehört hat. Eine Annahme von GitHub Flow war schließlich, dass immer auf Produktivsysteme deployt werden kann. Daher wird in GitLab Flow ein weiterer langlebiger Branch namens `production` eingeführt, der immer den Stand der Produktionsumgebung widerspiegelt.

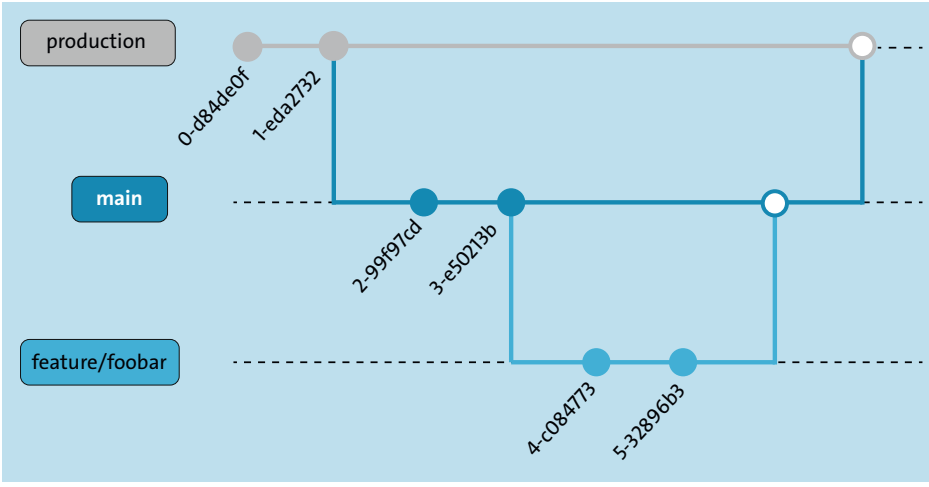

Abbildung 5.6 GitLab-Flow erweitert GitHub-Flow um den »production«-Branch.

Die üblichen Entwicklungsarbeiten finden auch hier in kurzlebigen Feature-Branches statt, die nach einem Code-Review in den `main`-Zweig gemergt werden. Der wesentliche Unterschied ist, dass nicht nach jedem Merge dorthin automatisch ein Deployment auf die Produktionsumgebung erfolgt. Wenn ein Deployment erfolgen soll, dann wird explizit von `main` nach `production` gemergt. So erlangen Sie eine stärkere Kontrolle über den Status des Deployments.

Aber auch hier kann es Einschränkungen geben, etwa dann, wenn eine Staging-Umgebung benötigt wird, in der nochmals separat diverse Tests durchgeführt werden, bevor der Code auf die Produktivumgebung ausgerollt wird. Entsprechend können Sie zwischen `main` und `production` einen `pre-production`-Branch einfügen, der als Zwischenschritt dient, um stärkere Kontrolle für Staging- und Produktions-Deployments zu haben.

Aber auch für Release-Branches gibt es eine Lösung. Dies ist insbesondere dann relevant, wenn Software an Kunden ausgeliefert wird, die diese selbst ausrollen müssen. In diesem Fall werden nicht zwangsläufig `production`- oder `pre-production`-Branches benötigt, sondern Sie erzeugen auf Basis des `main`-Branchs sogenannte Release-Branches à la `1.1-stable`, `1.2-stable` und so weiter. Die üblichen Entwicklungsarbeiten laufen weiterhin gegen den `main`-Branch, auf dessen Basis die Release-Branches erzeugt werden. Bei Bugfixes für ältere Releases kann entsprechend mit Cherry-Picks gearbeitet werden.

Eine vollständige Information über GitLab Flow und die genauen Befehle und Anweisungen finden Sie in der GitLab Dokumentation:

https://docs.gitlab.com/ee/topics/gitlab_flow.html#introduction-to-gitlab-flow

Git Flow

2010 veröffentlichte Vincent Driessen auf seinem Blog eine Anleitung zu Git Flow: *https://nvie.com/posts/a-successful-git-branching-model/*

Das war in den Anfangstagen von Git, als noch nicht ganz klar war, wie man die nützlichen Funktionen in einer konsistenten Form nutzen kann. Herausgekommen ist dabei Git Flow, das sich schnell durchgesetzt hat.

Wenn Sie den Blog-Eintrag jedoch heute besuchen, sehen Sie ein Update aus dem Jahr 2020, in dem Vincent Driessen schreibt, dass er Git Flow nicht mehr als optimal ansieht, da es häufig viel zu komplex ist. Es müssen viele langlebige Branches erzeugt und verwaltet werden, die dann auch in die richtigen Entwicklungsbranches gemergt werden müssen.

Wie Abbildung 5.7 zeigt, gibt es bei Git Flow sehr, sehr viele Branches. Sie sehen einige langlebige Zweige wie `develop`, in den die Entwicklung mit kurzlebigen Feature-Branches fließt, sowie den `master`-Branch, was der letzten veröffentlichten Version entspricht. Dazwischen gibt es aber noch diverse Release-Branches sowie Hotfix-

© Rheinwerk Verlag, Bonn 2024

Branches, wenn in die verschiedenen Versionen Korrekturen eingepflegt werden müssen. Dazu müssen die Branches auch noch in die richtigen Branches gemergt werden, damit Fehler nicht nur in älteren, sondern auch in der jeweils aktuellen Entwicklungslinie korrigiert werden.

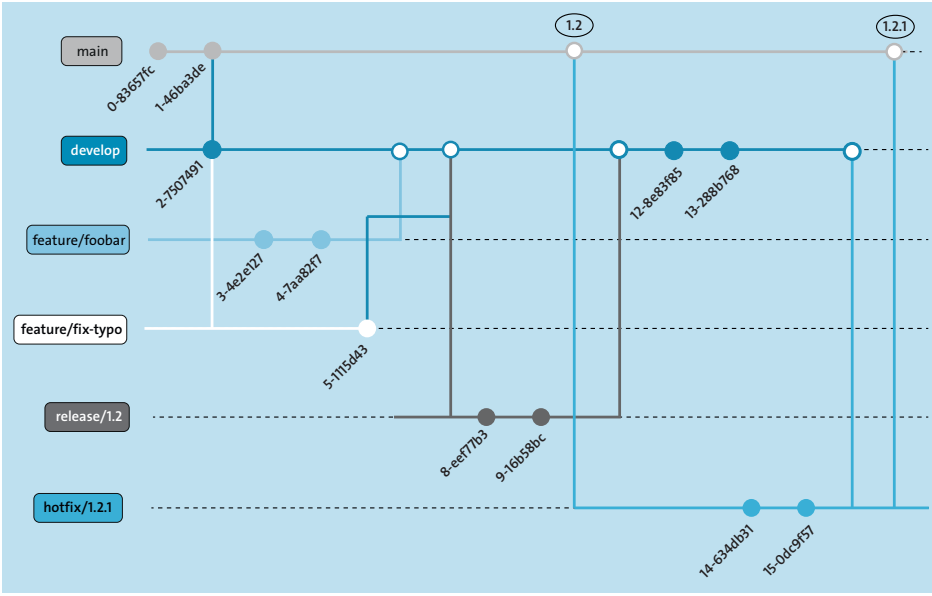

Abbildung 5.7 Git Flow ist, wie hier unschwer zu erkennen ist, ziemlich komplex, aber weit verbreitet.

Ich gehe hier bewusst nicht auf alle Details ein, da dies den Rahmen dieser Einführung sprengen würde. Wenn Sie mehr über diesen Workflow erfahren möchten, finden Sie alle Informationen im Blogpost.

Heutzutage setzen viele Firmen auf einen Workflow, der nah am *Git Flow* ist, aber einige Punkte abwandelt: Es wird mit mehreren langlebigen Branches gearbeitet sowie mit Feature- und Release-Branches. Die Feature-Branches werden nach einem Code-Review in den Hauptentwicklungsbranch gemergt, der meist `develop` genannt wird. Wenn ein Release ansteht, dann wird basierend auf dem Develop-Branch ein Release-Branch erzeugt, auf dem die letzten Korrekturen durchgeführt werden, bevor der Code dann zum `main`- oder `master`-Branch gemergt wird sowie zurück in den `develop`-Branch.

Für komplexe Setups, in denen viele verschiedene Versionsstränge gleichzeitig gepflegt und entwickelt werden müssen, ist die Komplexität von Git Flow notwendig. Das gilt beispielsweise für viele Projekte in der Industrie. Der Hauptnachteil ist allerdings, dass dieser Weg die Arbeit sehr kompliziert macht, da oft Branches abgezweigt und in die richtigen Ursprungsbranches zurückgeführt werden müssen, wobei Sie

höllisch aufpassen müssen, dass Sie nichts vergessen oder verwechseln. Wenn es die Anforderungen des Projekts erlauben, sollte man daher lieber einen einfachen Workflow verwenden, damit sich auch die Code-Reviews einfach und effizient gestalten.

Viel wichtiger als die eigentliche Umsetzung des Entwicklungs-Workflows ist allerdings, sich anzuschauen, welche Vorteile sich unter DevOps-Gesichtspunkten ergeben. Mit allen Methoden, wenn sie ordentlich umgesetzt werden, wird sich die Qualität der Software verbessern. Fehlerhafte und doppelte Implementierungen jeglicher Art fallen tendenziell früher auf.

Zudem kann voneinander gelernt werden, was langfristig ebenfalls zu verbesserter Qualität und Produktivitätssteigerung beiträgt. Mehr hierzu folgt im nächsten Abschnitt 5.4, wo es sowohl um Code-Review als auch um Pair Programming geht.

Ein weiterer wichtiger Punkt in diesem Kontext ist zudem, dass andere Personen aus fremden Teams zum Code beitragen können. Das können auch Mitglieder des Ops-Teams sein, die bei passender Fachkenntnis einige Fehler auch direkt im Sourcecode nachvollziehen und vielleicht sogar bei eins korrigieren können. Das ist insbesondere hilfreich, um mögliche Workarounds auf Betriebssicht zu vermeiden, wenn direkt die Ursache korrigiert werden kann, anstatt mühselig Symptome zu bekämpfen. Das ist Thema von Abschnitt 5.5, wo es um Inner Sourcing geht.

Reflexion

Ein klar definierter Workflow ist wichtig, damit alle beteiligten Personen wissen, wie entwickelt wird, um ineffizienten Wildwuchs zu vermeiden. Praktischerweise ist das ein Teil, den auch die Teams für sich festlegen können.

Je einfacher der Workflow ist, desto besser für das Team. Nichts ist nerviger als ein unnötig komplizierter Arbeitsablauf. Ein einfacher Workflow ist zudem eine gute und wichtige Ausgangslage für Code-Reviews und CI/CD.

5.3.3 Das Sourcecode-Management bei schick-gekleidet.de

Für *schick-gekleidet.de* hieß es auch beim Sourcecode-Management, die nötigen Modernisierungen durchzuführen. Allen an der Website-Entwicklung Beteiligten war es wichtig, ein Code-Review stringent einzuführen, was ohne hohe Komplexitäten durchgeführt werden sollte.

Mit Subversion ging das nicht wirklich gut. Der Wechsel auf das weitverbreitete Git war daher naheliegend, da im Wesentlichen GitLab und GitHub bekannt waren. Der Workflow sollte möglichst einfach sein, entsprechend lag der Einsatz von GitLab Flow nahe, da Git Flow für die Zwecke von *schick-gekleidet.de* zu kompliziert und GitHub Flow wiederum zu einfach ist.

© Rheinwerk Verlag, Bonn 2024

Reflexion

Wenn Sie noch keinen gescheiten Entwicklungs-Workflow nutzen, dann kann ich Ihnen nur empfehlen, zeitnah daran zu arbeiten. Wenn Sie einen nutzen, der aber viel zu komplex ist, etwa Git Flow, dann schauen Sie, ob es nicht besser wäre, ihn zu vereinfachen. Unnötig komplexe Arbeitsabläufe sind zu vermeiden.

Dazu gehören auch kleine Nebenaspekte, auf die ich nicht eingegangen bin. Es gibt viele Fälle, in denen jedes Mitglied im Team mit einem eigenen Fork des Repositorys arbeitet und von dort aus einen Pull bzw. Merge Request aufmacht. Die Verwaltung mehrerer Repositorys ist meist unnötig, sodass selbst in großen Teams lieber jeder mit kurzlebigen Branches auf dem zentralen Repository arbeiten sollte. Den Rest regeln die Einstellungen am Repository.

Vergessen Sie zudem nicht, dass das Sourcecode-Management und die Prozesse herum starken Einfluss auf den CI/CD-Server haben. Idealerweise kommen Sie mit möglichst wenigen Tools aus.

5.4 Code-Reviews und Pair Programming

In den vorherigen Abschnitten wurden schon kurz einige Vorteile und Nachteile von Code-Reviews und Pair Programming diskutiert. Damit wird man der Bedeutung dieser absolut zentralen Bestandteile eines guten Entwicklungs-Workflows aber nicht gerecht.

5.4.1 Code-Reviews

Die Durchführung von Code-Reviews in Teams, sowohl für Development- als auch für Operations-Tätigkeiten, ist grundsätzlich sinnvoll. Wenn dieser Austausch fehlt, weil im Team nicht diskutiert wird und alle vor sich hin arbeiten, leiden die Produktivität und die Produktqualität. Arbeiten werden gegebenenfalls doppelt und dreifach durchgeführt, Änderungen sind inkompatibel und sorgen für Mehraufwand, und es schleichen sich Fehler ein, weil Voraussetzungen und Arbeitsschritte nicht gut miteinander abgesprochen werden.

Ein einfacher Weg, um die Kollaboration, die Kenntnisse im Team wie auch die Code-Qualität zu verbessern, sind *Code-Reviews*. Die Grundidee dahinter ist, dass jede Codezeile, die in den Hauptentwicklungszweig des Projektes eingefügt werden soll, von mindestens einer weiteren Person überprüft wird. Das lohnt sich sowohl bei Projekten mit hoher als auch mit niedriger Qualität, da so idealerweise immer nur Verbesserungen an der Qualität des Codes hinzugefügt werden. Code-Review erleichtert letztlich die Kommunikation am Code und fördert zudem eine gemeinsame Entwicklungskultur.

Persönliches Exemplar für Thomas Bartholomäus

Die Ausgangslage muss dabei gar nicht so schlimm wie in dem XKCD-Comic sein, den Sie in Abbildung 5.8 sehen. Code-Reviews sorgen natürlich auch dafür, dass wirklich schlimmer Code herausgefiltert wird, sie verbessern aber auch den Input von erfahrenen Teammitgliedern. Erfahrungsgemäß gibt es immer noch etwas zu lernen und man erfährt Details, die man nicht wusste.

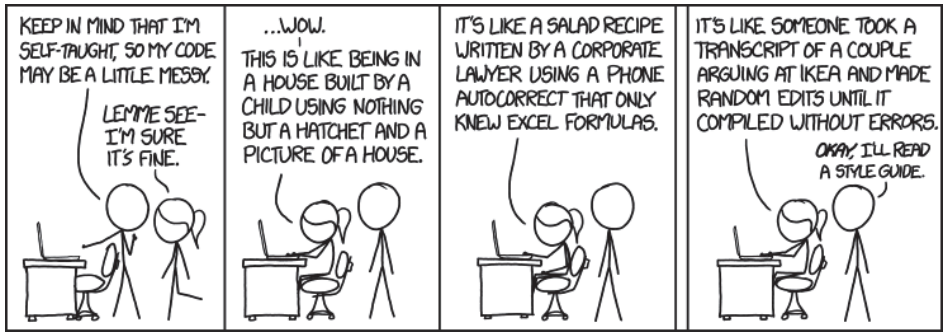

Abbildung 5.8 Code-Review in XKCD (Quelle: https://xkcd.com/1513/)

Bei Open-Source-Projekten ist Code-Review schon lange geläufig. Bei Firmen und anderen Organisationen sehe ich leider immer wieder, dass dies gar kein so üblicher Standard ist. Der häufigste Grund, den ich höre, ist: »Wir haben so viel zu tun, dafür haben wir gar keine Zeit!«, gefolgt von: »Aber davon haben unsere Kunden gar nichts, außer dass es länger dauert!«

Beide Begründungen halte ich, abgesehen von wenigen Ausnahmen, für vorgeschobene Gründe. Wie schon zu Beginn dieses Buches erwähnt, muss DevOps in der gesamten Organisation gelebt werden, und entsprechend auch von den Personen ganz oben in der Hierarchie. Dazu gehört dann auch, dass Code-Review stringent durchgeführt wird, was am Anfang ein wenig Zeit kostet, sofern man es bislang nicht gewohnt ist.

Diese Zeit sollte man sich nehmen, um etwaige Probleme in Zukunft schon frühzeitig zu erkennen, anstatt dass diese im Laufe der Zeit zu hohen technischen Schulden führen, wodurch man letztlich noch weniger Zeit für Code-Review hat. Und gute Code-Qualität führt letztlich auch zu einem besseren Produkt für den Kunden, wenn es weniger Bugs ins fertige Produkt »schaffen« und Fehler zügiger korrigiert werden können, da weniger Altlasten mitgeschleppt werden.

Checkliste für gute Code-Reviews

1. Halte die Änderungen möglichst klein, um Reviews effizient zu halten.
2. Beschreiben Sie nicht nur, *was* Sie geändert haben, sondern auch *warum*. Bonuspunkte gibt es, wenn Sie zusätzlich beschreiben, warum andere Lösungen *nicht* praktikabel waren.

© Rheinwerk Verlag, Bonn 2024

3. Schreiben Sie aussagekräftige Commit-Messages nach demselben Konzept wie in Punkt 2.
4. Sorgen Sie dafür, dass die Änderungen stets baubar sind und führen Sie, wenn nötig, ein Rebase durch.

Code-Review innerhalb des Entwicklungsteams

Damit der ganze Prozess ein wenig klarer wird, lohnt es sich, ein Beispiel anzuschauen.

Eine Entwicklerin hat erst vor wenigen Wochen bei *schick-gekleidet.de* angefangen und arbeitet an ihrem ersten Arbeitspaket. Sie kennt sich also mit der veralteten Code-Basis nicht sonderlich gut aus und erkennt, dass ihre Aufgabe möglicherweise nicht optimal gelöst ist. Sie ist sich daher nicht sicher, ob es mit dem vorhandenen Code nicht doch einen einfacheren Weg gibt.

Ohne ein Code-Review hätte sie den Code aber wohl doch einfach eingecheckt: Hauptsache, es funktioniert. Was will man sonst machen?

Nun, ganz einfach: um Hilfe fragen.

Um sicherzustellen, dass sowohl die Qualität des Codes stimmt als auch die Implementierung effizient ist, bittet sie eine Senior-Entwicklerin um ein Review. Die Kollegin ist schon länger dabei, kennt den Code viel besser und weiß auch, wo sich einige Fallstricke befinden, die zu Problemen führen können.

Die Senior-Entwicklerin gibt ihrer neuen Kollegin daher ein paar Hinweise, was sie noch anders machen könnte, um den Code zu optimieren. Und siehe da: Es fallen direkt mögliche Fehler auf, die später zu Problemen führen könnten, wenn andere Teile des Codes nicht direkt im Rahmen eines Refactorings ebenfalls überarbeitet werden (siehe Abbildung 5.9).

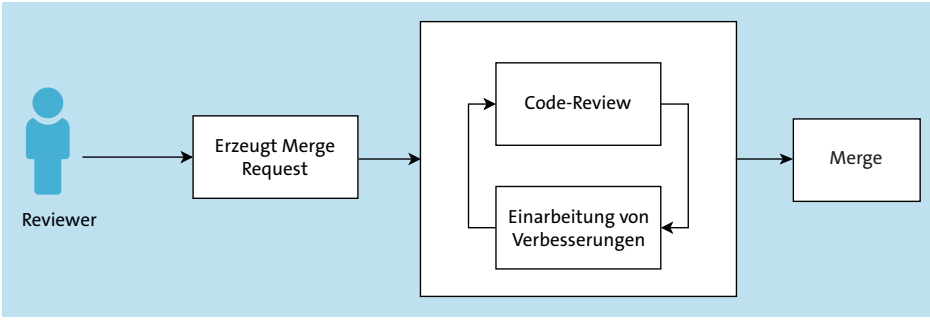

Abbildung 5.9 Der vereinfachte Prozess des Code-Reviews

Der Vorteil des Code-Reviews wird deutlich sichtbar: Mögliche Probleme werden schon frühzeitig erkannt und behoben. Der wichtige Punkt dabei: Wenn das Review durchgeführt wird, befindet sich die Entwicklerin gerade mental in dem Kontext der Änderung; sie weiß also, warum sie das Problem so gelöst hat. Einige Wochen später wäre sie da schon »komplett raus« und müsste sich erst wieder in die spezifische Frage einarbeiten. Die Gefahr, dass dabei etwas vergessen wird, ist groß.

Es gibt aber noch einen weiteren Vorteil: Die Entwicklerin lernt die Code-Basis deutlich besser und schneller kennen und lernt so, was sie bei der nächsten Implementierung beachten sollte. Aber auch für die Senior-Entwicklerin, die das Review durchführt, gibt es einiges zu lernen: Sie sieht andere Blickwinkel und stolpert über Probleme, die nur Leute mit frischen Augen erkennen. Somit kann auch sie besser zukünftige Herausforderungen frühzeitig für ihre eigenen Implementierungsarbeiten erkennen.

Das Argument »Das dauert doch viel zu lange!« zieht hier kaum noch: Bisher erfolgte das Ausrollen der Änderung ohnehin nur alle sechs Monate und führte dann regelmäßig zu Problemen. Diese Zeit wird eingespart, wenn es gelingt, mögliche Fehlerquellen schon früher zu beheben. Und langfristig wird das Projekt profitieren, wenn die Code-Qualität verbessert wird und mögliche Fehlerquellen schrittweise entschärft werden.

Diese internen Reviews innerhalb des Entwicklungsteams werden Sie sicherlich kennen. Noch wertvoller kann das Review werden, wenn dabei auch Kollegen aus anderen Teams hinzugezogen werden, die andere Kompetenzen mitbringen. Wenn Dev und Ops näher zusammenrücken, sind Code-Reviews ein einfaches Mittel, um Feedback zwischen den verschiedenen Aufgabengebieten abzuholen.

Code-Review zwischen Dev und Ops

Es geht bei DevOps nicht darum, dass kein Betriebsteam gebraucht wird. Noch machen sowohl die bisherigen Devs als auch die Ops beides gleichermaßen. Die Wahrheit liegt irgendwo in der Mitte: Personen mit Entwicklungstätigkeiten übernehmen nun mehr Verantwortung beim Betrieb der Produkte. Und wer vorher nur auf der Betriebsseite unterwegs war, lernt die Anwendungen genauer kennen. Und das eben in einem Team.

Da bisher komplett getrennt gearbeitet wurde, fehlte nicht nur der Einblick in den Code des jeweils anderen Teams, sondern es konnten auch keine Änderungen beigetragen werden, die jeweils im eigenen Kenntnisbereich lagen.

Bei *schick-gekleidet.de* war beispielsweise die Zugriffsgeschwindigkeit immer ein Problem – wie quasi bei jeder komplexen Webseite. Es wurde also die Anforderung formuliert, die Ladezeiten des Onlineshops zu verringern. Metriken lagen bis dato allerdings nicht vor, sodass das Problem nicht besser quantifiziert werden konnte.

© Rheinwerk Verlag, Bonn 2024

(Das Messen von Metriken ist zwar ein Aspekt vom CALMS-Modell, aber darauf gehe ich an dieser Stelle noch nicht näher ein.)

Im Rahmen der Entwicklungsarbeiten kam man zu dem Schluss, dass ein Caching-Server benötigt wird, der viele Daten zwischenspeichert, damit Inhalte zügiger abgerufen und ausgeliefert werden können. Um das umzusetzen, wurde vom Betriebsteam die Bereitstellung des Caching-Servers gefordert. Dazu muss das Betriebsteam die Infrastruktur bereitstellen und benötigt klare Anweisungen, wie der Server konfiguriert werden soll.

Bei der Implementierungsarbeit auf der Seite des Entwicklungsteams gab es allerdings immer wieder ein paar Anpassungen, die »eben noch schnell« gemacht werden sollten. Bisher gab es also keine vielversprechende Möglichkeit, gemeinsam am selben Code zu arbeiten.

Das Betriebsteam hatten keinen Einblick in den Code und konnte nicht die Konfigurationsparameter einsehen, die von Entwicklungsseite aus festgelegt werden mussten. Der Austausch funktionierte nicht; man konnte einander nicht helfen. Es dauerte viel zu lange, bis die Änderungen durch waren. Und auch dann war das Endergebnis immer verbesserungswürdig.

Durch die Öffnung des Codes, die Zusammenführung der Teams und die Einführung von Code-Review konnte dieser Ablauf deutlich verbessert werden: Statt des Ping-Pongs mit sehr viel Trial-and-Error konnte die Entwicklungsarbeit deutlich beschleunigt werden, da nun das Betriebsteam die nötigen Parameter kennt und Konfigurationen direkt im Review-Prozess gemeinsam bearbeitet werden.

Das betrifft nicht nur die Änderungen am Sourcecode des Onlineshops, sondern auch die Konfigurationsänderungen an der Server-Infrastruktur, wo beide Teams gegenseitig Änderungen vorschlagen können.

Code-Review zwischen Dev und QA

Bei DevOps geht es nicht nur darum, Dev und Ops zusammenzubringen, sondern alle Abteilungen zu integrieren, die am Produkt arbeiten. Zu diesen gehört auch das Qualitätssicherungsteam. Auch hier hilft es, wenn das Review zwischen den verschiedenen Tätigkeitsschwerpunkten stattfindet, etwa dann, wenn das Entwicklungsteam die Struktur der Tests umbauen muss und dabei den nötigen Input von denjenigen Personen bekommt, die sich hauptsächlich um die Qualitätssicherung kümmern.

Es gilt aber auch der umgekehrte Fall: Aus QA-Sicht ist ein Review durch den Entwickler immer wieder nützlich, um sicherzustellen, dass die Änderungen sinnvoll sind, etwa dann, wenn das Schreiben von Tests verbessert werden kann oder andere Testfälle abgedeckt werden müssen.

Persönliches Exemplar für Thomas Bartholomäus

Reviews beschleunigen

Ein Kernaspekt der DevOps-Idee ist, dass durch regelmäßiges Feedback und ebenso regelmäßige inkrementelle Aktualisierungen des auszuliefernden Produkts agiler gearbeitet werden kann. Damit ein Review nicht ewig dauert, sind ein paar Spielregeln zu beachten, die das Ganze für alle angenehmer machen. Schließlich sollen die Änderungen auch zügig einführt werden, und es soll eben nicht ewig dauern, bis alle Änderungsvorschläge zu Tode diskutiert wurden. Sonst hat man schließlich nichts gewonnen.

- **Liegen bleibende Reviews vermeiden**

  Ein häufiges Problem bei der Einführung von Reviews ist, dass diese häufiger mal liegen bleiben. Manchmal ein paar Tage, manchmal sogar Wochen. Grundsätzlich ist es im Interesse des Developers, dass die eigene Arbeit zügig abgeschlossen wird. Falls also niemand, wie angefordert, ein Review macht, sollte man hier nachhaken und ggf. eine andere Person zum Reviewen suchen. Aber auch hier gibt es Tool-Unterstützung, die das Ganze vereinfacht.

  Besser ist es, wenn es feste Termine und geregelte Abläufe gibt, die den Review-Prozess organisieren. Man sollte nie lange warten müssen, bis eine fertige Änderung geprüft wird.

- **Keine Schuldzuweisungen!**

  Absolut zentral für den Erfolg von Code-Reviews ist, dass es keine Schuldzuweisungen bei Problemen geben darf. Nur weil jemand den Code geschrieben hat, der zu Problemen führt, ist er oder sie nicht automatisch schuld, wenn es Probleme gibt. Und nur weil jemand den Code reviewt hat und den Fehler hätte finden sollten, ist auch diese Person nicht schuld, wenn der Fehler in die Produktion gelangt: Schließlich hätten ja zwei Personen den Fehler finden können.

  Sorgen Sie also für eine produktive Stimmung in den Reviews! Niemand sollte sich davor fürchten. Es sollte aber auch klar sein, dass kein offensichtlich schlechter und unfertiger Code eingereicht werden darf. Das Code-Review ist keine zusätzliche Instanz, in der Probleme gelöst werden, für die man selbst keine Antwort gefunden hat. Diese Schritte sollten vorher erledigt werden, denn sie blockieren und verkomplizieren sonst den Review-Prozess.

- **Überschaubare Änderungen – Keep it small!**

  Damit die Reviews effektiv ablaufen, sollten die Änderungen möglichst überschaubar sein. Je kleiner die Änderungen für ein Review sind, desto schneller und zügiger können und werden diese Änderungen durchgesehen. Das Ganze ist nicht weiter überraschend: Grundsätzlich sind Menschen faul. Je einfacher es einer Person gemacht wird, ein Review durchzuführen, desto eher wird das auch erledigt. Ein Review von einer Änderung mit fünf Zeilen ist tendenziell schneller gemacht als eine Änderung mit mehreren Tausend Zeilen, was nun mal einige Zeit in

© Rheinwerk Verlag, Bonn 2024

Anspruch nimmt – falls eine solche Änderung überhaupt in einem Review sinnvoll geprüft werden kann.

Der Vorteil von kleineren Review-Häppchen ist auch, dass viel häufiger integriert und ausgerollt werden kann und dass es so tendenziell zu weniger gravierenden Problemen kommt, da große Big-Bang-Integrationen vermieden werden. Und damit sind nicht nur Probleme auf Produktivsystemen gemeint, sondern auch mögliche Konflikte, wenn mehrere Personen an den gleichen oder an überlappenden Code-Zeilen arbeiten. Je schneller also etwas integriert wird, desto weniger Schmerz leidet der implementierende Entwickler wie auch das Team als Ganzes.

- **Reviewer-Roulette**

  Im Sinne der Kollaboration und des Wissenstransfers ist es wichtig, dass nicht immer die gleichen Leute den gleichen Teil des Codes verifizieren. Das mag in einigen Fällen zwar nützlich sein, wenn komplexe Sachverhalte besser von den Experten begutachtet werden, aber um weitere Experten aufzubauen, sollte eine gute Mischung stattfinden.

  Häufig sehe ich, dass immer dieselben Leute ihre liebsten Kolleginnen und Kollegen zum Review bitten, weil diese kaum Kommentare abliefern und den Code mit wenig Aufwand durchwinken. Dann hat man allerdings nicht nur nichts gewonnen, sondern eigentlich nur neue Nachteile in den Prozess implementiert.

- **Konstruktiv und hilfreich**

  Die Kommentare im Review sollten sinnvoll und nachvollziehbar sein, unabhängig vom Kenntnisstand des Urhebers. Kommentare wie »Das kann man zwar so machen, ist dann halt Kacke« mögen zwar manchmal richtig sein, helfen aber nicht weiter: Aus ihnen geht nicht hervor, was das Problem ist, und ein Vorschlag, wie es zu beheben ist, fehlt auch.

  Der eingesetzte Ton in der Sprache ist also wichtig: Bitte keine Vorwürfe, keine Verwendung von Sprache, die die andere Person als »dumm« darstellt. Zu Letzterem gehören auch Sprüche wie »Das ist doch ganz einfach!« oder das Wörtchen »offensichtlich«. Alles, was auf andere herablassend wirken könnte, ist nicht förderlich für eine harmonische Zusammenarbeit.

Zusammenfassend lässt sich also sagen, dass sich durch die Einführung von Code-Review ein einfaches Mittel nutzen lässt, um die Kollaboration zwischen den einzelnen Aufgabengebieten zu vereinfachen, damit die Teams zusammenrücken können und das Wissen geteilt wird. Also ganz im Sinne des »Sharings« im CALMS-Modell.

Das verbessert die Abläufe im Team, sorgt für eine höhere Produktqualität und vereinfacht den Support innerhalb des Teams, wenn die Anwendung ausgerollt wird. Und nicht zuletzt sorgen Code-Reviews für eine bessere Kultur im Projekt: Durch gescheites Reviewing werden quasi automatisch ein gemeinsames Commitment und Ownership erzeugt, da man nicht nur in der »eigenen Ecke« des Codes unterwegs ist.

Persönliches Exemplar für Thomas Bartholomäus

5.4.2 Code-Reviews vereinfachen

Einer der wichtigsten Gründe für Code-Reviews ist das sehr frühe Feedback für den Entwickler oder die Entwicklerin, die die Änderung einbringen möchten. Diese Rückmeldung besteht zum einen aus dem Feedback einer anderen Person aus dem Maintainer-Team, zum anderen aber aus den automatisch ablaufenden Checks. Diese automatischen Tests werden als Teil der Pipeline ausgeführt, sie sollten also als Teil des Code-Reviews automatisch ablaufen, ohne dass manuelle Eingriffe notwendig sind, um sie für einen Feature-Branch zu starten. Im Code-Review sollte direkt sichtbar sein, ob der Durchlauf der Pipeline erfolgreich war oder nicht.

Idealerweise geht man noch einen Schritt weiter: Es sollte mit einfachen Mitteln sofort ersichtlich sein, welche Auswirkungen die vorgenommenen Änderungen auf das Projekt haben. Dazu gehört z. B., ob sich die Qualität des Codes, die durch standardisierte Qualitätsmerkmale definiert ist, verändert hat. Je nach Anwendung kann es auch sinnvoll sein, schnelle Last- oder Performancetests durchzuführen, um zu sehen, ob die Änderungen zu kleineren oder größeren Performanceänderungen führen.

Diese Tests sollten immer möglichst schnell ablaufen, weil lange Wartezeiten kontraproduktiv sind. Idealerweise unterstützt die Plattform dann als Teil des Reviews, dass diverse Quality-Gates definiert werden, um einen Merge zu blockieren. Das ist insbesondere für Sicherheitslücken relevant, sodass stets entschieden werden kann, ob man wegen neu aufgetretenen Sicherheitslücken das Deployment auf Produktivumgebungen blocken sollte.

Durch die Sichtbarkeit der Pipeline im Code-Review-Prozess ist es für alle direkt ersichtlich, ob das Bauen der Software und die diversen Tests erfolgreich durchgelaufen sind, bevor jemand mit dem Review beginnt.

Für das Review selbst gibt es ebenfalls diverse kleinere Features, die den Alltag vereinfachen. So können, je nach eingesetzter Plattform, im Review-Prozess direkt Ergänzungsvorschläge an einzelne Zeilen notiert werden, um Fehler zu korrigieren oder andere Optimierungen direkt vorzuschlagen. Der Autor kann dann diese Änderungen im Gegenzug reviewen und direkt annehmen, sodass diese Teil des Merge bzw. Pull Requests werden (siehe Abbildung 5.10).

Dieses Feature ist relativ unscheinbar, aber sehr hilfreich, da es den Review-Prozess erheblich beschleunigen kann. Eine gute Plattform bietet ohnehin Metriken an, die zeigen, wie schnell Reviews durchgeführt werden. Je kleiner die Änderungen sind, desto schneller sollte es grundsätzlich gehen, da die Hürde kleiner ist, was auch ganz dem Ziel entspricht, Änderungen schnell integrieren und ausrollen zu können, selbst wenn sie noch nicht voll funktionsfähig sind.

Bitte vergessen Sie aber nicht, das ganze System zu betrachten, wenn Sie eine Teillösung evaluieren. So auch hier! Das beste System für das Sourcecode-Management

© Rheinwerk Verlag, Bonn 2024

nützt nicht viel, wenn zu viel Zeit und Mühe in halbherzige Integrationen mit den anderen Tools gesteckt werden muss.

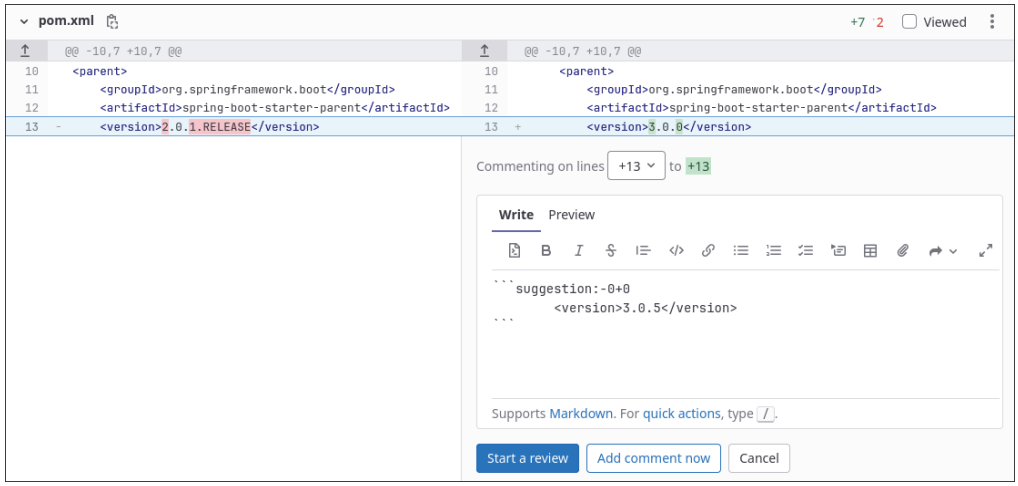

Abbildung 5.10 Ein Änderungsvorschlag direkt im Merge- oder Pull-Request erleichtert die Arbeit und ist sowohl für GitLab als auch für GitHub verfügbar (hier im Beispiel mit GitLab).

Approval Rules für den Prozess

Ein weiteres nützliches Konzept ist die Definition von Regeln, wer Merge- oder Pull-Requests annehmen und genehmigen darf und wer nicht. Je nach Größe des Projekts ist es sinnvoll, mindestens einen Approver für eine Änderung fest zu bestimmen. Dies ist zwar ein Eingriff, der sich negativ auf die Geschwindigkeit des Abschlusses eines Changes auswirkt, jedoch kann so mit einfachen Mitteln erzwungen werden, dass ein Review stattfindet.

Je nach Branche sind solche Approval Rules sogar gesetzlich vorgeschrieben. Dabei handelt es sich um Compliance-Richtlinien, auf die in Abschnitt 11.6 näher eingegangen wird. Vor allem in der Banken- und Finanzindustrie gibt es solche Regeln, die auch von Auditoren geprüft werden. Mit Approval Rules lässt sich dieses Problem schnell lösen.

Hilfreich kann die Datei `CODEOWNERS` sein, in der definiert wird, wer die Hauptansprechpartner und damit Eigentümer der einzelnen Dateien und Pfade im jeweiligen Repository sind. So kann gut kontrolliert werden, welche Änderungen an welchem Code von welchen Personen geprüft werden müssen, um die Qualität sicherzustellen. Durch die einfache Integration muss hier kein zusätzlicher Aufwand betrieben werden, um die passenden Reviewer zu finden, sondern dies geschieht automatisch. Dies ist insbesondere dann relevant, wenn viele Projekte in einem Repository verwaltet werden.

Persönliches Exemplar für Thomas Bartholomäus

Codeowners

Je nachdem, auf welche Git-Hosting-Lösung Sie setzen, ist die Nutzung der Funktion `CODEOWNERS` kostenpflichtig.

Eine `CODEOWNERS`-Datei kann etwa wie folgt aussehen:

```
src/ @dev-team
docs/ @docs-team
kubernetes/ @ops-team
```

In diesem Beispiel würden Änderungen im Verzeichnis `src` dazu führen, dass das Dev-Team ein Review durchführen muss. Wenn allerdings nur Änderungen an `docs` oder `kubernetes` durchgeführt wurden, wird kein zusätzliches Review vom Dev-Team benötigt.

Wichtig ist auch, den Durchlauf beim Review weiter zu verkürzen. Ein Aspekt ist etwa, dass der Review genehmigt wird, der Merge aber erst dann erfolgt, wenn die Pipeline erfolgreich durchgelaufen ist.

Bei größeren Projekten mit vielen nahezu gleichzeitigen Merges sollten sie hingegen sequenziell abgearbeitet werden. Das heißt, sie sollten in einen sogenannten *Merge Train* – manchmal auch *Merge Queue* genannt – aufgenommen werden, bei dem vor jedem Merge die Pipeline einmal durchlaufen wird und bei einem Fehlschlag der Pipeline der Merge abgebrochen wird. Wenn es keine Probleme gibt, werden die Merges kontinuierlich durchgeführt, aber eben nicht alle gleichzeitig. Das Hauptproblem bei einem gleichzeitigen Merge wäre, dass die Änderungen zum Zeitpunkt der Eröffnung des Merges bzw. des Pull Requests noch in Ordnung waren, für den Zeitpunkt des Merges gilt das aber nicht mehr zwangsläufig.

To Squash or Not to Squash

Durch die Commits erhalten Sie eine Historie Ihres Projekts, da jeder Arbeitsschritt sichtbar ist. Wenn Sie meinen bisherigen Hinweisen gefolgt sind, wird die Historie sehr kleinschrittig und damit unübersichtlich aussehen, da ich ja zu vielen kleinen Commits geraten habe. Eine Lösung dafür kann der *Squash* sein, in dem Commits zusammengefasst und zu einem einzigen Commit verschmolzen werden. Dies geschieht in der Regel bei einem Merge, wobei die gängigen Git-Plattformen einen Squash mit anschließendem Merge anbieten.

Dieser Vorgang hat Vor- und Nachteile. Grundsätzlich bin ich ja ein großer Freund davon, dass man mit einfachen Mitteln nachvollziehen kann, welche Änderungen durch einen Commit durchgeführt wurden. Das bedeutet, dass Commits so klein wie möglich gehalten werden sollten und man detailliert sehen soll, welche Änderungen wann committed wurden.

© Rheinwerk Verlag, Bonn 2024

Das hat aber auch Nachteile, da Sie im Fehlerfall nicht nur einen einzelnen Commit zurücknehmen können, sondern mehrere Commits prüfen und rückgängig machen müssen, was umständlich sein kann. Die Antwort auf die Frage, ob man Commits squashen sollte oder nicht, lautet wohl: Es kommt sehr darauf an, wie der Change genau aussieht.

5.4.3 Pair Programming

Das Pair Programming ist eine weitere Methode, die entweder alternativ oder ergänzend zur Steigerung der Code-Qualität implementiert werden kann. Obwohl es eigentlich unabhängig von DevOps nutzbar ist, finden sich viele Grundgedanken der DevOps-Kultur auch im Pair Programming wieder.

Die Idee hinter Pair Programming ist einfach: Zwei Personen arbeiten gemeinsam am gleichen Problem im gleichen Code. Sie schauen sich den vorhandenen Code gleichzeitig an und entwickeln die Lösung zu der Anforderung oder dem Bug.

Eine häufige Annahme ist, dass zwei Entwickler doppelt so viel Code abliefern wie nur eine Person. Das trifft aber nur für ganz einfache Fleißaufgaben zu, die man stumpf runtercodet. Je komplexer das Problem ist, desto eher können zwei Personen im Team ein Problem zügiger lösen als zwei Einzelkämpfer. Denn auch wer allein eine komplexe Aufgabe angeht, muss sich die Gedankengänge selbst erklären.

Rubber Duck Debugging

Eine weitverbreitete Technik in diesem Zusammenhang ist das sogenannte *Rubber Duck Debugging*. Sie stehen vor einem komplizierten Problem oder einem Fehler, den Sie nicht verstehen? Schnappen Sie sich eine Gummiente und erklären Sie ihr die Herausforderung. Aber langsam und kleinschrittig, die Ente ist ja kein Programmierprofi. Indem man das Problem möglichst einfach erläutert, stößt man meistens selbst auf die Lösung. Das Pair Programming geht noch einen Schritt weiter und ersetzt die Ente durch einen mitdenkenden Kollegen.

Im Pair Programming erklärt eine Person der anderen Person die aktuelle Herausforderung und die Gedankengänge. Eine Person schreibt den Code herunter und denkt an die nächsten Zeilen, die zweite Person denkt meist schon ein paar Schritte weiter. Das ist besonders dann hilfreich, wenn andere Teile des Sourcecodes parallel angeschaut werden müssen.

Die Vorteile sind offensichtlich: Mögliche Probleme fallen beim gemeinsamen Darüber-Reden schneller auf als bei einem asynchronen Code-Review. Mögliche Fehler lassen sich zügiger korrigieren, wenn man ohnehin im Gedankenfluss ist.

Persönliches Exemplar für Thomas Bartholomäus

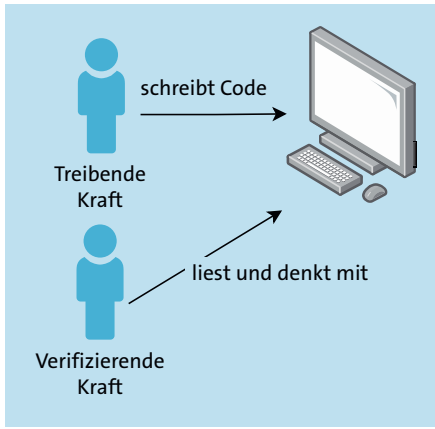

Abbildung 5.11 Eine Person schreibt den Code, während die andere Person ihren Fokus stärker auf das Lesen und Mitdenken setzt.

Pair Programming tut aber auch der Teamkultur gut, denn Wissen wird geteilt. Bei einer Session zwischen einem Grünschnabel und einem alten Hasen können beide gut voneinander lernen. Der Junior lernt die Code-Basis und das Domänenwissen kennen, der Senior erhält einen frischen Blick auf alte Probleme. Im Duo ist auch die Hürde kleiner, eine vermeintlich »dumme« Frage zu stellen, und die zwischenmenschliche Beziehung wird durch eine solche Session verbessert.

Meiner Erfahrung nach hat das Pair Programming auch noch einen weiteren positiven Nebenaspekt: Zwei Leute können sich zusammen oft besser konzentrieren, als wenn sie allein für sich arbeiten. Nachrichten in Messengern oder E-Mails, die meist kurzfristig nicht sonderlich wichtig sind, lassen sich beim gemeinsamen Arbeiten leichter nach hinten schieben, statt sofort auf sie zu antworten. Das verbessert und beschleunigt das Coden, da man nicht so schnell aus Gedankengängen geworfen wird.

Mob Programming

Und übrigens: Für das Pair Programming müssen beide Personen nicht unbedingt gemeinsam vor demselben Rechner sitzen: Das funktioniert mit den üblichen Videokonferenzlösungen ebenfalls gut.

Was ebenfalls möglich ist: Sie können nicht nur als Paar arbeiten, sondern gleich als »Mob« mit mehreren Personen. Ein englisches E-Book zu dem Thema findet sich unter *https://www.remotemobprogramming.org/*.

Durch Pair Programming wird also schneller besserer Code geschrieben. Ein separates Code-Review wird nach einer Pair-Programming-Session nicht mehr unbedingt benötigt, da schon direkt zwei weitere Augen auf den Code geschaut haben, während er programmiert wurde. Pair Programming und reguläres Code-Review müssen sich

© Rheinwerk Verlag, Bonn 2024

aber gar nicht ausschließen: Für einfachere Sachen, die man rasch allein implementieren kann, wird ein reguläres Code-Review durchführt; für »dicke Bretter« sucht man sich einen guten Partner und geht das Problem gemeinsam an.

In einer guten DevOps-Kultur heißt das natürlich, dass Ihr Gegenüber nicht unbedingt eine andere Entwicklerin sein muss. Es kann gut sein, dass sich das Problem zusammen mit einem Vertreter des Ops-, QA- oder Security-Teams viel besser lösen lässt. Je nach Problem und je nach Bedarf.

Reflexion

Bitte verzichten Sie nicht auf Code-Reviews, denn so können einige Fehler, technische Schulden und wertvolles Teilen von Wissen direkt in den Entwicklungsalltag eingebaut werden.

Die Einführung von Code-Review geht allein im eigenen Team, erfordert vermutlich aber auch in ihm einiges an Überzeugungsarbeit. Skeptische Kollegen werden anmerken, dass dann zu viel und zu langsam entwickelt werde und alles dadurch noch langsamer werde. Zu Beginn ist das auch richtig, daher ist der Fokus auf einen effizienten Workflow mit möglichst kleinen Änderungen essenziell.

Pair Programming lässt sich ebenfalls mit wenigen Hürden innerhalb des eigenen Teams einführen. Es ist hilfreich, wenn Sie es einsetzen, um an komplexen Sachverhalten zu arbeiten, ganz unabhängig davon, ob erfahrene oder weniger erfahrene Personen beteiligt sind.

5.5 Inner Sourcing – Code im Unternehmen teilen

Wenn Sie der Geschichte der Firma *schick-gekleidet.de* bisher gefolgt sind, werden Sie festgestellt haben, dass viele Probleme daher kommen, dass die Zusammenarbeit zwischen Entwicklungs-, Qualitätssicherungs- und Betriebsteam nicht funktioniert. Das ist aber bisher nicht die ganze Wahrheit. Es geht auch um die Zusammenarbeit der einzelnen Entwicklungsteams untereinander. Oder sollte man sagen, dass sie bisher eher gegeneinander gearbeitet haben?

Obwohl es ein einzelner Online-Shop ist, besteht die Anwendung von *schick-gekleidet.de* aus verschiedenen Teilen, die von verschiedenen Teams entwickelt werden. Das bedeutet, dass es unterschiedliche Teams gibt: Ein Team arbeitet beispielsweise am Backend für die Abrechnungen, ein anderes ist für die Suche verantwortlich und ein weiteres sitzt an dem Modul, mit dem den Kunden relevante Angebote gezeigt werden sollen.

Bisher arbeiteten die Teams alle komplett getrennt voneinander: Jedes Team hat das eigene Repository mit eigenen Zugängen, in die nur die Teammitglieder hereinschauen konnten.

Wie bereits thematisiert wurde, sieht so ein solches Vorgehen auf den ersten Blick gar nicht so doof aus: Die Teams müssen keine Angst haben, dass andere Leute ihren Code lesen und auf schlechte Gedanken kommen.

Aus dieser Organisation ergeben sich allerdings einige wesentliche Nachteile, denn die Teams müssen zwangsläufig zusammenarbeiten, da die verschiedenen Teile des Online-Shops sich gegenseitig über APIs aufrufen. Das wäre an sich kein Problem, wenn es denn eine vollständige, aktuelle und detaillierte Dokumentation gäbe.

Aber Hand aufs Herz: Haben Sie schon einmal eine interne API-Dokumentation gesehen, die vollständig, aktuell und vor allem hilfreich ist? Es kommt daher immer wieder zu Problemen, da gelegentlich mal etwas nicht so funktioniert, wie es dokumentiert wurde. Oder man sucht die Dokumentation vergebens und weiß daher überhaupt nicht, wie man die unterschiedlichen Teile zusammenbauen soll.

Bisher sah dann die Zusammenarbeit so aus, dass neue Tickets im System des anderen Teams angelegt wurden. So sollten Fehler und Probleme besprochen werden. Da die ganze Entwicklung allerdings dem Wasserfall-Modell folgte, gab es kaum Gelegenheiten, Bugs, Feature-Wünsche oder fehlende Dokumentation kurzfristig und agil zu ergänzen. Probleme wurden nicht gelöst und blieben stattdessen liegen.

Damit ein Ticket überhaupt bearbeitet wurde, mussten umfassende Problembeschreibungen und Diagnosen zusammengesucht werden. Viele Tickets provozierten zunächst Nachfragen, ob das andere Team die API auch wirklich richtig aufruft.

Das war ein für alle Beteiligten langwieriger und anstrengender Prozess, der nicht selten damit endete, dass Tickets kommentarlos geschlossen wurden, weil Teams, die auf Probleme in fremden Code-Teilen gestoßen sind, lieber eigene Workarounds gebaut haben, anstatt sich weiter mit der Kommunikation herumzuschlagen.

Anstatt das eigentliche Problem zu beheben, wurden also Symptome bekämpft und umschifft. Das wiederum führte zu weiteren technischen Schulden im Sourcecode – nichts ist so stabil wie ein temporärer Fix.

Diese Probleme sind schon schwerwiegend genug, dass sie die Arbeit an neuen Features oder das Beseitigen von Fehlern fast unmöglich machen. Aber auch wenn Sie die Prozesse durch bessere Technik vereinfachen, indem Sie unter anderem ein neues Ticketsystem einführen: Das bringt nichts, denn die Probleme menschlicher Natur sind weiterhin vorhanden.

Ständig kommt es zum »Fingerzeigen« auf andere Teams. Eigene Herausforderungen werden auf die Fehler von anderen Teams geschoben, sodass eine toxische Kultur entsteht: »Solange die da drüben das Ticket nicht bearbeiten, kann ich nicht weiterarbeiten.«

Und da niemand als Bremser gelten will oder gar für den schleppenden Fortschritt in anderen Teams verantwortlich sein möchte, führt das dazu, dass eigene Fehler gar

© Rheinwerk Verlag, Bonn 2024

nicht mehr eingestanden werden. Oder dass sie, als Krönung sozusagen, sogar als Features verkauft werden, die nicht korrigiert werden müssen. Man will sich ja selbst im besseren Licht darstellen. Schließlich geht es auch um den Ruf innerhalb der Firma.

Das sind Probleme der Firmenkultur, die sich nicht technisch lösen lassen. Stattdessen ist ein Umdenken erforderlich, und zwar unter dem Schlagwort *Inner Sourcing*. Inner Sourcing ist im Grunde nicht viel mehr als der Einsatz von Praktiken aus der Open-Source-Bewegung, nur eben innerhalb eines Unternehmens.

5.5.1 Open Source

Bevor wir uns das Inner Sourcing genauer anschauen, sollte klar sein, was die Praktiken und Werte sind, die in Open-Source-Projekten gelebt werden. Diese bewegen sich relativ nah an DevOps-Prinzipien, da beide einen hohen Wert auf eine offene Kultur legen, um Kollaboration zu ermöglichen. Dazu gehört eine hohe Transparenz, sodass für alle klar ist, wie der aktuelle Stand aussieht und woran jetzt und in Zukunft genau gearbeitet wird.

Grundsätzlich fällt alles unter Open Source, was unter einer Open-Source-Lizenz steht. Eine vollständige Liste finden Sie unter *https://opensource.org/licenses*.

Zentral ist dabei natürlich, dass der Code offengelegt ist. Sie können nicht nur den Code dieser Projekte einsehen, sondern ihn sogar anpassen und weiterverteilen. Dies führt dazu, dass Außenstehende problemlos neue Ideen vorschlagen und einfließen lassen können und sich das Projekt stets weiterentwickelt. Und genau dieser Aspekt fehlt häufig innerhalb von Firmen.

Für Open-Source-Projekte ist es überlebensnotwendig, dass sie auf DevOps-Kollaborationsplattformen wie GitHub oder GitLab liegen, auf denen man den Code mit der vollständigen Historie einfach einsehen kann. Dort liegt dann meist auch die Dokumentation zu dem Projekt, die erklärt, wie es gebaut wird, wie man es lokal zum Laufen bekommt, es auf Produktivsystemen deployt und nutzt. Nur so wird es Interessenten leicht gemacht, neue Features und Bugfixes zum Projekt beizutragen.

Code of Conduct

Neben diesen technischen Hilfestellungen haben die meisten Open-Source-Projekte inzwischen auch Regeln für die Zusammenarbeit veröffentlicht, den sogenannten *Code of Conduct*. Das Ziel ist, dass möglichst viele Menschen einfach mitarbeiten können. Weitere Informationen finden Sie unter *https://opensource.guide/code-of-conduct/* oder *https://www.contributor-covenant.org/*.

Das Ergebnis ist, dass Sie auf diese Weise zügig das Projekt herunterladen, bauen und zum Laufen bekommen, ohne dass Sie Berechtigungen anfragen müssen. Und wenn

diese Hürde aus dem Weg geräumt ist, macht es idealerweise Spaß, sich dort zu engagieren und das eigene Wissen und die eigene Zeit zu investieren.

Trauen Sie sich also ruhig, bei Open-Source-Projekten mitzuarbeiten. Natürlich kann man nicht zum Einstieg gleich Code zu sicherheitskritischen Stellen des Linux-Kernels einreichen, aber in fast allen Projekten gibt es kleinere Aufgaben, die man sich zum Einstieg vornehmen kann.

Sie können dann Verbesserungsvorschläge einreichen, die von erfahrenen Projektmitgliedern geprüft werden – gewissermaßen ein Code-Review. Diese Mitglieder entscheiden, ob die Änderung aufgenommen wird, ob weitere Anpassungen nötig sind … oder ob der Zusatz komplett abgelehnt wird.

Sie wissen vermutlich, dass es gar nicht so selten ist, dass man eine Open-Source-Software als Abhängigkeit im eigenen Projekt hat und dort auf einen Fehler oder ein fehlendes Feature stößt. Melden Sie solche Stellen beim Projekt. Noch besser wäre es natürlich, wenn Sie den Fehler direkt korrigieren und die Lösung beitragen. So profitieren letztlich alle davon, und zahllose Open-Source-Projekte beweisen, dass dies eine äußerst produktive Art ist, um Software zu entwickeln.

Genau diese Praktiken gilt es nun auch innerhalb einer Firma zu etablieren.

5.5.2 Der Weg zum Inner Sourcing

Beim Inner Sourcing geht es also darum, die Kultur und die Praktiken der Open-Source-Bewegung innerhalb der eigenen Organisation zu nutzen. In Abbildung 5.12 sehen Sie beispielsweise, dass Entwickler aus unterschiedlichen Projekten Zugriff auf die gemeinsam genutzte API haben, die ihrerseits von einer gesonderten Gruppe von *Maintainern* verwaltet wird.

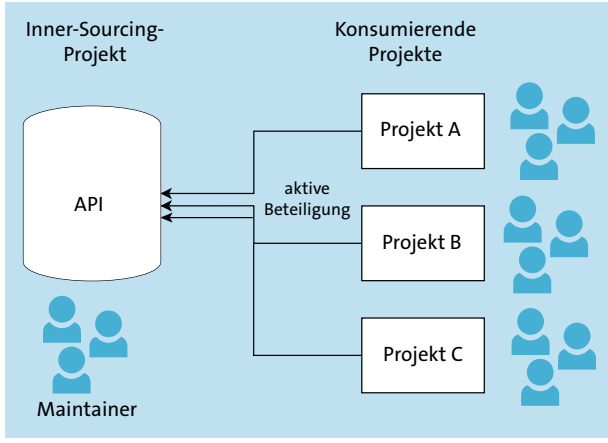

Abbildung 5.12 Beim Inner Sourcing arbeiten die Projekt-Teams aktiv an anderen Projekten mit.

© Rheinwerk Verlag, Bonn 2024

Der Weg dahin ist zwar aufwendig, bringt aber sehr viele Vorteile mit sich. Um das Ganze umzusetzen, muss ein Engagement der Führungsetage gegeben sein.

Denn ohne den Willen von ganz oben, diesen Weg zu gehen, lässt sich ein solcher Wandel nicht umsetzen. Hinzu kommt, dass solche Abschottungen häufig organisatorisch bedingt sind, weil es Policys gibt, die das Öffnen von Projekten verbieten. Sie können Inner Source also nicht mit dem Holzhammer durchsetzen, sondern müssen prüfen, wo mehr Offenheit und Transparenz sinnvoll sind.

Schritt 1: Offenlegen der Projekte

Der Weg zu mehr Offenheit und Transparenz sollte in mehreren Schritten gegangen werden. Der erste und einfachste Schritt ist das Offenlegen der Projekte, sodass alle zumindest den Sourcecode lesen können. Aber auch hier müssen Sie mit Bedacht vorgehen, denn so ganz einfach und ohne Risiken ist das natürlich auch nicht. Falls zuvor Passwörter oder andere Zugangsdaten hart im Quellcode gespeichert waren, müssen diese erst entfernt und durchrotiert werden, was noch weitere Änderungen erfordert.

Und noch komplizierter wird es, wenn auch die Projekte anderer Abteilungen betroffen sind. Falls beispielsweise auch das Betriebsteam Repositorys benutzt, um Konfigurationsdateien zu speichern und zu versionieren, muss sorgfältig geprüft werden, ob durch mehr Transparenz keine Sicherheitsprobleme entstehen. Informationen zu Zertifikaten und anderen sicherheitskritischen Bereichen müssen natürlich weiterhin geschützt werden. Das ändert aber nichts daran, dass es dem Entwicklungsteam hilft, wenn es besser weiß, wie das Setup der eigenen Software auf den Produktivsystemen aussieht.

Schritt 2: Zentralisierung der Repositorys

Durch die Freigabe des Quellcodes ist zwar ein wichtiger Schritt getan, allerdings sind die Repositorys dann immer noch auf verschiedenen Systemen verteilt. Hinzu kommt, dass selbst verwaltete Schatten-IT von einzelnen Teams nicht selten ist. Aber auch sonst gilt meist, dass nicht alle User überall Zugriff haben, und somit bringt ein Öffnen der Freigabe recht wenig.

Der entscheidende Schritt ist die Einrichtung einer zentralen Kollaborationsplattform, auf der zukünftig nicht nur der Code gehostet wird. Die Projekte müssen dafür umgezogen werden, was ebenfalls einiges an Arbeit erfordert, da Schnittstellen zu anderen Tools erneut konfiguriert werden müssen. Das ist kein Projekt für einen Nachmittag!

Persönliches Exemplar für Thomas Bartholomäus

Schritt 3: Dokumentation für die Projekte schreiben

Wenn Sie nun eine einheitliche Plattform für Ihre Repositorys haben, können Sie Dokumentation schreiben. Damit ist in diesem Zusammenhang nicht die Kommentierung im Sourcecode gemeint, sondern Erklärungen, wie man das Projekt baut, wie man es lokal zum Laufen bekommt und wie man es auf einem Produktivsystem deployt.

Durch diese Informationen wird erst ermöglicht, dass Personen aus anderen Teams die Software so weit nutzen können, dass sie Änderungen beitragen können. Die Hürde für den Einstieg muss so gering wie möglich sein, denn niemand hat Lust, erst mal eine Build-Umgebung zu rekonstruieren, die nicht dokumentiert ist.

Je schneller ein Teammitglied eine funktionsfähige Entwicklungsumgebung für das Projekt an den Start bekommt, desto schneller kann man mit der Arbeit beginnen und desto niedriger liegt die Schwelle, überhaupt aktiv zu werden. Und das völlig unabhängig davon, ob dieses Mitglied neu in dem Team ist oder eben aus einem anderen Team kommt.

Schritt 4: Kollaboration via Code-Review ermöglichen

Durch den Einsatz einer zentralen Kollaborationsplattform wie GitHub oder GitLab können nun Änderungen mit einfachen Handgriffen zu den Projekten beigetragen werden. Dabei wird der Code aber nicht einfach ersetzt, Sie müssen sich also keine Gedanken machen, dass andere Teams plötzlich Änderungen beitragen, ohne dass Sie etwas davon merken. Die Stichworte heißen *Merge Requests* bzw. *Pull Requests*.

Die Teams hinter dem Projekt, zu dem beigetragen wird, werden auf diesem Weg über Änderungen informiert und können ein Code-Review durchführen, wie in Abschnitt 5.4 beschrieben. Es muss kein großer Unterschied gemacht werden, ob die Änderung nun von innerhalb oder von außerhalb des Teams kommt. Wichtig ist nur, dass die Änderung eine gute Qualität hat und dass das Review ordentlich durchgeführt wird.

5.5.3 Vorteile von Inner Sourcing

Aber was für Vorteile ergibt sich nun aus dem Inner Sourcing und was hat es mit DevOps genau zu tun? Genau genommen kann natürlich auch Inner Sourcing ohne DevOps betrieben werden. Allerdings geht es bei DevOps ja vor allem um das Einreißen von Silos zwischen den Teams und um die Steigerung der Produktivität durch Effizienz, was eben durch Inner Sourcing erreicht wird.

Effizientere und effektivere Entwicklung

Der erste Punkt ist also, wenig überraschend, die effizientere und effektivere Entwicklung. Stellen Sie sich beispielsweise vor, dass es ein Team gibt, das Firewall-Regeln zwischen Quell- und Zielsystemen verwaltet. Dem vorgeschaltet ist ein Genehmigungs-

© Rheinwerk Verlag, Bonn 2024

prozess: Das Team des Quellsystems musste ein Ticket im Ticketsystem erstellen, um auf das Zielsystem in den Test- und Produktivumgebungen zugreifen zu können. Das Ticket musste vom Team des Zielsystems entsprechend genehmigt werden. Erst dann konnte das Betriebsteam mit der Bearbeitung des Tickets beginnen.

Die Änderung selbst dauerte in der Regel fünf Minuten, die einzelnen Prozessschritte sorgten allerdings für eine lange Wartezeit: Zuerst musste das Ticket vom Team des Zielsystems genehmigt werden, dann wartete man auf freie Zeit beim Betriebsteam, und schließlich überprüfte das Team des Quellsystems, ob auch wirklich alles funktionierte und nichts vergessen oder falsch gemacht worden war. Dieser Prozess dauerte oft mehrere Wochen für einen Eingriff, die nicht viel mehr als die Änderung einer Zeile war.

Inner Sourcing macht solche Abläufe einfacher und schneller: Wäre das Repository offen und eine einfache Zusammenarbeit möglich, hätte man sich das Ticket sparen können. Das Team des Quellsystems hätte direkt eine Änderung per Merge oder Pull Request vorschlagen können. Das Team des Zielsystems sowie das Betriebsteam hätten die Änderung nur noch genehmigen müssen und die Änderung wäre direkt ausgerollt worden.

Dies verkürzt die mehrwöchige Wartezeit bis zum Rollout, da einige Schritte kombiniert und somit reduziert werden können. Das ist nicht nur effizienter, sondern überwindet auch die organisatorischen Grenzen von drei verschiedenen Teams – und alle sind glücklich, weil die Änderung schneller durch ist. Es gibt keinen Engpass und der Prozess ist weniger fehleranfällig.

Aus Inner Sourcing können eigene Produktideen entstehen

Aus Sicht der Software-Entwicklung ergeben sich weitere Vorteile. Grundsätzlich sind Open-Source-Projekte so aufgebaut, dass sie wiederverwendet werden können. Dies bietet sich auch innerhalb von Organisationen an, um Doppelentwicklungen zu vermeiden. Teilweise entstehen durch solche Entwicklungen aus Projekten sogar eigene Produkte, die anderweitig für das Unternehmen genutzt werden können.

Aus Eigenentwicklungen werden Open-Source-Projekte

Wenn Inner Sourcing gut umgesetzt wird, können Sie schauen, ob man das eine oder andere Projekt nicht gleich unter einer Open-Source-Lizenz freigibt, um so der Community etwas zurückzugeben. Aber nicht nur das: Sie werden auch davon profitieren, wenn weitere Teams die eigene Software verwenden und womöglich auch weiterentwickeln.

Außer für einfache Prozesse sorgt das Inner Sourcing aber auch auf sanfte Art für bessere Code-Qualität. Da der Code von vielen Augen gesehen werden kann, enstehen weniger krude Workarounds, weil andere Teams, die von diesen Code-Teilen abhängig sind, sonst sofort Alarm schlagen würden.

Persönliches Exemplar für Thomas Bartholomäus

Höhere Zufriedenheit bei Mitarbeitenden

Aber nicht zuletzt sollten Sie nicht vergessen, dass diese Arbeitsweise zu einer besseren Mitarbeitermotivation führt. Niemand mag es, wenn man sich bei der Arbeit an schwierigen und komplexen Problemen auch noch mit anderen Hürden herumschlagen muss. Nicht selten hört man ja: Entwickler sind faul. Das stimmt (zumindest meistens, schließlich sind sie Menschen).

Je höher die Hürden sind, desto fauler ist man, daran etwas zu ändern. Desto wahrscheinlicher ist es dann, dass man stattdessen versucht, mit irgendwelchen Workarounds Probleme zu lösen, die man besser an der Quelle korrigieren sollte. Aber wirklich zufrieden ist damit niemand. Reduzieren Sie besser die Barrikaden, dann arbeiten alle glücklicher und effizienter.

Wie es nicht gehen sollte …

Ein Bekannter von mir ist ein großer Freund von Inner Sourcing und davon, Probleme an der Wurzel zu packen. Als GitLab-Admin in seiner Organisation hat er Einsicht in alle Projekte auf der Instanz. Eines Tages wollte er eine Änderung direkt zu einem Projekt beitragen, um Workarounds zu vermeiden. Er öffnete dazu einen Merge Request mit der gewünschten Änderung. Da Inner Sourcing aber nicht erwünscht war, hat er statt Lob für seine Hilfe Ärger bekommen: »Bitte nicht in fremde Projekte einmischen!« Und das, obwohl die Änderung sinnvoll und hilfreich war.

Es kommt also ganz auf die Mentalität und die Kultur an: Nur wenn es von oben auch gewollt und gewünscht wird, ist eine solche Arbeitsweise auch praktisch möglich.

Auch wenn der Aufwand für eine solche Migration hoch ist, überwiegen die Vorteile. Und leider ist es auch so, dass die Umstellungsphase reichlich Schmerzen bereiten wird, weil Arbeitsweisen und liebgewonnene Gewohnheiten umgestellt werden müssen, ohne dass man unmittelbar einen Vorteil erkennt.

Die neuen Strukturen müssen schließlich erst geschaffen werden, bis die Kollaboration reibungslos funktioniert. Und es kann auch frustrierend sein, wenn man den Aufwand auf sich nimmt, für eine gute Dokumentation und viel Transparenz sorgt, der Input von außen aber dann doch eher gering ausfällt, weil die Kenntnisse und das Interesse der anderen Teams für das eigene Projekt doch eher gering sind.

Dennoch überwiegen die Vorteile deutlich: Inner Sourcing ist ein wichtiger und unverzichtbarer Punkt in der Transformation zu DevOps. Es legt das Fundament für einen effizienten Software-Entwicklungslebenszyklus, und wenn das Fundament bereits wackelig ist, werden auch die nächsten Schritte wie der Build-Prozess, das Testen und das Deployment schwierig. Darauf gehe ich in den nächsten Kapiteln ein.

© Rheinwerk Verlag, Bonn 2024

5.5.4 Was ist eigentlich mit Monorepos?

Eine Frage, die ich immer wieder höre, lautet, ob *Mono Repositorys* (Monorepos) nicht nur aus DevOps-Sicht sinnvoll wären. Als Hauptgrund wird vermutet, dass es einfacher wäre, ein Refactoring des Quellcodes durchzuführen, da in einem Merge Request Änderungen an mehreren Projekten durchgeführt werden könnten.

Doch zunächst ein Schritt zurück: Was ist überhaupt ein Monorepo?

Spricht man von einem Monorepo-Ansatz, so bedeutet dies im Prinzip, dass es nur ein einziges Repository im gesamten Unternehmen gibt. In diesem Repository sind alle Projekte des Unternehmens enthalten. Dazu gehören sowohl aktive als auch inaktive Projekte.

Der bekannteste Fall eines Monorepos im produktiven Einsatz ist Google, das ein eigenentwickeltes Sourcecode-Management-Tool gebaut hat, um damit genau ein einziges Repository zu verwalten. Auch Meta Platforms, das Unternehmen hinter Facebook und Instagram, setzt im Wesentlichen auf ein Monorepo, bei dem die Git-kompatible Eigenentwicklung Sapling (*https://engineering.fb.com/2022/11/15/open-source/sapling-source-control-scalable/*) zum Einsatz kommt, die Ende 2022 als Open Source veröffentlicht wurde.

So ein Monorepository ist verdammt riesig und enthält die Historie von (fast) allen Projekten der Firma. Bei einem Mono-Repository-Ansatz mit Git stößt man allerdings schnell an die Grenzen der Software. Git ist nicht besonders dafür geeignet, mit exorbitant vielen Verzeichnissen und Dateien zu arbeiten. Für die meisten Unternehmen dürfte dies kein Problem sein, da sie mit eher kleinen Code-Basen arbeiten. Trotzdem kann die Größe des Repositorys schnell einige Gigabyte erreichen, was den gesamten Entwicklungsprozess verlangsamt, da z. B. die Pipeline das Repository fast jedes Mal neu klont. Dies kostet also nicht nur Zeit, sondern auch Geld, da mit einer höheren Belastung durch die Automatisierung zu rechnen ist.

Im allgemeinen Verständnis vieler, die den Begriff »Monorepository« verwenden, liegt jedoch fast immer ein Denkfehler vor. Denn für viele ist ein Monorepo ein Repository, in dem mehrere Projekte in einem Repository gespeichert sind, also zum Beispiel der Quellcode des Frontends, des Backends und der mobilen Apps. Manche Firmen sprechen sogar von mehreren Monorepos, was natürlich keinen Sinn macht, da der Name »Monorepo« implizit besagt, dass es genau *eines* ist. Ein passenderer Begriff wäre dann eher »Multi-Projekt-Repository«.

Ich möchte das Monorepo-Thema hier als Beispiel dafür verwenden, dass das bloße Kopieren von Ansätzen anderer, sehr großer Firmen nicht immer eine gute Idee ist. Es ist auch ein anschauliches Beispiel dafür, dass man nicht das große Ganze sieht.

Wie schon oft erwähnt, ist die Integration in sehr viele Phasen des DevOps-Lebenszyklus immens wichtig, um die Sichtbarkeit überall zu erhöhen. Es gibt diverse Vorteile und auch Nachteile bei der Verwendung eines Monorepos. Ich werde hier nicht

Persönliches Exemplar für Thomas Bartholomäus

auf alle eingehen, aber es lohnt sich immer, einen Blick auf die Tools zu werfen, die man einsetzt, um zu sehen, inwieweit sie diesen Ansatz wirklich unterstützen.

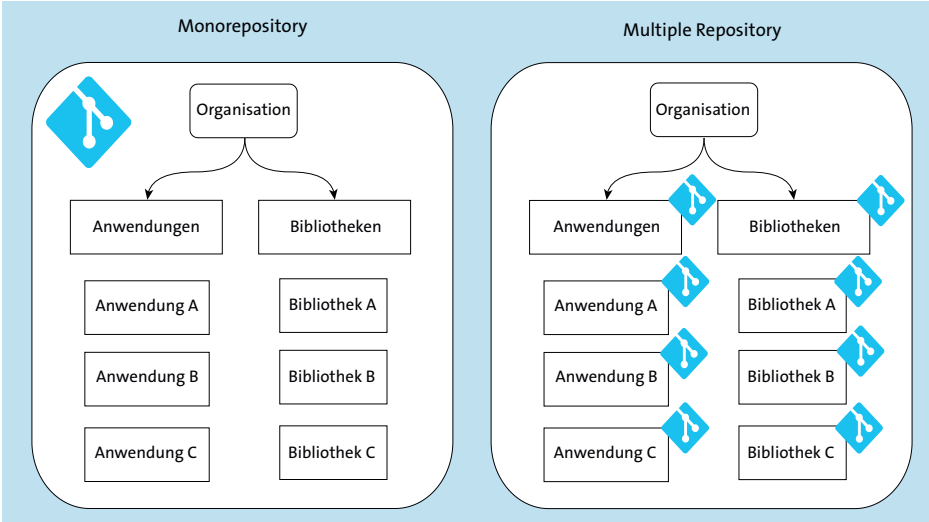

Abbildung 5.13 Bei einem Monorepository gibt es genau ein Repository, während es multiplen Repositorys eben mehrere Repositorys gibt.

Einer der Hauptgründe für ein Monorepository (oder auch für ein Multi-Projekt-Repository) ist die vereinfachte Möglichkeit, interne Abhängigkeiten zu verwalten. Und dieser Anwendungsfall ist gar nicht so selten. Häufig ist es zum Beispiel so, dass es innerhalb eines Unternehmens mehrere Projekte gibt, die die gleiche Abhängigkeit verwenden. So weit nichts Besonderes. Wie sieht nun aber der Prozess aus, wenn eine Änderung an der Abhängigkeit vorgenommen werden soll?

Im klassischen Konzept, also ohne die Verwendung eines Monorepos, ist eine Änderung in zwei Repositorys gleichzeitig notwendig: einmal in der Bibliothek, wo die eigentliche Änderung durchgeführt wird, und dann die nächste Änderung im Hauptprojekt. Letztere hängt jedoch von der Änderung der Abhängigkeit ab. Hier muss also zunächst gewartet werden, bis die Änderung eingearbeitet ist, bevor die Änderung im Hauptprojekt durchgeführt werden kann.

Der Prozess dauert also automatisch etwas länger und führt zu längeren Entwicklungszeiten. Zusätzlich könnte es auch noch sein, dass mehrere Projekte diese Abhängigkeit verwenden und diese auch eine Änderung benötigen. Diese Änderung muss – auch wenn es sich vielleicht nur um eine kleine Anpassung handelt – in jedem Projekt nachgepflegt werden.

Hier zeigt sich also bereits ein wesentlicher Vorteil von Monorepos bzw. Multi-Projekt-Repos: Eine Änderung an einer Abhängigkeit kann in einem Rutsch und in einem Review durchgeführt werden. Komplexere Abhängigkeiten beim Mergen von

© Rheinwerk Verlag, Bonn 2024

Änderungen über verschiedene Repositorys hinweg werden vermieden, und gleichzeitig kann das Review transparenter durchgeführt werden, da man in einer Ansicht und mit einem Review die Änderung projektübergreifend überblicken kann.

Was ich hier allerdings bislang nicht betrachtet habe, ist das CI- (und CD-)Setup, denn das hat auch einen wesentlichen Einfluss auf die Arbeitsweise.

Die Realität ist nämlich, dass es zwar einige Vorteile gibt, wie etwa die angesprochene Möglichkeit, mit einem Review gleich alle Änderungen in mehreren Projekten mitzuziehen; allerdings muss das Ganze auch einfach über die Pipeline automatisierbar gemacht werden, wo jedes Mal nur das gebaut wird, was auch wirklich gebraucht wird.

Die heutigen Toolchains bilden das allerdings nicht sonderlich einfach ab, sodass der Aufbau mühselig ist. Es wird also tendenziell eher dazu führen, dass ein Monorepo komplex einzurichten ist, und auch die langfristige Wartung ist meistens eher anstrengend. Denn nicht für jeden Change will man alle Projekte neu bauen. So muss man einen hohen Engineering-Aufwand in die CI/CD-Toolchain hereinstecken, um effiziente Pipelines zu implementieren.

Der Trade-off ist damit also viel zu hoch, sodass man lieber kleinere, voneinander unabhängige Projekte und somit Repositorys anlegen sollte, um die Arbeit mit ihnen einfach zu halten. Dann hat man allerdings den Nachteil, dass einige Änderungen gleich in mehreren Repositorys implementiert und getestet werden müssen. Dieses Problem ist allerdings überwiegend für größere Firmen relevant, da bei kleineren Firmen der Overhead vernachlässigbar ist.

Es hängt am Ende auch an der Software-Architektur: Bei lose gekoppelten Microservices klappt so etwas besser als bei einer monolithischen Architektur, wo jede Änderung sowieso dazu führt, dass einmal alles gebaut werden muss. Je stärker gekoppelt die Abhängigkeiten sind, desto eher lohnt es sich, sie in einem Repository zu verwalten. Je loser die Abhängigkeiten gekoppelt sind, desto eher dürfte es sinnvoll sein, diese in getrennten Repositorys zu verwalten.

Info-Website

Unter *https://monorepo.tools/* findet sicheine gut gestaltete Website, die viel tiefer in die Thematik rund um Monorepos eingeht.

Hier ist also die Frage, wie sehr die verschiedenen Tools auf dem Markt die eigenen Ideen überhaupt unterstützen. Die beste Idee bringt letztlich nichts, wenn sie nur mit sehr großem Aufwand umgesetzt werden kann.

Riesige Technologie-Firmen wie Google können sich das vielleicht leisten – die meisten Firmen sind jedoch weder so groß noch dazu fähig. Stattdessen lautet mein Appell: Schauen Sie sich immer besser die verschiedenen Tools an, um zu herauszu-

Persönliches Exemplar für Thomas Bartholomäus

finden, was damit am besten unterstützt wird. Kompromisse muss man nämlich immer eingehen.

Reflexion

Je weniger Befugnisse Sie in Ihrer Organisation haben, desto schwieriger ist es, Inner Sourcing einzuführen. Das betrifft sowohl das Öffnen der Repositorys als auch die Umstellung der Entwicklungsprozesse in einer Weise, dass jeder in der Organisation etwas beitragen kann.

Wenn Sie DevOps-Prinzipien leben und auch danach arbeiten, dann ist die reine (technische) Umsetzung hingegen weniger das Problem. Einfache Schritte in die richtige Richtung wären etwa, internen Personen Zugriff zum Projekt zu gewähren, die regelmäßig Bug-Reports aufmachen. So können diese Zugänge direkt als Testballon verwendet werden: Welche Richtlinien müssten dadurch geändert werden? Zeigen sich schon Effizienzgewinne durch die engere Kollaboration?

5.6 Fazit

In diesem Kapitel habe ich die Verwaltung des Quellcodes behandelt. Vielleicht haben Sie am Anfang gedacht, dass es sich um ein sehr technisches Thema handelt. Inzwischen haben Sie aber sicher erkannt, dass es vielmehr eine Frage der Kultur und damit der Zusammenarbeit ist als eine Frage der Technik selbst.

Mit Code-Review und Pair Programming wird sichergestellt, dass die Qualität und die Sicherheit stimmen, bevor Änderungen endgültig in das Projekt einfließen; und durch Inner Sourcing wird dieser Ansatz auch für Personen außerhalb des Teams zur Verfügung gestellt, um eine bestmögliche Zusammenarbeit zu ermöglichen und damit Zeit und Aufwand so effizient wie möglich zu gestalten.

© Rheinwerk Verlag, Bonn 2024

Kapitel 6
Continuous Integration und der Build Prozess

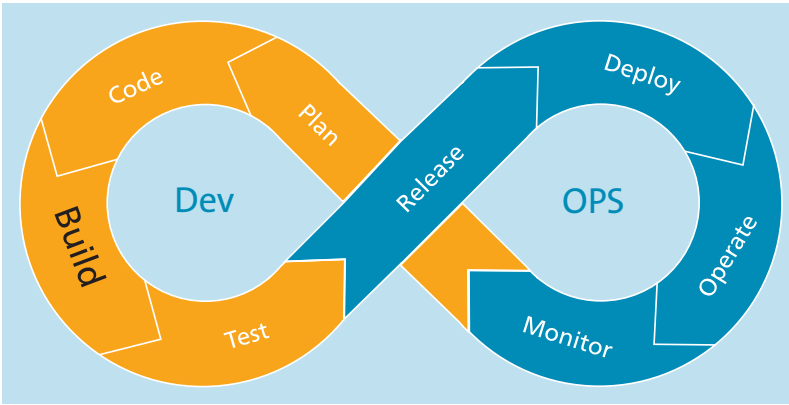

Abbildung 6.1 Build: Der Code wird kompiliert oder so verpackt, dass er in einer Umgebung laufen kann.

Nachdem wir uns angeschaut haben, wie die direkte Kollaboration zwischen den verschiedenen Teams und Projekten auf der Ebene des Quellcodes verbessert werden kann, geht es nun mit dem nächsten Schritt weiter.

Tatsächlich war schon das letzte Kapitel der Einstieg in das Thema *Continuous Integration*, bei dem es kurz gesagt darum geht, möglichst zeitnah den Code in den Hauptentwicklungsbranch zu integrieren. Um das zu bewerkstelligen, ist ein gescheiter Build-Prozess nötig. Und genau darum geht es in diesem Kapitel.

6.1 Die typischen Probleme im Build-Prozess

Sie wissen schon, dass der Online-Shop von *schick-gekleidet.de* aus verschiedenen Modulen und Applikationen besteht, die auch von unterschiedlichen Teams entwickelt werden. So gibt es verschiedene Backend-Systeme, wie ein Billing-Modul für alle Abrechnungen, oder auch den Teil des Online-Shops, in dem die neuesten Angebote aufbereitet und präsentiert werden. Und damit nicht genug: Es gibt auch noch zwei Teams, die sich jeweils um die Mobile Apps für Android und für iOS kümmern.

Persönliches Exemplar für Thomas Bartholomäus

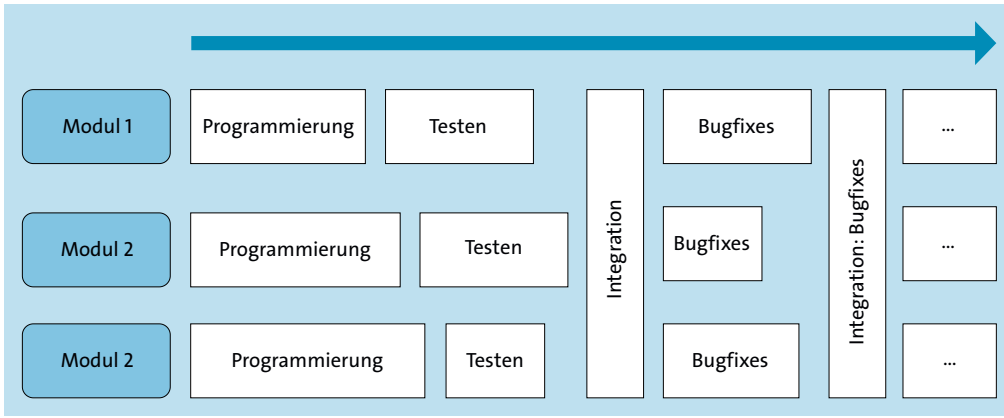

Abbildung 6.2 Typische Probleme im Build-Prozess: Die Module werden getrennt voneinander entwickelt und es kommt immer wieder zu Wartezeiten zwischen den Integrationsphasen.

Die Teams arbeiten, wie wir festgestellt haben, alle in getrennten Repositorys und damit getrennt voneinander. Die Build-Prozesse, um das jeweilige Projekt zu bauen, haben sich daher über die Jahre unterschiedlich entwickelt und sind daher nicht sonderlich effizient. Warum nicht effizient? Jedes Team und jedes Teammitglied baut während der sechsmonatigen Entwicklungszyklen jeweils ausschließlich seinen Teil selbst. Erst am Ende werden alle Teile zusammengefügt, wo dann im großen Stil integriert wird. Dabei sind Konflikte natürlich »vorprogrammiert«, weil es immer wieder zu Schnittstellenänderungen zwischen den Modulen kommt, die nicht kommuniziert wurden.

Wie man aus Abbildung 6.2 erkennen kann, laufen die Entwicklungsphasen für die einzelnen Module nie parallel. Der hier abgebildete Fall ist sogar noch harmlos: Die Programmierphase und die Testphase haben zwar unterschiedliche Längen und es kommt auch zu Wartezeiten bis zur Integration, aber nach der Integration erfolgen lediglich Bugfixes und dann die Integration der Bugfixes.

Das klingt schon kompliziert, und das ist es auch. Häufig kommt es dann zu etlichen Fehlern und Konflikten in der Integrationsphase, sodass gar nicht richtig integriert werden kann, was den ganzen Prozess verzögert.

Idealerweise geht man solche Probleme also frühzeitig an, sodass früh klar ist, ob die Änderung, die man in der Entwicklungsphase tätigt, sich nahtlos integrieren lässt und nicht zu Problemen in den anderen Modulen führt.

Im weiteren Verlauf dieses Kapitels werden wir uns anschauen, wie sich ein solches Big-Bang-Release entschärfen lässt.

© Rheinwerk Verlag, Bonn 2024

6.1.1 Onboarding mit Stolpersteinen

Die Schwierigkeiten beim Build sind allerdings viel häufiger nachgelagerte Probleme, die erst später im Entwicklungsprozess auftauchen. Die Probleme mit den unterschiedlichen Builds starten schon sehr viel früher. Nämlich dann, wenn jemand Neues im Team anfängt und eingearbeitet werden muss.

Eine der ersten Aufgaben für den Frischling bei schick-gekleidet.de, direkt nach der Installation des Betriebssystems, ist es also, die Entwicklungsumgebung für das Projekt einzurichten. Das ist bei einzelnen Teams mal besser, mal schlechter gelöst, aber bei allen fehlt Dokumentation, sodass viel händisch ausprobiert werden muss. Keine Entwicklungsumgebung ist genau gleich, man kann also bislang nicht einmal von einem Team zum nächsten wechseln, ohne dass Hand angelegt werden muss.

Ein konkretes Problem über alle Teams hinweg ist, dass die Abhängigkeiten, die für das Bauen des Projektes gebraucht werden, weder dokumentiert sind, noch gibt es irgendwelche Automatismen, um die Einrichtung zu beschleunigen. Eine neue Mitarbeiterin geht während des Onboardings also hin und lädt sich eine Arbeitskopie des Projekts aus dem Repository herunter.

Um ein Gefühl für die Entwicklung der Software zu bekommen, wollte sie das Projekt zunächst einmal selbst bauen. Erst wenn das geschehen ist, kann die neue Mitarbeiterin anfangen, an den ersten kleinen Bugfixes zu arbeiten, bevor sie sich auf komplexe Features stürzt, die erfahrungsgemäß deutlich länger dauern. Solche *low hanging fruits*, die wenig Arbeit erfordern, aber den ganzen Integrationsprozess einmal durchgehen, sind für den Einstieg in eine neue Code-Basis ideal, da sie nicht zu viel Zeit und Aufwand bedeuten.

Hier stößt die neue Mitarbeiterin allerdings auch schon auf das erste Problem: Ohne Dokumentation oder eine progammiersprachenübliche Automatisierung lässt sich das Projekt nicht bauen, da viele Abhängigkeiten gebraucht werden, die manuell auf dem Entwicklungsrechner in die richtigen Verzeichnisse kopiert werden müssen. Hier stellt sich ihr die Frage: Welche Abhängigkeiten in welcher Version werden gebraucht?

Ohne einen guten Build-Prozess bleibt dem neuen Teammitglied nur übrig, das Build-Skript auszuführen, auf den Abbruch zu warten und den Abhängigkeiten hinterherzulaufen. Da diese auch nicht im Repository des Projekts hinterlegt sind, müssen sie zwangsläufig aus dem Internet herunterladen und im richtigen Verzeichnis ablegt werden – die Version wird hoffentlich schon passen. Der Build muss dann noch mal gestartet werden, bis die nächste Fehlermeldung aufploppt – und diese Abhängigkeit muss wieder gesucht, gefunden, heruntergeladen und korrekt abgelegt werden …

Persönliches Exemplar für Thomas Bartholomäus

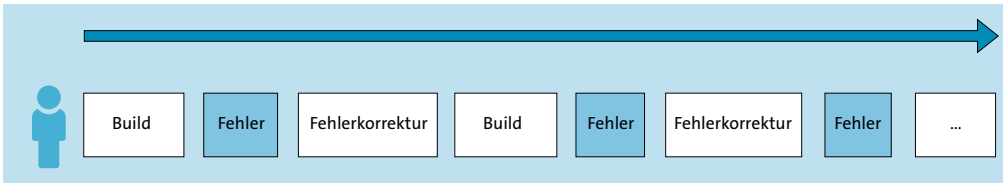

Abbildung 6.3 Build-Error: Ständig kommt es zu Fehlern, die manuell korrigiert werden müssen.

Dieser Zyklus (siehe Abbildung 6.3) muss immer wieder wiederholt werden, bis sich das Projekt problemlos durchbauen lässt. Das ist offensichtlich nicht nur zeitintensiv, sondern hat auch wenig Mehrwert und ist ziemlich mühselig.

Andere Teams hatten das gleiche Problem. Aber dort ist man schon ein kleines Stück weiter. Als Lösung des oben beschriebenen Problems erstellte man einfach eine virtuelle Maschine mit allen Entwicklungstools und -abhängigkeiten, die sich alle kopieren konnten, um schneller loslegen zu können.

Das Problem ist aber auch hier: Das wurde einmalig gemacht, und zwar händisch. Abhängigkeiten, die nicht mehr gebraucht oder aktualisiert wurden, werden nicht nachgezogen. Was also eigentlich eine einigermaßen gute Idee war, verlangsamte in der Praxis den Entwicklungsprozess nur noch mehr, da alle Teammitglieder mit dieser VM hantieren müssen, die deutlich langsamer als der eigentliche Arbeitsplatzrechner ist.

Die Probleme mit dem Build wiederholen sich jedes Mal, wenn eine Abhängigkeit aktualisiert oder ersetzt wird, weil die anderen im Team das nicht mitbekommen haben. Außerdem ziehen sich die Probleme dann ebenfalls durch alle nachgelagerten Teams durch, da auch das QA-Team oder das Betriebsteam den gleichen Prozess ebenfalls durchlaufen müssen, nur mit einem deutlichen zeitlichen Verzug, sodass dann gleich mehrere dieser Probleme gleichzeitig auftauchen.

Es wird Sie nicht überraschen, dass ein solch anstrengender Build-Prozess dazu führt, dass möglichst wenig und selten gebaut wird. Man testet nur den eigenen Code und lässt die Finger von anderen Projekten oder gar dem Shop als Ganzem, denn wer weiß, was einem bei diesen Builds mal wieder um die Ohren fliegt.

Das führt zwangsläufig dazu, dass niemand einen Überblick über den ganzen Build-Prozess hat: eine klassische Silo-Bildung. Die Kenntnisse der Mitarbeiter beschränken sich vollständig auf die eigenen Projekte, sodass sodass niemand mehr versteht, wie der Online-Shop als Ganzes programmiert ist.

6.1.2 Seltene Integrationen führen zu Build-Schwierigkeiten

Jedes Subteam des Online-Shops arbeitet also mit seinen eigenen umständlichen Build-Schritten. Entsprechend sind auch die Tickets bei *schick-gekleidet.de* gestaltet:

© Rheinwerk Verlag, Bonn 2024

umständlich und zu lang. Mehrere Änderungswünsche und Feature Requests werden in ein Ticket gepackt, sodass es sehr lange dauert, bis ein Ticket abgearbeitet ist.

In der Praxis sieht es also so aus, dass ein Entwickler über einen Zeitraum von mehreren Wochen am Feature arbeitet und dann im Team meldet, dass er fertig ist. Vor dem finalen Einchecken des Codes in das Repository lädt er sich allerdings die Änderungen vom Rest des Teams herunter, die in der Zwischenzeit implementiert wurden.

Das sorgt natürlich für Merge-Konflikte, da mehrere Personen gleichzeitig am gleichen Code gearbeitet haben. Diese Konflikte müssen vor dem Check-in aufgelöst werden, wofür viel kommuniziert werden muss, bis alle Unklarheiten beseitigt sind. Niemand kann noch auf Anhieb die Konflikte überblicken, die durch die schiere Größe der Änderungen entstehen, die in der Zwischenzeit aufgelaufen sind (siehe Abbildung 6.4).

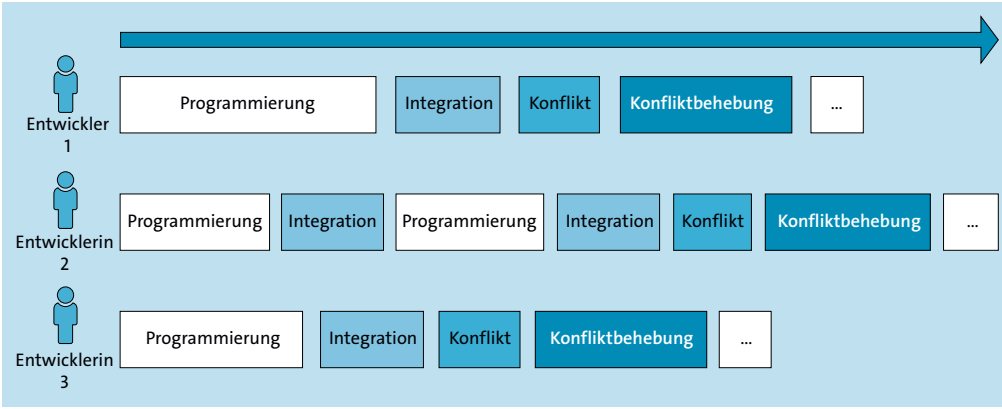

Abbildung 6.4 Not Continuous Integration: Bei (fast) jedem Integrationsversuch kommt es zu Konflikten, die wiederum zu langen Konfliktbehebungen führen.

Für die beteiligten Personen ist das immer frustrierend, denn die Änderungen im Repository einzupflegen, ist sehr fehleranfällig und mühselig. Zu den Merge-Konflikten gesellt sich immer wieder zwischenmenschlicher Ärger, weil viele einschneidende Änderungen nicht klar kommuniziert worden sind, obwohl diese grundsätzlich bekannt waren.

Manchmal dauert die Auflösung der Konflikte dann Tage, und die Reimplementierung der Änderung verschlingt genauso viel Zeit und Energie wie die ursprüngliche Arbeit. Und damit nicht genug: Nachdem die Konflikte im Quellcode aufgelöst wurden, wird der Code in das Repository gepusht und ist somit für die anderen im Team verfügbar, die dann wieder auf ähnliche Probleme stoßen, wenn sie die Änderungen pullen: ein Teufelskreis.

Schließlich warten also alle stets ab, wenn die neuesten Änderungen integriert werden sollen. Oder jemand kündigt an, dass bald eine größere Änderung gepusht wer-

Persönliches Exemplar für Thomas Bartholomäus

den soll, mit dem Effekt, dass die übrigen Entwickler ihre Arbeit einstellen, um abzuwarten, bis die anderen Integrationen durch sind. Schließlich sind das nur noch mehr Wartezeiten und Ineffizienzen. Kleine Korrekturen wurden so auch vermieden, da die Integration eben zeitaufwendig war.

Und das ist nur die Mühsal eines einzelnen Teams! Die Probleme skalieren sich weiter hoch und werden umso komplexer, je mehr Änderungen betroffen sind – also wenn die einzelnen Module integriert werden müssen. Die Zusammenarbeit ist aber eigentlich für das gesamte Projektteam notwendig, damit der ganze Shop erfolgreich gebaut werden kann. Mit diesem Problem befassen wir uns in Kapitel 8, wo es um das Release-Management geht.

6.1.3 Nur wenige Tests

Zur Aufgabe des Entwicklungsteams bei *schick-gekleidet.de* gehört auch die Entwicklung von Tests. Oder besser gesagt: Einige Teammitglieder schreiben Tests für einige Teile ihrer Implementierungen, wenn sie einen Bedarf gesehen haben. Das war insbesondere dann der Fall, wenn etwas zwar implementiert wurde, es aber nicht auf Anhieb funktioniert hat. Das Debugging ist dann einfacher, wenn die Teammitglieder einen kleinen Testfall schreiben konnten.

Dieses Vorgehen wurde allerdings nie stringent durchgeführt. So gibt es zwar einige Unit-Tests, die Methoden einer Klasse getestet haben, allerdings werden diese nie in Gänze ausgeführt. Das führt dazu, dass immer wieder Fehler bei den Kunden erscheinen, obwohl diese Bugs eigentlich hätten gefunden werden können, wenn die Tests gepflegt worden wären.

Warum werden die Tests so stiefmütterlich behandelt? Bei *schick-gekleidet.de* hat das erneut viel mit den hohen Mauern zwischen den Teams zu tun. Das Entwicklungsteam ist überzeugt, dass das Testen Aufgabe des Qualitätssicherungsteams sei: »Schließlich sind die doch dafür eingestellt worden, die Projekte zu testen.« Diese Sichtweise und die Probleme und Herausforderungen, die sich aus ihr ergeben, werden näher in Kapitel 7 betrachtet.

6.1.4 Ein Build-Server hinter verschlossenen Türen

Damit geprüft werden kann, ob das Projekt auch grundsätzlich gebaut wird, hat sich jedes Team bei *schick-gekleidet.de* einen Server hingestellt, auf dem jede Nacht der Build der eigenen Projekte durchlaufen soll. Dieser Server steht unter dem Tisch eines Mitarbeiters und wird händisch verwaltet.

Auch hier ist die Grundidee lobenswert: Das Team soll schnell und direkt Feedback bekommen, wie der Stand des Projekts ist, also ob sich das Projekt erfolgreich bauen lässt und ob die definierten Unit-Tests durchlaufen.

© Rheinwerk Verlag, Bonn 2024

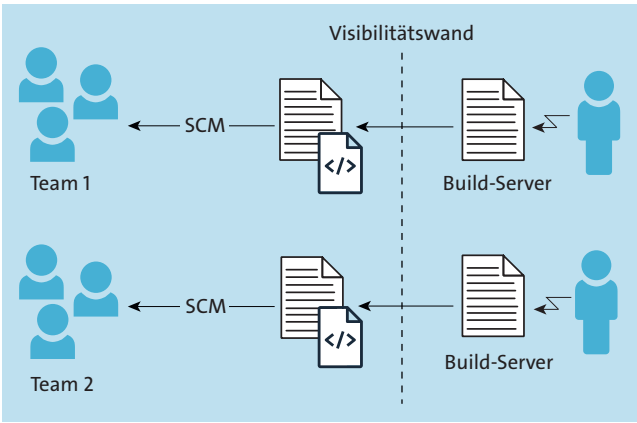

Abbildung 6.5 Kein Einblick in den Build-Server: Es gibt zwar für die jeweiligen Teams Build-Server, doch hat meist nur eine Person Zugriff auf den Server. Bei Fehlern gibt es so keine einfache Möglichkeit, den Fehler zu untersuchen.

Die Umsetzung ist allerdings mangelhaft: Nach jeder größeren Änderung, die etwa die Version einer Abhängigkeit verändert oder sonst tiefer in den Code eingearbeitet wird, bricht der Build. Zwar ist das genauso gewünscht, denn man will doch genau auf diese Probleme hingewiesen werden. Es nutzt aber nichts, wenn niemand Zugriff auf den Build-Server hat und somit niemand überprüft, wie es um den Build steht.

Und zudem ist dieser fehlschlagende Build ja kein gravierendes Problem, sondern nur eine fehlende Abhängigkeit, die einfach gepflegt werden müsste. Da der Rechner aber unter dem Tisch eines bestimmten Mitarbeiters steht, der damit auch für die Einrichtung und die Pflege des Build-Servers verantwortlich ist, fallen diese Anpassungen im Alltagsgeschäft »hinten runter«.

Organisatorisch hat es also gleich mehrere Gründe, warum der Build bei *schick-gekleidet.de* regelmäßig fehlschlägt: Die Handhabung ist zu umständlich, und es gibt keine einfache Möglichkeit für die Teammitglieder, die Fehlschläge zu sehen und zu begutachten. Daher hat es sich eingebürgert, diese Probleme einfach zu ignorieren. Und dies betrifft nicht nur den reinen Build, sondern auch Unit-Tests, die nie nachgezogen und angepasst werden.

Als Folge liegt dann auch noch die Arbeit für die anderen Arbeitskollegen im Team brach, da diese die Probleme im Laufe der Zeit auch bei ihnen auftreten, sobald sie nach einigen Wochen ihre Arbeit an ihrem Ticket in das Repository integrieren wollen.

Neben diesen organisatorischen Problemen gibt es auch technische Unzulänglichkeiten. Die Abhängigkeiten werden manuell installiert, und das nur von den wenigen Personen, die auf den Build-Server Zugriff haben. Der Rest des Teams ist darüber eigentlich gar nicht unglücklich, weil man so weniger Verantwortung trägt und somit bei fehlgeschlagenen Builds nicht schuld gewesen sein kann – schließlich hat

Persönliches Exemplar für Thomas Bartholomäus

man ja keinen Zugang zu diesem System und schließlich hätte jemand anderes die aktualisierte Abhängigkeit dort installieren können.

Und zu guter Letzt ist es auch problematisch, dass ein eigenes Script geschrieben wurde, das das Projekt baut. Dort werden viele Abhängigkeiten aus fest verdrahteten Verzeichnissen kopiert, deren Inhalte ebenfalls händisch verwaltet wurden.

Aber auch damit nicht genug, denn hier geht es nur um die Herausforderung, dass ein Team ein einziges Modul baut. Der Online-Shop bei *schick-gekleidet.de* selbst benötigte noch einen weiteren Server, der ähnlich eingerichtet wurde. Dieser hat die Aufgabe, die Ergebnisse der einzelnen Subteams zu empfangen, um dann den ganzen Online-Shop zu bauen.

Dieses Setup ist zwar in der Theorie eingerichtet, wurde in der Praxis allerdings nie funktionsfähig: Da die einzelnen Module weitestgehend nie baubar waren, ließ sich das ganze Projekt auch nicht bauen. Da die einzelnen Teams schon Schwierigkeiten haben, ihren eigenen Code zeitnah und möglichst fehlerfrei zu integrieren, klappt das mit sich ändernden Schnittstellen im Gesamtprojekt noch seltener.

Das Resultat ist dann, dass Deployments noch schwieriger werden, weil alle Integrationen noch einmal geradegerückt werden mussten. An etlichen Stellen kommt es also zu Bottlenecks bei der Implementierung – vorwiegend dann, wenn ein Release des Online-Shops ansteht. Die Fehler früh im Prozess führen zu exponentiellen Problemen in den nachfolgenden Stages.

Reflexion

Eines der hier genannten Probleme ist häufig in Firmen zu sehen, lässt sich aber ohne eine große Änderung der Kultur beheben: das Geraderücken des Build-Prozesses, insbesondere für den Onboarding-Prozess.

Prüfen Sie, ob sich Ihre Projekte ohne Stolpersteine bauen lassen, und schauen Sie, ob es nicht noch mehr Aspekte gibt, die störend sind und keinen echten Mehrwert bringen.

6.2 Modernes Build-Management

Das war nun eine ziemlich lange Beschreibung des Ist-Zustands bei *schick-gekleidet.de*, voller Fallstricke und Probleme. Wie geht es besser?

Der größte Nutzen der DevOps-Transformation für das Build-Management liegt in der Automatisierung. Ihnen wird aufgefallen sein, dass sich die meisten Probleme auf einen Punkt zurückführen lassen: zu viel Handarbeit. Das bedeutet: zu viele Einzelschritte, die Abstimmung und Koordination erfordern.

© Rheinwerk Verlag, Bonn 2024

Zu oft kam es zu Komplikationen beim Mergen von größeren Änderungen, was dazu führte, dass sich die Projekte nicht mehr bauen ließen. Zu oft warteten diverse Teammitglieder auf die Fertigstellung des Features einer anderen Person.

Das Ziel muss also sein, den kompletten Build zu verschlanken und möglichst viele Arbeitsschritte zu standardisieren und dann zu automatisieren. Das Ziel ist im ersten Weg der *The Three Ways* beschrieben: Das ganze System muss in den Blick genommen werden. Es geht nicht darum, irgendwie ein Feature zu programmieren, das dann händisch in den Code des Teams integriert wird. Der eigentliche Punkt ist, dass die Verbesserung oder die Korrektur möglichst schnell bei den Endnutzern landet.

Das geht nur, indem der Prozess so optimiert wird, dass die Änderungen schneller und mit wenigen Fehlern im Build landen. Außerdem sollte auch das gesamte System verstanden werden, was bisher kaum möglich war, da alle Build-Prozesse sowohl unterschiedlich als auch schlecht dokumentiert waren.

Befassen wir uns aber zunächst mit der Ursache der Probleme, nämlich mit dem Verständnis, was eine *Änderung* genau ist. Dafür müssen Sie abermals einen Schritt zurücktreten und sich anschauen, wie die Projektplanung eines neuen Features oder eines Bugfixes überhaupt aussieht.

Bei *schick-gekleidet.de* wurde der ganze Auftrag in ein Ticket gepackt; die Aufgabe war zwangsläufig sehr, sehr groß. Eine Implementierung dauerte in vielen Fällen Wochen, wenn nicht sogar Monate und produzierte Hunderte oder Tausende Codezeilen, die sich nur unter großen Schmerzen in den Entwicklungsbranch integrieren lassen. Das galt sowohl für die Entwickler selbst als auch für die weiteren Kolleginnen und Kollegen im Team, die von diesen Änderungen abhängig waren. Das Thema hatten wir ja im vorherigen Kapitel bereits angesprochen, und hier gehen wir nun einen Schritt weiter.

Die bisherigen Tickets waren so gesehen also eher sogenannte *Epics*, in denen mehrere unterschiedliche Schritte in neuen Tickets erfasst werden, die abgearbeitet werden müssen, bis das ganze Feature fertig ist. Für die grundsätzliche Orientierung ist das nützlich, und es hilft auch, Abhängigkeiten innerhalb des neuen Features zu verstehen – für die eigentliche Umsetzung ist es jedoch Gift.

Stattdessen: Halten Sie die Tickets, die letztlich bei den Mitgliedern des Entwicklerteams ankommen, möglichst klein. Es sollte auch klar definiert sein, wann die Arbeit daran abgeschlossen ist. Sowohl eine Entwicklerin als auch eine Projektleiterin, die den Inhalt des Tickets liest, sollte also idealerweise ohne weitere Nachfragen verstehen können, was damit gelöst werden soll. Je kleiner die einzelnen Schritte sind, desto eher fallen schon frühzeitig Planungsfehler auf, denn erst beim detaillierten Aufschreiben der benötigten Schritte fallen Lücken auf (siehe Abbildung 6.6).

Wenn nun die Arbeit an einem Ticket gestartet wird, sollte die Änderung, die getätigt wurde, möglichst zeitnah in den Hauptentwicklungsbranch einfließen. Idealerweise

geschieht das mindestens einmal täglich, wenngleich die Arbeit am Feature nicht komplett abgeschlossen ist. Wichtig ist allerdings, dass das Projekt immer baubar bleiben sollte: sowohl für einen selbst als auch für die weiteren Mitglieder im Team. Dazu gehört auch die erfolgreiche Ausführung der automatisierten Tests.

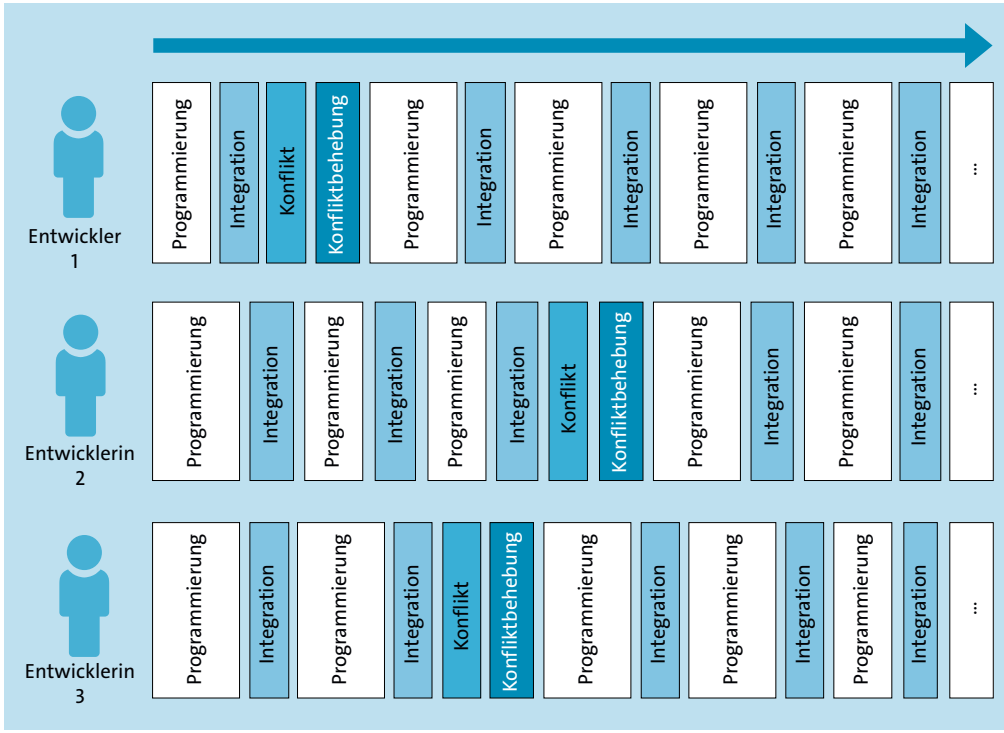

Abbildung 6.6 Je kleiner die Arbeitspakete sind, desto kleiner sind auch die Programmierpakete und desto schneller sind diese Teile – möglichst mit wenig Konflikten – integriert.

Breaking Changes

Breaking Changes sind immer eine große Herausforderung. Sie erfordern ein hohes Maß an Koordination und Kollaboration, was komplexer wird, je mehr Abhängigkeiten beachtet werden müssen. Eine Möglichkeit zur Kommunikation von solchen Änderungen ist ein Versionierungsschema wie *Semantic Versioning*, das ich in Abschnitt 8.3 vorstelle.

Durch dieses Verfahren werden auch Code-Reviews, die in Abschnitt 5.4 thematisiert wurden, deutlich erleichtert: Kleinere, überschaubare Änderungen lassen sich leichter nachvollziehen und somit zügiger reviewen. So kommt es deutlich seltener zu einer *Merge-Hell*, in der viele verschiedene riesige Änderungen aufwendig integriert werden müssen.

© Rheinwerk Verlag, Bonn 2024

Vergessen Sie nie das Ziel des Ganzen: die zügige Bereitstellung der Verbesserungen für die Endnutzer. Die soeben kurz erläuterten Aspekte sind alles Aspekte, die unter den Begriff *Continuous Integration* fallen, was im Folgenden noch genauer betrachtet wird.

Reflexion

Ich wage mal zu behaupten, dass für den überwiegenden Teil meiner Leserinnen und Leser »modernes« Build-Management gar nicht mehr so neu und modern ist. Zwar sehe ich immer wieder, dass es einige Unternehmen gibt, in denen das nicht der Fall ist. Es lässt sich aber schon sagen, dass »modernes« Build-Management inzwischen Usus ist.

6.3 Continuous Integration

Dass der Build-Prozess so schwerfällig ist, liegt also hauptsächlich an den großen, unhandlichen und fehleranfälligen Code-Monstern, die in großen Abständen integriert werden müssen. Besser ist es, kleinere Änderungen zu haben, die stetig und in kürzeren Abständen integriert werden können. Dieses kontinuierliche Integrieren, engl. *Continuous Integration*, sorgt dafür, dass das der gesamte Entwicklungsprozess effizienter abläuft. Wie der Name schon verrät, geht es darum, kontinuierlich den Code in das Produkt zu integrieren. Fehler im Produkt und andere Unzulänglichkeiten fallen dadurch tendenziell früher auf.

Die Probleme des Build-Vorgangs bei *schick-gekleidet.de* haben Ihnen die Vorteile von Continuous Integration (CI) bereits verdeutlicht. Continuous Integration geht übrigens auch ohne DevOps, aber DevOps ist ohne die Vorteile von Continuous Integration nicht denkbar.

Continuous Everything

Continuous Integration (CI) wird in der Regel im Zusammenhang mit *Continuous Delivery* erwähnt, sodass von *CI/CD* die Rede ist: Continuous Integration, gefolgt von Continuous Delivery. Den Begriff *Continuous Deployment* gibt es auch noch, der aber nicht das Gleiche wie Continuous Delivery bezeichnet (aber dazu später mehr). Jetzt geht es zunächst einmal um die reine Integration.

Die Umsetzung von Continuous Integration setzt sich aus einem technischen und einem organisatorischen Teil zusammen. Zum einen muss die Kultur der kleinen, agilen Häppchen stimmen, es muss aber auch die notwendige Automatisierung vorhanden sein, um Änderungen problemlos zu verarbeiten.

Persönliches Exemplar für Thomas Bartholomäus

Die Probleme bei *schick-gekleidet.de* entstanden primär daraus, dass der Code der Entwickler viel zu spät integriert wurde, was zu großen Konflikten bei der Integration führte. Folglich war das Projekt kaum zu bauen. Weil auf einem Server gebaut wurde, auf den nur wenige Leute Zugriff haben und der die meiste Zeit ignoriert wurde, war es selten der Fall, dass das Projekt ohne Probleme gebaut werden konnte.

Im Rahmen der DevOps-Transformation geht es nun also darum, möglichst viele dieser Einschränkungen aufzuräumen. Die Technik hilft dabei, die Kultur zu verbessern.

Die erste Regel, die es zu beachten gilt, lautet, dass der Build des Hauptentwicklungsbranches immer »grün« sein sollte: Der letzte Build sollte erfolgreich abgeschlossen worden sein, was bedeutet, dass es auch möglich sein sollte, das Projekt erneut zu bauen. Hieraus leiten sich zwei Nachfolgefragen ab:

1. Wie ermöglicht man die Sichtbarkeit des Build-Prozesses?
2. Wie stellt man sicher, dass nur Code integriert wird, der sich bauen lässt?

Der wichtigste Punkt beim Umstieg auf Continuous Integration ist die Erhöhung der Sichtbarkeit von möglichen Problemen. Ein einfacher Workaround für die Situation bei *schick-gekleidet.de* wäre daher, dass die Zugänge zum bisherigen Build-Server automatisch erstellt werden, sobald neue Leute zum Team stoßen. Jetzt können zwar alle den Build-Status sehen, aber das bringt zunächst noch nichts, solange sich niemand um die Probleme kümmert.

Der Idealzustand wäre also, eine maximale Transparenz innerhalb der Organisation herzustellen. Das Mindeste ist jedoch die maximale Transparenz innerhalb des Teams: Alle Teammitglieder müssen dafür sensibilisiert sein, dass ein fehlgeschlagener Build zu weiteren Problemen in der nahen Zukunft führen wird und sofort behoben werden muss.

Das Integrieren der Änderungen und das Bauen der Software dürfen daher keine Anhängsel sein, die nach der eigentlichen Arbeit folgen, sondern müssen ein fester Teil des Tagesablaufs sein.

Abhilfe schafft ein CI-Server, der genau dafür gebaut ist. Heutzutage werden für diesen Zweck hauptsächlich *GitLab CI* oder *GitHub Actions* genutzt, da sie sich nahtlos in das Sourcecode-Management der jeweiligen Plattformen integrieren. Eine große Verbreitung hat aber ebenfalls noch *Jenkins*. (Die genauen Vor- und Nachteile der einzelnen CI-Server-Lösungen und was es dabei sonst zu beachten gibt, um möglichst effizient zu arbeiten, werden in Abschnitt 6.4 näher betrachtet.)

Vereinfacht gesagt, führt der CI-Server definierte Aktionen aus, die durch Trigger ausgelöst werden. Ein Trigger kann, wie im Beispiel von *schick-gekleidet.de*, ein regelmäßiger Build sein, der täglich zu einer bestimmten Zeit ausgeführt wird. Häufig wird das *Daily* oder *Nightly Build* genannt.

© Rheinwerk Verlag, Bonn 2024

Problematisch in diesem konkreten Beispiel ist allerdings, dass nur alle Änderungen, die über den Tag verteilt integriert wurden, gemeinsam gebaut wurden. Wenn hier also Fehler auftreten, die es im Build davor bislang nicht gab, dann ist nicht direkt ersichtlich, welche Änderung von welcher Person zu diesem Fehler führte.

Das Konzept von Nightly Builds ist daher in der Regel überholt. Stattdessen sollte die Pipeline so konfiguriert sein, dass bei jedem Commit auf jedem Branch ein Build ausgeführt wird. Die Skripte, die für den Build (aber nicht ausschließlich für ihn) ausgeführt werden, sind Teil der Pipeline.

Unter *Pipeline* versteht man die Zusammensetzung mehrerer Aktionen, die nacheinander ausgeführt werden, um das Projekt zu bauen, die verschiedenen Tests auszuführen sowie um ein Deployment auf eine Umgebung auszuführen (siehe Abbildung 6.7). Eine grüne Pipeline ist eine gute Pipeline, denn dann waren alle Schritte erfolgreich. Wenn ein Schritt fehlschlägt, werden nachfolgende Jobs aus der Pipeline nicht weiter ausgeführt.

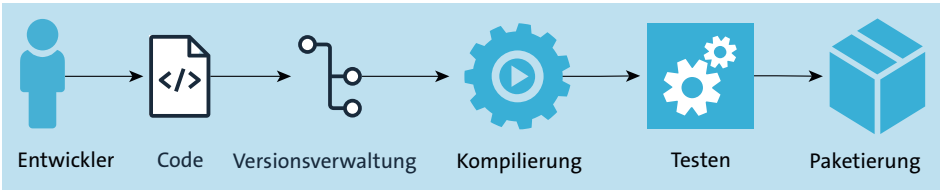

Abbildung 6.7 Die Continuous-Integration-Pipeline besteht aus der Versionsverwaltung, der Kompilierung des Projekts, dem Testen und der Paketierung.

Pipelines sollten so definiert werden, dass sie reproduzierbar sind. Mit einem modernen CI-Server ist das kein Problem, da auf ihm eingebaute Mechanismen verfügbar sind, die das Ganze vereinfachen. Dazu gehören saubere Umgebungen, in denen die Projekte ohne lokale Abhängigkeiten gebaut werden können und in denen alle Daten genutzt werden, die im Repository des Projekts liegen.

Bei *schick-gekleidet.de* gab es einige Probleme, die es zu korrigieren galt. Ganz vorn steht die Verwaltung der Abhängigkeiten. Diese wurden bisher händisch auf dem Server verarbeitet. Das ist ein generelles Problem, wenn das Projekt sich nicht direkt aus dem Quellcode bauen lässt.

Aber was heißt das nun genau? Alles, was im Repository definiert ist, sollte genügen, um das Projekt zu bauen. Nutzen Sie dafür die Build-Werkzeuge und die Paketmanager der eingesetzten Programmiersprachen, um stabile und reproduzierbare Builds zu vereinfachen.

Bei Java ist das entweder *Maven* oder *Gradle*. Mit beiden lassen sich in Konfigurationsdateien alle Abhängigkeiten definieren, die zum Bauen und zum Laufenlassen des Dienstes gebraucht werden, sodass sowohl Maven als auch Gradle die Abhängig-

Persönliches Exemplar für Thomas Bartholomäus

keiten verwalten können. Das sollte für einen Großteil derer, die das hier lesen, keine Neuigkeit, sondern Usus sein.

Die benötigten Abhängigkeiten werden automatisch von der Paketverwaltung von einer zentralen Stelle gezogen. Das betrifft nicht nur öffentliche Abhängigkeiten, die meistens unter einer Open-Source-Lizenz stehen, sondern auch interne Abhängigkeiten, die über ein eigenes Paket-Repository bereitgestellt werden sollten.

Wenn also eine neue Version einer Abhängigkeit genutzt wird, dann muss neben den Änderungen, die im Code benötigt werden, lediglich die Konfigurationsdatei angepasst werden, damit im nächsten Build die aktualisierte neue Abhängigkeit gezogen wird. Ein manuelles Einloggen auf einem pseudeo-zentralen Server ist dann nicht notwendig. Gleichzeitig hilft das natürlich auch beim Setup für die lokale Entwicklungsumgebung.

Kein alter Hut

Wenn Sie bereits länger professionell in der Software-Entwicklung arbeiten, werden Sie wissen, dass eine gute und automatisierte Verwaltung der Abhängigkeiten ein zentraler Faktor für gute Builds ist – das ist eine etablierte Best Practice. Es ist aber leider so, dass diese Best Practice natürliche Feinde hat: Komplexität und Ausnahmen.

Wenn ein Build so unübersichtlich wird, dass die Abhängigkeiten und das Konfigurationsmanagement aus dem Ruder laufen, fällt man gerne auf eigentlich überwundene »Workarounds« zurück. Steuern Sie dort möglichst schnell und entschieden gegen! Es sollte immer der Standard sein, dass alle Abhängigkeiten direkt im gleichen Repository liegen oder dass sie aufgrund einer Automatisierung durch den Build gezogen werden. Alles andere führt nur dazu, dass die Prozesse verlangsamt werden und die Entwickler unproduktiver sind, als es nötig wäre.

Wichtig ist zudem, dass die vollständige Automatisierung des Prozesses nicht nur durchgezogen wird, sondern auch sichtbar gemacht wird. Und sobald Fehler auftreten, sollten diese natürlich auch korrigiert werden.

Und genau das ist auch der nächste Punkt, den es sich lohnt anzuschauen, und auch die zweite Frage, die weiter oben genannt wurde: Wie stellt man sicher, dass nur baubarer Code integriert wird?

Eine gängige Praxis ist, dass man die Änderungen nicht unkontrolliert in den Hauptentwicklungsbranch integriert, sondern dass mit kurzlebigen Feature-Branches gearbeitet wird. Diese sind nicht nur für das vereinfachte Code-Review sinnvoll, sondern helfen auch sicherzustellen, dass nur baubarer Code integriert wird.

Auch dabei hilft ein CI-Server. Diese Server machen es sehr einfach, den Build-Prozess für jeden Branch automatisch anzustoßen. Nach dem Pushen eines Feature-

© Rheinwerk Verlag, Bonn 2024

Branchs und dem Eröffnen eines Merge bzw. Pull Requests sollte die Pipeline automatisch starten, das Projekt bauen und die definierten Tests ausführen.

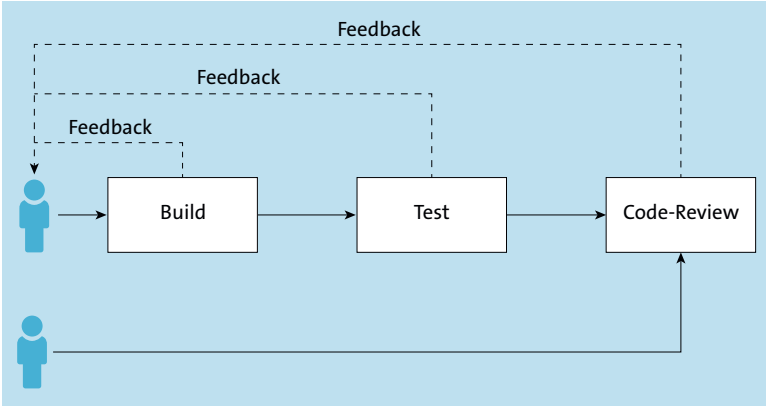

Abbildung 6.8 Wichtig ist, dass jeder Schritt (Build, Test und vor allem das Code-Review) Feedback liefert, um Verbesserungen zu implementieren.

Das ist der Schlüssel für das kontinuierliche Feedback, denn so kann man gut steuern, dass nur baubarer Code integriert wird. So lassen sich auch viel einfacher kleinere Änderungen beitragen, ohne dass es zu großen Wartezeiten kommt. Ein erster Schritt ist also, die Größe der Änderungen herunterzuschrauben, dann hilft das Tool den Prozess zu unterstützen, statt im Weg zu stehen.

Die Pipeline nicht überfrachten

Das Testen und die Qualitätssicherung selbst werden in Kapitel 7 näher behandelt und sind ebenfalls elementarer Bestandteil der CI/CD-Pipeline. Das Bauen und Testen des Projekts sollte möglichst zügig vonstattengehen, damit der Entwickler innerhalb von wenigen Minuten Feedback bekommt, ob die Änderungen nicht zu schwerwiegenden Fehlern führen. Nur so lässt sich die Lead-Time verkürzen.

Testen Sie allerdings nicht ohne Sinn und Verstand: Ich habe im Laufe meiner Karriere schon einige Pipelines gesehen (und musste selbst damit arbeiten), bei denen alles, was irgendwie automatisierbar war, in die Builds und Tests gesteckt wurde. Das führte dann zu einer Pipeline-Laufzeit von 4 Stunden. Das ist eindeutig zu lang.

Durch die Einführung der Automatisierung rund um den Build-Prozess wird letztlich auch die Zusammenarbeit spürbar verbessert. Wichtigster Punkt aus diesem Blickwinkel ist, dass die Probleme sofort sichtbar werden und direkt behoben werden können. Die Fehler landen nicht bei anderen Teammitgliedern, sondern fallen sofort auf, sodass man sich sofort um sie kümmern muss – und nicht die eine Teamkollegin, bei der der Bug nach drei Monaten auftaucht und die sich zunächst in den Kontext und die Zusammenhänge einarbeiten müsste.

Persönliches Exemplar für Thomas Bartholomäus

Ebenso fällt typisches Finger-Pointing bei Problemen weg, weil nun das Team sowohl den Code als auch den Build-Prozess verantwortet. Aber nicht nur die Entwickler im eigenen Team profitieren davon, sondern auch das Betriebsteam.

Häufige und regelmäßige Builds machen es deutlich wahrscheinlicher, dass das entscheidende Deployment in die Prod-Umgebung glattläuft. (Grundsätzlich wird alles, was das Deployment vereinfacht und sicherer macht, auf große Freude beim Betrieb stoßen.)

Und auch das Qualitätssicherungsteam darf nicht vergessen werden. Es hat zuvor in einem getrennten Repository gearbeitet und musste immer die Änderungen aufwendig nachziehen, was stets mit Workarounds und Ausnahmen einherging. Ein standardisierter, automatisierter Build-Prozess kommt den Testern also am stärksten zugute.

6.4 CI-Server und die Pipelines

Da nun klar ist, dass ein CI-Server mit einer entsprechenden Pipeline das Leben deutlich einfacher macht, geht es jetzt um einen Überblick über verschiedene CI-Server, die meistens nicht nur Continuous Integration ermöglichen, sondern auch Continuous Delivery. Erwarten Sie aber bitte keine ausführliche Pro- und Contra-Analyse, denn welches Werkzeug sich am besten für Ihre Aufgaben eignet, kann nur durch eine individuelle Evaluation festgestellt werden. Lassen Sie uns stattdessen den Blick auf die verschiedenen Konzepte und Ideen richten und schauen, wo vielleicht der eine oder andere Stolperstein liegt.

6.4.1 Der grundlegende Aufbau einer Pipeline

Im ersten Schritt sollten Sie sich genauer anschauen, was eine Pipeline überhaupt ist, woraus sie besteht und was die Kernkonzepte einer modernen Software-Pipeline sind. Erst im Schritt danach sehen wir uns an, wie Pipelines geschrieben werden können und welche CI-Server verbreitet sind.

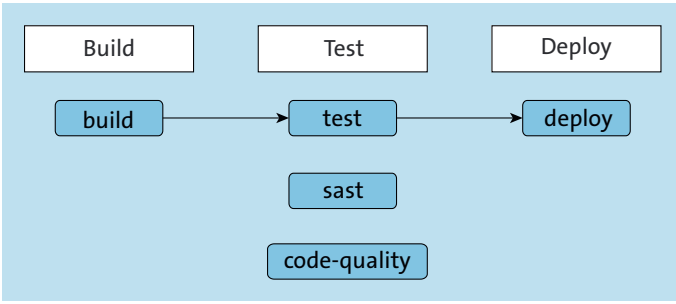

Abbildung 6.9 Der konzeptionelle Aufbau einer einfachen Pipeline

© Rheinwerk Verlag, Bonn 2024

Der grundlegende Aufbau ist bei jedem CI-Tool im Grunde gleich, auch wenn die Bezeichnungen vereinzelt unterschiedlich sein können: Es gibt *Stages* und es gibt *Jobs*. Jobs sind dabei immer Teil einer Stage. Eine Stage ähnelt in gewissem Sinne den DevOps-Stages: Diese gruppieren diverse Aktivitäten, die in der Pipeline ausgeführt werden.

Üblicherweise gibt es mindestens diese drei Stages: *Build*, *Test* und *Deploy* (siehe Abbildung 6.9). In jeder dieser Stages ist mindestens ein Job enthalten. Ein Job in der *Build-Stage* baut etwa das Projekt. Aus dem Build-Job fallen Artefakte heraus, die dann in der nächsten Stage benötigt werden, da dort je nach Programmiersprache und -umgebung mit dem Kompilat getestet werden muss.

Dies ist gleichzeitig eine Optimierung der Durchlaufgeschwindigkeit der gesamten Pipeline. Die Jobs sollten grundsätzlich so schnell wie möglich laufen, dementsprechend sollten die Jobs auch so geschrieben und konfiguriert sein. Im konkreten Beispiel heißt das, dass im Job `test` die Build-Artefakte aus dem Job `build` wiederverwendet werden. Wenn der Build-Job allerdings fehlschlägt, dann bricht die gesamte Pipeline ab und gibt früh Feedback.

Je nach Komplexität und Bedarf existieren in der *Test-Stage* einige Jobs, die die verschiedenen Tests ausführen, darunter Unit-Tests, Integrationstests oder diverse Security-Tests, wie sie auch in Abschnitt 11.4 vorgestellt werden. Typisch sind auch Tests für die Code-Qualität, um automatisiert die Einhaltung von Code-Layout zu prüfen.

In der dritten Stage, der *Deploy-Stage*, mit dem gleichnamigen Job, wird dann das Deployment durchgeführt. Dies kann automatisiert passieren, oder es wird erst nach einem manuellen Bestätigen durch berechtigte Personen ausgerollt.

Jeder Job benötigt dabei eine Laufzeitumgebung, in der alle Tools enthalten sind, die für die Absolvierung des Jobs gebraucht werden. Bei einem Java-Projekt, das mit Maven gebaut wird, wird also eine Umgebung gebraucht, in der Maven installiert ist. Früher wurden hierfür einfach auf einem System fest die entsprechend benötigten Abhängigkeiten installiert. Der Job konnte so immer wieder laufen, ohne dass man großartig etwas ändern musste. Eines der Probleme dabei ist, dass die Build-Umgebung für jeden Job dann nicht immer komplett sauber ist und die Abhängigkeiten immer manuell installiert werden müssen.

Heutzutage haben sich Software-Container für diese Aufgabe durchgesetzt, bei denen Container-Images genutzt werden, in denen alle Abhängigkeiten enthalten sind. Für viele Aufgaben und Anforderungen existieren bereits etliche Container-Images, so etwa für Maven. Im öffentlich zugänglichen DockerHub finden Sie beispielsweise eine Menge vorgefertigte Images, die teils von den Teams hinter den Projekten gepflegt werden, teils aus anderen Quellen stammen. Wenn das gewünschte Container-Images in der passenden Version und mit den richtigen Abhängigkeiten nicht vorhanden ist, ist es relativ einfach, bestehende Images zu ergänzen.

Persönliches Exemplar für Thomas Bartholomäus

Der Hintergrund dafür ist, dass alle Jobs reproduzierbar ausführbar sind, weil sie auf festen Container-Images beruhen. Ein Debuggen von Problemen, die nur in Umgebungen auftreten, wenn diese vorher nicht sauber aufgeräumt worden sind, muss heutzutage der Vergangenheit angehören. Alles andere ist ziemlich ineffizient. (Ausnahmen bestätigen natürlich die Regel.)

Ebenfalls ist es viel einfacher möglich, die Pipeline-Jobs zu skalieren, da klar definiert ist, welcher Container wo gebraucht wird. So kann eine Container-Plattform genutzt werden, um die Jobs verteilt und parallel bauen und testen zu lassen. Durch den Einsatz von vorgefertigten Images sind die Systeme auch deutlich einfacher automatisierbar, da nirgendwo händisch Umgebungen installiert und gewartet werden müssen, um eine Abhängigkeit zu nutzen. Stattdessen müssen eben die Container-Images verwaltet werden.

6.4.2 Skalierung und Reproduzierbarkeit

Moderne CI-Pipelines sorgen für Skalierung und Reproduzierbarkeit. Dies sind die beiden größten Vorteile, und sie sollten immer das Ziel sein.

Skalierung

Zunächst zur *Skalierung*: Damit ist gemeint, dass die Aufgabe, den Quellcode zu einem nutzbaren Artekfakt zu bauen, so organisiert wird, dass es keinen Unterschied macht, ob es sich um ein kleines Hello-World-Beispiel handelt oder um die gesamte Produktionsumgebung. Dies erreichen Sie, indem Sie die einzelnen Schritte, die dazu nötig sind, vollständig automatisieren. Das bedeutet auch, dass die Verwaltung der Infrastruktur rund um die CI/CD-Pipeline auf ein Minimum reduziert werden sollte.

Während es früher üblich war, dass es ein sehr großes Team gab, das die CI/CD-Infrastruktur betreute, ist genau dies heute nur noch in Organisationen der Fall, die nicht so ganz mit der Zeit gehen. Das betrifft dann nicht nur das Bauen der Pipeline selbst, sondern auch die Verwaltung der Infrastruktur, die nicht immer gut skaliert und daher einen hohen Engineering-Aufwand erfordert.

Je größer ein Unternehmen ist, desto eher wird es Teams geben, die sich nur um die Infrastruktur rund um CI/CD kümmern. Schauen Sie sich genau an, in welchem Maße diese Teams in den gesamten Prozess involviert sind und wie viele Leute in ihnen arbeiten. Die bloße Existenz eines solchen Teams würde ich per se nicht verteufeln, im Gegenteil: Mit den richtigen Kenntnissen können diese sehr hilfreich sein. Im Idealfall kann jedoch jedes Team für sein Projekt vorgefertigte Container-Images nutzen, die sich mit wenig Aufwand direkt aus den offiziellen Images erstellen lassen und die in einer firmeninternen Registry gespeichert werden. Einige Tools müssen vermutlich nachinstalliert werden, aber das kann dann direkt Teil des Pipe-

© Rheinwerk Verlag, Bonn 2024

line-Jobs sein; oder man stellt ein separates Container-Image bereit, das stets neu gebaut wird.

Container-Images lassen sich einfach skalieren, indem ein Orchestrierungswerkzeug wie *Kubernetes* genutzt wird. Das sorgt nicht nur für Stabilität und Ausfallsicherheit der Produktivumgebung, sondern ist auch für Build-Umgebungen praktisch, wenn eine automatisch Hoch- und Herunterskalierung konfiguriert werden kann, je nachdem, wie die Auslastung gerade ist. Nichts ist für einen Entwickler nerviger als eine Pipeline, die nicht läuft, weil die Infrastruktur nicht dafür bereit ist. Aber es ist auch nicht nötig, dass die Pipeline ständig Ressourcen frisst. Schauen Sie, dass Ihr Orchestrierungswerkzeug möglichst komplett durchautomatisiert ist, was durch die gängigen modernen CI-Tools schon gegeben ist bzw. durch diese stark vereinfacht wird.

Problematischer ist es, wenn keine Container-Technologie eingesetzt werden kann. Das ist etwa dann der Fall, wenn hardwarenah gearbeitet wird, etwa wenn Software direkt auf Embedded-Geräte installiert werden muss, um diese zu testen. Das lässt sich mit Containern nicht gut lösen.

Aber auch in diesem Fall sollte möglichst viel automatisiert werden – wenn es nicht anders geht, dann halt mit Bash-Skripten oder ähnlichen Werkzeugen, solange sie sich einfach warten lassen. Das Ziel ist es ja, dass die Teams, die CI/CD-Infrastruktur als Werkzeug verwenden, davon nicht sonderlich genervt werden, sondern einen Mehrwert erhalten. Dabei hilft die Autonomie, die die Teams durch das Werkzeug bekommen. Es soll *mit* dem Werkzeug gearbeitet werden und nicht *an* dem Werkzeug.

Statische Systeme, bei denen die Abhängigkeiten händisch installiert werden müssen, haben den wesentlichen Nachteil, dass diese wenig flexibel sind, um sie für neue oder andere Projekte einzusetzen. Wenn ein Team also mit einer neuen Technologie startet und dafür andere Build-Tools benötigt, kann es sich selbst um das Container-Image kümmern, das ebenfalls regelmäßig und automatisch neu gebaut werden kann, wenn es nötig ist.

Reproduzierbarkeit

Der zweite wichtige Punkt ist die *Reproduzierbarkeit*: Im Prinzip ist die logische Schlussfolgerung aus den vorherigen Abschnitten, dass das Projekt immer reproduzierbar gebaut werden kann.

Zunächst ist es wichtig, dass bei jedem Build und beim Test der Software immer die gleichen Ergebnisse geliefert werden, weswegen saubere Umgebungen mithilfe von Containern heute der Stand der Technik sind. Sie sorgen dafür, dass immer vom gleichen Ausgangspunkt gestartet wird und dass die Bedingungen für jeden Build gleich sind.

Diese Forderung lässt sich noch erweitern: Wenn der Sourcecode und die Build-Umgebung bekannt sind, sollte es möglich sein, dass auch jemand anders Ihre Anwendung bauen kann.

Persönliches Exemplar für Thomas Bartholomäus

Das hört sich zunächst selbstverständlich an, ist es aber keineswegs. Wenn Sie einmal versuchen, ein riesiges Open-Source-Projekt wie Firefox oder Libre Office aus den Quellen zu bauen, werden Sie feststellen, was dies für ein riesiger Aufwand ist. Und dabei ist meistens noch nicht einmal sicher, dass das Ergebnis, also »Ihr« Firefox, Bit für Bit wirklich dem entspricht, was Sie über die Installationsdateien herunterladen können.

Reproducible Builds

Das Projekt *https://reproducible-builds.org/* setzt sich dafür ein, dass offene Software reproduzierbar gebaut werden kann, da dies Vertrauen schafft. Es ist nämlich nur eine Sache, dass man den Quellcode des Browsers Firefox prüfen kann, um sicherzustellen, dass sich dort keine Backdoors verstecken. Wenn man aber nicht gleichzeitig sicher sagen kann, dass dieser Quellcode auch genau gleich zum genutzten Programm gebaut wird, hat man nicht viel gewonnen.

Schauen Sie sich die Hinweise und Überlegungen des Projekts gut an: Es ist eine Goldgrube für alle, die sich mit dem Build und Deployment von Software beschäftigen.

Das reproduzierbare Bauen von Anwendungen ist insbesondere in regulierten Umgebungen wichtig, um sicherzustellen, dass die Pipeline immer die gleiche Ausgabe liefert. Wenn ein Bugfix für einen älteren Stand ausgeliefert werden muss, dann sollte die Build-Infrastruktur das durch die Nutzung von Container-Images relativ schmerzfrei ausführen können, da in der Regel keine Systeme benötigt werden, von denen keine Build-Dateien mehr exisitieren. Oder anders formuliert: Es sollte genauso einfach sein, eine Version zu bauen, die 9 Monate alt ist, wie die ganz aktuelle Fassung.

6.4.3 Deklarative Pipelines vs. Scripted Pipelines

Bevor wir uns genauer eine Auswahl der zurzeit verbreitetsten CI-Server auf dem Markt anschauen, geht es um grundlegende Konzepte. Im ersten Schritt geht es hier darum, wie die Pipelines geschrieben werden, also zum einen um die *Syntax*, in der die Pipelines geschrieben werden, und zum anderen um die *Sprache*, die dafür verwendet wird.

Der erste Punkt ist, ob die Pipeline *deklarativ* definiert wird oder ob sie *gescriptet* wird. Bei einer *Scripted Pipeline* wird die Pipeline in einer Programmiersprache geschrieben. Der Vorteil ist, dass in der Regel der vollständige Funktionsumfang einer regulären Programmiersprache genutzt werden kann. Je nach CI-Server gibt es unterschiedliche Programmiersprachen, die unterstützt werden. Beim CI-Server *Jenkins* ist das etwa *Groovy*, das in einer Java-VM läuft. Zusätzlich zu den regulären

© Rheinwerk Verlag, Bonn 2024

Features der Programmiersprache werden auch noch weitere CI-Server-abhängige Aspekte genutzt, um etwa die einzelnen Stages und Jobs einer Pipeline zu definieren.

Scripted Pipelines

Mit der Macht einer vollständigen Programmier- bzw. Scripting-Sprache können fast alle Lösungen umgesetzt werden, die man sich so ausdenkt, und das in verschiedenen Arten und Weisen. Das macht das Ganze sehr flexibel, da sehr viele Wege zum Ziel führen. So kann man etwa Schleifenkonstrukte nutzen, um Jobs für die Pipeline zu generieren.

Das Ganze verkehrt sich allerdings auch schnell in einen großen Nachteil: Die hohe Flexibilität macht die Pipelines schnell komplex, was nicht förderlich für ihre Wartbarkeit ist. Zudem müssen alle, die die Pipeline schreiben oder anpassen wollen, die Sprache beherrschen. Auf diese Weise wird häufig der Quellcode schnell so kompliziert, dass die Wartung aufwendig ist, und es sorgt zudem dafür, dass bei jedem Projekt die Implementierung unterschiedlich verläuft.

Wichtig für so gut wie alle Automatisierungen im DevOps-Umfeld ist, dass diese für alle Teammitglieder leicht wartbar und anpassbar sind. Das sind Scripted Pipelines häufig nicht, da Spezialwissen benötigt wird. Das Resultat ist dann häufig, dass es zentrale Teams gibt, die stets zur Unterstützung herangezogen werden oder gar für die verschiedenen Teams die Pipelines schreiben. Das ist nicht nur aufwendig, sondern auch ineffizient, da es die Autonomie der Teams unnötig reduziert. Die jeweiligen Teams sollten in der Lage sein, die Pipelines selbst zu schreiben bzw. bei Bedarf selbst eigene Jobs zur Pipeline hinzuzufügen.

Häufig ist das Problem, dass zu viel an den DevOps-Werkzeugen gearbeitet wird, statt mit den Werkzeugen zu arbeiten, was zu Over-Engineering führt. Das Kernziel eines cross-funktionalen Teams sollte schließlich das Entwickeln und das Betreiben der Anwendung sein und nicht das große Management der Automatisierungen.

Die Rolle der Programmiersprachen

Die Rolle der Programmiersprachen im DevOps-Umfeld ist gar nicht so spannend, wie es Artikel mit Titeln wie »Diese Programmiersprache solltest du als DevOps Engineer können!« glauben machen wollen.

Stattdessen: Es gibt keine Programmiersprache, die sich per se besser oder schlechter für DevOps-Pipelines eignet. Und das ist auch gut so.

Trotzdem kann man natürlich einige Sprachen hervorheben, die besonders oft genutzt werden und die man daher beherrschen sollte: Bash und Python. Wenn Ihre Umgebung bereits in der Cloud angekommen ist und viele moderne Tools nutzt, ist auch Go wichtig. Näher gehe ich darauf in Abschnitt 14.3.3 ein, wenn es um den technischen DevOps-Lernpfad geht.

Persönliches Exemplar für Thomas Bartholomäus

Deklarative Pipelines

Die Alternative dazu ist der *deklarative Ansatz*, bei dem die Pipeline durch eine Konfigurationsdatei beschrieben wird. Dazu wird eine Markup-Sprache wie YAML verwendet, was ironischerweise für »YAML Ain't Markup Language« steht. Die logische Syntax wird durch korrekte Einrückungen gewährleistet, was bei komplexeren YAML-Dateien umständlich sein kann. Eigentlich sind YAML-Dateien auch ohne Vorkenntnisse sehr einfach lesbar – es sei denn, man übertreibt es mit den Einrückungen und verschiedenen Ebenen. Das ist dann meist nicht mehr ganz so einfach les- und schreibbar.

Bei Pipelines ist das häufig allerdings kein großes Problem, da diese in der Regel nicht sonderlich komplex sind. Und wenn doch, dann sollte man eher schauen, dass man das Setup entsprechend vereinfacht. Zugegeben: Das ist einfacher gesagt als getan. Gute und moderne CI-Server bringen eigene Keywords mit, die in der Pipeline genutzt werden können, um die verschiedenen Funktionen vom CI-Server zu verwenden.

Das macht das Ganze komfortabel und die Wartbarkeit ist in sehr vielen Fällen sehr simpel, da keine weitere Programmiersprache erlernt werden muss. Alle im Team können daher schnell Änderungen vornehmen, da die Hürde relativ niedrig ist.

Nicht das Rad neu erfinden

Ein häufiger Fehler in der Praxis ist, dass man versucht, das Rad neu zu erfinden. Das ist aber bei der richtigen Auswahl der Tools nicht mehr notwendig! Viele Probleme sind heutzutage schon oft genug gelöst worden, sodass man nicht auf eine eigene Idee zurückgreifen muss. Für alle beteiligten Personen ist es ohnehin besser, so nah wie möglich am Standard des verwendeten Tools und der verwendeten Technik zu bleiben. Das ist besonders im Hinblick auf die Einarbeitung sinnvoll, sodass diese Hürde so niedrig wie nötig gehalten werden kann.

6.5 DRY und KISS: »Don't repeat yourself« und »Keep it simple, stupid!«

Ganz unabhängig von der expliziten Lösung für CI/CD-Pipelines ist es wichtig, dass Sie einige Elemente beachten, damit die Automatisierungen auch möglichst effizient verwaltet werden können.

6.5.1 Zentrale Pipelines vermeiden

Oft sieht man, dass versucht wird, eine große Pipeline für die ganze Firma zu zentralisieren und alle Projekte darüber zu verwalten. Kurioserweise sieht man solche

© Rheinwerk Verlag, Bonn 2024

Ansätze umso häufiger, je größer die Firma ist. Der Grundgedanke dahinter ist, dass man den einzelnen Software-Teams den Aufwand abnehmen will, eigene Pipelines zu entwickeln. Das ist auch vollkommen nachvollziehbar: Die Teams sollen ja am Value-Stream arbeiten und nicht mit den Werkzeugen drumherum.

Wenn man also an zentraler Stelle eine Pipeline zusammenstellt, dann sollten die Teams nur noch diese Pipeline nutzen und die Verantwortung für den Build abgeben. Dazu existiert meist ein eigenes Team, das nur dafür da ist, diese Pipeline zu schreiben. Ungünstigerweise wird dieses Team dann häufig »DevOps-Team« genannt.

Grundsätzlich ist die Idee ja gut und nachvollziehbar. Allerdings bringt sie große Nachteile mit sich, die dann genau zum Gegenteil führen.

Das Problem ist nämlich, dass es gerade in großen Firmen viele unterschiedliche Builds geben wird, die noch dazu vollkommen anders funktionieren. Eine iOS-App wird ganz anders gebaut als eine Server-Anwendung, die in Java geschrieben ist. Wenn dies wirklich alles in einer zentralen Pipeline abgebildet werden soll, müssen zahlreiche Weichen und Abzweigungen erstellt werden. Die Folge ist eine unglaubliche Komplexität, die in kurzer Zeit unwartbar und undurchschaubar wird.

Aber auch organisatorisch ist das der falsche Ansatz. Gerade das Team, das für die spezielle Anwendung verantwortlich ist, kennt den Build am besten und sollte sich daher auch um die Pipeline kümmern. Es weiß, welche Konfigurationen notwendig sind und welche Abhängigkeiten vorhanden sein müssen. Daher sollten diese Entwickler auch für die Pipeline verantwortlich sein, mit der ihr Code gebaut wird. Und das sorgt auch dafür, dass das Team autonom an seiner Anwendung arbeiten kann und nicht auf eine zentrale Pipeline angewiesen ist.

Allerdings: Das sollte nicht bedeuten, dass jedes Team nach Gutdünken eigene Tools und Prozesse nutzen sollte. Wenn Sie einige grundlegende Hinweise befolgen, die ich Ihnen gleich vorstelle, werden ganz von allein ähnliche Pipelines entstehen, in die sich auch die Kollegen aus anderen Teams einfach einarbeiten können. Aber es entsteht keine zentralisierte Monster-Pipeline, die ein ganz eigenes Team benötigt, damit sie läuft.

Compliance

Es gibt einen Punkt, der eine zentrale Pipeline doch notwendig machen kann, nämlich diverse Compliance-Vorgaben. So kann es sinnvoll oder notwendig sein, dass über einen finalen Check sichergestellt wird, dass diese Vorgaben für alle Anwendungen gelten, die die Firma baut. Das ist allerdings ein Thema für sich, um das es in Abschnitt 11.6 geht. Nur kurz dazu: Mit Compliance-Checks sollen konkrete Qualitätsrichtlinien eingebunden und eingehalten werden. Diese können beim Build geprüft werden, sie sind aber kein zwingendes Argument für eine große zentrale Pipeline.

6.5.2 Pipeline-Bausteine bereitstellen

Anstatt also eine zentrale Pipeline bereitzustellen und diese allen Teams überzustülpen (oder dies zumindest zu versuchen), sollten Sie stattdessen eine zentrale Bibliothek mit einzelnen Pipeline-Bestandteilen bereitstellen. Ganz unabhängig vom konkret eingesetzten CI/CD-Tooling können Sie so dafür sorgen, dass man viele kleinere Bestandteile aufbaut, die von den Teams verwendet werden können.

Damit schlägt man gleich mehrere Fliegen mit einer Klappe: Zum einen ist die Verwaltung vergleichsweise einfach. Nicht alle Teams benötigen alle Teile einer Pipeline; diese können von den Teams dann entsprechend selbst genutzt werden. Zum anderen lernt auch das Team den Umgang mit dem CI/CD-Tooling, was zwingend notwendig ist.

Dazu muss nicht jeder Experte darin sein, allerdings sollten die Teams die Pipeline selbst anpassen können. Damit sind sowohl die eigenen, selbst geschriebenen Pipeline-Bestandteile gemeint als auch die Zusammenarbeit in den Bestandteilen der zentralen Pipeline-Bibliothek.

Beim Bau der zentralen Ppeline-Bibliothek gilt primär: KISS, DRY und YAGNI.

- *KISS* ist ein Akronym für »Keep it simple, stupid«. Der Gedanke dahinter ist, dass man die Lösung möglichst einfach hält. Im Rahmen einer DevOps-Pipeline bedeutet das vor allem, dass möglichst wenig geschachtelt wird und lieber mit den Einzelbestandteilen der Pipeline gearbeitet wird. Je einfacher die Pipeline-Bestandteile sind, desto einfacher ist es auch, sie zu warten und zu nutzen.
- Das Grundprinzip hinter *DRY* dürfte den meisten bekannt sein: »Don't repeat yourself.« Anstatt also immer wieder das Gleiche erneut zu schreiben, sollten Sie lieber versuchen, so viel wie möglich zu abstrahieren und dabei das KISS-Prinzip nicht zu vergessen.
- Auch *YAGNI* ist wichtig. Das ist ein Akronym für »You ain't gonna need it«, was so viel heißt wie: Du wirst es ohnehin nicht benötigen. Oft sieht man Lösungen, bei denen zu viel in die Zukunft gedacht wird, was den Code komplex macht und dafür sorgt, dass alles viel länger dauert. Häufig werden spezielle Use-Cases eingebaut, die ohnehin nicht gebraucht werden.

Unabhängig von KISS, DRY und YAGNI gilt, dass diese Grundsätze nicht nur für den Build-Prozess gelten, sondern natürlich auch für das Testen und das Deployment. Hierfür bietet es sich an, eine Art »Bibliothek« zu schreiben, die mit Default-Werten daherkommt. Die Teams sollten dann diese Bestandteile importieren und auch Möglichkeiten haben, einige Anpassungen für ihr eigenes Projekt selbst vornehmen zu können.

Fangen Sie aber nicht an, in der zentralen Bibliothek ihre eigenen Grenzfälle abzubilden. Das erhöht die Komplexität, dauert länger und macht es für die anderen Teams schwieriger, sie zu nutzen. Einstellungen, die Sonderfälle betreffen, müssen dann

© Rheinwerk Verlag, Bonn 2024

jeweils überschrieben werden. Erst wenn Konfigurationen doch von anderen Teams gebraucht werden, sollten sie in die zentrale Pipeline-Bibliothek verschoben werden.

Wichtig ist, dass man möglichst keine Abhängigkeiten zwischen den einzelnen Bestandteilen einbaut, sonst hat man wieder einen langen Rattenschwanz an Abhängigkeiten, die man irgendwie verwalten und debuggen muss, was mühselig ist.

6.5.3 Visibilität schaffen

Daher gilt es, einen guten Überblick für die Teams bereitzustellen, damit diese

1. die zentralen Bestandteile selbst begutachten können,
2. idealerweise nachvollziehen können, wie diese funktionieren, und
3. diese Bestandteile dann entsprechend in ihren Projekten testen und dann produktiv nutzen.

Je mehr Verständnis dafür da ist, desto besser ist das ganze Vorgehen.

Denn so können die Teams immer noch einzelne Module der Pipeline einfach herausnehmen und selbst ersetzen, wenn diese auf zentraler Ebene nicht funktionieren. Danach können diese ebenfalls zentral in die Bibliothek werden, um dort zur Wiederverwendbarkeit beizutragen.

Allgemein gilt daher: Eine zentrale Bibliothek ist gut und umso wichtiger, je größer die Firma ist. Häufig bringen die verschiedenen CI/CD-Tools allerdings auch noch eigene Templates und Lösungen mit, die ebenfalls genutzt werden sollten, um wie schon gesagt das Rad nicht neu zu erfinden. Das gilt schließlich nicht nur innerhalb der Firma, sondern auch außerhalb.

6.6 Ein Überblick über CI-Server

Es gibt viele verschiedene CI-Server, die allesamt grundsätzlich zum Ziel führen. Bevor Sie sich für einen dieser Server entscheiden, sollten Sie allerdings etliche Fragen klären, etwa zum Aufwand bei Einrichtung und Betrieb, zum Aufwand, die Pipeline zu schreiben, und zur Skalierbarkeit.

Einige Lösungen sind in vielen Firmen schon früh eingeführt worden und haben im Laufe der Zeit sehr gut den Entwicklungs-Workflow unterstützt. Allerdings merkt man schnell, dass man das heute nicht mehr so machen würde, da es aufgrund von verschiedenen Gründen aus der Zeit gefallen ist.

Grundsätzlich ist immer wichtig, dass ein CI-System für diejenigen, die es täglich nutzen müssen, möglichst einfach zu verwenden ist: mit geringen Hürden, aber auch mit genug Autonomie. Idealerweise muss man nicht jedes Mal wieder von vorne anfangen, erhält allerdings trotzdem die Möglichkeit für Anpassungen.

Persönliches Exemplar für Thomas Bartholomäus

Es gibt viele wichtige Kriterien, die bei der Auswahl von CI/CD-Systemen relevant sind. Die wesentlichen Aspekte, auf die ich in den folgenden Abschnitten immer wieder eingehen werde, sind:

- **Wartung des Systems:** Wie viel Aufwand muss man in die Wartung der Systeme stecken? Hierzu zählen auch noch Aspekte zur Skalierbarkeit, weil eine Skalierung häufig mit den Wartungsarbeiten einhergeht.
- **Komplexität beim Bau von Pipelines:** Wie einfach oder schwer ist es, Pipelines zu bauen?
- **Rechteverwaltung:** Wie lassen sich die Rechte verwalten, wer darf etwas und wer darf etwas nicht?
- **Security-Lücken:** Dieser Aspekt wird leider häufig ignoriert, ist aber wichtig. Die Software-Architektur des Systems hat Auswirkungen auf die Sicherheit, auch das muss betrachtet werden. Das geht auch Hand in Hand mit der Wartung, denn ausgelieferte Sicherheitspatches müssen auch eingespielt werden.
- **Erweiterbarkeit:** Jedes System bietet verschiedene Arten, um die Fähigkeiten zu erweitern. Alles hat Vor- und Nachteile, und auch diese müssen beurteilt werden.
- **Übersichtlichkeit:** Usability ist ein großes Thema. Daher ist auch die Übersichtlichkeit wichtig. Sieht man etwa schnell, ob das Projekt erfolgreich durchbaut oder nicht?
- **Rolle im DevOps-Gesamtkonzept:** Und zum Schluss gilt es auch noch zu schauen, welche Rolle das Tool m Gesamtkonzept der DevOps-Toolchain spielt.

Keine vollständige Analyse!

Wie auch bisher geht es mir nicht darum, hier eine vollständige Analyse aller verfügbaren CI/CD-Systeme für Sie niederzuschreiben. Es geht mir vielmehr darum, Ideen, Konzepte, Probleme und Lösungen zu beschreiben, sodass Sie bei einer ggf. benötigten Analyse schon ein solides Grundwissen besitzen.

Der Fokus liegt auf den drei verbreitetsten CI/CD-Systemen: *Jenkins*, *GitLab CI* sowie *GitHub Actions*. Natürlich gibt es noch weitere, aber sie zu beschreiben würde nicht nur den Rahmen sprengen, sondern es wäre meistens auch wenig hilfreich. Nachdem wir uns diese drei Tools angesehen haben, sollte Ihnen hoffentlich ohnehin klar sein, warum andere Tools heute und vermutlich auch in Zukunft wenig Relevanz haben werden.

6.6.1 Jenkins

Der älteste und wohl noch verbreitetste CI-Server ist Jenkins. Jenkins findet man in vielen Organisationen, obwohl der Trend dahin geht, sich von ihm zu verabschieden.

© Rheinwerk Verlag, Bonn 2024

Noch sieht man Jenkins allerdings weiterhin häufig, und es wird auch noch Jahre dauern, bis er von der Bildfläche verschwinden wird. In diesem Abschnitt möchte ich vor allem darstellen, warum Jenkins so verbreitet ist und warum man in Zukunft hoffentlich nicht mehr so viel von Jenkins sehen wird.

Buchempfehlung

Im Rheinwerk Verlag ist auch das Buch »Continuous Integration mit Jenkins« (*https://www.rheinwerk-verlag.de/continuous-integration-mit-jenkins/*) erschienen. In diesem Buch setzt sich Marco Schulz vertieft mit Jenkins auseinander.

Jenkins ist sehr flexibel einsetzbar. Was früher als Stärke angesehen wurde, ist inzwischen eine Schwäche geworden. Es gibt in Jenkins sehr viele Wege, um ein Problem zu lösen, was dazu führt, dass jede Jenkins-Installation und jedes Setup nach kurzer Zeit völlig unterschiedlich aussieht.

Die Installation von Jenkins ist schnell erledigt: Jenkins selbst ist in Java geschrieben und läuft daher auf vielen verschiedenen Hosts. Meistens werden Linux-Server genutzt oder Jenkins wird selbst in einem Container gestartet. Die Wartung der Infrastruktur und das Anlegen der Pipelines ist jedoch eine viel schwierigere Aufgabe.

Architektur und Skalierung

Jenkins besteht aus zwei Bestandteilen: Der *Controller* verwaltet die Pipeline, und die *Agents* verbinden sich mit ihm, um die Jobs der Pipelines auszuführen.

Agents und Controller

Ältere Namen für den Controller und die Agents waren »Master« und »Slaves«. Diese Terminologie wird aufgrund der rassistischen Historie dieser Bezeichnungen nicht mehr offiziell verwendet. Man findet diese Begriffe allerdings an vielen Stellen immer noch.

Diese Architektur verleitet dazu, einen zentralen Controller für alle Teams zu schaffen, auf dem dann die unterschiedlichen Pipelines ausgeführt werden. Wie Sie bereits aus Abschnitt 6.5.1 wissen, ist das keine gute Idee, denn das bedeutet, dass alle Abhängigkeiten, Plugins usw. auf diesem Controller installiert werden müssen. Das führt erstens zu dem Problem, dass ein sehr großer Jenkins-Controller mit vielen Pipelines und Plugins kaum mehr ohne ein eigenes Team wartbar ist. Wenn sich mehrere Projekte einen Jenkins-Controller teilen müssen, wird es fast immer chaotisch, da sich die Einstellungen und Plugins ins Gehege kommen.

Das zweite Problem ist, dass viele und komplizierte Pipelines auch große Last für den Jenkins-Controller bedeuten. Auch das geht selten gut, wenn viele Leute gleichzeitig auf ihn zugreifen müssen, und sorgt für Frust, wenn eine eigentlich einfache Pipeline im Schneckentempo abgearbeitet wird.

Persönliches Exemplar für Thomas Bartholomäus

Daher werden meistens viele kleinere Jenkins-Controller aufgesetzt, zum Beispiel einer für jedes Team oder jede Abteilung. Daraus ergibt sich jedoch das Problem, dass die ganzen unabhängigen Instanzen nicht zentral verwaltet werden können. Die Administration ist schwierig und von Handarbeit geprägt, was wir ja eigentlich gerade wegautomatisieren wollten.

Abhilfe schafft das Plugin *Jenkins Configuration as Code*. Mit ihm lässt sich der Jenkins-Controller in YAML-Code definieren und konfigurieren, was zuvor viele Klicks in der UI erforderte. Für die verschiedenen Instanzen innerhalb der Organisation existieren dann mehrere YAML-Dateien, in denen die relevanten Konfigurationen enthalten sind, wie etwa die notwendigen Passwörter und die genutzten Plugins. Idealerweise containerisiert man das Ganze, sodass der Jenkins-Controller etwa auf einem Container-Host hochgefahren wird, in dem bei jedem Update ein neues Image gebaut und deployt wird.

Und es hat den Vorteil, dass die Beschreibung der Pipeline beim eigentlichen Projekt gespeichert werden kann und auf diese Weise versioniert und portabel ist! Für die Reproduzierbarkeit des Builds ist das ein wichtiger Punkt. Wenn Pipelines stattdessen manuell konfiguriert werden, wird die Definition im Jenkins-Controller selbst abgespeichert. Sie ist somit *nicht* als Code definiert, und somit erfolgt auch keine Versionierung. (In der Praxis sieht man dann oft, dass die Definitionen getrennt vom eigentlichen Projekt gespeichert und versioniert werden, was auch nicht hilfreich ist.)

Pipelines mit den Projekten verwalten

Wenn Sie solche veralteten Pipelines finden, sollten Sie zusehen, dass das modernisiert wird. Das schafft viele Vorteile wie erhöhte Visibilität und bessere Wartbarkeit, was spätere Umzüge deutlich erleichtert.

Aber auch, wenn so viel automatisiert werden kann, ist der Engineering-Aufwand immer noch hoch, wenn viele kleine Installationen betreut und verwaltet werden müssen. Bei Problemen und Updates muss für jede Instanz ein Wartungsfenster gefunden und kommuniziert werden, und auch jede Automatisierung muss kontrolliert ausgeführt werden, um Probleme zu entdecken und zu beheben.

Neben dem Controller müssen auch noch die Agents verwaltet werden. Wie üblich gibt es bei Jenkins sehr viele verschiedene Möglichkeiten, wie die Agents installiert, verwaltet und skaliert werden können. Heutzutage sollten Sie auf containerisierte Agents setzen, die automatisch hoch- und herunterskalieren, und zwar abhängig davon, wie viel Workload gerade gebraucht wird.

Rechteverwaltung

Ein weiteres Problem bei der Wartung und Administration von Jenkins ist die Rechteverwaltung, die man sich eigentlich für das Gesamtkonzept der DevOps-Toolchain

© Rheinwerk Verlag, Bonn 2024

anschauen muss, denn Jenkins ist der zentrale Hub der Automatisierung. Alle anderen Tools müssen an Jenkins angebunden werden: Git-Server, diverse Tools für die Qualitätssicherung, Security und Deployment usw.

Für jedes Tool müssen also die Credentials hinterlegt und konfiguriert werden; das können Passwörter, Tokens oder andere Methoden für die Zugangskontrolle sein. Der Jenkins-Server wird damit zum zentralen Ziel Ihrer Infrastruktur und muss dementsprechend abgesichert werden. Sie brauchen unbedingt ein durchdachtes System der Beschränkungen und sollten den Zugang möglichst strikt reglementieren, um Missbrauch zu vermeiden.

Neben diesen Credentials gibt es auch noch die Rechteverwaltung in Jenkins selbst. Diese ist nah verwandt mit den Credentials, da auch mit ihr eingeschränkt werden muss, wer was darf und was nicht. Meistens ist es notwendig (und sinnvoll), dass die Konfiguration des Git-Servers dupliziert wird: Wer den Code einchecken und verändern darf, darf auch den Build starten und bearbeiten. Das bedeutet, dass Sie diese Rechte an zwei Stellen pflegen müssen. Auch das ist ein gewisser Aufwand und erhöht die Komplexität des Setups. Aber so lassen sich die Rechte für einzelne Nutzer und Nutzergruppen in Jenkins verwalten. Auch rollenbasierte Strategien sind möglich, gekapselt in einem Plugin.

Security-Lücken

Wie jede Software hat auch Jenkins Sicherheitslücken, von denen immer mal wieder welche gefunden werden. Diese Sicherheitslücken sind zum einen in Jenkins selbst enthalten, zum anderen auch in den Plugins.

Bei Updates muss Jenkins meist neu gestartet werden, was ziemlich disruptiv ist. Jenkins-Updates werden daher aufgeschoben, da es schwierig ist, eine geeignete Uhrzeit für einen Neustart zu finden, wenn alle Pipelines auf einem zentralen Server liegen. Gibt es eine Vielzahl unabhängiger Instanzen, ist Handarbeit angesagt. Im Hinblick auf die Sicherheit ist das natürlich nicht gut.

Eine sichere und praktikable Verwaltung von Jenkins allein für die Infrastruktur ist ziemlich aufwendig und erfordert ein Team, das ziemlich groß sein muss, wenn es sich auch um eine große Firma handelt.

Limitierter Funktionsumfang und Plugin-Hölle

Der Basisumfang von Jenkins ist ziemlich limitiert. Fast alle Funktionen werden erst nutzbar, wenn man Plugins installiert. Selbst wichtige Core-Funktionen werden mit Plugins abgebildet; ein Beispiel ist etwa das Git-Plugin, das es ermöglicht, mit Git-Repositorys zu arbeiten. Sie finden für fast jede Aufgabe ein Plugin, und falls doch mal keine vorgefertigte Lösung existiert, können Sie auch selbst etwas entwickeln. So lassen sich Komplexitäten aus der Pipeline in ein Plugin verlagern.

Persönliches Exemplar für Thomas Bartholomäus

Das Problem ist allerdings, dass sich diese Aufgaben damit nicht auflösen, sondern dass noch mehr Aufwand entsteht, da sichergestellt werden muss, dass die unterschiedlichen Plugins zusammenpassen. Jenkins ist berüchtigt für Kompatibilitätsprobleme, denn während die Core-Plugins vom Jenkins-Team selbst entwickelt und gepflegt werden, ist das bei den Zusatz-Plugins Aufgabe der Community. Einige Plugins werden aktiv gepflegt, bei anderen steht die Wartung still.

So kann es passieren, dass man von Plugins abhängig ist, die zwar tief im Workflow und der DevOps-Toolchain verdrahtet sind, die allerdings nicht mehr weiterentwickelt werden und so die Updates des ganzen Systems bremsen. Kurz gesagt: Alle Jenkins-Plugins werden Teil Ihrer Software-Supply-Chain (siehe Abschnitt 11.5) und müssen entsprechend gepflegt werden. Bei vielen Jenkins-Installationen stellt sich schnell heraus, dass das kaum mit vertretbarem Aufwand möglich ist.

Marketplaces & Co.

Obacht gilt natürlich nicht nur bei Jenkins-Plugins, sondern grundsätzlich bei allen ähnlichen Angeboten. Einen eigenen Marketplace zu haben, auf dem Plugins und Templates angeboten werden, scheint inzwischen fast ein Aushängeschild vieler Plattformen geworden zu sein. Seien Sie damit aber sehr vorsichtig! Sie machen sich mit Plugins und Co. nicht nur funktional von fremden Angeboten abhängig, sondern handeln sich schlimmstenfalls auch üble Sicherheitsprobleme ein.

Zu diesen Sicherheitsproblemen, die an und für sich schon schlimm genug sind, kommt, dass eine Vielzahl an Plugins dazu führt, dass der Überblick verloren geht. Besonders gilt dies, wenn in unterschiedlichen Pipelines verschiedene Plugins für die gleichen Aufgaben genutzt werden. Niemand kann dann einfach zwischen Projekten hin- und herspringen, da man sich erst einmal damit auseinandersetzen muss, auf welchem Fundament die Pipeline überhaupt errichtet wurde.

Übersichtlichkeit

Ein CI-Server sollte eigentlich dafür sorgen, dass man wichtige Informationen einfach sehen kann. Die Visibilität des Projektstatus wichtig, wie man so schön sagt.

Konkret heißt das, dass es schnell und einfach ersichtlich sein sollte, ob das Projekt ordentlich baut und ob alle Tests erfolgreich durchlaufen werden. Jenkins liefert dafür aber zu viele verschiedene Möglichkeiten. Beispielsweise wäre es eigentlich der saubere Umgang, wenn die hierarchische Struktur aus dem Sourcecode-Management auch in Jenkins abgebildet wird. Das macht es einfach, sich zurechtzufinden. Dummerweise erzwingt Jenkins diesen Weg nicht nur nicht, sondern er bietet standardmäßig noch nicht einmal diese Möglichkeit an, da Jenkins losgelöst von Quellcode-Repository existiert. Diese Verbindung muss erst hergestellt werden und erfolgt – Sie ahnen es vielleicht schon – über ein Plugin. (Früher musste das komplett

© Rheinwerk Verlag, Bonn 2024

händisch geschehen, sodass für jeden Branch und für jedes Projekt eine eigene Pipeline konfiguriert werden musste. Da das mühselig ist, wurde es oft nicht gemacht, weshalb selten Pipelines für kurzlebige Branches ausgeführt wurden.)

Sie brauchen also unbedingt das Plugin *Branch Source* für GitHub, GitLab oder Bitbucket, damit automatisch Projekte in Jenkins angelegt werden und der Struktur der eingebundenen Repositorys entsprechen. Dazu wird dort nach einem *Jenkinsfile* gesucht, in dem die Pipeline definiert wird. (Der Standard ist zwar ein einzelnes Jenkinsfile, aber das ist nur *ein* Standard von theoretisch vielen, die umgesetzt werden können.) Wenn ein Branch verschwindet, dann verschwindet auch der entsprechende Job. Diese einheitliche Struktur macht es einfach, sich zurechtzufinden, sowohl im eigenen Projekt als auch beim Blick in andere Teams.

Wesentlicher Knackpunkt ist natürlich, dass viele Jenkins-Instanzen eben nicht nach Best Practices eingerichtet sind und häufig ein Wildwuchs herrscht.

Wenn Sie stattdessen GitHub oder GitLab einsetzen (siehe Abschnitt 6.6.2 und Abschnitt 6.6.3), dann ist die Funktionalität von *Branch Source* ein eingebautes Feature, das ohne Konfiguration funktioniert, da die Repositorys und die CI-Engine direkt miteinander verknüpft sind.

Ziehen wir ein Fazit: Ist es besser, alle erdenklichen Funktionen über Plugins modular ergänzen zu können, oder setzt man doch besser auf eine monolithische One-Size-fits-all-Lösung? Beide Ansätze haben ihre Vorteile, die Antwort liegt wohl irgendwo in der Mitte. Meiner Meinung nach sollte die CI/CD-Plattform einen gewissen Basissatz an Funktionen bieten, auf die man sich absolut verlassen kann und die vom Core-Team gepflegt werden. Eigene Erweiterungen für Details und spezielle Use-Cases sind okay, aber der Zugriff auf Repositorys und der Bau der Software müssen reibungslos und ohne viel Handarbeit funktionieren.

Das entspricht allerdings nicht den Erfahrungen, die ich mit Jenkins gemacht habe, da entschieden zu viel in Plugins ausgelagert wird. Deren Qualität ist mal gut, mal eher mäßig. Zudem sorgen die vielen Individualisierungen dafür, dass die Benutzerfreundlichkeit und die Lernkurve leiden – niemand kann so recht behaupten, dass er oder sie Jenkins beherrsche, denn kein Setup gleicht dem anderen. Man muss sich disziplinieren, weil sonst zu viele unterschiedliche Plugins und Funktionen genutzt werden können, die die Verwaltung noch komplizierter machen – das bedeutet dann doch oft, dass es ein dezidiertes Jenkins-Team gibt, das für die Konfiguration verantwortlich ist, was man gerade vermeiden will.

Komplexität beim Bau von Pipelines

In Jenkins gibt es sehr viele verschiedene Arten, wie Pipelines konfiguriert und verwaltet werden können. Das macht die Verwaltung flexibel, aber auch kompliziert, da jedes Team einen eigenen Ansatz wählen kann.

Die älteste Variante ist, dass man sich die Pipeline-Konfiguration im Web-Interface zusammenklickt und dort die Shell-Skripte und sonstigen Integrationen hinterlegt. Das entspricht einer manuell konfigurierten Pipeline und sollte niemals gemacht werden, da der Build so nicht reproduzierbar ist.

Stattdessen müssen die Pipelines in einer Datei angelegt werden, nämlich in dem schon erwähnten *Jenkinsfile*. Sie wissen schon, dass es zwei verschiedene Arten von Pipelines gibt: *scripted Pipelines* und *deklarative Pipelines*. Jenkins unterstützt natürlich beide Ansätze – und noch mehr: Die Ansätze können auch vermischt werden, was den Pipeline-Bau sehr flexibel macht. Diese Flexibilität ist gleichzeitig auch das größte Problem bei Jenkins, wenn es um die Definition von Pipelines geht. Jede Pipeline kann beliebig komplex werden.

Für scripted Pipelines nutzt Jenkins die Sprache *Groovy*, die in einer Java-VM ausgeführt werden kann. Programmiersprachentypische Elemente wie Schleifen und Funktionen können verwendet werden.

Um das Problem zu umschiffen, dass jede Pipeline unterschiedlich ausschaut, gibt es auch in Jenkins deklarative Pipelines. Damit sehen die Pipelines instanzübergreifend relativ ähnlich aus, was Wartung und Einarbeitung vereinfacht.

```
pipeline {
    agent {
        docker {
            image 'maven:3.8.1-openjdk-1'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn install -DskipTests –batch-mode
                                -Dmaven.repo.local=./.m2/repository'
            }
        }
    }
}
```

Listing 6.1 Eine deklarative Jenkins-Pipeline

Das Beispiel aus Listing 6.1 zeigt eine deklarative Jenkins-Pipeline. Innerhalb der Pipeline-Definition werden Agents und Stages definiert. In der Agent-Konfiguration wird angegeben, dass das spezifizierte Maven-Image für alle Jobs genutzt werden soll. Innerhalb der Build-Stage wird dann ein `step` definiert, in dem Maven mit ein paar Parametern innerhalb der Shell ausgeführt wird.

© Rheinwerk Verlag, Bonn 2024

Halten Sie Pipelines einfach, und entscheiden Sie sich für einen Stil

Wenn Sie Jenkins einsetzen und einen bunter Blumenstrauß an verschiedenen Arten von Pipeline-Definitionen nutzen, sollten Sie auf einen einheitlichen deklarativen Standard migrieren, um Wildwuchs zu verringern. Idealerweise würde ich allerdings sowieso dazu raten, von Jenkins weg zu migrieren.

Die Rolle von Jenkins im DevOps-Gesamtkonzept

Es gibt sehr viele Gründe, warum immer mehr Organisationen von Jenkins Abschied nehmen und sich anderen Lösungen zuwenden. Der wesentliche Punkt ist der hohe Aufwand für die Wartung der Systeme. Durch die Kombination vieler verschiedener Plugins wird wenig dafür getan, die Toolchain einfach zu halten. Da es sehr schwer ist, unterschiedliche Jenkins-Installationen gleichförmig zu administrieren, bilden sich Insellösungen, was es schwer macht, in unterschiedlichen Projekten den Überblick zu behalten. Darunter leidet die Developer-Experience, denn man arbeitet mehr an den Pipelines als an der eigentlichen Anwendung.

Theoretisch ist es zwar möglich, alle gewünschten Funktionen in Jenkins so umzusetzen, in der Praxis gestaltet es sich allerdings ziemlich schwierig.

6.6.2 GitLab CI/CD

Die Umsetzung von Build-Pipelines erfolgt in GitLab ziemlich anders als bei Jenkins. GitLab betrachtet sich als komplette DevOps-Plattform, bei der die CI/CD-Komponente neben dem Sourcecode-Management die ältesten Kernelemente waren, bevor GitLab sich zur allumfassenden Plattform entwickelt hat. Viele Aspekte, die ich bei Jenkins bemängelt habe, sehen bei GitLab anders aus, da es sich eben um eine vollständige Plattform handelt, wodurch der Grundaufbau schon anders »gestrickt« ist.

Architektur und Wartung

Auch GitLab gliedert sich in zwei Teile: Es gibt den *GitLab Server* und den *GitLab Runner*. Der GitLab Server beinhaltet alle wesentlichen Funktionen der Plattform. Das betrifft nicht nur das Sourcecode-Management und die CI/CD-Verwaltung, sondern auch weitere Features wie Package-Registrys, Security-Angebote und Monitoring-Lösungen.

GitLab as a Service und GitLab self-managed

GitLab lässt sich als Service in der Cloud mieten oder lokal selbst verwalten. Die selbst verwaltete Instanz sieht man Europa am häufigsten. Wenn Sie GitLab nicht selbst hosten wollen, nutzen Sie *GitLab SaaS* auf *GitLab.com*, was mehr oder weniger den gleichen Funktionsumfang hat.

Persönliches Exemplar für Thomas Bartholomäus

Der *GitLab Runner* ist die Komponente, die von dem GitLab Server die Jobs der Pipelines abholt und entsprechend ausführt.

Der *GitLab Server* lässt sich über das Omnibus-Paket für die gängigen Linux-Distributionen installieren. In diesem Paket sind alle Abhängigkeiten enthalten, die benötigt werden, um GitLab zum Laufen zu bringen. Das beinhaltet auch die PostgreSQL-Datenbank. Ein Release erfolgt jeden Monat und kann über die Paketverwaltung im Betrieb aktualisiert werden. Gegebenenfalls kommt es zu einer kurzen Downtime, was allerdings keine Einschränkungen für die laufenden Pipelines hat. Aber auch Zero-Downtime-Upgrades sind möglich, wenn ein hochverfügbares Setup installiert und verwaltet wird. Letzteres wird allerdings erst nötig, wenn mehrere Tausend Nutzer auf dem System arbeiten.

Während der GitLab Server ausschließlich unter Linux läuft, lässt sich der GitLab Runner auf einer breiten Auswahl von Systemen installieren. Der GitLab Runner kann mit verschiedenen *Executors* installiert werden, damit Jobs auf verschiedenen Umgebungen ausgeführt werden können, etwa direkt auf dem System per Shell-Executor, in Docker-Containern oder mit automatischer Skalierung über Container in Kubernetes-Clustern. Die Skalierbarkeit ist besonders dann gegeben, wenn mit Containern gearbeitet wird, da dann automatisch skalierbare Infrastrukturen mit relativ geringem Aufwand betrieben werden können.

Im Gegensatz zu anderen Systemen liefert GitLab nur vorgefertigte Integrationen aus und kann nicht über ein klassisches Plugin-System erweitert werden. Das hat sicherlich Nachteile; meiner Meinung nach überwiegen aber die Vorteile: Die Integrationen kommen direkt aus erster Hand und werden vom Hersteller direkt entwickelt und getestet, wodurch Breaking Changes selten sind und meistens auch angekündigt werden.

Security-Lücken

Jede Anwendung hat Sicherheitslücken, so auch GitLab. Jedoch muss bei GitLab in der Regel nur der Server aktualisiert werden, was verhältnismäßig einfach ist. Bis auf eine kurze Downtime bei gängigen Installationen hat das allerdings keinen wesentlichen Einfluss auf die Pipelines: Die können ganz normal weiterlaufen.

Auf Probleme kann also grundsätzlich schnell und zügig reagiert werden, da diese gefixt und über die Paketverwaltung bereitgestellt werden. Plugins und andere Abhängigkeiten, die ein Update blockieren, gibt es nicht. Hier ist im Grunde nur ein generelles Patchmanagement nötig.

Erweiterbarkeit über Integrationen

GitLab unterstützt keine Plugins, sondern kommt mit fertigen Integrationen in andere Tools daher. Andersherum geht es allerdings auch: Es gibt Tools, die sich in

© Rheinwerk Verlag, Bonn 2024

GitLab integrieren. Das ist dann kein Plugin im klassischen Sinne, da lediglich die vorhandenen Schnittstellen genutzt werden, um die Integration zu bewerkstelligen. Die Tools sprechen direkt miteinander, es werden keine Funktionen ausgelagert.

Schauen Sie sich als Beispiel die Integrationen von *Jira* in GitLab und umgekehrt an. So gibt es innerhalb von GitLab eine Integration zu Jira, damit Sie überall in GitLab zu den entsprechenden Jira-Tickets springen können. Es gibt aber eine Integration andersherum: Code, Branches und Merge-Requests können in Jira dargestellt werden.

Im Rahmen der CI/CD-Pipeline haben klassische Tool-Integrationen als Plugins in der Praxis gar keine hohe Relevanz, denn GitLab versucht, alles aus einem Guss zu liefern. Einige Funktionen, die relevant sind, sind dann aber möglicherweise in der kostenpflichtigen Enterprise-Edition enthalten. Wenn Sie doch andere Werkzeuge in Ihrer Pipeline nutzen wollen, können Sie auf offene Standards setzen.

Ein typisches Beispiel dafür ist etwa die Integration von Security-Scannern: GitLab bringt einen Standardsatz von Security-Scannern mit. Die komplette Verwaltung inklusive der Aktualisierungen erfolgt durch GitLab selbst. Indem die Scanner per vordefinierten Templates in die Pipeline eingebunden werden, können diese ohne viel Aufwand genutzt werden.

Wer hingegen separate, eigene Security-Scanner nutzen möchte, kann dies auch tun. Man muss sich dann nur eben selbst um die Ausführung in der Pipeline kümmern. Je nachdem, ob im konkreten Fall dann auch noch die Enterprise-Edition in Ultimate verwendet wird oder nicht, kann auch eine Integration in das Vulnerability-Management von GitLab erfolgen.

Die Kernaussage sollte aber klar sein: GitLab hat im Bereich von CI/CD keine direkte Möglichkeit, mit Plugins zu arbeiten, aber weil sich jegliche Dinge in der Pipeline ausführen lassen und ein starkes Basis-Set vorhanden ist, sind Plugins ohnehin häufig gar nicht so notwendig. GitLab bezeichnet sich selbst schon aus guten Gründen als *DevOps-Plattform*.

Und abermals ist hier eine gute Gelegenheit, über ein wesentliches Mantra von DevOps-Prinzipien zu sprechen: die *Visibilität von Arbeit und Fortschritten*. Diese ist gerade bei GitLab sehr gut umgesetzt, da alles in einer Plattform gesammelt ist – vorausgesetzt natürlich, dass alle Funktionen auch genutzt werden.

Da es in diesem Teil vor allem um die Übersichtlichkeit von CI/CD geht, ist es erneut wichtig, auf das Zusammenspiel zwischen Sourcecode-Management und CI/CD zu schauen. Grundsätzlich kann jedes Projekt nur eine Pipeline besitzen. Dadurch ist nicht nur rasch ersichtlich, wie gerade der Status der Pipeline für den entsprechenden Commit ist, sondern diese Pipeline lässt sich auch zügig erreichen.

Persönliches Exemplar für Thomas Bartholomäus

Rechteverwaltung

In GitLab gibt es fest definierte Benutzerrollen, die in den Gruppen und den Projekten den Benutzern zugewiesen werden. Für die reine CI/CD-Komponente hat es gar nicht so viel Auswirkung, außer vielleicht, wenn es um Deployments geht.

Wichtiger in diesem Kontext ist, dass die Rechte übergreifend für die gesamte Plattform gelten. Wenn etwa in der Pipeline ein Container-Image gebaut und es in die GitLab-interne Container-Registry gepusht wird, dann werden keine zusätzlichen Konfigurationen für die Zugänge zu den verschiedenen Systemen benötigt. So lassen sich die vielen verschiedenen Funktionen der Plattform effizient verwalten, ohne dass man ständig für einzelne Nutzer die Berechtigungen anpassen muss, wie es etwa in Jenkins der Fall ist.

Komplexität beim Bau von Pipelines

Die Pipelines in GitLab werden in YAML-Dateien definiert. Dafür wird in jedem Repository eine *.gitlab-ci.yml*-Datei angelegt, in der die Stages und Jobs der Pipeline definiert werden.

Dieses Setup ist relativ simpel gehalten, auch wenn sich damit einige komplexe Setups bauen lassen. Da die YAML-Datei direkt im Repository liegt, muss keine Konfiguration erfolgen, um die Pipeline zum Laufen zu bekommen, da das Repository Bestandteil des Systems ist.

Wenn der CI/CD-Prozess über ein externes Tool abgebildet wird, müssen in Jenkins für jedes Projekt oder jede Gruppe nicht nur die Zugänge konfiguriert werden, sondern es muss auch eingerichtet werden, dass die Pipelines entsprechend automatisch konfiguriert und verwaltet werden.

Diese Frage stellt sich bei GitLab nicht, da dies natürlicher Bestandteil der Plattform ist. Es muss lediglich die Datei *.gitlab-ci.yml* angelegt werden – der Rest geschieht völlig automatisch. Man erspart sich einiges an Arbeit und verringert mögliche Fehlerquellen. Bedingung ist natürlich, dass ein nutzbarer Runner konfiguriert ist, der die Pipeline-Jobs ausführt.

Wie ich bereits kurz angesprochen habe, gibt es verschiedene Arten, um die Jobs der Pipelines auszuführen. Empfehlenswert ist es, Container zu verwenden, um die entsprechenden Jobs auszuführen. Das führt zu sauberen Build-Umgebungen und es lässt sich einfacher skalieren. Um den Wartungsaufwand der eingesetzten Container möglichst gering zu halten, sollten die Container-Images der Programmiersprachen bzw. Build-Tools verwendet werden.

Die Pipelines werden deklarativ definiert, sodass sich Pipeline-Jobs grundsätzlich einfach lesen lassen. Die Komplexität der YAML-Datei ist gering: Es gibt zwar einige Keywords, die man anwenden kann, diese sind aber gut dokumentiert.

© Rheinwerk Verlag, Bonn 2024

```
stages:
- build
- test

[…]

build:
    stage: build
    image: maven:3.8.1-openjdk-15
    script:
        - mvn install -DskipTests –batch-mode
              -Dmaven.repo.local=./.m2/repository
    artifacts:
      paths:
        - .m2
        - target
```

Listing 6.2 Ausschnitt einer ».gitlab-ci.yml«

In dem Ausschnitt einer *.gitlab-ci.yml*, den Sie in Listing 6.2 sehen, werden zwei Dinge definiert: zum einen die Stages `build` und `test`, zum anderen ein Job namens `build`, der in der `build`-Stage ausgeführt wird. Hier wird ein Maven-Container genutzt, um dann im `script`-Teil Maven laufen zu lassen. Damit die herausfallenden Artefakte für nachfolgende Tests im Test-Job genutzt werden können, werden noch zwei Pfade definiert, die als Artefakte gespeichert werden sollen, damit diese mit im nächsten Job verwendet werden können. Von den Artefakten abgesehen, ist das das gleiche Beispiel wie auch zuvor bei der deklarativen Jenkins-Pipeline aus Listing 6.1.

Da GitLab CI auch nicht mehr ganz so neu ist, sind schon etliche Features eingeflossen, die von den Standard-Use-Cases abweichen. Auch in diesem Fall brauchen Sie das Rad meistens nicht neu zu erfinden, da es für viele speziellere Funktionen eigene Keywords gibt, die dann gegebenenfalls in der Enterprise-Edition verfügbar sind.

Natürlich gibt es noch sehr viel mehr Details, die es zu beachten gilt. Das würde aber auch den Rahmen dieses Kapitels sprengen. Zusammenfassend lässt sich allerdings schon sagen, dass die meisten Pipelines relativ ähnlich gehalten sind, weshalb es nicht jedes Mal eine hohe Lernkurve bedeutet, sich in die Definition einer Pipeline einzulesen.

Komplexe Setups gibt es natürlich auch. Hierfür würde man dann mit hierarchischen Templates arbeiten, die die Wiederverwendbarkeit erhöhen. Im Herbst 2023 erschienen allerdings mit *CI/CD Components* erste Schritte, um die Wiederverwendbarkeit nochmals deutlich zu vereinfachen und somit zu erhöhen.

Persönliches Exemplar für Thomas Bartholomäus

Die Rolle von GitLab CI/CD im DevOps-Gesamtkonzept

Die Rolle von GitLab CI/CD im DevOps-Gesamtkonzept ist ziemlich eindeutig: Bei GitLab liegt der Fokus nun mal darauf, eine vollständige DevOps-Plattform zur Verfügung zu stellen. Die Nähe des Sourcecodes zu CI/CD ohne viel Konfiguration und ohne die Notwendigkeit oder das Bedürfnis, fehlende Funktionen mit Plugins nachzurüsten, ist der Kernbestandteil von GitLab.

Das ist auch der wesentliche Unterschied zu Jenkins: Während dort alles Mögliche konfigurierbar und erweiterbar gestaltet ist, ist das nicht der Ansatz, den GitLab verfolgt. Anstatt gefühlt unendliche Implementierungsmöglichkeiten zu erlauben, gibt es in GitLab nur wenige Möglichkeiten, die Funktionen umzusetzen.

Diese sind dafür dann einfach umzusetzen, sodass keine große Einarbeitung erforderlich ist, wenn zwischen GitLab-Instanzen von verschiedenen Organisationen gewechselt wird. Natürlich gibt es auch hier etliche Ausnahmen, aber an dieser Stelle geht es vor allem um den Standard.

Während wir uns in diesem Kapitel im Wesentlichen mit CI befassen, ist auch die Anbindung an Funktionen aus dem *Continuous-Delivery*-Umfeld relevant. Das muss zwangsläufig Hand in Hand gehen. So lassen sich unter anderem aus der Pipeline-Definition die relevanten Deployment-Umgebungen ansprechen, damit diese auch in der Benutzeroberfläche von GitLab dargestellt werden, um die nahtlose Integration zu gewährleisten.

6.6.3 GitHub Actions

GitHub Actions ist die Automatisierungskomponente von GitHub. Ihr Kernaufgabengebiet ist das Setup von CI/CD-Pipelines, es können aber auch andere Automatisierungen damit konfiguriert und ausgeführt werden.

Im Vergleich zu GitLab und Jenkins ist dieses Angebot noch vergleichsweise neu: Es wurde 2018 vorgestellt. GitHub geht seitdem einen ähnlichen Weg wie GitLab und baut sich ebenfalls in eine zentrale Plattform um, womit der komplette Software-Delivery-Lifecycle abgebildet werden soll.

Installation und Wartung

Wie auch bei GitLab muss man bei GitHub zwei verschiedene Komponenten unterscheiden: den *Server* und die *Runner*. Beim Server gilt es allerdings ein paar Unterscheidungen zu machen, vor allem wenn man sich SaaS und eine selbst gemanagte Server-Instanz anschaut.

Aber der Reihe nach: Bekannt ist GitHub primär für das Angebot GitHub.com, auf dem sehr viele Open-Source-Projekte gehostet werdem. Auf GitHub.com landen alle neuen Funktionen schnell und regelmäßig. Neben der kostenfreien Variante gibt es

© Rheinwerk Verlag, Bonn 2024

auch noch kostenpflichtige Subskriptionen mit mehr Funktionen für Teams und Firmen. GitHub.com wird wenig überraschend direkt von GitHub verwaltet und weiterentwickelt. GitHub wiederum gehört vollständig zu Microsoft.

Für Enterprise-Kunden gibt es zudem *GitHub Enterprise Server* sowie *GitHub Enterprise Cloud*. Während Letzteres eine dedizierte GitHub-Instanz ist, die von GitHub gehostet wird, richtet sich GitHub Enterprise Server an Kunden, die es selbst hosten wollen. Zudem kann auch das SaaS-Angebot über *GitHub.com* genutzt werden.

Es ist allerdings wichtig, Folgendes hervorzuheben: Der Funktionsumfang von *GitHub Enterprise Cloud* und *Server* hängt dem von GitHub.com häufig einige Zeit hinterher. Es sind zwei verschiedene Produkte, die nicht zeitgleich mit den gleichen Funktionen beliefert werden. Und das betraf auch *GitHub Actions*. Während es für GitHub.com schon im Jahr 2018 verfügbar war, folgte es für GitHub Enterprise Server erst im Januar 2021.

Neben dem Server muss man sich auch noch die Funktionen der *GitHub Actions-Runner* anschauen. Auf GitHub.com sind Runner verfügbar, die von allen genutzt werden können. Wenn man eigene Runner hinzufügen möchte, ist das mit self-hosted Runnern möglich. Diese können für Linux, macOS und Windows heruntergeladen und installiert werden. Ein richtiges Paket existiert dafür nicht, denn man muss ein *.zip*- bzw. *tar.gz*-Paket herunterladen, um es anschließend zu starten. Anschließend können Jobs ausgeführt werden.

Die Runner lassen sich automatisch skalieren, die Möglichkeiten waren im Stand Winter 2023 im Vergleich zu GitLab aber deutlich reduziert. Es werden offiziell zwei automatische Skalierungen empfohlen. Das eine ist ein Controller für Kubernetes, mit dem sich die Actions auf Kubernetes-Clustern ausführen lassen. Die Alternative ist ein Terraform-Modul für die Skalierung auf AWS. Andere Skalierungsmöglichkeiten existieren zwar auch, sind aber – genauso wie das empfohlene Terraform-Modul für AWS – nur von Drittanbietern erhältlich und müssen daher recht aufwendig gepflegt und integriert werden.

Security-Lücken

Die Handhabung von Sicherheitslücken bei GitHub ist relativ ähnlich zu der bei GitLab. Auf GitHub.com werden Sicherheitslücken regelmäßig gefixt und ausgerollt, sodass man davon fast nichts mitbekommt. Für Enterprise Server werden ebenfalls regelmäßig Sicherheitsaktualisierungen angeboten, die eingespielt werden müssen. So weit, so normal.

Wichtig ist aber, vielmehr darauf zu schauen, wie die Sicherheit bei den GitHub Actions gehandhabt wird: Dort kann man schnell und einfach fremde Actions einführen, die man stets aktuell halten muss. Dann ist der Weg zu den Sicherheitsproblemen nicht weit, die man von Jenkins-Installationen mit Hunderten von Plugins kennt.

Persönliches Exemplar für Thomas Bartholomäus

Rechteverwaltung

Bei der Rechteverwaltung in GitHub sind zwei Aspekte relevant: die Rechteverwaltung der Nutzer und die Berechtigung des GitHub-Tokens, der von Workflows genutzt wird, um andere Daten zu bearbeiten.

Für die Repositorys und auf Organisationsebene können für einzelne Nutzer und Nutzergruppen Berechtigungen gesetzt werden. In der kostenfreien Variante können die Rollen *Read*, *Triage*, *Write*, *Maintain* und *Admin* konfiguriert werden, die mit einem statischen Satz an Berechtigungen daherkommen. Mit GitHub Enterprise hingegen lassen sich die Berechtigungen feingranularer konfigurieren.

Denn wie auch bei GitLab sind hier die Berechtigungen über die verschiedenen Funktionen hinweg relevant. Sie müssen die Rechte also nicht an mehreren Stellen pflegen, um den Zugriff auf den Sourcecode, die CI/CD-Plattform und auch die Paket- und Container-Registry zu verwalten.

Komplexität beim Bau von Pipelines

GitHub setzt für die Definition von Automatisierungen auf YAML-Dateien. Die Besonderheit ist, dass die Automatisierungen nicht auf die Ausführung von CI/CD-Pipelines limitiert sind, sondern dass auch weitere Aktionen getriggert werden können, wenn andere Events auftreten. In der Dokumentation finden Sie eine lange Liste an Events, die für diese Automatisierungen genutzt werden können:

https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

Für CI/CD gibt es relevante Events wie `push` oder `pull_request`. Ein Push-Event wird getriggert, wenn neue Commits auf den Servern gepusht werden. Ein Pull-Request-Event wird unter anderem dann getriggert, wenn ein Pull-Request angelegt wird. Außerhalb von typischen CI/CD-Use-Cases können auch Automatisierungen beim Erstellen von Issues oder beim Setzen von Labeln an den Issues getriggert werden. Das Ganze macht die Ausführung von Workflows sehr flexibel.

Die Workflows werden ebenfalls in YAML geschrieben und sind somit deklarativ. Definiert werden sie im entsprechenden Repository im Verzeichnis *.github/workflows/*. In diesem Verzeichnis können mehrere Workflows hinterlegt werden, sodass die verschiedenen Use-Cases in getrennten Dateien definiert werden können.

Ein Workflow kann aus mehreren Jobs bestehen und jeder Job aus mehreren Steps. In den Steps werden alle benötigten Schritte ausgeführt, etwa für den Bau und den Test des Projektes. Listing 6.3 zeigt beispielsweise einen Workflow für den Bau eines Java-Projekts mit Maven:

```
name: Java CI

on: [push]
```


© Rheinwerk Verlag, Bonn 2024

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Build with Maven
        run: mvn install -DskipTests -batch-mode
                         -Dmaven.repo.local=./.m2/repository
```

Listing 6.3 Ein Workflow in GitHub Actions

Dieses Beispiel ist im Grunde das gleiche wie auch im Abschnitt zu Jenkins und GitLab CI. Auch hier wird ein Java-Projekt mittels Maven gebaut. Die Syntax sieht auch hier wieder ein wenig anders aus.

Nachdem der Name für diesen Workflow gesetzt wurde, wird noch das Event spezifiziert, denn hier soll der Workflow bei jedem Push laufen. Spannender wird es, wenn man sich die Job-Definition anschaut. Wie auch in den vorangegangenen Beispielen ist hier lediglich ein Job namens `build` definiert. Dieser besteht hier allerdings aus mehreren Stufen, nämlich drei.

Vor dem ersten Schritt wird definiert, dass diese Projekt unter Ubuntu ausgeführt werden soll. Ein containerisiertes Setup erfolgt in diesem Beispiel nicht, wäre aber auch möglich, wenn man im Bereich `uses` ein Docker-Image spezifiziert. Im ersten Schritt wird dann der Git-Checkout ausgeführt, sodass eine Arbeitskopie des Repositorys für den Workflow zur Verfügung steht.

In den nächsten beiden Schritten werden dann die vordefinierten Actions verwendet, die die Laufzeitumgebung für den Build bereitstellen. Das ist die Installation der benötigten Java-Umgebung. Anschließend wird darin Maven ausgeführt, um das Projekt zu bauen.

Erweiterbarkeit

Ein praktisches Feature ist, dass die Wiederverwendbarkeit in GitHub Actions einfach gestaltet ist. So gibt es einen Marketplace für Actions (*https://github.com/marketplace?type=actions*), mit denen man vordefinierte Steps in die eigenen Workflows übernehmen kann. So muss man nicht jedes wieder von vorne loslegen, sondern greift auf bestehende Implementationen zurück.

Persönliches Exemplar für Thomas Bartholomäus

Grundsätzlich ist die Möglichkeit, Plugins zu nutzen, gut: Man spart sich die Arbeit, muss das Rad nicht neu erfinden und nutzt stattdessen die Arbeit aus der Community, denn alle können Actions auf den Marktplatz laden.

Das hat aber auch hier wieder Auswirkungen auf die Supply-Chain Ihres Projekts. Überlegen Sie sich gut, ob und welche Abhängigkeiten Sie sich einhandeln wollen. Prüfen Sie zumindest, von wem die Action stammt. Empfehlen würde ich, nur offizielle Actions mit folgendem Hinweis zu nutzen: *Verified creator: GitHub has verified that this action was created by actions.*

Hierbei handelt es sich um eine offizielle Action, die von GitHub Actions selbst kommt. Außerdem gibt es noch verifizierte Actions von Drittanbietern, die man mit relativ gutem Gewissen nutzen kann.

Aufpassen sollten Sie bei nicht verifizierten Actions, die von normalen Nutzern angelegt wurden. Am besten lassen Sie solche Angebote links liegen, zumindest müssen Sie aber den Code prüfen, um sicherzustellen, dass hier keine offensichtlichen Sicherheitslücken vorhanden sind und die Action (nur) das tut, was sie verspricht. Denken Sie immer daran, dass diese Actions Zugang zum Code und zu angeschlossenen Systemen erhalten können.

Weiterhin sollten Sie die Version der Actions fix spezifizieren, um nicht in mögliche Probleme bei neueren Versionen hineinzulaufen. Für jede Action, die Sie von externen Anbietern beziehen, ist es notwendig, dass bei jeder neuen Version geprüft wird, ob noch alles so funktioniert, wie man es erwartet. Gibt es Inkompatibilitäten oder Änderungen, die man nicht direkt versteht?

Letztlich hat man also noch einen Layer an Abhängigkeiten, bei dem es etliche Faktoren zu beachten gilt, um sicher, aber auch aktuell zu sein. Und das führt zu einem gewissen Mehraufwand, dessen man sich bewusst sein sollte. Im Vergleich zu Jenkins ist es zwar einfacher, diese Abhängigkeiten zu aktualisieren, doch muss es für jedes Projekt geprüft und erledigt werden.

Übersichtlichkeit

Die Übersichtlichkeit ist in GitHub kein so großes Problem, vor allem dann nicht, wenn man auf das Zusammenspiel von Sourcecode-Management und CI/CD schaut.

Zu jedem Commit erreicht man etwa schnell die spezifizierten Workflows und die darin enthaltenen Actions. Da Workflows aber nicht nur an Commits hängen, sondern auch an den Events, können diese schnell über den Actions-Menüpunkt im Repository erreicht werden.

Fremde Personen können sich also auch relativ schnell in ein Projekt einfinden und sehen dort einen verhältnismäßig standardisierten Ablauf, so wie man es auch bei GitLab gewohnt ist. Einzig Jenkins ist hier wohl die Ausnahme, bei der es etliche Möglichkeiten gibt, zu einer Lösung zu kommen.

© Rheinwerk Verlag, Bonn 2024

Die Rolle von GitHub (Actions) im DevOps-Gesamtkonzept

Für viele dürfte es an dieser Stelle nicht überraschend sein, dass die Rolle von *GitHub* und *GitHub Actions* im Gesamtkonzept der technischen DevOps-Implementierung ähnlich essenziell ist, wie es auch bei *GitLab* der Fall ist.

Weniger Tools, weniger Handoffs, eine geringere Komplexität und ein einheitliches Rechtekonzept sind dafür wichtig. Hier sprechen also die nahezu gleichen Argumente für GitHub, die auch für GitLab gelten. Stellenweise gibt es natürlich einige Unterschiede, die diversen Use-Cases sind einfacher oder schwieriger umzusetzen.

Dieses Thema nehmen wir in Kapitel 13 noch genauer unter die Lupe, wenn es um die DevOps-Plattformen als Gesamtes geht. Dann werden die hier bereits genannten Aspekte tiefer begutachtet.

6.6.4 Sonstige CI-Server und Tools

Es gibt noch zahlreiche weitere Tools, mit denen Sie den CI/CD-Prozess umsetzen können. Grundsätzlich werden Sie wahrscheinlich mit allen ans Ziel kommen, allerdings ist nicht alles komfortabel und einfach zu nutzen.

In diesem Teil des Kapitels haben wir uns vor allem drauf konzentriert, dass ein generelles Konzept gewählt wird, bei dem CI/CD Teil der gesamten SDLC-Plattform sein sollte – eben was heute DevOps-Plattformen genannt und näher in Kapitel 13 thematisiert wird.

Neben GitLab und GitHub spielt da auch noch *Azure DevOps* rein. Letzteres ist, wie auch GitHub, Teil von Microsoft und ist ebenfalls nicht allzu selten in Firmen zu finden. Je näher man an Microsoft-Technologien wie .NET und der Azure Cloud ist, desto eher findet man auch Azure DevOps.

Interessant wird definitiv sein, wie Microsoft die Zukunft gestaltet. Eine langfristige Zukunft von Azure DevOps halte ich eher für unwahrscheinlich, da die interne Konkurrenz zu GitHub groß ist, aber das ist lediglich Spekulation.

Auch Atlassian hat Angebote für CI/CD in petto. Die deutlich ältere Lösung ist *Bamboo* und kann ausschließlich für On-Premise-Installationen verwendet werden. Dadurch, dass es von Atlassian entwickelt wird, ist eine enge Integration in *Jira* und *Bitbucket* gegeben. Zwar ist es keine einheitliche Plattform, allerdings kommt dieser »Atlassian Stack« einer Plattform schon relativ nahe.

Für die Cloud ist Bamboo nicht verfügbar, dafür existiert *Bitbucket Pipelines*, die zweite CI/CD-Lösung aus dem Hause Atlassian. Die Pipelines werden hier auch in YAML definiert.

Es gibt noch zahlreiche weitere Tools wie *Teamcity* von Jetbrains, *Circle CI* und Konsorten. Das Grundkonzept mit einem zentralen Server sowie der Definition der Pipe-

line ist fast überall gegeben. Andere Konzepte werden etwa von *dagger.io* verfolgt oder auch von Tekton als Cloud-Native-CI/CD-Lösung.

6.6.5 CI bei schick-gekleidet.de

Je länger sich die Verantwortlichen bei *schick-gekleidet.de* die verschiedenen Problemstellungen und Tools angeschaut haben, desto deutlicher wurde, dass auch bei der Continuous Integration die Komplexität gering gehalten werden sollte, sodass jedes Team seine Pipelines im Wesentlichen selbst verwalten kann.

Zudem war wichtig, dass die Infrastruktur möglichst einfach zu verwalten ist, weswegen Jenkins direkt aus dem Raster fiel. Nicht sonderlich verbreitete Nischenlösungen waren ebenfalls nicht interessant, denn es wurde vermutet, dass es schwieriger ist, für sie Leute anzulernen oder bei Problemen schnell Lösungen zu finden.

Die Wahl von GitHub oder GitLab war also auch hier relativ naheliegend. Welches Tool bei *schick-gekleidet.de* genutzt werden sollte, wurde allerdings heiß diskutiert: Die einen kannten GitHub vor allem aus Open-Source-Projekten, die anderen schätzten die Ausgereiftheit und den Open-Source-Ansatz von GitLab.

Reflexion

Continuous Integration (CI) ist essenziell für die DevOps-Idee. Eine wichtige Grundlage ist eine saubere Verwaltung des Sourcecodes, wie in Kapitel 5 bereits beschrieben.

Unabhängig von der verwendeten Technologie ist es wichtig, dass Sie die Integrationen möglichst klein halten, damit diese schneller ausgeführt werden können. Die Pipelines unterstützen letztlich diesen Aspekt.

Fast jedes Mal, wenn ich das Setup von CI/CD-Umgebungen in diversen Unternehmen sehe und darüber rede, stelle ich fest, dass sehr hohe technische Schulden zu finden sind. Das hängt zum einen mit den verwendeten Tools zusammen, aber auch mit dem Bedürfnis Einzelner, das Rad stets neu erfinden zu müssen.

Genau deswegen sollten Sie schauen, dass Sie die Pipeline und deren Definition möglichst schlank halten. Vermeiden Sie »super intelligente« Pipeline-Setups, und versuchen Sie, so nah am Standard des CI/CD-Tools zu bleiben wie möglich.

Ich habe in meiner Karriere sehr viel an und mit Jenkins gearbeitet – etwas, was mir früher auch Spaß gemacht hat, heute aber kaum noch nötig ist, da GitLab und GitHub hier in der Zwischenzeit gute Arbeit geleistet haben. Die verwaltungstechnischen Schwächen, die Jenkins aufweist, sind in GitLab und GitHub ausgebügelt.

Je weniger Sie *an* einem Werkzeug arbeiten, sondern *mit* ihm, desto besser. Und genau deshalb migrieren viele von Jenkins weg. Und genau deswegen empfehle ich trotz meiner jahrelangen Erfahrung mit Jenkins, es zukünftig nicht mehr zu verwenden.

© Rheinwerk Verlag, Bonn 2024

6.7 Fazit

Der Build-Prozess ist einer der ersten Schritte in einer CI/CD-Pipeline. Obwohl er als technisch angesehen werden kann, spielen hier viele Aspekte aus den DevOps-Prinzipien hinein. Wichtigster und offensichtlichster Punkt aus dem CALMS-Modell ist das **A** für Automation: Der Build-Prozess muss ordentlich automatisiert werden, und zwar durch die Nutzung eines CI-Servers und des Paketmanagers der jeweiligen Programmiersprache.

Das allein schafft schon sehr viele Vorteile in Bezug auf die Entwicklerproduktivität für aktuelle Teammitglieder und auch für das Onboarding – und das eigentlich fast unabhängig von DevOps.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 7
Die Qualität sicherstellen

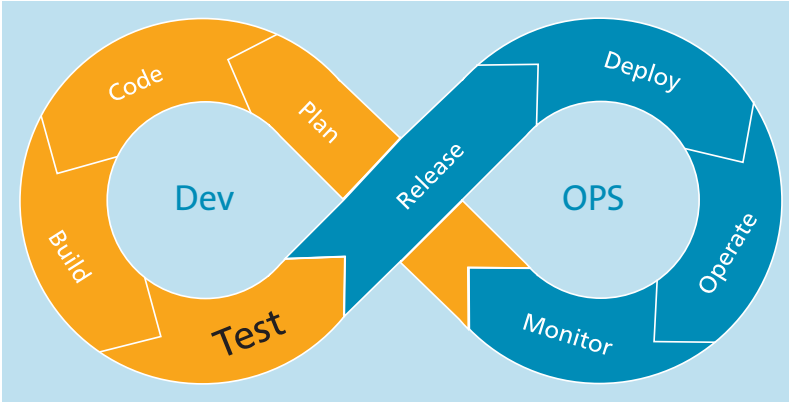

Abbildung 7.1 Test: Der Code wird getestet, um Fehler zu identifizieren und sicherzustellen, dass er den Anforderungen entspricht.

Im letzten Schritt des Entwicklungsteils des DevOps-Lifecycles geht es um die Qualitätssicherung. Hier stellen sich zwei Fragen: Wie findet sich das Qualitätssicherungsteam in die cross-funktionale Matrix ein, und wie verändert das DevOps-Paradigma die Testtätigkeiten für Personen mit verschiedenen Tätigkeitsschwerpunkten?

Es geht bei DevOps ja nicht nur um das Zusammenbringen von Dev und Ops, sondern *auch* das QA-Team muss integriert werden.

7.1 Die typischen Probleme beim Testing

Die Software von *schick-gekleidet.de* wurde bisher komplett getrennt von der eigentlichen Entwicklungsarbeit gestestet. Sowohl die Teamstruktur als auch der Prozess im Entwicklungszyklus selbst sorgten dafür, dass die Programmierarbeit und die anschließende Qualitätssicherung nichts miteinander zu tun hatten.

Das führt zu vielen Nachteilen, aber kaum zu Vorteilen. Die Teams waren schlicht nach den getrennten Aufgaben aufgebaut worden, und so lief es über die Jahre ohne große Änderungen, schließlich hatte man es ja schon immer so gemacht.

7.1.1 Die Teams in ihren Silos

Die Devs warfen nach dem Ende der Entwicklungstätigkeiten das Projekt über die große Mauer rüber zum QA-Team, denn »die sind schließlich für das Testen zuständig«. Die Arbeit des QA-Teams fand dann in einem eigenen Repository und in einer eigenen Testinfrastruktur statt. Der Code für die Tests und die entsprechenden Konfigurationen und Builds waren also von der Umgebung der Entwickler getrennt.

Das Hauptproblem bei dieser Arbeitsweise war, dass das QA-Team immer den Änderungen an den einzelnen Modulen des Online-Shops hinterherrennen musste. Während der Entwicklungsarbeit am Online-Shop veränderten sich sowohl der Code als auch die Benutzeroberfläche so sehr, dass die bisher geschriebenen Tests nicht mehr funktionierten; der Aufwand, der in die Pflege geflossen war, war also umsonst.

Da eine händische Aktualisierung der Tests kaum zu leisten war, wurde stattdessen sehr viel manuell getestet – gerne auch mal mit Checklisten und Tabellen, die Definitionen und die Abfolge für das manuelle Testen enthielten. Aber auch diese »Dokumentation« war häufig nicht mehr aktuell, da sich die Menüführung im Online-Shop ständig verändert. Der Fokus lag also auf manuellen End-To-End-Usertests, bei denen schlicht die Menüs nachgeklickt wurden. Dies war auch von der Führungsetage der Firma so gewollt: Schließlich ist das Menü ja der Teil, der beim Nutzer ankommt, und somit ist es ja das Wichtigste.

Schauen wir uns zunächst wieder den Ist-Zustand an: Wie wurde bei *schick-gekleidet.de* getestet und versucht, die Qualität sicherzustellen?

Die Hauptarbeit des QA-Teams fing immer nach der Implementierungsphase des Entwicklungsteams an, wie Abbildung 7.2 zeigt. Es gab also immer wieder Zeiten, in denen Leerlauf herrschte, weil gerade die Implementierungsphase lief.

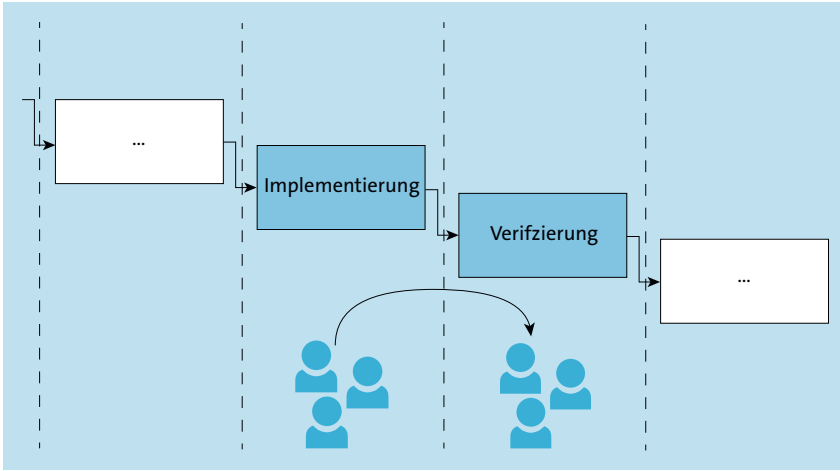

Abbildung 7.2 Das Entwicklungsteam wirft das Projekt an das QA-Team weiter, ohne dass ein direkter Austausch stattfindet.

© Rheinwerk Verlag, Bonn 2024

Da der Shop aus vielen unterschiedlichen Modulen besteht, die unabhängig voneinander entwickelt wurden, gab es auch mehrere QA-Teams, die für ein Modul zuständig waren. Oder auch für mehrere, was die Komplexität des Testens noch erhöhte. Kein Modul des Shops konnte auf die gleiche Weise getestet werden, also mussten auch die unterschiedlichen QA-Teams unterschiedlich arbeiten.

Wenn das QA-Team einen Fehler gefunden hatte, konnte dieser nicht direkt korrigiert werden. Stattdessen mussten Tickets beim Entwicklungsteam aufgemacht werden, in denen die Fehler beschrieben wurden. Das Entwicklungsteam musste dann nacharbeiten. Beliebt machten sich die QA-Teams dadurch natürlich nicht, schließlich erzeugten sie für das Entwicklungsteam nur mehr Arbeit. Gleichzeitig freute sich das QA-Team, denn es wurde daran gemessen, wie viele Fehler es fand.

Dass offensichtliche Fehler gefunden und nach einiger Zeit korrigiert wurden, stellte noch den Idealfall dar. Häufig wurde der Ablauf weiter verkompliziert, da überhaupt nicht klar war, wie ein Feature genau umgesetzt werden sollte bzw. wie etwas funktionieren sollte. »It's not a bug, it's a feature!«, den Spruch hörte man zu oft. Das wurde zwar mit einem lustigen Unterton dahingesagt, der Kern des Problems blieb aber ungelöst: Wer entscheidet überhaupt, ob etwas wie geplant funktioniert? Das konnten eigentlich nur die Kollegen des Business-Teams sagen, die die Anforderungen ursprünglich formuliert hatten. Der Weg zu den Sales- und Marketing-Leuten war jedoch noch deutlich länger, und selbst wenn man dort einen Ansprechpartner hatte, wusste dort niemand mehr so genau, was man vor 6 Monaten eigentlich genau wollte.

7.1.2 Unterschiedliche Verständnisse von Anforderungen

Die Probleme der Kommunikation zwischen den Teams wurden umso deutlicher, je näher der Release-Termin rückte. Bevor die Freigabe erfolgen konnte, musste die fachliche Prüfung von einem separaten Team durchgeführt werden. Dieses Team musste also herausfinden, ob die definierten Business-Anforderungen eingehalten wurden.

Hierbei geht es also um das Deployment auf die Produktivumgebung. Eine echte Staging-Umgebung gab es nicht, höchstens einmal händisch konfigurierte Testumgebungen, wenn wirklich große Features getestet werden mussten. Diese entsprachen natürlich nie der »echten« Prod-Umgebung, daher fielen Fehler nicht auf, sondern es knallte erst auf Produktivumgebungen.

Da man das natürlich unbedingt verhindern wollte, konzentrierten sich die Tester fast ausschließlich auf technische Details und Stolpersteine, die das Deployment und den Betrieb behindern konnten. Man klickte sich durch die Menüs und versuchte, die Anwendung zum Absturz oder zu offensichtlichen Fehlern zu bringen. (Das war auch deswegen praktisch, weil solche nicht zu leugnenden Fehler dafür sorgten, dass die Tester ihre Metriken schnell erfüllten.)

Persönliches Exemplar für Thomas Bartholomäus

Was dabei allerdings unter den Tisch fiel, war die Frage, ob die Business-Anforderungen überhaupt erfüllt werden. Das ist schließlich das Wichtigste für den Endnutzer des Systems. Wenn Anforderungen nicht richtig umgesetzt wurden, fiel es immer erst im allerletzten Schritt auf, nämlich wenn die Änderungen den Sales- und Marketing-Leuten präsentiert wurden, die sie in Auftrag gegeben hatten. Das sorgte oft genug für Fundamentaldiskussionen, wenn sich zeigte, dass ein Feature, an dem nun mehrere Monate lang gearbeitet worden war, nicht so aussah, wie es sich das Business-Team vorgestellt hatte.

Dieses Feedback ging also im letzten Moment zurück zum Entwicklungsteam, das diese Informationen erst etliche Monate nach der eigentlichen Entwicklung bekam. Der ganze Prozess musste also von vorn losgehen: Das Entwicklungsteam arbeitete erneut am Code, der eigentlich vor einigen Monaten abgeschlossen war, korrigierte diesen, baute wieder eine Version für das QA-Team, das sie dann erneut prüfte (und ggf. noch mal Fehler fand), damit dann wieder die fachliche Abnahme stattfinden konnte.

Erst nach der fachlichen Abnahme mit vielen Verschiebungen und Umgebungen konnten die Pakete der einzelnen Module an das Betriebsteam übergeben werden, um das Deployment zu starten.

7.1.3 Die Anzahl der Fehler als Metrik

Ich habe bereits angerissen, dass man bei *schick-gekleidet.de* den Erfolg des QA-Teams daran misst, wie viele Fehler es findet. Diese Metrik ist – wenig überraschend – nicht gut. Denn selbst, wenn der unmögliche Fall auftreten sollte, dass ein Entwicklungsteam Software programmiert, die keinerlei Fehler hat, dann muss auch hier das QA-Team sehr viel Arbeit leisten, um das Ganze auch zu verifizieren. Und wenn das QA-Team nichts finden würde, dann würde es entsprechend negativ bewertet, da es ja keine Fehler gefunden hat.

Zugegeben: Jedem sollte klar sein, dass jede Software Bugs hat. Aber selbst bei fast perfekter Software wird das Team viel Zeit und Arbeit investieren müssen, um alles zu kontrollieren. Und wenn es nichts Gravierendes findet, heißt das keineswegs, dass die Teammitglieder schlechte Arbeit geleistet haben.

Den Teams bei *schick-gekleidet.de* half das allerdings nicht: Die einzelnen Teams und Personen wurden nun mal daran gemessen, wie viele Fehler sie fanden. Diese Metrik wurde in der Chefetage festgelegt, sodass die Leidtragenden die normalen Mitarbeiter und Mitarbeiterinnen waren. Hier zeigt sich wieder, dass die Kultur der DevOps-Transformation auf allen Ebenen gelebt werden muss: Falsche Ziele zur Messung des Erfolgs führen zu falschen Arbeitsweisen in und zwischen den Teams.

© Rheinwerk Verlag, Bonn 2024

Metriken und Ziele

Das große Problem bei Metriken ist, dass sie leicht mit Zielen verwechselt werden. Oder – fast noch schlimmer – dass sie schleichend zu Zielen werden. Natürlich ist es ein gutes Ziel, dass Software möglichst fehlerfrei sein sollte. Und das Zählen von gefundenen Bugs ist auch eine sinnvolle Metrik. Aber wenn diese beiden Punkte vermischt werden, leiden letztlich sowohl die Qualität als auch die Statistik.

Als Analogie: Entwickler sollen Code schreiben, das ist das Ziel. Und Codezeilen kann man ganz einfach zählen. Kommen Sie aber bloß nicht auf die Idee, beide Punkte miteinander zu vermischen und aus der Messung das Ziel zu machen! Es ist nämlich ganz bestimmt kein gutes Ziel, möglichst viele Zeilen Code zu produzieren ...

7.1.4 Is it fixed yet?

Zurück zum Testen: Wenn das QA-Team einen Fehler findet, dann ergeben sich hieraus einige langwierige Prozesse, die zu weiteren Wartezeiten und Kontext-Switchen führen: Wenn eine Person aus dem QA-Team ein Ticket für einen Fehler aufmacht, dann befindet sich diese Person gerade in dem Kontext des Fehlers. Nachdem ein Ticket aufgemacht, aber nicht bearbeitet wurde, liegen dieser Fehler und der Kontext dazu erst einmal brach, bis innerhalb der nächsten Wochen eine Fehlerkorrektur bereitsteht.

Der QA-Mitarbeiter widmet sich also dem nächsten Aspekt und vergisst darüber allmählich den Kontext der Fehlersituation, weil unmöglich alles dokumentiert werden kann. Nachdem dann die Korrektur bereitsteht, ist dieser Kontext erneut verloren gegangen, da einiges an Zeit verstrichen ist.

Das ist dann jeweils nicht nur ein Problem für das Mitglied des QA-Teams, sondern auch für den Entwickler: Auch dieser weiß aktuell nicht mehr viel von der Implementierung, die er damals getätigt hat. Beide Personen müssen sich also jedes Mal wieder neu in das Problem und den Kontext hineindenken, um den Fehler richtig zu korrigieren. Beides führt dazu, dass viel Zeit verstreicht, da die Kontext-Switche viel zu häufig sind.

Da diese Prozesse bei *schick-gekleidet.de* so langsam liefen, musste aufwendig überprüft werden, ob gemeldete Fehler gefixt wurden, denn es fehlte schlicht die Sichtbarkeit – das kennen Sie schon vom Build-Prozess. Dort ging es um den einfachen Fall, ob der Build erfolgreich war oder nicht. Beim Testen wiegt diese fehlende Sichtbarkeit noch schwerer. Da beide Abteilungen getrennt voneinander arbeiteten, wussten die Tester erst mit Abschluss des Entwicklungsprozesses, woran gerade gearbeitet wurde, und das Entwicklungsteam hatte keinen Einblick in die Art und Weise, wie das QA-Team die Tests durchführt.

Persönliches Exemplar für Thomas Bartholomäus

Das führte wiederum häufig dazu, dass das Entwicklungsteam einige Fehler nicht nachstellen konnte, da nicht alle Informationen von den verschiedenen Systemen der Teams für alle zugreifbar waren. Einige QA-Teams arbeiteten etwa mit Werkzeugen für das Test-Management, bei denen etliche manuelle Tests mit jedem Zwischenschritt dokumentiert werden. Auf diese Tools hatten viele Entwicklungsteams allerdings keinen Zugriff, sodass wieder kein Weg an manuellem Herauskopieren vorbeiging, wenn man sicherstellen wollte, dass alle Informationen übergeben worden waren – was natürlich in der Praxis nie richtig klappte.

Reflexion

Dass Teams in eigenen Silos arbeiten, ist ein leidiges Thema, auch bei der Qualitätssicherung. Fehlerhafte Anreize und schlechte Kommunikation sorgen dafür, dass man sich länger mit Problemen herumschlägt, als es eigentlich sein müsste.

Sie sollten sich anschauen, ob es in Ihrer Organisation Metriken oder Fehlanreize gibt, die es zu vermeiden gilt. Und stärken Sie den Gedanken, dass gefundene Fehler etwas Gutes sind und nicht nur im Aufgabenbereich der Tester liegen.

7.2 Testen als Teil des DevOps-Prozesses

Das grundsätzliche Ziel von Tests im DevOps-Konzept ist, Fehler frühzeitig zu finden, bevor diese auf Produktivsystemen Ärger machen. Durch das frühe Korrigieren der gefundenen Fehler wird die Qualität sichergestellt, was dann wiederum häufige Deployments mit deutlich geringerem Risiko ermöglicht.

In der DevOps-Acht wird das Qualitätssicherungsteam noch vor dem Betriebsteam (Ops) in den Software-Development-Lifecycle eingebunden. Denn DevOps ist nicht, wie der Name suggeriert, ein reiner Zusammenschluss von Dev und Ops, sondern eine generelle Philosophie zur Zusammenarbeit.

Je nachdem, wie groß das Unternehmen ist, gibt es verschiedene Möglichkeiten, das QA-Team in den Entwicklungszyklus einzubinden. Grundsätzlich gilt, dass so früh wie möglich getestet werden sollte, ganz unabhängig davon, wie die Rollenbezeichnung am Ende ist.

In kleineren Unternehmen ist diese Integration oftmals nicht notwendig, da keine individuellen QA-Teams existieren, sondern sich das Entwicklungsteam um die Tests kümmert. Allerdings gibt es auch in solchen Firmen einige technische und kulturelle Aspekte, die beachtet werden müssen, insbesondere im Hinblick auf die Automatisierung.

In größeren Unternehmen (wie bei *schick-gekleidet.de*) mit einem eigenständigen QA-Team sorgt die DevOps-Transformation für größere Änderungen. Hier wird näm-

© Rheinwerk Verlag, Bonn 2024

lich die Test-Abteilung aufgeteilt. Jedes Entwicklungsteam bekommt mindestens eine Person aus dem ursprünglichen QA-Team zugewiesen, die als Experte rund um die Qualität bereitsteht und die relevanten Test-Aufgaben innerhalb des Teams übernimmt.

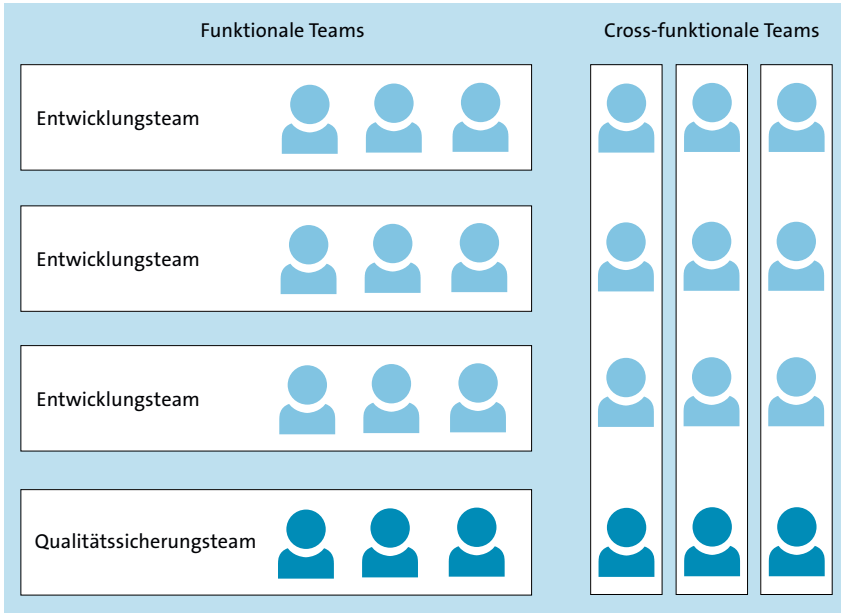

Abbildung 7.3 Funktionale Teams vs. cross-funktionale Teams: Das ursprüngliche Qualitätssicherungsteam wird aufgeteilt. Seine Mitarbeiter werden jeweils Teil von cross-funktionalen Teams.

Das ursprüngliche QA-Team verschwindet allerdings nicht vollständig. Es bleibt als zweites, separates Team innerhalb der Firma vorhanden. Das kann so funktionieren, dass das QA-Team weiterhin den Dev-Teams zuarbeitet und bei Bedarf Aufgaben übernimmt. Oder seine Mitglieder verstehen sich als Teil des Entwicklungsteams und arbeiten zusätzlich im reinen QA-Team an gemeinsamen Themen, die jenseits der reinen Entwicklungsarbeit benötigt werden. Sie sind grundsätzlich verantwortlich für alle Qualitätssicherungsfragen und sind in dieser Rolle dafür zuständig, ausgiebige Test-Konzepte für das restliche Team zu entwickeln und auch den Testprozess selbst zu optimieren.

Wie genau die Teamzugehörigkeit umgesetzt wird, ist allerdings fast nur eine Frage für das Organigramm. Wichtiger ist zu schauen, wie sich die Rolle und somit die Aufgaben der QA-Mitarbeiter ändern, die nun Teil von zwei Teams sind: einmal weiterhin im QA-Team und dann eben jeder für sich in den bisherigen Entwicklungsteams.

Es ist oftmals so, dass viele ähnliche Probleme in der Entwicklung auftreten, nur mit zeitlichem Verzug. Anstatt dass jedes Team eigene Lösungen evaluiert und einführt,

Persönliches Exemplar für Thomas Bartholomäus

ist es sinnvoller, wenn halbwegs einheitlich gearbeitet wird und die Erkenntnisse der anderen Teams genutzt werden. Ein Entwickler möchte etwa verschiedene Test-Frameworks ausprobieren, um zu schauen, welches am besten passt. Anstatt dies allein an der eigenen Code-Basis auszuprobieren, sollte dies zusammen mit den Experten aus der QA-Abteilung geschehen.

Damit verändern sich auch die Rolle, die Verantwortung und die Aufgaben des bisherigen reinen Qualitätssicherungsteams. Durch die Einrichtung der cross-funktionalen Teams, in denen ein Teammitglied mit QA-Fokus beim täglichen Arbeiten zwar im Entwicklungsteam sitzt, aber im engen Kontakt zu anderen Teammitgliedern mit QA-Fokus aus den anderen Teams steht, kann auf kurzem Wege Knowledge-Sharing betrieben werden (siehe Abbildung 7.4). Das ursprüngliche Team kann also direkt von den Fehlern und Erkenntnissen der anderen Teams lernen, begünstigt durch den offenen Austausch über Team-Grenzen hinweg.

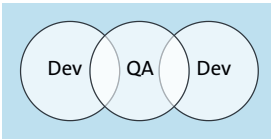

Abbildung 7.4 Dev und QA und Dev überschneiden sich.

Betrachten wir das unter den Gesichtspunkten der DevOps-Prinzipien und des CALMS-Modells, kommen allein durch diese Änderung der Teamstruktur zwei Aspekte direkt zu Vorschein: *Culture* und *Sharing*.

7.2.1 Tests in der Build-Pipeline

Bis jetzt haben wir uns also das **C** und das **S** des CALMS-Modells angeschaut. Kümmern wir uns nun um das **A**, die Automatisierung.

Durch die Integration der Teammitglieder mit Qualitätssicherungsfokus in das bisher reine Entwicklungsteam eröffnen sich einige Vorteile. Das Hauptproblem des ehemaligen QA-Teams war ja, dass es erst am Ende der Entwicklungsphase mit seiner eigentlichen Arbeit beginnen konnte und bis dahin keinen Einblick in den Quellcode hatte. In der Testphase mussten die Mitglieder des QA-Teams also immer ihre Test-Skripte den Änderungen entsprechend korrigieren und dadurch viel händisch testen.

Es ist natürlich nicht sehr sinnvoll, so weiterzuarbeiten. In der DevOps-Welt soll das ordentliche Testen der Standard werden. Fehler sollen so früh wie möglich gefunden und korrigiert werden, um die zuvor erwähnten langen Zyklen zum Korrigieren der Fehler drastisch zu verkürzen. Eine saubere Software mit möglichst wenigen Fehlern und vor allem mit nachvollziehbaren automatisierten Tests ist die Grundbedingung

© Rheinwerk Verlag, Bonn 2024

dafür, dass das Team später auch willens ist, die Software auf Produktivsysteme auszurollen. Denn niemand würde freiwillig etwas auf Produktivsysteme ausrollen wollen, wenn er überhaupt kein Gefühl dafür hat, ob Fehler ausgeschlossen werden können.

In Kapitel 6 haben Sie bereits gelernt, wie eine Build-Pipeline arbeitet. In dieser Pipeline wurde bisher nur das Projekt reproduzierbar gebaut. Nach dem erfolgreichen Build ist das Ausführen von Tests elementarer Bestandteil der Pipeline. Die Tests laufen direkt, wenn der Build-Prozess erfolgreich war. Dadurch wird gewährleistet, dass das Teammitglied frühzeitig Feedback erhält.

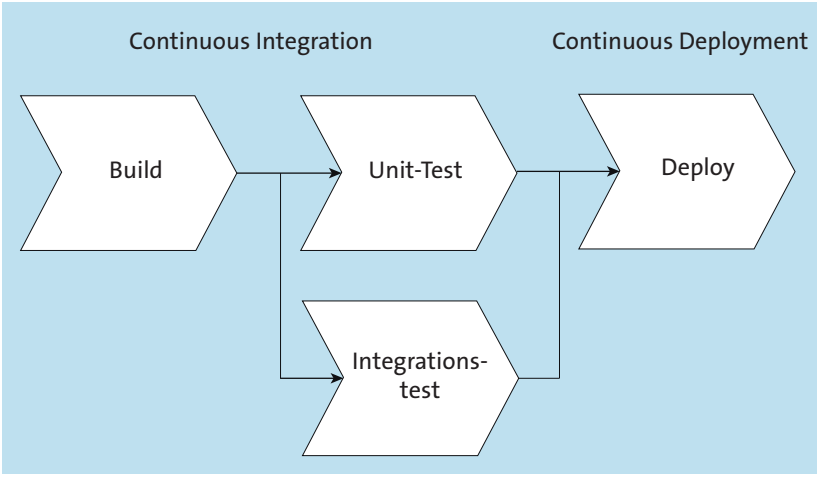

Abbildung 7.5 Die CI/CD-Pipeline besteht aus Build, Tests und Deploy.

Aber treten wir zunächst noch einmal einen Schritt zurück: Das erste Problem bei *schick-gekleidet.de* war ja, dass das QA-Team keinen Zugriff auf das Repository hatte. Durch die Öffnung des Repositorys und die Ergänzung des Teams um QA-Spezialisten wurde dieses Problem auf den ersten Blick direkt angegangen.

Die jeweiligen Personen können nun direkt sehen, welche Änderungen vorgenommen werden, und direkt die automatischen und manuellen Testprozesse anpassen. Außerdem: Je näher an der Quelle man die Tests ausführen kann, desto einfacher lassen sich Abhängigkeiten zwischen der Anwendung und den Tests frühzeitig aufdecken. Wenn das nicht gegeben ist und die Tests nur periodisch oder selten ausgeführt werden, hat man immer noch keinen Vorteil geschaffen. Wir schauen uns dazu in Abschnitt 7.2.3 an, wie Tests so in die CI-Pipeline integriert werden, dass sie automatisch ausgeführt werden und nicht vergessen werden können.

So richtig gelöst wird das Problem durch die bisherigen Maßnahmen aber noch nicht: Das neue QA-Teammitglied ist zwar jetzt Teil des Teams, allerdings wurden keine Prozesse angepasst, und es wurde kein kulturelles Onboarding durchgeführt.

Persönliches Exemplar für Thomas Bartholomäus

Die Entwickler im Team arbeiten also zunächst erst mal so weiter wie bisher. Ob nun die eine oder andere Person mit Fokus auf der Qualitätssicherung mit im Team ist, ist für die Entwickler zunächst irrelevant, da in ihren Köpfen immer noch die Auffassung vorherrscht, dass das QA-Teammitglied ja nur Fehler finden möchte, um die Entwickler damit zu nerven.

Tatsächlich sollte sich der Fokus des QA-Teammitglieds durch die Inkludierung in das bisher reine Entwicklungsteam verändern. Es geht nun nicht mehr allein darum, möglichst viele Fehler vor dem Deployment zu finden, sondern darum, dem Team zu helfen, Probleme so früh wie möglich zu identifizieren. Außerdem müssen die Qualitätsspezialisten den Entwicklern zeigen, wie die Prozesse und die Software angepasst werden sollten, um offensichtliche Fehler zu vermeiden – und das so früh im Prozess wie möglich. Das QA-Teammitglied ist also der Enabler der Test-Konzepte und auch Subject-Matter-Experte im Team rund um das Thema Qualitätssicherung.

Dazu zählt beispielsweise, dass die Tests beim Refactoring von Code direkt mitgedacht und angepasst werden. Schrittweise kann man sich so zum *Test-Driven Development* (siehe Abschnitt 7.2.4) vorarbeiten. Das kann aber nur funktionieren, wenn die Entwickler die QA-Fachleute direkt in ihre Entwicklungsarbeit miteinbeziehen.

Der nächste Punkt ist, dass der Test-Aufbau so gestaltet sein muss, dass Tests so einfach wie möglich ausgeführt werden können. Das betrifft nicht nur den Arbeitsworkflow für den einzelnen Entwickler und die einzelne Entwicklerin, sondern auch die Automatisierung als Teil der CI-Pipeline. Denn die besten Tests bringen nichts, wenn sie nie ausgeführt werden.

Die Dauer eines Test-Durchlaufs

Idealerweise entspricht der Run durch eine CI-Pipeline mit Build- und Testvorgängen etwa einer Kaffeepause oder einem Toilettengang. Dies ist allerdings nicht bei jeder Programmiersprache und jedem Projekt möglich, daher muss man immer wieder abwägen, was das größere Übel ist: länger warten, dafür Fehler frühzeitig im Prozess finden – oder zügiger die nächsten Entwicklungen angehen, weil gravierende Fehler sowieso nicht so schnell ersichtlich werden?

7.2.2 Unterschiedliche Tests für unterschiedliche Aufgaben

Zunächst sehen wir uns aber erst einmal die verschiedenen Arten von Tests an, die nach Möglichkeit vollständig automatisiert ablaufen sollten, damit man möglichst viele Fehler rechtzeitig findet:

- Unit-Tests
- Integrationstests
- Systemtests
- Akzeptanztests

© Rheinwerk Verlag, Bonn 2024

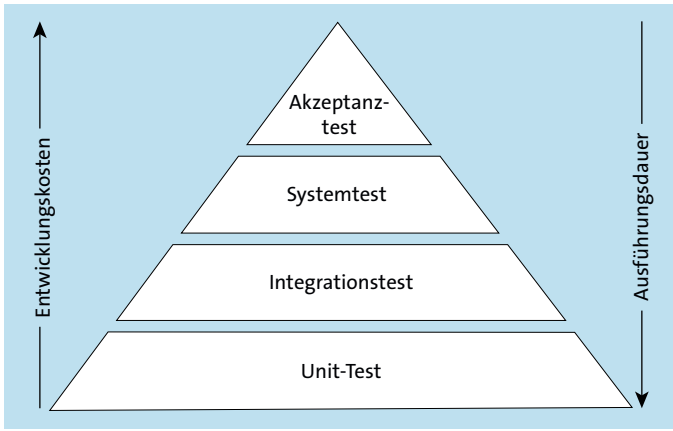

Abbildung 7.6 Die Testpyramide zeigt, wie die Tests aufeinander aufbauen, auch in Hinblick auf ihre Entwicklungskosten und Ausführungsdauer.

Unit-Tests

Unit-Tests sind die kleinsten Tests, die in der Regel nicht nur schnell zu schreiben sind, sondern sich auch rasch ausführen lassen. In einem Unit-Test wird lediglich eine Funktion oder Methode einer Klasse getestet. Unit-Tests sind dafür da, schnell und zügig elementare Fehler in der Implementierung der gerade implementierten Methode zu finden. Denn wenn sich schon auf diesem niedrigen Level Fehler einschleichen, die nicht auffallen, dann ist ein Debuggen an späterer Stelle aufwendig und somit mühsam. Daher gibt es idealerweise zu jeder Methode, die man schreibt, einen passenden Unit-Test.

Wenn Sie schon einmal in einer älteren Codebasis, in der es keine oder nur wenige Unit-Tests gibt, Tests ergänzen mussten, wird Ihnen aufgefallen sein, dass die Verständlichkeit des Codes direkt damit korreliert, wie einfach es ist, Unit-Tests zu schreiben. Zu lange und zu komplizierte Methoden, die man nicht auf Anhieb versteht, kann man schlecht testen – das gilt besonders dann, wenn eine Methode gleich mehrere Aufgaben erledigen soll. Zur Erhöhung der Lesbarkeit und Testbarkeit ist ein Refactoring von bestehendem Quellcode also meistens unabdingbar. (Für ein Brownfield-Projekt, was schon länger existiert und älter ist, wird wohl einiges an Refactoring notwendig sein, um gescheite Unit-Tests schreiben zu können.)

Das Schreiben von Unit-Tests hat somit auch einen positiven Effekt auf die Lesbarkeit und die Qualität des Quellcodes. Wenn man Probleme hat, einzelne Methoden mit ihrer implementierten Funktion zu testen, dann ist dies ein guter Indikator, dass diese Methode aufgebrochen und aufgeteilt werden muss.

Das Problem von Unit-Tests ist, dass sie wirklich nur diese eine Methode testen und nicht mehr. Abhängigkeiten von anderen Objekten, die benötigt werden, müssen als

Persönliches Exemplar für Thomas Bartholomäus

Mock-Objekte abstrahiert werden, damit möglichst nur die Methode der zu testenden Klasse getestet wird.

Integrationstests

Um mehr als die einzelne Methode einer Klasse zu testen, gibt es auf der nächsten Stufe die *Integrationstests*. Diese gehen einen Schritt weiter als Unit-Tests, da sie mehrere Komponenten einer Anwendung testen. Der Aufwand, diese Tests zu schreiben, ist höher als bei Unit-Tests, da hier systematischer getestet werden muss. Abhängigkeiten innerhalb einzelner Komponenten und Module müssen zunächst aufgebaut werden.

Integrationstests sind insbesondere deswegen sinnvoll, weil erst so die kombinierte Funktionalität der verschiedenen Klassen sichtbar wird. Je mehr Komponenten in einem Integrationstest benötigt werden, desto komplexer ist das vollständige Testen, da sehr viele verschiedene Aspekte in einer Matrix abgebildet und getestet werden müssen. Das ist der Hauptgrund, warum das Schreiben von Integrationstests teurer ist als von Unit-Tests und warum ihre Ausführung in der Regel etwas länger dauert.

Systemtests

Sowohl Unit- als auch Integrationstests können nur laufen, wenn die Anwendung baubar ist. Die Tests werden also gegen die gebaute Software ausgeführt. Im nächsten Schritt nach diesen beiden Testtypen folgt in der Regel der *Systemtest*. Für diesen Test muss die Anwendung allerdings laufen, da die ganze Anwendung gestartet wird.

Wichtig ist also im ersten Schritt, dass die Anwendung gestartet und deployt werden kann. Bei dem Online-Shop von *schick-gekleidet.de* müssen also alle Module von den verschiedenen Teams lauffähig sein, damit man das Deployment durchführen kann, um dann die Systemtests auszuführen.

Diese lassen sich ebenfalls durchautomatisieren – etwa durch Tests, die über das User-Interface gehen, um einmal einen vollständigen End-to-End-Test durchzuführen. Durch den Aufbau der Testumgebung und das Starten des Online-Shops in diesem konkreten Beispiel dauert nicht nur der Start der Testumgebung lange, sondern auch die Testdurchläufe nehmen mehr Zeit in Anspruch, da sich die Wartezeiten durch die Ladezeiten aufaddieren.

Dementsprechend sind auch die Entwicklungskosten hoch, weil diese Bereiche häufiger angepasst werden müssen. Das ist etwa jedes Mal dann der Fall, wenn eine Änderung an der Oberfläche erfolgt, die vielleicht nicht die Funktionalität ändert, aber dafür sorgt, dass das Mapping für die automatischen UI-Tests nicht mehr funktioniert.

© Rheinwerk Verlag, Bonn 2024

Akzeptanztests

Die einzigen Tests, die eher nicht automatisch erfolgen können, sind die *Akzeptanztests*. Diese werden häufig *User Acceptance Tests*, kurz UAT, oder auch *Abnahmetests* genannt. Diese Tests sollen prüfen, ob die Implementierung so erfolgt, wie es gewünscht war. Hier können Fehler auftreten, da eine Funktion zwar technisch korrekt implementiert sein kann, aber die Umsetzung nicht den eigentlichen Anforderungen entspricht. Entsprechend helfen dann auch die vorher komplett durchautomatisierten Unit-, Integrations- und Systemtests nicht.

Bei einem Akzeptanztest wird meistens direkt in der grafischen Oberfläche gearbeitet, indem man sich automatisiert durch die Anwendung klickt, um zu schauen, ob alles wie gewünscht funktioniert. Die Herausforderung ist, dass man auch schauen muss, ob dieser Akzeptanztest richtig durchgeführt wird. Dabei ist es erst einmal egal, ob manuell oder automatisiert vorgegangen wird. Die Frage ist eher: Wie bindet man den Auftraggeber ein und wie sammelt man das Feedback? Damit bewegen Sie sich schon im Bereich des *Requirements Engineerings*, das nicht mehr Thema dieses Buchs ist.

Unterschiedliche Programmiersprachen für Tests und Entwicklung

Ein häufiges Problem für die effiziente und effektive Zusammenarbeit zwischen Entwicklungs- und QA-Team ist, dass verschiedene Programmiersprachen genutzt werden. So ist die eigentliche Anwendung in Java geschrieben, während Akzeptanztests typischerweise eher in Skriptsprachen wie Python oder JavaScript geschrieben werden. Auch wenn APIs getestet werden sollen, ist es eigentlich besser, wenn dazu die Sprache der Anwendung verwendet wird. Oft ist es aber einfacher und schneller, Skriptsprachen für die Tests zu nutzen.

Für die Kollaboration ist das ungünstig, da nicht alle Entwickler jede Programmiersprache beherrschen, was es vielen Entwicklern erschwert, die Tests selbstständig zu schreiben. In vielen Firmen gibt es ohnehin schon zu viele verschiedene Programmiersprachen, was die Kollaboration einschränkt. Wenn nun auch noch innerhalb eines Teams zwischen Sprachen gewechselt wird, dann hilft das noch weniger.

Idealerweise schaut man also, dass möglichst die gleichen Programmiersprachen verwendet werden, damit man sich auf das Schreiben der Tests konzentrieren kann, ohne zwangsläufig eine neue Sprache lernen zu müssen.

7.2.3 Tests automatisieren

Tests zu schreiben und ausführen zu lassen, ist schön und gut. Wichtig ist aber auch, dass man die Testdauer optimiert. Viel zu lange laufende Tests sind zu vermeiden, wenn man schneller und häufiger Änderungen beitragen möchte.

Persönliches Exemplar für Thomas Bartholomäus

Auch die Analyse des Codes – darunter fallen sowohl die Code-Qualität als auch das Performance- und Load-Testing – kann in die Pipeline eingebunden werden. Manuelle Tests sollten also möglichst komplett automatisiert werden, denn sonst wird die *Lead Time* nicht angemessen auf ein niedriges Niveau heruntergeschraubt. Komplett vermeiden kann man manuelle Tests allerdings nicht in jeder Umgebung. Vor allem im Finanzsektor sieht man häufig, dass sehr viel manuell getestet wird – teilweise, weil man es nicht anders kennt und sich auf End2End-Tests konzentriert, teilweise aber auch, weil man aufgrund der Regulatorik manuell testen muss.

Testautomatisierung in der Pipeline

Die Automatisierung der Tests erfolgt in der CI/CD-Pipeline. Das Feedback, ob die Tests durchgelaufen sind, erfolgt durch das mögliche Fehlschlagen des entsprechenden Jobs in der Pipeline.

Die gängigen Test-Frameworks generieren in der Regel eine XML-Datei im JUnit-Format. Diese XML-Dateien enthalten Informationen über die ausgeführten Testfälle, zu welchen Zeilen sie gehören und ob und welche Tests fehlgeschlagen sind. Die verschiedenen CI/CD-Tools können diese XML-Dateien einlesen, grafisch aufbereiten und auswerten. Wichtig für das große Ganze ist, dass alle Beteiligten rasch sehen, wie viele Tests erfolgreich waren und wo es Fehler gibt. Tests, die Fehler zurückliefern, müssen dann entsprechend in der Pipeline als fehlgeschlagen markiert werden und deutlich sichtbar sein.

Tests, die zwar ausgeführt werden, aber nicht ausgewertet werden, sind witzlos und sollten tunlichst vermieden werden. Denn niemand wird in die Logs der ausgeführten Tests in den Pipeline-Jobs aktiv hereinschauen.

Zu weiteren Konfigurationsfragen zählt, ab wann der gesamte Run der Pipeline als fehlgeschlagen markiert werden soll. Sie können nämlich bei den meisten CI/CD-Tools einstellen, dass es nicht nur einen »grünen« Modus und einen »roten« Modus gibt, sondern auch einen »gelben« Modus für den Fall, dass nur wenige Tests fehlschlagen oder nur Warnungen aufgetreten sind.

Obwohl dieses Feature sicherlich seine Berechtigung hat, bin ich kein Fan von ihm. Tests sollten immer korrekt durchlaufen, ohne Ausnahme. Wenn man eine Anzahl oder gar eine Prozentzahl definiert, unter der es fehlschlagende Tests geben darf, führt man das Konzept ad absurdum. Einige Tests sind zwar wichtiger als andere, aber ob das nun gravierende Fehler sind oder nicht, ist pauschal fast unmöglich einzuschätzen.

Flackernde Tests

Ein eher nerviges Thema sind sogenannte »flackernde« Tests. Das sind Tests, die gefühlt nach Lust und Laune mal erfolgreich durchlaufen oder mal eben nicht. Das

© Rheinwerk Verlag, Bonn 2024

kann für den Entwickler ziemlich nervig werden, da fehlschlagende Pipelines wegen einzelner fehlschlagender Tests eher keinen Mehrwert bringen. Ich kann Ihnen nur empfehlen, solche Tests entweder endgültig zu fixen oder auszubauen. Das Fixen heißt dann allerdings auch, dass ein Problem in der Umgebung liegen kann, in der die Tests ausgeführt werden.

Test-Coverage

Eine häufige Metrik für die Messung, ob gut getestet wird, ist die *Test-Coverage*. Auf Deutsch spricht man auch von *Testabdeckung*. Diese Metrik gibt an, wie viele Anweisungen, Abzweigungen und Pfade im Code von Tests während der Testausführung abgedeckt werden. Grundsätzlich wird angenommen, dass eine hohe Test-Coverage besser ist. So pauschal würde ich das allerdings nicht sagen wollen.

Ein wesentlicher Vorteil einer hohen Testabdeckung ist, dass eben sehr viele Pfade im Code getestet werden. Fehler werden also ziemlich sicher auffallen, erst recht, wenn der Code regelmäßig verändert wird.

Eine weitere Konsequenz einer hohen Testabdeckung ist, dass die Stabilität und Robustheit erhöht werden. Insbesondere im Hinblick auf das kontinuierliche Ausrollen von Änderungen ist das immens wichtig.

Der bloße Blick auf die Testabdeckung ist allerdings kein Allheilmittel, da sie zwar angibt, wie *viel* getestet wird, aber nicht, wie *gut* die Tests sind. Ernennen Sie also nicht eine hundertprozentige Coverage zum Ziel. Das ist genauso wenig zielführend, wie nach der absoluten Zahl der gefundenen Fehler zu fragen.

Aussagekräftiger ist auch hier oft die *Tendenz*. Wenn sich die Testabdeckung über einen längeren Zeitraum verschlechtert, deutet das darauf hin, dass es Probleme geben kann. Um sicherzustellen, dass möglichst umfassend und gut getestet wird, sollten die Tests nicht nur in der Pipeline berechnet, sondern auch sichtbar macht werden. Idealerweise geschieht das direkt in der Entwicklungsumgebung. Einige IDEs unterstützen dies von Haus aus.

Zusätzlich sollte die Test-Coverage auch im Review-Prozess sichtbar gemacht werden. Wenn Sie GitLab nutzen, können Sie die die Test-Coverage im Merge-Request bei einem Code-Review anzeigen. Etwaige Probleme werden also dann spätestens im Review sichtbar.

Testdauer optimieren

Je mehr Tests man schreibt und ausführt, desto länger wird die Gesamtlaufzeit der Pipeline. Häufig lässt sich die Ausführung der Tests allerdings stark parallelisieren. Und wenn das nicht so einfach geht, dann sollte man das Test-Setup so umbauen, sodass es möglich wird.

Persönliches Exemplar für Thomas Bartholomäus

Beispiel: Statt 1 × 10.000 Tests nacheinander auszuführen, was insgesamt 20 Minuten dauert, sollte man schauen, dass man möglichst 4 × 2.500 Tests ausführt, um die Laufzeit auf sechs oder sieben Minuten zu reduzieren. Wer sich jetzt wundert, warum es nicht fünf Minuten sind: Auch das Provisionieren der Test-Infrastruktur, der Download des Repositorys und die Vorbereitung für die Ausführung der Tests kosten etwas Zeit, was man nicht vernachlässigen sollte.

Eine skalierbare Container-Infrastruktur kann Ihnen helfen, mehr Testprozesse gleichzeitig auszuführen. Diverse Caching-Mechanismen können zusätzlich dazu beitragen, dass der Start der Tests nicht allzu lange dauert.

Zukünftig ist das Thema *Observability*, um das es in Kapitel 10 geht, auch für die Optimierung von Pipelines und Tests hilfreich. Denn so lassen sich womöglich etwaige Bottlenecks schneller ausfindig machen.

Die Code-Qualität mit statischer Code-Analyse sicherstellen

Für die allgemeine Code-Qualität und besonders für die Security ist die statische Code-Analyse relevant. Dabei wird der Code analysiert, ohne ihn auszuführen – statisch eben.

Wie auch bei der Test-Coverage ist hier die Identifizierung von Inkonsistenzen und Syntaxfehlern wichtig, bevor der Code in der Produktivumgebung landet. Daher ist die statische Code-Analsye auch ein wichtiger Bestandteil der CI-Pipeline, um solche Fehler frühzeitig zu erkennen und somit das Risiko zu minimieren.

Auch für die Kollaboration ist es hilfreich, wenn die Code-Qualität gut ist: Ein einheitliches Code-Schema und gut lesbarer Code sind auch für Code-Reviews hilfreich und sind generell für die zügige und idealerweise fehlerfreie Weiterentwicklung relevant.

Wie auch bei der Test-Coverage ist hier die Tendenz entscheidend. Meldungen in der statischen Code-Analyse sollten zwar grundsätzlich zeitnah bearbeitet werden, aber nicht alle Meldungen sind wichtig. Auch hier sollte man darauf achten, dass sich die gesamte Bilanz der statischen Code-Analyse in der Tendenz positiv entwickelt.

Einfache Scans direkt beim Schreiben des Codes werden als *Linter* bezeichnet. Der relevante Punkt ist, dass eine statische Analyse durch ein Programm erfolgt. Idealerweise wird dies also sowohl lokal in der Entwicklungsumgebung als auch über den CI-Server angestoßen. Beides ist relevant, und zwar mit unterschiedlichen Ansätzen. Ein früheres Scannen in der IDE ist praktisch, damit man schon beim Programmieren sieht, wenn unsauberer Code geschrieben wird. Das ist derselbe Level wie die Rechtschreibkorrektur in einer Textverarbeitung.

Damit diese Regeln einheitlich angewendet werden und für alle Beteiligten gelten, sollten Linter auch über den CI-Server getriggert und die Ergebnisse ausgewertet werden.

© Rheinwerk Verlag, Bonn 2024

Ein verbreitetes Tool hierfür ist *SonarQube*. Es existiert sowohl in einer freien Open-Source-Variante als auch in einer kommerziellen Variante mit mehr Features. Für das lokale Scannen in der IDE gibt es ein eigenes Tool namens *SonarLint*.

Das Praktische an SonarQube ist, dass es sehr viele verschiedene Programmiersprachen unterstützt. Somit ist der Einstieg in die Nutzung relativ einfach. Zudem existieren für die verschiedenen CI-Server Plugins und Integrationsmöglichkeiten, sodass die Ergebnisse schnell sichtbar werden.

Eine weniger verbreitete Alternative ist *Code Climate*. Auch hier sieht man das übliche Muster: Code Climate steht in der Basis-Ausstattung unter einer Open-Source-Lizenz, weitere Features sind hingegen kommerziell. Es wird eine große Bandbreite an Programmiersprachen unterstützt.

Die Code-Qualität ist allerdings nicht nur für reinen Quellcode relevant. Auch für Konfigurationsdateien oder Dateien für Infrastructure-as-Code sind Linter üblich und nützlich. Auch für Terraform-Code oder Dockerfiles kann es hilfreich sein, Linter zu verwenden, da Sie direkt Hinweise auf empfohlene Vorgehensweisen erhalten, die Sie sonst vielleicht mühselig selbst herausfinden müssten.

Load- und Performance-Testing

Im Rahmen des Testings können auch Performance-Tests ausgeführt werden. Hier gibt es verschiedene Arten von Performance-Tests. Ein typischer Anwendungsfall ist eher ein Load- oder Performance-Test, bei dem die Anwendung ausgelastet wird, um zu schauen, was für Auswirkungen das hat. Ein Tool hierfür ist etwa *k6*, das zu Grafana Labs gehört. Aber auch andere Tests sind möglich, etwa Stresstests oder Scalability-Tests.

Solche Tests werden nicht bei jedem Pipeline-Durchlauf auf jedem Feature-Branch ausführt, dafür dauern sie viel zu lange. Stattdessen sollten diese Tests nur auf dem Entwicklungsbranch ausgeführt werden. Aber auch hier gelten Einschränkungen, denn auch eine störrische Pipeline auf dem Hauptentwicklungszweig sollte vermieden werden. Stattdessen ist es besser, diese Tests bei einem Nightly Build ausführen, sodass man immer noch relativ zeitnah Feedback bekommt, ob es zu starken Performance-Einbußen gekommen ist.

Aber auch hier gilt: Es kommt wieder erneut stark darauf an, wie das Setup gebaut ist. Wenn man ohnehin »echtes« CI/CD betreibt und mehrfach täglich auf Produktivumgebungen deployt, dann dürften solche Performance-Tests eher eine untergeordnete Rolle spielen. Wenn Sie nur selten deployen, sind solche Tests jedoch sehr wichtig, um die Qualität der Software zu bestimmen.

Nützlich für diese Aufgaben ist beispielsweise *Apache JMeter* (*https://jmeter.apache.org/*), mit dem Sie verschiedene Typen von Anwendungen, Servern und Protokollen testen. Neben dem HTTP-Protokoll für Webanwendungen sind auch etliche weitere Schnittstellen wie REST, SOAP, JDBC oder auch LDAP testbar.

Persönliches Exemplar für Thomas Bartholomäus

Unabhängig davon, welches Tool Sie nutzen: Wichtig ist, dass Sie *sinnvolle* Tests durchführen und diese auch regelmäßig auswerten. Vollumfängliche Performance-Tests nehmen viel Zeit in Anspruch. Und denken Sie an die Feedback-Schleife, denn sonst hilft auch die beste Implementierung von Load- und Performance-Tests nicht.

Typische Fehler zu Beginn sind etwa, dass ein zu starker Fokus auf rein lesende oder rein schreibende Requests gelegt wird. Für typische Anwendungen liegt die Wahrheit irgendwo in der Mitte, da sie sowohl Daten abrufen als auch Daten schreiben. Es sollte also eine gesunde Mischung getestet werden.

Manchmal ist es schlicht sinnlos, jede Änderung zu testen, sondern man sollte den Test lieber parallel einmal die Woche oder auch nächtlich ausführen lassen. Problematisch ist dann nur, dass diese Fehler ebenfalls sichtbar gemacht werden müssen, was sich bei einer separaten Pipeline schwieriger gestalten kann.

Probleme bei Load- oder Performance-Tests können stets unterschiedliche Ausprägungen haben. So funktioniert die Anwendung grundsätzlich, die Last bzw. die Performance ist nur schlechter geworden. Hier ist primär der langfristige Trend wichtig.

Security-Tests

An dieser Stelle ignoriere ich noch die *Security-Tests*. Diese sind ebenfalls wichtig und erlangen stetig höhere Popularität in den Pipelines. Der Grundgedanke ist hier ähnlich wie bei den »normalen« Tests: Probleme sollen möglichst früh gefunden werden. Das ganze Thema rund um Security in DevOps wird allgemein auch als *DevSecOps* bezeichnet. Darum geht es in Abschnitt 11.3.

Testen bei schick-gekleidet.de

In Hinblick auf die Tools gab es bei *schick-gekleidet.de* keine großen Diskussionen: Die verschiedenen CI-Tools unterstützen die Ausführung von Tests. Auch die Parallelisierung der Testausführung durch eine Matrix-Konfiguration, um die Gesamtlaufzeit zu reduzieren, ist bei allen Tools ziemlich gängig, sodass hier keine großen Unterschiede bestanden.

Reflexion

Eine hohe Anzahl von automatisierten Tests mit einer hohen Testabdeckung ist wichtig und notwendig. Achten Sie darauf, dass sich diese Tests schnell ausführen lassen, indem möglichst viel parallelisiert wird. Je schneller die Tests durchlaufen, desto besser, denn niemand wartet gerne auf den Abschluss der Pipeline.

Nutzen Sie weitere Tools in der Pipeline, um stets den Überblick zu behalten, wie es mit der Test-Coverage und Code-Qualität aussieht. Seien Sie nicht zu streng bei der Einhaltung dieser beiden Aspekte: Wichtiger ist es, die Tendenz zu betrachten.

© Rheinwerk Verlag, Bonn 2024

7.2.4 Test-Driven Development

Im Rahmen der Entwicklungstätigkeiten wird in der agilen Software-Entwicklung und daher im DevOps-Umfeld häufig auf *Test-Driven Development* gesetzt. Die Idee dahinter ist, dass zuerst ein Test geschrieben wird und dann erst die eigentliche Funktion implementiert wird. Anstatt zuerst zu entwickeln und dann zu testen, überlegt man sich zunächst den Test und schaut dann, wie man die Anforderung am besten umsetzt (siehe Abbildung 7.7.)

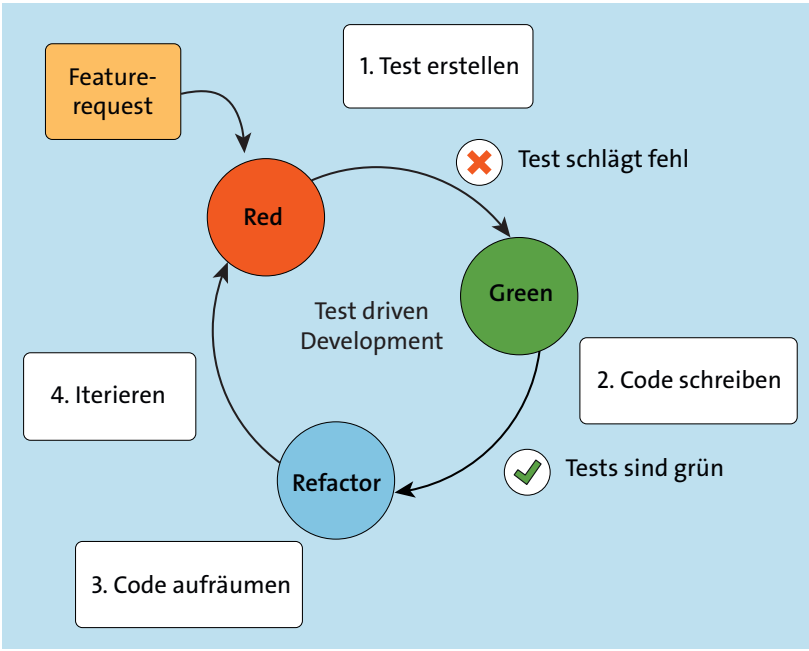

Abbildung 7.7 Test-Driven Development: erst den Test schreiben, dann die Implementierung und dann refaktorieren und iterieren.

So werden zwei Vorteile kombiniert:

- Es wird direkt ein Test-Case geschrieben, der auf Anhieb fehlschlägt, weil die eigentliche Implementierung bislang nicht vorhanden ist.
- Der Test und entsprechend die Implementierung der Funktion wird viel bedachter geschrieben, als wenn man es andersherum machen würde.

So kümmert man sich automatisch darum, Tests zu schreiben und dabei zu überlegen, wie eine gute Implementierung aussehen kann. Durch die schnell ausführbaren Tests kann einfach nachvollzogen werden, ob die Implementierungsarbeit abgeschlossen ist, ohne großartig dann noch testen oder überlegen zu müssen.

Persönliches Exemplar für Thomas Bartholomäus

Fast alle diese Tests sollten automatisch in der CI-Pipeline ausgeführt werden. Der Punkt ist hier vor allem, dass nicht nur nach bestimmten Meilensteinen getestet wird, sondern regelmäßig und so früh wie möglich.

Als Entwickler kümmert man sich also um die Implementierung des Testfalls sowie um die eigentliche Implementierung. Falls die Tests bisher lokal ausgeführt wurden, war es immer ein Stolperstein, dass es viel Wartezeit gab oder man zunächst Abhängigkeiten hinterherlaufen musste, die lokal nicht vorhanden waren.

Deswegen sollten Tests stets in der CI-Pipeline ausgeführt werden, bevor dieser Branch in den Hauptentwicklungsbranch gemergt werden. Im Rahmen des Code-Reviews in einem Merge bzw. Pull Request sollten sowohl der Build als auch die Tests ausgeführt werden, um einen umfassenden Überblick über den Stand des Features zu erlangen. So bekommen Entwickler direktes Feedback, ob alles so funktioniert oder ob nicht doch Fehler in anderen Tests auffallen. Erst wenn der gesamte Run durch die Pipeline vollständig durchlaufen wurde, erfolgt der Merge in den Hauptentwicklungsbranch. Fehler können so korrigiert werden, bevor sie andere Teammitglieder stören (siehe Abbildung 7.8).

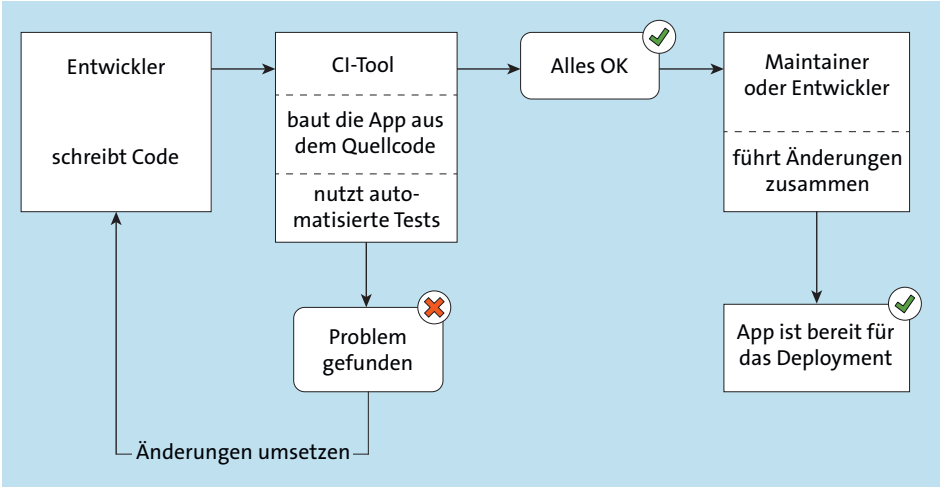

Abbildung 7.8 CI-Pipeline, Test, Review – der vereinfachte Workflow

Im Rahmen des Code-Reviews sollte der Reviewer daher auch prüfen, ob die die Problemlösung und die Testfälle korrekt und nachvollziehbar implementiert wurden. Gemergt werden sollte allerdings wirklich nur dann, wenn alle Tests durchlaufen. Bei größeren Projekten, in denen sehr viele Entwicklungen parallel ablaufen, muss dann auch geschaut werden, dass man gegen den Zielbranch einen Dry-Merge durchführt, um die Tests laufen zu lassen.

Unit-, Integrations- und Systemtests lassen sich offensichtlich automatisch ausführen. Bei *Akzeptanztests* könnte hingegen der Eindruck entstehen, dass man diese gar

© Rheinwerk Verlag, Bonn 2024

nicht in die CI-Pipeline stecken kann, weil sie ja manuell ausgeführt werden. An dieser Stelle ist der fließende Übergang der CI-Pipeline in die CI/CD-Pipeline entscheidend, also dass die Continuous-Integration-Pipeline um Continuous-Delivery-Funktionen ergänzt wird.

7.3 Fazit

Das frühe, automatisierte Testen ist Kernbestandteil von DevOps und der dazugehörigen CI-Pipeline. Die Teamstruktur verändert sich durch die Inkludierung von QA-Teammitgliedern in das Entwicklungsteam. Indem man die Quellcodes des Projekts und die Tests zusammenführt, erschafft man eine *Single Source of Truth* für das gesamte Projekt.

Dadurch, dass das Entwickler-Team geöffnet wird und neue Teammitglieder mit QA-Fokus ein Bindeglied zum teamübergreifenden QA-Team bilden, wird Silo-Denken im Entwicklungsprozess durchbrochen. Informationen zu ähnlichen Problemen und Herausforderungen gelangen viel schneller aus den Schwesterteams zu den betroffenen Entwicklern.

Frühes und automatisiertes Testen ist der wesentliche Grundstein, damit man dann im nächsten Schritt die Continuous-Delivery-Aspekte angehen und implementieren kann. Mit ihnen befassen wir uns im nächsten Kapitel.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 8
Continuous Delivery und Deployment

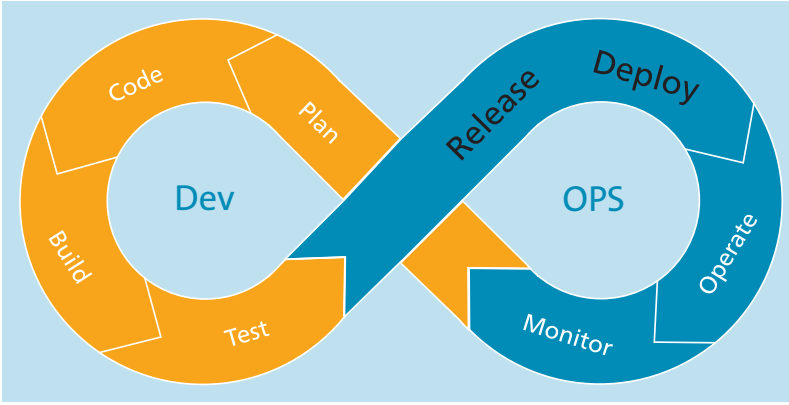

Abbildung 8.1 Release & Deploy: Die Software wird für das Deployment vorbereitet und in der Produktionsumgebung veröffentlicht.

In den bisherigen Kapiteln dieses Buches haben wir nur den Development-Prozess betrachtet, in dem es um die Planung, das Programmieren, das Bauen und Testen der Anwendung ging. Außen vor waren bisher das Betriebsteam und dessen Tätigkeitsfelder. Es geht also nun um das *Ops* aus Dev*Ops* und darum, welche Probleme zwischen getrennten Dev- und Ops-Teams bestehen und welche Vorteile sich aus der Zusammenlegung der beiden Teams ergeben.

8.1 Die typischen Probleme beim Release-Management

Alle Module des Online-Shops von *schick-gekleidet.de* – und somit alle Teams – sind grundsätzlich voneinander abhängig. Bei einem Release mussten alle Arbeiten gleichzeitig fertig sein, damit alles in einem Rutsch veröffentlicht werden konnte. Daher wurde nur grob zweimal im Jahr eine neue Version ausgerollt. Die Abstimmung zwischen den Teams dauerte sehr lange, und irgendwo kam es immer zu Verzögerungen.

Das Hauptproblem ist, dass vor dem Release alle Module einzeln zusammengesetzt werden mussten. Die Teams arbeiteten in getrennten Repositorys hinter ihren großen Wänden. Abstimmungen zwischen den Teams waren eigentlich nötig, wurden

allerdings nur halbherzig über das Ticketsystem gelöst, und selbst das kam nur dann vor, wenn jemand daran dachte, dass eine Änderung am eigenen Projekt Auswirkungen auf die anderen Teams hat.

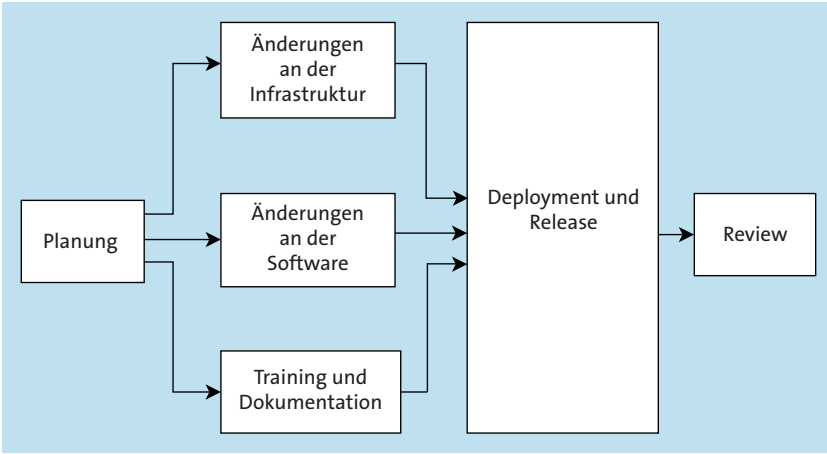

Abbildung 8.2 Beim Release-Management im Wasserfall-Prozess warten die verschiedenen Aufgaben, bis es ein zentrales Release und Deployment gibt.

Bei einem neuen Release waren die Anpassung des Quellcodes und die Aktualisierung der Software nur zwei der relevanten und nötigen Änderungen. In den meisten Fällen musste zusätzlich die Infrastruktur verändert oder gar neu aufgebaut werden, um das Deployment zu ermöglichen, bevor die neue Version vor dem Go-Live getestet werden konnte.

8.1.1 Separate Handhabung von Änderungen und Dokumentationen

Das Team, das sich um API-Änderungen im Backend kümmerte, dokumentierte bei seiner Entwicklungsarbeit die Änderungen nur halbherzig. Der Grund hierfür war, dass bei Bedarf Änderungen durchgeführt wurden, die als Anforderungen von den verschiedenen Teams hereinkamen. Die Dokumentation der API wurde jedoch erst angepasst, nachdem die Entwicklungsarbeit abgeschlossen war und der Code in die Testphase übergegangen war. Zu diesem Zeitpunkt mussten die Entwickler also noch einmal ihre Änderungen durchgehen, um sie zu erfassen und zu dokumentieren.

Durch die Zeit, die zwischen der Implementierung und der Dokumentation verging, ging wertvoller Kontext verloren, was noch dadurch schlimmer wurde, dass nicht immer diejenige Person die Dokumentation schrieb, die diese Änderungen an der API auch durchgeführt hatte.

Die beiden Teams, die die iOS- und Android-App entwickeln, hatten dadurch häufig das Nachsehen, da diese beiden Teams am stärksten von den Änderungen an der API

© Rheinwerk Verlag, Bonn 2024

abhängig sind. Es war also eher die Regel als die Ausnahme, dass sie am Ende der Entwicklungsphase nochmals einige Änderungen nach-implementieren mussten, weil die API-Änderungen bislang nicht oder nur teilweise dokumentiert worden waren.

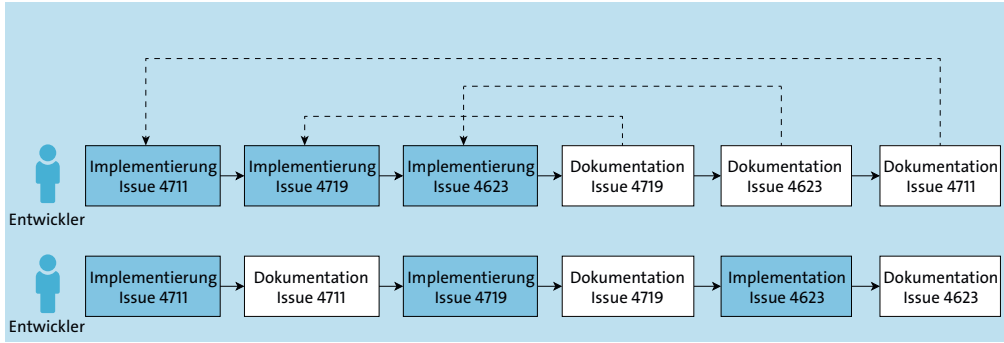

Abbildung 8.3 Der obere Fall sollte nicht auftreten: Durch die Entkopplung der Implementierung von der Dokumentation kommt es aufgrund des Kontextverlustes häufig zu Fehlern.

Dieser Vorgang verzögerte das Release immer wieder um einige Tage oder gar Wochen, weil neue Abstimmungen und Korrekturen nötig waren, da mit dem anstehenden Deployment die Mobile Apps nicht mehr vollständig funktionieren.

Wie auch bei den bisher beschriebenen Problemen waren wieder die fehlenden Abstimmungen zwischen den Teams die Hauptursache für die Probleme. Das Entwickeln im stillen Kämmerlein führte also nicht nur während der Implementierungsphase zu Problemen, es trug auch zu einem schwierigeren Release-Prozess bei.

Nachdem zumindest die Abstimmungen erledigt waren und sich der Code bei allen Teams im Freeze befand – es durfte also erst mal nichts Wesentliches angepasst werden –, ging es zunächst darum, das Release zu bauen, um es auf eine *Staging-Umgebung* zu deployen. Damit ist eine Umgebung gemeint, die grundsätzlich der Produktivumgebung ähnelt – gewissermaßen die Vorstufe für den eigentlichen Produktivbetrieb.

Wenn dann die weiteren abschließenden Tests erfolgreich waren, wurde ein Deployment auf die Produktivumgebung durchgeführt, sodass der Shop für die Kunden mit den neuesten Änderungen verfügbar war.

8.1.2 Langwieriger Release-Prozess

Dieser Release-Prozess musste nahezu komplett manuell durchgeführt werden. Wie in Abschnitt 5.1 beschrieben, hatten die meisten Teams jeweils nur einen Buildserver, auf dem die Projekte periodisch gebaut wurden – wenn denn der Bau überhaupt erfolgreich war. Zugriff gab es nur für die Teamleitung, daher war es für den vollstän-

digen Release-Prozess nötig, dass alle Teamleiter und Teamleiterinnen zur etwa gleichen Zeit verfügbar sein mussten, um ein erfolgreiches Release zu gewährleisten.

Der Release-Prozess selbst wurde von einem Release-Manager verwaltet. Dieser war für die Kommunikation mit den verschiedenen Teamleitern zuständig, die wiederum mit ihren Teams sprachen, wenn etwas nicht funktionierte.

Der Prozess folgte wiederum einem langen, vordefinierten Release-Plan und war natürlich mit den zwischenzeitlich durchgeführten Änderungen inkompatibel.

Die Teamleitungen waren dafür verantwortlich, dass die Teams den Code so bereitstellten, dass auf dem Build-Server ein lokaler Build erstellt werden konnte. Sobald dies erledigt war, sammelten die Teamleitungen die verschiedenen Dateien, die für das Deployment nötig waren, händisch zusammen.

Hierbei kam es immer wieder zu Fehlern, da Dateien fehlten oder Artefakte eingesammelt wurden, die gar nicht benötigt wurden. Das fiel immer erst dann auf, wenn die Daten schon beim Betriebsteam lagen, das ein Deployment auf die Produktionsumgebung vorbereitete. Ein programmiersprachenabhängiges Paketformat, was eigentlich üblich ist, wurde nur von wenigen Teams genutzt.

8.1.3 Automatisierungen, die sich nicht lohnen

Natürlich wäre es eine Idee gewesen, diesen ganzen Aufwand zu automatisieren, aber da nur zweimal im Jahr deployt wurde, stufte man diese Arbeit als unnötig ein: Man ging davon aus, dass der Aufwand für die Automatisierung höher ist als der Nutzen. Unter diesen Umständen ist das sogar nachvollziehbar.

Stattdessen war ein Release-Manager dafür zuständig, dass die kompilierten Artefakte der Projekte an das Betriebsteam übergingen. Diese wurden dazu einfach auf einem Fileserver abgelegt, damit das Betriebsteam mit dem Deployment auf der Staging-Umgebung beginnen konnte. Natürlich war auch bei diesem Schritt der zeitliche Rahmen immer knapp bemessen. Der Termin für das Release der neuen Version stand schon fest, und alle Verzögerungen im Entwicklungs- und Testprozess waren hier spürbar.

Viel Zeit konnte am Ende aber nicht mehr gewonnen werden, denn auch das Deployment auf die Staging-Umgebung bestand aus Handarbeit: Jemand aus dem Betriebsteam kopierte die übergebenen Artefakte einfach per Hand an die (hoffentlich) richtige Stelle.

Das Betriebsteam war dabei auf Anleitungen zur Inbetriebnahme des jeweiligen Moduls angewiesen. Diese Anleitungen sollten von den Dev-Teams in Form einer umfassenden Dokumentation erstellt und geliefert werden. In dieser Dokumentation sollten besonders alle Abhängigkeiten aufgeschlüsselt werden. Zumindest in der Theorie. In der Praxis sah es jedoch so aus, dass es ständig zu Konflikten kam.

© Rheinwerk Verlag, Bonn 2024

Die Dokumentation war löchrig, veraltet und passte nicht zur Konfiguration der Staging- und schließlich auch der Prod-Umgebung. Aber wie auch? Die Entwickler hatten ja keinen Einblick in (oder gar Einfluss auf) die genauen Versionsstände und Einstellungen des Servers. Und was das Betriebsteam in seiner Produktivumgebung so alles machte, wusste auch niemand so genau.

8.1.4 Anfeindungen zwischen den Teams

Die Folge waren teils giftige Anfeindungen, bei denen die Entwickler unter dem Generalverdacht der Faulheit standen, während den Mitgliedern des Betriebsteams vorgeworfen wurde, sie seien einfach nur zu inkompetent, um ein paar Module an den richtigen Ort zu legen.

Das Deployment auf die Staging-Umgebung dauerte also immer seine Zeit, und Spaß machte es niemandem. Um zumindest die Konflikte ein wenig einzudämmen, wurde daher entschieden, dass das Betriebsteam bei Deployment-Fehlern immer den Kontakt mit dem Release-Manager suchen sollte.

Der Release-Manager wiederum sprach mit den jeweiligen Teamleitern, die im Anschluss mit ihren Teams die Probleme beim Deployment diskutierten. Das Ergebnis dieser internen Teamsitzungen war meistens wenig überraschend: Die Fehler lägen woanders, und jemand anderes müsse sich um die Probleme kümmern. Eine gemeinsame Koordination und der Einblick in die Fehler der jeweils anderen Teams fehlten, sodass bei jedem Deployment das Chaos wieder von vorn losging.

Nachdem dann das Betriebsteam mit viel manueller Arbeit, digitalem Klebeband und weiteren undokumentierten Änderungen die Staging-Umgebung des Online-Shops zum Laufen gebracht hatte, ging es für das QA-Team in den Endspurt. Seine Mitglieder mussten noch einige End-to-End-Tests durchführen, um sicherzustellen, dass auch wirklich keine Fehler enthalten waren. Wie üblich fielen dabei noch ein paar Fehler auf, sodass die einzelnen Teams abermals nacharbeiten mussten. Die Änderungen wurden jeweils also wieder auf die Staging-Umgebung kopiert – mit allen bereits erwähnten Schmerzen.

Nachdem die Freigabe vom QA-Team vorlag, musste auch noch der Release-Manager die Freigabe für das Deployment geben. Abschließend wurde noch von der Geschäftsführerin von *schick-gekleidet.de* eine Freigabe angefordert.

8.1.5 Deployment auf Produktivsystemen mit Hindernissen

Das Betriebsteam erhielt schließlich über das Ticketsystem die Aufgabe, das Deployment auf die Produktivumgebung auszurollen. Dabei war es wichtig, dass das Update zu einer bestimmten Uhrzeit an einem bestimmten Tag durchgeführt werden sollte. Beispielsweise wurde eine große Änderung geplant, die gleichzeitig mit den Black-

Persönliches Exemplar für Thomas Bartholomäus

Friday-Angeboten ausgerollt werden sollte, zu denen zeitgleich sowohl online als auch im Fernsehen Werbung geschaltet wurde.

Ein erfolgreiches Deployment zur richtigen Zeit ist also aus mehreren Gründen wichtig: Zum einen geht bei einem zu späten Deployment Umsatz verloren, weil Kunden auf den Online-Shop gehen, dort aber womöglich nicht die Angebote sehen, die sie erwartet haben, obwohl diese beworben wurden.

Zum anderen darf das Deployment auch nicht zu früh ausgerollt werden, weil sonst schon Angebote vergriffen sein könnten, die erst später beworben werden, was wiederum zu Unmut bei den Kunden führt, die möglicherweise dann nicht mehr wiederkommen.

Zu dem Zeitpunkt, an dem das Deployment auf die Produktivumgebung ansteht, wurde daher intern ein Wartungsfenster angekündigt. Das Betriebsteam fing also wieder an, die Dateien der Module auf die Produktivumgebung zu kopieren. Und das natürlich auch wieder in Handarbeit. Und obwohl man sich natürlich bemühte, die fehlenden Abhängigkeiten und andere Stolpersteine zu beseitigen, über die man sich schon beim Deployment auf die Staging-Umgebung geärgert hatte, traten wieder Fehler auf, die eigentlich längst behoben sein sollten.

Aber nicht nur das! Es gab auch spannende neue Probleme, die das Betriebsteam vor der engen Frist in den Wahnsinn trieben. Das lag zum einen daran, dass sich die Staging-Umgebung zu stark von der Produktivumgebung unterschied. Aus Kostengründen wurde an der Staging-Umgebung nämlich etwas gespart, sie war also nicht exakt so aufgebaut wie die Produktivumgebung.

Zum anderen wurde die Staging-Umgebung gern für Experimente und Tests benutzt, die man nicht in der Produktivumgebung nachstellen konnte. Jeden Handgriff, den man im Staging-Bereich machte, hätte man ja in Prod nachholen müssen – völlig undenkbar bei der Menge an händischen Änderungen.

Und schließlich machten sich nicht nur Probleme im Entwicklungsprozess bemerkbar, auch mit den Tests haperte es. Es zeigte sich nämlich, dass bei den End-to-End-Tests immer nur auf den Einkaufsprozess geachtet wurde, denn mit ihm wird schließlich das Geld verdient.

An den Arbeitsablauf für das Einbinden von neuen Produkten hatte niemand gedacht, sodass die neue Ware für die Black-Friday-Angebote gar nicht erst eingepflegt werden konnte. Und schließlich wurde im Staging-Bereich nur mit einer deutlich reduzierten Datenbasis gearbeitet, sodass Performance-Probleme nicht auffielen.

Das Entwicklungsteam musste außerhalb seiner normalen Arbeitszeiten erneut nacharbeiten und warf die Änderungen wieder »über den Zaun«. Bei solchen kurzfristigen, ungeprüften Changes sind Fehler natürlich keine Überraschung, und so kam es, dass nun fälschlicherweise zu viele Informationen in eine Logdatei geschrie-

© Rheinwerk Verlag, Bonn 2024

ben wurden, da Debug-Ausgaben nicht entfernt wurden, wodurch die Festplatten regelmäßig vollliefen.

Anstatt nun das Problem an der Quelle zu lösen, schrieb ein findiger Admin ein Skript, das die Logdatei regelmäßig löscht. Weil damit das Problem gelöst zu sein schien, benötigte man ja kein Ticket beim Entwicklungsteam. Kommunikation wurde als anstrengend angesehen, und die Entwickler würde es bestimmt nicht stören, dass die Logs alle paar Stunden gelöscht werden ...

Zum Zeitpunkt des geplanten Go-Lives wurden dann die Dienste neu gestartet, damit das Deployment abgeschlossen werden konnte und die neue Software lief. Zumindest idealerweise. Oftmals stand man vor dem Problem, dass die Dienste nicht wieder sauber anliefen und Fehler auftraten, die man durch bessere Tests im Staging-Bereich eigentlich hätte abfangen können.

Die einzige Lösung war die Operation am offenen Herzen: Eingriffe in das laufende Prod-System, damit der Shop möglichst schnell wieder online war und funktionierte. Das war der Horror für das Betriebsteam und zudem immense Arbeit für die Entwickler, die diese Änderungen auch wieder in ihren Dev-Stages nachstellen mussten, wenn man das Abenteuer nicht beim nächsten Mal wiederholen wollte.

8.1.6 Fazit

Einige der aufgezeigten Probleme tauchen meiner Erfahrung nach häufig auf – vielleicht nicht alle zusammen, aber jedes Deployment hält Überraschungen parat. Die Ursache für die Probleme ist, dass wenig automatisiert wurde, weil man davon ausging, dass sich das ja nicht lohnt, wenn man nur wenige Male pro Jahr ein Deployment macht. Und da es bisher immer »so irgendwie« funktioniert hat, sah man in der Führungsetage keinen Bedarf an zusätzlichen Automatisierungslösungen und anderen Prozessen. Für alle Projektbeteiligten war ein Release immer mit hohem Stress verbunden, da viel koordiniert werden musste und immer verschiedene Fehler auftraten. »Aber so war es nun einmal.«

Reflexion

Beim Release-Management, gepaart mit dem Deployment, wird erstmals ein Blick auf die Zusammenarbeit zwischen Dev und Ops geworfen. Bevor Sie sich der Automatisierung dieses Prozesses widmen, ist es wichtig, dass das Team den Schalter im Kopf umlegt und die bestehenden Probleme erkennt.

Die hier exemplarisch dargestellten Probleme sind in einem gewissen Rahmen üblich. Als ersten Schritt bietet es sich hier ebenfalls an, zu untersuchen, ob es Missgunst zwischen Dev und Ops gibt. Diese gilt es dann so weit es geht auszuräumen.

Investieren Sie frühzeitig in Teambuilding-Maßnahmen für Ihr Team, um die Zusammenarbeit zu stärken.

Persönliches Exemplar für Thomas Bartholomäus

8.2 Continuous Delivery und Deployment implementieren

Im Rahmen der DevOps-Transformation wird es also Zeit, nun wirklich Dev und Ops zu einem gemeinsamen Team mit einem einzigen Ziel zusammenzubringen. Im vorherigen Kapitel wurde ja bereits das Qualitätssicherungsteam mit in die bisher reinen Entwicklungsteams eingebunden. Jetzt geht es darum, auch das Betriebsteam in die verschiedenen Teams einzubeziehen.

8.2.1 Devs und Ops zusammenbringen

Ein häufiger Mythos rund um DevOps ist, dass das Entwicklungsteam jetzt auch einfach den Betrieb der Anwendung mit übernimmt. Das klingt dann immer so, als wenn die Arbeit des bisherigen Betriebsteams super easy und so einfach wäre, dass man diese wegrationalisieren könnte. Schließlich gibt es ja auch die Cloud mit vielen Automatisierungen, in der Infrastruktur on demand verfügbar ist.

So ist es natürlich nicht! Was stattdessen gefordert wird, sind eine engere Zusammenarbeit, zielorientierte Kommunikation und eine stärkere gemeinsame Verantwortung.

Um besser nachvollziehen zu können, was das heißt, lohnt es sich, sich noch mal die Aufgaben der verschiedenen Rollen gründlich anzusehen, um dann zu schauen, wie sich diese in einem cross-funktionalen Team umsetzen lassen.

Treten wir also einen Schritt zurück: Welche Probleme wollen wir hier eigentlich lösen? Das Ziel ist, dass ein Service für die Nutzer bereitgestellt wird. Dazu muss nicht nur die Software entwickelt und getestet, sondern eben auch betrieben werden. Das waren in getrennten Teams auch getrennte Ziele, die nicht ohne Weiteres unter einen Hut gebracht werden konnten. In Abschnitt 8.1 haben Sie gesehen, wie diese Probleme in der Praxis bei *schick-gekleidet.de* aussehen.

Das starre Aufteilen der Aufgaben und Mitarbeiter führt zu einer Arbeitskultur, die sich bestenfalls durch ein Nebeneinander, viel zu oft aber durch ein Gegeneinander auszeichnet. Diese »Unkultur« muss aufgebrochen werden, um die Symbiose zu erschaffen. Dafür steht der DevOps-Gedanke.

So, wie sie in Abbildung 8.4 dargestellt wird, ist die Teamstruktur natürlich sehr stark vereinfacht. Sie können auch nicht erwarten, dass man Entwickler und Admins in einen Raum stecken kann und via Zauberhand ein DevOps-Team entsteht. Stattdessen müssen die Personen abgeholt und ordentlich geschult werden. Es muss zudem geschaut werden, wie die Kenntnisse und Erfahrungen aussehen, wo die individuellen Stärken und Schwächen liegen.

Klar ist, dass sich die Rollen und Aufgaben aller beteiligten Personen verändern werden: Das bisherige Ziel des reinen Entwicklungsteams war die Weiterentwicklung der Software, während das Betriebsteam den stabilen Betrieb der Anwendung zu verant-

© Rheinwerk Verlag, Bonn 2024

worten hat. Dass diese Ziele in gewissem Maße konträr zueinander sind, ergibt sich aus den Anforderungen.

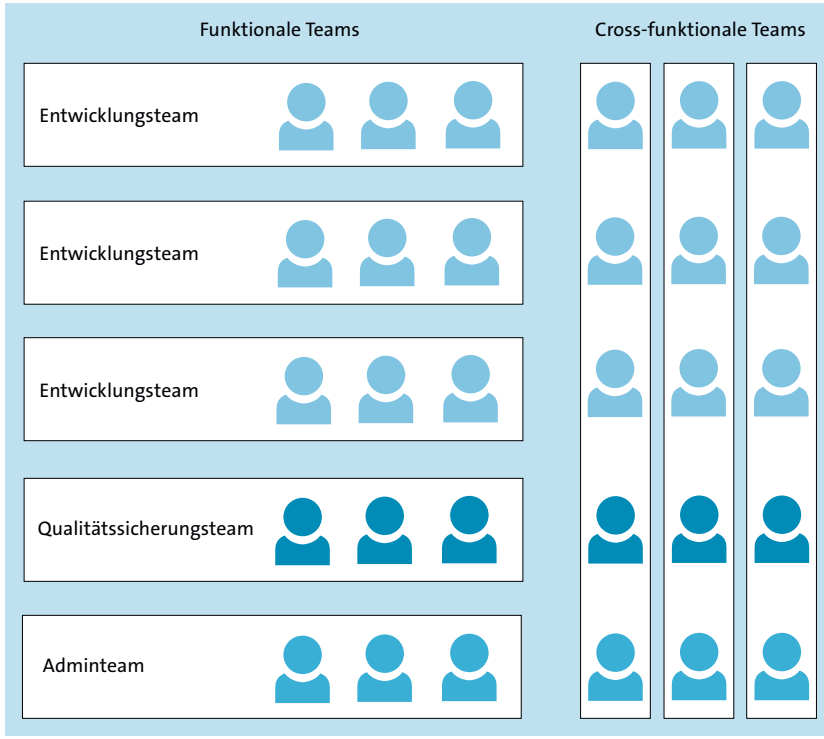

Abbildung 8.4 Zu dem cross-funktionalen Team gehören nicht nur Dev und QA, sondern auch Ops.

Das ist einer der wichtigsten Gründe, warum ein bloßes Zusammenwerfen der Rollen oder gar das Wegstreichen des bisherigen Betriebsteams nicht hilfreich ist, denn die unterschiedlichen Ziele bleiben ja bestehen.

Stattdessen müssen beide Rollen mit beiden Zielen näher zusammenrücken, damit cross-funktionale Teams entstehen, die an einem Strang ziehen. Wie das im Detail umgesetzt werden kann, schauen wir uns in Kapitel 12 an, in dem es um die Transformation zu einer modernen DevOps-Kultur geht.

Die bisherige Administratorin aus dem Betriebsteam wird also viel näher an den eigentlichen Code rücken und sollte keine Berührungsängste haben, bei der Paketierung und der Vorbereitung des Deployments mit anzupacken. Im Gegenzug gilt für die Entwickler, dass sie sich viel intensiver um das Deployment und den Betrieb kümmern und dort Verantwortung übernehmen müssen. Das Entwicklungsteam bringt dabei die Erfahrung rund um den Code, die Entwicklungshistorie und die Architektur des eigenen Projekts mit; seine Mitglieder wissen also, wie die Anwendung funktioniert und was hinter den einzelnen Features steckt.

Persönliches Exemplar für Thomas Bartholomäus

Das ist Wissen und Erfahrung, die das Betriebsteam nicht haben kann. Dafür wissen die Admins, wo die Probleme im Betrieb der Anwendung stecken, wo es beim Deployment hakt und wann sich die Anwendung auch mal anders als erwartet verhielt.

Bisher war die Schnittstelle zwischen den beiden Teams auf der technischen Ebene das Paketformat: Das Entwicklungsteam wusste, in welchem Format es die Anwendung paketieren musste. Das Betriebsteam erwartete das definierte Paketformat, damit es das Artefakt entgegennehmen konnte, um es in Betrieb zu nehmen.

An dieser Stelle liegt der erste Ansatzpunkt: Nur weil das Paketformat bekannt war, hieß es noch lange nicht, dass neue Versionen auch ohne Probleme in Betrieb genommen werden konnten. Das führte dazu, dass in den strikt getrennten Teams die Bälle immer hin- und hergespielt wurden. Das sollte im cross-funktionalen Team nach DevOps-Prinzipien nicht mehr passieren: Arbeiten Sie nicht gegeneinander, sondern gehen Sie zielführend das Problem an, das es konkret zu lösen gilt! Alle bringen ihre Erfahrungen und Kenntnisse ein, indem sie direkt und auf kurzen Wegen miteinander sprechen.

Ein Ort dafür kann ein *Daily Standup* sein, das schon längst gängige Praxis in agilen Teams ist. In einem solchen kurzen Meeting werden aktuelle Herausforderungen und Probleme angesprochen. Wenn eine Entwicklerin bei einem anstehenden Deployment Probleme vermutet und sich dazu mit den Admins austauschen will, ermöglicht ein solches unkomplizierte Meeting den Austausch von Erfahrungen und Blickwinkeln.

Hinzu kommen *Trainings* und *Workshops*, in denen man voneinander lernt. Idealerweise werden diese direkt im Team organisiert und durchgeführt. Stellen Sie sich also keinen langweiligen Frontalunterricht vor, in dem ein Entwickler den Admins das Programmieren beibringen will – das ginge vollkommen in die falsche Richtung. Viel besser sind kleine, unbürokratische Anlässe für den Wissensaustausch, am besten im Dialog über ein Thema.

Nehmen wir an, dass die alte monolithische Anwendung in eine Mikroservice-Architektur überführt und containerisiert werden soll, damit sie auf einer Container-Orchestrierungsplattform betrieben werden kann.

Das Paketformat ist entsprechend ein Container-Image. Bis wir so weit sind, liegt aber viel Arbeit vor dem Entwicklungsteam, das die alte Anwendung umbauen muss. Dabei macht es sich natürlich das Leben leicht und nutzt Base-Images, die möglichst viele Werkzeuge und Abhängigkeiten mitbringen. Das ist praktisch, führt aber dazu, dass das Image unnötigerweise mehrere Gigabyte groß wird.

Was für die Entwicklerinnen und Entwickler im ersten Moment gar kein Problem ist, lässt die Admins sofort aufhorchen: Mehrere Gigabyte für jedes neue Deployment? Das wird zu Problemen beim Rollout und zu mehr Kosten führen.

© Rheinwerk Verlag, Bonn 2024

Da beide Teams jetzt am gleichen Tisch sitzen, fällt dieses potenzielle Problem schnell auf und kann gelöst werden, bevor das verwendete Base-Image zu fest im Container verankert wird. Alle bringen ihre Expertise ein, sei es zum Betrieb von Containern oder zur Paketierung der Software. Wichtig ist, dass eng zusammengearbeitet wird, um das neue gemeinsame Ziel umzusetzen: die zügige und strukturierte Weiterentwicklung *und* den sicheren und stabilen Betrieb der Anwendung. Beide Rollen unterstützen sich im Laufe der Zeit fortwährend dabei, dass beide Ziele besser umgesetzt werden.

Neue Aufgaben im Team

Genauso wie es weiterhin Software-Entwickler geben wird, wird es auch weiterhin eine Art Betriebsteam geben, allerdings mit geänderten Aufgabengebieten. Der Fokus verschiebt sich in Richtung des Anwendungsbetriebs, in enger Zusammenarbeit mit den Entwicklern. Alle arbeiten gemeinsam an und mit der CI/CD-Pipeline und sorgen dafür, dass Änderungen möglichst schnell bei den Kunden landen.

Betrieb, Infrastruktur, Plattform

In sehr großen Unternehmen wird es ein Team geben, das sich mehr um die reine Infrastruktur kümmert. Die Verwaltung der Infrastruktur – ob sie als »echtes« Blech existiert oder in der Cloud, spielt eigentlich keine Rolle – wird die Hauptaufgabe sein, während das Deployment und der Betrieb der einzelnen Anwendungen nun von den Kollegen übernommen werden.

Die Bezeichnungen *Plattformteam* oder *Infrastrukturteam* können daher passender sein, da es dabei nicht um den Betrieb der Anwendung geht. Die Aufgabe des Infrastrukturteams ist es stattdessen, eine stabile Plattform bereitzustellen, damit die anderen Teams auf ihr arbeiten können.

Im cross-funktionalen Anwendungsteam, aber auch im Plattform- und Infrastrukturteam, wird die DevOps-Idee spürbar sein: Da alles schneller und agiler vonstattengeht, ist es notwendig, Handarbeit zu automatisieren und Prozesse möglichst zu standardisieren. Auch wer weiterhin »nur« für die Verfügbarkeit der Infrastruktur zuständig ist, muss bei kurzfristigen Deployments rasch reagieren können. Das geht nur, indem auch diese Aufgaben möglichst automatisiert werden – es bringt nichts, wenn eine neue Version zwar in ein paar Minuten fertig für das Deployment ist, es aber zwei Wochen dauert, bis eine neue Festplatte eingebaut ist, um den gewachsenen Platzbedarf zu bedienen.

Der Schlüssel hierzu ist *Infrastructure as Code* (IaC), also die Idee, dass selbst die Infrastruktur durch Befehle und Konfigurationen schnell und einfach angepasst werden kann – darum wird es in Abschnitt 9.5.1 ein wenig ausführlicher gehen. Dazu nutzen Sie entweder entsprechende Tools, die recht stark an Programmiersprachen erin-

nern, oder es kann je nach Umfang sogar bedeuten, dass eigene Software entwickelt und gewartet werden muss. Dabei können Sie auf die Skills der neuen Kollegen zurückgreifen, um manuelle Arbeit in möglichst gute Automatisierungsskripte zu überführen.

So profitieren alle vom Austausch untereinander: Admin-Veteranen lernen, wie sie mit ein paar Zeilen Code Aufgaben lösen, während die Kollegen aus der Software-Entwicklung praktische Erfahrungen im Betrieb sammeln. Das sorgt dafür, dass alle viel größeres Vertrauen in die CI-Pipeline entwickeln, wenn sie wissen, dass am Ende ein sicheres Deployment in der Produktionsumgebung steht.

Dazu müssen alle im Team den grundlegenden Aufbau sowohl der CI-Pipeline als auch der eigentlichen Anwendung verstanden haben, was aber natürlich nicht bedeutet, dass alle jedes Detail kennen. Es wird immer Spezialisten für unterschiedliche Aufgaben geben, die nun jedoch im engen Austausch miteinander stehen.

Schmerzfreies Deployment

Indem auf die Erfahrung aller Teammitglieder zurückgegriffen wird, lassen sich Probleme beim Release-Prozess direkt an der Wurzel packen. Wenn es beim Build oder beim Ausrollen hakt, gibt es schnelles Feedback. Diese Rückmeldung muss ohne Umwege dort landen, wo sie hingehört, nämlich bei dem Teammitglied, das das Problem lösen kann. Anstatt also wie früher über Tickets hin- und herzukommunizieren und Aufgaben von sich wegzuschieben, sieht das ganze Team direkt, dass etwas zu tun ist.

Im Grunde gibt es drei verschiedene Aufgabengebiete, die alle vom Team abgedeckt werden müssen. Zwei davon habe ich schon erwähnt: die Software-Entwicklung sowie der Betrieb der Anwendung. Beide Aufgabengebiete haben einen fließenden Übergang, verbunden durch die CI/CD-Pipeline.

Ich würde hier nicht unbedingt von verschiedenen Aufgaben und verschiedenen Rollen sprechen. Klar eingrenzen kann man das nicht, und es ist stark davon abhängig, wie die Software und die Firma strukturiert sind. Wichtig ist, dass die ursprünglich getrennten Aufgaben auch, aber nicht nur, mithilfe von Automatisierung verknüpft werden. Letztendlich müssen sich also alle im Team die Hand geben, um effizient miteinander zu arbeiten.

Direkte Mitarbeit fördern

Um den Prozess der Mitarbeit zu verbessern, sollten alle über Pull bzw. Merge Requests gemeinsam an Problemen arbeiten und Änderungen beitragen, anstatt Pingpong mit Tickets zu spielen. Dabei gelten die gleichen Regeln und Prinzipien wie beim Code-Review (siehe Abschnitt 5.4) – mit dem kleinen Unterschied, dass es nicht mehr nur um Software-Quellcode geht, sondern dass es auch Konfigurationsdateien und Dokumentation sein können, an denen gearbeitet wird.

© Rheinwerk Verlag, Bonn 2024

Änderungen können von den beteiligten Personen dann einfach selbst getestet werden, bevor sie für alle in die eigentliche Entwicklung einfließen. Man spart auf diese Weise nicht nur viel Zeit, sondern auch Nerven und Probleme an nachgelagerter Stelle.

Zentral ist, dass der Release-Prozess möglichst komplett automatisiert ist. Hier spielt *The First Way* der Three Ways eine wichtige Rolle, da so bewerkstelligt wird, dass die Änderungen möglichst früh von Dev zu Ops fließen und somit vom Business zum Kunden. Das bisher halbjährliche Release war auch der Grund, warum nichts automatisiert wurde: Man würde viel Zeit und Aufwand in die Automatisierung stecken, die aber nur zweimal im Jahr genutzt wird.

Wenn Deployments häufiger durchführt werden sollen, dann ist eine vollständige Automatisierung unabdingbar. Nebenbei werden auf diese Weise viele Fehler vermieden, die zuvor aufgetreten sind, da möglichst vollständig automatisiert wird und nicht nur teilweise. Das wird sicherlich nicht alles beim ersten Versuch und auf Anhieb ohne Probleme funktionieren. Stattdessen geht es um ein stetiges Herantasten, bei dem mit jedem Schritt die Prozesse verbessert werden.

Hier spielen mehrere Faktoren hinein: Aus rein technischer Sicht ist die Automatisierung des Deployments nicht wahnsinnig kompliziert, sobald sie einmal steht. Man muss sich allerdings das große ganze Bild anschauen, wo sich immer die Frage stellt: Wie sehr vertraut man wirklich auf die eigene Software und die Fähigkeit des eigenen Teams, idealerweise mehrfach am Tag Änderungen auszurollen?

Schritt für Schritt die Deployment-Frequenz erhöhen

Um als Endziel täglich mehrfach Deployments auszurollen, müssen Schritt für Schritt die Abstände zwischen den Deployments verringert werden. Durch jedes Deployment sammelt das Team Erfahrungen und erarbeitet sich Vertrauen in die eigene Software sowie in den Release-Prozess, um ohne Angst und mit möglichst wenig Fehlern regelmäßig Deployments ausrollen zu können.

Zur Erinnerung: Wie Sie in Abbildung 8.5 sehen können, wird das Risiko für Probleme immer größer, je mehr Zeit zwischen zwei Deployments liegt. Wie gehabt gilt daher, die Größe der Veränderungen möglichst gering zu halten.

Dafür müssen langfristig die Abstände zwischen den Deployments auf Produktivumgebungen nach und nach verkürzt werden. Da man allerdings nicht direkt mit der Produktivumgebung starten kann, sollte man sich zuerst auf die übrigen Umgebungen konzentrieren, bevor man sich dann um die Produktivumgebung kümmert.

Im Rahmen der CI/CD-Pipeline wird wie gehabt das Projekt gebaut und werden die verschiedenen Tests ausgeführt. Hinzu kommt nun, dass im Rahmen des Builds das Projekt nicht nur »normal« gebaut, sondern auch paketiert wird.

Persönliches Exemplar für Thomas Bartholomäus

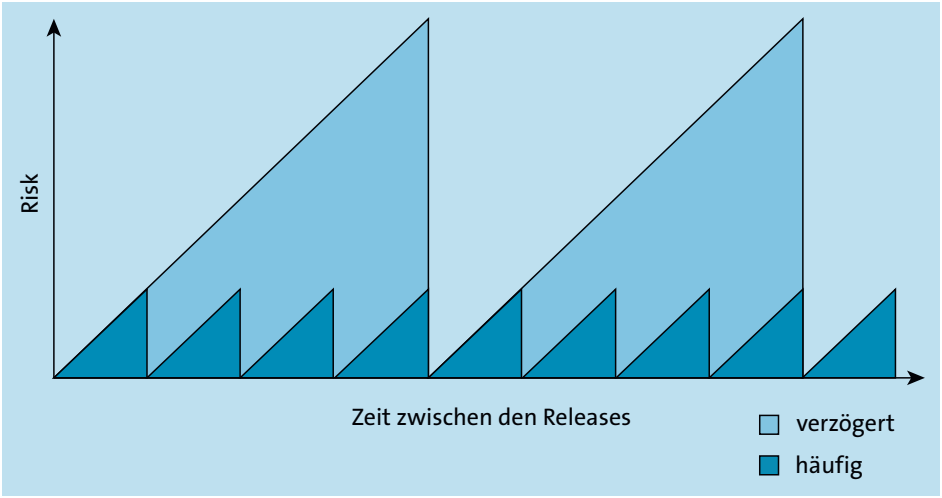

Abbildung 8.5 Je länger die Zeit zwischen den Releases ist, desto höher ist das Risiko für Probleme, da sich größere Änderungen aufgestaut haben.

Dieses Paket wird benötigt, um im Schritt darauf das Deployment anzustoßen. Dazu wird dieses Paket in eine Package-Registry hochgeladen, sodass es eine Versionshistorie der gebauten Pakete gibt. Im Falle eines Rollbacks kann damit schnell eine ältere Version ausgerollt werden.

Die CI/CD-Pipeline sollte für alle Teammitglieder zugänglich sein, sodass alle nachvollziehen können, ob das Projekt baut und auch alle Tests erfolgreich durchlaufen. Ohne Überblick, was die Prozesse und Tools leisten, wird es schwierig, ein gemeinsames Ownership der Software aufzubauen.

Bevor also ein Deployment auf der Produktionsumgebung ausgerollt wird, sollte die Automatisierung auf Staging- und Review-Umgebungen bevorzugt behandelt werden, um das Vertrauen in Tools, Prozesse und letztlich natürlich auch die Menschen zu erhöhen.

8.2.2 QA-, Staging- und Prod-Umgebungen

Durch die neuen Teamstrukturen und das neue Mindset aus Abschnitt 8.2.1 haben wir den Deployments den Schrecken genommen. Entwicklungsarbeit liegt nun nicht mehr für Monate auf Halde, sondern wird schnell und schmerzlos ausgeliefert.

Wenn man CI/CD vollständig umsetzt, dann werden nach jedem Merge die Änderungen auf die Staging-Umgebung direkt ausgerollt. So erhält man zügig Feedback, ob das Deployment der Neuerungen überhaupt funktioniert, und kann es fachlich prüfen.

Ob und wann nun auf die Produktionsumgebung deployt wird, hängt von ein paar Faktoren ab. Wenn man CI/CD stringent umsetzt, dann kann eigentlich sofort auf die

© Rheinwerk Verlag, Bonn 2024

Prod-Umgebung deployt werden, nachdem das Deployment auf der Staging-Umgebung erfolgreich war. Ganz Abenteuerlustige lassen sogar eine Staging-Umgebung weg. In der Praxis wird dies jedoch selten so umgesetzt. Üblicher ist, dass das Deployment auf die Prod-Umgebung manuell angestoßen wird, um mehr Kontrolle zu haben, wann und ob das Deployment durchgeführt werden soll.

Wie dies genau abläuft, hängt von verschiedenen Faktoren ab: vom Vertrauen in die Software, von der Uhrzeit am entsprechenden Wochentag und von den konkreten Business-Kriterien. Häufig wird allerdings nicht völlig blind deployt, sondern mit Feature-Flags und Canary-Deployments gearbeitet (siehe Abschnitt 8.4).

In der Praxis wird meistens so gearbeitet, wie Abbildung 8.8 zeigt. In der Entwicklung gibt es eigene Review-Umgebungen, in denen schnell Features ausprobiert werden können. Wenn sich diese bewähren, fließen sie in den Main-Branch und werden auf eine Staging-Umgebung deployt. Dort erfolgt die Qualitätssicherung: Funktioniert alles so, wie es soll? Erst wenn dieser Schritt abgeschlossen wurde, erfolgt das Deployment in die Produktion.

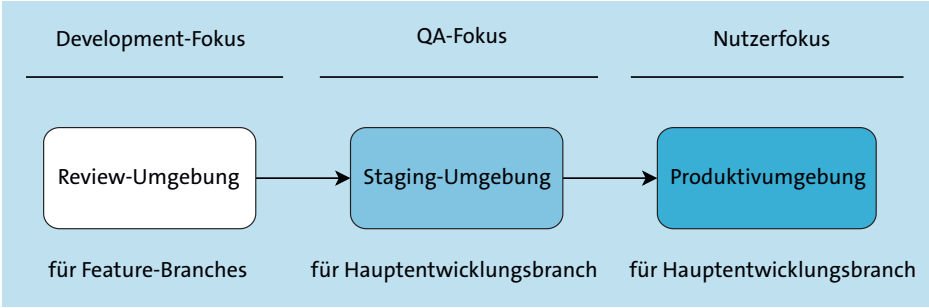

Abbildung 8.6 Die verschiedenen Umgebungen haben jeweils einen anderen Fokus.

Review-Umgebungen für Entwickler

Das Ziel einer CI/CD-Pipeline und des Peer-Reviews der Änderungen ist, dass die Arbeiten an einem Feature oder einer Fehlerkorrektur möglichst vor dem Merge in den Hauptentwicklungsbranch komplett abgeschlossen sein sollten. Um sagen zu können, wann ein Feature wirklich funktioniert, wie es soll, braucht es Umgebungen, um die Änderungen kurz auszuprobieren und zu begutachten. Diese Umgebungen werden daher Review-, QA- oder Dev-Umgebungen genannt.

Abbildung 8.7 zeigt, dass die Entwicklung von Features oder das Beheben eines Bugs in Review-Umgebungen getestet werden. Dort ist aber Endstation: Die Umgebungen sind reine Spielwiesen, die nichts mit der Produktionsumgebung zu tun haben und ganz anders aufgebaut sein können. Wenn die Änderungen akzeptiert werden, fließen sie in den Hauptzweig zurück, der auf einer Staging-Umgebung getestet wird.

Persönliches Exemplar für Thomas Bartholomäus

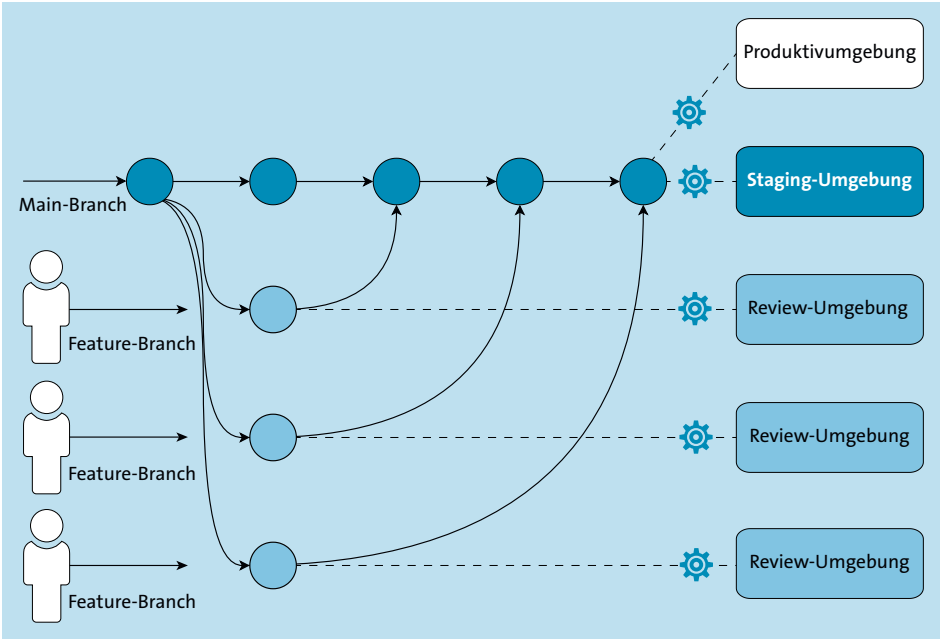

Abbildung 8.7 Für jeden Feature-Branch kann es Review-Umgebungen geben, aber in der Regel gibt es nur eine Staging- und eine Produktivumgebung.

Damit dies zügig und ohne Reibungsverluste geschieht, muss die Anwendung automatisiert deployt werden. So wird jede Änderung getestet, bevor es zu einem Merge in den Hauptzweig kommt. Im Review werden gleich mehrere Dinge abgedeckt:

- Das Projekt wird gebaut.
- Das Projekt wird getestet.
- Das Projekt wird deployt.
- Die Änderungen werden reviewt.

So findet man bereits frühzeitig viele mögliche Fehlerquellen, die sonst erst sehr viel später aufgetreten wären. Durch die vollständige Automatisierung wird sichergestellt, dass Probleme rechtzeitig auffallen, was den Druck auf das Deployment auf der Produktivumgebung deutlich verringert.

Das Deployment auf die Review-Umgebung hat zudem den Vorteil, dass eine fachliche Abnahme früh erfolgen kann. Man schiebt also viele Dinge, die bisher erst sehr viel später im gesamten Entwicklungsprozess fällig würden, an einen viel früheren Zeitpunkt, um die Probleme und Fehler dann zu korrigieren, wenn diese implementiert wurden. Das ist letztlich ein nahtloser Übergang von der Qualitätssicherung aus Kapitel 7. Und es geht in das Thema von Kapitel 9 über, in dem es um den Betrieb der Anwendung geht.

© Rheinwerk Verlag, Bonn 2024

Staging-Umgebungen als Vorstufe für die Produktion

Nachdem das Code-Review und das Deployment in die Testumgebungen erfolgreich absolviert wurden, steht der eigentliche Test an: Läuft die Anwendung auch in der Praxis so, wie sie soll?

Dazu muss die Anwendung unter den gleichen Bedingungen getestet werden, unter denen sie auch später laufen wird – dass die einzelnen Features durch die Reviews gekommen sind, bedeutet ja noch nicht, dass auch später alles funktionieren wird.

Diese Vorstufe vor dem Ernstfall nennt man *Staging*. Der Hauptzweig wird deployt und unter realen Bedingungen getestet.

Auch bei *schick-gekleidet.de* gab es neben der Produktivumgebung eine Staging-Umgebung, allerdings kam es bei jedem Deployment aus mehreren Gründen zu etlichen Fehlern:

- Der erste wesentliche Punkt für die zahlreichen Probleme beim Deployment war, dass die Produktivumgebung viel zu stark von der Staging-Umgebung abwich. Das betraf nicht nur die Infrastruktur an sich, sondern auch die Daten und die Konfiguration.

  Die Staging-Umgebung sollte aber eigentlich eine abgespeckte Version der Produktivumgebung sein: Sie sollte in ähnlichen Netzen liegen, vielleicht mit weniger Infrastruktur, die für die Hochverfügbarkeit benötigt wird; aber die eigentliche Konfiguration sollte gleich sein.

  Je unterschiedlicher die Konfigurationen sind, desto wahrscheinlicher ist es, dass zwischen der Automatisierung der Deployments für die Staging- und der Produktivumgebung Sonderlocken eingebaut werden müssen. Je unterschiedlicher sie sind, desto wahrscheinlicher schleichen sich in der Produktivumgebung Fehler ein, die im Staging nicht aufgefallen sind.
- Der nächste essenzielle Punkt ist die Automatisierung der Infrastruktur. Alle Umgebungen sollten mit möglichst wenig Aufwand mit neuen Konfigurationen bestückt werden können. Die gesamte Infrastruktur sollte sich zügig und ohne große Angst neu aufbauen lassen.

  Hier wird vor allem viel von *immutable*, also unveränderlicher Infrastruktur und *mutable* Infrastruktur gesprochen. Dieses Thema spreche ich in Abschnitt 9.2 an, in dem es um die Frage »On-Premise vs. Cloud Hosting« geht.
- Der dritte Punkt ist, dass die Daten auf Staging- und Produktivumgebungen möglichst ähnlich sein sollten. Das hat weniger mit der reinen Infrastruktur zu tun, sondern betrifft die Datensätze innerhalb der deployten Software.

  Für *schick-gekleidet.de* heißt das also, dass möglichst alle Daten aus dem produktiven Shop auch in der Staging-Umgebung enthalten sein sollten – konkret also alle Produkte, die online verkauft werden. Bei einigen Daten muss man zwangsläufig

Persönliches Exemplar für Thomas Bartholomäus

Abstriche machen, etwa bei echten Kundendaten, die nicht auf Staging-Umgebungen zu finden sein sollten, um den Datenschutz zu wahren.

Testdaten im Staging-Bereich

Neben Unterschieden beim technischen Setup und unterschiedlichen Konfigurationen besteht auch oftmals ein weiteres Problem, das sich nicht einfach beheben lässt: Die Daten auf den Systemen sind unterschiedlich. Das fängt bereits damit an, dass nicht alle Produkte aus dem Online-Shop auf der Staging-Umgebung enthalten sind.

In der Staging-Umgebung gibt es aus Gründen des Datenschutzes jedoch auch keine echten Kundendaten. *Eigentlich* sollte es kein Problem sein, wenn stattdessen mit wenigen Platzhaltern gearbeitet wird, aber gerade, wenn es um Fragen der Skalierung geht, kann nicht ausgeschlossen werden, dass es in der Produktion zu Problemen kommt, obwohl im Staging alles funktionierte.

Deployment-Umgebungen automatisieren

Sie haben bereits gelernt, dass eine Anwendung in verschiedenen Umgebungen wie Produktion, Staging und Review deployt werden muss, und zwar für jeden Feature-Branch, der in den Hauptzweig einfließen soll.

Dabei gibt es natürlich viele Wege, die nach Rom führen. Das ein händisches Setup der Umgebungen nicht sinnvoll ist, wird Sie nicht überraschen. Sie können stattdessen frische Infrastrukturen immer wieder automatisiert hochziehen oder als Alternative *Golden Images* erstellen.

Frischgebackene Infrastrukturen

Eine Möglichkeit ist, dass man bei Bedarf die komplett benötigte Infrastruktur für die Umgebung auf Abruf hochzieht. Damit das automatisiert gelingen kann, muss eine Cloud bereitstehen – ganz unabhängig davon, ob es sich um eine private oder eine öffentliche Umgebung handelt. Außerdem muss selbstverständlich eine CI/CD-Plattform verwendet werden, in der die einzelnen Schritte ausgeführt werden. Unser Ziel ist, dass jedes Mal, wenn ein Feature-Branch im SCM-System gepusht wird, die Pipelines loslaufen und die Umgebung bauen und die neue Version der Anwendung deployen.

In der Praxis erweist sich dies jedoch als ziemlich aufwendig: Damit die Infrastruktur bereitsteht, wird eine lange Kette an Tools benötigt. Zunächst muss mit *Terraform* die virtuelle Infrastruktur hochgezogen werden. Anschließend brauchen wir ein Tool wie *Ansible*, das ebenfalls über die Pipeline gesteuert wird und das alle Abhängigkeiten installiert, damit die Anwendung deployt werden kann.

Je nach Deployment-Konzept kann auch Ansible selbst verwendet werden, um das Build-Artefakt aus der Pipeline auf dem neuen System zu deployen. Auf jeden Fall

© Rheinwerk Verlag, Bonn 2024

muss es die Systemkonfiguration übernehmen, wozu auch die Einrichtung des Webservers und der Zertifikate gehört.

Dieses Verfahren ist zwar sicherlich möglich, hat aber einen großen Nachteil: Es wäre verdammt langsam. Das Aufbauen der Infrastruktur benötigt Zeit. Das Bespielen der Systeme ebenfalls. Und beides erzeugt hohe Kosten: Eine Vielzahl von virtuellen Maschinen zu erzeugen, erfordert mehr Leistung, als Container zum Laufen zu bringen.

Ein weiteres Problem ist auch, dass eine Vielzahl an Tools benötigt wird, die von allen beherrscht werden müssen, die die Pipeline nutzen.

Golden Images

Eine alternative Methode ist das Bereitstellen von *Golden Images*. Die Idee dahinter ist, dass Images vorbereitet werden, in denen das ganze Basissystem mit allen Abhängigkeiten vorkonfiguriert ist.

Dieses Golden Image muss nur noch in einer Cloud oder einem Virtualisierungshypervisor gestartet werden und steht dann direkt für das Artefakt aus dem Build des Feature-Branches zur Verfügung.

Vielleicht erinnert Sie das direkt an das Container-Prinzip!

Der Ablauf ist ziemlich ähnlich, auch wenn die darunter liegende Technologie eine andere ist. In beiden Fällen wird erst ein Betriebssystem benötigt bzw. installiert und dann anschließend die Abhängigkeiten, die für den Betrieb der Anwendung relevant sind.

Im Vergleich zum vorherigen Beispiel gibt es hier zwei Pipelines: Die eine Pipeline ist für den Bau des Golden Images zuständig, während die andere das zuletzt gebaute Golden Image verwendet, um es zu starten und die Anwendung zu deployen.

Um Golden Images zu bauen, können Sie *Packer* (*https://developer.hashicorp.com/hcp/docs/packer*) von HashiCorp nutzen. Das Tool stand zunächst unter einer Open-Source-Lizenz, bis HashiCorp die Lizenzen für seine Projekte umgestellt hat. Alternativ können Sie auch die Features des Hypervisors nutzen, um ein Abbild einer Muster-VM zu erzeugen.

Dieses Image muss dann, ebenfalls äquivalent zu einem Container-Image, in einer Registry abgespeichert werden, damit es als Basis verwendet werden kann, wenn dann die Anwendung deployt werden soll.

Der wesentliche Vorteil bei diesem Ansatz ist, dass die Pipeline deutlich schneller abläuft. Die Golden Images müssen nicht jedes Mal auf Abruf gebaut werden, sondern es reicht, wenn das Golden Image einmal täglich oder bei Bedarf gebaut wird. Die wichtigsten Elemente liegen ohnehin im Build-Artefakt.

Durch das regelmäßige Neubauen der Golden Images fallen Fehler schneller auf und Sicherheitslücken des Betriebssystems werden zügiger gepatcht.

Persönliches Exemplar für Thomas Bartholomäus

Gerade für Produktivumgebungen ist das auch sinnvoll, weil Sie so das »Cattle not Pets«-Paradigma (siehe Abschnitt 9.2.1) auch in einer nicht containerisierten Welt umsetzen können. Dadurch wird auch manuelles Eingreifen in die laufenden Systeme verringert, weil nach einer Weile allen auffällt, dass sie Probleme an der Quelle lösen müssen, wenn die Probleme nachhaltig gelöst werden sollen.

Ein sauberes Golden Image ist ebenso wichtig wie ein sauberes Container-Image. Und so kann auch ein Golden Image hilfreich sein, wenn man Canary-Deployments oder A/B-Testing umsetzen möchte, auch wenn man eben nicht auf Container setzt. Dafür müssen die Golden Images möglichst sauber und wiederverwendbar gebaut werden.

8.2.3 Deployment an Freitagen

Sollte man Deployments an Freitagen durchführen? Es wird ja immer gescherzt, dass der Freitag *read-only* ist: keine Änderungen, keine Deployments, nichts, was Überstunden verursachen könnte!

Eigentlich ist das auch völlig nachvollziehbar: Freitags möchten Sie früher Feierabend machen und keinen Stress mit einer möglichen Downtime nach einem Deployment auf die Produktionsumgebung haben, die nun mal auch bei kleineren Änderungen geschehen kann. Das Wochenende wollen Sie ja schließlich in Ruhe, abseits der Arbeit, verbringen.

Aber gilt diese Logik dann nicht auch für jeden Wochentag? Wo ist der Unterschied, wenn man sicherstellen will, dass man am frühen Abend keine Downtime provozieren will?

Sie werden es wahrscheinlich schon erahnt haben: Es kommt gar nicht auf den Freitag an, sondern es geht schlicht um die Frage, ob es zu »kritischen« Terminen Änderungen in der Produktionsumgebung geben soll. Bei einem Online-Shop wie *schickgekleidet.de* spielt nicht nur der Wochentag und die Uhrzeit eine Rolle, sondern auch die Frage, ob gerade eine größere Marketing-Kampagne läuft. Das Weihnachtsgeschäft ist für viele Firmen im Einzelhandel sehr wichtig, hier darf also eigentlich keine Downtime passieren. Sollte man daher während dieser Zeit komplett auf Deployments verzichten?

Nein, denn man kann auch den kritischen Deployments zu schwierigen Zeiten ein wenig »den Zahn ziehen«. Das Ziel einer eingespielten DevOps-Umgebung ist, dass Deployments

- in kleinen, überschaubaren Teilen vorliegen, sodass man die Auswirkungen abschätzen kann,
- alle vorherigen Stufen der Qualitätssicherung in der CI-Pipeline durchlaufen haben und man sich sicher ist, dass der Build funktioniert,
- durch Rollback-Mechanismen, Canary- und A/B-Tests begleitet werden. (siehe Abschnitt 8.4).

© Rheinwerk Verlag, Bonn 2024

Wichtig ist zudem zu betrachten, ab wann eine Aufgabe in der DevOps-Welt als »erledigt« angesehen werden kann. Während in der »alten« Welt die Arbeit für die meisten Teammitglieder fertig ist, wenn der Code integriert wurde, arbeiten nach DevOps-Prinzipien alle zusammen weiter daran, dass das Produkt reibungslos bei den Kunden landet. Es kümmern sich also alle weiter um das Deployment in die Produktion und um die Aktivierung der Funktionen, um bei etwaigen Fehlern schnell eingreifen zu können.

Es muss also verinnerlicht werden, dass es um eine andere Art zu arbeiten geht. Wenn man Continuous Delivery und Continuous Deployment *richtig* macht, dann sind Deployments am Freitag kein Problem. An solchen kritischen Tagen sollte trotzdem viel Zeit eingeplant werden, damit erst dann gemergt und deployt wird, wenn man das Ergebnis auch entsprechend überwachen und beobachten kann. Wichtig ist auch hier: Machen Sie nur so viel, wie Sie und Ihr Team sich zutrauen. Das ist ein Prozess, der nicht sich nicht von heute auf morgen ändert. Und nur weil Sie vielleicht vor ein paar Wochen noch ohne Angst und Probleme freitags deployt haben, heißt das nicht, dass es sich zukünftig nicht wieder ändern kann. Treten Sie also lieber einen Schritt zurück, und schauen Sie, was man verbessern kann, falls es zu Problemen kam. Ganz unabhängig davon, welcher Wochentag es nun gerade war.

Reflexion

Ich kann immer nur wiederholen, dass der Weg hin zu Continuous Delivery nicht nur eine rein technische Angelegenheit ist. Den Schalter in den Köpfen des nun crossfunktionalen Teams umzulegen, ist nicht sonderlich einfach, zumindest wenn man »richtiges« Continuous Delivery erreichen möchte.

Das sieht man in der Praxis auch heute noch eher selten. Viele Firmen im deutschsprachigen Raum behaupten zwar, dass sie Continuous Delivery machen, aber wenn ich dann ein wenig nachbohre, fällt schnell auf, dass vielleicht alle zwei Wochen am Ende des Sprints deployt wird. Das ist zwar sehr viel besser, als alle sechs Monate ein Deployment auszurollen, Continuous Delivery ist das allerdings auch nicht!

Sowohl auf der technischen als auch auf der kulturellen Ebene kann ich nur empfehlen, dass Sie die Frequenz der Deployments stetig erhöhen. Und wenn Sie auf Widerstand weiter oben in der Hierarchie stoßen, dann sollten Sie auf jeden Fall schauen, dass zumindest eine Staging-Umgebung aufgebaut wird und dass auch mit Review-Umgebungen gearbeitet wird.

Wichtig ist auch, dass Sie die Umstellung mit messbaren Metriken begleiten. Damit lässt sich deutlich besser quantifizieren, ob etwas erfolgreich war oder nicht. Viel zu häufig sieht man eher Deployments nach dem Motto: »Wird hoffentlich schon funktionieren.« Aber Hoffnung ist keine Strategie!

Wenn das Deployment reibungslos funktioniert, dann kann das ein Grundstein für folgende Diskussionen sein, um nicht nur kontinuierlich auf Staging-Umgebungen zu deployen, sondern auch auf Produktionsumgebungen.

Persönliches Exemplar für Thomas Bartholomäus

8.3 Build-Management für Deployments

Der Bau von Anwendungen sollte immer aus dem Quellcode geschehen. Das heißt, dass mit dem Klonen des Repositorys und den darin auszuführenden Build-Schritten alle nötigen Abhängigkeiten selbstständig heruntergeladen werden.

8.3.1 Die Frage nach den Versionsnummern

Die Frage, wie und ob Versionsnummern vergeben werden, ist besonders im DevOps-Umfeld ein viel diskutiertes Thema. Wenn oft und schnell deployt wird, ändern sich die Versionsnummern rasch und verlieren dementsprechend ihre Aussagekraft. Die Zeiten von Windows 95 und 98, in denen man das Jahr des Releases an der Versionsnummer ablesen konnte, sind vorbei.

Wie und ob es überhaupt eine Versionszählung gibt, hängt davon ab, ob »echtes« Continuous Delivery genutzt wird und ob Ihre Software extern, d. h. außerhalb Ihres eigenen Projekts, genutzt wird.

Eindeutige Versionsnummern sind insbesondere dann wichtig, wenn Ihre Software extern als Abhängigkeit genutzt wird. Wenn Sie also eine Software veröffentlichen, die auch von anderen bezogen und installiert werden soll, sollten Sie ein eindeutiges Schema nutzen, das sich am *Semantic Versioning* orientiert. Schließlich wollen sowohl Sie als auch die Nutzer der Software wissen, welche Version welche Features bietet und in welcher Version bestimmte Fehler nicht mehr enthalten sind. Kommunizieren Sie das klar, und halten Sie sich stringent an die Vorgaben des Schemas.

»Externe Abhängigkeiten« heißt allerdings nicht, dass es ausschließlich außerhalb der eigenen Organisation relevant ist. Eine externe Abhängigkeit kann auch innerhalb der Organisation existieren, zum Beispiel wenn ein anderes Team von diesem Projekt abhängt. Wenn das nicht der Fall ist, dann benötigt man auch keine klassischen Versionsnummern. Denn gerade bei echter Continuous Delivery, mit der Nutzung von A/B-Tests, Canary-Deployments und so weiter gibt es zahlreiche Versionen, die gleichzeitig laufen können.

Semantic Versioning

Semantic Versioning dürfte wohl die bekannteste Art der Definition von Versionsnummern sein. Auf der Webseite *https://semver.org/lang/de/* finden Sie eine längere Beschreibung des Konzepts. Die Zusammenfassung ist, dass eine Versionsnummer aus drei Teilen mit diesem Muster besteht:

```
MAJOR.MINOR.PATCH
```

1. `MAJOR` wird erhöht, wenn API-inkompatible Änderungen veröffentlicht werden,
2. `MINOR` wird erhöht, wenn neue Funktionalitäten, die kompatibel zur bisherigen API sind, veröffentlicht werden, und

© Rheinwerk Verlag, Bonn 2024

3. `PATCH` wird erhöht, wenn die Änderungen ausschließlich API-kompatible Bugfixes umfassen.

Außerdem sind Bezeichner für Vorveröffentlichungen und Build-Metadaten als Erweiterungen zum `MAJOR.MINOR.PATCH`-Format verfügbar.

In der Praxis sieht man allerdings immer wieder, dass dieses Namensschema nicht reibungslos funktioniert: Häufig verstecken sich in Patches dann doch neue Versionen. Teilweise sind in Minor-Versionen wiederum API-inkompatible Änderungen enthalten, die dort eigentlich nicht hingehören.

Das ist gar nicht so verwunderlich: So liest sich das Schema zwar eindeutig, aber es ist nicht immer klar, wie eine Änderung bewertet werden muss. Wenn man etwa an einem Dienst arbeitet, ist eine Fehlerkorrektur nicht immer ohne Kompatibilitätsbruch korrigierbar. Das müsste dann streng genommen als Minor-Version veröffentlicht und entsprechend kommuniziert werden, dabei ist der Fehler in der bestehenden Version enthalten, worauf andere dann Übergangslösungen bauen müssen.

Interne Versionsnummern

Wenn die eigene Software keine externen Abhängigkeiten hat, dann kann man theoretisch komplett ohne Versionsnummern auskommen – oder zumindest ohne klassische, »künstliche« Versionsnummern. Eine Art von Versionierung wird weiterhin benötigt, denn beim Einsatz von echtem Continuous Delivery braucht man eine Versionsbezeichnung, um nachvollziehen zu können, welcher Stand auf den verschiedenen Umgebungen läuft.

Ein Ansatz ist, dass man schlicht die Commit-IDs des Git-Repositorys verwendet. Hierfür wird kein separates Tagging benötigt, da man anhand der Commit-ID den Stand des Projektes ermitteln kann. Da idealerweise ohnehin regelmäßig deployt wird, reicht eigentlich die Commit-ID für sich aus.

Ideal ist das aber auch nicht in jedem Fall. Gegebenenfalls kann die Commit-ID mit einer Versionsnummer kombiniert werden, sodass etwas Ähnliches wie Semantic Versioning (SemVer) umgesetzt wird. Das ist gerade dann hilfreich, wenn man auf Dashboards schaut, wo verschiedene laufende Versionen anhand von Versionsnummern identifiziert werden müssen. Die Commit-ID sorgt dann für die Rückverfolgbarkeit.

8.3.2 Paketierung

Bei der Paketierung gibt es gar nicht so viel zu beachten. Natürlich spielt hier die Versionsnummer wieder eine Rolle, aber ansonsten muss »nur« ein auslieferbares Paket erstellt werden, das die Build-Artefakte enthält.

Je nachdem, ob es sich um interne oder auch um externe Releases handelt, müssen verschiedene Arten von Paketformaten unterstützt werden. Das ist abhängig von der verwendeten Programmiersprache.

Grundsätzlich wird zwischen der *programmiersprachenabhängigen Paketierung* und der *Paketierung für die Zielplattform* unterschieden. Letztere kann sowohl ein Container-Image sein als auch ein Paket für Deb- und RPM-basierte Linux-Distributionen.

Im Rahmen des Paketbaus ist ein Aspekt diskussionswürdig. Ich sehe oft, dass Pipeline-Jobs zu oft laufen und Pakete wiederholt gebaut werden. So läuft die Pipeline zum ersten Mal, wenn das Release für eine Staging-Umgebung gebaut und deployt wird. Wenn das erfolgreich war, wird die Pipeline erneut gestartet, das Projekt abermals gebaut und getestet und die Artefakte werden auf die Produktivsysteme deployt.

Wenn man es richtig macht, dann braucht man das Projekt nicht noch einmal neu zu bauen. Ganz im Gegenteil: Es ist ungünstig, wenn nicht das genau gleiche Paket genutzt wird, das auch durch die Staging-Umgebung gelaufen ist. Es ist also besser, wenn das Paket einmal gebaut und dann durch die verschiedenen Umgebungen durchgereicht wird. So wird sichergestellt, dass auch wirklich das Paket ausgerollt wird, das man auch zuvor getestet hat.

Auch hier spielen ein paar mehr Faktoren hinein, wie oft und auf welche Umgebungen man tatsächlich deployt. Wichtig ist hier auch, dass man eine Package-Registry nutzt, um eben das Projekt nur einmal zu bauen und beim Deployment das Artefakt aus der Package-Registry zu ziehen. Um die Package-Registry und auch um die Container-Registry geht es in wenigen Abschnitten, zuvor lohnt sich ein Blick auf die Containerisierung im Build-Prozess.

8.3.3 Containerisierung

Bei der Container-Technologie verzahnen sich CI und CD stark, denn wenn im CI-Teil der CI/CD-Pipeline ein Container gebaut wird, wird er in der Regel dann auch deployt.

In diesem Abschnitt geht es erst einmal nicht darum, wie und wann Container im Betrieb relevant und praktisch sind und wann man diese gegenüber Bare Metal und VMs bevorzugen sollte. Springen Sie dazu in Abschnitt 9.2.

Containerbau für Anwendungen

Ob Sie Ihre Anwendung in einen oder mehrere Container portieren können, hängt von der Architektur der Software ab. Gerade Webanwendungen wie der Webshop von *schick-gekleidet.de* eignen sich meist sehr gut dafür.

Grundsätzlich funktioniert der Containerbau so: Alle Informationen, die für den Bau benötigt werden, werden in einer Baubeschreibung deklariert, dem sogenannten

© Rheinwerk Verlag, Bonn 2024

Dockerfile. Dort legen Sie zunächst das *Base-Image* fest, das das Fundament darstellt. Im allereinfachsten Fall sind das sehr schlanke Linux-Betriebssysteme, die nur ein Minimum an Funktionen mitbringen. So sorgen Sie dafür, dass Ihre Container möglichst wenig Ballast mit sich herumtragen; es kann aber einfacher sein, auf ein Image zurückzugreifen, in dem schon ein Webserver wie Nginx oder Apache vorinstalliert sind.

Dann werden weitere Informationen spezifiziert: Welche Dateien müssen wohin kopiert werden? Welche Ports müssen geöffnet und weitergeleitet werden? Welche Umgebungsvariablen müssen gesetzt sein? Insgesamt kann dies so wie in Listing 8.1 aussehen:

```
# Verwenden des offiziellen Node.js-Basisimages aus dem Docker Hub
FROM node:14
# Arbeitsverzeichnis innerhalb des Containers festlegen
WORKDIR /app
# Die package.json und package-lock.json
# in das Arbeitsverzeichnis kopieren
COPY package*.json ./
# Abhängigkeiten Ihrer Anwendung installieren
RUN npm install ...
# Restlichen Anwendungscode in das Arbeitsverzeichnis kopieren
COPY . .
# Port exponieren, den Ihre Anwendung verwendet (z. B. 3000)
EXPOSE 3000
# Anwendung beim Containerstart starten
CMD [ "node", "app.js" ]
```

Listing 8.1 Ein Dockerfile, das eine Webanwendung verwaltet

Mit dieser Beschreibung können Sie eine Container-Engine wie etwa *Docker* anweisen, Ihre Anwendung zu bauen und zu starten. So haben Sie stets eine reproduzierbare Anleitung für den Bau.

Linting für den Container-Image-Bau

Es gibt verschiedene Tools, die das *Dockerfile* linten, also auf Fehler und Unstimmigkeiten prüfen können. Ein Tool dafür ist etwa kics: *https://kics.io/*

Die Wahl des Base-Images

Bei der Wahl des Basis-Betriebssystems haben Sie viele Möglichkeiten. Sie können selbstverständlich eine »normale« Linux-Distribution wie Ubuntu oder Debian nutzen; sinnvoller ist es aber zumeist, auf spezialisierte Distributionen wie Alpine Linux zurückzugreifen. Red Hat bietet unter *https://catalog.redhat.com/software/containers/search* besonders zertifizierte Base-Images an.

Achten Sie stets darauf, dass das Base-Image wirklich vom Upstream-Projekt kommt und auch geprüft ist. Verwenden Sie keine Container-Images für den Build-Prozess, deren Ursprung Sie nicht genau verifizieren können, und denken Sie an die Grundprobleme der Software-Supply-Chain (siehe Abschnitt 11.5.)

Wenn Sie Golang als Programmiersprache verwenden, können Sie sich sogar das Basis-Image sparen: Da Kompilate von Golang-Projekten alle Abhängigkeiten in einer einzelnen ausführbaren Binärdatei mitbringen, kann ein solches Container-Image nur aus einer Datei bestehen.

Grundsätzlich sollten Container-Images möglichst klein und schlank gehalten werden, denn sie werden immer wieder neu gebaut und deployt. Je kleiner sie sind, desto schneller läuft alles ab. Aber nicht nur das: Wo nichts ist, kann nichts kaputtgehen und für Sicherheitsprobleme sorgen. Und nebenbei spart man sich Kosten beim Storage für die Container-Images.

Außer auf das geeignete Base-Image sollte beim Bau auf sogenannte *Multi-Stage-Builds* gesetzt werden, damit Build-Tools, die nicht für die Ausführung des Containers gebraucht werden, nicht im Container-Image landen und es aufblähen. Dazu wird ein Clean-up im Build-Schritt ausgeführt, in dem alles Unnötige entfernt wird. Außerdem sorgt das Caching von Container-Image-Layern dafür, dass der Bau schneller abläuft.

Buchempfehlung

Beim Bau und bei der Ausführung von Containern gibt es natürlich noch viel mehr zu beachten. Ein guter Startpunkt ist das Buch »Docker« von Bernd Öggl und Michael Kofler, das auch im Rheinwerk Verlag erschienen ist.

Schauen Sie sich außerdem die Best Practices für Dockerfiles an, die Sie in der Dokumentation von Docker finden:

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Die Container-Engine

Den Bau und das Ausführen der Container übernimmt die Container-Engine. Docker war dabei lange Zeit der marktbeherrschende Platzhirsch, sodass Software-Container auch heute oft noch »Docker-Container« genannt werden und man stets von Dockerfiles spricht. Docker Inc., die Firma hinter Docker, hat in den letzten Jahren jedoch einige Entscheidungen getroffen, die besonders bei IT-Profis auf Kritik stießen. Unter dem Namen *Open Container Initative* (OCI) hat sich eine Gruppe gebildet (zu der auch Docker Inc. gehört), die Standards für Software-Container formuliert.

© Rheinwerk Verlag, Bonn 2024

Daher gibt es inzwischen einige Alternativen zu Docker, die OCI-kompatible Images bauen und starten können. *Podman* mit dem Build-Werkzeug *Buildah* sind dabei am wichtigsten; insgesamt werden Sie einen ganzen Zoo an Tools finden, wenn Sie sich mit dem Thema näher beschäftigen.

Meine Empfehlung lautet: Machen Sie sich über die Container-Engine nicht zu viele Gedanken. Ja, es gibt Unterschiede, und zwar sehr gewichtige, die die Architektur von Docker und seinen Nachfolgern betreffen. Dabei geht es besonders um die Frage, welche Rechte die Container-Engine selbst haben muss, um Container ausführen zu können. Das ist nicht unwichtig, spielt im Entwicklungsalltag allerdings keine so große Rolle. Wichtiger ist es, dass die Container-Images sinnvoll abgespeichert, versioniert und verwaltet werden, damit man sie für das Deployment verwenden kann. Darum geht es gleich im nächsten Abschnitt.

8.3.4 Container-Registry und Package-Registry

Sowohl die Pakete als auch die Container-Images müssen in einer Registry abgespeichert werden, damit sie verwaltet und für weitere Schritte verwendet werden können.

Meiner Meinung nach ist dies das *langweiligste* Werkzeug im DevOps-Toolstack. Die Kernfunktionalität ist bei allen gleich, wenn man es stark vereinfacht. Die wichtigsten Vertreter sind *Nexus* von Sonatype und *Artifactory* von JFrog:

- *https://www.sonatype.com/products/sonatype-nexus-repository*
- *https://jfrog.com/artifactory/*

Auch GitHub und GitLab bieten eigene Container- und Package-Registrys an. Der wesentliche Vorteil bei diesen Angeboten ist, dass keine komplexe Berechtigungsprüfung durchgeführt werden muss, da sich diese Registrys nahtlos in das Berechtigungskonzept innerhalb der Projekte und Repositorys einreihen. Allerdings beinhalten diese »eingebauten« Registrys nicht alle Funktionen, die gerade für Enterprise-Kunden wichtig sind. Für kleinere Projekte reichen sie sicherlich aus; in größeren Firmen findet man eher Nexus oder Artifactory.

Beide Lösungen bringen alles mit, was man für das Management von Paketen und Containern braucht. Sie unterstützen die verschiedensten Programmiersprachen und Paketformate. Das macht die Verwaltung für das Entwicklungsteam einfach.

Paketmanager und Registrys

Neben diesen Registrys, die grundsätzlich alle Artefakte verwalten können, gibt es auch Paketmanager, die nur für einzelne Programmiersprachen entwickelt werden. Prinzipiell können Sie auch solche Tools nutzen (etwa *Pip* für Python), aber so schaffen Sie ein Sammelsurium an Tools, das die Integrationsarbeit erhöht. In einem sehr

kleinen Team mit einer sehr (!) überschaubaren Zahl an Programmiersprachen und Tools kann das funktionieren. In einem Unternehmensumfeld mit verschiedenen Teams, zahlreichen Projekten und Dutzenden Artefakten, die jeden Tag gebaut und aktualisiert werden, sieht die Sache aber anders aus. Dort kommen Sie um ein professionelles Artefakt-Management nicht herum.

Auch für die Container-Registry gibt es noch weitere Standalone-Tools. Neben dem bereits erwähnten Docker Hub trifft man auch zudem auf Harbor, (*https://goharbor.io/*), das sich in der eigenen Infrastruktur leicht hosten lässt und sich nur auf Container-Images konzentriert. Aber auch die gängigen Cloud-Provider bieten ihre eigene Container-Registry an.

Obwohl die reine Speicherung und Verwaltung nicht sonderlich spannend ist, sollten Sie trotzdem genau hinschauen. Denn die Registry verwaltet alle Artefakte, die aus der Entwicklungsarbeit entstehen. Daher muss geschaut werden, welche Personengruppen welche Pakete hochladen, löschen und verändern dürfen. Auch sollten Sie Retention-Policys definieren, die festlegen, wie lange Artefakte vorgehalten werden. Für den effektiven Einsatz der CI/CD-Pipeline ist das sehr wichtig, denn es werden tagtäglich etliche Pakete und Container gebaut. Alles kann und muss man nicht aufheben, dafür würde der Speicherplatz gar nicht reichen. Es muss aber gewährleistet sein, dass wichtige Artefakte archiviert werden. Dazu ist eine gute Versionierung wichtig.

Dependency-Proxy

Ein weiteres wichtiges Feature ist das Dependency-Proxy, das sich um die Verwaltung von Abhängigkeiten kümmert und diese vorhält. Das Dependency-Proxy zieht die benötigten Abhängigkeiten aus den Primärquellen. Das spart bei jedem Durchlauf der Pipeline Zeit, da nicht jedes Mal kleinere oder größere Abhängigkeiten aus dem Internet gezogen werden müssen, und es sorgt zudem dafür, dass weniger Speicherplatz und Netzwerkbandbreite gebraucht wird.

Application-Security

Während die Ablage der Artefakte eher ein Verwaltungsakt ist, ist die *Application-Security* ein spannender Aspekt. Sowohl Nexus als auch Artifactory bringen eigene Security-Scans mit und können auch außerhalb der Pipeline Artefakte scannen. Sie können dann festlegen, dass nur Pakete oder Images abgelegt werden dürfen, bei denen keine (gravierenden) Sicherheitslücken gefunden wurden. So können Sie für alle Projekte bestimmen, dass Abhängigkeiten nur gepullt werden können, wenn keine kritischen Sicherheitslücken enthalten sind.

Auf den ersten Blick klingt das nicht verkehrt: So vermeidet man, dass unsichere Software ausgerollt wird. Sie sollten diese Features daher durchaus nutzen und entsprechend konfigurieren.

© Rheinwerk Verlag, Bonn 2024

Verlassen Sie sich allerdings nicht zu sehr darauf, und treten Sie wieder ein paar Schritte zurück, um sich das Big Picture anzuschauen: Die Registry wird erst relativ spät im Entwicklungsprozess aktiv, was nicht gut für die Visibility ist. Wenn bereits in der Entwicklungsphase veraltete Abhängigkeiten mit Sicherheitslücken genutzt werden, fliegt Ihnen das Problem erst im Deployment um die Ohren. Das ist viel zu spät, denn dann ist die Entwicklung schon längst abgeschlossen. Das widerspricht dem *Shift-Left-Ansatz* (siehe Abschnitt 11.3.2), der besagt, dass diese Scans nicht nur so früh wie möglich ausgeführt werden sollten, sondern dass auch die Visibilität möglichst früh gegeben sein sollte.

Wenn Sie daher Abhängigkeiten über ein Dependency-Proxy verwalten, sollte es als ein transparentes Proxy arbeiten, damit es nicht erst beim Deployment aktiv wird, sondern für alle Pulls während der Entwicklungsphase gilt. Somit werden dann nicht nur die Pipelines beim Deployment kontrolliert, sondern es werden auch Abhängigkeiten geprüft, die das Entwicklungsteam lokal nutzt, um das Projekt zu bauen. Das sorgt dafür, dass nur »saubere« Pakete und Container-Images verwendet werden.

Wie schon bei der Frage nach der Container-Engine ist es mir auch hier wichtig, zu betonen, dass die Frage nach dem »richtigen« Tool nicht entscheidend ist. Viel wichtiger ist, dass Security-Scans an der richtigen Stelle ausgeführt werden. Und noch viel wichtiger ist, dass sie auch ausgewertet und korrigiert werden!

Reflexion

Die Wahrscheinlichkeit ist hoch, dass Sie beim reinen Build-Prozess nicht viel anpassen müssen. Pakete, die gebaut werden, landen grundsätzlich in der Package-Registry.

Das Gleiche gilt auch für den Container-Image-Bau. Achten Sie hier darauf, dass die gebauten Container-Images möglichst schlank sind und auch wirklich stetig neu gebaut werden.

Folgende Fehler stören typischerweise den Projektverlauf: Artefakte werden nicht richtig abgelegt oder sind nicht aktuell. Stellen Sie sicher, das so etwas nicht passiert!

8.4 Rollbacks, Kanarienvögel und Feature Flags

Bisher ging es in diesem Kapitel darum, welche Einflüsse die DevOps-Prinzipien auf die technische Umsetzung von CI/CD haben können. Primär wurde im Rahmen von CI/CD auf das Vertrauen in die Software und die Prozesse geschaut. Dabei wurde immer darauf gesetzt, dass das Deployment von jetzt auf gleich vollständig ausgerollt wird.

Wenn Sie sich bisher gefragt haben, ob es wirklich eine gute Idee ist, alle Änderungen eines Deployments in einem Rutsch freizuschalten, vor allem wenn Deployments

gleich mehrfach pro Tag ausgeführt werden, kann ich Sie beruhigen: Es gibt diverse Möglichkeiten, um zwar oft und viel zu deployen, aber nicht gleich alles gleichzeitig in Betrieb zu nehmen.

Genau genommen geht es um die Unterschiede zwischen Continuous Delivery und Continuous Deployment. Beide sind nah verwandt und werden häufig auch für das Gleiche gehalten, doch gibt es ein paar Unterschiede:

- Bei der *Continuous Delivery* werden kontinuierlich ein oder mehrere Software-Pakete ausgeliefert. Dies ist in der Regel unabhängig von der Umgebung, es geht also viel mehr darum, *etwas* ausliefern zu können, unabhängig davon, ob es auch in Betrieb geht.
- Beim *Continuous Deployment* geht man abermals einen Schritt weiter und nimmt die Änderungen dann auch in Betrieb. Continuous Delivery ist also Teil des Deployments. Beim Continuous Deployment geht es eben darum, dass der komplette Prozess mit abgebildet ist: vom Coden über das Testen hin zum Deployment.

Das ist in etwa so, wie ein Waschmaschinen-Hersteller ständig Waschmaschinen an Kunden ausliefert. Das bloße Ausliefern heißt noch nicht, dass die Maschinen auch direkt in Betrieb gehen, das wäre schließlich der nächste Schritt.

8.4.1 Rollbacks

Wenn man mehrere Male pro Tag die Software ausrollt, kommt es unweigerlich zu Fehlern. Es werden zwar nicht die riesigen Fehler wie bei einem halbjährlichen Deployment sein, doch können Probleme nie vollständig vermieden werden. Daher geht es im DevOps-Umfeld immer um die Fehlerkultur: Was tut man, wenn Fehler passieren, und welche Schlüsse zieht man daraus?

Dies ist zwar in erster Linie ein menschlicher Aspekt, doch schauen wir uns an dieser Stelle erst einmal an, wie man auf technischer Ebene ansetzen kann, um die Fehler und die aus ihnen folgenden Kosten zu minimieren.

Dank vollständiger Release-Automatisierung lassen sich nicht nur neue Versionen veröffentlichen, sondern lässt sich auch ebenso schnell ein Rollback durchführen. Dazu muss nur die Pipeline ausgeführt werden, um eine ältere Version auszurollen. Ganz so einfach ist es aber natürlich nicht:

- Wenn jegliche Downtime vermieden werden soll, dann ist ein vollständiges Rollback auf eine ältere Version am sinnvollsten. Dazu muss bekannt sein, welche Version noch einwandfrei läuft.
- Wenn allerdings nur vereinzelt Fehler auftreten, die das System weder komplett noch teilweise aus dem Betrieb nehmen, dann kann es sinnvoller sein, schnell einen Fix zu entwickeln und diesen unmittelbar auszurollen.

© Rheinwerk Verlag, Bonn 2024

Die Entscheidung, welcher Weg im Fehlerfall eingeschlagen wird, ist nicht einfach und hängt stets vom individuellen Fall ab. Sie brauchen unbedingt ein gutes Monitoring, das das Deployment transparent macht, damit Sie einschätzen können, ob eine schnelle Fehlerkorrektur »vorwärts« oder eine Zurücknahme der Änderungen der beste Weg ist.

Gut, dass Sie diese Entscheidung in einem DevOps-Team nicht alleine treffen müssen, sondern auf die Expertise und die Erfahrung sowohl der Admins als auch der Entwickler zurückgreifen können! Hören Sie dabei aber wirklich alle Beteiligten: Die Entwicklerin, die am Feature gearbeitet hat, kann wahrscheinlich am besten einschätzen, wo es hakt. Wichtig ist dabei, dass im Fehlerfall konstruktiv zusammengearbeitet wird und man sich nicht in Schuldzuweisungen verstrickt (siehe Abschnitt 9.4.2, »Blameless Post-Mortems«.)

Datenbankschemata

Die wohl wichtigste Frage bei einem Rollback ist, wie Änderungen an den Datenbankschemata gehandhabt werden. Wenn also mit einem Release das Datenbankschema verändert wurde und es keine Abwärtskompatibilität gibt, dann funktioniert ein Rollback nicht, weil es sonst zu anderen Fehlern in der Anwendung führt.

Besonders in solchen Fällen ist es unbedingt notwendig, dass das Entwicklerteam das Deployment begleitet, weil sonst durch ein fehlerhaftes Rollback Datenverlust und Inkonsistenz entstehen können.

8.4.2 Schrittweises Aktivieren mittels Blue-Green- und Canary-Deployments

Eine Möglichkeit, um die Zahl fehlerhafter Deployments zu reduzieren, sind die Blue-Green-Deployments sowie Canary-Deployments. Beide Deployment-Arten sind technische Möglichkeiten, um beim Ausrollen von Deployments nicht alle Änderungen gleichzeitig für die Endnutzer verfügbar zu machen.

Blue-Green-Deployments

Bei Blue-Green-Deployments gibt es zwei Systeme, auf denen die Software in zwei verschiedenen Versionen deployt wird. Das ist zum einen das gerade aktive (grüne) Deployment und das gerade inaktive (blaue), was dem nächsten Deployment entspricht.

Bei der Software-Entwicklung werden die Tests möglichst voll automatisiert, und möglichst viel soll so früh wie möglich getestet werden. Allerdings lassen sich nicht alle Sachen auf den Review- und Staging-Umgebungen automatisiert testen, und auch das manuelle Testen kann bei diesen Umgebungen an Grenzen stoßen. Letzt-

lich kann man wirklich erst sicherstellen, ob alles funktioniert, nachdem die neueste Version auf die Produktivumgebung deployt wurde.

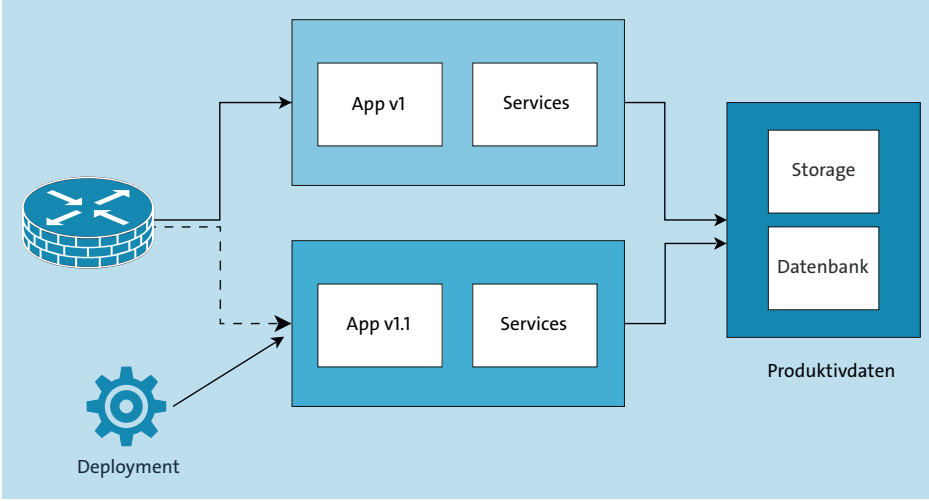

Abbildung 8.8 Bei Blue-Green-Deployments werden zwei Versionen der gleichen App deployt, aber nur eine wird mit Traffic versorgt.

Die Idee hinter Blue-Green-Deployments ist, dass man das Deployment der Anwendung sowie die Inbetriebnahme in zwei Schritte aufteilt. Die bisherige Version der Anwendung läuft zunächst normal auf der Produktivumgebung weiter und wird zur »grünen« Umgebung.

Wenn nun eine neue Version deployt wird, dann wird die »blaue« Umgebung genutzt. Die Umgebung ist inaktiv, das heißt, es kann zwar auf sie deployt werden, allerdings führt kein Traffic auf diese Systeme. Es ist also immer nur eines der beiden Deployments gerade aktiv.

Falls Fehler auftreten, besteht keine große Eile, da nur die inaktive Umgebung angefasst wurde und die aktive Umgebung weiterhin mit der alten Version läuft.

[!]

Ein Blue-Green-Deployment ist nicht überall möglich

Wichtig ist allerdings, dass die Software auch so geschrieben ist, dass sie zweimal in verschiedenen Versionen gleichzeitig laufen kann und dass Fehler auch auftreten, wenn noch kein Nutzertraffic vorhanden ist. Wie auch bei Rollbacks sind hier Datenbankschema-Änderungen mit großer Vorsicht zu betrachten, denn bei solchen Änderungen wird es auch hier zu Problemen kommen.

Der Vorteil von Blue-Green-Deployments ist, dass es direktes Feedback gibt, ob das Deployment auf technischer Ebene funktioniert. Falls nicht, können in Ruhe weitere Deployments nachgeschoben werden, bis die Fehler korrigiert sind. Sofern keine

© Rheinwerk Verlag, Bonn 2024

Beeinträchtigungen zu erwarten sind, kann das System umgestellt werden, sodass die Nutzer auf die neu ausgerollte Version zugreifen und nicht mehr auf die alte.

So wird sichergestellt, dass immer eine funktionsfähige Version in Betrieb ist. Falls es doch zu Fehlern kommen sollte, kann auch schnell ein Rollback durchgeführt werden, bei dem noch nicht einmal die Deployment-Pipeline mit der älteren Version ausgeführt werden muss. Stattdessen legt man einfach den Schalter im Loadbalancer um, um wieder die Nutzer auf den älteren Stand der Software zu schicken.

Warum macht man dann nicht immer Blue-Green-Deployments? Weil es sehr aufwendig und teuer ist, eine komplett redundante zweite Umgebung vorzuhalten. Vor allem wenn keine Skalierung durch Cloud-Systeme gegeben ist, ist das im Alltag meist einfach zu teuer und wird nur für besonders schwierige oder große Deployments genutzt.

Außerdem sind Blue-Green-Deployments nicht immer möglich, nämlich dann nicht, wenn große Änderungen am Datenbankschema unvermeidlich sind. Bei solchen Änderungen muss die Architektur der Anwendung so angepasst werden, dass Änderungen am Code abgekoppelt von den eigentlichen Änderungen an der Datenstruktur sind. Diese Änderungen müssen implementiert und bekannt sein und könnten unter Umständen die Komplexität erhöhen.

Canary-Deployments

Canary-Deployments sind eine Erweiterung des Blue-Green-Deployments. Die Grundidee und der Name stammen aus dem Bergbau, wo früher Kanarienvögel in Kohleminen eingesetzt wurden. Die Vögel dienten als Indikator: Wenn in der Mine zu viele giftige Gase austraten, verendeten die Kanarienvögel, was für die Arbeiter in der Kohlemine das Zeichen war, die Mine schleunigst zu verlassen.

Bei Canary-Deployments werden zum Glück keine Kanarienvögel geopfert. Wie auch bei den Blue-Green-Deployments gibt es bei Canary-Deployments zwei Produktivumgebungen, auf denen die Deployments ausgeführt werden können (siehe Abbildung 8.9):

- die Umgebung, auf der die bisherige Version läuft, und
- eine zweite inaktive Umgebung, auf der dann die neue Version läuft.

Zu Beginn ist hier also alles wie bei den Blue-Green-Deployments. Die Umgebungen werden binär an- oder ausgeschaltet. Das heißt konkret, dass die Umgebungen entweder vollständig aktiv oder inaktiv sind. So läuft eben der Traffic der Endnutzer entweder komplett oder gar nicht über eine der Umgebungen.

Anders sieht es bei Canary-Deployments aus: Hier wird stetig ein kleiner Prozentsatz des Traffics auf das neue Deployment umgeleitet. In dieser Zeit wird beobachtet, ob es bei echtem, produktivem Traffic zu Fehlern kommt.

Persönliches Exemplar für Thomas Bartholomäus

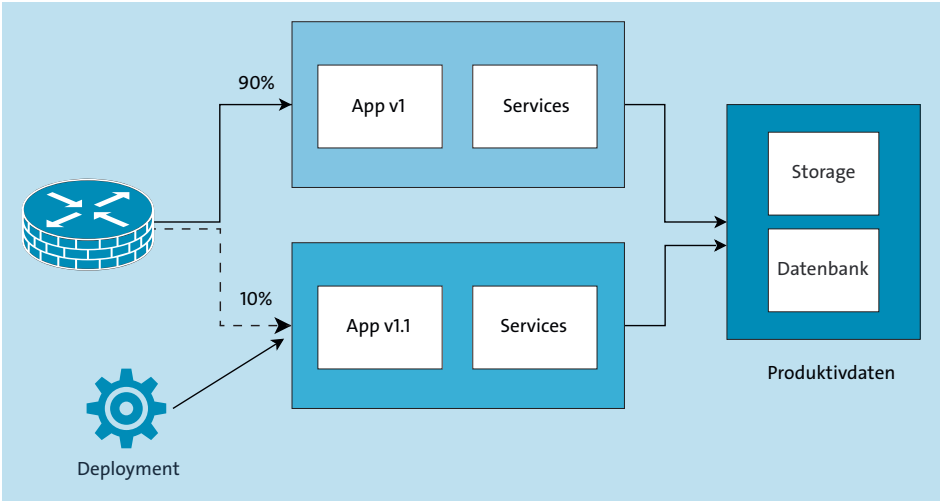

Abbildung 8.9 Bei Canary-Deployments wird nur ein Teil des Traffics auf die neue Version geleitet.

Bei einem neuen Deployment mit der neuen Version wird schrittweise der Prozentsatz hochgedreht. Hier gibt es zwei Möglichkeiten:

- Entweder gibt es genau eine Umgebung mit mehreren Servern, auf denen die Artefakte schrittweise ausgetauscht werden und damit der Prozentsatz des neuen Deployments hochgeschraubt wird,
- oder das Deployment wird auf zwei gleich großen Umgebungen ausgespielt und der Traffic wird umgeleitet. Dieser Weg über einen Loadbalancer ist die häufigere Variante, die entsprechend von der Infrastruktur unterstützt werden muss.

Die großen Cloud-Dienstleister bieten für ihre Deployment-Umgebungen verschiedene Strategien an, und auch über Kubernetes sind Canary-Deployments möglich. Das große Problem dabei ist die gleichzeitige Überwachung des Systems. Es lässt sich zwar bis zu einem gewissen Grad alles automatisieren, aber das hilft nicht viel, wenn man nicht weiß, ob die Bedingungen für ein reibungsloses Deployment erfüllt werden.

Canary-Deployments sind daher in der Theorie wesentlich einfacher als in der praktischen Umsetzung und vollkommen von gutem Monitoring abhängig. Wenn Sie nicht live analysieren können, wie das Deployment abläuft, bringt die schrittweise Umsetzung wenig und Sie fahren mit einfachen Blue-Green-Deployments besser.

Unabhängig davon, wie die technische Umsetzung aussieht, ist der typische Workflow gleich: Während Sie das Deployment »hochschrauben«, beobachten Sie das Monitoring, um zu schauen, ob irgendwelche kleineren oder größeren Fehler auftreten (siehe Kapitel 10, »Vom Monitoring zur Observability«). Dies hängt vor allem

© Rheinwerk Verlag, Bonn 2024

damit zusammen, wie viele Nutzer gerade aktiv sind und welche Funktionen sie nutzen. Wenn es zu keinen Fehlern kommt, kann das neue Deployment weiter hochgeschraubt werden. Falls es jedoch zu Fehlern kommt, kann es wieder heruntergedreht werden, sodass wieder nur die alte Version alleine läuft.

So kann das Deployment gestoppt und rückgängig gemacht werden, sobald Probleme auftauchen. Wenn keine Fehler auftreten, kann das Deployment also schrittweise ausgeweitet werden.

Keep it simple

Solch ein Vorgehen funktioniert natürlich nur, wenn auch eine ausreichende Anzahl an Besuchern das System nutzen. Wenn nur sehr wenige Nutzer das System auslasten, dann kann man sich diese Komplexität getrost sparen.

Unabhängig vom Entdecken funktionaler Fehler können Canary-Deployments auch für andere Tests verwendet werden. Ein Beispiel sind *Performance-Tests*. Diese Tests lassen sich zwar auf Staging-Umgebungen ausführen, auf denen Traffic simuliert wird, allerdings wird man nie zu einer Simulation kommen, wie sich echter Traffic verhält.

Auch hier können Canary-Deployments helfen. Wenn bei einem Release Performance-Verbesserungen implementiert wurden, kann der Traffic schrittweise auf das neue Deployment umgeleitet werden, sodass über das Monitoring geprüft werden kann, ob die Performance-Verbesserungen auch tatsächlich sichtbar sind. Falls es zu Fehlern oder zu hoher Last kommt, lässt sich der Traffic schnell und einfach kontrolliert herunterregeln, ohne dass es zu großartigen Einschränkungen für die Nutzer des Systems kommt.

Sowohl Blue-Green-Deployments als auch Canary-Deployments sind technische Umsetzungen, um Fehlerquellen zu verringern und idealerweise zu vermeiden. Beide Möglichkeiten lassen sich allerdings erst umsetzen, wenn Vertrauen in die Release-Prozesse vorhanden ist, und das geht nicht von heute auf morgen. Auch hier gilt: *Erst die Menschen abholen, dann Prozesse etablieren und dann die Technik*. Andersherum geht es zwar auch, das führt allerdings nur zu mehr Schmerzen, viel Frust und wenig Erfolg.

Eine weitere Möglichkeit, die ebenfalls in Betracht gezogen werden kann, sind Feature Flags.

8.4.3 Feature Flags

Feature Flags, auch *Feature Toggles* genannt, ermöglichen das Aktivieren von einzelnen Features nach dem Deployment. Die Idee dahinter ist, dass neue oder geänderte

Funktionen nach dem Deployment zwar mit einer neu veröffentlichten Version laufen, dass allerdings immer noch der alte Code aktiv ist.

Konkret heißt das, dass im Quellcode zwei Implementierungen vorhanden sind: einmal die bisherige alte sowie die neue Implementierung. Über einen *Feature Flag Service* wird im Quellcode geprüft, welcher Teil des Codes ausgeführt werden sollen. So erlangt man stärkere Kontrolle und eine stärkere Trennung zwischen dem Deployment der Anwendung und dem tatsächlichen Aktivieren der neu implementierten Funktion.

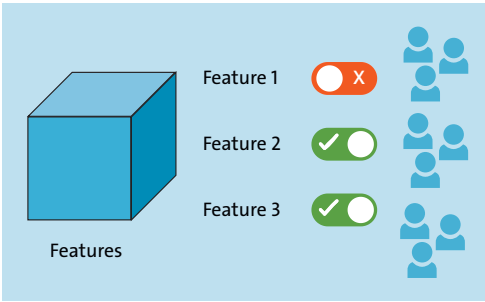

Abbildung 8.10 Feature Flags ermöglichen das Aktivieren diverser Features für einzelne Nutzergruppen.

Für *schick-gekleidet.de* ist folgender Fall üblich: Im Rahmen von Angebotsphasen zu diversen Marketing-Kampagnen müssen speziell angepasste Angebotsseiten zu einer bestimmten Uhrzeit aktiv sein. Wenn die Angebotsphase der groß beworbenen Kampagne beginnt, muss also die neue Version lauffähig und aktiv sein – was bisher allerdings häufig nicht gepasst hat, weil Deployments viel zu selten ausgerollt wurden. Die Deployments mussten zudem genau zu dieser Uhrzeit durchgeführt werden. Keine Minute früher oder später.

Eine bessere Alternative bieten Feature Flags. Das Deployment kann schon viel früher ausgerollt werden, sodass ein Stressfaktor reduziert wird. Die Angebotsseite ist zum Zeitpunkt des Deployments noch deaktiviert, hat also keinen Einfluss auf das Live-System. Zum Start der Kampagne muss nicht zwangsläufig jemand aus dem DevOps-Team anwesend sein, um das Deployment durchzuführen. Stattdessen kann im Feature-Flag-Service zum gewünschten Termin der Schalter umgelegt werden, um die Funktion zu aktivieren.

Zudem können Feature Flags genutzt werden, um Funktionen auf dem Produktivsystem testweise für einzelne Nutzer freizuschalten. Feature Flags müssen nicht zwangsläufig binär sein, sondern können ähnlich wie bei Canary-Deployments verwendet werden, um Features prozentual auszurollen, aber nicht limitiert auf das vollständige Rollout. Mit einem etwas komplexeren Setup lassen sich Feature Flags auch für einzelne Nutzer oder Gruppen schalten, sodass beispielsweise interne

© Rheinwerk Verlag, Bonn 2024

Nutzer bereits auf ein Feature zugreifen können, während externe Besucher es noch nicht sehen – und das auf Produktivsystemen.

Das heißt natürlich nicht zwangsläufig, dass alle Änderungen ohne Qualitätssicherung auf die Produktivsysteme deployt werden sollten, wo sie hinter einem Feature Flag versteckt werden. Die bisherigen Schritte mit früheren Tests in der CI-Pipeline, dem Deployment auf Review-Umgebungen, auf das ein Code-Review folgt, sollten auch weiterhin passieren. Feature Flags sollten also nicht als *Quality Gate* missbraucht werden, sondern »nur« für mehr Flexibilität beim Ausrollen neuer Features sorgen und helfen, die Auswirkungen von Fehlern möglichst stark zu verringern.

Feature Flags deaktivieren

Wenn die Feature Flags für alle Nutzer aktiviert wurden und nicht mehr gebraucht werden, sollten sie auch zügig aus dem Code entfernt werden. Denn durch den Einsatz von Feature Flags existieren im Quellcode meistens zwei oder sogar mehr Implementierungen eines Features. Alles, was nicht mehr gebraucht wird, sollte im nächsten Deployment entfernt werden. Es muss also immer ein Aufräumen stattfinden, um die Qualität der Software hoch zu halten und Nebenwirkungen auszuschließen.

Es gibt verschiedene Arten, mit Feature Flags zu arbeiten. Eine Möglichkeit ist es, ein Flag in Konfigurationsdateien zu hinterlegen, womit einzelne Funktionen aktiviert und deaktiviert werden können. Dieses Feature Flag wäre dann für das ganze Deployment aktiv. Das Aktivieren eines solchen Feature Flags erfordert dann allerdings ein weiteres Deployment und dauert vergleichsweise länger. Der Einsatzzweck ist hier vorwiegend das technische Aktivieren von diversen Funktionen. Diese einfache Variante ist allerdings nicht sonderlich komfortabel. Außerdem ist diese Möglichkeit eher nur für die vollständige Instanz der Anwendung gedacht und ist damit nicht sehr flexibel.

Eine Alternative ist daher ein entsprechender Feature-Flag-Service. Die Anwendung muss dann über eine API mit dem Feature-Flag-Service kommunizieren, um die Daten für jedes Flag abzurufen. Dafür existieren häufig Client-Librarys für die gängigen Programmiersprachen, damit die Abfrage komfortabel erfolgen kann.

Ein kommerzieller Anbieter ist *LaunchDarkly* (*https://launchdarkly.com/*), und in der Open-Source-Welt werden Sie bei *Unleash* (*https://www.getunleash.io/*) fündig. An Unleash ist besonders hervorzuheben, dass Sie fein konfigurieren können, welcher Benutzer oder welche Benutzergruppe ein Feature aktiviert bekommen soll.

Auch GitLab, das einen Feature-Flag-Service mitbringt, nutzt im Backend Unleash. So können Sie die Berechtigungsstruktur von GitLab weiterverwenden, allerdings fehlen einige Unleash-Features.

Persönliches Exemplar für Thomas Bartholomäus

Die Gefahren der Automatisierung

Knight Capital, eine Investmentfirma, hat durch die fehlerhafte Konfiguration von Feature Flags Hunderte Millionen Dollar verloren. Dort wurde ein Deployment auf acht Servern ausgerollt, das allerdings nur auf sieben Servern erfolgreich war. Der Name des Feature Flags wurde allerdings doppelt verwendet: einmal auf dem neuen Release und einmal auf dem alten Release.

Das alte Release war noch auf einem Server aktiv, weshalb beim Aktivieren des Feature Flags sieben von acht Servern ganz normal operierten. Der achte Server verhielt sich dann allerdings unerwartet komplett anders, was bei einer High-Frequency-Trading-Plattform fatal ist, da etliche Verkäufe durchgeführt wurden, die so hätten nicht passieren dürfen. Da das Deployment selbst ebenfalls manuell ausgerollt worden war und das Monitoring versagte, fiel der Fehler erst auf, als ziemlich viel Geld verpulvert worden war.

8.5 Deployment-Ziele – Wohin mit dem Deployment?

Wo landet das Deployment, wenn das cross-funktionale Entwicklungsteam das fertige Artefakt gebaut hat? In diesem Abschnitt gibt es eine grundlegende Einführung in die Continuous Delivery.

8.5.1 Deployments mit Kubernetes orchestrieren

Wenn Sie Ihre Deployments so organisiert haben, dass Sie Ihre Software zuverlässig und schnell ausliefern können, haben Sie schon sehr viel erreicht. Die Königsdisziplin, die daran anschließt, ist die *Orchestrierung* der Deployments, die heutzutage eigentlich immer mithilfe von Containern stattfindet.

Dabei geht es darum, dass alle Dienste wie in einem Orchester harmonisch zusammenspielen. Oder, um ein Bild zu bemühen, das etwas näher am Value Stream der IT ist: Stellen Sie sich einen Software-Container (siehe Abschnitt 8.3.3) als einen Pappkarton vor, der mit Ihrer Software gefüllt wird. Wenn man damit ein wenig Übung hat, ist das Verpacken keine große Aufgabe mehr. Die Orchestrierung ist dann die komplette Warenwirtschaft eines Unternehmens, die dafür sorgen muss, dass immer genug, aber nie zu viele gefüllte Pappschachteln genau zum richtigen Zeitpunkt am richtigen Ort sind.

Wenn Sie schon einmal die Logistik einer großen Firma gesehen haben, wissen Sie, wie komplex und anspruchsvoll diese Tätigkeit sein kann. Dies ist bei Software-Containern leider genauso. Das Tool, das uns dabei hilft, also quasi der Dirigent des Orchesters, ist Kubernetes.

© Rheinwerk Verlag, Bonn 2024

Kubernetes ist nicht nur im DevOps-Kontext ein großes Thema. In der Einleitung schrieb ich ja bereits: »Immer wieder habe ich gesehen, wie Firmen und deren Mitarbeiter nach Kubernetes rufen, ohne sich mit den dazugehörigen Fragen beschäftigt zu haben. Bevor Sie sich also aus dem Bauch heraus für einzelne Tools entscheiden, sollten Sie die Konzepte der einzelnen Tools und Techniken verstanden haben.«

Damit Sie das Ganze verstehen, muss ich jetzt ein wenig ausholen und Ihnen die Ideen und Konzepte etwas ausführlicher erklären. Insbesondere geht es darum, wie man Anwendungen deployt, skaliert, aktualisiert und vielleicht auch mal ein Rollback durchführt, denn das ist die Stärke von Kubernetes.

Ein kurzer Blick auf die Historie

Kubernetes wurde initial im Jahr 2014 veröffentlicht. Hauptentwickler war damals Google. Das proprietäre Tool *borg* von Google diente als Vorbild, da Google schon lange containerisierte Anwendungen für seine Dienste laufen ließ. Kubernetes wird allerdings schon lange unabhängig von Google weiterentwickelt, auch wenn die Grundideen und Konzepte ihre Google-Vergangenheit nicht verleugnen können.

Was ist eigentlich Kubernetes?

Kubernetes ist ein ganzes System zum Verwalten von Container-Anwendungen mit sehr vielen Funktionen und unglaublicher Komplexität. In wenigen Sätzen kann man Kubernetes kaum beschreiben, aber versuchen wir es einmal ganz grundsätzlich: Kubernetes ist eine quelloffene Software, mit der containerisierte Anwendungen in einem Cluster orchestriert und skaliert werden. Prinzipiell ist Kubernetes also dafür da, Anwendungen zu betreiben, die in Containern laufen – Sie können es sich wie ein Betriebssystem vorstellen, das aber gänzlich unabhängig von der Hardware ist.

So weit, so langweilig. Spannend (und anspruchsvoll) ist, welche technischen Möglichkeiten uns Kubernetes im Bereich CI/CD gibt.

Kubernetes und Komplexität

Um es gleich vorwegzunehmen: Kubernetes hat eine sehr steile Lernkurve und muss mit recht großem Aufwand betreut werden. Auch wenn die Vorteile unbestritten sind, müssen Sie sich zunächst fragen, ob Sie die Features von Kubernetes sinnvoll nutzen können.

Kubernetes lässt sich keinesfalls »nebenbei« erlernen und erfordert grundlegende Änderungen an der Software-Architektur und der Infrastruktur. Wenn Sie nicht genau sagen können, welche Probleme Sie wie mit Kubernetes lösen wollen, sollten Sie erst einen Schritt zurück treten und prüfen, ob der Aufwand gerechtfertigt ist. Denn eine »Kubernetisierung« der eigenen Umgebung zum Selbstzweck sorgt nur für mehr Arbeit und Ärger.

Mit Kubernetes lassen sich containerisierte Anwendungen *relativ* einfach und standardisiert deployen und betreiben. Software-Container sind zwar auch ohne ein solches automatisiertes Deployment eine gute Idee; ihre Vorteile spielen sie aber erst dann so richtig aus, wenn sie richtig verwaltet werden.

Dazu gehört zum Beispiel, dass Kubernetes mit der korrekten Konfiguration der Anwendung automatisch mehr Container hochfahren kann, um hohe Last zu verteilen. Wenn die Last sinkt, kann es dann auch wieder herunterskalieren. Stellen Sie sich vor, dass *schick-gekleidet.de* keinen Webshop betreiben würde, sondern ein traditionelles Kaufhaus: Kubernetes würde dafür sorgen, dass je nach Andrang mehr Kassen aufgebaut, besetzt und eröffnet würden. Die Kunden würden direkt an die neuen Kassen geleitet werden, sodass keine Wartezeiten entstehen und alle zügig bedient werden. Und wenn der Ansturm vorbei ist, würde es die Betriebsfläche wieder verkleinern und die Anzahl der Kassen reduzieren.

Diese Deployment-Features erlauben Tricks, die in traditionellen Anwendungen nicht oder nur mit großen Schmerzen umsetzbar sind. Wenn Sie beispielsweise ein riskantes Deployment planen und Fehler nicht ganz ausschließen können, hilft Ihnen Kubernetes mit einem automatischen Rollback, falls etwas schiefgeht – die Techniken, die in Abschnitt 8.4 vorgestellt wurden, lassen sich eigentlich erst mit Kubernetes so richtig umsetzen. Und auch selbst-heilende Mechanismen sind sowohl auf Container-Ebene als auch auf Node-Ebene enthalten. Wenn ein Node ausfällt, dann sorgt Kubernetes dafür, dass die Container auf den anderen Nodes verteilt und gestartet werden. Hochverfügbarkeit lässt sich so gut umsetzen, da der Ausfall von einzelnen Bereichen automatisch ausgeglichen wird.

Daneben gibt noch viel mehr Funktionen; halten Sie sich immer vor Augen, dass Kubernetes eigentlich von Google für die Verwaltung seiner Rechenzentren entwickelt wurde. Falls Ihre IT-Infrastruktur kleiner ist als die von Google, werden Sie wahrscheinlich mit vielen Teilen von Kubernetes nie in Berührung kommen. Schauen wir uns für den Einstieg lieber an, wie Sie Kubernetes in einer Testumgebung aufsetzen können, um selbst erste Erkundungsschritte damit zu unternehmen.

Entscheidungshilfe: Kubernetes oder nicht?

Wenn Sie dieses Kapitel lesen, kann der Eindruck entstehen, dass für die technische Implementierung von DevOps-Prinzipien und CI/CD kein Weg an Kubernetes vorbeiführt. Das ist natürlich nicht richtig.

Richtig ist aber, dass viele Dinge bei Deployments außerhalb von Kubernetes selbst gebaut werden müssen. Während die Deployments mit Kubernetes einem gewissen Standard folgen, kann das bei Eigenbaulösungen eher wild aussehen: Viele Aspekte und Tools müssen miteinander verknüpft werden: Konfigurationsmanagement, Provisionierung der Umgebung, Deployment-Script, Integration in das Monitoring und noch vieles mehr.

© Rheinwerk Verlag, Bonn 2024

Um Java-Projekte zu deployen, wird vielleicht »einfach« `mvn deploy` in der Pipeline ausgeführt. Das mag zwar funktionieren, bildet aber auch nur einen kleinen Aspekt ab, denn die Ziel-VM muss schließlich auch existieren, und sie muss zuvor provisioniert werden, etwa mit Terraform. Anschließend ist das Konfigurationsmanagement gefragt, damit überhaupt die Laufzeitumgebung vorliegt.

Sind diese Einzelschritte in Ihrem Projekt so komplex, dass es öfter zu Fehlern und Downtimes kommt? Ist der Weg zum Deployment so umständlich, dass neue Mitarbeiter erst Monate brauchen, bis sie den Workflow verstanden haben und ihn selbst beherrschen? Brauchen Sie die Skalierungs- und Deployment-Features von Kubernetes? Dann sollten Sie über einen Umbau Ihrer Anwendung und Infrastruktur nachdenken.

Ansonsten gilt: YAGNI – *You ain't gonna need it*. Nicht jeder Aspekt, den ich hier im Buch behandle, muss zwangsläufig umgesetzt werden.

Kubernetes aufsetzen

Kubernetes lässt sich zwar in der offenen Grundform nutzen, für den produktiven Einsatz bieten sich aber eher kommerzielle Distributionen an, die Ihnen Arbeit abnehmen und Support leisten. Zudem müssen Sie entscheiden, ob Sie Kubernetes im eigenen Rechenzentrum oder in der Public Cloud betreiben wollen.

Dokumentation

Kubernetes ist ausführlich dokumentiert. Unter *https://kubernetes.io/docs/home/* finden Sie sowohl Tutorials für den Einstieg als auch tiefergehende Informationen. Auch Profis können unmöglich alle Optionen und Details rund um Kubernetes auswendig wissen; die Doku sollte also stets konsultiert werden.

Die große Hürde ist, dass man die richtigen Infos für die Version finden muss, die man gerade einsetzt. Kubernetes entwickelt sich sehr schnell weiter, Features werden im Monatstakt eingeführt, geändert, umbenannt und wieder abgekündigt. Auch wenn die Grundkonzepte inzwischen stabil sind, ist in den Details noch immer sehr viel Bewegung.

Kubernetes in Eigenregie

Wenn Sie nun Kubernetes auf selbst verwalteter Infrastruktur installieren möchten, gibt es verschiedene Möglichkeiten. Wie auch bei Linux gibt es für Kubernetes eigene Distributionen, die die vielen Funktionen von Kubernetes bündeln und einfach nutzbar machen. Besonders bei Updates hilft es, wenn diese über eine Distribution organisiert werden, aber auch die Installation und das Management der Kubernetes-Cluster werden einfacher. In der Praxis sieht man daher kaum Cluster, die mit Kubernetes-Hausmitteln angelegt wurden, meist wird auf kommerzielle Distributionen gesetzt.

Persönliches Exemplar für Thomas Bartholomäus

Vanilla Kubernetes

Für die »pure« Open-Source-Variante von Kubernetes hat sich der Begriff *Vanilla Kubernetes* eingebürgert. Es ist der Standard, ohne einen besonderen Geschmack. Schokostreusel oder Karamellsauce können Sie natürlich auch selbst ergänzen, die Distributionen bringen solche Zusatzfeatures bereits mit.

Wenn Sie Ihre Cluster nur mit Kubernetes-Hausmitteln verwalten wollen, müssen Sie mit dem Tool `kubeadm` arbeiten. Das ist nicht einfach, besonders wenn man nicht viel Erfahrung im Umgang mit Clustern hat – das gilt im Grunde für alles rund um Kubernetes. Die Distributionen bieten eigene Administrationswerkzeuge und oft auch Weboberflächen, die Ihnen die Arbeit ein bisschen einfacher machen.

Die wichtigste Kubernetes-Distribution ist *OpenShift* von Red Hat. OpenShift ist eine kommerzielle Lösung, die mit eigenem Support von Red Hat angeboten wird. Es basiert zwar auf Kubernetes, setzt aber einige Dinge anders um. Während bei Standard-Kubernetes das Kommandozeilentool `kubectl` verwendet wird, nutzen Sie unter OpenShift das Werkzeug `oc`.

OpenShift bringt eine Menge Tools mit, die Sie in einem eigenständigen Kubernetes-Cluster selbst installieren und verwalten müssen. So ist eine Container-Registry bereits dabei, genauso wie ein Web-Interface. OpenShift unterscheidet sich bei der Benutzer- und Rollenverwaltung (*Role Based Access Control* (RBAC)) und bei der Netzwerkverwaltung deutlich vom Standard-Kubernetes. Wer beispielsweise das Routen-Konzept von OpenShift gewohnt ist, wird Probleme haben, sich wieder mit dem Netzwerk-Stack von Kubernetes anzufreunden.

Eine andere kommerzielle Alternative ist *Tanzu* von VMware, das ein eigenes Tooling für die Installation und Verwaltung von Kubernetes-Clustern bietet. Besonderer Fokus liegt auf der Integration in andere VMware-Angebote: Wenn Sie bereits vSphere für die Verwaltung von virtuellen Maschinen einsetzen, kann Tanzu eine Alternative zu OpenShift sein.

Die dritte wichtige Distribution ist *Rancher*, das von den Nürnberger Linux-Spezialisten SUSE entwickelt wird, und es gibt noch zahlreiche weitere kommerzielle Lösungen von anderen Anbietern. Eine gute Übersicht finden Sie unter *https://nubenetes.com/matrix-table*.

Für kleine Installationen oder für Lernzwecke ist *k3s* (*https://k3s.io/*) eine gute schlanke Alternative, die im Vergleich zu Vanilla Kubernetes weniger Ressourcen benötigt.

Kubernetes in der Cloud

Prinzipiell spricht nichts dagegen, diese Kubernetes-Distributionen einfach auf einem angemieteten Server in der Cloud zu nutzen. Wenn man allerdings die Verwal-

© Rheinwerk Verlag, Bonn 2024

tung der Hardware auslagern will, möchte man meistens auch direkt das ganze Drumherum abgeben. Dazu bieten alle großen Cloud-Dienste eigene Kubernetes-Distributionen an:

- Amazon Elastic Kubernetes Service (EKS)
- Microsoft Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)

Diesen Weg nennt man *Managed Kubernetes*, da die eigentliche Verwaltung von Kubernetes an die Cloud-Dienste abgegeben wird. Sie greifen über Kommandozeilen-Tools oder die Weboberfläche darauf zu, legen Container und Dienste an und planen Ihre Infrastruktur:

- Aus wie vielen Compute-Nodes soll der Cluster bestehen?
- Was passiert im Fehlerfall?
- usw.

Auch das ist noch komplex genug und erfordert gute Planung und Zeit, um sich einzuarbeiten.

Die letzte Variante sind vollständig verwaltete Kubernetes-Cluster. GKE bietet etwa einen Autopilot-Modus, in dem Cluster mitsamt der darunter liegenden Compute-Infrastruktur komplett von Google verwaltet werden, sodass man sich nicht selbst darum kümmern muss und sich voll auf die Nutzung des Clusters konzentrieren kann. Bei On-Premises-Systemen wäre so ein Service kaum denkbar. Wenn etwa hohe Last herrscht, werden mit dem Autopilot nicht nur die Pods der Anwendungen hochskaliert, sondern es können sogar automatisch neue Nodes hinzugefügt werden. Wenn diese zusätzlichen Nodes nicht mehr benötigt werden, können sie einfach wieder automatisch heruntergefahren werden.

Das ist unheimlich praktisch für alle, die die Anwendungen »nur« entwickeln und betreiben wollen, denen der darunter liegende Stack jedoch ziemlich egal ist. Wichtig ist nur, dass es funktioniert und dass die Automatisierung keine unnötige Arbeit verrichten muss.

Damit sind Sie natürlich weniger flexibel, und in der Abrechnung werden Sie auch einen Unterschied feststellen – die Cloud-Dienste lassen sich den Service natürlich gut bezahlen. Sie müssen genau abwägen und kalkulieren, ob Do-It-Yourself oder All-inclusive für Ihre Anforderungen besser geeignet ist. Beim ersten hier vorgestellten Modell werden in der Regel die Compute-Nodes abgerechnet, beim letzteren Modell eher die Anzahl der laufenden Pods, also die Container. Wie Ihre Software aufgebaut ist und was Sie mit Kubernetes vorhaben, hat also einen direkten Einfluss auf die Kosten.

Persönliches Exemplar für Thomas Bartholomäus

Wer betreibt die Kubernetes-Cluster?

Der Betrieb eines produktiv genutzten Kubernetes-Clusters ist eine anspruchsvolle Aufgabe, auch wenn Distributionen genutzt werden, die zusätzliche Tools bieten und Support liefern. Es ist unrealistisch, dass Entwicklerteams das nebenbei leisten können, und auch für die Ops-Teams, die für den stabilen Betrieb zuständig sind, ist ein Kubernetes-Cluster eine große Herausforderung. Daher wird der Betrieb des Kubernetes-Clusters meist an ein eigenes Team ausgelagert, das hauptsächlich aus Spezialisten für den Betrieb besteht – allerdings angereichert mit Leuten, die die Anwendung und ihren (Container-)Bau gut kennen.

Wenn Ihr Team bzw. Ihre Firma zu klein ist, um ein eigenes Rechenzentrum mit einem Kubernetes-Cluster selbst zu verwalten, sind die Angebote der Cloud-Dienstleister eine gute, wenn auch ziemlich teure Alternative. Unterschätzen Sie auf keinen Fall den Administrationsaufwand, den Kubernetes mitbringt. Mit einem *Managed Service* in der Cloud lässt sich viel Arbeit im Betrieb reduzieren, aber auch dann brauchen Sie und Ihr Team ein Grundverständnis der Komponenten, die dort arbeiten.

Komponenten

Wenn man sich die Komponenten anschaut, aus denen Kubernetes besteht, dann gibt es zwei Blickwinkel: einmal die Infrastruktur des Clusters selbst und dann die Komponenten, die für den Betrieb der Anwendungen zuständig sind.

Control Plane und Node-Komponenten

Die Infrastruktur von Kubernetes besteht – abstrakt betrachtet – aus zwei Teilen: aus der *Control Plane* und aus den *Worker-Nodes*, die von ihr verwaltet werden. (Ein Controller-Node kann gleichzeitig auch Worker sein, auf dem auch Nutzlast ausgeführt wird. Das ist bei kleineren Clustern üblich, aber natürlich ein Kompromiss.)

Die Komponenten der Control Plane laufen auf *Controller-Nodes* und sind für den Kubernetes-Cluster selbst notwendig. Dort laufen Dienste wie `etcd`, `kube-apiserver`, `kube-schedule`, `kube-controller-manager` und noch ein paar Komponenten mehr.

Jeder Node, der reguläre Workload bereitstellt, benötigt mindestens das Kubelet und Kube-Proxy. Das *Kubelet* ist der Agent, der den Node auf dem `kube-apiserver` registriert und ihn mit dem Cluster bekannt macht; es ist grundsätzlich dafür zuständig, dass die Container in einem Pod tatsächlich laufen.

Das *Kube-Proxy* abstrahiert den Netzwerk-Zugriff und sorgt dafür, dass die Nodes mit den Controllern und untereinander kommunizieren können. Zusätzlich wird auf jedem Node eine *Container-Runtime* benötigt, denn irgendwie müssen die Container ja ausgeführt und gestartet werden.

© Rheinwerk Verlag, Bonn 2024

Container, Pods, Services, Ingress und was es sonst so alles gibt

Der Kubernetes-Welt bringt eine Vielzahl von Begriffen und Bezeichnungen mit, die für das Verständnis wichtig sind, aber leider nicht selbsterklärend und oftmals uneindeutig sind und sich auch noch ständig ändern. Versuchen wir es aber einmal, indem wir uns den Weg anschauen, den der Traffic von der Außenwelt bis tief in den Cluster nimmt (siehe Abbildung 8.11). Damit sollte klarer sein, was für typische Kubernetes-Objekte es gibt und wofür diese gut sind – und Sie können sich natürlich vorstellen, dass ich dabei einige Punkte ziemlich vereinfache.

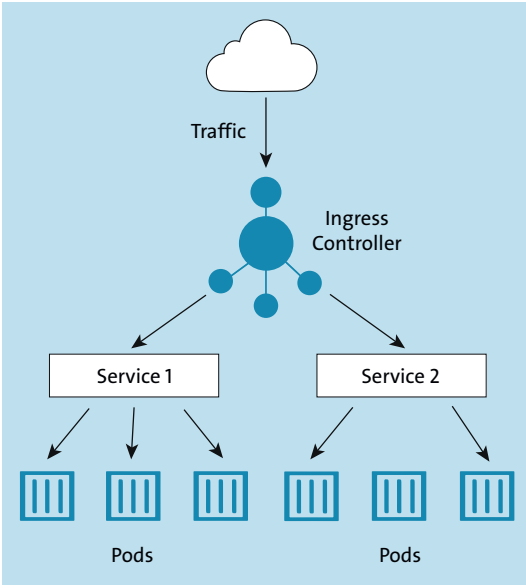

Abbildung 8.11 Dieses Ablaufdiagramm zeigt stark vereinfacht, wohin der Traffic im Kubernetes-Cluster fließt.

Wenn ein Nutzer eine URL aufruft, dann landet der Aufruf zunächst beim Loadbalancer oder einem Reverse-Proxy. *Traefik* (*https://traefik.io/traefik/*) ist ein verbreiteter Reverse-Proxy und Loadbalancer im Kubernetes-Umfeld.

Damit Traefik allerdings weiß, auf welche Route es lauschen soll und wohin der Traffic geleitet werden soll, muss ein *Ingress-Objekt* angelegt werden. In diesem Ingress-Objekt wird festgelegt, über welche URL und welchen Pfad der Traffic zu welchem Port von welchem Service weitergeleitet werden soll.

Der *Service* ist innerhalb des Clusters auch für andere Anwendungen erreichbar. In der Definition wird dann fast nur der Service mit dem *Deployment* verknüpft.

Von einem *Deployment* zu sprechen, kann etwas verwirrend sein, da es den Begriff auch außerhalb von Kubernetes für – nun ja – Deployments gibt. In der Spezifikation eines Deployments wird angegeben, welche Container laufen sollen, welcher und wie

viel Storage für die Anwendung benötigt wird und wie viele Instanzen laufen sollen. Wenn besondere Konfigurationen wie beispielsweise Zugriffsrechte benötigt werden, müssen *ConfigMaps* und *Secrets* dort eingebunden werden. ConfigMaps sind Konfigurationsdateien, die in einen Container eingebunden werden. Passwörter und andere Schlüssel werden in Secrets verwaltet.

Es gibt neben Deployments auch noch *ReplicaSets*, *StatefulSets* und *DaemonSets*. Ein ReplicaSet wird zum Beispiel über ein Deployment automatisch erzeugt, um die Anzahl der Replicas zu verwalten. Außerdem können CronJobs für wiederkehrende Aufgaben definiert werden.

Ein *ReplicaSet* erzeugt einen *Pod*. In einem Pod sind ein oder mehrere Container enthalten, die benötigt werden, um die Anwendung zu betreiben.

Und zum Schluss gibt es auch noch das Storage-Thema. Für persistenten Storage können *Persistent Volumes* sowie *Persistent Volume Claims* definiert werden, die von den Containern eingebunden werden.

Anwendungen deployen

Ein wesentlicher Vorteil von Kubernetes ist, dass es eine einheitliche API für Deployments bietet. Während bei Systemen ohne Kubernetes häufig jedes Deployment komplett anders ausschaut, sorgt Kubernetes für eine gewisse Einheitlichkeit. So können alle nachvollziehen, wie die Deployment-Pipeline aussieht und welche Aktionen nötig sind, um sie anzulegen. Gerade wenn man sich in neue Projekte einarbeitet, gewinnt man so einen guten Überblick über die Abläufe.

[!]

Zugangsrechte für Cluster

Da die cross-funktionalen Teams die Anwendung selbst deployen, müssen sie Zugang zu den Clustern bekommen. Sie sollten hier allerdings strikt zwischen den einzelnen Aufgaben unterscheiden, denn für gewöhnlich gibt es Produktiv-Cluster und Cluster für Staging-, Review- und Test-Umgebungen. Auf diese Weise können Sie Visibilität herstellen, da alle Teammitglieder Einblick in die internen Bereiche bekommen.

Der Zugriff auf den Cluster für Produktivsysteme wird hingegen eingeschränkt. Das kann über einen RBAC-Ansatz geschehen; es ist ein großer Vorteil von OpenShift, dass dies dort einfach möglich ist. Außerdem können einzelne Bereiche des Clusters über Namespaces abgegrenzt werden. Mit *Namespaces* lassen sich die verschiedenen Objekte logisch voneinander trennen, was sich ebenfalls auf die Berechtigungsstruktur herunterbrechen lässt und somit die Teams und deren Anwendungen voneinander isoliert.

Änderungen am Produktiv-Cluster müssen selbstverständlich genau überwacht und eingeschränkt werden. Achten Sie aber auch auf die anderen Cluster, besonders

© Rheinwerk Verlag, Bonn 2024

wenn Sie in der Cloud arbeiten: Falsche Konfigurationen können den ganzen Cluster durchdrehen lassen, und durch die automatische Skalierung werden dann viele Ressourcen verschwendet. Das kann sehr schnell sehr teuer werden. Ohne einen Code-Review und ein entsprechendes Change-Mangement sollte niemand Änderungen am Cluster vornehmen dürfen.

(Automatische) Skalierung

In klassischen Systemen ist die Skalierung der Ressourcen ein großes Problem. Oftmals wurde auf eigenes Skripting gesetzt, damit virtuelle Maschinen on-the-fly erzeugt und bei Nichtgebrauch verworfen wurden. Das ging natürlich nur, wenn das Monitoring die entsprechenden Infos direkt liefert, was erfahrungsgemäß eher nicht der Fall war. Stattdessen war Blindflug angesagt.

Kubernetes nimmt Ihnen diese Arbeit ab, denn dafür existiert der *Horizontal Pod Autoscaler*, mit dem ein Deployment oder ein StatefulSet hoch- oder herunterskaliert wird – je nachdem, wie ausgelastet das System ist (siehe Abbildung 8.12).

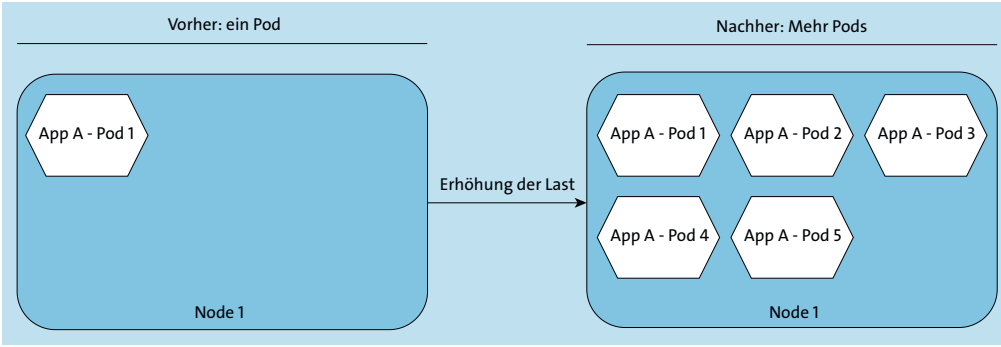

Abbildung 8.12 Bei höherer Last wird die Anzahl der Pods erhöht und später wieder reduziert.

Horizontale Skalierung bedeutet, dass mehrere Kopien der Anwendung gestartet werden, man zieht sie also bildlich gesprochen in die Breite. (Es werden mehr Kassen geöffnet, um beim Kaufhaus-Beispiel zu bleiben.) Damit das sinnvoll funktioniert, muss Ihre Anwendungen auch entsprechend designt und geschrieben worden sein (siehe Abschnitt 9.3.4, »Cloud-native«). Ein bloßes »Lift & Shift« einer monolithischen Anwendung führt nur zu einem zusätzlichen Layer, der den Betrieb und das Deployment komplizierter macht.

Die praktischen Features, die Kubernetes bietet, bringen erst einen Vorteil, wenn man auch gewillt ist, die Architektur entsprechend anzupassen. Und dazu gehört am Ende auch die Implementierung von echtem CI/CD und der DevOps-Prinzipien.

Updates und Rollbacks

Kubernetes kann seine große Stärke bei der Verwaltung von Updates und möglichen Rollbacks ausspielen. Da es den Traffic über Loadbalancer verwaltet, kann (relativ) einfach konfiguriert werden, wie eine neue Version Ihrer Anwendung ausgerollt wird. Einen Überblick über die unterschiedlichen Strategien finden Sie in Abschnitt 8.4.

Wenn es zu Fehlern nach einem Deployment kommt, kann ein *Rollback* ausgeführt werden, damit man direkt wieder zur vorherigen Version springen kann. Das ist meistens schneller und schmerzfreier, als die Pipeline erneut laufen zu lassen. Kubernetes speichert in der `etcd`-Datenbank zumindest zeitweise die Historie der Deployments, über die Sie Fehler rückgängig machen können.

Eng damit verwandt ist das *Self-Healing* des Clusters. Auch hier kann die Konfiguration im Detail sehr komplex werden, denn fast alles lässt sich genauestens einstellen. Die Idee ist grundsätzlich, dass Container automatisch neu gestartet werden, wenn Anwendungen abstürzen oder zu viele Fehler loggen. So kann sichergestellt werden, dass zumindest die Anwendung weiterläuft, auch wenn der Fehler nicht sofort korrigiert wurde.

Labels und Annotations

Kubernetes nutzt Labels und Annotations in den Manifesten, um die Ressourcen zu beschreiben. Gerade Letzteres ist zu Beginn gewöhnungsbedürftig und auch etwas verwirrend, da diese Annotations genutzt werden, um Automatisierungen durchzuführen. Diese Zusammenhänge sind oftmals undurchsichtig und schwer verständlich.

Labels können einfach wie gängige Beschreibungen genutzt werden: Man setzt einfache Key-Value-Werte an die Objekte, die dokumentieren, was dieses Objekt macht, wozu es gehört und ob es über ein anderes Tool verwaltet wird (siehe Listing 8.2).

```
metadata:
  labels:
    app.kubernetes.io/name: redis
    app.kubernetes.io/instance: redis-dev-testing
    app.kubernetes.io/version: "7.2.1"
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: shop
    app.kubernetes.io/managed-by: helm
```

Listing 8.2 Label in Kubernetes

Sie können dort auch den Namen Ihres Teams und einen Ansprechpartner hinterlegen, damit Cluster-Admins bei Fragen wissen, wen sie kontaktieren müssen.

© Rheinwerk Verlag, Bonn 2024

Annotations sind wesentlich spannender, denn sie dienen zum »Einhaken« in andere Tools. Mit solchen Hooks sorgen Sie beispielsweise dafür, dass das Monitoring-System Prometheus weiß, welche Systeme es wie zu überwachen hat. Sie müssen an den Services entsprechende Annotations setzen, auf die Prometheus horcht und durch die es weiß, an welchen Endpunkten es die Daten abholen kann. Auch für das Management von Ingress-Routen können Annotations verwendet werden.

Das Prinzip ist am Ende ziemlich ähnlich: Die Teams haben volle Kontrolle über ihre Anwendung und können sich in weitere Tools des Kubernetes-Ökosystems per Annotations einhängen.

Das Ökosystem

Um Kubernetes hat sich in den letzten Jahren ein ganzes Ökosystem aus Zusatz-Tools gebildet, die fast alle mit Cloud Native zu tun haben. Diese Tools gehören nicht zum Kern von Kubernetes (der ist schon komplex genug) und übernehmen oftmals zentrale Aufgaben beim Cluster-Management.

Paket-Management mit Helm

Helm ist ein Paket-Manager für Kubernetes; Sie brauchen ihn also, um externe Projekte einfach installieren zu können. Viele Anwendungen aus dem Cloud-native-Umfeld kommen mit einem *Helm-Chart* daher, der von den Projekten selbst verwaltet werden kann. Helm kann auch für das Deployment der eigenen Anwendung genutzt werden, indem die Manifest-Dateien, das Kubernetes-Deployment, ConfigMaps, Services, Ingress und so weiter dort gesammelt werden.

Helm ist also eigentlich ein Mechanismus für das Templating von Anwendungs-Deployments. Mit Helm kann man die YAML-Manifeste als wiederverwendbare Vorlagen anlegen, sodass die gleiche Anwendung mehrfach deployt werden kann, nur eben mit einer anderen Konfiguration. Dafür wird immer auf eine *values.yaml*-Datei verwiesen, in der die Konfiguration für die jeweilige Instanz der Anwendung enthalten ist. Das ist besonders praktisch, wenn man in mehrere Environments etwas deployen möchte: neben den eher statischen Produktions- und Staging-Umgebungen vor allem auch die temporären Review-Umgebungen.

Sonstige relevante Tools

Je eingehender Sie sich mit Kubernetes beschäftigen, desto mehr Tools werden Sie finden, die in modernen Infrastrukturen unverzichtbar sind. Dazu gehören Monitoring- und Observability-Werkzeuge wie *Prometheus*, aber auch Service-Meshes wie *Istio* oder *Linkerd* (siehe Kapitel 10.) Bei komplexen Deployments helfen *Argo CD* und *Flux*, die wir uns gleich ansehen, wenn es um GitOps geht.

Persönliches Exemplar für Thomas Bartholomäus

HashiCorp Nomad

Das Projekt *Nomad* der Firma HashiCorp ist eine Alternative zu Kubernetes. Nomad will einfacher zu nutzen und zu verwalten sein – wenn Sie sich in Kubernetes einarbeiten, werden Sie nachvollziehen können, woher dieser Wunsch kommt. Ein Haken an Kubernetes ist nämlich, dass nur containerisierte Anwendungen deployt werden können. Das ist bei Nomad anders: Hier werden sowohl Container als auch nicht containerisierte Anwendungen unterstützt.

Ansonsten sind die Kernaspekte ähnlich: Die Ressourcennutzung soll effizient ablaufen, es sind Selfhealing-Funktionen enthalten und Zero-Downtime-Upgrades sind möglich.

Nomad ist im Vergleich zu Kubernetes ziemlich schlank. Es besteht nur aus einer einzigen `nomad`-Binärdatei, die auf den Servern installiert werden muss und als Agent dient.

Ob Nomad als Orchestrierungs-Tool für Sie eine Alternative sein kann, hängt vor allem davon ab, ob schon Kubernetes-Know-how vorhanden ist. Wenn ja, sollten Sie besser auf Kubernetes setzen, da es das viel größere, verbreitete und etablierte System ist. Auf dem Arbeitsmarkt finden sich einfacher Personen mit Kubernetes-Kenntnissen als mit Nomad-Know-how. Falls Sie sich fortbilden, werden Sie mit Kubernetes-Fachwissen eher punkten als mit Nomad.

Davon abgesehen: Nomad ist für kleinere bis mittelgroße Umgebungen gut geeignet und kommt besser als Kubernetes mit traditionellen Software-Projekten oder einer Mischung aus neuen und alten Projekten zurecht. Wenn nichts dagegenspricht, auf eine Nischenlösung zu setzen, kann Nomad eine sehr sinnvolle Alternative sein.

GitOps

Der Name *GitOps* ist wie »DevOps« eine Zusammensetzung, diesmal aus den Begriffen »Git« und »Ops«. Der Betrieb einer Infrastruktur wird also mit der Versionsverwaltung von Git zusammengebracht.

Je nachdem, wen man fragt, hört man unterschiedliche Definitionen von »GitOps«. Der Begriff selbst ist relativ neu, kommt aus der Kubernetes-Welt und beschreibt die Art und Weise, wie Anwendungen deployt werden und wie der dazugehörige Code definiert und ausgerollt wird. Geprägt (und vermarktet) hat diesen Begriff hauptsächlich die Firma *Weaveworks*. Das Ganze fing 2017 an. Mittlerweile finden sich diverse Projekte und Produkte, die in diese recht neue GitOps-Kategorie fallen.

Damals definierte Weaveworks GitOps über diese vier Prinzipien (*https://www.weave.works/blog/the-history-of-gitops*):

- Der Zielzustand sollte deklarativ definiert werden,
- die Dateien sollten in Git versioniert sein,

© Rheinwerk Verlag, Bonn 2024

- genehmigte Änderungen sollten automatisch angewandt werden und
- Agents sollten sicherstellen, dass der Zielzustand eingehalten wird, und notfalls alarmieren.

Die Idee ist, dass alle Deployments in YAML-Dateien definiert werden, die dann von Kubernetes im Cluster angewandt werden. Dafür sind die Agents zuständig, die ebenfalls im Cluster laufen.

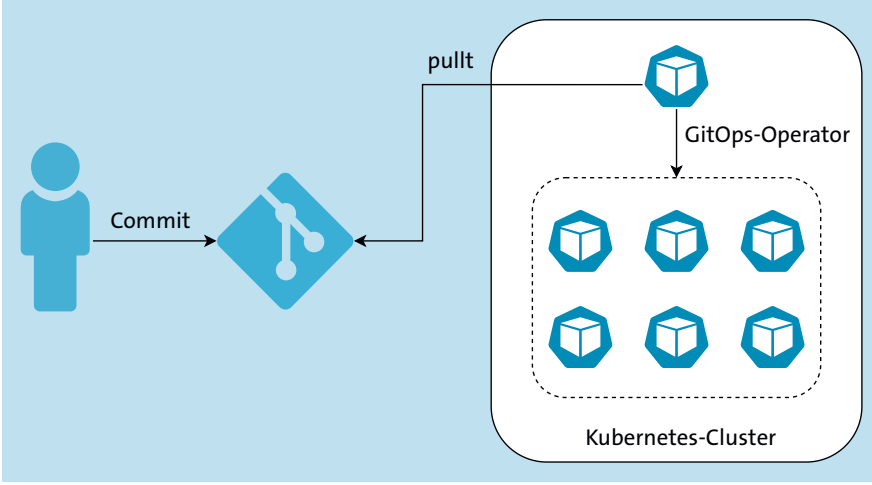

Abbildung 8.13 Das GitOps-Pattern: Der GitOps-Operator arbeitet im Kubernetes-Cluster. Er holt sich die Konfiguration aus einem Git-Repository und wendet sie innerhalb des Clusters an.

Es wird also ein Agent benötigt, der aus einem Repository den deklarativen Code zieht und ihn auf dem System anwendet. Wichtig ist, dass es sich um ein Pull-Prinzip handelt und nicht um ein Push-Prinzip. (Es ist das gleiche Prinzip wie beim Konfigurationsmanagement mit *Puppet*, siehe Abschnitt 9.5. Grundsätzlich ist die Idee also nicht neu.)

Pull vs Push

Die Frage, wie Änderungen übermittelt und verarbeitet werden, hat große Auswirkungen auf das Setup, insbesondere in Hinblick auf Netzwerkfreigaben. Unterschieden wird zwischen Deployments, die *push-based* oder *pull-based* ablaufen.

- Beim push-based Deployment wird von außen das Artefakt in die Zielsysteme kopiert. Das passiert automatisiert in einer Pipeline.
- Wenn pull-based gearbeitet wird, holt sich hingegen das Zielsystem selbst die eigene Konfiguration sowie benötigte Artefakte und führt das Deployment durch.

Einige Jahre später bildete sich eine Arbeitsgruppe namens *OpenGitOps*, die die GitOps-Prinzipien überarbeitete (*https://opengitops.dev*). Die vier Prinzipien lauten nun:[1]

1. **Deklarative Definition**: Auf dem System, das durch GitOps verwaltet wird, muss der gewünschte Zielzustand deklarativ ausgedrückt werden.
2. **Versioniert und unveränderlich:** Der gewünschte Zustand wird auf eine Weise gespeichert, die Unveränderlichkeit und Versionierung erzwingt und eine vollständige Historie aufbewahrt.
3. **Automatisches Pullen:** Die Agents ziehen automatisch die gewünschten Deklarationen aus der Quelle.
4. **Automatischer Abgleich:** Die Agents beobachten kontinuierlich den aktuellen Systemzustand und versuchen, den gewünschten Zustand herzustellen.

Diese vier Prinzipien sind gut und nachvollziehbar. Es gibt allerdings auch andere Interpretationen von GitOps, die sich nicht mit der Definition von Weaveworks oder OpenGitOps decken. GitLab beispielsweise versteht GitOps als Kombination aus Infrastructure as Code, Merge Requests sowie CI/CD.

Es ist also gleichgültig, ob es ein Pull- oder Push-Mechanismus ist, da beide Varianten letztlich die Software ausrollen. Für GitLab sind also sowohl Terraform als auch Ansible GitOps-Tools, obwohl es kein automatisches Pullen und keinen automatischen Abgleich gibt.

Wie auch immer die konkrete technische Implementierung aussieht: Wichtig ist, dass mit Code-Reviews gearbeitet wird und dass die Historie in einem Git-Repository getrackt ist. Denn auf einen wichtigen Punkt bin ich bislang nicht eingegangen: Ein sauber definiertes GitOps-Repository für mehrere Anwendungen deployt alles in einem Rutsch durch, während bei einem klassischen »Push«-Deployment über eine Pipeline für jedes Paket etliche Pipelines neu gestartet werden müssen, um die Deployments zu tätigen. Bei einem größeren Ausfall lässt sich daher schneller und einfacher der Ursprungszustand wiederherstellen.

GitOps hat allerdings auch einige Nachteile, denn so findet eine Trennung zwischen Dev und Ops statt. Hier sollte man also aufpassen, dass man zwar einige Probleme löst, aber nicht wieder anfängt, Silos aufzubauen.

In der Kubernetes-Welt sind zwei GitOps-Tools besonders relevant. Das ist zum einen *Flux* und zum anderen *Argo CD*. Um das Fazit vorwegzunehmen: Mit beiden Tools kommt man ans Ziel.

1 Vielleicht fällt Ihnen auf, dass von Git dabei gar keine Rede mehr ist. Stattdessen können die Definitionen auch in einem S3-Bucket liegen oder es kann sogar Subversion genutzt werden. Ob das eine gute Idee ist, sei mal dahingestellt.

© Rheinwerk Verlag, Bonn 2024

Flux

Flux (*https://fluxcd.io/*) wurde maßgeblich von Weaveworks entwickelt, steht inzwischen jedoch unter der Schirmherrschaft der Cloud Native Computing Foundation. Es ist ein Agent, der innerhalb eines Kubernetes-Clusters läuft und die Konfiguration, die es ausrollen soll, per Pull-Prinzip aus einem Repository zieht.

Für den Einsatz von Flux muss ein Git-Repository entsprechend eines spezifizierten Schemas aufgebaut werden. Es unterstützt reine Kubernetes-Ressourcen in YAML-Dateien und Helm-Charts. Wenn Sie Helm-Charts einsetzen, kann der Rollout komplett vom Entwicklungsteam gesteuert werden.

In einem separaten Repository wird definiert, welche Version ausgerollt werden soll. Hier gibt es verschiedene Möglichkeiten, wie das Deployment ausgeführt werden soll. Eine Möglichkeit ist etwa, dass jede Version einen separaten Commit benötigt, in dem die Versionsnummer hochgezogen wird. Das ist jedoch zusätzliche Arbeit, die vielleicht gar nicht gebraucht wird.

Je nach Umgebung und Anforderung kann es auch sinnvoll sein, dass Minor-Versionen automatisch aktualisiert werden, Major-Versionen jedoch nur nach einem Approval deployt werden dürfen. In dem Fall müsste nur bei einer neuen Major-Version ein neuer Commit erfolgen.

Flux besteht im Großen und Ganzen aus zwei Teilen: aus dem Agent selbst, der im Cluster läuft, sowie aus einem Kommandozeilentool, um den Agent zu verwalten. Eine Weboberfläche gibt es in der freien Variante allerdings nicht, sodass man weiterhin auf Kommandozeilenwerkzeuge wie `kubectl` oder andere Tools zurückgreifen muss.

Argo CD

Auch Argo CD (*https://argoproj.github.io*) ist ein GitOps-Tool, das ein Projekt der Cloud Native Foundation ist. Ähnlich wie Flux ist es ebenfalls ein Agent, der in Kubernetes läuft, und auch Argo CD zieht die Konfiguration per Pull-Prinzip aus einem Repository.

Im Gegensatz zu Flux besitzt Argo CD eine Weboberfläche, in der sich der Aufbau der Cluster visuell darstellen lässt, und auch sonst ist der Funktionsumfang etwas größer. Aber auch der technische Ansatz ist ein wenig anders: Der Agent von Flux muss etwa auf jedem Cluster installiert werden, den Flux betreuen soll. Das sieht bei Argo CD anders aus. Dort gibt es einen zentralen Argo-CD-Server, der die Deployments auf einen oder mehrere Cluster betreut.

Während Flux sehr auf die typischen Admins abzielt (unter anderem, da eine Weboberfläche fehlt), merkt man bei Argo CD, dass der Fokus eindeutig auf dem Developer-Workflow liegt. Zudem kommen dort noch einige Benutzermanagement-Funktionen dazu, die in Flux fehlen.

Persönliches Exemplar für Thomas Bartholomäus

8.5.2 Deployments orchestrieren bei schick-gekleidet.de

Bei *schick-gekleidet.de* wollte man die Deployments wesentlich flexibler gestalten und auch moderne Features wie Rollbacks und Feature Flags nutzen. Entsprechend fiel auch die Wahl auf Kubernetes. Das bedeutete jedoch auch, dass großer Schulungsaufwand nötig war, da sich die Arbeitsweise deutlich veränderte.

Die Infrastruktur musste dazu so verändert werden, dass alles darin automatisch hoch- und herunterskaliert werden konnte – dieses Problem wird uns in Kapitel 9 wieder begegnen. Zunächst wurde die horizontale Skalierung auf Kubernetes-Ebene konfiguriert, sodass bei höherer Last mehr Pods gestartet wurden. Wenn zu viel Last auf dem Cluster lag, wurde ein neuer Worker-Node automatisch provisioniert und dem Cluster hinzugefügt, sodass die Last noch breiter verteilt werden konnte. Techniken wie Canary-Deployments und A/B-Testing konnten dann nach und nach ebenfalls umgesetzt werden.

Das ging nicht von heute auf morgen, sondern es dauerte eine Zeitlang, bis die Architektur des Shops so weit umgebaut war, dass dies technisch möglich war – ganz unabhängig vom Team, das sich ebenfalls umstellen musste.

Reflexion

Die technische Umsetzung von Continuous Delivery ist, wie Sie unschwer erkennen können, eine komplexe Aufgabe.

Insbesondere das Augenmerk auf Kubernetes bei einigen Firmen ist immer wieder spannend. Natürlich ist Kubernetes relativ komplex. Aber spannender ist, sich die Frage zu stellen, ob das Problem denn auch komplex ist. Wenn die Lastverteilung über die Dienste zu jeder Uhrzeit anders ist und auch eine einheitliche Form von Deployments gefahren werden muss, dann ist Kubernetes wiederum eine gar nicht so komplexe Lösung. Diese Verhältnis sollten Sie also beachten.

Wenn Sie hingegen kein komplexes Problem haben, dann ist Kubernetes eher nichts für Sie. Daher auch hier mein Rat: Schauen Sie sich Ihr Problem an und fragen Sie sich, was wirklich relevant ist. Nicht jede Firma braucht A/B-Tests und Canary- oder Blue/Green-Deployments. Nicht jede Software muss stark skalieren. Nur mit den richtigen Fragen kann auch eine angemessene Lösung gefunden werden.

8.6 Fazit

Continuous Delivery mit DevOps-Prinzipien ist eine Wissenschaft für sich. Ordentliches Continuous Delivery umzusetzen ist alles andere als trivial, da viele technische Voraussetzungen erfüllt werden müssen.

© Rheinwerk Verlag, Bonn 2024

Dieses Kapitel sollte Ihnen einen guten Einblick vermittelt haben, welche Herausforderungen und Möglichkeiten Sie bei der technischen Implementierung von Continuous Delivery erwarten können.

Dazu führt meist kein Weg an Kubernetes vorbei, und das dürfte sich in den nächsten Jahren auch nicht ändern. Hier gilt es also abzuwägen: Benötigen Sie wirklich die Funktionen von Kubernetes und können Sie es zuverlässig betreiben und supporten?

Denn eines ist in der Technik wichtig: Auch die besten Tools und Techniken helfen nicht, wenn die Mitarbeitenden keine oder nur wenig Erfahrung haben. Dann ist die Wahrscheinlichkeit hoch, dass man scheitert. Und dabei sollte man stets im Blick behalten, dass es in Zukunft auch anders aussehen kann: Einige erfahrene Leute werden gehen, einige Neulinge werden kommen.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 9
Den Dienst betreiben

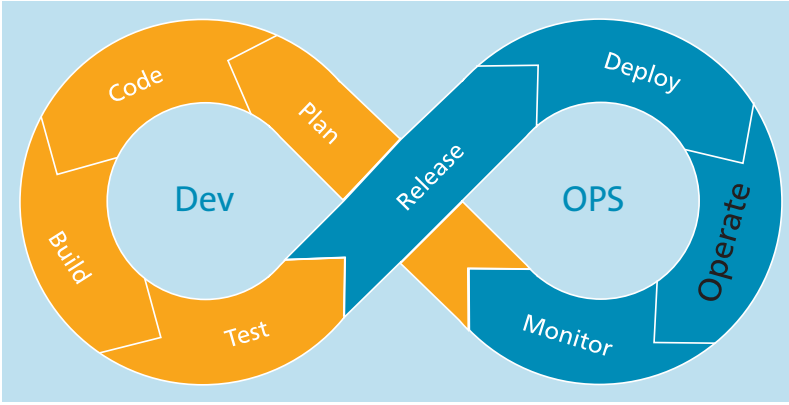

Abbildung 9.1 Operate: Die Software wird in der Produktionsumgebung betrieben und verwaltet.

In Kapitel 8 ging es um das Fertigstellen des Releases und um das Deployment der Anwendung. Dabei haben Sie wahrscheinlich schon erkannt, dass die Grenzen zwischen Release und Deployment zum Betreiben der Anwendung fließend sind.

In diesem Kapitel schauen wir uns den Betrieb der Anwendung an. Im Vergleich zur klassischen Systemadministration haben sich die Aufgabengebiete der reinen Betriebsteams verändert, da ihre Mitglieder zusammen mit den Entwicklern in einem Team sitzen. Schauen wir uns an, wie sich die Jobrolle verändert und welchen praktischen Einfluss die DevOps-Idee auf das Tagesgeschehen hat – sowohl für Entwickler als auch für die Admins.

9.1 Die typischen Probleme beim Betreiben der Dienste

Ich habe bereits erläutert, dass das Betriebsteam enger mit dem bisher reinen Entwicklungsteam zusammenarbeitet, indem die Admins als vollwertige Mitglieder in die entsprechenden Teams integriert werden. Dies führt zu einer verstärkten Verschmelzung von *Dev* und *Ops*. In diesem Kapitel richte ich den Fokus auf die Aufgaben aus Sicht des Betriebsteams, die eng mit denen des Infrastrukturteams verbunden sind, das die benötigte Infrastruktur bereitstellt.

Day 0, Day 1, Day 2

In der DevOps-Welt hat sich die Unterscheidung in Day-0-, Day-1- und Day-2-Operations eingebürgert. *Day 1* ist das Deployment, also der Zeitpunkt, ab dem der Dienst betrieben wird. Alles, was danach zu tun ist, fällt unter Day-2-Aufgaben. Was vor dem Deployment kam, also die Anforderungsanalyse und die eigentliche Entwicklung, sind Teile von *Day 0*.

9.1.1 Langwierige Planungen rund um die Infrastruktur

Da bei *schick-gekleidet.de* immer für sechs Monate im Voraus geplant wurde, hatte dies langfristige und schwerfällige Planungen rund um die Infrastruktur zur Folge. Durch neue Marketingkampagnen und durch die generelle Steigerung der Umsätze musste sehr früh abgeschätzt werden, wie die Server-Infrastruktur in der Zukunft aussehen sollte. Anforderungen mussten rechtzeitig formuliert werden, Server konnten nicht kurzfristig bereitgestellt werden.

Die Abschätzung, wie viel Last zu erwarten war und wie sich die Besuchszahlen des Shops in den nächsten sechs bis zwölf Monaten entwickeln würden, war nicht einfach, denn häufig war die Glaskugel kaputt. Diese wurde nämlich benötigt, weil jedes Team diverse Annahmen machte, aber niemand echte Daten verwendete. Wie die Auslastung genau aussah, änderte sich mit jedem Deployment, sodass die Jahresplanung eher ein Raten war.

Das hatte gleich mehrere Gründe. Das Monitoring der Systeme war genauso aufgesplittert wie die Team-Struktur, daher wurden keine Daten aufbereitet, mit denen das Finanzteam, die Business-Analysten und die Entwickler hätten nachvollziehen können, welche Anforderungen aus dem geplanten Wachstum entstehen würden.

Für die Verwaltung der Hardware ist das Infrastrukturteam von *schick-gekleidet.de* verantwortlich. Die Hardware wird im eigenen Rechenzentrum betrieben. Zu den typischen Aufgaben gehört neben der Beschaffung und dem Einbau der Server auch das Auswechseln von defekter Hardware, also der Tausch von Servern, Teilen der Netzwerk-Infrastruktur oder Geräten der redundanten Stromversorgung.

Das Infrastrukturteam arbeitete eng mit dem Betriebsteam zusammen. Dieses übernahm die Hardware, sobald diese komplett verbaut war, und installierte auf ihr im ersten Schritt das Basis-Betriebssystem.

Im Rahmen der Jahresplanung für die jeweils neuen Versionen wurde immer eine Planung für das Adminteam und das Infrastrukturteam erstellt. Jedes Jahr lief der Support für einen Teil der Hardware aus und musste ausgetauscht werden. Da immer höhere Besuchszahlen erwartet wurden und zudem neue Features mehr Rechenpower verlangten, musste immer neue Hardware beschafft werden.

© Rheinwerk Verlag, Bonn 2024

9.1.2 Hardware-Austausch mit Hindernissen

Das Infrastrukturteam hatte also ein ausreichendes Budget für die Beschaffung neuer Hardware, die im Laufe des Jahres bestellt und eingebaut wurde. Vor dem Einbau der neuen Hardware musste allerdings erst einmal die alte, ausgediente Hardware ausgetauscht werden. Hier war eine enge Zusammenarbeit mit dem Betriebsteam nötig, um den Betrieb des Online-Shops sicherzustellen.

Bei dieser Zusammenarbeit zwischen den beiden Teams kam es immer wieder zu Konflikten, obwohl es auf den ersten Blick so schien, als ob beide Teams eigentlich ziemlich ähnliche Ziele verfolgten: Beide sollten sicherstellen, dass der Online-Shop ordentlich betrieben wird. Für das Infrastrukturteam bedeutete das die Betreuung der Hardware, für das Betriebsteam die Betreuung der Software, bevor dann die Anwendung auf die Server deployt wird.

Eigentlich wäre es einfach gewesen, an einem Strang zu ziehen, wenn die beiden Teams nicht mit unterschiedlichen Support-Zeiträumen zu kämpfen gehabt hätten:

- Das Infrastrukturteam musste mit den Support-Zeiträumen der Hardware arbeiten: Teilweise gab es gar keinen Support mehr, was bei Ausfällen von Hardware dazu führte, dass kein Ersatz mehr verfügbar war.
- Das Betriebsteam wiederum musste die Support-Zeiträume des Betriebssystems im Blick behalten. Mit der Zeit wurden immer mehr Sicherheitslücken bekannt, die nicht korrigiert werden konnten, weil Betriebssysteme eingesetzt wurden, die schon fast die Volljährigkeit erreicht hatten.

Auch war es an der Tagesordnung, dass Updates der Betriebssysteme per Hand eingespielt werden mussten, da kein automatisches Patch-Management vorhanden war; Updates wurden erst installiert, wenn sich ein Admin auf das System eingewählt hatte, beispielsweise um ein Deployment aufzuspielen oder einen Fehler zu untersuchen. Dass das aus sicherheitstechnischen Gründen keine gute Idee ist, sollte hoffentlich allen klar sein.

Die unterschiedlichen Support-Zeiträume führten zu Henne-Ei-Problemen: Das Betriebsteam konnte das Basis-Betriebssystem nur aktualisieren, wenn neue Hardware bereitstand, um den Betrieb des Online-Shops zu gewährleisten. Das Infrastrukturteam konnte wiederum nur neue Server einbauen, wenn vorher alte Server außer Betrieb genommen wurden.

Und das war nicht die einzige Abhängigkeit. Auch zwischen dem Entwicklungsteam und dem Adminteam funkte es: Die genutzte Middleware konnte nur aktualisiert werden, wenn auch das Basis-Betriebssystem aktualisiert wurde. Dafür musste aber auch die Anwendung angepasst werden, die dann hingegen nur auf neueren Maschinen mit neuerem Betriebssystem und der neueren Middleware lief.

Persönliches Exemplar für Thomas Bartholomäus

Falls Sie jetzt denken, dass dieses Szenario ein wenig zu sehr an den Haaren herbeigezogen ist: Genau so etwas habe ich in einem großen Unternehmen so beobachtet. Hohe Mauern und fragwürdige Architekturentscheidungen führten zu einem Deadlock bei wichtigen Aktualisierungen, da komplexe (oder weniger komplexe) Abhängigkeiten nicht aufgelöst wurden. Verschlimmert wurde das Ganze dadurch, dass sich niemand dafür verantwortlich fühlte, dieses Problem effektiv anzugehen.

9.1.3 Unvorteilhafte Serverauslastung

Auch die Verantwortlichen für den Betrieb bei *schick-gekleidet.de* mussten stets die Deadlines der Entwicklungsteams im Auge behalten, damit die Arbeiten an den Servern vor dem nächsten großen Deployment abgeschlossen wurden. Das sorgte stets für viel Stress, denn die wenigen großen Deployments waren komplett unflexibel geplant und daher davon abhängig, dass die Server rechtzeitig bereitstanden. Das bedeutete viel Handarbeit, denn es existierten nur wenige Automatismen, um die Server zu installieren oder gar um den täglichen Betrieb sicherzustellen.

Ein weiteres Problem für das Betriebsteam war die ungleiche Auslastung der Server: Einige Systeme waren für ihre Aufgaben überdimensioniert, andere für die Last viel zu schmal ausgelegt, was auch die Kunden zu spüren bekamen, wenn der Shop mal wieder langsam reagierte. Die Server waren alle zweckgebunden, man konnte am Hostnamen jedes Servers ablesen, wofür er eingerichtet wurde.

Dadurch, dass zu Beginn des Jahres immer schon die komplette Hardware verplant wurde, rannten die Systeme letztlich immer nur den Änderungen hinterher. Immer wieder kam es zu brenzligen Situationen, in denen das ganze System oder wichtige Teile ausfielen. Für die Kunden von *schick-gekleidet.de* bedeutete das, dass sie ihre Bestellungen nicht tätigen konnten, und für die Firma, dass sie keine Einnahmen erzielte.

Teilweise war die erhöhte Last auf Änderungen zurückzuführen, die durch die Weiterentwicklung des Shops entstanden. Für das Betriebsteam war das sehr mühselig, da es von den Veränderungen an der Software immer überrascht wurde. In den Tagen und Wochen nach einem großen Release bedeutete das immer mehr Arbeit sowie viele Einsätze in der Nacht, weil etwas nicht so funktionierte, wie es sollte.

Konflikte zwischen Admins und Entwicklern

Diese ganzen Probleme führten immer zu Frust und zu Finger-Pointing. Der Hauptkritikpunkt des Betriebsteams war stets, dass es zu wenig oder fehlerhafte Informationen zum Betrieb der Anwendung erhalten habe. Da jedes Entwicklungsteam andere Programmiersprachen und Tools verwendete, musste sich das Betriebsteam jedes Mal in die Eigenheiten der Software und der Laufzeitumgebung einarbeiten.

© Rheinwerk Verlag, Bonn 2024

9.1.4 Regelmäßige Ausfälle in der Nacht

Das Schlimmste, was einem Online-Shop passieren kann, ist eine außerplanmäßige Downtime. Wenn der Shop nicht funktioniert, wird kein Umsatz generiert und die Kunden verlieren das Vertrauen in die Marke und die Firma.

Das Betriebsteam bei *schick-gekleidet.de* war also dafür zuständig, dass der Online-Shop immer läuft. Üblich ist dafür ein Bereitschaftsdienst außerhalb der gängigen Bürozeiten. Es war nicht unüblich, dass die Admins kurzfristig in der Nacht geweckt wurden, weil das Monitoring angesprungen war. Häufiges Problem: Ein Dienst auf einem Server lief nicht mehr. Da es sich um ein hochverfügbares Setup handelte, war das zu dem Zeitpunkt erst mal nicht kritisch, da die übrigen Dienste ja noch weiter liefen.

Trotzdem musste sich ein Admin in der Nacht um das Problem kümmern, also den Fehler untersuchen und ihn idealerweise korrigieren. Besonders wichtig dabei war, dass der Fehler nicht auch auf den anderen Servern auftrat und somit den Online-Shop ausfallen ließ.

Oftmals war der Grund ziemlich trivial: Beim genaueren Blick auf den ausgefallenen Dienst fiel auf, dass die Festplatte eines Servers vollgelaufen war. Nachdem man ein wenig Platz auf dem Server geschaffen hatte, konnte der Dienst wieder gestartet werden. Und zum Glück war nur dieser eine Server betroffen, der Admin konnte sich nach einer Weile wieder schlafen legen.

In der nächsten Nacht wurde er erneut geweckt: Schon wieder lief der Dienst nicht. Diesmal kam der Alarm allerdings von einem anderen Server. Schon wieder war die Platte vollgelaufen und wieder half es, ein wenig Platz zu schaffen und den Dienst erneut zu starten.

Der eigentliche Fehler war natürlich, dass nicht untersucht wurde, warum die Platte volllief. Es wurde nur das Nötigste gemacht, damit die Warnungen nicht mehr auftraten; am nächsten Morgen war das eigentliche Problem nicht mehr sichtbar, eine echte Ursachensuche war also schwierig.

Als der Fehler in der nächsten Nacht wieder auftrat, nahm sich endlich jemand Zeit, um dem Problem auf den Grund zu gehen. Dabei wurde festgestellt, dass die Anwendung seit dem letzten Deployment in bestimmten Fällen viel zu viele Daten in ein Logfile schrieb, was zum Volllaufen des Festplattenspeicherplatzes führte. Der Admin war abermals genervt, da die dritte Nacht im Eimer war.

Am nächsten Morgen öffnete er beim Entwicklungsteam ein Ticket mit den Beobachtungen, die er gemacht hatte. Dadurch, dass schnell Platz geschaffen wurde, gab es allerdings nur den »Augenzeugenbericht« und keine Logdateien, die sich das Entwicklerteam anschauen konnte. Also änderte sich nichts, weil man das Problem nicht reproduzieren und verstehen konnte. Zwar war das Entwicklungsteam informiert, allerdings konnte es ohne weitere Informationen nichts dagegen tun.

Persönliches Exemplar für Thomas Bartholomäus

Der Admin wiederum wurde in der nächsten Nacht abermals geweckt und das Spielchen begann von vorne: Ein Fehler trat auf, und die Entwickler bekamen kein zeitnahes, vollständiges Feedback zur Ursachensuche. Das führte letztlich immer nur zu Frust zwischen Betriebsteam und Entwicklungsteam, da immer die anderen schuld waren.

Natürlich hatten beide Teams ihren Anteil daran. Die Problemlösung sollte sein, die Hürden und die Probleme gemeinsam anzugehen.

Reflexion

Wenn Sie sich in Ihrem Unternehmen umschauen, werden Sie einige Probleme von *schick-gekleidet.de* auch bei Ihnen wiederfinden. Je nachdem, welche Rolle Sie bekleiden, ist es wahrscheinlich, dass zwar einige der Probleme existieren, diese für Sie allerdings nicht sichtbar sind.

Das ist insbesondere dann der Fall, wenn Sie aus dem Entwicklungsteam keinen Einblick in den Betrieb der Anwendung haben. Teilweise liegt es auch daran, dass die Anwendung direkt beim Kunden läuft und somit die Hürde hoch ist; teilweise besteht das Problem aber auch, weil es die genannten Silos gibt.

Wenn Sie keinen oder nur wenig Einblick in den Betrieb der Anwendung haben, sprechen Sie doch einfach eine nette Person an, um sich auf kurzem Dienstweg erklären zu lassen, wie die Anwendung läuft. Wichtig ist dabei, dass Fragenstellen und Zuhören notwendig ist.

Das gilt übrigens auch in der anderen Blickrichtung: Wenn Sie für den Betrieb zuständig sind, dann kann ein kurzer Besuch beim Entwicklungsteam helfen, einige Probleme schon frühzeitig aufzudecken, selbst wenn noch getrennte Teams existieren.

9.2 Aufbrechen der stark gekoppelten Infrastruktur-Architektur

In Kapitel 8 haben wir bereits besprochen, wie Deployments schneller und flexibler ablaufen können. Mit diesen Erkenntnissen können Sie sich daran machen, die starre Infrastrukturplanung aufzubrechen und Systeme zu erstellen, die sich flexibler skalieren und auslasten lassen.

Wie immer, wenn es um die Infrastruktur geht, lassen sich grundlegende Probleme nicht von heute auf morgen lösen – denken Sie nur an das Chaos mit der Deutschen Bahn, das uns wahrscheinlich noch Jahre begleiten wird. Wichtig ist auch hier wieder, dass Sie das ganze System betrachten und Entscheidungen davon abhängig machen, ob sie sich in eine größere Strategie einordnen und das Gesamtergebnis verbessern. Es ist kontraproduktiv, wenn Sie zwar wunderbar flexible Cloud-Infrastrukturen anmieten, diese dann aber unverändert mit genau den gleichen Anwendungen

© Rheinwerk Verlag, Bonn 2024

betreiben wie zuvor. Eine schwerfällige Infrastruktur, die in die Cloud kopiert wird – man spricht von *Lift and Shift* –, sorgt nur für höhere Kosten.

Automatisierung ist nämlich schön und gut, aber wenn man schlechte Prozesse und eine schlechte Kultur einfach nur durchautomatisiert, hat man nichts gewonnen. Treten Sie einen Schritt zurück, schauen Sie die eigentlichen Probleme an, und finden Sie eine holistische Lösung.

Bei *schick-gekleidet.de* war die sehr starke Kopplung von Anwendung, Betriebssystem und Server das Hauptübel. Unterschiedliche Support-Zeiträume, uneinheitliche Konfigurationen und die allgemein schwerfällige Planung machten zügige Upgrades der Infrastruktur sehr mühselig. Dies wurde bereits dadurch verbessert, dass durch schnellere und kürzere Deployments die Lead Time deutlich gesenkt wurde, sodass flexibler auf Änderungen reagiert werden konnte. Das Betriebsteam konnte so einfacher Downtimes für Hardware-Upgrades planen. Die Orchestrierung der Deployments mit Kubernetes (siehe Abschnitt 8.5.1) vereinfachte die Skalierung der Anwendung, und die Anpassungen der Software-Architektur hin zu Microservices sorgten für weniger Last-Probleme.

9.2.1 Cattle not Pets

Eines der Probleme ist, dass die Infrastruktur sehr starr und wenig beweglich war. Die Server hatten Namen, die von der Software abhingen, die auf ihnen deployt wurde. Die Server wurden über ihren langen Support-Zeitraum mit den Software-Updates versorgt – zumindest mehr oder weniger.

Im DevOps-Umfeld wird hingegen nach dem Prinzip *Cattle not Pets* gearbeitet. Die Grundidee dahinter ist, dass die Server nicht mehr wie Haustiere behandelt werden. Jeder Server (also jedes Haustier) hatte zuvor einen eigenen Namen und wurde nach bestem Gewissen gepflegt – so wie man sein Haustier regelmäßig füttert und streichelt.

DevOps macht damit Schluss. Die Server bekommen keinen festen Namen und werden nicht verhätschelt. Stattdessen wird die Infrastruktur wie Nutzvieh behandelt: In einer Rinderherde hat keines der Tiere einen Namen, und wenn eines der Viecher krank ist, wird es erschossen statt aufgepäppelt. Genauso sollte mit der Infrastruktur umgegangen werden: Alles ist vergänglich. Wenn etwas kaputt ist, wird es einfach neu erstellt und ersetzt, anstatt es aufwendig zu reparieren.

Cattle vs. Pets: Vegane Edition

Wer lieber eine tierfreundliche Alternative sucht, der kann sich das gleiche Szenario mit alten Eichen und einem Maisfeld vorstellen: Eichen werden mehrere Hundert oder gar Tausend Jahre alt und sollten geschützt werden. Maisfelder wiederum werden jedes Jahr neu angepflanzt und am Ende der Saison abgeerntet. In der Infra-

struktur gilt das Gleiche: Anstatt im Garten mit der Hunderte Jahre alten Eiche zu sprechen und sie zu pflegen, gehen Sie lieber wie der Bauer mit seinem Maisfeld vor, das stetig neu gepflanzt wird.

Das klingt zwar etwas hart – ist es für das Nutzvieh auch –, aber das Prinzip bringt viele Vorteile: Sowohl die Infrastruktur als auch die Anwendungen sollten verwerfbar sein und sich leicht rekonstruieren lassen – und nicht einzigartig und unersetzlich sein. Durch die vollständige Automatisierung im Code ist es möglich, dass man den Stack effizient verwalten kann.

Wenn etwas nicht funktioniert, kann es problemlos neu ausgerollt werden, was die Austauschbarkeit und Wartbarkeit deutlich verbessert. Wenn fehlerhafte Konfigurationen ausgeliefert wurden, vielleicht sogar auf Hunderten Servern, lässt sich eine Korrektur viel schneller neu ausrollen, als jeden einzelnen Server zu reparieren.

Server wegwerfen

Wenn hier vom »Wegwerfen« von Servern gesprochen wird, ist natürlich nicht damit gemeint, dass die physischen Server weggeworfen werden. Das wäre ökologisch und ökonomisch weder sonderlich sinnvoll, noch ließe sich das sinnvoll automatisieren. Das Wort »wegwerfen« soll natürlich abstrakt verstanden werden: Das Wegwerfen von virtuellen Maschinen oder Containern ist die Regel. Aber auch ein automatisches Plätten und Neuinstallieren von physischen Maschinen ist für bestimmte Use-Cases möglich und üblich.

All das waren Änderungen, die die Arbeit des Betriebsteams bei *schick-gekleidet.de* deutlich einfacher machten, obwohl es nicht direkt etwas mit dem Betrieb der Anwendung zu tun hatte. Erst der Blick aufs große Ganze und die engere Zusammenarbeit führten dazu, dass diese Probleme gelöst werden konnten. Denn auch, wenn man dem Betrieb einen Haufen neuer Server mit neuen Betriebssystemlizenzen spendiert hätte: Die eigentlichen Probleme konnten nur gemeinsam gelöst werden.

Schauen wir uns dazu die Details an. Zunächst geht es um die Frage, wo die Anwendung überhaupt gehostet werden sollte. Zur Auswahl stehen »echte« Server im eigenen Rechenzentrum – man spricht von *Bare Metal* –, eine virtualisierte Infrastruktur sowie moderne Container-Systeme.

Die Wahl der Infrastruktur hängt von vielen Faktoren ab

Ich möchte mich hier auf eine Betrachtung der technischen Umsetzung beschränken. In der Praxis ist es jedoch so, dass die Entscheidung nicht allein davon abhängt, was technisch machbar ist, sondern dass oftmals andere Faktoren ausschlaggebend sind. Wenn Sie beispielsweise mit Daten arbeiten, die besonders sensibel sind, dann

© Rheinwerk Verlag, Bonn 2024

kann eine eigene Serverlandschaft die einzig mögliche Lösung sein. Auch wenn Sie spezielle Hardware nutzen müssen, werden Cloud-Angebote keine Alternative sein.

So kann es auch Sinn ergeben, dass man abhängig von den unterschiedlichen Fähigkeiten im Team eine passende Lösung findet. Denn am Ende ist es – ganz unabhängig von der konkreten Lösung – wichtig, dass das Team das vollständige System versteht, um bei Problemen schnell handeln zu können.

9.2.2 Infrastruktur abstrahieren

Abbildung 9.2 gibt Ihnen einen ersten Überblick: Die Ausgangssituation ist immer der Server aus echtem Blech. Im einfachsten Fall wird auf ihm direkt das Betriebssystem installiert. Wenn virtuelle Maschinen genutzt werden, wird die Hardware abstrahiert, sodass mehrere virtualisierte Systeme genutzt werden können. Im Beispiel ganz rechts wurden die VMs noch zusätzlich mit mehreren Containern ausgestattet. In diesem Fall ist die Kopplung von der Hardware quasi aufgehoben.

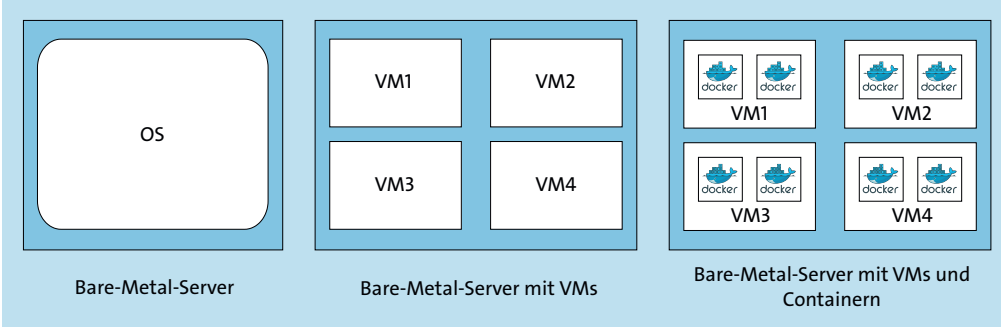

Abbildung 9.2 Hardware, virtuelle Maschinen und Container

Provisionierung und Support der Hosts

Bare-Metal-Server sind stets die Ausgangslage aller Infrastrukturen. Sie sind besonders für die eigenen Rechenzentren relevant, allerdings bieten auch viele Cloud-Provider nichtvirtualisierte Server an, die Sie mieten und nutzen können. Ihr Metall steht dann nicht bei Ihnen im Serverschrank, sondern in einem Rechenzentrum, das jemand anderes verwaltet.

Werfen wir zunächst einen Blick auf die On-Premises-Rechenzentren: Wenn dort mit Bare-Metal-Servern hantiert wird, muss sehr viel selbst gemacht werden. Von der Bestellung über den Einbau der Server bis zu allen Aufgaben rund um das Netzwerk und Storage sind Sie für alles verantwortlich. Denken Sie auch an eine unterbrechungsfreie Stromversorgung und an eine gute Klimatisierung, damit es den Rechnern im Sommer nicht zu heiß wird.

Sie sollten zumindest die Bestückung mit dem Basis-Betriebssystem automatisieren. Dafür müssen einige Vorbedingungen erfüllt werden, die netzwerkseitig geschehen. Sind diese erfüllt, provisionieren Tools wie *Foreman* (*https://www.theforeman.org*) die Bare-Metal-Server mit dem Betriebssystem.

Wenn Sie sich die Arbeit mit der Hardware und dem eigenen Rechenzentrum sparen wollen, gleichzeitig aber die größtmögliche Kontrolle behalten möchten, ist der Einsatz von Bare-Metal-Servern in der Cloud oder in einer *Colocation* (Colo) denkbar. Wesentlicher Vorteil ist, dass ein Tool-Stack für die Provisionierung bereitgestellt wird, den man mit der API und den eingesetzten Tools provisionieren kann. Das ist deutlich wartungsärmer und flexibler, als es bei typischen On-Premises-Rechenzentren der Fall ist.

Warum ist Bare Metal als Depoyment-Ziel für Anwendungen auch im Cloud-Zeitalter noch sinnvoll? Zwar lassen sich virtuelle Maschinen oder Container wesentlich flexibler einsetzen, für einige Use-Cases braucht man allerdings »echte« Hardware. Das ist besonders dann der Fall, wenn ein eigener Virtualisierungsstack benötigt wird oder wenn spezielle Hardware gebraucht wird, die sich nicht in der Cloud mieten lässt. Außerdem müssen Sie sich so keine Gedanken um *Noisy Neighbours* machen, also um virtuelle Maschinen, die auf dem gleichen Host laufen und den Betrieb stören könnten.

Mit Bare-Metal-Servern haben Sie die höchste Flexibilität beim Deployment und die volle Kontrolle über die Performance des Servers. Das erkaufen Sie sich mit erhöhtem Verwaltungsaufwand der Server und mit wenig Portabilität, schließlich ist da alles relativ fix verdrahtet.

Virtuelle Maschinen

Virtuelle Maschinen sind der erste Schritt, um die enge Bindung der Deployments mit der Hardware aufzubrechen. Anstatt dass die Anwendung direkt »auf dem Blech« läuft, wird sie in einem virtualisierten Rechner installiert. Diese Maschinen werden von einem sogenannten *Hypervisor* verwaltet, der die Abstraktionsschicht darstellt. Abbildung 9.3 zeigt das Grundprinzip.

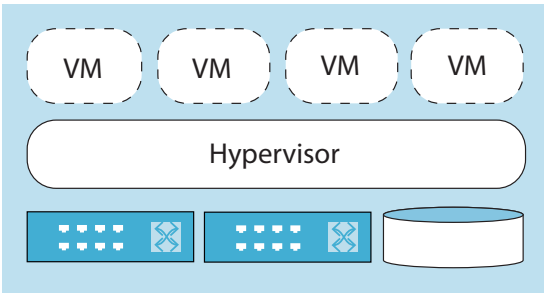

Abbildung 9.3 Ein Hypervisor verwaltet die VMs und abstrahiert sie von der Hardware.

© Rheinwerk Verlag, Bonn 2024

Durch den Einsatz der VMs werden die Workloads wesentlich portabler. Wenn ein darunter liegender Server ausgetauscht werden muss, müssen die auf ihm laufenden VMs nur heruntergefahren, umgezogen und woanders wieder hochgefahren werden. Je flexibler die Anwendung das unterstützt, desto besser und einfacher ist das Management.

Durch die Virtualisierung kann zumindest schon einmal das Problem gelöst werden, dass sich die Server langweilen. Anwendungen, die in VMs stecken, lassen sich zusammen auf einem Host betreiben, was Kosten spart. Das hat auch direkten Einfluss auf die Provisionierung der Infrastruktur und auf das Deployment der Apps, weil so Automatisierungen möglich werden, die starres Blech nicht zulässt.

Einer der wesentlichen Nachteile von virtuellen Maschinen ist, dass man sich weiterhin um das Betriebssystems kümmern muss. Das Hochfahren von VMs und das Installieren und Verwalten der benötigten Pakete dauert seine Zeit. Das lässt sich zwar automatisieren, doch addieren sich die Minuten. Stellen Sie sich vor, Sie wollen nur einen kleinen Serverdienst anbieten; in einer virtuellen Maschine muss trotzdem ein ganzes Betriebssystem vorgehalten und Ihr Dienst ganz traditionell installiert werden. An diesem Punkt setzen Container an, um die es bereits in Abschnitt 8.3.3 ging und deren Betrieb wir uns in Abschnitt 9.2.3 noch genauer anschauen. Zunächst gibt es aber noch einige Infrastruktur-Fragen zu klären, beispielsweise welches Betriebssystem überhaupt im Betrieb der Anwendungen eingesetzt werden sollte.

Das Betriebssystem

Die Frage, auf welches Betriebssystem man setzen sollte, stellt sich bei Servern eigentlich kaum. Linux hat hier eine marktbeherrschende Stellung, Windows spielt bis auf kleine Nischen – etwa bei .NET – praktisch keine Rolle. Im DevOps-Bereich sind Linux-Kenntnisse daher sehr wichtig.

Linux wird in vielen unterschiedlichen Distributionen angeboten. Bei klassischen Servern spielt die Frage nach dem Support-Zeitraum die größte Rolle bei der Entscheidung. Im Firmenumfeld sind Ubuntu, SUSE und Red Hat und seine Derivate relevant, da sie zuverlässigen Support über berechenbare Zeiträume vom Hersteller anbieten. Falls Sie mit Einschränkungen bei diesem Support leben können, bieten sich auch andere Distributionen wie Debian an, da die technischen Unterschiede gering sind.

Für das Host-System machen Sie grundsätzlich mit keiner der erwähnten Distributionen etwas falsch. Hauptsache ist, dass Sie die Sicherheitsupdates während der Lebenszeit zeitnah einspielen (das heißt idealerweise am gleichen oder nächsten Tag, spätestens innerhalb einer Woche) und dass Sie im Blick haben, wann die Versionen nicht mehr unterstützt werden.

Persönliches Exemplar für Thomas Bartholomäus

Nun haben aber virtuelle Maschinen und auch Container ebenfalls ein Betriebssystem. Und Sie wissen ja schon, dass gerade Container sehr flüchtig sind und eigentlich nie lange benutzt werden. Bei jedem Run der CI/CD-Pipeline werden neue Container gebaut und deployt, die dann nach kurzer Zeit wieder ersetzt werden. Spielt es dann noch eine Rolle, ob das Betriebssystem darin 7 oder 10 Jahre lang unterstützt wird, wenn jeder Container doch nur einige Tage oder Wochen lebt?

Daher verliert die Zeit, während der ein Betriebssystem mit Aktualisierungen versorgt wird, in der DevOps-Welt an Bedeutung. Man geht sogar in die entgegengesetzte Richtung: Das Betriebssystem für Container soll *immutable* sein, also unveränderlich. Wenn es dort Probleme gibt, wird nicht repariert, sondern weggeworfen und neugebaut – Rindviecher statt Haustiere.

Infrastruktur für App-Deployments

Wenn nun also die Hardware bereitsteht und das Betriebssystem installiert ist, kann die Anwendung mitsamt der benötigten Laufzeitumgebung deployt werden.

Die Herausforderung mit Blick auf DevOps ist, dass es durch die starre und recht wenig flexible Struktur kaum oder nur mit hohem Aufwand möglich ist, dass die Anwendung skalieren kann. Einer der wesentlichen Nachteile bei Bare Metal ist auch, dass die vollständige Hardware genutzt wird, selbst wenn nur viel weniger Leistung benötigt wird.

Viele Anwendungen laufen nicht mit voller Leistung, nutzen also die verfügbaren Ressourcen nicht aus, vor allem, da die Bare-Metal-Server meist recht groß dimensioniert sind. Diese Server kosten natürlich einiges an Geld, man holt aber nur sehr wenig daraus. Das Setup ist nicht flexibel, da die Apps direkt und fest den Servern zugewiesen sind. Ein Arbeiten nach dem »Cattle not Pets«-Prinzip ist dann nicht üblich und nicht möglich.

Dadurch, dass die Apps fest mit den Servern verdrahtet sind, sind die verschiedenen Schichten aus Hardware, Basis-Betriebssystem, Laufzeitumgebung und Anwendung sehr stark miteinander gekoppelt, was zu den Problemen aus Abschnitt 9.1 führt. Ein einfacher Austausch eines Servers ist nicht möglich, da die Anwendung zunächst aufwendig umgezogen werden muss.

Und da man bei DevOps auf der technischen Ebene keine feste Verdrahtung haben will, sondern flexibel bleiben möchte, werden heutzutage kaum noch Bare-Metal-Server betrieben.

Das wird zwar in einer Public Cloud ein wenig aufgeweicht, weil quasi beliebig viele Rechner bereitstehen, die man nutzen kann. Aber wenn man eine Public Cloud fast genauso nutzt wie ein eigenes Rechenzentrum, dann hat man kaum etwas gewonnen, sondern wiederholt die gleichen strukturellen Probleme nur an einem anderen Ort.

© Rheinwerk Verlag, Bonn 2024

9.2.3 Container für schnellere Deployments

Software-Container haben in den letzten Jahren die IT-Welt im Sturm erobert, da sie es (relativ) einfach machen, eine Anwendung mit allen Abhängigkeiten in ein standardisiertes Format zu packen. Dieses Paket lässt sich einfach ausrollen und läuft dann isoliert vom Hostsystem und anderen Containern. Eine solche Entkopplung vom Hostsystem ist mit Bare-Metal-Servern unmöglich zu erreichen; mit virtuellen Maschinen ginge es schon eher, aber auch dort fehlen die einfachen Möglichkeiten, um solche Images zu bauen. (Wenn in einer Cloud-Umgebung mit virtuellen Maschinen gearbeitet wird, dann sollte die Infrastruktur auch so aufgebaut werden, dass die VMs einfach neu gebaut werden können, quasi wie überdimensionierte Container.)

Sie haben Container bereits in Abschnitt 8.3.3 kennengelernt, als es um den Build und die Paketierung der Anwendung ging. Der Build ist allerdings nur eine Herausforderung, die sich durch die Container-Technologien vereinfacht hat. Auch beim Betrieb der Container sind heutzutage Kunststücke möglich, die mit dem direkten Server-Hosting oder virtuellen Maschinen nur schwer umsetzbar sind. Dabei geht es um die sogenannte *Orchestrierung*, also um das Zusammenspiel vieler unterschiedlicher Container-Dienste. Das wichtigste Tool dafür ist *Kubernetes* (siehe Abschnitt 8.5.1), aber auch mit den »Hausmitteln« *Docker Compose* und *Docker Swarm* können Sie Deployments planen, wenn Sie nicht die Features brauchen, die Ihnen Kubernetes bietet.

Der wichtigste Vorteil von Container-Systemen im Betrieb ist die Skalierung. Je nach Bedarf können mal mehr, mal weniger Container einer Anwendung gestartet werden, die dafür sorgen, dass Lastspitzen abgefangen werden und Hochverfügbarkeit gegeben ist. Sie erhalten so eine bessere Kontrolle über die Ressourcennutzung der einzelnen Anwendungen und können auch die Nutzung von CPU- und Arbeitsspeicher-Ressourcen mit einfachen Mitteln so limitieren, dass ein Container keine anderen laufenden Dienste auf demselben Host behindert. Das ist mit einem traditionellen Hosting nicht so einfach möglich; allerdings muss auch bei Container-Systemen die Hardware mitspielen und eine Skalierung zulassen – auch Container können die Grenzen der Physik nicht aushebeln. Darum geht es ausführlicher in Abschnitt 9.3, der vom Einsatz in der Cloud handelt.

Ein weiterer Kernbestandteil von Containern sind die Health-Checks, die sowohl beim Bauen als auch beim Deployment der Container definiert werden können. Die Idee dahinter ist, dass der Container selbstständig prüfen kann, ob die Anwendung, die darin enthalten ist, auch korrekt läuft. Bei einem Webservice kann ein einfacher Health-Check sein, ob der HTTP-Statuscode 200 zurückgegeben wird. Ist das nicht der Fall, dann wird ein Fehler geworfen. Das Resultat wird dann genutzt, um den Container neu zu starten, in der Hoffnung, dass es dann funktioniert. Und solange der Health-Check nicht funktionsfähig ist, werden nur die gesunden Container mit Traffic bedient. Idealerweise bekommt der Endnutzer von einzelnen fehlerhaften Containern also nichts mit. Das ist etwa dann hilfreich, wenn ein Container-Host Probleme hat, aber die anderen normal weiterfunktionieren.

Persönliches Exemplar für Thomas Bartholomäus

Health-Checks sind eine Möglichkeit, um mit Fehlern umzugehen. Je nach Ausführungsort der Container gibt es noch etliche weitere, etwa dass ein automatisches Rollback auf ein älteres Container-Images ausgelöst wird, wenn Health-Checks fehlschlagen.

Diese Funktionen der Orchestrierungsplattformen wie Kubernetes lassen sich allerdings erst dann nutzen, wenn das Fundament ausreichend flexibel ist, um wirklich gut und schnell mit Containern zu jonglieren. Diese Flexibilität findet man in der Cloud, in der die Anwendung vollkommen von der Hardware entkoppelt wird.

State of DevOps Report

Im *State of DevOps Report* von 2022 war eines deutlich: Container-Orchestrierer waren das beliebteste Deployment-Ziel. Darunter fallen Tools wie Kubernetes und Docker. In der befragten Gruppe gaben 54 % an, dass dies das primäre Deployment-Ziel ist. VMs in der Cloud kamen auf 42 %.

9.3 Cloud-Computing

Die berühmt-berüchtigte Cloud ist häufig eine Lösung, um die Automatisierung der Infrastruktur voranzutreiben. Was die Cloud ist, wofür sie gut ist und wofür sie nicht gut ist, wird kontrovers diskutiert.

In diesem Abschnitt geht es darum, welche Rolle die Cloud im Rahmen von DevOps spielt und welche Verbesserungen sich damit umsetzen lassen, die vorher so nicht möglich waren, um die Zusammenarbeit der Teams zu stärken.

Die Kosten der Cloud

Ausdrücklich kein großes Thema werden die Kosten sein, da sich diese Frage nur schwierig abschätzen lässt und von vielen Faktoren abhängt– es kommt ganz darauf an, wie die Cloud genutzt wird. Wenn man die Cloud nur als Kopie des eigenen Rechenzentrums versteht, verschenkt man die Vorteile einer solchen Lösung und sorgt für ziemlich hohe Kosten. Ausnahmen bestätigen hierbei die Regel.

9.3.1 Was ist die Cloud?

Aber was ist nun »die Cloud« überhaupt? Der Begriff *Cloud-Computing* wird inflationär verwendet, und ob es eine gute Idee ist, die eigene Infrastruktur auf fremde Server zu verlagern, kann kontrovers diskutiert werden. Die Antwort auf die Frage, ob sich die Cloud *lohnt*, lautet daher ganz abstrakt: Es kommt darauf an.

© Rheinwerk Verlag, Bonn 2024

Um diese Frage für Ihre konkrete Situation zu beantworten, müssen Sie verstehen, was mit der Cloud überhaupt gemeint ist. Nützlich ist dazu immer noch die Definition des NIST (*National Institute of Standards and Technolgy*) aus dem Jahre 2011, die Sie hier finden:

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Grundsätzlich wird die Definition in drei Aspekte unterteilt:

- essenzielle Charakteristiken
- Service-Modelle
- Deployment-Modelle

In der Definition sind fünf essenzielle Charakteristika gelistet:

- Das erste Charakteristikum ist der *On-Demand Self-Service*. Die Idee dahinter ist, dass der Konsument einseitig sowohl Rechen- als auch Netzwerkleistung selbstständig beantragen kann, ohne dass auf der Provider-Seite etwas händisch erledigt werden muss.
- Der zweite Aspekt ist ein *breiter Netzwerkzugriff*. So sollen Zugriffe über das reguläre Netzwerk verfügbar sein, sodass es von den normalen Endgeräten genutzt werden kann.
- Zentral ist der Gedanke des *Resource Poolings*. Die Idee hierbei ist, dass mehrere Kunden die gleichen Ressourcen nutzen können, auch wenn diese auf der gleichen Hardware laufen. Die Ressourcen sind dabei voneinander abgeschottet. Je nach Kundenbedarf können diese Ressourcen entsprechend auch verworfen oder verschoben werden. Dem Kunden ist dabei nicht klar, wo genau seine Software läuft – von groben Kennzahlen wie dem Land, dem Ort und dem Rechenzentrum abgesehen.
- Ein weiterer wichtiger Punkt ist die *Rapid Elasticity*. Darunter versteht sich, dass sich die Ressourcen schnell und in einigen Fällen auch automatisch sowohl hoch- als auch herunterskalieren lassen, je nachdem, wie der Bedarf gerade da ist. Das fühlt sich in vielen Fällen *unlimitiert* an und sollte jederzeit möglich sein.
- Der letzte Aspekt ist der *Measured Service*. Darunter versteht man, dass die Nutzung und der Verbrauch von Ressourcen jeweils gemessen werden und dass daraus auch Berichte erzeugt werden. Diese können für Abrechnungen genutzt werden.

9.3.2 Cloud-Modelle

Wo läuft die Cloud eigentlich? Häufig wird Cloud-Computing mit der *Public Cloud* gleichgesetzt. Das sind die Angebote großer Dienstleister wie Amazon, Microsoft oder Google, bei denen jeder Kunde werden kann. Abgerechnet wird nach verbrauchter Leistung oder Zeit. Das ist allerdings nur eines von mehreren Modellen.

Persönliches Exemplar für Thomas Bartholomäus

Einige Organisationen gehen den Weg der *Multi-Cloud*, um *Vendor Lock-in* zu vermeiden und Ausfällen einer einzelnen Cloud vorzubeugen. Diese Idee ist zwar gut, in der Praxis jedoch sehr aufwendig und teuer. Die Daten liegen in verschiedenen Rechenzentren verschiedener Cloud-Provider, und die unterschiedlichen APIs erfordern zusätzliche Entwicklungsarbeit. Zudem ist das Verschieben von Daten zwischen Rechenzentren zeitaufwendig, mit hohen Latenzen und Kosten verbunden.

Neben der Public Cloud gibt es auch noch die *Private Cloud*. Diese läuft im eigenen Rechenzentrum und es werden die zuvor genannten Charakteristika erfüllt. Die Kunden sind dann entsprechend einzelne Teams oder Abteilungen innerhalb der Organisation. Technisch betrachtet ist es eigentlich unerheblich, ob man in einer Public oder einer Private Cloud arbeitet. Es kommt nur darauf an, dass man die Automatisierung der Cloud nutzen kann, unabhängig davon, ob sie intern bereitgestellt oder von extern eingekauft wird. Wenn es um den Datenschutz und die Kosten geht, unterscheiden sich die beiden Angebote jedoch deutlich.

Eher seltener findet man die *Community Cloud*, bei er es nicht nur einen Betreiber gibt, sondern mehrere, die gemeinsam einen bestimmten Zweck verfolgen und dafür die Cloud bereitstellen.

Zu guter Letzt existiert dann noch das Modell der *Hybrid Cloud*, bei dem die Private Cloud mit einer Public Cloud kombiniert wird. So kann zwar in der Regel die Rechenleistung von der Private Cloud ausreichend sein, aber wenn dann doch mehr Leistung gebraucht wird, dann werden Ressourcen der Public Cloud beansprucht.

State of DevOps Report

Laut des jüngsten *State of DevOps Report* wächst die Cloud-Nutzung stetig, während die Option »keine Cloud« weiter sinkt. Mit »Cloud-Nutzung« ist nicht nur die Nutzung der Public Cloud gemeint, sondern auch die Nutzung einer Hybrid, Multi bzw. Private Cloud.

In der Zukunft wird es in der IT nur noch wenige Inseln geben, die nicht in irgendeiner Form Ressourcen in einem fremden Rechenzentrum nutzen. Die Grundkonzepte sollten Sie daher kennen, auch wenn Sie nicht gleichzeitig alle Ihre Daten und Anwendungen in Richtung Microsoft oder Amazon schicken wollen.

9.3.3 Service-Modelle

Beim Cloud-Computing wird unterschieden zwischen

- Infrastructure as a Service (kurz: IaaS),
- Platform as a Service (kurz: PaaS) und
- Software as a Service (kurz: SaaS).

© Rheinwerk Verlag, Bonn 2024

Da wir an dieser Stelle aus Sicht der Infrastruktur auf das Thema Cloud-Computing schauen, liegt der Fokus hier auf *Infrastructure as a Service* sowie auf *Platform as a Service*.

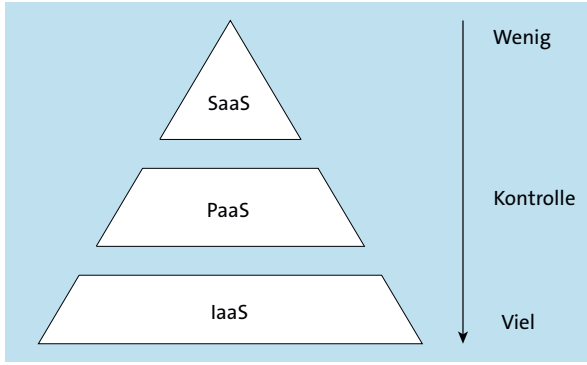

Abbildung 9.4 Die drei Arten »Infrastructure as a Service«, »Platform as a Service« und »Software as a Service« bauen aufeinander auf.

Infrastructure as a Service

Infrastructure as a Service bedeutet, dass die IT-Infrastruktur wie Server, Netzwerk und Storage dynamisch bereitgestellt wird. Unter DevOps-Gesichtspunkten ist es spannend, dass die Teams durch IaaS selbst die Infrastruktur administrieren können.

Damit wird es möglich, dass die cross-funktionalen Teams nahezu den vollständigen Stack – von der Infrastruktur bis zur Anwendung – selbst verwalten können. Das betrifft nicht nur die Produktivumgebung, sondern auch die schnelle und unkomplizierte Bereitstellung von Staging- und Review-Umgebungen, die in Abschnitt 8.2.2 schon thematisiert wurden.

Der Kerngedanke ist, dass die Teams möglichst autonom agieren können. Das geht am besten, wenn man auf eine vollständige Automatisierung setzt, indem alles als Code definiert wird.

Die Infrastruktur sollte also, so weit es geht, vollständig über Code verwaltet werden, inklusive der Nutzung von Reviews, wenn Änderungen an der Infrastruktur vonnöten sind. Der Entwicklungs-Workflow aus Kapitel 5 wird genutzt, um die Infrastruktur zu verwalten (siehe Abschnitt 9.5, »Konfigurationsmanagement: Everything as Code«). Die starke Kopplung der einzelnen »Schichten« wird abgebaut; die Infrastruktur, das Betriebssystem, die Middleware, die Anwendung usw. sind nur noch lose gekoppelt.

Damit das funktioniert, müssen Sie sich den ersten der »Three Ways« immer wieder vor Augen führen: Das Team muss das ganze System verstehen.

Persönliches Exemplar für Thomas Bartholomäus

IaaS umsetzen

Bei Infrastructure as a Service (IaaS) dominieren die drei großen amerikanischen Unternehmen die Branche: *Amazon Web Services* (AWS), *Microsoft Azure* und die *Google Cloud Platform* (GCP). Zu den bekanntesten deutschen Cloud-Anbietern gehören beispielsweise *Hetzner* und *IONOS*. Diese sind jedoch deutlich kleiner und haben eine geringere weltweite Präsenz.

Aus technischer Sicht ist es weitgehend unerheblich, welchen Anbieter Sie wählen, denn in erster Linie geht es um grundlegende Konzepte und Ideen. Auf einer höheren Ebene sind die verschiedenen Cloud-Anbieter ziemlich austauschbar.

In der Regel bieten alle diese Anbieter auf der Infrastrukturebene die Möglichkeit an, virtuelle Maschinen in verschiedenen Konfigurationen zu starten, einschließlich Netzwerkkonfigurationen und Freigaben zwischen Systemen.

Wichtiger ist es, sich zu überlegen, wie man die bereitgestellten Features effizient nutzen kann, um die DevOps-Prinzipien praktisch umzusetzen. Ein Schlüsselaspekt ist, wie viel Verantwortung die Teams selbst übernehmen wollen, sollen und können. Mit IaaS wird der gesamte Hardware-Betrieb ausgelagert, aber die Verwaltung und Wartung der virtuellen Infrastruktur erfordert dennoch Arbeit. Hier kommen Tools aus dem Bereich *Infrastructure as Code* sowie Konfigurationsmanagement ins Spiel, die wir uns in Abschnitt 9.5 anschauen. Dabei spielt die Automatisierung eine entscheidende Rolle, um die Vorteile effektiv nutzen zu können.

Wenn die Cloud-Angebote richtig genutzt werden, sorgen sie für mehr Flexibilität und hohe Verfügbarkeit des Service. Das geht aber nicht ohne ein Team, das sie betreut und über die entsprechenden Kenntnisse verfügt. Die Cloud-Nutzung macht also Spezialisten für den Betrieb keineswegs obsolet, sondern nimmt ihnen – idealerweise – Aufgaben rund um die Verwaltung ab.

Beispielsweise wird kein Management der Hardware nötig sein. Somit werden andere Fachkenntnisse erforderlich sein; es ist wichtiger, dass man Automatisierungsskripte schreiben kann, die mit den Cloud-APIs interagieren, als dass man besonders geschickt mit dem Schraubenzieher ist.

Was die Skalierung angeht: Nahezu alle Cloud-Provider bewerben ihre Dienste damit, dass sie dynamisch hoch- und herunterskalieren können. Aber: Brauchen Sie für Ihre Anwendung überhaupt eine (automatische) Skalierung der darunterliegenden Infrastruktur? Oft ist das sinnvoll, nicht selten sieht man aber, dass eine viel zu komplexe und teure Infrastruktur aufgebaut wird, obwohl die Auslastung minimal ist.

Platform as a Service

Die Alternative – oder auch Ergänzung – zu IaaS ist *Platform as a Service* (PaaS). In diesem Fall muss das Team, das für eine Anwendung zuständig ist, die Infrastruktur nicht selbst provisionieren und verwalten. Die Teammitglieder müssen sich also

© Rheinwerk Verlag, Bonn 2024

nicht mehr um die Hardware, das Betriebssystem und die Middleware kümmern. Ob die Hardware und der Stack von einem internen Team oder von einem externen Anbieter verwaltet werden, ist letztendlich unerheblich. Diese Aufgabe übernimmt in einer Private Cloud das Plattformteam, bei der Public Cloud liegt die Verantwortung beim Dienstleister.

Da diese Aufgaben entfallen, kann man sich ganz auf die Entwicklung und Bereitstellung der Anwendung konzentrieren. Die Kopplung zwischen Anwendung und Plattform ist zwar noch vorhanden, aber die Abhängigkeit von Plattform und Infrastruktur wird wegabstrahiert. Es spielt also keine Rolle mehr, wo die Anwendungen tatsächlich laufen, solange die Teams ihre Ressourcen automatisiert erstellen und nutzen können.

Welche *Managed Services* genau angeboten werden und wie die Dienste genau heißen, unterscheidet sich natürlich überall ein wenig; die Basics wird man aber überall finden. Zum Beispiel bietet *AWS Elastic Beanstalk* viele verschiedene Programmiersprachen wie Java, .NET, Python und mehr. Hinzu kommen praktische Funktionen wie die automatische Skalierung, wodurch bei erhöhter Last keine manuelle Arbeit erforderlich ist und gleichzeitig eine hohe Verfügbarkeit gewährleistet wird. Darüber hinaus werden Funktionen wie Blue/Green-Deployments unterstützt.

Persistierung von Daten

Anwendungen müssen häufig Daten persistieren, und es gibt verschiedene Arten von Storage-Systemen, die jeweils unterschiedliche Einsatzzwecke haben. Die wichtigsten sind Block Storage und Object Storage:

- *Block Storage* teilt den Speicher in Blöcke von Bytes auf, die in das Betriebssystem eingebunden werden, ähnlich wie eine physische Festplatte.
- *Object Storage* hingegen ermöglicht es Anwendungen, Daten über eine API abzulegen, ohne die gesamte Dateisystemgröße reservieren zu müssen. Dies ist kostengünstiger, aber langsamer.

Ein Beispiel für Object Storage ist AWS S3 (*Simple Storage Service*), bei dem AWS sich um die skalierbare Speicherung der Daten kümmert. Die API von AWS S3 wird auch von anderen Storage-Lösungen implementiert, sodass Anwendungen, die mit einer S3-API kommunizieren können, Daten auch außerhalb von AWS S3 verwalten können. Eine Open-Source-Alternative mit einer S3-kompatiblen API ist *MinIO*.

Ein weiterer wichtiger Aspekt sind Datenbanken, die von verschiedenen Cloud-Anbietern als Managed Services für verschiedene Dialekte wie MySQL, PostgreSQL oder NoSQL-Datenbanken wie MongoDB angeboten werden. Das Einrichten hochverfügbarer Datenbank-Cluster erfordert viel Arbeit, ebenso wie das Konfigurieren von Backups (und Wiederherstellungen), selbst wenn Sie eine Datenbank als Managed Service erwerben. Hochverfügbarkeit erfordert tiefes Wissen, insbesondere

Persönliches Exemplar für Thomas Bartholomäus

wenn Sie es selbst implementieren. Managed Services können dedizierte Datenbankadministratoren oder Database Reliability Engineers nicht ersetzen, ihnen aber auf jeden Fall bei der Arbeit helfen.

Container betreiben

Auch für das Betreiben von Containern bieten verschiedene Cloud-Anbieter unterschiedliche Lösungen an, sodass Sie sich nicht um die darunter liegende Plattform und Infrastruktur kümmern müssen. Bei AWS gab es beispielsweise im Herbst 2023 elf verschiedene Produkte, die mit Containern in Verbindung standen. Daher ist es wichtig, sorgfältig zu prüfen, welche Lösungen Ihren Anforderungen entsprechen und welche möglicherweise nicht notwendig sind.

Das manuelle Betreiben eines eigenen Kubernetes-Clusters ist anspruchsvoll, weshalb ein Managed Kubernetes-Cluster, der in der Cloud verwaltet wird, eine sinnvolle Alternative sein kann, um die Arbeitsbelastung des Teams zu reduzieren und sich auf das Wesentliche zu konzentrieren. Cloud-Anbieter bieten Automatisierungen an, um Kubernetes-Cluster bereitzustellen und zu skalieren, sodass Sie diese Aufgaben nicht manuell erledigen müssen. Damit machen Sie sich jedoch ziemlich stark von Ihrem Cloud-Dienstleister abhängig; der Wechsel zu einem lokalen Cluster ist nur mit großem Aufwand möglich. Zudem ist das Mieten eines solchen Clusters sehr kostspielig – was der bessere Weg für Ihre Situation ist, lässt sich nur durch das Evaluieren der eigenen Anforderungen herausfinden. Einige Hinweise dazu finden Sie in Abschnitt 8.5.1.

Probleme und Herausforderungen

Im Hinblick auf Platform as a Service (PaaS) gibt es einige Probleme und Herausforderungen, die berücksichtigt werden sollten. Am wichtigsten ist das Vendor Lock-in: Viele der PaaS-Dienste sind stark an die jeweilige Cloud-Umgebung angepasst. Wenn Sie es sich einmal dort bequem gemacht haben, ist der Aufwand, um diese Umgebung wieder zu verlassen, immens groß. Preiserhöhungen und geänderte Service-Bedingungen treffen Sie voll, und falls Ihr Dienstleister das Angebot vollkommen einstellt, haben Sie ein Problem.

Abhilfe kann darin bestehen, nur PaaS-Dienste zu nutzen, die auf einem Open-Source-Projekt basieren. Eine *Database as a Service* (DBaaS), die beispielsweise auf PostgreSQL basiert, ist weniger anfällig für einen harten Vendor Lock-in als proprietäre Lösungen.

Ein gewisser Lock-in ist allerdings immer enthalten. Schmerzen bei einer möglichen Migration sind ebenfalls immer vorhanden, egal ob bei proprietären Lösungen einer bestimmten Cloud oder bei einer selbst gemanagten Lösung in der eigenen Infrastruktur.

Am Ende müssen Sie also eine Abwägung treffen mit sehr vielen Faktoren. Ein wesentlicher Faktor sollte sein, dass ein offener Standard eingesetzt wird.

© Rheinwerk Verlag, Bonn 2024

Lift und Shift – und die Frage nach der Software-Architektur

Eine Frage, die ich in diesem Buch bisher noch gar nicht näher betrachtet habe, ist die Frage nach der Software-Architektur. Viele fragen sich oder erwarten, dass DevOps nicht mit monolithischen Projekten funktioniert oder dass ein Microservices-Ansatz verfolgt werden muss.

Wenn Sie das Buch bis zu dieser Stelle durchgelesen haben, sollte Ihnen klar sein, dass das für den allgemeinen Erfolg von DevOps keine so große Relevanz hat. Auf der technischen Ebene ist es natürlich wichtig, und das hat auch Einfluss auf die Teamstruktur. Aber das ist nur ein Faktor von vielen, der die allgemeinen Konzepte nicht verändert.

Wird über eine mögliche Migration zur Cloud diskutiert, kommen allerdings ein paar Punkte auf den Tisch, die man nicht vernachlässigen sollte. An vielen Stellen hört man etwa was von »Lift and Shift« in die Cloud. Der Ansatz ist einfach: Es wird lediglich die Anwendung so, wie sie funktioniert und betrieben wird, in die Cloud »gehoben«.

Die Vorteile der Cloud macht man sich damit nicht zunutze. Klar, man vermeidet dann sämtliche Hardware-nahe Arbeit, aber die restlichen Vorteile bleiben auf der Strecke. Eines der praktischen Features ist ja, dass man die Systeme bei Bedarf einfacher hoch- und herunterskalieren kann. Das geht aber nur, wenn es die Architektur der Anwendung auch zulässt.

Wenn man lediglich einen »Lift and Shift«-Ansatz wählt, dann hat man zunächst nur deutlich höhere Kosten, ohne einen wirklichen Mehrwert zu haben. Kurzfristig ist der Weg zwar gut; mittel- und langfristig muss die Architektur jedoch auf eine Cloud-native-Architektur umgestellt werden, damit die Vorteile der Cloud sinnvoller genutzt werden.

Software as a Service

Einen Schritt weiter als PaaS geht *Software as a Service* (kurz: SaaS). Hier wird die Software direkt den Kunden angeboten. Anstatt einen Online-Shop also selbst zu betreiben, mieten Sie einen fremden Shop an. Das ist natürlich weniger flexibel als eine Eigenentwicklung, kann aber eine passende Lösung sein, wenn Sie sich überhaupt nicht mehr um die Infrastruktur und ihre Bereitstellung kümmern wollen.

Für einen Online-Shop wie *schick-gekleidet.de* wäre der große Vorteil bei diesem Modell, dass durch geschickte Automatisierung Ressourcen sparsamer verwendet werden, denn die Last auf einem Online-Shop schwankt stark. In Angebotsphasen wie zu Weihnachten oder während Marketing-Kampagnen ist immer mehr los als in anderen Phasen. Durch ein SaaS-Modell kann dies gut abgefangen werden, denn die technische Umsetzung erfolgt durch den Anbeiter des SaaS-Dienstes. Der Nachteil ist natürlich, dass man Kontrolle abgeben muss und daher Flexibilität verliert.

Persönliches Exemplar für Thomas Bartholomäus

[!]

Skalierung ist wichtig, aber ...

Ein häufiger Fehler ist, dass zu viel Engineering-Aufwand betrieben wird, um eine Skalierung in die gefühlte Unendlichkeit zu ermöglichen. Meist wird dieser Aufwand nicht gebraucht, weil so viel Last nicht auf das System kommt. Betreiben Sie kein Over-Engineering, bevor Sie sich nicht ziemlich sicher sind, dass Sie dieses auch brauchen.

Es kommt allerdings nur selten vor, dass das ganze Kerngeschäft einer Firma so ausgelagert wird. Aus DevOps-Sicht stellt sich eher die Frage, ob man bestimmte Dienstleistungen einkaufen will: Möchte man Sourcecode-Management und CI/CD als SaaS anmieten oder möchte man es selbst betreiben? Will man eine eigene Monitoring- und Observability-Lösungen betreiben oder nutzt man einen Drittanbieter dafür? »Buy or Build?«

Die Antwort ist nicht einfach. Sofern Datenschutz und Privacy-Gründe nicht dagegensprechen, kann es sinnvoll sein, das Hosting den Experten zu überlassen und diese Dienste einzukaufen. Entscheidend ist, dass die Kosten und Risiken genau evaluiert werden. Wenn die DevOps-Toolchain unbrauchbar ist, dann steht die ganze Firma still, da keine Änderungen implementiert und deployt werden können.

Mein Rat ist folgender: Starten Sie mit der Analyse des Status quo, und machen Sie die Entscheidung davon abhängig, wie groß die Schmerzen mit dem aktuellen Setup sind. Je größer die Probleme sind, um die Infrastruktur rund um CI/CD am Laufen und aktuell zu halten, desto sinnvoller ist es, auf eine SaaS-Lösung zu setzen.

Abbildung 9.5 zeigt die unterschiedlichen Modelle und die Aufteilung der Aufgaben noch einmal im Überblick.

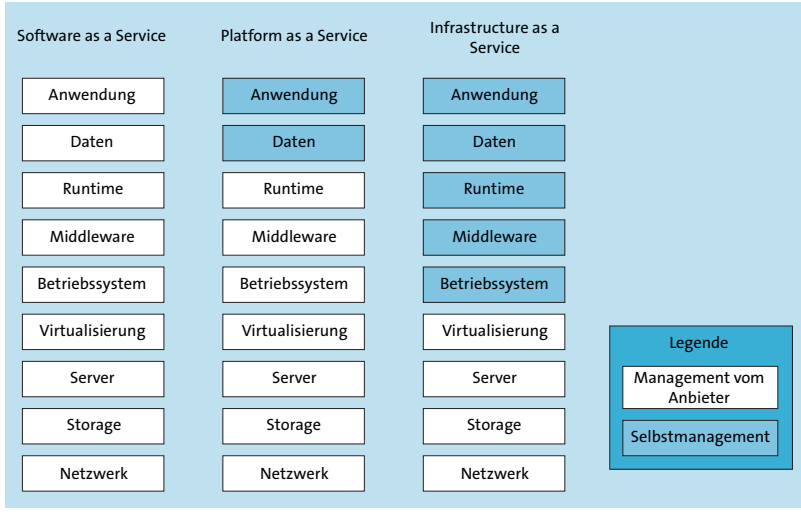

Abbildung 9.5 IaaS vs. PaaS vs. SaaS: Bei »Platform as a Service« wird mehr vom Anbieter gemanagt als bei IaaS, aber weniger als bei SaaS.

© Rheinwerk Verlag, Bonn 2024

9.3.4 Cloud-native

Ich habe bereits ausgeführt, dass eine Anwendung, die einer monolithischen Software-Architektur folgt und per *Lift and Shift* in die Cloud gehoben wird, nur höhere Kosten verursacht, aber die Vorteile nicht ausnutzt. Software, die explizit für die Cloud entworfen wurde, wird *Cloud-native* genannt. Mit Cloud-native lassen sich die technischen Aspekte der DevOps-Prinzipien einfacher umsetzen. Diese Services lassen sich dann zügiger entwickeln und können mit CI/CD-Pipelines gebaut, getestet, ausgerollt und auch zurückgerollt werden.

Die Designelemente von Cloud-native-Anwendungen sind Skalierbarkeit, Elastizität und Flexibilität, also genau die Vorteile, die eine Cloud bietet. Sehr abstrakt gesprochen bedeutet das, dass Anwendungen nicht als große, monolithische Einheiten geplant werden sollten, sondern dass sie aus vielen kleinen Teilen bestehen sollten, sogenannten *Microservices*. Diese sollten sich unabhängig voneinander entwickeln und betreiben lassen; ihr Zusammenspiel macht die Anwendung aus.

In einem Webshop sollte beispielsweise das Billing-Backend unabhängig von anderen Teilen der Software laufen. Wenn der Zahlungsvorgang zu lange dauert, weil gerade zu viele Kunden Bestellungen aufgeben, sollte dieser Teil flexibel skaliert werden können. Bei richtig großem Ansturm multipliziert man gleich die Instanzen des Webshops und sorgt über einen Loadbalancer dafür, dass die Kunden gleichmäßig verteilt werden. Das geht allerdings nur, wenn man sich bereits beim Design der Anwendung darüber Gedanken gemacht hat, wie etwa Sessions über unterschiedliche Instanzen verwaltet werden.

Technisch wird daher häufig auf Container gesetzt, da diese sich leicht starten, beenden und auf einer Vielzahl von Maschinen schnell hoch- und herunterskalieren lassen. Das geht nur sinnvoll, wenn die Anwendung einfach überwacht werden kann. Während beim klassischen Monitoring Anwendungen und Server fest vorgegeben werden, entstehen und verschwinden in der Cloud-native-Welt ständig Instanzen der Anwendung. Das Monitoring muss sich seine Ziele also ständig neu suchen (siehe Kapitel 10, »Vom Monitoring zur Observability«.)

Eine bestehende Anwendung zu einer Cloud-native-Architektur umzubauen, ist sehr aufwendig, und auch im Betrieb ändert sich einiges. Einfacher ist es meistens, direkt mit einer Neuentwicklung zu starten, die von Beginn an für den flexiblen Einsatz in der Cloud designt wurde.

The Twelve Factor App

Einen sehr guten Überblick über die Prinzipien und Methoden der Entwicklung einer Cloud-native-App bietet die Seite *https://12factor.net/* von Adam Wiggins. Darauf beziehen sich beispielsweise auch die Best Practices, die Google für den Bau und Betrieb von Containern in der Cloud empfiehlt:

https://cloud.google.com/architecture/best-practices-for-operating-containers?hl=de

Bevor Sie nun vorschnell Ihre komplette Anwendung über den Haufen werfen: Eine solche Microservice-Architektur spielt ihre Vorteile nur aus, wenn auch konsequent auf die Vorteile der Cloud gesetzt wird und Bedarf an Skalierung und Flexibilität besteht. Zudem muss der gesamte Entwicklungs- und Deployment-Workflow dazu passen. Die Cloud-native-Welt ist sehr schnelllebig und es gibt häufig Breaking Changes, die direkten Einfluss auf andere Anwendungen haben. Es muss also noch strenger darauf geachtet werden, dass es zu keinen Brüchen in der API zwischen den verschiedenen Diensten kommt, damit diese weiterhin sauber verwaltet und weiterentwickelt werden können. Ich würde behaupten, dass man eine gute DevOps-Kultur auch ohne Microservices und Cloud-native-Anwendungen umsetzen kann, aber Cloud-native ohne DevOps wird nicht funktionieren.

Cloud Native Computing Foundation

Wenn Sie sich in diesem Bereich weiterbilden wollen, sollten Sie sich die *Cloud Native Computing Foundation* (*https://www.cncf.io*) ansehen, die sich für die Belange der Cloud-native-Projekte einsetzt. Sie ist Teil der Linux Foundation. Die Foundation ist die Dachorganisation für einige Konferenzen wie die *KubeCon* und die *CloudNativeCon*.

Die CNCF bewertet die Projekte unter ihrem Dach mit den Reifegraden »sandbox«, »incubating« und »graduated«. In der Sandbox sind sehr viele Projekte, die recht neu sind. Nach einer gewissen Entwicklungszeit landen diese in »incubating«. Die wichtigsten und ausgereiftesten Projekte landen schließlich in der Kategorie »graduated«. Die bekanntesten Projekte sind Kubernetes, Prometheus und Helm.

Wenn Sie sich die *Cloud Native Landscape* unter *https://landscape.cncf.io/* anschauen, werden Sie sich fast erschlagen fühlen von einer Vielzahl an Projekten, die für die Cloud-native-Welt relevant sind. Alle diese Projekte arbeiten in und mit der Cloud. Man muss natürlich nicht alle kennen, und nicht alle waren von Anfang an Cloud-native-Entwicklungen. Dort können Sie sich aber einen Überblick darüber verschaffen, was sich im Bereich der Cloud-Infrastrukturen tut.

9.3.5 Die Cloud bei schick-gekleidet.de

Innerhalb der Firma wurde heiß diskutiert, wie die Cloud-Strategie aussehen sollte. Viele fokussierten sich dabei hauptsächlich darauf, ob es einer der großen amerikanischen Cloud-Provider sein sollte oder ein kleinerer deutscher Anbieter. Die Kostenfrage stand auch immer wieder zur Debatte, genauso wie die Frage, auf welche Cloud migriert werden sollte.

Das Business von *schick-gekleidet.de* findet zu einem großen Teil im Weihnachtsgeschäft statt. Hier werden etliche Angebote gefahren und es wird weitaus mehr Geld eingenommen als im Rest des Jahres. Der Verkaufsschlager ist jedes Jahr ein neuer »Ugly Christmas Sweater«.

© Rheinwerk Verlag, Bonn 2024

Für die Infrastruktur hatte das bisher Folgen: Die meiste Zeit des Jahres langweilte sich die Hardware im eigenen Rechenzentrum, denn die Auslastung war niedrig. Je näher Weihnachten kam, desto höher wurde die Last auf den Servern. Zu Hochzeiten waren die Kapazitäten also regelmäßig überfordert: Es kam zu langsameren Bestellprozessen und Kunden sprangen entnervt ab.

Die Cloud-Strategie sollte diese Probleme lösen: Es sollte idealerweise nur so viel Infrastruktur vorhanden sein, wie gerade auch gebraucht wird. Das ist nachts deutlich weniger als abends, im Sommerloch weniger als im Vorweihnachtstrubel. Die automatische Skalierung war also hilfreich, um sowohl Kosten zu sparen als auch Lastspitzen abzufangen. Welche Cloud es nun letztendlich wurde, war eigentlich irrelevant, denn es kommt nur auf die Tools und die Architektur an, die genutzt werden. Im Rahmen der Modernisierung wurde viel auf Container-Technologien umgestellt, und langsam, aber sicher wurde auch ein grundlegender Umbau des Shops in eine Cloud-native-Anwendung geprüft.

Reflexion

Die Provisionierung und die Verwaltung einer Cloud-Infrastruktur sind eine Wissenschaft für sich – Sie müssen sehr genau evaluieren, wie Ihre Anforderungen aussehen und welche Sprünge Sie Ihrer Anwendung und den Arbeitsprozessen zutrauen. Eine schwerfällige Anwendung einmalig in die Cloud zu hieven, wird erst mal nur für Kosten sorgen, ohne dass sich viel ändert.

Ja, Cloud-Provider kosten Geld, und das auch nicht so wenig. Eigene Hardware zu betreiben, kostet jedoch auch. Die Kosten lassen sich in der Cloud jedoch gut optimieren, wenn Sie die Vorzüge entsprechend nutzen. Dabei dreht sich alles um Automatisierungen und die bedarfsgerechte Skalierung.

Besonders wenn Sie sich für die Modelle *Platform* und *Software as a Service* interessieren, wird der Kostenaspekt komplizierter. Ist es günstiger, interne Expertise aufzubauen, also ein Team einzustellen und zu trainieren? Oder gibt man einem Cloud-Provider viel Geld dafür, dass man alles fertig von der Stange bekommt? Dabei dürfen Sie allerdings nicht vergessen, dass auch die Cloud-Dienste verwaltet werden müssen – allerdings mit deutlich weniger Personal.

Grundsätzlich ist eine gewisse Abhängigkeit immer vorhanden: Entweder Sie sind abhängig von den internen Kollegen oder von dem Cloud-Anbieter.

9.4 Stärkere Zusammenarbeit von Dev und Ops

Zurück zur Kultur! Wie bei den übrigen Teilen von DevOps geht es auch beim Betrieb der Dienste um die Verbesserung der Zusammenarbeit zwischen den unterschiedlichen Teams. In diesem cross-funktionalen Team bekleiden also auch weiterhin ver-

schiedene Personen verschiedene Rollen mit ihren jeweiligen Aufgaben (siehe Abbildung 9.6).

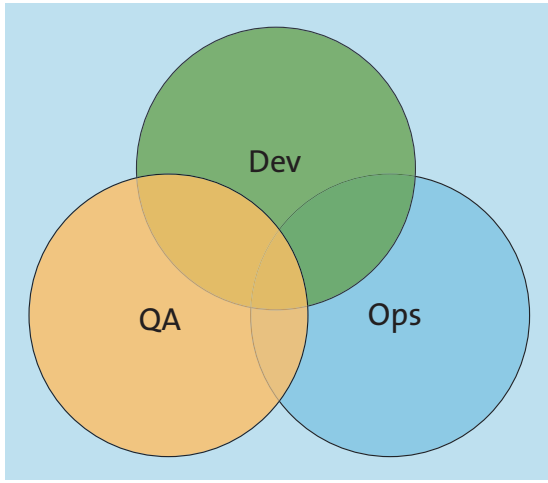

Abbildung 9.6 Dev, QA und Ops überlappen sich in einigen Aufgabengebieten nun viel mehr – je nach konkretem Schwerpunkt der Person mal mehr und mal weniger.

An diesem Punkt stellen sich jetzt folgende Fragen: Was passiert bei einer Downtime? Und: Wie wird mit Fehlern umgegangen?

9.4.1 Die Bereitschaft ist für (fast) alle da!

Eine der ersten Aussagen, die man vom bisherigen Betriebsteam hört, wenn es heißt, dass nun auch die Entwickler Deployments durchführen sollen, ist:

»Aber dann müssen sie doch auch den Bereitschaftsdienst übernehmen!«

Und ja, das ist korrekt! In einem cross-funktionalen Team, das nach DevOps-Prinzipien arbeitet, ist es unabdingbar, dass auch Entwickler den Bereitschaftsdienst übernehmen. Nicht umsonst hört man häufig den Spruch: »You built it, you run it, you own it.« Es ist ganz im Sinne des Teams, dass diejenigen, die das Projekt entwickeln, es auch laufen lassen und somit verantworten.

Bei *schick-gekleidet.de* wurde immer eine Person aus dem Betriebsteam geweckt, wenn das Dateisystem regelmäßig vollief. Im DevOps-Team hängt die Bereitschaft nun nicht mehr nur an dem klassischen Betriebsteam, sondern an einem Team mit verschiedenen Rollen.

Neben einem Admin wird nun auch ein Entwickler für den Bereitschaftsdienst eingeteilt, aber auch andere Rollen können hilfreich sein. Besonders sollte bei Security-Incidents schnell gehandelt werden können.

© Rheinwerk Verlag, Bonn 2024

Vertragliche Grundlagen

Dass Personen den Bereitschaftsdienst übernehmen, hat gegebenenfalls auch Auswirkungen auf ihren Arbeitsvertrag. Das Thema ist wichtig und sollte nicht vergessen werden. Das betrifft auch Arbeitszeitregelungen sowie die angepasste Bezahlung.

Nur weil mehrere Personen im Bereitschaftsdienst sind, heißt das allerdings nicht, dass auch alle benachrichtigt werden müssen. Es ist auch weiterhin im Interesse des Teams, dass primär jemand alarmiert wird, der für den Betrieb zuständig ist, bevor diese Person den Auftrag dann weitergibt, je nachdem, wie der Bedarf ist und welche Fehler auftreten.

Zurück zum vorangegangenen Beispiel: Das Problem mit dem volllaufenden Dateisystem wird untersucht und es wird entdeckt, dass die Anwendung eine Logdatei schreibt, die zu dem Fehler führt. Da das bisherige Betriebsteam durch die Zusammenführung der Teams und der Aufgaben nun auch fachlich deutlich näher an der Software ist, schaut der Mitarbeiter im Bereitschaftsdienst sich auch den Inhalt der Logdatei an.

Anstatt hier also direkt zum Löschen zu springen, kann dieser Mitarbeiter besser beurteilen, was das grundsätzliche Problem ist, denn er war ja in die Entwicklungsarbeiten eingebunden. Aber nur, weil hier eine größere Vertrautheit mit dem Sourcecode und dem Projekt vorhanden ist, heißt es nicht unbedingt, dass die Person das Problem direkt lösen kann. Im ersten Schritt ist es zunächst erst mal wichtig, zu erkennen, ob es ein gravierendes Problem ist oder nicht.

Wenn in der Logdatei etwa steht, dass Fehler geworfen werden, wenn Kunden versuchen, Waren in den Warenkorb zu legen, und somit den Kauf nicht abschließen können, dann führt das zu Umsatzverlusten und unzufriedenen Kunden. In diesem Fall wäre es also fatal, wenn man den Entwickler im Bereitschaftsdienst nicht alarmiert. Nur gemeinsam lässt sich dieses Problem lösen, und man kann idealerweise direkt ein Release durchführen, das das Problem behebt.

Das ginge zwar auch ganz ohne DevOps-Prinzipien, aber nur, wenn offen und transparent innerhalb der Teams kommuniziert wird. Ist das grundsätzlich nicht der Fall, fehlen Informationen, um sinnvolle Entscheidungen treffen zu können. Und genau aus solchen Gründen wird man dann immer wieder wegen Problemen geweckt, die man an der Wurzel korrigieren sollte, anstatt nur die Symptome zu behandeln.

Bereitschaftsdienst für alle hat den wesentlichen Vorteil, dass bei Alarmierungen gemeinsam an verschiedenen Lösungen gearbeitet wird.

Das gemeinsame Arbeiten ist mit getrennten Teams und den damit verbundenen Silos meist nicht so ausgeprägt, da die anderen Personen meist unbekannt sind und

Persönliches Exemplar für Thomas Bartholomäus

sich zu sehr auf Ziele des eigenen Teams fokussiert wird statt auf das Ziel der Organisation. Einem Freund hilft man tendenziell schließlich eher als einer unbekannten Person.

Fassen wir zusammen: Es ist wichtig, dass am Bereitschaftsdienst immer Personen mit verschiedenen Hintergründen und Kenntnissen beteiligt sind. Es sollte zudem klar definiert sein, wann wer informiert werden soll, um die Störungen effizient zu behandeln. Dazu gehört auch ein gesundes Rotieren des Bereitschaftsdienstes.

9.4.2 Blameless Post-Mortems

Auch beim Arbeiten nach DevOps-Prinzipien wird es immer mal wieder Downtimes geben. Eigentlich lautet die Frage gar nicht, ob es Downtimes geben wird, sondern wann sie auftreten und wie mit ihnen umgegangen wird. (Die Frage nach dem »wie« ist übrigens ganz unabhängig davon, ob nun nach DevOps-Prinzipien gearbeitet wird oder nicht.)

Wenn es zu Downtimes kommt, sind die einzig wichtigen Fragen, warum das Problem aufgetreten ist und wie es zukünftig vermieden werden kann. Im Rahmen von DevOps-Prinzipien ist es wichtig, dass offen mit Fehlern umgegangen wird, unabhängig davon, wer am Ende schuld war. Idealerweise ist die Kultur ohnehin so, dass alle von Anfang an wissen, dass es völlig in Ordnung ist, dass Fehler passieren. Für das gemeinsame Arbeiten ist es unbedingt notwendig, dass eine offene Fehlerkultur gelebt wird. Nicht selten sieht man allerdings, dass Fehler, wenn sie auftreten, nur notdürftig behoben und anschließend unter den Teppich gekehrt werden. Auf diese Weise wird aber weder erkannt, was das Problem war, noch lernt irgendjemand neue Dinge dazu. Es sind allerdings zwei Grundsätze zu beachten:

- Bereits geschehene Fehler sollen nicht erneut begangen werden.
- Es sollte detailliert nachvollziehbar sein, warum welche Fehler überhaupt aufgetreten sind, damit man aus ihnen lernen kann.

In der DevOps-Welt nennt man diesen Vorgang *Blameless Post-Mortem*. Bei ihm geht es darum, dass nach einer Downtime oder einem anderen gravierenden Fehler dokumentiert wird, was genau vorgefallen ist, warum dieser Fehler aufgetreten ist und warum diese Probleme nicht vorher aufgefallen sind. Ganz wichtig: Bei diesem Vorgang soll niemandem Schuld zugewiesen werden, da Fehler menschlich sind.

Blameless Post-Mortems können für jegliche Art von Incidents geschrieben werden. Das bisher genannte Beispiel war eine Downtime, aber auch Security-Incidents können relevant und hilfreich sein. Bei einem Blameless Post-Mortem handelt es sich letztlich um ein ausformuliertes Dokument über einen Incident, das rekapituliert, was passiert ist. Das Herunterschreiben ist keine Bestrafung, sondern eine Retrospektive, um dabei neue Erkenntnisse zu sammeln.

© Rheinwerk Verlag, Bonn 2024

Aber was steht nun genau in dem Dokument? Wichtig ist, im ersten Schritt zu beschreiben, was für eine Auswirkung der Incident hatte und welche Schritte nötig waren, um den Fehler zu korrigieren. Essenziell ist dabei, dass der *Root Cause* gefunden und dokumentiert wird. Wenn ein Root Cause nicht in einem Blameless Post-Mortem zu finden ist, dann ist es kein gutes Post-Mortem. Ebenfalls sollten *Follow-up Actions* definiert werden, damit ein solcher Incident nicht erneut auftreten und frühzeitig verhindert werden kann.

Wann genau sollte ein Post-Mortem geschrieben werden? Es gibt verschiedene Arten von Triggerpunkten, die verwendet werden können:

- eine Downtime, die auch bei den Endnutzern spürbar wird
- ein Security-Incident, bei dem Daten abgegriffen wurden
- Datenverluste
- ein Ausfall von Kernkomponenten der internen Infrastruktur

Es wird viele Fälle geben, wo mal ein Downgrade gemacht wird, weil dort Fehler passiert sind. Nicht für jeden Fall ist es nötig, ein Post-Mortem zu schreiben, denn nichts ist unnötiger, als sich extra Arbeit zu machen, wenn kein Nutzen ersichtlich ist. Problematisch wäre es eher, wenn ein Rollback nicht funktioniert. In diesem Fall wäre, um beim konkreten Beispiel zu bleiben, ein Post-Mortem wohl sinnvoller.

Wichtig ist, dass hier für die eigene Organisation ordentliche Kriterien definiert werden, wann ein Post-Mortem geschrieben werden sollte und wann vielleicht auch nicht. Diese Kriterien sollten nie in Stein gemeißelt sein, sondern sollten, wie auch der Sourcecode generell, immer im stetigen Wandel sein. Nach dem Schreiben eines nicht hilfreichen Post-Mortems kann man zum Beispiel immer noch die eigenen Regeln überarbeiten.

Viele Firmen und Organisation machen ihre Post-Mortems auch öffentlich, und das, obwohl einige Post-Mortems der Kategorie »peinlich« zugeordnet werden könnten. Aber genau das ist auch der Spirit von Blameless Post-Mortems: Selbst vermeintlich peinliche Fehler sind Fehler, die nicht passieren sollten und trotzdem geschehen sind.

Post-Mortem bei GitLab

Ein Paradebeispiel für ein öffentliches Post-Mortem ist von GitLab im Februar 2017 veröffentlicht worden:

https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/

Darin geht es um einen Incident vom 31. Januar 2017, bei dem initial ein Datenbank-Administrator die Synchronisierung auf dem Secondary-Server korrigieren wollte. Er löschte dazu das Verzeichnis auf dem Dateisystem mit der Absicht, dass dann die Synchronisierung von vorn starten sollte. Leider führte er dies auf dem Primary-

Server aus statt auf dem Secondary. Das Resultat war, dass *GitLab.com* direkt offline war und Backups eingespielt werden mussten.

Das allein wäre schon ein anschauliches Beispiel für ein Post-Mortem. Allerdings ist die Story hier noch nicht zu Ende: Die Backups ließen sich nämlich nicht einspielen. Es gab zwar verschiedene Arten von Backups, allerdings funktionierten viele nicht, da sie nie wirklich getestet worden waren. Letztlich war *itLab.com* für über 18 Stunden offline und es gab einen Datenverlust von ein paar Stunden.

Dieser Incident passierte einige Jahre, bevor ich selbst GitLab-Mitarbeiter wurde. Ich beobachtete das Ganze an dem Tag von außen. Viele Kollegen und Bekannte in verschiedenen Firmen und Organisationen, die ich kenne, verfolgten ebenfalls das Outage. Das Resultat war, dass viele nur wegen dieses Post-Mortems ihre eigenen Backups geprüft haben, damit sie nicht auch in solch einem Schlamassel landen. Das öffentliche Teilen von Post-Mortems kann also sowohl für intern als auch für extern Lesende von Vorteil sein. Denn so kann ein Blameless Post-Mortem eine Möglichkeit sein, Wissen zu teilen und dabei gleichzeitig künftige Fehler zu vermeiden.

9.4.3 Kommunikationslösungen und ChatOps

In diesem Buch habe ich bereits über die Probleme von Silos in den Teams und über die Bedeutung effektiver Kommunikation geschrieben. Bisher ging es aber noch nicht um Lösungen für die interne Kommunikation innerhalb der Organisation.

Es ist wichtig zu betonen, dass Kommunikation nicht auf die Verwendung von Ticketsystemen, Code-Reviews und ähnlichen Tools beschränkt sein sollte. Tools wie *Slack*, *Microsoft Teams*, *Mattermost*, *Rocket.Chat* und ähnliche sollten nicht nur als einfache Chat-Plattformen betrachtet werden. Sie können in verteilten Organisationen dazu genutzt werden, Fragen in anderen Teams zu stellen und die Zusammenarbeit über Team- und Rollengrenzen hinweg zu fördern. Leider wird dieser Aspekt in vielen Unternehmen oft vernachlässigt.

Häufig wird lediglich darauf geachtet, dass ein Chat-Tool vorhanden ist und vielleicht sogar in ein Videokonferenzsystem integriert ist. Obwohl dies zweifellos wichtig ist, geht eine gesunde Kommunikationskultur über einfaches Chatten hinaus. Hierbei gibt es viele mögliche Fehlerquellen. Ich sehe oft, dass nur geschlossene Kanäle erstellt werden und öffentliche Kanäle entweder gemieden werden oder sogar verboten sind. Dies verstärkt die Silowände zwischen den Teams, da es nicht möglich ist, einfach in einen Kanal zu wechseln, um eine Frage zu stellen.

Ein weiterer häufiger Fehler ist die Nutzung eines Chat-Tools als Dokumentationsquelle für Arbeitsprozesse. Obwohl diese Tools über eine Suchfunktion verfügen, ist diese oft nicht effizient genug, um wichtige Informationen zu finden und zu bearbei-

© Rheinwerk Verlag, Bonn 2024

ten. Ein Chat-Tool ist kein Ort, an dem Entscheidungen dokumentiert und Arbeitsprozesse beschrieben werden sollten!

Noch wichtiger ist, dass Raum für den Austausch von fachlichen und nicht fachlichen Aspekten geschaffen wird. Oft gibt es in Unternehmen Menschen mit vielfältigen Kenntnissen in Programmiersprachen, Frameworks und Tools. Diese Kenntnisse haben sie in früheren Positionen oder als Hobby erworben; sie werden zwar derzeit im Unternehmen nicht genutzt, könnten aber bei zukünftigen Entscheidungen hilfreich sein.

Wenn ein Unternehmen eine offene Kultur fördert und auch außerhalb der unmittelbaren Geschäftstätigkeit Raum für den Austausch solcher Themen bietet, kann dies aus DevOps-Perspektive ebenfalls von Bedeutung sein.

SCHREIEN SIE RUHIG MAL!!!

Es sollte auch für vermeintlich völliges Off-Topic Raum geschaffen werden. Mein Lieblingsbeispiel ist ein Channel namens `#all-caps`. Es gibt nur eine Regel in diesem Channel: Es darf ausschließlich in Großbuchstaben geschrieben werden. Ein solcher Chat sorgt dafür, dass Leute ungehemmt Lappalien hineinschreiben: Sie »beschweren« sich über Dinge, die falsch laufen, oder schreiben rein, welche Dummheiten sie gemacht haben und was sie daraus gelernt haben.

Durch die Nutzung von Großbuchstaben kommt die Kommunikation zwar als Schreien rüber, aber alle wissen, dass es nicht ernst gemeint ist. So treten per Zufall auch gerne einmal Prozesse mit Stolpersteinen zutage, von denen jemand genervt ist und die man idealerweise direkt anpasst. Das ist nicht ohne Grund mein absoluter Lieblings-Channel.

Kommunikation via Chat ist aber nicht das Einzige, wofür eine gute und sinnvolle Kommunikationslösung genutzt werden kann. Praktisch können auch Chatbots sein, mit denen sich ChatOps umsetzen lässt.

ChatOps

Die Idee hinter ChatOps besteht darin, Chatbots einzusetzen, um die Administration der Infrastruktur und der Anwendungen zu erleichtern. Dies umfasst nicht nur die Möglichkeit, Deployments über das Chat-Tool Ihrer Wahl auszulösen, sondern auch die verbesserte Sichtbarkeit von Informationen. Da der Zugriff auf diese Systeme sehr einfach ist, kann mit ihnen gut Sichtbarkeit erzeugt werden. Nicht jeder wird ständig das CI/CD-Tool überprüfen, um festzustellen, ob gerade ein Deployment durchgeführt wird oder nicht. ChatOps kann als eine Methode zur Weitergabe von Informationen im Team betrachtet werden, durch die die Kommunikationsstruktur vereinfacht wird.

Persönliches Exemplar für Thomas Bartholomäus

Ein solcher Ansatz kann auch für nichttechnische Mitarbeiter nützlich sein. Einige Informationen sind für sie wichtig, obwohl sie möglicherweise keine Erfahrung mit der technischen Umsetzung haben. Auf diese Weise können auch sie über Deployments informiert werden, ohne in ein Tool schauen zu müssen, mit dem sie wenig vertraut sind.

Typischerweise werden Alarme des Monitoring-Systems in spezifische Chat-Kanäle integriert, die neben einigen Informationen auch Links zu weiteren Details bereitstellen. Etwas fortgeschrittener ist es, auch Canary- und Blue-Green-Deployments über Chatbots auszulösen und anzupassen, sodass das Team stets den Überblick behält. Dies gilt auch für Rollbacks. Dabei ist der Chatbot nur der Auslöser; die dauerhafte Protokollierung und die Konfiguration finden im Deployment-Tool statt, um die Nachverfolgbarkeit zu gewährleisten.

Reflexion

Das Niederreißen der Silos sowie die Automatisierungen von Deployments und Infrastruktur sind Kernbestandteile beim Betrieb der Anwendungen und der Infrastruktur, auf der sie laufen.

Wenn Sie überlegen, auf die Cloud zu setzen, dann sollten Sie auch schauen, dass Sie die Skalierungsmöglichkeiten effektiv nutzen. Eine Migration der Anwendung in die Cloud ohne Anpassung ist als erster Schritt zwar nötig, aber mittel- und langfristig nicht sinnvoll.

In der Industrie sieht man immer mehr den Schwenk in Richtung Cloud. Das bezieht sich nicht nur auf die eigenen Anwendungen, sondern auch auf Dienste von Drittanbietern, die etwa als SaaS angeboten und genutzt werden können.

Die Cloud ist allerdings nur ein Aspekt beim Betrieb der Anwendung. Im Betrieb geschehen Fehler, und diese sollten Sie definitiv nicht unter den Teppich kehren, sondern etwa über Blameless-Post-Mortems im Unternehmen verteilen. Wenn Sie Angst vor Konsequenzen haben, sollten Sie vielleicht im kleinen Rahmen anfangen: erst im Team, dann in der Abteilung und dann allmählich den Empfängerkreis vergrößern.

Starten Sie probeweise mit dem Berichten von kleinen Fehlern und Stolpersteinen, damit Sie später auf größere Probleme hinweisen können. Damit bekommen Sie ein besseres Gefühl dafür, wie auf solche Fehlerberichterstattungen in der Firma reagiert wird.

9.5 Konfigurationsmanagement: Everything as Code

Im Laufe des Kapitels wurde immer wieder darauf hingewiesen, dass die Infrastruktur möglichst vollständig automatisiert werden sollte. Die Definition der Zielumge-

© Rheinwerk Verlag, Bonn 2024

bung sollte immer »as Code« vorliegen. Idealerweise werden alle Arbeitsabläufe und Umgebungen als Code definiert. Das betrifft sowohl die Infrastruktur mit Terraform als auch das Konfigurationsmanagement mit Ansible oder Puppet oder die Deployments nach Kubernetes mit einem GitOps-Pattern.

Dafür gibt es zwei wichtige Gründe:

- Der Zielzustand ist auf diese Weise klar definiert, und alle können einsehen, wie der Stand gerade ist, und per Pull bzw. Merge Request an einer Änderung mitwirken.
- Die Reproduzierbarkeit ist gegeben: Eine Konfiguration kann beliebig analysiert und nachgestellt werden.

Ein häufiges Problem bei einem manuellen oder nur halb automatisierten Konfigurationsmanagement ist, dass viele Dienste zwar auf vielen verschiedenen Servern ausgerollt wurden, dass bei Problemen aber immer wieder händisch eingegriffen wurde, um die auftretenden Probleme zu korrigieren.

Das ist zwar kurzfristig häufig eine schnelle und einfache Lösung für das Problem, allerdings werden die Änderungen so gut wie nie auf anderen Instanzen des Dienstes auf anderen Servern nachgezogen. So kommt es zu Inkonsistenzen, dem sogenannten *Configuration Drift*: Eigentlich gleiche Systeme verhalten sich ein bisschen unterschiedlich, was die Fehlersuche sehr kompliziert macht.

Stattdessen sollten Konfigurationen einheitlich versioniert und ausgerollt werden. Wenn ein Problem gelöst wird, dann nicht nur auf dem speziellen Server, sondern zentral über das Konfigurationsmanagement, das dann die Aufgabe übernimmt, diesen Soll-Zustand an alle überwachten Systeme zu verteilen.

Anders entstehen unweigerlich Inkonsistenzen, die nicht nur nervig sind, sondern auch ineffizient: Die gleiche Arbeit wird immer und immer wieder gemacht. Ein Fehler tritt auf, er wird untersucht, das Problem wird gefixt, und dann geht es später wieder zurück, wo der gleiche Fehler noch mal auftritt. Spaß macht das nicht.

Je nachdem, welche Lösung man wählt, ist der Code eher deklarativ oder eher imperativ. Deklarativ bedeutet, dass im Code ein Soll-Zustand beschrieben wird, während beim geskripteten, imperativen Ansatz konkret die Arbeitsschritte genannt werden müssen, die ausgeführt werden sollen. Beides hat Vor- und Nachteile; ich ziehe einen *deklarativen Ansatz* vor. So lässt sich leichter *Idempotenz* erreichen. Das heißt, es sollte bei mehrfacher Ausführung dazu führen, dass immer der gleiche Zielzustand erreicht wird. Wenn beispielsweise definiert wird, dass eine Software auf dem Zielsystem installiert sein soll, führt der deklarative Ansatz dazu, dass nichts passiert, wenn die Software schon vorhanden ist – der Soll-Zustand ist ja schon erreicht. Bei einem imperativen Ansatz würde die Installation trotzdem wieder ausgeführt werden, denn das war ja der Befehl.

Persönliches Exemplar für Thomas Bartholomäus

Durch die Definition als Code ist auch die Kollaboration zwischen den Teams gesichert: Wenn eine Konfigurationsänderung benötigt wird, kann direkt ein Änderungsvorschlag im Repository mit dem Code gemacht werden.

Solche Änderungen sollten stetig ausgerollt werden. Es geht zwar auch ohne, aber das führt nur wieder dazu, dass jemand händisch auf den Systemen herumpfuscht, was man vermeiden sollte.

9.5.1 Infrastructure as Code mit Terraform

Mit Terraform lassen sich Infrastrukturen als Code definieren und erzeugen. Daneben gibt es noch diverse andere Tools wie Pulumi (*https://www.pulumi.com/*) und Crossplane (*https://www.crossplane.io/*), aber auch Ansible, Puppet und Saltstack, die hauptsächlich für das Konfigurationsmanagement bekannt sind, können auch für Infrastructure as Code verwendet werden, allerdings mit einigen Einschränkungen.

Buchempfehlung

Im Rheinwerk-Verlag ist ebenfalls das Buch »Terraform – Das Praxisbuch für DevOps-Teams und Administratoren« erschienen. Dort wird Infrastructure as Code mit Terraform detailliert behandelt:

https://www.rheinwerk-verlag.de/terraform/

OpenTofu

Im Herbst 2023 stellte das Unternehmen HashiCorp die Lizenzen seiner Projekte – darunter auch Terraform – von der Open-Source-Lizenz *Mozilla Public License* (v2.0) auf die *Business Source License* (BSL) um, die nur noch »source-available« ist. In der Open-Source-Community kam dieser Schritt nicht gut an, daher bildeten sich schnell Abspaltungen. Unter dem Dach der Linux Foundation wird jetzt an OpenTofu (*https://opentofu.org/*) gearbeitet. Hier dürfte zukünftig noch einiges passieren.

Terraform wurde im Juli 2014 von der Firma HashiCorp veröffentlicht. Um die Infrastruktur zu definieren, kommt eine eigene Konfigurationssprache zum Einsatz, nämlich die *HashiCorp Configuration Language* (kurz: HCL). Diese Sprache ist eine Eigenentwicklung und sowohl maschinen- als auch menschenlesbar. Sie sollte einfacher als JSON zu lesen und mächtiger und weniger fehleranfällig als YAML sein.

In diesen Konfigurationsdateien werden die benötigten Ressourcen deklarativ definiert. Dazu benötigt Terraform *Provider*, die gewissermaßen den HCL-Code in die Anweisungen der dahinter liegenden Anbieter übersetzen. Es gibt zahlreiche Provider für alle größeren und viele kleinere Cloud-Anbieter.

© Rheinwerk Verlag, Bonn 2024

In den Konfigurationsdateien werden etwa die virtuelle Maschine auf einem Cloud-Provider, die Nutzung der Datenbank des Cloud-Providers als Managed Service, Storage, DNS und virtuelle Netzwerke definiert. Idealerweise wird die komplette Infrastruktur in der HCL beschrieben, sodass alles damit provisioniert und verwaltet wird.

Die Terraform-Projekte werden dann wie übliche Software-Entwicklungsprojekte behandelt, auch wenn hauptsächlich die Admins Terraform nutzen. Sie arbeiten in Git-Repositorys und nutzen Git-Workflows, um Änderungen beizutragen und zu reviewen. So kommen alle Vorteile des Sourcecode-Managements ins Spiel, denn über das Repository sehen Sie nicht nur, wie die Infrastruktur aussieht, sondern Sie haben auch eine Historie, an der Sie ablesen können, wie sich die Infrastruktur entwickelt hat. Jede Änderung wird dokumentiert.

Bedingung dafür ist, dass ausschließlich Terraform für die Änderungen an der Infrastruktur genutzt wird. Es muss also auch in solchen Projekten eine Pipeline geschrieben werden, die Terraform ausführt, um entsprechende Änderungen auszurollen.

Neben dem reinen Code benötigt Terraform auch noch ein Backend, in dem der *Terraform State* gespeichert wird (siehe Abbildung 9.7). In dieser Datei wird die aktuelle Konfiguration persistiert. Dadurch kann Terraform nachvollziehen, ob und welche Änderungen es im Vergleich zur vorherigen Ausführung gab. Standardmäßig wird diese Datei lokal gespeichert und sollte ganz im Sinne der Zusammenarbeit allen Mitgliedern im Team zugänglich sein.

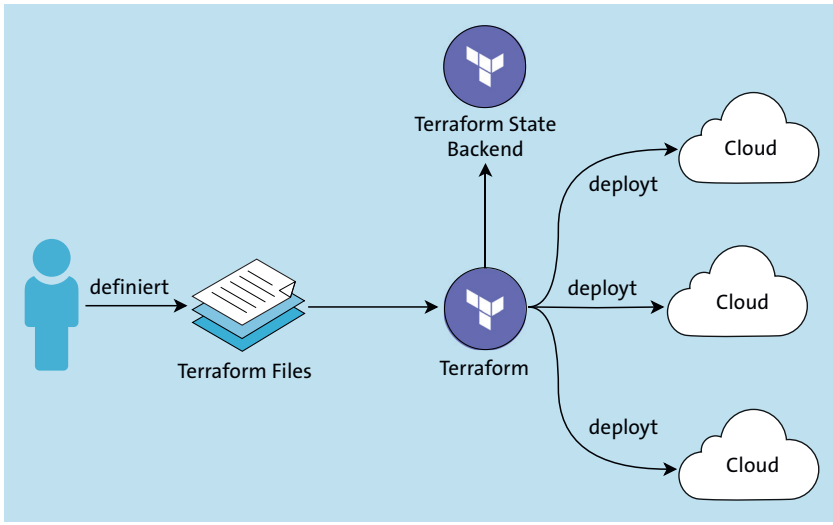

Abbildung 9.7 Die Funktionsweise von Terraform

Wie zuvor schon erwähnt, gibt es neben Terraform auch noch weitere Tools aus dem »Infrastructure as Code«-Bereich. Jedes Tool verfolgt seinen eigenen Ansatz. *Pulumi* (*https://www.pulumi.com/*) ist sehr flexibel einsetzbar, da es eine Vielzahl an Pro-

grammiersprachen unterstützt und Konstrukte wie Schleifen direkt verfügbar sind. Das sorgt dann allerdings auch dafür, dass es viele unterschiedliche Wege zum Ziel gibt; meistens dauert es ein bisschen, bis man sich in fremden Umgebungen zurechtfindet und versteht, was passiert.

Noch schwieriger ist der Einstieg bei *Crossplane* (*https://www.crossplane.io/*). Crossplane setzt auf Kubernetes und nutzt dessen Features, um die Zielumgebungen hochzuziehen. Das ist ziemlich komplex, hat aber den Vorteil, dass geänderte Quelldateien direkt ausgeführt werden, ohne dass mit *State Backends* oder Pipelines hantiert werden muss.

[!]

Ressourcen löschen

Ein Problem ist, wie mit gelöschten Ressourcen umgegangen wird. Wenn Sie Dateien löschen, die die Infrastruktur abbilden, sollte dann die Infrastruktur dahinter auch weggeräumt werden oder nicht?

Was genau passiert, hängt vom eingesetzten Tool ab. Einige Tools setzen aus guten Gründen darauf, dass das Fehlen von Information nicht bedeutet, dass die Infrastruktur automatisch weggeräumt wird: Auch Löschungen müssen als Auftrag formuliert werden.

Falls Sie stattdessen mit Setups arbeiten, bei denen fehlende Informationen als Löschaufträge interpretiert werden, dann müssen Sie sehr sorgfältig arbeiten, denn Fehler können gravierende Folgen haben. Dabei reicht schon ein geänderter Name, der zur Identifizierung von Datenzugriffspunkten genutzt wird, damit Daten ruckzuck gelöscht werden könnten.

Und Datenverluste können immer wieder auftreten. Daher ist ein gescheiter Backup-und-Restore-Prozess unabdingbar. Das sind typische Ops-Aufgaben, die bei einem DevOps-Team häufig »hinten runterfallen«. Wenn Backups angelegt werden, sollten diese auch regelmäßig getestet werden. Es ist zwar schön und gut, dass man die Infrastruktur und die Konfiguration der darauf laufenden Anwendungen hochgezogen bekommt, aber wenn dann die benötigten Daten fehlen, hat man ein gravierendes Problem.

Nachdem die Infrastruktur steht, müssen die Systeme, die man provisioniert hat, konfiguriert und mit Software bestückt werden. Ob es dabei um Bare Metal oder um virtuelle Maschinen geht, spielt keine große Rolle, schließlich brauchen beide auf ihrem installierten Betriebssystem weitere Software mit einer Konfiguration.

In der Vergangenheit wurden solche Arbeiten händisch gemacht oder es wurden Skripte geschrieben, die die Anwendungen auf den Systemen installiert, Konfigurationsdateien ausgerollt und Dienste in der richtigen Reihenfolge gestartet haben.

Dieses Vorgehen ist zwar besser, als auf gar keine Automatisierung zu setzen, allerdings ist das Ganze aufwendig zu warten und fehleranfällig. Abhilfe schafft ein Kon-

© Rheinwerk Verlag, Bonn 2024

figurationsmanagement-Tool, das die Arbeit mit den Konfigurationen abstrahiert. Derartige Tools bringen zwar wieder eine gewisse Komplexität mit; nach einer Lernkurve lässt sich aber effizient mit ihnen arbeiten, sodass Hunderte Mausklicks durch einen Auftrag ersetzt werden können. Damit gelingt auch der Neubau der Infrastruktur deutlich besser, da ihr Soll-Zustand verwaltet und reviewt vorliegt.

Konfigurationsmanagement für Infrastruktur und Anwendungen

Prinzipiell können Sie mit Tools wie *Ansible* oder *Puppet* sowohl Ihre Infrastruktur verwalten als auch die eigentliche Anwendung konfigurieren. Durch die immer weitere Verbreitung von Containern und Orchestrierungs- und Skalierungs-Tools wie *Kubernetes* wird zumindest die Anwendungskonfiguration weniger wichtig, da diese dann nicht mehr mit klassischem Konfigurationsmanagement erledigt wird. Für die Konfiguration des Basis-Betriebssystems sieht das allerdings wieder anders aus: Hier werden Tools wie Puppet und Ansible weiterhin benötigt.

9.5.2 Ansible vs. Puppet

Es gibt zwei Tools im Rahmen des Konfigurationsmanagements, die besonders verbreitet sind: Ansible und Puppet. Natürlich gibt es noch etliche weitere, darunter auch Chef oder Saltstack. Da Ansible und Puppet allerdings die wohl verbreitetsten Tools sind, will ich mich auf diese beiden konzentrieren.

Ansible

Ansible ist heutzutage das wohl meistgenutzte Werkzeug für das Konfigurationsmanagement; die meisten IaC-Projekte bestehen inzwischen darauf. Dafür gibt es gute Gründe, aber nicht alles ist super schön und einfach mit Ansible. Aber der Reihe nach!

Ansible-Buch

Auch zu Ansible gibt es beim Rheinwerk Verlag ein umfangreiches Buch:

https://www.rheinwerk-verlag.de/ansible-das-praxisbuch-fuer-administratoren-und-devops-teams/

Ansible steht unter eine Open-Source-Lizenz, ist in Python geschrieben und existiert seit 2012. Im Jahr 2015 wurde die Firma hinter Ansible von der Firma Red Hat aufgekauft, die es seitdem in ihr Portfolio integriert hat.

Wenn man sich das Konzept von Ansible anschaut, dann sollte schnell klar sein, warum es so schnell erfolgreich wurde und weite Verbreitung gefunden hat. Ansible basiert auf einer sehr einfachen Architektur (siehe Abbildung 9.8), bei der sich alles um die *Playbooks* dreht, die sich gut verwalten und versionieren lassen.

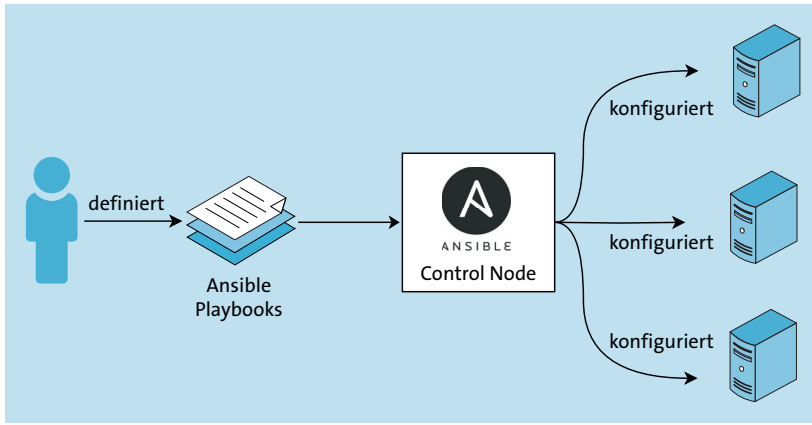

Abbildung 9.8 Über den »Ansible Control Node« werden die Server konfiguriert.

Anweisungen, die auf den Systemen ausgeführt werden sollen, werden in den Playbooks in YAML-Code beschrieben. YAML ist grundsätzlich sehr einfach zu lesen, wenn aber zu viele Einrückungen nötig sind, geht dennoch schnell der Überblick verloren. In den Playbooks findet sich eine Sammlung von Tasks, die in YAML definiert werden. Wenn solche Playbooks ausgeführt werden, dann werden diese Tasks von oben nach unten abgearbeitet.

```
- name: Install apache httpd
    ansible.builtin.apt:
        name: apache2
        state: present
        update_cache: yes

- name: Start service httpd, if not started
    ansible.builtin.service:
        name: apache2
        state: started
```

Listing 9.1 Einen Webserver mit Ansible installieren und starten

In dem Beispiel aus Listing 9.1 sind etwa zwei Tasks definiert. Im ersten Task wird das `apt`-Modul von Ansible verwendet, um das Paket `apache2` zu installieren. Oder genauer gesagt: Es soll sichergestellt werden, dass das Paket installiert ist. Das entspricht der deklarativen Natur von Ansible. Es wird immer der gewünschte Zustand beschrieben, und Ansible kümmert sich dann darum, dass dieser erreicht wird.

Im zweiten Schritt wird dann ein weiterer Task definiert, der sicherstellt, dass der Dienst entsprechend gestartet ist. Wie bereits erwähnt, wird von oben nach unten gearbeitet.

© Rheinwerk Verlag, Bonn 2024

Das klingt zunächst wie ein Skript, und genau so lässt es sich auch auf einem abstrakten Level beschreiben: Ein Playbook ist eine Sammlung von YAML-Schnipseln, die jedoch schöner zu schreiben und lesen sind. Wichtig ist, dass Ansible die Aufträge linear von oben nach unten abarbeitet.

Ansible lässt sich sehr einfach nutzen, es muss nur Python auf dem Controller und den Zielsystemen installiert sein. Ansonsten wird nur noch ein SSH-Zugang benötigt. Damit lässt sich zügig ein System erreichen, um die Ansible-Playbooks auszuführen.

Ansible arbeitet nach dem Push-Prinzip: Vom Controller-Node werden SSH-Verbindungen zu den Zielsystemen aufgebaut, um dann die Abfolge der Tasks auszuführen. Dadurch ist die Infrastruktur für die Nutzung von Ansible schön schlank gehalten.

Das Ganze hat allerdings auch ein paar Nachteile. Dafür ist erst mal wichtig zu verstehen, was *Idempotenz* in diesem Kontext heißt. Grundsätzlich sind die meisten Tasks in Ansible idempotent. Beispielsweise lässt sich definieren, dass eine Konfigurationsdatei auf die Server hochgeladen werden soll. Wenn man das mehrfach ausführt, liegt sie genau einmal da.

Dadurch, dass die Tasks immer nacheinander ausgeführt werden, kann das allerdings dazu führen, dass mehrfach verschiedene Konfigurationsdateien hochgeladen werden, die einander überschreiben. Bei einfachen Setups ist das kein Problem; bei einem komplexen Setup kann das allerdings komplizierter werden, da man dann genau schauen muss, wie und mit welchen Bedingungen die Datei hochgeladen wird.

Ein weiterer Nachteil des Verfahrens ist die Geschwindigkeit. Bei jedem Task muss geprüft werden, ob eine Änderung nötig ist; erst dann wird diese ausgeführt – oder eben nicht. Eine Parallelisierung ist nicht so einfach möglich, da das Playbook schließlich von oben nach unten ausgeführt wird.

Playbooks für viele Server und mit unterschiedlichen Diensten, die bespielt werden müssen, sind dann nicht nur komplex zu schreiben, sondern es dauert auch eine ganze Weile, bis der Durchlauf von Ansible abgeschlossen ist. Das sind dann häufig gute Zeitpunkte, um Kaffee kochen zu gehen.

Wenn man Pech hat, kommt kurz vor Ende eine Fehlermeldung, dass etwas nicht geklappt hat und der Durchlauf bricht ab. Solche Fehler muss man entsprechend so umschiffen, dass diese nicht zu einem fehlerhaften System führen. Das Ausführen mit korrigiertem Code muss man dann allerdings nochmals angehen, wobei man schon durch die Nutzung von Tags einige Schritte überspringen kann.

Puppet

Eine weitverbreitete Alternative zu Ansible ist Puppet, wobei Puppets Popularität in den letzten Jahren deutlich abgenommen hat. Puppet ist in Ruby geschrieben, dementsprechend sind viele Sprachkomponenten nah an Ruby angelehnt.

Persönliches Exemplar für Thomas Bartholomäus

Im Gegensatz zu Ansible arbeitet Puppet nach dem Pull-Prinzip. Auf den zu verwaltenden Servern muss der Puppet-Agent installiert sein, der sich die Konfigurationen in regelmäßigen Abständen abholt und auf den Servern ausrollt (siehe Abbildung 9.9). Der Agent muss also vorher provisioniert werden, was zusätzliches Tooling benötigt.

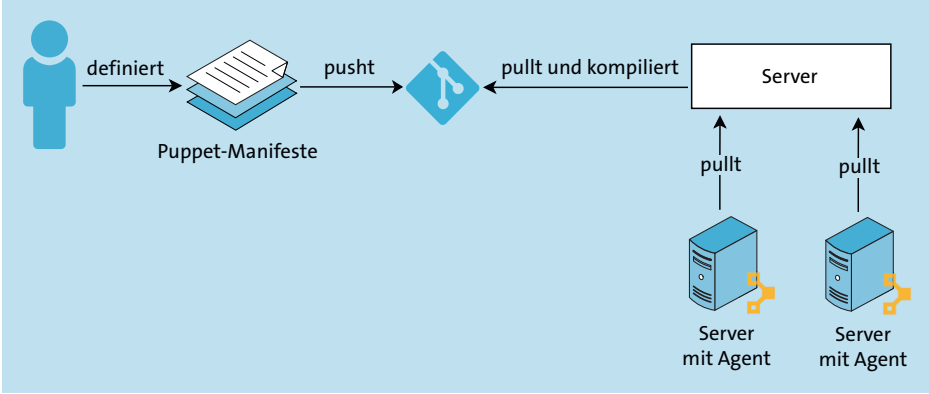

Abbildung 9.9 Der prinzipielle Aufbau bei Puppet: Die Server pullen die Konfigurationen vom Puppet-Server, der diese aus dem Git-Repository zieht und kompiliert.

Running Gag

Einen Running Gag bei der Diskussion, ob nun Puppet oder Ansible verwendet werden sollte, hört man immer wieder: »Ansible ist praktisch, um den Puppet-Agent auszurollen, da Puppet das selbst nicht kann.«

Der Einstieg über Ansible ist ziemlich einfach, was man bei Puppet nicht so sagen kann.

Was bei Ansible die Playbooks sind, sind bei Puppet die *Manifeste*. In den Manifesten wird in einer eigenen Sprache deklarativ der Zielzustand der Systeme definiert. Diese Sprache ist an Ruby angelehnt. Gleiches trifft auch auf die Sprache für Templates zu, wobei es mit EPP (*Embedded Puppet*) und ERB (*Embedded Ruby*) zwei Dialekte gibt.

```
# execute 'apt-get update'
exec { 'apt-update':
  command => '/usr/bin/apt-get update'
}

# install apache2 package
package { 'apache2':
  require => Exec['apt-update'],
  ensure => installed,
}
```


© Rheinwerk Verlag, Bonn 2024

```
# ensure apache2 service is running
service { 'apache2':
  ensure => running,
}
```

Listing 9.2 Einen Webserver mit Puppet ausrollen

Das Beispiel aus Listing 9.2 macht drei Dinge:

- Im ersten Block wird definiert, dass `apt-get update` ausgeführt werden kann. Da hier nicht linear von oben nach unten gearbeitet wird, wird diese Bedingung nur definiert und mit dem entsprechenden Kommando verknüpft.
- Im zweiten Block wird dann definiert, dass sichergestellt werden soll, dass das Paket `apache2` installiert ist. Dafür wird allerdings benötigt, dass die Paketquellen vorher aktualisiert werden.
- Der dritte Block ist selbsterklärend: Es wird sichergestellt, dass der Dienst `apache2` läuft.

Der Vorteil der Sprache für die Definition der Manifeste ist, dass die Gesamtheit der Manifeste in der Regel nur dann funktioniert, wenn die Manifeste auch syntaktisch korrekt implementiert wurden. Wie im angeführten Beispiel werden etwa die Abhängigkeiten der Ressourcen definiert, sodass Puppet weiß, in welcher Reihenfolge die Aufgaben ausgeführt werden müssen.

Während bei Ansible das Playbook von oben nach unten abgearbeitet wird, prüft Puppet jedes Mal, ob die Manifeste valide sind. Ist das nicht der Fall, werden keine Änderungen ausgerollt. Ein Vorteil dieses Verhaltens ist, dass keine halbgaren Schritte ausgeführt werden, wie es bei Ansible der Fall sein kann. Puppet arbeitet gerade bei komplexen, mehrstufigen Rollouts schneller als Ansible, und da ein Auftrag nicht in der Mitte fehlschlagen kann, braucht man deutlich weniger Anläufe, bis alles funktioniert.

Ein weiterer Vorteil ist, dass es ein großes Ökosystem an Modulen gibt, die Puppet nutzen kann. So etwas Ähnliches gibt es zwar in Ansible auch, allerdings wird dort stärker zwischen der Definition der Manifeste und den Daten getrennt, mit denen die die Konfigurationen ausgerollt werden.

Wenn Sie Puppet einsetzen wollen, brauchen Sie ein *Monitoring*. Da es im Pull-Prinzip läuft, muss sichergestellt werden, dass die Manifeste überall ordnungsgemäß ausgerollt werden. Wenn die Rollouts nicht durch ein Monitoring-Tool überprüft werden, kann es gut passieren, dass nicht auffällt, wenn es Fehler bei den Konfigurationsänderungen gibt.

Die Vorteile klingen gut, aber es gibt einige Gründe, warum Puppet an Popularität verloren hat. Während Sie mit Ansible fast ohne Voraussetzungen loslegen können, müssen erst recht hohe Hürden überwunden werden, bis die Automatisierung von

Puppet läuft. Der zusätzliche Aufwand sorgt für mehr Komplexität, die noch dadurch gesteigert wird, dass für die Manifeste eine eigene Sprache gelernt werden muss.

Ich würde daher gerade für den Einstieg und für einfachere Aufgaben zu Ansible greifen. Falls Ihre Infrastruktur dann so gewachsen ist, dass die Playbooks zu unübersichtlich werden, können Sie sich immer noch Puppet anschauen.

Fazit

Neben Ansible und Puppet gibt es noch eine Vielzahl anderer Tools für das Konfigurationsmanagement. Bekannteste Vertreter sind *CFEngine*, *Saltstack* und *Chef*. Für das Endresultat ist das konkrete Tool nicht ausschlaggebend, alle haben ihre Vor- und Nachteile, ein bisschen ist die Wahl auch Geschmackssache.

Wichtig ist: Wenn Sie ein Tool für das Konfigurationsmanagement einsetzen, dann tun Sie das so konsequent wie möglich. Je gleichförmiger und standardisierter die Konfigurationen sind, desto besser. Und wenn alles als Code vorliegt, lassen sich die typischen Entwickler-Workflows auf die Infrastruktur anwenden. Änderungen und Updates werden durch Code-Reviews überprüft und abgesichert. Bei einem Fehler kann durch ein Rollback die alte Konfiguration wiederhergestellt werden.

Reflexion

Sowohl Puppet als auch Ansible ermöglichen es Ihnen, die Konfiguration der Systeme und Anwendungen als Code abzuspeichern. Vor allem Ansible bietet sich an, wenn Sie schnell wiederkehrende Aufgaben auf Systemen automatisieren wollen. Das geht auch ganz gut, wenn Sie noch nicht auf vollständige Automatisierungen der Infrastruktur in der Cloud setzen.

Eine Möglichkeit, um zu prüfen, ob das Konfigurationsmanagement sauber umgesetzt wurde, ist die Nutzung von SSH. Während es für einen klassischen Systemadministrator üblich war, sich per SSH auf einem System einzuloggen, um eine Konfiguration anzupassen, sollte jetzt eher eine Änderung im Ansible-Code im Repository geschehen. Je weniger SSH verwendet wird, um etwas nachzuschauen oder anzupassen, desto näher kommt man der vollständigen Automatisierung.

9.6 Chaos-Engineering

Wenn Ihr Team bereits nach DevOps-Prinzipien arbeitet und Sie testen wollen, wie gut Ihre Arbeitsabläufe sind, dann sollten Sie sich das *Chaos-Engineering* anschauen.

Grundsätzlich ist das Ziel ja, dass Software korrekt, zuverlässig und sicher läuft. Und zwar immer, auch unter schwierigen Bedingungen. Was man aber stattdessen oft sieht, ist das Motto: »Es wird schon irgendwie funktionieren, schließlich haben wir bis hierhin ja überlebt.«

© Rheinwerk Verlag, Bonn 2024

Das bedeutet nichts anderes, als dass nur gehofft wird, statt aktiv zu schauen, ob alles so funktioniert, wie es sein sollte. Im Verlauf dieses Buches habe ich schon viele Aspekte genannt, die für die Verbesserungen der Qualität sorgen. Das sind etwa Code-Reviews beim Programmieren (wie in Kapitel 5 betrachtet) oder eben die Test-Pyramide mit den verschiedenen Arten von Tests, die in Kapitel 7 beschrieben wurde. Auch Deployments auf Review- und Staging-Umgebungen decken Fehler frühzeitig auf.

9.6.1 Systeme zum Ausfall bringen

All diese wichtigen Bestandteile des Entwicklungsprozesses sind gut und notwendig. Trotzdem kann man noch einen Schritt weiter gehen. Denn viele Systeme sind sehr komplex und es ist schwierig, wirklich alle Komponenten und alle möglichen Probleme richtig zu verstehen.

Insbesondere wollen Sie ja wissen, ob sichergestellt werden kann, dass der Dienst im Wesentlichen noch weiterhin funktioniert, wenn ein Teilsystem ausfällt. Vorwiegend beim Einsatz von Microservices werden die Gesamtsysteme ziemlich komplex. Aber wie arbeitet das System, wenn ein oder mehrere Teilsysteme ausfallen (aus welchen Gründen auch immer)?

Viele Firmen bauen zwar ihre Systeme hochverfügbar auf, testen diesen Aufbau allerdings kaum. Wenn es dann zu Ausfällen kommt, hört man immer wieder: »Das hätte eigentlich nicht passieren sollen!«

Beim Chaos-Engineering wird bewusst ein »Chaos« erzeugt, um Teile des Systems zu Fall zu bringen. Dazu erzeugt man ganz bewusst auf Produktivsystemen hohe Last und viel Stress, um zu schauen, wie sich das System nun verhält. Aus diesen Experimenten kann man dann viel lernen, um das gesamte System robuster zu machen, damit es bei einem tatsächlichen Ansturm zu weniger Problemen kommt.

Netflix hat mit diesem Ansatz schon früh begonnen. Netflix entwickelte dafür das Tool *Chaos Monkey* (*https://github.com/Netflix/chaosmonkey*). Eines der Kernelemente ist, dass es wahllos mit virtuellen Kabeln spielt und Server ausschaltet oder virtuelle Maschinen und Container aus der Produktivumgebung wirft. Ihr System sollte so gebaut sein, dass es trotzdem funktioniert.

Neben Chaos Monkey gibt es noch das *Chaos Toolkit* (*https://chaostoolkit.org/*) sowie im Cloud-native-Bereich auch *Chaos Mesh* (*https://chaos-mesh.org/*) und *Litmus Chaos* (*https://litmuschaos.io/*). Mehr Informationen finden Sie auf der Webseite *o11y.love*:

https://o11y.love/topics/chaos-engineering/

Obwohl im Namen »Chaos-Engineering« das Wort »Chaos« steckt, ist der Engineering-Teil alles andere als chaotisch. Es muss ordentlich geplant werden, damit das Ganze auch kontrolliert durchgeführt wird. Denn nur so kann man ordentlich etwas dazulernen.

Es gibt verschiedene Ebenen, die getestet werden. Das sind im Wesentlichen die Infrastruktur, das Netzwerk sowie die Anwendung. Auf jeder Ebene werden Tests durchgeführt; beim Testen der Infrastruktur werden beispielsweise zufällig Server ausgeschaltet oder eine hohe Last erzeugt, um zu schauen, was dann passiert.

Beim Netzwerk-Test wird die Netzwerkbandbreite zwischen diversen Systemen gedrosselt oder es werden zufällig Pakete gedroppt. Beides sind Fälle, die häufig auftreten können, daher sollten Sie sie kontrolliert in der Produktivumgebung testen. Erweiterte Chaos-Experimente erlauben auch überschriebene DNS- und HTTP-Antworten, um zu evaluieren, wie sich die Applikation im Fehlerfall verhält.

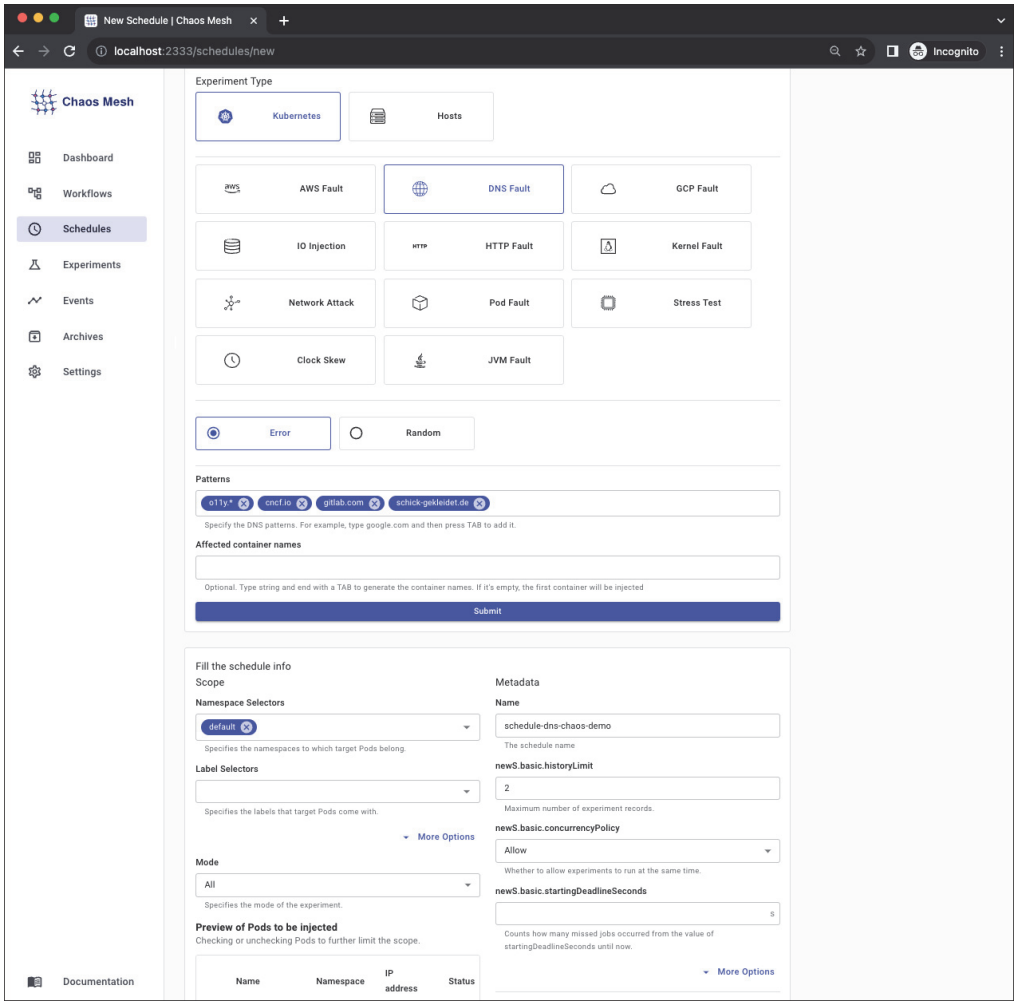

Abbildung 9.10 Wiederkehrende Experimente in der »Chaos Mesh«-UI, um DNS von bestimmten Domains zu stören

© Rheinwerk Verlag, Bonn 2024

Bleibt zum Schluss noch die Anwendung. Auch diese sollten Sie mit diversen »chaotischen« Experimenten testen. Wie verhält sie sich unter Last oder was passiert, wenn eigentlich unsinnige Requests an sie gerichtet werden? Wenn dabei keine Fehler auftreten und die Anwendung für den Endkunden weiterhin benutzbar bleibt, wissen Sie, dass Sie und Ihr Team vieles richtig machen.

Chaos bei Pokémon Go

Spannend in diesem Zusammenhang fand ich einen Blogpost von Google Cloud zum Launch von Pokémon Go:

https://cloud.google.com/blog/products/containers-kubernetes/bringing-pokemon-go-to-life-on-google-cloud?hl=en

Die Engineers von Google berichten darin, dass sie als Worst-Case eine Überschreitung des geplanten Traffics um den Faktor 5 angenommen hatten. Tatsächlich kam es aber so, dass 50-mal so hoher Traffic wie erwartet auf die Pokemon-App einprasselte. Direkt zum Launch funktionierte vieles nicht, sodass sie einige lastintensive Teile des Spiels kurzfristig deaktivieren mussten.

Ob die Google-Leute damals schon Chaos-Engineering genutzt haben und ob Sie auch einen so hohen Traffic getestet hätten, weiß ich nicht. Das Beispiel zeigt aber, wie hilfreich solche Tests sein können. Natürlich kann man nicht alles testen – das ist nun mal so –, aber ein bisschen Chaos im Vorfeld sorgt dafür, dass man dem Launch gelassener entgegensehen kann und mehr Vertrauen in das eigene Produkt entwickelt.

9.6.2 Chaos-Engineering ohne Chaos, aber mit Plan

Damit man beim Durchführen von Chaos-Engineering nicht im Chaos versinkt, sollte man mit einem konkreten Plan arbeiten, damit gezielt vorgegangen werden kann und die Ergebnisse auch messbar sind. Die Website *Principles of Chaos* (*https://principlesofchaos.org/*) gibt für die Praxiseinsätze vier Schritte an (siehe Abbildung 9.11):

1. **Definition des Status**: Es muss eine Definition des stabilen Systems erfolgen, in der das normale Verhalten erwartet wird.
2. **Definition der Hypothese:** Es muss eine Hypothese aufgestellt werden, in der erwartet wird, dass der stabile Zustand sowohl in der Kontrollgruppe als auch in der Versuchsgruppe weiterhin besteht.
3. **Anpassung von Variablen:** Es sollten nun echte Ereignisse implementiert werden, die den Zustand des Systems durcheinanderbringen. Letztlich sollen also Fehler injiziert werden. Das sind beispielsweise ausfallende Server und Festplatten, nicht mehr richtig funktionierende Netzwerkverbindungen oder eben völlig überlastete Teilsysteme.

Persönliches Exemplar für Thomas Bartholomäus

4. **Widerlegung der Hypothese:** Jetzt muss verglichen werden, wie sich das System verändert hat, und das Verhalten anhand von Metriken die Kennzahlen ausgewertet werden.

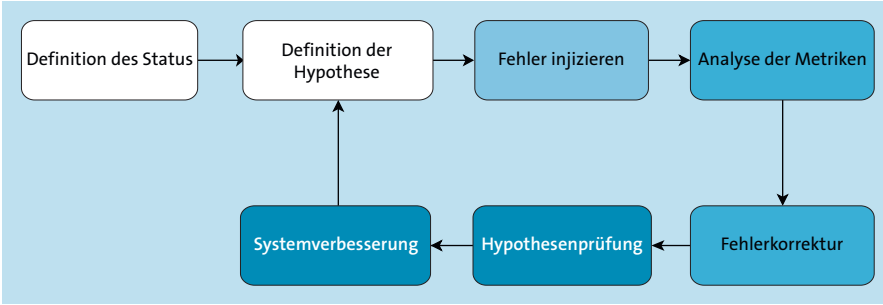

Abbildung 9.11 Chaos-Engineering Schritt für Schritt: gar nicht so viel Chaos

Anschließend können dann nächste Schritte definiert werden. Das Prinzip von Chaos-Engineering geht davon aus, dass man den Test auf Produktivumgebungen ausführt. Das wird man vermutlich nicht direkt im ersten Schritt wagen, sondern zunächst vorsichtig in Staging-Umgebungen arbeiten.

Der Hintergrund des Ganzen ist ja, dass man das ganze System vollständig verstehen sollte. Probleme lassen sich dann in diese vier Kategorien einteilen:

1. **Bekanntes Wissen** (*known Knowns*) – Das sind Dinge, die in der Organisation bekannt sind und die die Mitarbeiter verstehen.
2. **Bekanntes Unwisssen** (known Unknowns) – Dazu gehören die Teile, von denen bekannt ist, dass die Mitarbeiter sie nicht vollständig verstehen.
3. **Unbekanntes Wissen** (unknown Knowns) – Das sind wiederum die Teile, die die Organisation zwar versteht; sie weiß aber nicht, dass hier Probleme lauern.
4. **Unbekanntes Unwissen** (unknown Unknowns) – Und zum Schluss gibt es noch die Dinge, die der Organisation weder bewusst sind noch vollständig verstanden werden.

Ganz unabhängig vom Chaos-Engineering sollte möglichst viel unter »Bekanntes Wissen« fallen. Bei den anderen drei Fällen gilt es, ihren Einfluss so gut es geht zu minimieren. Dabei kann Chaos-Engineering helfen.

Ein Vorteil ist daher, dass durch das Chaos-Engineering die Robustheit des ganzen Systems erhöht werden kann. Dazu muss man sich aber auch trauen, es auf Produktivsystemen auszuführen. Auch mit dem richtigen Monitoring und den richtigen Metriken sollte man vielleicht eher die Zeit wählen, zu der ohnehin wenig Last auf den Systemen ist, um Ausfallrisiken zu minimieren. Was nämlich häufig ignoriert wird, ist der Umstand, dass bei kontrollierten Ausfällen, wie es hier der Fall ist, die

© Rheinwerk Verlag, Bonn 2024

Fehler viel besser, schneller und günstiger korrigiert werden können, als wenn Fehler bei hoher Last mit ausschließlich echten Nutzern auftreten.

So erhalten die Teams ein deutlich besseres Verständnis davon, wie das gesamte System funktioniert, wo es eventuell Probleme geben kann und wie man diese korrigieren kann, um tatsächlich Probleme rechtzeitig zu finden – vor allem dann, wenn diese nicht ganz so offensichtlich sind.

Sobald man diese Tests einmal oder mehrfach durchgeführt hat, kann man das Ganze auch automatisieren und regelmäßig ausführen, um die Zuverlässigkeit stetig zu erhöhen.

Wie so häufig gilt auch hier: Die Umsetzung funktioniert nur, wenn Führungskräfte solche Experimente zulassen. Haben diese Personen vor einem solchen Test zu viel Angst, zeugt das davon, dass sie generell der Produktivumgebung nicht vertrauen. Aber gerade dann sollte man erst recht testen, um im Ernstfall nicht vor einem großen Problem zu stehen!

Reflexion

Chaos-Engineering verhält sich im Betrieb der Anwendung und der Infrastruktur wie die Blue-Green-Deployments und Feature-Flags aus Abschnitt 8.4: Es sind Arbeitsweisen, die Sie nur ausführen können, wenn Sie volles Vertrauen in Ihr Team und Ihre Anwendung haben.

Im ersten Schritt sollten Sie mit Chaos-Engineering-Methoden auf Staging-Umgebungen experimentieren, bevor Sie damit auf Produktionsumgebungen gehen. Schon damit erreichen Sie viele Einsichten, selbst wenn in Ihrer Organisation das Ausführen auf Produktionsumgebungen noch als zu gefährlich betrachtet wird.

9.7 Reliability Engineering

Eine weitere Rolle im DevOps-Umfeld ist das Reliability Engineering, die allerdings nur recht schwammig definiert ist und sich zudem noch in verschiedene Bereiche aufspaltet. Die wohl bekannteste Rolle ist der *Site Reliability Engineer* (SRE), die maßgeblich von Google entwickelt und geprägt wurde.

9.7.1 Site Reliability Engineering

Eine richtig klare Definition des *Site Reliability Engineers* gab es auch bei Google nicht, denn je nach Team und je nach Produkt sah die Rolle dort anders aus. Die wesentlichen Prinzipien lassen sich jedoch allgemeingültig zusammenfassen, denn nicht ohne Grund hat sich die Art der Rolle im Laufe der Jahre auch außerhalb von Google weit verbreitet.

Persönliches Exemplar für Thomas Bartholomäus

Bei SREs handelt es sich um *Engineers*, die große verteilte Systeme entwickeln und betreuen. Ein *Site Reliability Engineer* kümmert sich um sehr viele Aspekte, die sowohl beim Software-Engineering als auch beim Systems-Engineering anfallen, mit dem Fokus auf der *Reliability* (Zuverlässigkeit) der Anwendung und des ganzen Systems.

Das klingt jetzt zunächst nur nach einem »fancy« Begriff für die alte Rolle des Systemadministrators, die Details sind jedoch spannend: SREs arbeiten eng mit den Entwicklungsteams zusammen, um die Systeme in den Bereichen Fehleranfälligkeit und Skalierbarkeit zu stärken. Sie unterstützen zudem die Teams bei der Implementierung von Monitoring, Tests und bei Deployments, insbesondere in Hinblick auf die Stabilität im Betrieb.

Ob man den Begriff SRE nun verwendet oder nicht: Google ist es damit gelungen, die Verfügbarkeit der Anwendung in den Mittelpunkt zu stellen, was für die Kunden ja der zentrale Punkt ist. Für sie ist es unerheblich, ob Google Maps nicht funktioniert, weil eine Entwicklerin ein fehlerhaftes Feature deployt hat oder ob Gmail keine Mails zeigt, weil ein Admin etwas falsch gemacht hat. Für die Kunden zählt nur, *dass* es funktioniert, und dafür sind alle verantwortlich.

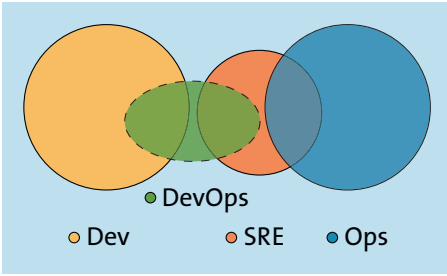

Abbildung 9.12 DevOps-SRE: Die Tätigkeiten von SRE und Ops überlappen sich teilweise, rücken aber deutlich näher an das Dev-Team heran als in klassischen Silos.

Die Tätigkeiten von SREs liegen also zwischen Dev und Ops (siehe Abbildung 9.12). Dabei gibt es primär Überlappungen zwischen SREs und dem Ops-Team. Bei dieser Art der Zusammenarbeit übernehmen SREs mehr Verantwortung für den Betrieb der Anwendung und haben einen viel stärkeren Entwicklungshintergrund, als es bei klassischen Administratoren aus dem Ops-Team der Fall ist. Das führte dazu, dass Aspekte wie das Testen, die in der Entwicklung schon immer wichtig waren, unter neuen Begriffen wie *Fault Injection* und *Chaos-Engineering* (siehe Abschnitt 9.6) auch für den Betrieb etabliert werden.

Durch diese Art der Zusammenarbeit übernehmen die SREs auch den Bereitschaftsdienst für die Anwendung. Durch die starke Nähe zum Software-Engineering mit Fokus auf Reliability müssen die reinen Entwickler nicht unbedingt Bereitschaftsdienste mit übernehmen.

© Rheinwerk Verlag, Bonn 2024

Im Falle von Incidents sind die SREs dafür verantwortlich, dass Incidents behoben werden, inklusive Debugging und Troubleshooting; und sie sind die zentrale Kommunikationsbrücke zwischen den verschiedenen Teams, um die benötigten Korrekturen einzubringen und zu deployen. SREs sollen also die Brücke zwischen Dev und Ops schlagen, indem sie einen starken Software-Engineering-Fokus haben, also auch mal programmieren können, sich aber hauptsächlich um die Stabilität und Skalierbarkeit der Anwendung kümmern.

Bücher zum Site Reliability Engineering

Unter *https://sre.google/books* finden Sie diverse Bücher von Google, in denen beschrieben wird, wie die SREs dort arbeiten. In ihnen findet man sehr viele interessante Einblicke.

Vergessen Sie aber nicht, dass Google ein verdammt großer Laden ist, der fast ausnahmslos alle Dienste, die betrieben werden, in sehr hohem Maße skalieren muss. Die Wahrscheinlichkeit ist hoch, dass Ihr Unternehmen nicht mit den gleichen Herausforderungen zu kämpfen hat. Trotzdem lassen sich viele Arbeitsweisen adaptieren und viele Ideen sind auch in kleineren Umgebungen hilfreich.

9.7.2 Database-Reliability-Engineering

Database Reliability Engineers (DBREs) arbeiten nach einem ähnlichen Prinzip, nur mit dem anderen Hintergrund, dass es bei ihrer Tätigkeit rein um die Datenbank für die Anwendung geht und nicht um die Anwendung selbst.

Auch hier ist eine enge Zusammenarbeit erforderlich, damit Performance-Probleme, die von den DBREs identifiziert werden, auch von den Entwicklern entsprechend korrigiert werden.

Hier werden ebenfalls gute Software-Engineering-Kenntnisse benötigt, damit die Brücke zwischen den Datenbankabfragen, die von den Entwicklern geschrieben werden, und der Datenbank selbst effizient geschlagen wird.

Nicht verwunderlich: Database Reliability Engineers müssen die Anwendung, die sie mitbetreuen, ebenfalls gut kennen, und zwar auf der Software-Engineering-Basis. Sie mögen vielleicht nicht die Anwendung selbst weiterentwickeln, geben aber dem Entwicklungsteam wertvolle Unterstützung, damit die Anwendung mit der Datenbank zuverlässig läuft und entsprechend auch skalieren kann.

Daher gilt auch hier: Kollaboration und Kultur sind wichtig! Wie die technische Implementierung aussieht, ist grundsätzlich erst mal zweitrangig.

Persönliches Exemplar für Thomas Bartholomäus

9.8 Fazit

In diesem Kapitel wurde behandelt, wie die Zusammenarbeit im Betrieb der Anwendung erfolgen kann und welche Vorteile es gibt, wenn Dev und Ops sehr viel enger zusammenarbeiten.

Nah verwandt mit dem Betrieb der Anwendung ist allerdings auch noch das Thema rund um das Monitoring der Systemlandschaft, mit dem wir uns in Kapitel 10 befassen.

© Rheinwerk Verlag, Bonn 2024

Kapitel 10
Vom Monitoring zur Observability

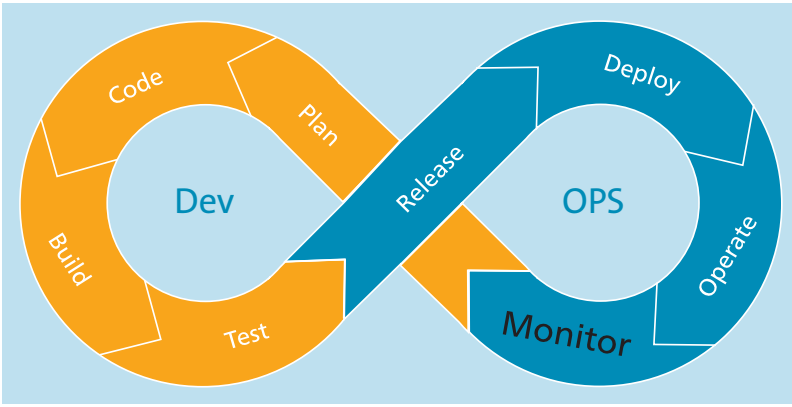

Abbildung 10.1 Monitor: Die Software und die Infrastruktur werden überwacht, um Probleme zu erkennen.

Der letzte Schritt in der DevOps-Schleife ist das Monitoring. Nach dem Monitoring geht es direkt wieder mit der Planung weiter, schließlich sollen die Erkenntnisse aus dem Monitoring ja für die fortlaufende Entwicklung genutzt werden.

Im Hinblick auf die Drei Wege ist das eines der Kernelemente des zweiten und dritten Weges: der Flow von Ops zu Dev und somit vom Endnutzer zum Business. Hier geht es also im Wesentlichen darum, was wir von den Endnutzern bei jeder Iteration lernen können, um stetig neue Verbesserungen zu implementieren und auszurollen.

Auch in Hinblick auf das CALMS-Modell liegt in diesem Kapitel der Fokus primär auf dem **M** für *Metriken*. Aber wie so häufig in DevOps spielt ohnehin alles zusammen, und so gibt es hier auch den Fokus auf das *Sharing* dessen, was man aus dem Monitoring und der Observability gelernt hat. Auch das ist ebenfalls ziemlich wichtig. Dass Automatisierungen hier ebenfalls starke Einflüsse haben, sollte an dieser Stelle des Buches wohl niemanden mehr überraschen.

Monitoring ist ein typisches Thema, von dem die meisten Leute in der klassischen Software-Entwicklung nicht viel mitbekommen haben, da man dies bislang als ein Thema angesehen hat, das nur das Betriebsteam betrifft. Wie gut die Performance der Anwendung ist oder was für Probleme über das Monitoring sichtbar werden, blieb dem Dev-Team verborgen, denn das war schließlich nicht seine Aufgabe.

Persönliches Exemplar für Thomas Bartholomäus

Wir wollen es besser machen und in diesem Kapitel zeigen, wie Monitoring ein wichtiges Bindeglied zwischen Dev und Ops werden kann. Relevant sind das Loggen von Ereignissen, Zuständen und anderen nützlichen Daten über die Dienste, um Aussagen über die Verfügbarkeit und die Performance machen zu können. Wir wollen aber über dieses klassische Monitoring hinausgehen und einen Blick auf die *Observability* werfen.

Dieses Kapitel befasst sich nicht nur mit dem reinen Blick auf die Anwendung und die Infrastruktur. Es lohnt sich nämlich auch ein Blick darauf, was DevOps gebracht hat und welche Metriken und Analyse-Möglichkeiten hierfür existieren.

10.1 Keine Sichtbarkeit bei schick-gekleidet.de

In den bisherigen Kapiteln wurde beschrieben, wie der Online-Shop von *schick-gekleidet.de* über einen langen Zeitraum entwickelt, getestet und später dann auch deployt wurde. Sobald das Betriebsteam den neu entwickelten Code für das Deployment übernahm, ging die Verantwortung für den Betrieb komplett an das Betriebsteam über. Dazu gehörte auch das Monitoring.

10.1.1 Ausfälle der Dienste sind von der Tagesordnung

Aber wozu gibt es eigentlich Monitoring? Dafür gibt es mehrere Gründe, und alle zielen darauf ab, dass der Betrieb sichergestellt wird. Zunächst ist erst mal wichtig, dass überhaupt Probleme erkannt werden.

Zum Glück führen nicht alle Kleinigkeiten direkt zum Ausfall des Systems, aber trotzdem sollten Probleme möglichst frühzeitig erkannt werden, damit diese behoben werden können, bevor größere Schwierigkeiten aufttreten. Es geht also darum, einen ständigen Überblick über den Status des gesamten Systems zu haben.

Reaktive Symptombehandlung

Eine häufige Herausforderung beim klassischen Monitoring ist allerdings, dass nur Symptome sichtbar werden, d. h., dass man Probleme erst dann bemerkt, wenn diese schon aufgetreten sind. Das können größere oder kleinere Schwierigkeiten sein, und erfahrungsgemäß lässt sich von der Größe der Auswirkungen kaum auf das wirkliche Ausmaß der Ursache schließen: Eine falsche Code-Zeile oder ein kleiner Fehler im Deployment kann dafür sorgen, dass nichts mehr funktioniert.

Wir müssen also direkt die Frage stellen: »Was genau ist ein Problem?« Jedes Problem im System hat andere Auswirkungen. Einige Probleme sind »nur« Warnungen, die es bald zu korrigieren gilt, einige haben Auswirkungen auf die Endnutzer.

© Rheinwerk Verlag, Bonn 2024

Die folgenden Status haben sich in klassischen Service-Monitoring-Systemen als Norm entwickelt:

- `OK`: Das System arbeitet wie erwartet.
- `CRITICAL`: Es arbeitet *nicht*.
- `WARNING`: Es funktioniert, aber nicht so wie erwartet.
- `UNKNOWN`: Der Status ist unbekannt.

Als Faustregel hat sich die Definition etabliert, dass Probleme, die bei den Endnutzern auftreten, also komplette Ausfälle oder Teilausfälle bedeuten, unter `CRITICAL` fallen und schnellstens behoben werden sollten. Zudem sollte möglichst vermieden werden, dass Probleme überhaupt kritisch werden.

Die erste Aufgabe des Monitorings ist also, sofort auf kritische Probleme hinzuweisen, damit diese behoben werden. Es darf nicht passieren, dass man erst durch Fehlermeldungen der Nutzer davon erfährt, dass ein Dienst nicht funktioniert. `WARNING` sollte man auch nicht außer Acht lassen: Bleibt dieser Zustand bestehen, könnte später ein kritischer Zustand erreicht werden.

Wenige Fehlermeldungen von Endkunden sind noch zu viel

Wie viele gemeldete Probleme von End-Usern sind noch okay? Das lässt sich schwer beantworten, aber es gilt die Grundregel, dass nur die wenigsten Betroffenen tatsächlich Fehler melden. Wenn Sie also eine Handvoll Bug-Reports bekommen, dass mit Ihrem Shop etwas nicht stimmt, dann müssen Sie womöglich alles stehen und liegen lassen und sich auf die Suche nach dem Problem machen. Es ist sehr wahrscheinlich, dass dieses Problem zahlreiche Kunden betrifft, die es aber schlicht nicht melden und den Kauf einfach abbrechen.

Das Monitoring soll aber nicht nur dabei helfen, Probleme überhaupt zu erkennen. Die zweite Aufgabe ist, Sie beim Troubleshooting zu unterstützen, also Material für die Fehleranalyse zu liefern, damit die Störung möglichst detailliert beschrieben werden kann.

Und dabei hapert es bei *schick-gekleidet.de* schon mächtig. Zwar gab es ein rudimentäres Monitoring, wodurch das Betriebsteam erfuhr, ob der Dienst lief oder nicht. Dies bestand aber weitestgehend aus einem bloßen Ping, der sich darauf beschränkte, ob der Webshop erreichbar war. Wie der genaue Status war und welche Konsequenzen die aktuelle Auslastung für den allgemeinen Betrieb hatte, war zunächst nicht ersichtlich.

Das Hauptaugenmerk beim Monitoring von *schick-gekleidet.de* lag nur auf der Prüfung, ob die verschiedenen Dienste liefen. Das waren häufig nur binäre Meldungen: Entweder es geht oder es geht eben nicht. In letzterem Fall wurde alarmiert. Zusätz-

Persönliches Exemplar für Thomas Bartholomäus

lich existierten gängige Checks, die weniger auf die Anwendungen selbst schauten, sondern auf das darunterliegende Betriebssystem.

Typisch für solche Checks sind Filesystem-Checks: wie hoch etwa der Füllstand des Dateisystems ist, ob kritische und nicht kritische Aktualisierungen von Paketen eingespielt werden müssen oder ob etwa TLS-Zertifikate abgelaufen sind.

Zugriff auf das Monitoring hatte lediglich das Betriebsteam, einen Einblick für das Entwicklungsteam gab es hingegen nicht. Auch die Alarmierungen bei Meldungen im Monitoringsystem gingen demnach lediglich an das Betriebsteam.

Das bedeutet, dass jede Warnung im Monitoringsystem, eine händische Fehlersuche nach sich zog, weil nachgeschaut werden musste, was nun das eigentliche Problem war. Natürlich wäre es dann besser gewesen, wenn das Problem dann nicht nur temporär gefixt worden wäre, sondern eine echte Fehleranalyse erfolgt wäre, um dem strukturellen Problem auf die Spur zu kommen.

Ohne die richtigen Informationen und ohne das Wissen, wie die einzelnen Teile des Systems zusammenhängen, war es den Admins in den meisten Fällen allerdings nur möglich, die Symptome zu beheben, anstatt die Wurzel des Problems anzupacken. Das führte selbstverständlich dazu, dass die gleichen Probleme immer und immer wieder auftauchten – ein Problem der Silos zwischen den Teams.

Fehlende Visibilität

Es ist ein typisches Problem, das Admins, das Monitoring-Setup und die getrennten Teams für das gleiche Projekt betrifft: Es fehlt an Visibilität in alle Richtungen. Die Entwickler haben überwiegend keinen Zugriff auf das Monitoringsystem, und die Admins haben keinen Einblick in die Software-Entwicklung.

Durch die fehlende Visibilität und die Art und Weise, wie mit Monitoringsystemen gearbeitet wird, wird zwangsläufig immer nur reaktiv gehandelt. Probleme fallen häufig erst dann auf, wenn schon etwas schiefläuft. Dabei ist es dann schon viel zu spät.

Bei *schick-gekleidet.de* sieht man das immer wieder, da gab es nämlich regelmäßig Ausfälle des Online-Shops. Es kam zwar selten vor, dass der Shop komplett offline ging, aber immer wieder fielen Teilsysteme aus. Das hatte häufig zur Folge, dass die Kunden von Einschränkungen betroffen waren, da einige Funktionen nicht richtig reagierten.

Manchmal fiel etwa das Billing-System aus, was für einen Teil der Kunden keinen Einfluss hatte, da sie sich nur die Artikel angeschaut, aber nichts gekauft hatten. Für diejenigen, die allerdings einen Kauf tätigen wollten, ging dann leider nichts, und dementsprechend floss dann auch kein Geld in die Kassen und die Lager blieben weiterhin voll. Aus betriebswirtschaftlicher Sicht ist das der schlechtestmögliche Zustand, den es zu verhindern gilt.

© Rheinwerk Verlag, Bonn 2024

Wie? Indem individuelle Benchmarks und Zielvorgaben erarbeitet werden, die das Monitoringsystem überwachen muss. Lassen Sie sich nicht von einer Datenmenge in Sicherheit wiegen, wenn ein großer Teil dieser Daten für Sie gar nicht relevant oder zumindest nur für die weitere Fehlersuche interessant ist. Stattdessen muss klar definiert sein, welche Systeme, API-Endpunkte, Webseiten usw. zentral sind. Überlegen Sie sich, wie Sie ihre Funktion testen und überwachen können. Das Operationsteam, das für den reibungslosen Betrieb verantwortlich ist, wird hier die meisten Erfahrungswerte beisteuern können, aber viele subtile Überwachungskriterien werden aus den Fachabteilungen und von den Entwicklern kommen.

Wenn Sie so eine Anzahl von wichtigen Systemen gesammelt und priorisiert haben, können Sie das Monitoring viel zielgerichteter und effizienter betreiben und erreichen echte *Visibilität*.

Alarm nach dem Deployment

Die intensivste Zeit für die Admins und die anderen Teams war die Zeit rund um die (wenigen) Deployments. Die meisten Probleme passierten nach einer Änderung, ein großer Teil von ihnen ploppte auch im Monitoringsystem auf.

Einige Fehler fielen allerdings erst auf, wenn Kunden diese Fehler meldeten; sie ergänzten häufig gewissermaßen das Monitoringsystem. So ein Zustand führt natürlich zu abgebrochenen Käufen und verlorenen Kunden, die nicht bereit sind, Bugs während eines Einkaufs zu tolerieren.

In den vorherigen Kapiteln diente immer wieder das Problem mit dem volllaufenden Dateisystem als Beispiel. Das Problem wurde immer erst dann erkannt, nachdem es schon aufgetreten war und diverse Schwellenwerte erreicht wurden.

So rannte das Adminteam immer den Problemen hinterher und löschte alle auftretenden Feuer im System. Sinnvoller ist es aber offensichtlich, das Problem an der Wurzel zu fassen und sich schon proaktiv zu kümmern, wenn abzusehen ist, dass das Dateisystem in den nächsten Tagen volllaufen wird.

Shortcuts vermeiden

Eine alternative Lösungsmöglichkeit besteht darin, das Dateisystem zu vergrößern. Damit wird allerdings unter Umständen das Problem nur um ein paar Stunden oder Tage verlagert. Je nach Umgebung kann das zu zusätzlichen Kosten führen.

Die Frage, die sich hier stellt, lautet allerdings vielmehr: »Dürfen sie das eigentlich?« Wie sind denn eigentlich die Befugnisse und Freiheiten geregelt? Dafür fehlen meist das Verständnis und die Einblicke in die Anwendung.

Aber auch das ist eigentlich nur eine Behandlung des Symptoms. Noch besser wäre es, wenn analysiert würde, woher der große Platzbedarf kommt und welches System dafür verantwortlich ist, dass hier im großen Stil Daten auf die Festplatten geschrieben werden.

Persönliches Exemplar für Thomas Bartholomäus

Vielleicht ist es ja ein Fehler, dass so viele Daten produziert werden; vielleicht ließe sich das auch besser organisieren. Dafür müssten die Admins, die dafür zuständig sind, den Müll wegzubringen, aber wissen, woher dieser Datenhaufen überhaupt kommt.

Fehlende Feedback-Schleife

Stattdessen wurden bei *schick-gekleidet.de* durch die strikte Trennung von Entwicklungsteam und Betriebsteam die Verbesserungsmöglichkeiten und Probleme nicht diskutiert. Problemlösungen wurden als Workarounds von den Admins direkt implementiert, anstatt das Problem tatsächlich in der Software zu beheben.

Im Beispiel eines volllaufenden Dateisystems wurde der Speicher erweitert, jedoch der Grund nicht analysiert: Möglicherweise schreibt die Applikation zu viele Logdaten in der Form von langen Fehler-Stacktraces.

Das Entwicklungsteam bekam so keine Informationen vom Betrieb mit. Etwaige Probleme mit der Software blieben verborgen, da kein Zugriff auf das Monitoring und die Tickets aus dem Produktivsystem bestand.

Die Kommunikation zwischen den verschiedenen Teams lief nur über E-Mails und ein gemeinsames Ticketsystem, was nicht zu verwechseln ist mit den eigenen Ticketsystemen der Teams selbst, auf die die jeweils anderen Teams keinen Zugriff hatten. Die Fehleranalyse ging nur schleppend voran, weil alle Informationen häppchenweise im Frage-Antwort-Verkehr ermittelt wurden oder oftmals auch missinterpretiert wurden.

Unbearbeitete Alarmierungen

Weiteres Problem bei *schick-gekleidet.de* war, dass viele Probleme zwar im Monitoring aufploppten, aber häufig nicht bearbeitet wurden. Aktualisierungen der Abhängigkeiten und des Betriebssystems wurden etwa ignoriert.

Jedes Mal, wenn Security-relevante Aktualisierungen verfügbar waren, erschien zwar im Monitoringsystem eine entsprechende Meldung. Diese Hinweise wurden aber geflissentlich ignoriert, da man schon genug mit den gravierenderen Problemen zu kämpfen hatte, die für Ausfälle sorgten. Zusätzliche Änderungen durch Updates konnten die Admins dann bestimmt nicht gebrauchen.

Das Betriebsteam war also eigentlich ständig im Feuerwehr-Einsatz: An vielen Stellen musste immer ein Feuer gelöscht werden, irgendwo loderte es immer. So fehlte Zeit, um sich um Sicherheitsaktualisierungen zu kümmern, sodass diese wiederum das Monitoringsystem fluteten. Ein Überblick über die anderen Probleme wurde somit immer schwieriger, und andere Alarme gingen einfach unter.

© Rheinwerk Verlag, Bonn 2024

Das Problem ließe sich eigentlich relativ einfach lösen, indem man das Einspielen der Aktualisierungen kontrolliert automatisiert. Aber dafür fehlte die Zeit: Die ging nämlich für die vielen Feuerwehr-Einsätze drauf.

10.1.2 Performance, Performance!

Klassisches Service-Status-Monitoring bietet wenig Einblick in die eigentliche Performance des Systems oder des Service – oder in deren historischen Verlauf oder zukünftigen Trend. Aus diesem Anspruch heraus hat sich das *Metrik-Monitoring* entwickelt, das Metriken als sogenannte *Zeitreihen* abspeichert. Dabei werden der Name der Metrik, der Wert sowie zusätzliche Attribute (*Tags*) abgespeichert und können in Graphen in einem Zeitstrahl visualisiert werden.

Wenn man heutzutage von Monitoring-Tools spricht, sind Metriken immer mit inkludiert oder der Standard wie in *Prometheus*. Es gibt verschiedene Arten von Metriken, die hier vorwiegend im Betrieb der Anwendung relevant sind. Das sind *Counter*, *Gauges*, *Histogramme* und *Summarys*. Was es damit auf sich hat, wird in Abschnitt 10.3.2 näher behandelt, wo wir uns Metriken in Praxisbeispielen mit Prometheus näher ansehen.

Dass ein Monitoring, das nur erfasst, ob ein Dienst läuft, manchmal nutzlos ist, hatte sich sogar bis *schick-gekleidet.de* herumgesprochen. Die Admins sammelten auch alle möglichen System-Metriken, mit denen sie die Performance der Systeme erfassten. So gab es für jeden Server Dashboards, auf denen ersichtlich war, wie hoch gerade die Auslastung von CPU, Arbeitsspeicher und des Netzwerks war.

So konnte wunderbar abgelesen werden, dass es nach einem Deployment zu Performance-Problemen kam und die Auslastung des Webshops unangenehm hoch war. Im Monitoring gab es immer wieder Meldungen dazu, die sogar Alarme auslösten. Problematisch für das Betriebsteam war allerdings, dass es keine Ahnung hatte, ob die hohe Auslastung mit den letzten Änderungen zu tun hatte oder ob es – und das wäre der positive Fall – einfach mehr Andrang im Webshop gab.

Gab es also einfach mehr Besucher? Schließlich lief auch eine Marketing-Kampagne parallel zum neusten Deployment. Oder konnte es sein, dass die Performance des Shops rapide abgenommen hat? Oder, als dritte Alternative: Lag es an einem Mix aus beidem, d. h., gab es einen höheren Andrang, sodass nun Probleme auftauchten, die bei einer niedrigeren Auslastung ignoriert werden konnten? Hier stehen also etliche Fragezeichen.

Eigentlich könnte man dies gut analysieren, wenn es denn historische Daten gäbe, mit denen man die aktuellen Engpässe vergleichen könnte. Die Monitoring-Daten wurden allerdings nicht so lange aufbewahrt, und um einen Vergleichspunkt mit einer vorherigen Version zu haben, hätte man schon hinter das letzte große Deploy-

Persönliches Exemplar für Thomas Bartholomäus

ment schauen müssen. Allerdings lag das bereits ein halbes Jahr zurück – vollkommen undenkbar, dass aus diesem Datenwust noch sinnvolle Erkenntnisse abgeleitet werden können!

So konnte sich das Betriebsteam nur auf Erinnerungen und Erfahrungen verlassen, um zumindest grob schätzen zu können, ob der aktuelle Traffic auch schon früher zu viel Last führte. Wichtig wäre hier, dass man die verschiedenen Metriken isoliert voneinander betrachten könnte, um faktenbasiert Entscheidungen zu treffen und Änderungen zu implementieren.

Latenzen

Manchmal sind es aber auch ganz andere Probleme, die zu Problemen führen. So sind beispielsweise Latenzen Probleme, die mit klassischen Monitoring-Tools nicht unbedingt sofort ersichtlich sind. Das Problem an ihnen ist, dass alles zu funktionieren scheint, aber lange Antwortzeiten können den gesamten Prozess verzögern und an ganz anderen Stellen für Probleme sorgen.

Das Entwicklungsteam, das maßgeblich für die Performance der Anwendung verantwortlich war, konnte überhaupt nicht einschätzen, was das Problem sein könnte. Ohne Zugriff auf das Monitoring und Informationen über die Auslastung der Produktionsumgebung waren seine Versuche, die Performance der Webseite zu verbessern, trotz aller Bemühungen zum Scheitern verurteilt.

Wie so oft in diesem Buch geht es auch hier um die Transparenz der vorhandenen Informationen. Das gilt auch für das Monitoring, denn auch hierfür wäre es sinnvoll, wenn Zugriff auf die Systeme vorhanden ist. Im ersten Schritt gilt es, die Zusammenarbeit zu erhöhen, indem einfach und ohne hohe Hürden Zugriffe auf die Systeme zum Standard werden.

10.1.3 Logs

Eng verwandt mit dem Monitoring ist das *Logging*. Dabei werden Zustände und Events mit einem Zeitstempel und Hinweistext von der Applikation in das Terminal oder in eine Log-Datei geschrieben. Ein Beispiel ist das Logging von Programm-Start und -Ende, von Datenbankverbindungschecks oder von unerwarteten Fehlermeldungen, die die Admins sich ansehen sollten. Oftmals findet man auch mehr Debug-Informationen, zum Beispiel SQL-Statements oder HTTP-Abfragen, die Entwicklern das Leben leichter machen.

Logs sind eigentlich gut, denn je detaillierter protokolliert wird, was im System passiert, desto besser können Fehler untersucht werden. Sie sind ein elementarer Bestandteil beim Debuggen von Problemen, und sie gehören zu gutem Monitoring dazu.

© Rheinwerk Verlag, Bonn 2024

Zu viel des Guten schadet aber: Wir haben in einem Beispiel bereits gesehen, dass die Logs dafür sorgen können, dass das Dateisystem vollläuft, was wiederum dazu führt, dass das Monitoring-System Alarm schlägt. Logs, die nur aus Meldungen bestehen, die niemand anschaut, erfüllen keinen Nutzen.

Ein wesentlicher Vorteil von Logs ist, dass diese ziemlich einfach zu schreiben sind. Die einfachste Variante schreibt reinen Text in das Logsystem, sortiert nach verschiedenen Log-Leveln: `Debug`, `Info`, `Warn`, `Error` und `Fatal`. Das Logsystem fügt den Zeitstempel hinzu, sodass alle Einträge in einer Zeitlinie betrachtet werden können.

Diese einfache Filtermöglichkeit mit Log-Leveln sorgt dafür, dass je nach eingestelltem Level mal mehr und mal weniger Logs ausgegeben werden. Wie diese Level jedoch definiert werden, ist ein großes Problem.

Bei *schick-gekleidet.de* gab es den Fall, dass die Webanwendung entschieden zu viele Meldungen loggte, die dann geflissentlich von den Admins ignoriert wurden, denn in diesem Heuhaufen war nur mit größtem Aufwand eine Nadel zu finden. Andere Teile der Anwendung loggten weitestgehend nichts, und das, was ausgegeben wurde, war maximal unbehilflich: `System Error. System failed`.

Auch hier ist die Lösung eine engere Zusammenarbeit zwischen Dev und Ops: Wenn es zu Ausfällen kommt, bei denen keine sinnvollen Ausgaben im Log stehen, muss dies entsprechend kommuniziert werden, damit beim nächsten Mal mehr Informationen bereitstehen. Oder, noch besser: Das entsprechende Entwicklerteam muss in die Fehlersuche im Betrieb eingebunden werden.

Wer den Code geschrieben hat, wird deutlich schneller Zusammenhänge verstehen, was das Debugging vereinfacht. Diese Erfahrung aus der Produktion sorgt dann auch dafür, dass klar wird, welche Informationen in welcher Stufe geloggt werden müssen – und was nur zu Performance-Einbußen führt.

Damit die unterschiedlichen Teams Zugriff auf die Log-Meldungen des Produktivsystems bekommen, braucht es ein entsprechendes Tooling, das die Meldungen archiviert, durchsuchbar macht und aufbereitet.

Reflexion

Sie müssen die hier dargestellten Probleme auch an dieser Stelle im Kontext der Probleme und Herausforderungen aus den vorherigen Kapiteln betrachten.

Schauen Sie, was auf Ihre Umgebung zutrifft oder nicht. Wenn Sie im Betriebsteam sind, bieten Sie dem Entwicklungsteam Zugriff auf das Monitoring-System an. Andersherum sollten Entwickler Funktionen, die genau überwacht werden sollten, dem Betriebsteam nennen, damit ein entsprechendes Monitoring eingerichtet werden kann.

Viele Probleme lassen sich schon durch Kontakte und Gespräche auf dem kurzen Dienstweg lösen oder zumindest verringern, auch wenn es die offizielle Struktur in

der Firma nicht vorsieht. Idealerweise sorgen Sie dafür, dass diese Anpassungen, wie der Zugriff auf das Monitoring-System für alle relevanten Personen, direkt regulär umgesetzt werden.

10.2 Mit Durchblick kommt Weitsicht

Um die Situation bei *schick-gekleidet.de* zu verbessern, muss das Monitoring von einem rein reaktiven Prozess in weitsichtiges Planen und Handeln überführt werden. Es wird Sie an dieser Stelle also nicht überraschen, dass im ersten Schritt das Monitoring-Tool für alle in der Organisation geöffnet werden sollte. Dies gibt dem Entwicklungsteam Einblicke in den Betrieb und sorgt dafür, dass diese Erkenntnisse für die Verbesserung der Software genutzt werden können.

Dabei werden die Grundprinzipien des Monitorings nicht auf den Kopf gestellt. Auch im DevOps-Umfeld wird ähnlich wie beim traditionellen Ansatz gearbeitet. So geht es auch hier um die Visibilität in die laufende Umgebung, inklusive aller möglichen Probleme.

Das Ziel des gemeinsamen Teams nach DevOps-Prinzipien ist, dass die Änderungen zügiger an die Endnutzer ausgeliefert werden. Im Rahmen der »Three Ways« in Hinblick auf das Monitoring sind sowohl die Feedback-Schleife aus dem zweiten Weg interessant als auch die kontinuierliche Verbesserung aus dem dritten Weg.

Im traditionellen Umfeld gab es keinen klaren Prozess, wie die Beobachtungen aus dem Monitoring-System zurück in die zukünftige Entwicklung fließen sollten. Durch die Silos zwischen den Teams war das auch nicht sonderlich verwunderlich.

Bei *schick-gekleidet.de* gab es das Problem, dass neue Lastspitzen nach nahezu jedem großen Deployment erreicht wurden und dass es unklar war, was für einen Einfluss die Performance der neuesten Änderungen hierbei hatte. Genau hier kommt es auf die *Observability* an. Anstatt nur die reine Verfügbarkeit und die reine Performance der Anwendung zu prüfen, geht es nun darum, dass viel tiefer in jeden Request hineingeschaut werden kann, um etwaige Probleme zu finden.

Das Ziel ist, dass man durch die Kombination von Logs, Metriken und Traces so tief in die laufende Anwendung blicken kann, dass mögliche Probleme detailliert erfasst werden, um ebendiese ausmerzen und verbessern zu können. Wie das konkret aussieht, sehen wir uns jetzt an.

10.2.1 Observability-Engineering

Heutzutage wird der Oberbegriff *Observability* benutzt, früher wurde meist von *Application-Performance-Monitoring* (APM) gesprochen. Im Großen und Ganzen ist

© Rheinwerk Verlag, Bonn 2024

damit etwas Ähnliches gemeint, nämlich dass viel tiefer in die laufende Anwendung geschaut wird.

Während reines, klassisches Monitoring auch weiterhin benötigt wird, um bei Problemen reagieren zu können, setzt Observability auf einen anderen Ansatz. Hier arbeitet man nicht reaktiv, sondern wirft einen Blick in die Zukunft, arbeitet also *proaktiv*. Es geht darum, dass auftretende Probleme möglichst schnell gefunden werden. Nachdem die Alarmierung da ist, wird angeschaut, was das genaue Problem ist.

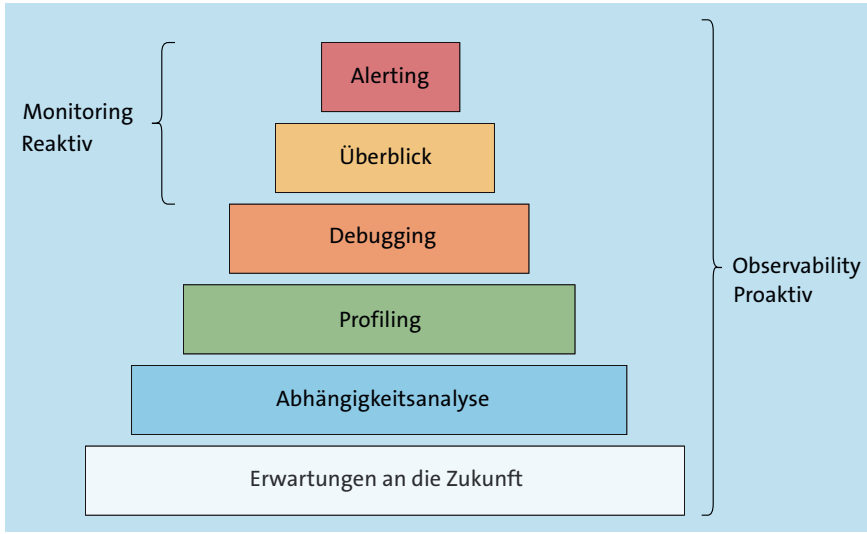

Abbildung 10.2 Monitoring ist meistens eher reaktiv, während bei Observability proaktiv über den ganzen Stack gedacht wird.

Wie Sie in Abbildung 10.2 sehen, hört das Monitoring also mit dem Überblick über das System auf. Observability hingegen kann und soll auch genutzt werden, wenn kein Problem existiert. Es liefert Daten für das Debugging, beim Profiling und bei der Analyse der Abhängigkeiten.

Das geschieht, um die Erwartungen an die zukünftige Performance eleganter formulieren zu können. Das Ziel ist, zu jeder Zeit Fragen zum aktuellen Zustand, zu Verläufen und zur Zukunft aller Services in der Umgebung stellen zu können. Dadurch kann man auch Korrelationen und Einblicke in bisher nicht bekannte Probleme erhalten, wo man bisher gar nicht wusste, wie man diese Frage formulieren sollte (sogenannte *unknown Unknowns*).

Dabei haben sich bei der Observability anfänglich drei Säulen für die Datensammlung entwickelt: *Logs*, *Metriken* und *Traces*. Sowohl Logs als auch Metriken stammen aus dem klassischen Monitoring-Ansatz, sind also keine Hexerei. Bei Traces geht es darum, dass man den gesamten Ablauf einer Anfrage komplett nachvollziehen kann. Dazu komme ich gleich noch einmal.

Persönliches Exemplar für Thomas Bartholomäus

Zusätzlich haben sich noch weitere Observability-Daten entwickelt, etwa im Bereich des Profilings, der End-to-End-Tests und des Error-Trackings. Die Community ist uneins, ob die drei Säulen noch legitim sind; mehr zu diesem Thema findet sich auf der Seite *https://o11y.love/#what-is-observability*. Um mit Observability starten zu können, empfiehlt sich der Fokus auf Logs, Metriken und Traces.

Wenn man nun Logs, Metriken und Traces zusammenfasst, kommt man also zu Observability-Praktiken. Aber was heißt das in der Praxis? Nehmen wir den Fall, dass ein Kunde im Shop von *schick-gekleidet.de* den Fehler meldet, dass er den Kauf nicht abschließen kann, weil ein `500 Internal Server Error` als Fehlermeldung erhält.

In diesem Fall musste zuvor erst mal jemand aus dem Betriebsteam das Ticket übernehmen und es prüfen. Im ersten Schritt muss nachgeschaut werden, woher dieser Fehler stammt und welches System verantwortlich ist. Typischerweise bedeutet das, dass man in die Logs schaut und hofft, dass sich dort Stacktraces finden lassen, die zur Fehlerbeschreibung passen.

Für den Admin und die ggf. hinzugezogenen Entwickler bedeutete dies einen ziemlich hohen Aufwand, da aus den (spärlichen) Informationen des Kunden und der Vielzahl an Informationen aus den Logs überhaupt einmal Zusammenhänge interpretiert werden mussten: Warum trat der Fehler gerade jetzt bei gerade diesem Kunden auf?

Bugreports vs. Observability

In diesem Beispiel stehen die Chancen übrigens gut, dass das reaktive Monitoring-System keinen Alarm geschlagen hätte, falls dieses Problem wirklich nur bei einem einzigen Kunden auftritt. Falls es ein größeres Problem ist, das alle Kunden betrifft, *muss* der Fehler bereits anders auffallen. Sie können sich in diesem Fall nicht auf Bugreports von Kunden verlassen. Wir gehen in diesem Beispiel aber davon aus, dass es sich um einen *Edge Case* handelt, der nur bei einem User auftritt.

Die Lösung für dieses Problem ist, dass die Fehlermeldung als Arbeitsauftrag für das Entwicklerteam verstanden wird. Natürlich ist es erst mal die Aufgabe der Admins, den Betrieb des Webshops wieder so weit zu gewährleisten, dass der Kunde seinen Einkauf abschließen kann. Viel wichtiger ist aber, dass das Ticket damit nicht abgeschlossen ist, sondern weitergereicht wird.

Und damit es in der Entwicklungsabteilung entsprechend bearbeitet werden kann, wird dort ein Einblick in Logs, Metriken und Traces notwendig. Es darf also keine Silos geben, die die Zusammenarbeit verhindern. Stattdessen sollen in cross-funktionalen Teams solche Probleme ausführlich und unter Einbeziehung verschiedener Perspektiven diskutiert werden. Unter DevOps-Gesichtspunkten heißt es vor allem, dass bei Problemen und Herausforderungen schneller und proaktiver reagiert werden kann, um ähnlichen Problemen in Zukunft vorzubeugen.

© Rheinwerk Verlag, Bonn 2024

Ein weiterer Knackpunkt ist, dass für diese Fehlerbehebung eine Vielzahl unterschiedlicher Systeme eine Rolle spielen, deren Informationen zusammengeführt werden müssen: Für Monitoring, Logging, Metriken und auch Traces existieren meist unterschiedliche Tools, mit eigenen Berechtigungen, die allesamt konfiguriert werden müssen. Der Zweck von Observability-Tools, die ich in Abschnitt 10.3 vorstelle, ist, diese drei Dinge zu kombinieren, damit klar ist, was der Kunde genau gemacht hat, und man genau den kompletten Aufruf im System rekonstruieren kann, inklusive des kompletten Traces, des Logs und der Metriken, die dazugehören.

Diese unterschiedlichen Informationen gehören einfach dazu, damit mit möglichst geringem Aufwand Fehler nachvollzogen werden können und damit klar ist, warum und in welchem Kontext genau dieser Fehler auftrat. Es ist nämlich gerade in Produktivsystemen aussichtslos, die Ausgangssituation exakt zu reproduzieren, die zu einem Fehler geführt hat – und ein »Can't reproduce« als Antwort auf ein Ticket hilft auch niemandem weiter.

Observability-Engineering kann auch unabhängig von konkreten Fehlermeldungen genutzt werden, die Endnutzer schicken. Mithilfe der Daten, die das Observability-Management-Tool aktiv nutzt, wird deutlich, wo sich potenzielle Bottlenecks verstecken. Hierzu werden alle verfügbaren Metriken genutzt, um zu schauen, auf welchen Teilen des Systems eine besonders hohe Last liegt, welche Fehler damit zusammenhängen und was ursächlich für einen Bottleneck sein kann.

10.2.2 Einblicke in Prozesse mit Tracing

Um Fehler in einer Anwendung verstehen zu können, sind Log-Dateien und Metriken hilfreich, aber nicht ausreichend – das werden alle bestätigen können, die schon einmal einem Fehler hinterhergelaufen sind, der nur unter ganz besonderen Voraussetzungen auftrat und sich kaum isolieren ließ.

Dabei muss man genau den Zeitpunkt des Fehlers in der Applikation mit den Zeitstempeln in den Log-Daten abgleichen. Bei verteilten Systemen und bei Microservices erfolgen diese Log-Einträge oftmals zur gleichen Zeit, und man starrt in ein buntes Chaos voller möglicher Programmabläufe von unterschiedlichen Host-Systemen. Was hier praktisch wäre, ist eine Verkettung aller Log-Ereignisse.

In diesen Fällen ist *Tracing* unbedingt notwendig. Grundsätzlich geht es darum, dass ein kompletter Flow eines Requests oder einer Transaktion durch ein System getrackt wird.

Indem Sie Tracing nutzen, erhalten Sie tiefe Einblicke in die einzelnen Funktionsaufrufe. Sie können so sehen, wie lange einzelne Aufrufe dauern, um etwaige Bottlenecks in der Anwendung aufzudecken, die wiederum sonst zu Einschränkungen in der Performance und somit in der Funktionalität führen.

Persönliches Exemplar für Thomas Bartholomäus

Dabei wird ein kompletter Request untersucht (siehe Abbildung 10.3): Welche Methode wurde aufgerufen? Welche Argumente wurden hierfür übergeben? Welche Werte besaßen diese Argumente? Wie lange dauerte der jeweilige Aufruf?

Wenn dieser Ansatz fehlt, müssen Probleme aufwendig nachgestellt werden, was in den seltensten Fällen in Produktivsystemen möglich ist.

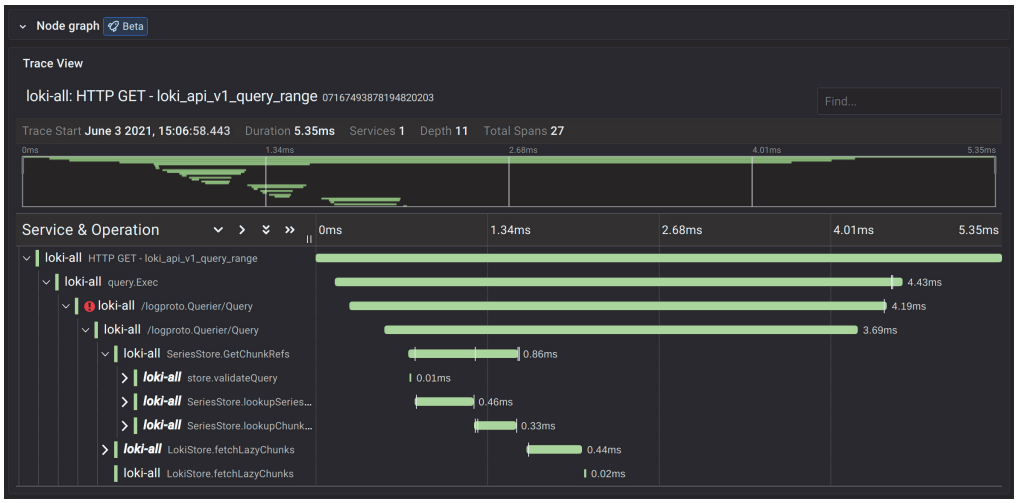

Abbildung 10.3 Ein View eines Trace in Grafana

All diese Informationen sind wichtig, damit man die Zeit für jede Stufe hat, sodass man hier schneller, besser und überhaupt Verbesserungsmöglichkeiten finden kann – und das mit Daten aus dem Produktivbetrieb. Hat man diese Daten nicht, ist es eher schwierig, Bottlenecks zu finden, da die realen Daten und realer Traffic fehlen.

Instrumentierung von Telemetrie-Daten

Observability-Tools funktionieren natürlich nur, wenn die entsprechenden Daten auch im System landen: ohne Daten keine Observability. Die eigenen Anwendungen müssen also instrumentiert werden, sodass sie die Daten liefern, mit denen das Observability-Tool dann arbeiten kann. *OpenTelemetry* (*https://opentelemetry.io/*) ist etwa eine Sammlung und Spezifikation von APIs, SDKs und Tools, um Metriken, Logs und Traces zu instrumentieren, zu sammeln und für das eingesetzte Observability-Tool zu exportieren. Mehr dazu folgt in Abschnitt 10.3.

Ein Tool, das viele Ähnlichkeiten zu einem Tracing-Tool aufweist, ist der *Profiler* im Browser. Die modernen Browser wie Firefox oder Chrome besitzen in der Webkonsole einen Profiler, der anzeigt, welche Daten geladen werden und wie lange die Aufrufe im JavaScript benötigen. Das ist ein gängiges Tool, das im Frontend-Development genutzt wird. Im Gegensatz zum Tracing werden beim Profiling allerdings die Zustände

© Rheinwerk Verlag, Bonn 2024

immer wieder ganz kurz gestoppt, um die Daten zu erfassen, was beim Tracing nicht notwendig ist.

10.2.3 A/B-Tests

Wenn man die Monitoring- und Observability-Tools richtig eingerichtet hat und nutzt, dann sind A/B-Tests hilfreich, um Experimente durchzuführen. Vorbedingung ist allerdings, dass man volles Vertrauen in die Anwendung und in das kontinuierliche Deployment hat, denn sonst kommt man erst gar nicht dazu, mit A/B-Tests zu beginnen.

A/B-Tests sind Tests, in denen zwei Varianten implementiert werden, um zu prüfen, welche Variante bessere Ergebnisse liefert. Bei *schick-gekleidet.de* können zum Beispiel zwei (oder mehr) verschiedene Varianten implementiert werden, um die Angebote des Tages auf der Startseite zu bewerben. Je nach Aufbereitung des Tagesangebots klicken mal mehr oder mal weniger Leute darauf.

Hier können viele Faktoren hineinspielen: die Größe der Werbung, der Text, die farbliche Gestaltung und noch vieles mehr. Anhand der *Conversion-Rate*, also des Prozentsatzes der Kunden, die auf das Angebot klicken, lässt sich der Erfolg unterschiedlicher Designs messen (siehe Abbildung 10.4).

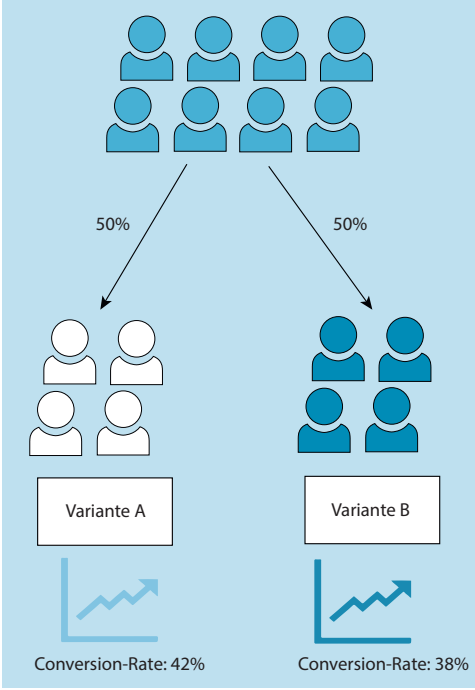

Abbildung 10.4 A/B-Tests: Zwei Varianten der Software für jeweils 50 % der Nutzer können zu unterschiedlichen Conversion-Raten führen.

Je nach Art und Weise der Implementierung werden häufig zwei verschiedene Varianten der sonst gleichen Software ausgerollt und über den Load-Balancer wird der Traffic dann auf die beiden Deployments zu jeweils 50 % verteilt. So kann man praktikabel und schnell prüfen, welche Lösung besser ist, anstatt lange zu philosophieren und Mutmaßungen anzustellen.

Aber nicht nur für die Conversion-Rate kann man A/B-Tests verwenden. Hilfreich kann es auch sein, wenn Performance-Verbesserungen implementiert wurden, die bei einem kleinen Teil der Besucher getestet werden sollen. Auch hier können Sie mit A/B-Tests prüfen, welche Implementierung besser läuft.

Den (Miss-)Erfolg von A/B-Tests teilen

Im Rahmen des *Sharings* nach dem CALMS-Modell sollten Erfolge, aber auch Misserfolge von A/B-Tests im Unternehmen geteilt werden, damit auch andere Teams, die gegebenenfalls vor ähnlichen Herausforderungen stehen, aus diesen Erkenntnissen lernen können.

Das Ganze läuft dann auch als Teil von *Continuous Improvement*, was ja einem der Kern-DevOps-Prinzipien entspricht. Die Erkenntnisse aus den A/B-Tests sollten daher stetig in die nächsten Entwicklungsphasen mit hineinspielen.

10.2.4 Business-Monitoring

Neben dem technischen Monitoring sollte auch das eigentliche *Business* überwacht werden. Bisher war häufig nur die Rede von genereller Last, Nutzung und Performance. Am Ende ist allerdings wichtig, wie gut oder schlecht das Business läuft.

Für die Mitarbeiter von *schick-gekleidet.de* ist also wichtig, dass sie möglichst viel in ihrem Online-Shop verkaufen. Bisher gab es nur einmal im Monat Berichte über die Verkaufszahlen der vergangenen 30 Tage. Dabei wurden Daten aus dem Billing-Backend gezogen und in einigen manuellen Schritten in Excel aufbereitet.

Das war zwar besser als nichts, aber wieder nur ein rein reaktiver Blick auf die Lage: Die Daten erlauben ausschließlich den Schluss darauf, wie gut der vergangene Monat gelaufen ist. Kurzfristige Auswertungen oder gar Tests, wie konkrete Anpassungen das Einkaufsverhalten beeinflussen, gab es nicht.

Besser wäre es, das Ganze nicht nur zu automatisieren, sondern auch einen Real-Time-Einblick in die Geschäftsdaten zu bekommen. Hier lassen sich gleich mehrere Metriken ermitteln, die sowohl für die Business-Analysten als auch für den Betrieb sinnvoll sind.

Aus Business-Sicht ist in erster Linie zunächst interessant zu wissen, wie viel Umsatz gerade erzeugt wird. Diese Daten fallen schon an, da alle Verkäufe in der Datenbank gespeichert werden. Diese Daten gilt es aufzubereiten, sodass Umsatzzahlen für die

© Rheinwerk Verlag, Bonn 2024

Geschäftsführung sichtbar werden. Es lassen sich aber noch etliche verwandte Metriken ermitteln, die alle in eine ähnliche Richtung gehen, darunter zum Beispiel:

- Anzahl der Verkäufe
- Anzahl der verkauften Elemente
- Füllstände im Lager
- Conversion-Rate

Der springende Punkt ist, dass diese Daten, die sowieso schon erhoben werden, in Echtzeit ausgewertet werden, damit sie auf verschiedene Zeiträume wie Minuten, Stunden, Tage oder auch Wochen und Monate aggregiert werden können.

So ließe sich unter anderem direkt darstellen, wie eine Marketing-Kampagne, die im Rahmen des letzten großen Deployments gefahren wurde, konkret performt, indem die Daten mit anderen Zeiträumen vergleichbar werden. Der Erfolg der Marketing-Kampagne lässt sich dadurch mit den normalen, gängigen Daten vergleichen, um den Erfolg der Kampagne zu validieren.

Mit den Änderungen am Monitoring-System bekommt das Business-Team also in Echtzeit einen Einblick, wie gut der Shop gerade läuft. Und nicht nur das: Wenn man CI/CD und DevOps richtig umsetzt, können A/B-Tests und Feature-Flags effektiv dafür genutzt werden, auch während der laufenden Marketing-Kampagne Änderungen durchzuführen, um zu schauen, womit man die Conversion-Rate hochtreiben kann.

Diese Erkenntnisse sind wiederum für den Betrieb der Infrastruktur wichtig, denn mit den detaillierten Business-Kennzahlen und erweiterten Metriken lässt sich den verschiedenen Stunden des Tages ein Geldbetrag beimessen. Mit diesem Betrag können etwaige Umsatzeinbußen realistischer beziffert werden.

Dadurch lässt sich ein Systemausfall eines Online-Shops gut beziffern. Dass ein Ausfall am frühen Abend, wenn die meisten Kunden unterwegs sind, deutlich teurer ist als ein Ausfall mitten in der Nacht, liegt auf der Hand.

Mit entsprechenden Auswertungen lässt sich dies quantifizieren, und es kann auch gezeigt werden, wie stark einzelne Funktionen zum Umsatz beitragen. Sie können insbesondere neue Komfortfunktionen, die für spürbar höhere Systemlast sorgen, einem A/B-Test unterziehen und das Ergebnis mit den Umsätzen oder Klicks in Beziehung setzen. Ohne langes Herumrätseln können Sie so datengestützt entscheiden, ob sich die Implementierung lohnt.

Insbesondere im Hinblick auf Stabilitätsprüfungen mittels Chaos-Engineering können dann experimentellere Dinge bevorzugt zu den Zeiten ausgeführt werden, zu denen ohnehin weniger Umsatz generiert wird. Die damit zusammenhängenden Downtimes können somit realitätsnäher beziffert werden, um auf Basis dieser Kennzahlen weitere Entscheidungen und Verbesserungen umzusetzen.

Persönliches Exemplar für Thomas Bartholomäus

10.3 Tools für Monitoring, Observability und Tracing

In Abschnitt 10.1 wurde bereits viel über die alte Welt des reaktiven Monitorings und die neu aufkommende Welt von Observability berichtet. Klassische Monitoring-Themen sind heute auch noch genauso relevant wie die neuen Tools aus dem Observability-Universum.

Wichtig ist der Einsatz im Gesamtkonzept: Alle relevanten Informationen müssen für alle nutz- und sichtbar sein. Das bezieht sich nicht nur auf die Infrastruktur, auf der die Anwendungen laufen, sondern auch auf die Anwendungen selbst. Je tiefer man in das System hereinschauen kann, desto besser.

Monitoring-Tools werden Sie sicherlich schon im Einsatz haben, wenn Sie eine Anwendung oder eine Infrastruktur betreuen. Aber sehen Sie in Ihrem Setup wirklich alles auf einen Blick, was Sie wissen wollen? Haben auch andere Personen Zugang zu Daten, die für ihre Arbeit wichtig sind? Und: Sind die Checks und Kontrollen auch hilfreich? Oder wird durch eine Datenflut vermittelt, dass alles im Blick ist, obwohl man wichtige Sachen übersieht?

SSH vermeiden

Ich habe folgende Erfahrung gemacht: An der Nutzung von SSH lässt sich gut ablesen, ob das Monitoring ordentlich umgesetzt ist. Denn wenn das Monitoring des ganzen Tech-Stacks nach Best Practices umgesetzt ist, wird kein SSH-Zugang auf den Systemen mehr benötigt.

Wenn Sie das nächste Mal SSH nutzen, um nachzuschauen, ob etwas noch funktioniert, dann können Sie sich angewöhnen, ein Ticket aufzumachen, um eine Verbesserung für das Monitoring-System zu vermerken. Es ist ein guter Indikator dafür, dass das Monitoring nicht die richtigen Informationen liefert.

Dasselbe gilt für Log-Dateien: Wenn Sie sich auf einem Server anmelden und sich per Hand Logs ausgeben lassen müssen, sollten Sie einen zentralen Log-Server einplanen, der alle wichtigen Logs zentral archiviert und durchsuchbar macht.

Das gilt besonders dann, wenn mit virtuellen Maschinen oder Containern gearbeitet wird – wenn Sie also unter Kubernetes den Befehl `kubectl logs` brauchen. Das sind Informationen, die Sie eigentlich schon im Blick haben und nicht extra aufrufen müssen sollten.

(Wichtig: Gemeint sind dabei Produktivumgebungen. Bei Umgebungen, die für Tests und Entwicklungen genutzt werden, ist das natürlich etwas anderes, denn dort ist ein so umfassendes Monitoring-Setup meistens Overkill.)

10.3.1 Systeme mit Icinga/Nagios überwachen

Zu den klassischen Monitoring-Systemen gehören Tools wie Nagios und Icinga. Icinga war ursprünglich eine Abspaltung von Nagios. Die neuere Icinga2-Version ist

© Rheinwerk Verlag, Bonn 2024

hingegen ein kompletter Rewrite des Monitoring-Systems und liefert eine neue Konfigurationssprache und besseren Support für verteilte Monitoring-Umgebungen. Altbewährte Schnittstellen zu Plugin-Skripten, die ausgeführt werden, um Host/Service-Zustände (`OK`, `WARNING`, `CRITICAL` und `UNKNOWN`) zu erfassen, findet man aber doch wieder. Metriken müssen mit externen Tools als Integration nachgerüstet werden.

Wie bereits in Abschnitt 10.1 beschrieben, wird mit diesen Tools reaktiv gearbeitet: Es werden Überwachungen konfiguriert, die häufig entweder binäre Checks sind oder anhand von festgelegten Schwellenwerten getriggert werden. Wenn dann ein Alarm getriggert wird, werden über die konfigurierten Alarmierungsmethoden Benachrichtigungen herausgeschickt.

Solche Monitoring-Systeme – ich nenne sie hier mal *klassische* Monitoring-Systeme – haben allerdings ein wesentliches Problem: Die konfigurierten Checks sind meistens relativ hart mit den existierenden Systemen verdrahtet.

Gerade wenn man nach dem Prinzip »Cattle not Pets« arbeiten möchte, dann ist die Arbeit mit Monitoring-Lösungen wie Icinga nicht sehr komfortabel, da diese Tools von Haus aus keine Auto-Discovery mitbringen, sondern für die Dienste, die überwacht werden sollen, konfiguriert werden müssen.

Beim Auto-Discovery machen sich die Server und Dienste mit geringem Konfigurationsaufwand selbst sichtbar, sodass der Monitoring-Server diesen automatisch einbindet. Das ist insbesondere in der Cloud nützlich, da immer wieder Server auftauchen und verschwinden. Sicherlich lässt sich einiges per Scripting automatisieren, doch auch das führt zu einer höheren Komplexität.

Ein weiterer Aspekt ist, dass hauptsächlich die Infrastruktur überwacht wird. Zur Infrastruktur zähle ich hier vorwiegend die Server selbst und das Netzwerk. Aber auch die Anwendungen gehören in einem gewissen Maße dazu, allerdings häufig nur die zuvor angesprochenen binären Checks: also eben, ob die Anwendung läuft oder nicht.

Performance-Monitoring ist in vielen Fällen in einem gewissen Umfang enthalten; der Fokus ist allerdings ein anderer, sodass es sich häufig nicht so *richtig* anfühlt und die Funktionen limitiert sind. Meistens geht es nur um die »Gesundheit« der Systeme, also beispielsweise darum, wie viel Plattenplatz noch verfügbar ist.

Icinga2-Best-Practices

Im Blog der Firma Netways, die maßgeblich Icinga2 entwickelt, findet sich eine Serie zu den Best Practices mit Icinga2, bei der näher definiert ist, wie man das Icinga2-Setup idealerweise aufbaut:

https://www.netways.de/blog/series/icinga-2-best-practice/

Icinga2 und das ältere Nagios sind nicht die einzigen Monitoring-Tools mit größerer Verbreitung. Es gibt auch noch Tools wie *Zabbix* und *Check_MK*, die ebenfalls eine

gewisse Verbreitung in der Praxis haben und sich auch den Herausforderungen von Cloud-nativem Monitoring widmen. Zabbix bietet beispielsweise Kubernetes-Monitoring, ebenso wie Check_MK, das sich zudem im Bereich von Service-Auto-Discovery als Vorreiter etabliert hat. In der Cloud-native-Welt hat sich *Prometheus* zum De-facto-Monitoring-Standard entwickelt. Es wird im nächsten Abschnitt betrachtet.

Alerting und Incident-Management

Monitoring ist gut und wichtig. Aber wenn Probleme im Monitoring-System auftauchen, dann muss auch zuverlässig alarmiert werden. E-Mail-Benachrichtigungen sind bei gravierenden Problemen nicht angemessen, da diese in der Masse häufig untergehen.

Es sollten also sowohl Tools für das Alerting als auch für das Management von Incidents zum Einsatz kommen. Bekannte kommerzielle Tools sind *PagerDuty* und *Opsgenie*. Letzteres ist erneut ein Tool aus dem Hause von Atlassian. In etlichen weiteren Tools sind auch Funktionen zum Alerting enthalten, so bietet auch GitLab einige Funktionen aus dem Bereich an.

Zu diesen Tools gehört auch das Management, wer wann und wie benachrichtigt werden soll. Ein typischer Fall ist, dass definiert wird, in welcher Rotation welche Personen benachrichtigt werden müssen. Wichtig ist, dass keine kritischen Warnungen verpasst werden und dass die Tools eine entsprechende Verwaltung mitbringen.

10.3.2 Monitoring mit Metriken und Time-Series-Datenbanken

Um Metriken zu erfassen sowie zu speichern, werden in der Regel keine relationalen Datenbanksysteme genutzt, da diese nicht darauf ausgelegt sind, sehr viele Metriken mit Zeitstempel praktikabel und performant zu erfassen und abzurufen. Stattdessen existieren spezialisierte Time-Series-Datenbanken, bei denen die Zeit als zusätzliche Dimension stets mit den Daten gespeichert wird.

Ein weiterer Vorteil kann das automatische Verdichten von älteren Daten sein, um Speicherplatz zu sparen. So kann man immer noch die Graphen der letzten Jahre analysieren. Ein Beispiel ist etwa, dass man nur noch die Datenpunkte alle fünf Minuten langfristig behält. Wenn man die Daten etwa jede halbe Minute sammelt, ist das für den Status quo gut und wichtig; langfristig wird das aber nicht mehr wirklich benötigt.

Heutzutage sind sowohl *Prometheus* als auch *InfluxDB* weit verbreitet. Im Grunde muss ein wenig unterschieden werden zwischen reinen Time-Series-Datenbanken und dem Monitoring. Tatsächlich ist es häufig allerdings eher ein Mix aus beidem, zumindest wenn man die Tools drumherum einbezieht.

Sowohl Prometheus als auch InfluxDB haben unterschiedliche Konzepte und Ansätze, was auch nicht sonderlich verwunderlich ist. In älteren Setups findet man häufig noch Tools wie *Graphite* (*https://graphiteapp.org*), das allerdings mit einigen

© Rheinwerk Verlag, Bonn 2024

architektonischen Einschränkungen daherkommt, nur wenig weiterentwickelt wird und keine nennenswerte Relevanz mehr hat.

Obwohl sowohl Prometheus als auch InfluxDB eigene Visualisierungsmöglichkeiten enthalten, wird häufig noch zusätzlich Grafana für die Visualisierung verwendet. *Grafana* wird in Abschnitt 10.3.3 näher beschrieben.

Prometheus

Prometheus ist sowohl ein Monitoring-Tool als auch eine Time-Series-Datenbank. Es ist eine Cloud-native Anwendung, daher wird für die Installation ein Container-Setup empfohlen, das sich über Helm-Charts anlegen lässt. Es ist eines der ersten Projekte nach Kubernetes, das den »Graduated«-Status bei der *Cloud Native Computing Foundation* (CNCF) erlangt hat.

Ähnlich wie Kubernetes wurde Prometheus von Ex-Googlern ins Leben gerufen, auch wenn es nicht direkt aus dem Hause Google stammt. Auch Prometheus ist in der Programmiersprache Go geschrieben, was in der Cloud den Vorteil mit sich bringt, dass es sich sehr einfach auf allen möglichen Systemen deployen lässt – das Kompilat der Go-Anwendung liegt als einzelne Binärdatei vor, die über alle Abhängigkeiten verfügt.

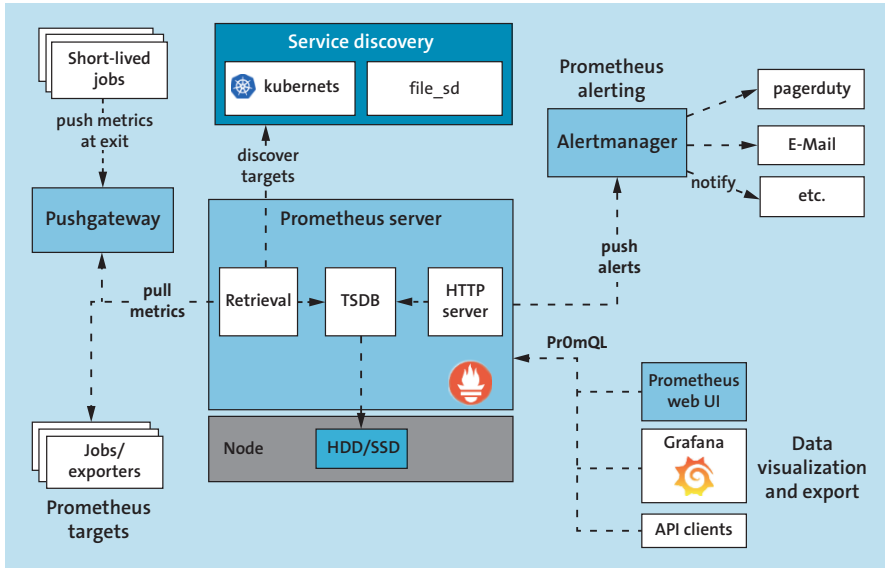

Abbildung 10.5 Überblick über die Prometheus-Architektur (Quelle: https://prometheus.io/docs/introduction/overview/)

Das Prometheus-Projekt besteht allerdings trotzdem aus verschiedenen Komponenten. So ist der *Prometheus-Server* das Kern-Element. Dieser sammelt nicht nur die Daten ein, sondern speichert diese auch und gibt natürlich auch die Daten aus, wenn

man Abfragen stellt. Abbildung 10.5 gibt einen Überblick über die Architektur des Monitoring-Stacks, die natürlich je nach Anwendungszweck anders aussehen kann.

Die ersten Metriken

Wichtig für die Architekturplanung ist, dass Prometheus nach dem Pull-Prinzip arbeitet. Es grast somit die konfigurierten Server ab und holt sich von dort die Daten, die es abspeichern muss. Das Schöne dabei ist, dass Prometheus relativ simpel gestrickt ist und man mit einfachen Mitteln auch prüfen kann, ob die Metriken von den Diensten überhaupt zur Verfügung gestellt werden.

Die Server und Dienste, die überwacht werden müssen, stellen dabei ihre Metriken per HTTP zur Verfügung. Der Prometheus-Node-Exporter ist etwa ein Exporter, der Metriken des Nodes per HTTP bereitstellt. Er ist unter `http://192.168.2.3:9100/metrics` erreichbar und liefert unter anderem diese Zeilen zurück:

```
# HELP node_network_up Value is 1 if operstate is 'up', 0 otherwise.
# TYPE node_network_up gauge
node_network_up{device="eth0"} 1
node_network_up{device="lo"} 0
node_network_up{device="wlan0"} 0
```

Unter dem angegebenen Interface wird schlicht eine Liste an Metriken mit ihren aktuellen Daten ausgegeben. Der Prometheus-Server sammelt genau diese Daten zu genau dem Abrufzeitpunkt ein und speichert diese ab. Demnach muss der Prometheus-Server immer in regelmäßigen Abständen das sogenannte *Scraping* durchführen, ansonsten kommt es zu Lücken in den Datensätzen.

Prometheus unterstützt *Service-Discovery*; Sie müssen also nicht alle Endpunkte manuell in die Konfiguration eintragen, sondern die Endpunkte werden automatisch erkannt und hinzugefügt. Hierbei wird eine Vielzahl von Cloud-Anbietern (Azure, Google Cloud, AWS, Hetzner Cloud etc.), Container-Orchestratoren (Kubernetes, Nomad etc.) sowie System-Diensten (DNS, HTTP etc.) unterstützt.

Zunächst liefert der Node-Exporter »nur« Metriken des Nodes selbst aus. Einige Anwendungen bringen selbst ihre eigenen Prometheus-Endpunkte mit, über die sie vorgefertigte Metriken bereitstellen. Das ist etwa bei Kubernetes der Fall, aber auch Netzwerk- oder Storage-Appliances bringen gewöhnlich eigene Prometheus-Metriken mit.

Spannender ist es natürlich, selbst geschriebene Anwendungen zu überwachen. Entwicklungsteams können also eigene Endpunkte mit den eigenen Metriken definieren. Dafür existieren Bibliotheken für die gängigen Programmiersprachen, die das Ganze vereinfachen. Da das Ausgabeformat aber nur ein relativ einfaches Textformat ist, braucht es diesen zusätzlichen Aufwand oft noch nicht einmal.

Durch die Art und Weise, wie Prometheus funktioniert, liegt die Macht für die Bereitstellung der gewünschten Metriken primär beim Entwicklungsteam: Das Team kann

© Rheinwerk Verlag, Bonn 2024

selbst entscheiden, welche Metriken es erfassen und bereitstellen will. Falls Metriken hinzugefügt werden müssen, kann einfach der Endpunkt erweitert und die Anwendung neu ausgerollt werden, damit die Daten automatisch in Prometheus landen.

Daher ist hierbei eine gute Zusammenarbeit der Admin- und der SRE-Teams wichtig: Welche Metriken helfen beim Debugging der Produktionsumgebung; welche Daten sind für Entwickler noch relevant, um Probleme besser analysieren zu können?

Trainings

Julius Volz, der Mitgründer von Prometheus, bietet unter *training.promlabs.com* Trainingskurse an, die jeder für sich im eigenen Tempo durcharbeiten kann. Viele Trainings sind kostenpflichtig, andere wiederum sind kostenfrei abrufbar. Auf dem dazugehörigen YouTube-Channel *PromLabs* sind zudem weitere kürzere Erklärvideos abrufbar:

https://www.youtube.com/@PromLabs

Vier Metrik-Typen

Prometheus besitzt insgesamt vier verschiedene Typen von Metriken: Counter, Gauge, Histogram und Summary.

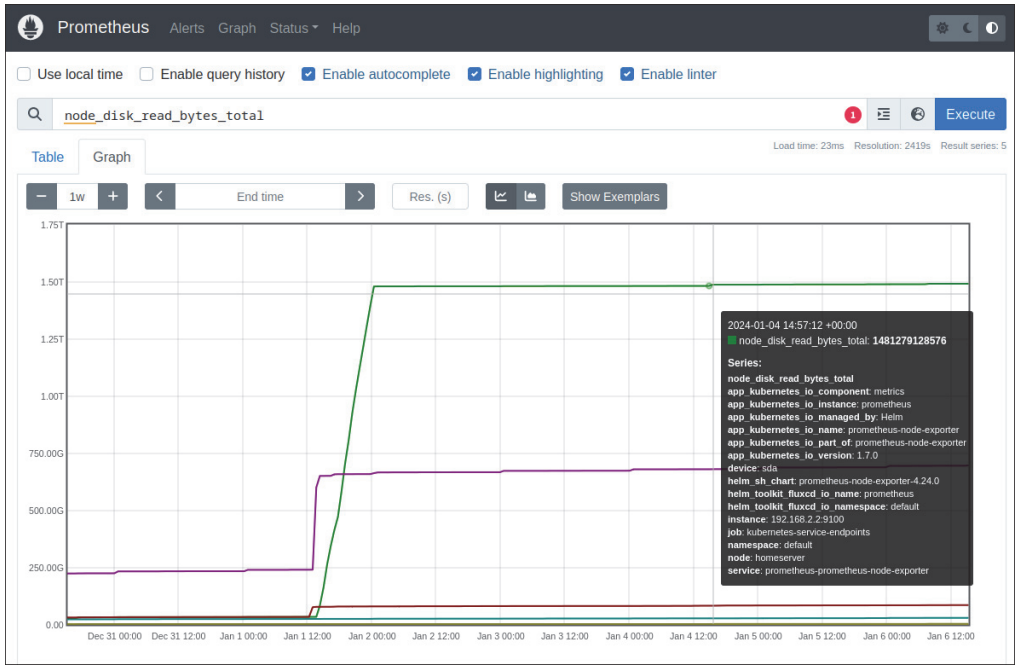

Abbildung 10.6 Visualisierung der Metrik node_disk_read_bytes_total des Typs Counter.

Der erste Typ ist der *Counter*, also ein einfacher Zähler. Metriken von diesem Typ zu ermitteln, ist denkbar einfach: Es wird schlicht jedes Mal eine Zahl hochgezählt. In der Regel wird bei jedem Neustart des Dienstes auch der Zähler wieder auf 0 gesetzt. Ein typischer Anwendungsfall für den Counter sind Zähler für Klicks, aufgetretene Fehler oder absolvierte Requests.

Wenn Sie die Daten mit einem Counter visualisieren möchten, dann müssen Sie beachten, dass der Zähler wirklich nur die Richtung nach oben kennt, wenn er nicht wieder auf null zurückgesetzt wird.

Anders sieht der zweite Typ aus. Ein *Gauge* ist ein numerischer Wert. Dieser kann nicht nur nach oben gehen, sondern auch nach unten. Eine typische Metrik ist etwa die Auslastung von CPU oder Arbeitsspeicher, die je nach Auslastung mal höher und mal niedriger sein kann.

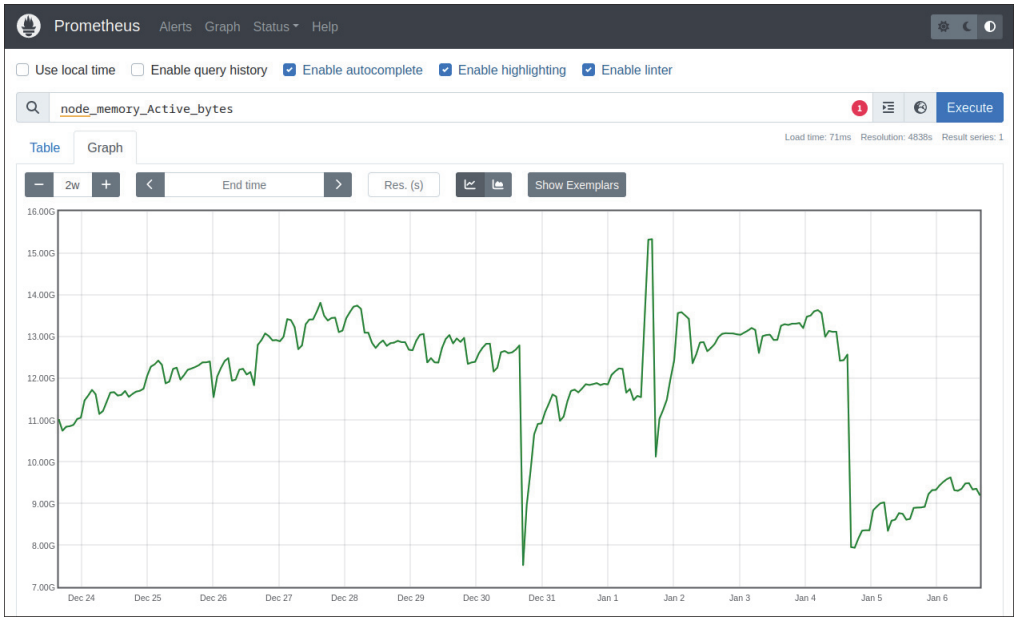

Abbildung 10.7 Visualisierung der Gauge-Metrik node_memory_Active_bytes.

Während der Counter und der Gauge einfach zu erklären und zu verstehen sind, sieht das bei den nächsten beiden Typen anders aus. Ein *Histogramm* ist nämlich ein wenig komplexer, und auch ich habe ein bisschen gebraucht, bis ich verstanden habe, was es genau ist und wie man es verwenden kann.

Ein Histogramm außerhalb der Prometheus-Welt kennen bestimmt viele. Um ein praktisches Beispiel zu nennen, kann man sich Requests anschauen, was schon als Beispiel für den Counter diente. Dort wurde aber lediglich hochgezählt; man konnte mit dem Counter also nur die Häufigkeit der Requests zählen.

© Rheinwerk Verlag, Bonn 2024

Was zusätzlich auch noch spannend ist, ist die Laufzeit der Requests zu messen: Wie viele Requests sind in unter 0,1 Sekunden beantwortet worden? Wie viele benötigen weit über 2 Sekunden? Mit solchen Metriken und ihrer Visualisierung lässt sich einfacher nachvollziehen, wie die Verteilung der Dauer der Requests aussieht und für wie viele Requests das jeweils zutrifft.

Die *Summary* ist der vierte Metrik-Typ. Sie ähnelt dem Histogramm, mit dem Unterschied, dass sich das Fenster für die Aggregationen variabel verschieben lässt.

Alle Metriktypen lassen sich zwar auch per Hand berechnen und ausgeben, aber man nutzt besser die Bibliotheken für die jeweilige Programmiersprache, die vom Prometheus-Projekt bereitgestellt werden, um sich die Arbeit zu erleichtern.

Labels

Metriken allein ergeben ohne sogenannte Labels in Prometheus keinen Sinn. Sie spielen eine wichtige Rolle, wenn man Abfragen erstellen und Metriken visualisieren möchte. In den folgenden Zeilen geht es beispielsweise um den Netzwerkzugriff:

```
node_network_up{device="eth0"} 1
node_network_up{device="lo"} 0
node_network_up{device="wlan0} 0
```

Die Metrik ist hier `node_network_up`. Das existiert dreimal, weil dort noch zusätzlich das Label `device` definiert ist. Konkret gibt es hier also drei Netzwerk-Devices: `eth0`, `lo` und `wlan0`. Aber nur das Netzwerk-Device `eth0` läuft gerade und ist daher entsprechend mit der `1` gekennzeichnet.

Labels sind Key-Value-Paare, um Metadaten zu der Metrik hinzuzufügen. Wenn man eigene Metriken exportieren möchte, dann sollte man also das obere Verfahren nehmen, anstatt etwa `node_network_up_eth0`, `node_network_up_lo` und äquivalente Namen zu verwenden.

Das hat einen einfachen Grund: Bei Abfragen oder für Visualisierungen lässt sich viel einfacher nach `node_network_up` suchen. Die Labels mit `device` werden dann entsprechend mitgeliefert. Ansonsten lässt sich der Name des Netzwerk-Devices nicht so komfortabel filtern.

Es können mehrere Labels zugeordnet werden, was auch häufig der Fall ist. Das ist insbesondere zum Filtern und Gruppieren notwendig.

Hier spielt eine Rolle, dass viele Nodes oder Container einfach verschwinden und wieder zurückkehren, da mal hoch- und mal herunterskaliert wurde. Durch den Einsatz von Labels sind mal Daten mit den entsprechenden Hostnamen verfügbar und mal nicht. Daraus lässt sich auch ableiten, wann ein Host vorhanden war und wann nicht. Das Label dient dadurch auch zur Identifizierung.

Persönliches Exemplar für Thomas Bartholomäus

Abfragen

Die Abfragesprache für Daten aus der Prometheus-Datenbank ist zu Beginn nicht so einfach und erfordert ein wenig Übung, bis man den Dreh raus hat.

Prometheus setzt auf eine eigene Abfragesprache namens *PromQL*, was kurz für »Prometheus Query Language« steht. Mit ihr können Daten in Echtzeit abgefragt werden.

Die Daten, die abgefragt werden, können entweder als simple Tabelle dargestellt oder direkt visualisiert werden. Eine einfache Abfrage- und Visualisierungsmöglichkeit bietet der Prometheus-Server selbst an. Diese ist aber mehr für Ad-hoc-Abfragen gedacht. In der Regel möchte man diverse Abfragen allerdings eher längerfristig in Panels abspeichern und visualisieren sowie Dashboards erstellen. Dafür eignet sich dann Grafana (siehe Abschnitt 10.3.3).

Eine ganz einfache Abfrage sähe wie folgt aus:

```
node_network_up
```

Das würde einfach alle Metriken ungefiltert ausgeben. Gefiltert wird auch hier über Labels:

```
node_netzwork_up{device='eth0'}
```

In diesem Fall würden dann alle Netzwerk-Devices, die `eth0` ausgeben, mit dem entsprechenden Status Prometheus bekannt gemacht.

Es geht aber auch deutlich komplizierter, das zeigt folgendes Beispiel:

```
100 - ((node_filesystem_avail_bytes
  {instance="192.168.2.2:9100",device!~'rootfs'} * 100) /
  node_filesystem_size_bytes{instance="192.168.2.2:9100",device!~'rootfs'})
```

Dieser Befehl besteht eigentlich nur aus einer Zeile und beinhaltet gleich mehrere Dinge: zum einen ein wenig Mathematik, und zum anderen werden weitere Ergebnisse gefiltert. Um genau zu sein, wird hier der Füllstand von Dateisystemen abgefragt, indem die Gesamtgröße und die verfügbare Größe an Speicher für eine bestimmte Instanz abgefragt werden, ohne das `rootfs` einzubeziehen.

PromLens

PromLens (*https://promlens.com/*) ist ein erweiterter Query-Builder. Er ist mittlerweile Open Source und wurde vom Prometheus-Mitgründer Julius Volz geschrieben. Mit PromLens lassen sich PromQL-Querys deutlich einfacher bauen. Das ist nicht nur zu Lernzwecken praktisch, sondern wird auch von erfahrenen Prometheus-Nutzern gern genutzt.

© Rheinwerk Verlag, Bonn 2024

Skalierbarkeit und Datensicherheit

Prometheus ist relativ simpel gehalten, was man merkt, wenn man sich die Skalierung anschaut. Wie ich bereits erwähnt habe, ist es ein Cloud-natives Projekt. Großartig skalieren lässt es sich hingegen nicht. Zumindest nicht so, wie man es anfänglich erwarten würde.

Prometheus holt sich die Daten von den Endpunkten regelmäßig ab und speichert diese. Solange das stetig durchgeführt wird, kommt es auch nicht zu verloren gegangenen Daten. Das heißt im Umkehrschluss allerdings auch, dass im Nachhinein keine Daten nach Prometheus migriert werden können. Prometheus speichert immer die Daten ab, die es beim Abrufen zu der jeweils aktuellen Zeit abgeholt hat. Dadurch lassen sich keine Daten nachträglich einschleusen.

Dazu gehört auch, dass Daten nicht einfach gelöscht werden können. Für die langfristige Speicherung braucht man nicht zwangsläufig Datensätze von allen 30 Sekunden, sondern hier reichen auch die Daten, die alle 5 Minuten oder gar stündlich anfallen. Auch das ist im Nachhinein nicht mit Prometheus möglich, stattdessen gibt es eigene Projekte wie *Cortex* (*https://cortexmetrics.io/*) oder *Thanos* (*https://thanos.io/*), die sich auf die Langzeitspeicherung von Daten aus Prometheus konzentriert haben. Diese Projekte lösen auch ein paar Skalierungsprobleme, die sich durch die Einfachheit von Prometheus ergeben.

Wer hingegen das Setup recht einfach halten möchte, der kann auch die Speicherfrist hochsetzen, sodass eben nicht nach kurzer Zeit schon die Daten gelöscht werden. Das führt natürlich zu größeren Datensätzen. Alternativ lassen sich auch Prometheus-Server kaskadiert einsetzen, die mit einem anderen Intervall die Daten abrufen.

Prometheus im DevOps-Kontext

Jetzt wurde schon sehr viel vom Setup und der Idee von Prometheus berichtet. Was noch geklärt werden muss, ist die Frage, was genau der Vorteil von Prometheus im DevOps-Kontext ist. Auf einige Punkte bin ich schon eingegangen: Es lassen sich relativ einfach Metriken für die eigene Anwendung bereitstellen, entweder mit den Bibliotheken oder auch ohne.

Durch einige Iterationen können also immer wieder neue Metriken ermittelt und komfortabel angelegt und auch wieder entfernt werden, ohne dass man ein separates Team erneut kontaktieren oder um Änderungen bitten muss. Das gibt die Natur des Scrapens automatisch mit her.

Die meisten Beispiele waren bisher rein technischer Natur. Man kann allerdings die gleichen Metriken auch für fachliche Metriken verwenden. Für unseren Online-Shop bietet es sich etwa an, die Business-relevanten Metriken auszugeben, wie die Größe des Warenkorbs oder der Einkäufe über bestimmte Zeiträume.

Auch A/B-Tests lassen sich durch den Einsatz weiterer Labels nutzen und können entsprechend visualisiert werden. Wenn man Prometheus im Kubernetes-Cluster

nutzt, müssen die entsprechenden Pods auch nur mit entsprechenden Kubernetes-Labeln versehen werden, damit neue Prometheus-Endpunkte automatisch per Auto-Discovery erkannt und abgerufen werden. Nachdem man das Ganze einmal durchblickt und eingerichtet hat, erhält man hier sehr hohe Flexibilität.

Die erfassten Metriken können dann auch zur Alarmierung verwendet werden. Das betrifft auch hier nicht nur die technischen Aspekte, sondern ebenfalls die fachlichen Aspekte, wenn denn die Metriken bereitgestellt und erfasst werden.

Dafür existiert der Alert-Manager, der für die Alarmierung genutzt werden kann, wie man es von einem Monitoring-System auch erwartet.

Die Alarme basieren auf den Metriken. So lässt sich einfacher alarmieren, wenn Probleme anstehen, bevor es zu spät ist.

InfluxDB

Während Prometheus sowohl eine Time-Series-Datenbank als auch ein Monitoring-Tool ist, ist *InfluxDB* eine reine Time-Series-Datenbank, die ebenfalls unter einer Open-Source-Lizenz steht. Hinter InfluxDB steht hingegen die Firma InfluxData Inc.

Die Besonderheit bei Prometheus war, dass es die Daten vom Server selbst abholt, also nach dem Pull-Prinzip arbeitet. Das ist bei Influx anders, denn hier werden die Daten von extern in die Datenbank geschrieben. Es arbeitet daher nach dem Push-Prinzip, so wie man es auch von SQL-Servern kennt. Abgespeichert werden können auch hier Metriken und Events.

Ein Monitoring mit InfluxDB ist daher grundsätzlich komplexer, es lässt sich aber besser skalieren, als es bei Prometheus der Fall ist. Die Option zur horizontalen Skalierung ist ein direkter Teil der kostenpflichtigen Premium-Lizenz.

Um die Daten in die Datenbank zu schreiben, können Bibliotheken für die gängigen Programmiersprachen verwendet werden. Dies bietet sich besonders dann an, wenn die Metriken der eigenen Anwendung gesammelt werden sollen. Alternativ gibt es noch die Möglichkeit, auf *Telegraf* zu setzen. Telegraf ist ein Agent-Server, der es mit einfacheren Mitteln ermöglicht, aus verschiedenen Quellen die Metriken einzusammeln und in die InfluxDB zu schreiben. Dies können sowohl Sensordaten über MQTT sein als auch Daten von Kubernetes oder sogar Prometheus.

Das Datenmodell ist ähnlich flexibel wie bei Prometheus: Die Daten können mit geringem Aufwand und mit recht großer Flexibilität in die Datenbank geschrieben werden, sodass auch immer wieder neue Metriken erstellt werden können.

Für die Abfragen wurde früher auf *InfluxQL* gesetzt, die starke Ähnlichkeiten zu SQL-Abfragen hatte. Mit Version 2 wurde die *Flux Query Language* eingeführt, die etwas mächtiger ist. Persönlich fand ich beim ersten Nutzen die Flux Query Language im Vergleich zu PromQL deutlich schwieriger zu verstehen.

© Rheinwerk Verlag, Bonn 2024

Bei InfluxDB sieht man stark, dass der Fokus auf der Datenbank liegt, was bei Prometheus anders ist. Auch die Datenhaltung ist deutlich flexibler; so kann man sowohl nachträglich Daten hinzufügen, indem man den entsprechenden Zeitstempel mit übergibt, als auch Daten nachträglich löschen. Das sind Dinge, für die Prometheus nicht gemacht wurde.

InfluxDB hat seine eigenen Visualisierungsmöglichkeiten. Da es in vielen Organisationen meist einen ganzen Zoo an verschiedenen Datenquellen gibt, ist es sinnvoller, auf Grafana zu setzen, das im nächsten Abschnitt näher betrachtet wird.

Sollte man also eher auf Prometheus oder eher auf InfluxDB setzen? Ich denke, die Antwort ist hier ein klares »Es kommt drauf an!«. Prometheus ist einfach gehalten, der Einstieg ist schnell erledigt und es können schnell und einfach Metriken bereitgestellt werden, die durch das Auto-Discovery abgerufen werden.

Auch der Einstieg in PromQL ist relativ einfach. Bei InfluxDB liegt hingegen der Fokus auf der deutlich besseren Datenhaltung. Je nachdem, welche Daten man wie aufbewahren muss, bietet sich eventuell eher InfluxDB an.

Am Ende gilt allerdings: Wichtig ist, womit sich das Team auskennt und womit man ans Ziel kommt. Das geht mit beiden Tools. Auch können natürlich beide parallel genutzt werden, je nach Einsatzzweck. Eine Mischung von beiden kann allerdings schnell dazu führen, dass hier wieder noch mehr Know-how in Wartung und Nutzung aufgebaut werden muss.

Reflexion

Wenn Sie ein modernes metrikbasiertes Monitoring-Tool einsetzen wollen, führt kein Weg an Prometheus vorbei. Das Tool ist einfach gehalten, bietet aber trotzdem alles, was man für die Überwachung der eigenen Umgebung braucht. Betrachten Sie die Einfachheit als Chance und nicht als Risiko, denn vor allem zu Beginn ist es einfach, einen kleinen Prometheus-Server hinzustellen und Daten zu sammeln. Die Skalierung können Sie immer noch später angehen.

Machen Sie sich stattdessen lieber mit dem Arbeiten mit Prometheus vertraut. Die Server-Komponente ist schnell eingerichtet und kann ohne großen Aufwand ausgerollt werden. Die Herausforderung ist eher das Lernen von PromQL, aber dabei unterstützen die Helferlein wie PromLens.

10.3.3 Datenvisualisierung mit Grafana

Sowohl Prometheus als auch InfluxDB legen ihren Fokus auf die Time-Series-Datenbank selbst. Die Visualisierung der Daten sollte daher ein spezialisiertes Tool übernehmen: Grafana

Mit Grafana können Sie Daten aus unterschiedlichen Datenquellen visualisieren. Neben den Time-Series-Datenbanken können Sie noch etliche weitere Typen von Datenquellen anbinden. Dazu gehören unter anderem Logging- und Document-Datenbanken, Systeme für das Tracing und Profiling oder klassische SQL-Datenbanken.

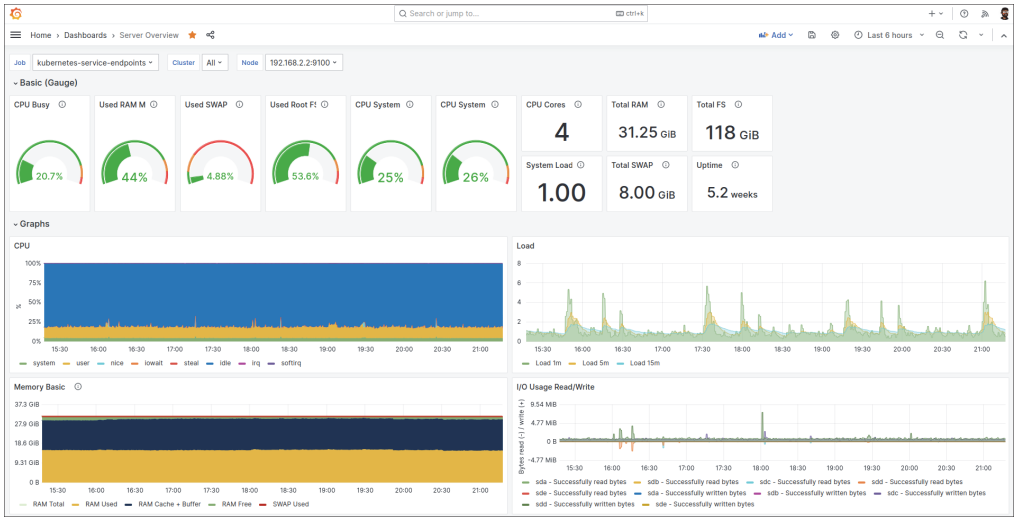

Abbildung 10.8 Ein exemplarisches Grafana-Dashboard eines Servers, der mit Prometheus überwacht wird

In Grafana lassen sich diverse Dashboards anlegen, die die Daten unterschiedlich visualisieren können. Die Qualität und die Flexibilität der verschiedenen Visualisierungen sind ziemlich hoch. Es gibt zahlreiche Möglichkeiten, wie Sie etwa in Abbildung 10.8 erkennen: Zeitserien, Balkendiagramme, einfache Statistiken, Pegeldiagramme, Tabellen, Kuchendiagramme oder auch Heatmaps. Es gibt noch etliche mehr; probieren Sie aus, wie Sie Ihre Daten am besten und intuitivsten anzeigen lassen können.

Die Visualisierungsoptionen stehen für die verschiedenen Datenquellen zur Verfügung; die große Kunst ist die Auswahl, welche Daten wirklich relevant sind und ständig im Blick behalten werden müssen. Durch die Breite an Konfigurationsmöglichkeiten wird viel Zeit benötigt. Die Entscheidung, welche Infos in Ihrer Umgebung wichtig sind, kann ich Ihnen nicht abnehmen, aber einige Hinweise aus der Praxis gibt es:

- Für den Start und die Arbeit mit Standardsoftware gibt es vorgefertigte Dashboards aus der Community unter *https://grafana.com/grafana/dashboards/*. Diese Dashboards können einfach importiert werden. Auch wenn die Qualität recht stark schwankt: Meistens findet sich dort eine gute Idee, auf der man eine eigene Visualisierung aufbauen kann.

© Rheinwerk Verlag, Bonn 2024

- Egal ob Sie »auf der grünen Wiese« starten oder ein Community-Dashboard nutzen: Betrachten Sie einmal eingerichtete Dashboards nicht als endgültige Lösung. Informationsbedürfnisse ändern sich, Fragestellungen kommen hinzu oder fallen weg. Sammeln Sie also Feedback, um die Darstellung anzupassen. Es kommt oft vor, dass man besser wieder bei null startet und eine neue Auswahl trifft.

  Das ist besonders dann relevant, wenn Dashboards für die eigene Anwendung mit eigenen Metriken entwickelt werden. Dabei müssen nicht nur die ausgewerteten Metriken immer wieder optimiert werden, sondern auch die Visualisierung. Sprechen Sie dazu mit allen, die die Dashboards nutzen.
- Überfrachten Sie die Dashboards nicht, sondern richten Sie besser unterschiedliche Oberflächen für unterschiedliche Zielgruppen ein. Je weniger Infos Sie auf einem Screen darstellen müssen, desto besser.

Das eigentliche Einbinden von Metriken geht inzwischen ziemlich leicht über den Metrics-Browser von der Hand, über den Sie sich zügig mit Autovervollständigungen die Metriken, Labels und sonstige Funktionen zusammenklicken können. Je nach Datenquelle ist die Benutzerführung allerdings nicht ganz so gut, wie sie heute etwa für Prometheus ist.

Reflexion

Aktuell führt an Grafana kein Weg vorbei. Praktikable und ausgereifte Alternativen gibt es nicht wirklich. Ein sehr junges Alternativprojekt ist *Perses*, (*https://github.com/perses/perses*) mit dem Dashboards als Code definiert werden können. Der praktische Einsatz ist allerdings noch Zukunftsmusik.

Sie werden schnell merken, dass Grafana viele nützliche Funktionen mitbringt, die das Erzeugen von Diagrammen einfach machen. Die Herausforderung ist eigentlich die Entscheidung, welche Infos wirklich dargestellt werden müssen.

10.3.4 Error-Tracking

Error-Tracking, auch Error-Monitoring oder Error-Logging genannt, ist ein Service, mit dem Fehler, die im System auftreten, an zentraler Stelle gesammelt und protokolliert werden. Wenn ein Benutzer also eine Anwendung nutzt und dort auf einen Fehler trifft, bei dem eine Exception geworfen wird, dann muss dieser Fehler festgehalten werden, und zwar mit Zeitstempel und weiteren Metadaten. Diese zentrale Fehlerliste dient dem Entwickler-Team als Startpunkt beim Beheben von Bugs.

Einer der bekanntesten Vertreter für das Error-Tracking ist *Sentry* (*https://sentry.io/for/error-monitoring/*). Clients und SDKs für die Fehlersuche sind als Open-Source-Software verfügbar und lassen sich schnell und einfach einbinden.

Sentry kann aber noch mehr als reines Error-Tracking und bietet auch Funktionen für die Performance-Messung und das Profiling an, denn Error-Tracking ist nur dann hilfreich, wenn es mit Traces und Logging angereichert wird. Und das ist genau der Weg von Observability-Plattformen, wo alles zusammenkommt, um den vollen Überblick zu erhalten.

Zu weiteren Anbietern des Error-Trackings gehören:

- Airbrake, *https://www.airbrake.io/error-monitoring*
- Bugsnag, *https://www.bugsnag.com/error-monitoring/*
- Datadog, *https://www.datadoghq.com/product/error-tracking/*
- integriertes Error-Tracking in GitLab, *https://docs.gitlab.com/ee/operations/error_tracking.html*
- HoneyBadger, *https://www.honeybadger.io/tour/error-tracking/*
- LogRocket, *https://logrocket.com/features/error-tracking-issue-management*
- Raygun, *https://raygun.com/platform/crash-reporting*

Dabei handelt es sich jedoch größtenteils um kommerzielle Dienste, die nicht so einfach zu nutzen sind wie Sentry.

Reflexion

Meiner Meinung nach ist kein Werkzeug wichtiger als das Error-Tracking. Für den Erfolg der DevOps-Idee ist es zentral, dass aus Fehlern gelernt wird.

Sie dürfen nicht verloren gehen, unter den Teppich gekehrt oder ignoriert werden. Organisieren Sie daher unbedingt ein System, das Ihnen hilft und alle Probleme zentral und hilfreich protokolliert.

10.3.5 Distributed Tracing

Beim Tracing geht es darum, dass man den Ablauf eines Requests nachvollziehen kann, mitsamt aller Daten, die dazugehören. Das ist grundsätzlich nichts Neues, ist im Laufe der Zeit aber komplexer geworden. Weil durch den Einsatz von Containern und der damit verbundenen Mikroservice-Architekturen inzwischen sehr viele Dienste verteilt laufen, kommen Sie mit klassischem Tracing, das sich nur auf einzelne Dienste beschränkt, nicht mehr weit.

Stattdessen brauchen Sie verteiltes Tracing, das einen Request über viele verschiedene Systeme verfolgen kann. Diese Abläufe möchte man nachvollziehen können, um etwaige Bottlenecks und Fehler zu finden.

© Rheinwerk Verlag, Bonn 2024

Um diese Daten zu erfassen, wird vor allem auf *OpenTelemetry* gesetzt. OpenTelemetry (*https://opentelemetry.io/*) ist ein Observability-Werkzeugkasten, um Telemetrie-Daten zu erzeugen und zu verwalten. Dazu gehören nicht nur Traces, sondern auch Metriken und Logs.

OpenTelemetry ist *kein* einzelnes Tool, sondern ein Framework und Toolkit, das in der eigenen Anwendung als Library eingebunden wird, um Telemetrie-Daten bereitstellen zu können. Um die Daten dann abzuspeichern, zu verwalten und zu durchsuchen, wird ein Backend benötigt, das mit diesen OpenTelemetry-Daten arbeiten kann.

Wenn Sie den Quellcode der zu überwachenden Anwendung nicht verändern können, bietet OpenTelemetry mit der sogenannten *automatischen Instrumentation* einen Weg, um an Telemetriedaten zu kommen. Wesentlich feiner können Sie vorgehen, wenn Sie den Quellcode verändern können und in Ihrem Code die OpenTelemetry-API aufrufen. Weitere Informationen dazu finden Sie unter *https://opentelemetry.io/docs/concepts/instrumentation/*.

Weitere Tracing-Backends

Eines der Backends für das Tracing ist *Jaeger*. Jaeger ist ein Projekt der Cloud Native Computing Foundation und hat dort den Status »Graduated«, genauso wie Prometheus und Kubernetes.

Es gibt noch zahlreiche weitere Tracing-Backends, darunter auch das bereits erwähnte Sentry oder *Tempo*, das aus dem Grafana-Projekt kommt und tief in Grafana integriert ist.

Mit dem Tracing starten

Die OpenTelemetry-Community hat eine eigene Demo-Applikation unter *https://opentelemetry.io/docs/demo/* entwickelt, die unterschiedliche Komponenten einbindet. Dort können Sie das Tracing einer Anwendung ausprobieren, ohne selbst etwas installieren oder instrumentieren zu müssen. Die Demo lässt sich entweder in einem lokalen Docker-Compose-Setup starten oder kann in einen Kubernetes-Cluster deployt werden. Die Demo-Applikation besteht aus mehreren Microservices. Diese stellen komplexe Abläufe für die Tracing-Analyse bereit – ein Webshop inklusive Checkout-Workflow.

Installieren Sie für die Auswertung zusätzlich Jaeger, um die Daten, die von der Demo-Applikation erzeugt werden, erfassen und auswerten zu können. Die Traces können dann über das Web-Interface von Jaeger betrachtet werden. Alternativ ist es eine gute Übung, Jaeger als Datenquelle in Grafana einzubinden, um einen zentralen Ort für alle Informationen zu haben.

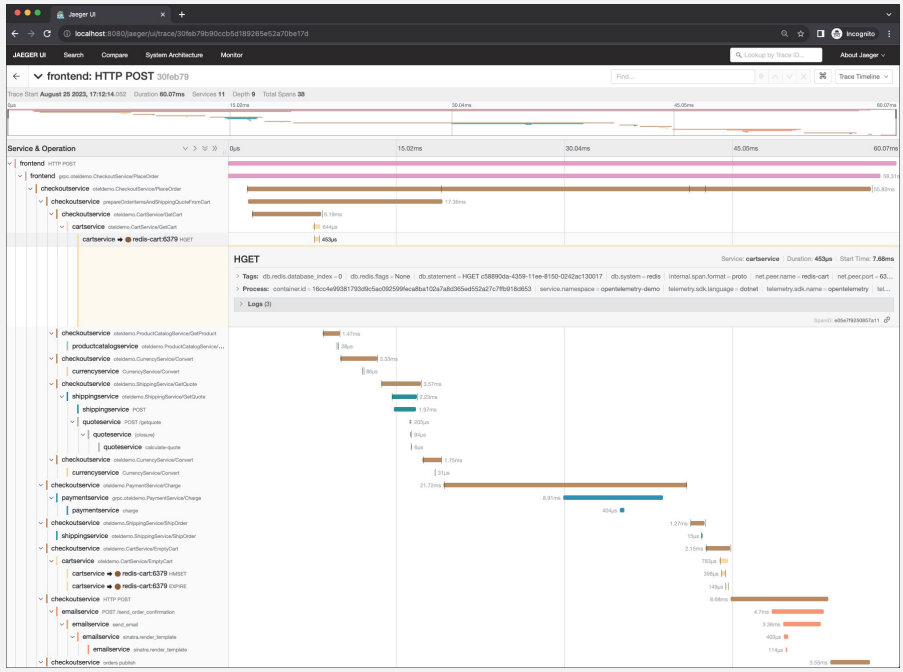

Abbildung 10.9 Visualisierung von Traces und Spans in Jaeger am Beispiel der OpenTelemetry-Demo und des Checkout-Service

10.3.6 Logging

Log-Server gibt es einige auf dem Markt. Viele von ihnen stehen unter einer Open-Source-Lizenz. Die Grundidee ist, dass die Logs von den verschiedenen Diensten und Anwendungen zu einem zentralen Dienst gesendet werden, damit die Logs dort gesammelt aufbewahrt werden können. Darauf lassen sich dann Analysen durchführen, die auf einer großen Anzahl an Quellen basieren.

Log-Server sind inzwischen meistens in den Observability-Plattformen zu finden, da Logs allein relativ langweilig sind und ihre wahre Stärke erst durch die Anreicherung mit Metriken und Traces entfalten.

Zu den bekannteren und verbreiteten Log-Servern gehören *Logstash* und *Graylog*. Grafana bietet mit *Loki* einen eigenen, relativ neuen Log-Server an. Log-Server kommen heutzutage allerdings kaum noch allein in freier Wildbahn vor. So ist eine Anbindung an Elasticsearch häufig unabdingbar, zumindest wenn man Graylog und Logstash anschaut.

Logstash ist auch Teil des *Elastic Stacks*. Früher wurde vom ELK-Stack gesprochen: Elasticsearch, Logstash und Kibana. In *Elasticsearch* sind die Daten gespeichert und

© Rheinwerk Verlag, Bonn 2024

darin werden auch die Daten durchsucht; mit *Logstash* werden die Logs eingesammelt und verwaltet, und *Kibana* ist für die Darstellung zuständig. Teil des Elastic Stacks ist auch noch *Beats*, ein Tool, das die Daten zu Logstash und Elasticsearch schiebt. Mittlerweile verwendet Elastic allerdings Agenten als Kollektoren und bietet dafür auch eine automatisierte Verwaltung und noch einiges mehr an.

Im Cloud-native-Umfeld ist *Fluentd* ziemlich verbreitet und ein Projekt der Cloud Native Computing Foundation: Es ist ein Logging-Layer zwischen den System- und Anwendungslogs und Werkzeugen zur Datenhaltung wie Elasticsearch, MongoDB oder Hadoop.

Auf kommerzieller Ebene gibt es ebenfalls etliche Lösungen. Die wohl verbreitetste ist *Splunk*. Auch bei Splunk ist das Log-Management nur eines der Produkte, die die Firma anbietet. Andere bekannte Anbieter sind Elastic, Dynatrace, SolarWinds, Datadog, Logz.io und NewRelic.

10.3.7 Service-Meshes

Container-Systeme und Softwarearchitekturen, die auf Microservices basieren, sind fraglos nützlich und sorgen dafür, dass Dienste viel einfacher und schneller bereitgestellt werden können. Andererseits müssen diese unterschiedlichen Dienste verwaltet und überwacht werden. Dabei geht es unter anderem um Routen, Timeouts, Verschlüsselungen, Authentifizierung und Autorisierungen, aber auch Logs und Metriken zwischen den Microservices sind relevant. Alles zusammengenommen, spricht man von einem *Service-Mesh* innerhalb einer Container-Orchestrierung wie Kubernetes.

Jeder Microservice arbeitet dann mit einem Proxy, der neben dem eigentlichen Microservice läuft. Dazu gesellen sich, unabhängig von der jeweils verwendeten Implementierung, eine Control Plane, die etwa mit Kubernetes-APIs oder anderen Infrastruktur-Services sprechen kann.

Bekannt sind vorwiegend drei Projekte: Linkerd, Istio und Cilium. *Linkerd* ist für die Nutzbarkeit und die Performance optimiert, während *Istio* mit viel mehr Features daherkommt und auch deutlich anpassungsfähiger ist. *Cilium* ist eigentlich ein ganzer Observability-Toolstack und implementiert Service-Meshes mithilfe von eBPF (*extended Berkley Packet Filter*). Alle Tools laufen unter Kubernetes; Linkerd und Cilium setzen sogar eine Kubernetes-Umgebung voraus.

Was bringt ein solches Service-Mesh nun genau, und wofür ist es bei alltäglichen Arbeiten gut? Beispielsweise lässt sich das Load-Balancing gut über Service-Meshes konfigurieren und testen, um den Traffic von den Instanzen der Microservices zu den anderen zu leiten.

Persönliches Exemplar für Thomas Bartholomäus

Und das ist nur ein Beispiel aus dem Routing. Auch Themen rund um die Resilienz sind relevant, etwa wenn ein Service nicht rechtzeitig antwortet, dann der Request erneut ausgeführt wird, aber an einen anderen Service gleichen Typs gerichtet wird.

Die nächsten beiden Punkte sind *Security* und *Observability*. Erstere ist insbesondere für Authentifizierungs- und Verschlüsselungsfunktionen relevant. Letztere führt dazu, dass es noch ein weiteres Dashboard gibt, auf dem man die Kommunikation zwischen den Microservices sehen kann. Das betrifft Routen ebenso wie Probleme, die darin auftauchen können.

Bevor Sie nun direkt loslegen wollen und ein Service-Mesh in Ihrer Kubernetes-Infrastruktur planen: Bedenken Sie, dass Sie damit einen weiteren Layer an Komplexität in der Software-Architektur erzeugen! Das ist nur dann sinnvoll, wenn vollkommen auf Microservices gesetzt wird. Wenn ein monolithischer Ansatz gewählt wird, dann wird kein Service-Mesh gebraucht.

Zudem bringt das Mesh deutliche Performance-Einbußen und eine höhere CPU-Last mit sich, da die Verwaltungsaufgaben schließlich auch noch erledigt werden wollen.

Reflexion

Service-Meshes sind ein komplexes Thema. Ich empfehle ihren Einsatz erst dann, wenn Sie schon gute Kenntnisse zu den übrigen Themen dieses Kapitel besitzen. In der Praxis ist ein vollumfängliches Setup eher selten zu finden.

10.3.8 Observability-Plattformen

Observability-Plattformen sind noch relativ jung. Wie zuvor bereits erwähnt, geht es bei ihnen um die vollständige Sammlung von Observability-Daten (Metriken, Traces, Logs etc.) und deren Analyse und um Abfragen in einer Gesamtsicht auf die Systeme.

Spannend ist, dass viele der Tools, die es in dieser Landschaft gibt, kaum neu sind. Früher fiel ihre Tätigkeit eher unter den Begriff *Application-Performance-Monitoring*, heute spricht man von *Observability*, was den ursprünglichen Aspekt etwas erweitert.

Beim Blick auf die Tools fällt primär eines auf: Es gibt verdammt viele Tools von verschiedenen Anbietern. Einige sind ziemlich ausgereift und sehr mächtig, stehen allerdings nicht unter einer Open-Source-Lizenz und sind häufig nur als SaaS-Lösung verfügbar.

Andere Lösungen sind wiederum Open Source, und bei ihnen wird mal mehr und mal weniger Arbeit benötigt, um das Setup einzurichten. Nicht umsonst wurden die einzelnen Aspekte schon größtenteils in diesem Kapitel angerissen.

© Rheinwerk Verlag, Bonn 2024

Schaut man sich die rein kommerziellen Tools an, erkennt man etliche Tools wieder, die zuvor unter Application-Performance-Monitoring zu finden waren: AppDynamics, New Relic, Datadog, Splunk, Instana ... hier gibt es etliche. Einige dieser Anbieter waren mal eigenständige Firmen, wurden im Laufe der Zeit aber von größeren Firmen geschluckt.

AppDynamics gehört jetzt etwa zu Cisco, Instana zu IBM, und New Relic wurde auch schon von einem Private-Equity-Unternehmen aufgekauft. Das ist insbesondere dann wichtig zu beachten, wenn man einigen Firmen mehr oder weniger vertraut, denn all diese Tools sind im Wesentlichen ausschließlich als SaaS-Lösung verfügbar, bei der man die Daten aus den eigenen Händen und an einen US-Konzern gibt.

Ein relativ neuer Player ist die Firma *Honeycomb.io* mit ihrem ebenfalls SaaS-only-Tool *Honeycomb*. Die Entwickler bei Honeycomb.io haben entscheidend das Thema und den Begriff Observability mitgeprägt. Von ihnen stammt auch das Buch »Observability Engineering«, das im O'Reilly Verlag erschienen ist (*https://www.oreilly.com/library/view/observability-engineering/9781492076438/*).

Was sich trotz Closed-Source-Solutions durchgesetzt hat, ist hingegen *OpenTelemetry*. Alle großen und kleinen Player am Markt unterstützen mittlerweile OpenTelemetry, sodass man zumindest bei der Instrumentierung auf eine freie Lösung setzen kann.

Aus der Open-Source-Ecke hingegen gibt es mit *Elastic* und *Grafana* zwei Firmen, die auf ursprüngliche Open-Source-Software setzen und ihre kommerziellen Produkte auf dieser basieren lassen. Bei Elastic gab es einen Lizenz-Wechsel, durch den das Tool nicht mehr als Open Source gilt. Stattdessen gab es eine Abspaltung namens *OpenSearch*.

Sowohl der *Elastic Stack* als auch der Stack von Grafana lassen sich auf eigener Hardware installieren und managen. Der Elastic Stack, früher auch als ELK-Stack bekannt, besteht aus Elasticsearch, Kibana, Beats und Logstash, wie bereits im Abschnitt zu den Log-Servern erwähnt wurde.

Grafanas Stack hingegen wird auch *LGTM-Stack* genannt. Die Abkürzung LGTM ist primär aus Merge und Pull Requests bekannt, wenn im Review ein Okay mit »Looks good to me!« gegeben wird. Im Grafana-Kontext steht das Kürzel aber für Loki, Grafana, Tempo und Mimir.

Während *Loki* ist der Log-Server ist, ist *Grafana* das bekannte Tool für die Datenvisualisierung, das die verschiedenen Backends unter einer Benutzeroberfläche vereint und nutzbar macht. *Tempo* ist die Lösung für Traces und *Mimir* die Lösung für Metriken. Mimir ist letztlich eine Langzeit-Storage-Möglichkeit von Prometheus, die stärker auf horizontale Skalierbarkeit und Hochverfügbarkeit setzt, was Prometheus von Haus aus nicht kann.

Persönliches Exemplar für Thomas Bartholomäus

Konkurrenz belebt bekanntlich das Geschäft, und so wird man in den nächsten Jahren innovative Features sehen, die Observability-Praktiken und DevOps-Workflows noch effizienter machen. Als Anwender hat man die Qual der Wahl, ob man SaaS-Plattformen einkauft oder doch Maintenance-Aufwand betreibt und in selbst gehostete Observability-Plattformen investiert. Auf jeden Fall sollten Sie sich Zeit für die Evaluierung nehmen.

Auch Themen wie *AIOps*, also AI-gestützte Hilfe beim Debuggen von Problemen, findet man immer mehr, und diese könnten ein Entscheidungskriterium für DevOps-Teams darstellen. Einen Einblick in AIOps gibt es in Abschnitt 14.2.

Reflexion

Es ist spannend zu sehen, wie viele Firmen ihre Produkte umgebaut und umgelabelt haben, damit »Observability« auch für sie passt. Das muss nicht schlecht sein, denn die Grundidee – ganz unabhängig vom Marketing – ist sehr gut. Hier sollten Sie sich eher entscheiden, ob Sie so ein Setup selbst mit Open-Source-Software hosten und verwalten wollen oder auf einen Dienst von einem SaaS-Provider setzen.

Wenn Sie sich die Tools des ganzen Buches anschauen, dann ist der Bereich Observability am jüngsten. Hier wird in den nächsten Jahren also sicherlich noch einiges passieren. Ein Innovationstreiber hierfür ist das angesprochene eBPF (*extended Berkley Packet Filter*).

10.3.9 Monitoring und Observability bei schick-gekleidet.de

Im Rahmen der Modernisierung des Toolstacks war es den Verantwortlichen bei *schick-gekleidet.de* wichtig, auf moderne und flexible Tools zu setzen. Diese sollten allerdings ausgereift sein.

Durch die Cloud-Strategie inklusive Kubernetes lag es daher nahe, direkt auf Prometheus zu setzen. So konnten sowohl die Exporter von bestehenden Services genutzt werden als auch schnell eigene Exporter geschrieben werden, sodass Business-Metriken bereitgestellt werden konnten. Für die Visualisierung kam Grafana zum Einsatz.

Bei der Observability-Plattform war es wichtig, die Hoheit über die Daten zu behalten. Es sollte also eine selbst gehostete Open-Source-Lösung werden, wobei allerdings festgestellt wurde, dass die Werkzeuge der genutzten Cloud-Umgebung bereits viele Features mitbringen. Um die Komplexität gering zu halten, wurde daher auf den Einsatz eines eigenen Elastic Stacks LGTM-Stacks verzichtet, bis sich dafür ein echter Use-Case aufdrängt. Auch Service-Meshes werden erst einmal ignoriert.

© Rheinwerk Verlag, Bonn 2024

10.4 Verfügbarkeit

Eng verbunden mit den Fragen nach Monitoring und Observability ist der Begriff der *Verfügbarkeit*. Dabei geht es wieder um das System als Ganzes. Und da kommt immer wieder die Frage auf, woran man denn die Verfügbarkeit genau festmacht und wer für sie verantwortlich ist.

Häufig wird die Verfügbarkeit anhand der reinen Erreichbarkeit des Systems festgemacht. So wird häufig mit der Anzahl der *Neunen* gemessen und gewertet. Eine Verfügbarkeit von 99,99 % bedeutet etwa, dass ein Dienst eine Downtime von 52,60 Minuten pro Jahr erlaubt. Das sagt allerdings noch nichts darüber aus, wie gut der Dienst überhaupt erreichbar ist. Ein verfügbarer Dienst, der aber so langsam ist, dass er unbrauchbar ist, hilft am Ende niemandem. Aber dafür sieht seine Verfügbarkeit auf dem Papier gut aus!

Um die Verfügbarkeit genauer zu definieren, müssen *Service-Level* festgelegt werden. Besonders drei Begriffe müssen dabei voneinander abgegrenzt werden.

- Service-Level-Agreement (SLA)
- Service-Level-Objectives (SLO)
- Service-Level-Indicators (SLI)

10.4.1 SLA: Service-Level-Agreements

Service-Level-Agreements gibt es nicht nur in der IT, sondern auch in anderen Branchen. Letztlich ist ein solches Agreement nicht viel mehr als ein Rahmenvertrag, in dem festgelegt wird, wie ein Dienstleister dem Auftraggeber die Leistungen zusichert. Im Falle eines Webdiensts geht es um den Vertrag zwischen der Firma und den Nutzern, bei einer Anwendungssoftware ist es ein Vertrag zwischen dem Hersteller und der Firma, die die Software einsetzt.

In einem SLA werden verschiedene Kennzahlen definiert, etwa die zuvor angesprochene Verfügbarkeit in Form einer Prozentzahl oder die Reaktionszeit, die festlegt, wie lange eine Störung anhalten darf.

In einem SLA werden immer auch Vertragsstrafen festgehalten. Es ist also ein wichtiges Werkzeug für die Haftung und die Frage, wer im Fehlerfall für die Behebung des Schadens zuständig ist und ab wann eine Leistung als nicht erbracht gelten kann. Wenn es so weit kommt, sind SLAs eine wichtige Basis in einem Rechtsstreit, in dem es um Schadensersatz oder den Ausfall von Leistungen geht.

SLAs gibt es jedoch nicht nur zwischen verschiedenen Firmen. So gibt es häufig auch SLAs innerhalb einer Firma, wenn das eine Team – oder die andere Firma im Konzerngeflecht – einen Dienst für andere Teams bereitstellt. In diesem Fall fließt intern kein oder weniger Geld, wenn die SLAs nicht eingehalten werden, es lauern aber

Persönliches Exemplar für Thomas Bartholomäus

immense Konfliktpotenziale. Schnell führen SLAs dazu, dass die Verantwortung für Misserfolge hin- und hergeschoben wird. SLAs, die nur für interne Verpflichtungen relevant sind, werden auch *Operating-Level-Agreements* genannt. Diese sind nur intern und liegen keiner vertraglichen Vereinbarung zugrunde.

Wenn etwa ein Server durch ein ungeschickt programmiertes Feature überlastet wird und deswegen der Webshop nicht mehr erreichbar ist, fangen schnell die Diskussionen an: War es die Schuld der Entwickler? Oder des Netzwerkteams, das ja die zugesicherte Verfügbarkeit nicht einhalten konnte?

Solche Diskussionen sind nur selten zielführend und sollten im Rahmen eines Blameless Post-Mortems diskutiert werden. Anstatt auf dem SLA herumzureiten, gilt es, gemeinsam an den Lösungen zu arbeiten, damit beide Seiten aus den Fehlern lernen und ihre Dienste verbessern können.

Im DevOps-Kontext muss also hinterfragt werden, ob SLAs noch sinnvoll sind. Eine Verfügbarkeit von 99,999 % liest sich zwar auf dem Papier hervorragend, ist aber nicht alles. Am Ende kann es trotzdem bedeuten, dass die Nutzer des Systems nicht zufrieden sind. Und das kann mit SLAs nicht abgebildet werden, da diese schlicht für Anwälte existieren.

Das nächste Argument gegen SLAs ist, dass eine möglichst hohe Verfügbarkeit unter Umständen gar nicht erstrebenswert ist. Oder, genauer gesagt: Erstrebenswert ist sie schon, aber ihre Kosten sind zu hoch. Je mehr »Neunen« enthalten sind, desto kürzer wird die Möglichkeit einer Downtime. 99,999 % bedeutet, dass wir von 5,26 Minuten *pro Jahr* reden.

Aus Engineering-Sicht ist das verdammt schwierig umzusetzen, und das führt letztlich auch dazu, dass Verbesserungen gar nicht oder nur mit extrem hohem Aufwand durchgeführt werden. Nehmen wir an, dass mit einem Eingriff in die Infrastruktur die Zugriffszeiten für die Endnutzer halbiert werden könnten. Dazu muss der ganze Dienst aber für 30 Minuten abgeschaltet und in einen Wartungsmodus versetzt werden. Ein SLA mit fünf Neunen würde dieses Update unmöglich machen, was für alle Seiten frustrierend ist.

Absolute Verfügbarkeit

Es gibt selbstverständlich Beispiele für Dienste, die absolute Verfügbarkeit benötigen. Denken Sie etwa an medizinische Angebote oder den Notruf, aber auch der Empfang der BBC ist für das Fortbestehen der Menschheit elementar wichtig: Wenn britische Atom-U-Boote den Radiosender BBC 4 nicht mehr empfangen können und auch sonst keine Verbindung zur Heimat mehr besteht, müssen die *Letters of last resort* (*https://en.wikipedia.org/wiki/Letters_of_last_resort*) geöffnet werden.

© Rheinwerk Verlag, Bonn 2024

Es muss klar sein, dass eine solche Verfügbarkeit teuer erkauft wird. Jede weitere Neun kostet exponentiell mehr Geld und Aufwand als die vorherige. Bedenken Sie dies, wenn Sie das nächste Mal SLAs verhandeln müssen.

Es gibt aber auch weitere Beispiele, die nicht direkt an der Verfügbarkeit hängen. So gibt es häufig Vereinbarungen, die besagen, dass bestimmte Fehler innerhalb einer bestimmten Zeit – sagen wir 24 Stunden – gefixt werden müssen.

Und hier können gleich mehrere Probleme auftreten. Zunächst stellt sich die Frage, wann diese 24 Stunden anfangen: Ab wann wird der Fehler gemeldet? Aber was macht man, wenn das betroffene Team den Fehler gar nicht reproduzieren kann, weil wichtige Informationen fehlen und es mehr als 24 Stunden dauert, bis diese nachgeliefert werden?

Sie sehen anhand dieser Beispiele: Ein SLA trägt nicht unbedingt dazu bei, dass daran gedacht wird, was für den Endnutzer relevant ist. Und um das zu bewerkstelligen, muss man eben den gesamten Kontext betrachten. Und genau hier kommen Service-Level-Objectives ins Spiel.

10.4.2 SLO: Service-Level-Objectives

Während ein SLA mehr ein formelles und rechtliches Konstrukt ist, verfolgt das *Service-Level-Objective* ein anderes Ziel, nämlich ein Leistungsziel, das vom Dienstleister selbst festgelegt wird. Man kann es eher als eine interne Zielvorgabe verstehen, die auf den Bedürfnissen der User basieren sollte. Ein SLO kann etwa eine bestimmte Verfügbarkeit oder Leistung des Systems festlegen, die der Dienstleister anstrebt.

Bei einem SLO geht es explizit *nicht* darum, dass ein rechtliches Konstrukt existiert, sondern dass die tatsächlichen Metriken festgelegt werden. Während ein SLA also verbindlich ist und rechtliche Auswirkungen festlegt, ist ein SLO flexibler. Der Fokus liegt nicht auf der rechtlichen Absicherung im Schadensfall, sondern auf der Frage, wie definiert werden kann, was für den Kunden bzw. Endnutzer des Systems den größten Nutzen bringt.

Auch bei SLOs gibt es Herausforderungen und typische Fehler, die vermieden werden müssen. Häufig kommt es vor, dass einige Objectives schlecht messbar sind, zu vage formuliert wurden, sich widersprechen oder sich gegenseitig blockieren. (Ein *einfach bedienbarer, aber sicherer* Webshop kann schon ein Ding der Unmöglichkeit sein ...)

Wichtig ist daher, dass SLOs klar und einfach definiert sind, um die Messbarkeit zu gewährleisten. Und auch wenn SLOs meistens nicht direkt an Sanktionen geknüpft sind, sollten sie natürlich trotzdem eingehalten werden, denn ihre Werte stehen fast immer im Zusammenhang mit den SLAs.

Persönliches Exemplar für Thomas Bartholomäus

10.4.3 SLI: Service-Level-Indicators

Ein *Service-Level-Indicator* ist der tatsächlich messbare Wert, der angibt, wie es um die Performance und die Verfügbarkeit des Systems oder von Teilsystemen bestellt ist. Die Frage ist hier also nicht wie bei SLAs, ob wir unsere Versprechen einhalten, oder wie bei SLOs, was die Ziele sind, sondern wie wir uns gerade schlagen.

Wenn wir beispielsweise in einer SLO definiert haben, dass die Reaktionszeiten des Webshops unter 400 ms liegen sollen, wäre der entsprechende SLI die gerade aktuelle Geschwindigkeit, die etwa bei 200 ms liegt. Das würde bedeuten, dass das SLO eingehalten wird. Was wiederum heißt, dass das verantwortliche Team sein SLA ebenfalls einhält.

Wie Sie erkennen, bauen diese drei Punkte aufeinander auf. Ein Service-Level-Indicator ist Bestandteil des SLOs, das wiederum Bestandteil eines SLAs ist. Daher gilt auch hier, dass man SLIs einfach hält und sie nicht verkompliziert. Definieren Sie eine kleine, aber sinnvolle Anzahl an Werten, die nützlich für die Endnutzer sind, denn darauf kommt es am Ende schließlich an.

10.4.4 Error-Budgets

In einer idealen Welt gibt es keine Ausfälle, und man könnte im SLA eine Verfügbarkeit von 100 % garantieren. Das ist allerdings völlig realitätsfern. Incidents verbunden mit Ausfällen wird es immer geben. Einigen Sie sich also mit Ihren Vertragspartnern auf realistische Vorgaben im SLA.

Durch gutes Monitoring sollten Sie einen Überblick darüber haben, mit wie vielen Ausfällen und Problemen Sie rechnen müssen. Historische Daten können zwar nicht ohne Weiteres in die Zukunft extrapoliert werden, lassen aber einen guten Schluss zu, was realistisch ist.

Nun können Sie aus der Differenz zwischen dem aktuellen Stand der SLIs und dem, was Ihnen das SLO und besonders das SLA erlauben, ableiten, wie viel *Spielraum* der Dienst hat. Dies wird *Error-Budget* genannt. Ein Blick auf Abbildung 10.10 zeigt, wie es aussehen kann.

Mit dieser Rechnung haben Sie eine quantifizierbare Angabe, aus der Sie ableiten können, wie viel Zeit Sie jeden Monat für Wartungsfenster und Routinearbeiten einplanen können. Bleiben wir beim einfachsten Beispiel, nämlich bei der Forderung nach einer Verfügbarkeit von 99,9 %. Dieses SLA entspricht einer erlaubten Downtime von 8 Stunden, 45 Minuten und 57 Sekunden pro Jahr. Auf den Monat umgerechnet, sind das 43 Minuten und 50 Sekunden. Gar nicht so wenig.

Genau diese Zeit ist das gesamte Error-Budget für die Verfügbarkeit des Diensts.

© Rheinwerk Verlag, Bonn 2024

Abbildung 10.10 Wenn der Wert des aktuellen SLIs über dem des SLAs für ein SLO liegt, ist ein Error-Budget verfügbar, das genutzt werden kann.

Reflexion

Fehler, Performance-Probleme und weitere Aspekte, die auf Unzulänglichkeiten hinweisen, sind immer etwas kritisch zu betrachten, wenn man die Kultur anpassen möchte. Niemand gibt gerne Fehler zu, das war auch schon im vorherigen Kapitel bei Blameless Post-Mortems relevant.

Die technische Umsetzung ist daher erneut nur ein Teil der Lösung. So können Sie mit den richtigen Tools aus dem Bereich der Observability zwar die Visibilität erhöhen, doch ist das nur ein Baustein für eine erfolgreiche Zusammenarbeit in einem cross-funktionalen Team.

Sehen Sie zu, dass Sie nützliche Metriken sammeln und für das Team sichtbar machen. Das betrifft sowohl operative Metriken als auch Business-relevante Metriken. Wenn das gewährleistet ist, dann sollten Sie schauen, dass Sie Error-Budgets spezifizieren und nutzen.

10.5 Fazit

In diesem Kapitel ging es um den vermeintlich letzten Teil des DevOps-Lifecycles, das Monitoring. Erkenntnisse aus Monitoring und Observability dienen als Input für die nächste Phase im DevOps-Lifecycle: Der Planung.

Wie auch bei allen anderen Kapiteln ist ebenfalls beim Monitoring eins wichtig: Die Visibilität in Daten und Fakten des Systems für alle beteiligten Personen. Nur so, kann effizient und zielgerichtet weitergearbeitet werden.

Persönliches Exemplar für Thomas Bartholomäus

Diese Informationen sollen und müssen in die nächsten Iterationen einfließen, um Probleme und Herausforderungen an der Wurzel zu behandeln, und zwar gemeinsam in einem cross-funktionalen Team.

Die Wahl der Tools im Monitoring und Observability-Feld ist spannend, denn in diesem Bereich tut sich aktuell sehr viel. Nichts falsch machen Sie mit dem Einsatz von Prometheus und Grafana, behalten Sie aber neue Entwicklungen im Blick, damit Ihnen nichts entgeht.

© Rheinwerk Verlag, Bonn 2024

Kapitel 11
Security und Compliance

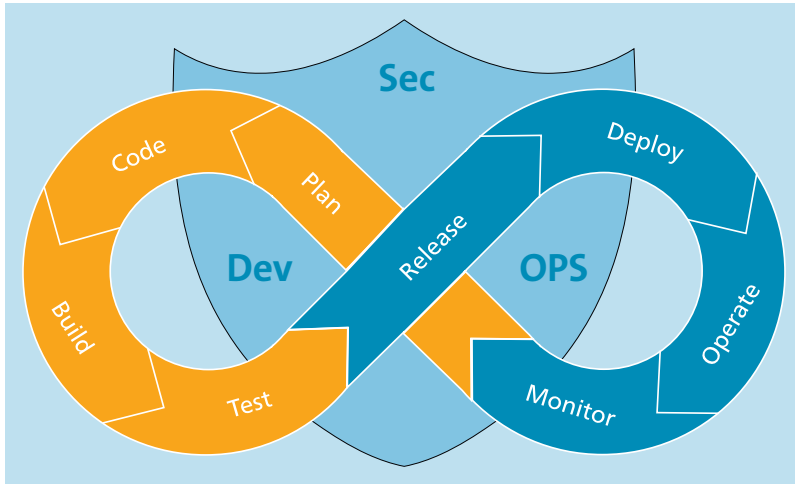

Abbildung 11.1 Die Sicherheit spielt in allen DevOps-Phaen eine wichtige Rolle.

In einem Team aus Entwicklung, Qualitätssicherung und Betrieb fehlt noch eines: das Security-Team. Je größer die Firma oder Organisation ist, desto wahrscheinlicher ist es, dass ein solches Team existiert, das seine Arbeit nachgelagert ausführt und nicht bereits Teil des Prozesses ist. Je kleiner die Firma ist, desto wahrscheinlicher ist es, dass es Subject-Matter-Experten im Bereich Security gibt, wobei es leider allzu häufig vorkommt, dass dieser Bereich entweder komplett ignoriert oder stark vernachlässigt wird.

Dieses Kapitel geht auf die verschiedenen Security-Mechanismen und Konzepte ein, die durch die DevOps-Idee besonders formuliert werden. Darunter fallen die Integration des Security-Teams in die Entwicklungsabteilung sowie die Einbindung von entsprechenden Tools und Reviews in die CI/CD-Pipeline.

Als Paradebeispiel werden in diesem Kapitel Lücken dienen, die Firmen mit gescheiten Prozessen und passendem Tooling schnell beheben können, da sie einen guten Überblick über die Abhängigkeiten haben und entsprechend schnell Updates und Bugfixes ausrollen können.

Persönliches Exemplar für Thomas Bartholomäus

Der Fokus liegt dabei auf der *Application-Security*. Dieser Teil von DevSecOps ist allerdings mehr als nur der reine Blick auf die Sicherheit der Anwendung. So ist auch ein Blick auf die *Supply-Chain-Security* nötig. Daneben ist die praktikable Umsetzung von Compliance-Richtlinien im Unternehmen ebenfalls ein Schwerpunkt in diesem Kapitel.

Am Rande werden wir immer wieder Ausflüge in die Security der Infrastruktur machen, aber das ist nicht der Fokus dieses Kapitels, da die Planung und Wartung der Infrastruktur meistens nicht Aufgabe des Software-Produkt-Teams ist.

11.1 Sicherheit stört den agilen Wasserfall

Bei *schick-gekleidet.de* gab und gibt es ein Security-Team, das sich um die Sicherheit der Anwendung sowie der Infrastruktur kümmert. Security ist ein wichtiges Thema, denn Lücken, die ausgenutzt werden und zu Datenlecks führen, sorgen nicht nur für Vertrauensverlust bei den Kunden. Sie können auch zu gravierenden Strafen führen, die auf jeden Fall vermieden werden sollten.

Beispielsweise schreibt die DSGVO empfindliche Strafzahlungen vor, wenn nicht sorgsam mit Daten umgegangen wird, und ein Versicherungsschutz ist nur dann möglich, wenn die Infrastruktur gewisse Standards und Regeln einhält. Inzwischen sollte sich also herumgesprochen haben, dass die Sicherheit wichtig ist.

ISO 27001 und KRITIS

Mit der ISO 27001 gibt es eine internationale Norm für Informationssicherheitssysteme, nach der sich Unternehmen zertifizieren lassen können. Noch einen Schritt weiter geht KRITIS. Unter diesem Schlagwort schreibt das Bundesamt für Sicherheit in der Informationstechnik (BSI) den Betreibern von sogenannten *kritischen Infrastrukturen* bestimmte Standards für die Verfügbarkeit und Sicherheit ihrer Dienste vor. Wenn Sie etwa ein Wasserwerk betreiben, sind Sie durch staatliche Vorgaben verpflichtet, sich um die Sicherheit zu kümmern. Für die Betreiber von Webshops und anderen Angeboten gelten diese Regeln nicht. Dort geht es oftmals wie im Wilden Westen zu, wenn es um die Sicherheit geht.

Um Sicherheit zu gewährleisten, hat das Security-Team bei *schick-gekleidet.de* im Wesentlichen drei Aufgaben:

- überprüfen und sicherstellen, dass die Anwendung selbst sicher ist
- überprüfen und sicherstellen, dass die Infrastruktur, auf der die Anwendung läuft, sicher ist
- überprüfen und sicherstellen, dass die Infrastruktur rund um das Netzwerk sicher ist

© Rheinwerk Verlag, Bonn 2024

Mit diesen drei Aufgaben ist das Security-Team nicht nur stark ausgelastet, sondern häufig auch überlastet, und es können nicht alle Aufgaben zufriedenstellend gelöst werden, da die Personaldecke zu dünn ist. Wie Sie anhand der Aufgaben schon erkennen können, muss das Security-Team mit fast allen zusammenarbeiten: mit dem Entwicklungsteam, mit dem Betriebsteam und mit dem Infrastruktur- und Netzwerkteam.

Alle drei Aufgaben sind aufwendig und verursachen viel Arbeit, die niemals abgeschlossen ist. Kontinuierlich muss geprüft werden, ob neue Features und Änderungen Sicherheitslücken beinhalten. Auch bei Änderungen an der Infrastruktur oder dem Netzwerk ist das Security-Team jedes Mal involviert und muss im Austausch mit den anderen Teams prüfen, ob alles sicher ist.

Dazu hat das Team diverse Regeln aufgestellt, insbesondere was den Netzaufbau sowie die Abschottung der verschiedenen Netze voneinander angeht. In diesen Regeln wird auch erläutert, welche Firewall-Regeln existieren und wie die Firewalls zwischen den Servern einzurichten sind.

Für die übrigen Teams bestand die Zusammenarbeit mit den Sicherheitsleuten also hauptsächlich aus der Bearbeitung langer Checklisten, die bei jeder Änderung erneut geprüft werden müssen. Nachfragen und Feedback sorgten nur für weitere Blockaden, da nicht immer alles vollständig dokumentiert wurde. Diese Checklisten landeten dann bei dem Team, das die Compliance-Regeln definiert.

Der Workflow sah dann meist wie folgt aus: Das Entwicklungsteam entwickelte während der langen Entwicklungsphase wie gehabt seine Anwendung weiter. Da die Entwicklungszeit im agilen Wasserfall-Modell etwa sechs Monate beträgt, erfolgen in dieser Phase etliche Änderungen, die dann auf erst auf eine Staging-Umgebung deployt werden, bevor es dann auf die Produktivumgebung deployt wird. So weit gab es erst mal keine Neuigkeiten.

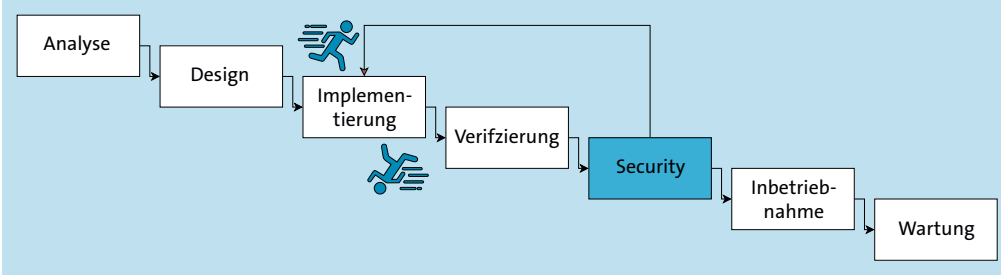

Abbildung 11.2 Das Security-Team kommt im Wasserfall-Modell kurz vor der Inbetriebnahme zum Zuge und blockiert häufig, was dazu führt, dass es im Projekt zurück zur Implementierungsphase geht.

Vor jedem großen Deployment des Online-Shops war das Security-Team das letzte Team im Prozess. Es prüfte jedes Mal, ob es Sicherheitslücken finden konnte. Die Teammitglieder nutzten dafür verschiedene Techniken und Methoden. Während dieser Phase prüften sie viele Fragen, unter anderem folgende:

- Lassen sich durch einen einfachen Penetrationstest Lücken finden? Und lassen sich diese praktisch ausnutzen?
- Ist beispielsweise SQL-Injection in den Formularen möglich?
- Sind in den genutzten Abhängigkeiten bereits Sicherheitsprobleme dokumentiert, die auch den eigenen Webshop betreffen?

Einige dieser Tests lassen sich automatisieren. Allerdings bedeuteten die großen Entwicklungsschritte von 6 Monaten, dass bei jedem Deployment alles überarbeitet werden musste. Denken Sie allein an das Prüfen der Abhängigkeiten: Da das Entwicklungsteam bisher weder die Paketmanager der jeweiligen Programmiersprache genutzt hat noch die Abhängigkeiten sauber und aktuell dokumentiert hat, musste das Security-Team häufig die verwendeten Abhängigkeiten manuell zusammensuchen und prüfen.

Da die Entwicklungsschritte so groß waren und es dementsprechend schwierig war, einen funktionierenden und reproduzierbaren Build zu bekommen, scheuten sich die Entwickler davor, die Abhängigkeiten anzufassen: »*Never touch a running system*, die Library tut doch schon seit 2013, was sie soll.«

Beim Security-Team sorgten solche Funde immer für das blanke Entsetzen, denn so fanden eine Menge bekannter und dokumentierter Sicherheitsprobleme ihren Weg in die Anwendung. Aber auch wenn die Entwickler alles richtig machen wollten und zu Beginn der Entwicklungsphase alles aktualisierten, sah das 6 Monate später schon wieder ganz anders aus, wenn das Deployment bei den Security-Leuten landete.

Das Security-Team nahm also vor dem anstehenden Deployment der neuen Version eine tiefgehende Überprüfung vor, an deren Ende eine Einschätzung stand, wie es um die Security bei diesem Release bestellt war. Da *Sicherheit* für das Entwicklungsteam keinen Stellenwert besaß, existierten stets etliche Lücken und Probleme: Das konnten die veralteten Abhängigkeiten sein; oftmals waren es aber auch schlicht Fehler im Quellcode, die korrigiert werden mussten.

Die Freigabe wurde daher nie im ersten Versuch gegeben, was für die Entwickler einen hohen Mehraufwand bedeutete, da sie sich nun wieder mit Code beschäftigen mussten, den sie vor 6 Monaten zum letzten Mal gesehen hatten. Und als Hilfestellung bekamen sie lediglich eine Liste an Dingen, die bemängelt wurden.

Dieser Vorgang wiederholte sich vor einem Deployment immer häufiger, und stets blockierte das Security-Team die Freigabe, da es entweder neue Sicherheitslücken gefunden hatte oder alte Probleme nicht richtig behoben worden waren. Die übrigen

© Rheinwerk Verlag, Bonn 2024

Teams waren daher nie gut auf die Security-Leute zu sprechen, die stets neue Features und dringend benötigte Änderungen am Webshop blockierten.

Im Wasserfall-Modell waren die Schmerzen am größten, aber wie sieht es jetzt in der DevOps-Welt aus?

11.2 DevOps mit getrenntem Security-Team

Nicht selten sieht man, dass zwar die Silos zwischen Entwicklungsteam, Qualitätssicherung und Betriebsteam abgebaut wurden, dass aber die Security weiterhin ein abgeschottetes Dasein am Ende des Value-Streams fristet, ohne Einfluss auf das cross-funktionale Team zu haben.

Es wurden also die ersten Schritte absolviert, um die Lead-Time für Änderungen herunterzuschrauben, aber die Sicherheit ist immer noch ein Nachgedanke, von dem man meint, dass man ihn am Ende über das fertige Feature anbringen könne. Sicherheit wird hier also als ein Extra verstanden, das man nach getaner Arbeit noch anbringt. Es wird also mit dem Häkchen an der Checkliste der Überprüfungen gleichgesetzt, die man aus dem Wasserfall-Konzept kannte.

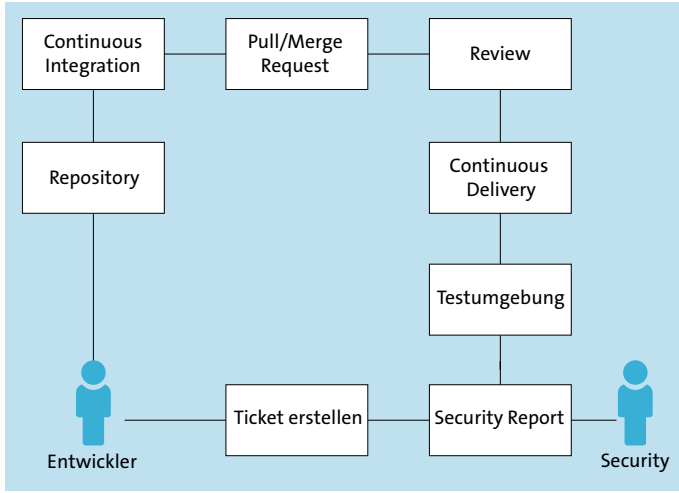

Abbildung 11.3 Der Workflow im traditionellen Konzept: Das Security-Team kommt erst verspätet im Workflow an die Reihe.

Der Workflow sieht dann so aus: Es wird munter entwickelt, und zwar jetzt nach den gängigen DevOps-Prinzipien: Der Code wird in das gemeinsam genutzte Repository committet, die Continuous-Integration-Pipeline läuft nach dem Pull bzw. Merge Request an und nach einem Review wird das Deployment auf eine Test-Umgebung angestoßen (siehe Abbildung 11.3).

Persönliches Exemplar für Thomas Bartholomäus

Da aber das Security-Team immer noch nachgelagert arbeitet, erfolgt das Erstellen des Security-Reports auch weiterhin nur manuell und erst am Ende der Entwicklungsphase. Und hier wird weiterhin blockiert, da die Probleme stets erst gefunden werden, wenn die eigentliche Arbeit schon vorbei ist.

Kein Zugriff auf das Security-Reporting

Ein häufiges Problem ist zudem, dass das Security-Reporting für das Entwicklungsteam nicht einsehbar ist. Das verstärkt die Silos zwischen den Teams, denn wenn die relevanten Personen nicht auf die gleichen Daten zugreifen können, dann ist das nicht nur ziemlich umständlich, sondern es wird dann häufig auch einfach ignoriert. Aus Entwicklungssicht ist das auch kaum verwunderlich: Was ich nicht sehen kann, das kann ich auch nicht korrigieren. Optimal ist so etwas nie.

11.2.1 To deploy or not to deploy?

Bei vielen Hinweisen des Security-Teams war das gar kein so großes Problem: Stellenweise mussten ein paar Codezeilen geändert werden, um etwa ein Feld in einem Formular gegen SQL-Injection abzusichern oder um einen Check zu platzieren. Das ist zwar lästig, aber machbar. Es gab aber regelmäßig üble Bombshells, die grundlegend das Design der Anwendung betrafen oder die verwendeten Abhängigkeiten bemängelten: Solche Probleme hätten schon viel früher im Entwicklungsprozess korrigiert werden müssen. Nun, da bereits ein ganzes Kartenhaus auf diesem Fundament errichtet worden war, ließen sich solche einschneidenden Änderungen nur noch mit größtem Aufwand vornehmen.

Ein einfacher Fall, der immer wieder auftrat, waren veraltete Abhängigkeiten, die nicht nur eklatante Sicherheitslücken aufwiesen, sondern gar nicht mehr gepflegt wurden. Sicherheitsupdates gibt es also nicht. Die einzige Lösung ist, das entsprechende Paket auszutauschen (siehe Abbildung 11.4).

Die Abhängigkeit war allerdings so tief im Code des Online-Shops verankert, dass ein umfangreiches Refactoring erfolgen musste. Kurz vor dem anvisierten Deployment ist das unmöglich, falls man die Deadline einhalten will … und man wollte sich auch nicht kurzfristig noch mehr ungetestete Änderungen am Fundament des Webshops einfangen.

Für das Security-Team und die Führungsetage von *schick-gekleidet.de* hieß es jetzt, abzuwägen: Soll das Release trotz der gravierenden Sicherheitslücken freigegeben werden? Oder soll das Deployment verschoben werden, bis die Lücken korrigiert sind? Hofft man, dass schon nichts passieren wird? Oder legt man noch einmal ganz von vorn mit der Planung und Umsetzung der neuen Features los?

© Rheinwerk Verlag, Bonn 2024

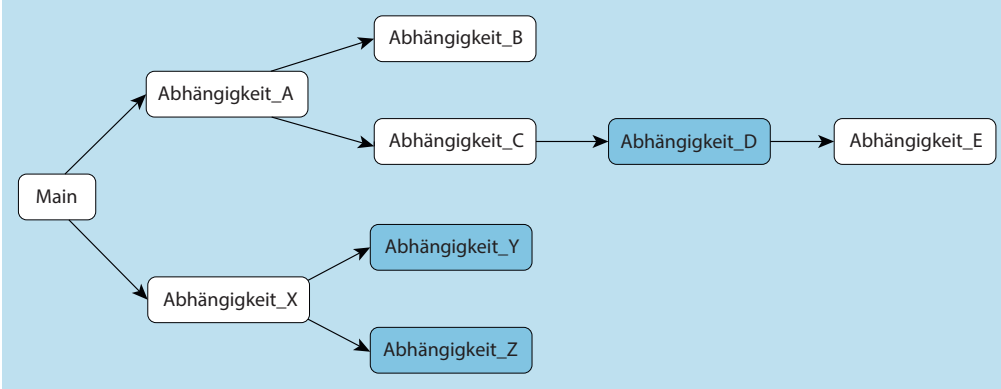

Abbildung 11.4 Dependency-Tree: Die beiden direkten Abhängigkeiten haben keine direkten bekannten Sicherheitslücken. Anders sieht es allerdings aus, wenn man sich den Abhängigkeitsbaum vollständig anschaut.

Beide Optionen bergen Risiken: Security-Lücken könnten ausgenutzt werden, sodass es verdammt teuer werden kann, sie im Shop zu belassen. Aber auch ein Verschieben ist kaum möglich, da das Weihnachtsgeschäft vor der Tür steht und bei den neuen Angeboten etliche Umsatzeinbußen zu erwarten sind, wenn der Shop nicht pünktlich gestartet wird.

11.2.2 Die Suche nach undokumentierten Abhängigkeiten

Im ganzen Trubel wurde daher immer nur das Nötigste geprüft. Ein gescheiter Security-Audit fand viel zu selten statt und wenn, dann dauerte es ewig, bis die Lücken halbherzig geschlossen wurden. Auch hier gab es wenig Konzepte, das strategisch anzugehen.

Aber auch nach einem Deployment im Betrieb war das Security-Team immer wieder gefragt. Das Security-Team hat die gängigen Security-Advisorys abonniert und hat somit im Blick, wenn neue Security-Lücken gemeldet werden, worauf das Team reagieren und die Informationen an die verschiedenen internen Teams weitergeben musste. Einige Lücken schaffen es mittlerweile sogar bis in die Tagesschau, sodass die Geschäftsleitung darauf aufmerksam wird und natürlich wissen will, ob die Lücke auch für *schick-gekleidet.de* relevant ist.

Denken Sie zum Beispiel an die *Log4shell-Lücke* der *log4j*-Bibliothek, die im Dezember 2021 aufkam und es bis in die Abendnachrichten geschafft hat. Wer für IT-Systeme verantwortlich war, fragte sich unweigerlich: »Sind wir davon eigentlich auch betroffen?« (Darauf komme ich in Abschnitt 11.5 noch mal zu sprechen, wenn es um die Supply-Chain-Security geht.)

Persönliches Exemplar für Thomas Bartholomäus

Das Security-Team musste also alle Anwendungen nach dieser Bibliothek absuchen. Das betraf sowohl die selbst entwickelten Anwendungen als auch Fremdapplikationen, die log4j einsetzen. Und wenn log4j gefunden wurde: In welcher Version liegt es vor?

Und wo sollte überhaupt gesucht werden? Da sich im Laufe der Zeit viel Schatten-IT eingeschlichen hatte, mussten sich die Security-Leute händisch durch zahlreiche Repositorys wühlen: Viele Abteilungen hatten ihre eigenen Git-Server, die nicht zentral durchsucht werden konnten; es war eine Heidenarbeit, bis man einen Überblick darüber hatte, welcher Code gerade überhaupt genutzt wird.

Bei dieser Überprüfung fielen dann Anwendungen auf, die uralte Versionen von log4j oder anderen Bibliotheken nutzten. Ein »Cool, wir sind an dieser Stelle gar nicht betroffen, unser Zeug ist *zu* alt!« hilft dann natürlich nicht weiter, denn diese Software wies natürlich auch Lücken auf – wenn auch nicht gerade die, nach denen man suchte.

Fassen wir zusammen: Ein großes Problem für das Security-Team war, dass es keinen aktuellen Einblick in den Quellcode der einzelnen Teams hatte. Die Abschottung der einzelnen Teams machte die Prüfung unglaublich aufwendig, zeitnahe Entscheidungen oder gar Lösungen waren nicht möglich.

Für die Geschäftsführung waren das keine schönen Neuigkeiten, weil diese eigentlich nur wissen wollte, ob die Lücke auch ausgenutzt werden konnte oder nicht. Selbst diese einfache Frage konnte nicht beantwortet werden, und von einem Plan, wie die Systeme zeitnah gepatcht werden konnten, war man weit entfernt.

11.2.3 Frust und Blockade

In den Entwicklerteams waren kaum Security-Kenntnisse vorhanden, da man sich stets darauf verließ, dass die Sicherheitsexperten schon dafür sorgen würden, dass alles sicher ist. Andererseits war der Ärger immer groß, wenn dann teils Monate nach dem Abschluss der Programmierarbeit eine lange Liste mit Mängeln ans Dev-Team überreicht wurde. Und da das Security-Team den Code ja nicht kannte, waren viele Punkte dieser Liste für die Entwickler unverständlich, nicht nachvollziehbar oder schlicht nicht umsetzbar.

Die Entwicklungsteams waren daher nie gut auf das Security-Team zu sprechen und nahmen die Security-Experten als bürokratische Nörgler und Bedenkenträger wahr. Die Security-Leute wiederum fragten sich, warum die Entwickler nicht einfach mal saubere und sichere Software schrieben: »Das kann doch nicht so schwer sein!« Schuld waren also immer die anderen, und das Finger-Pointing ging vor jedem Deployment von vorne los.

© Rheinwerk Verlag, Bonn 2024

Reflexion

Security ist leider ein Thema, das viel zu oft vernachlässigt wird. Entwicklungsteams und Führungskräfte denken sich häufig, dass schon nichts passieren wird. Zusammen mit den strukturellen Fehlern bei *schick-gekleidet.de* sorgte das nicht nur für handfeste Sicherheitsprobleme, sondern auch für viel Krach zwischen den Teams und für ineffiziente Abläufe.

Natürlich sind Security-Lücken schlecht und sollten möglichst zeitnah korrigiert werden. Das sagt nicht nur der gesunde Menschenverstand, sondern auch diverse rechtliche Grundlagen verlangen es.

Technikern in der »alten« Welt kann ich nur empfehlen, ihr Team immer wieder auf Sicherheitsrisiken hinzuweisen. Wichtig ist aber vor allem, dass auch Führungskräfte das Problem erkennen und verstehen, dass sie hier investieren müssen.

11.3 DevSecOps: Sicherheit in DevOps einbauen

DevOps, angereicht mit Security, wird *DevSecOps* genannt. Die Grundidee ist, Sie ahnen es schon, dass die Security Teil des gesamten Software-Delivery-Lifecycles wird.

Bevor wir uns in die Tiefen von DevSecOps stürzen, stellt sich erst einmal noch eine begriffliche Frage: Warum eigentlich Dev**Sec**Ops? Ist DevSecOps eine Teilmenge von DevOps oder ist DevOps eine Teilmenge von DevSecOps?

Auf den ersten Blick mag man denken, dass DevOps eine Teilmenge von DevSecOps ist. Die Begründung dafür ist, dass DevSecOps *mehr* ist als nur DevOps, ergo muss DevOps ja eine Teilmenge sein. Die Realität ist tatsächlich eher genau umgekehrt: DevOps ist als großes Ganzes zu verstehen, und DevSecOps ist eine Spezialisierung. Es ist eben DevOps, angereichert um Security. Es gibt auch gute Gründe, auf den Begriff zu verzichten: Das Qualitätssicherungsteam und das Plattformteam befinden sich auch nicht im Namen DevOps, ansonsten hieße es schnell *DevQASecPlatOps* oder so ähnlich.

DevSecOps vs. DevOps

Wenn von Dev**Sec**Ops statt von DevOps gesprochen wird, dann liegt das meistens daran, dass gerade der Security-Teil hervorgehoben werden soll. Wenn hingegen von DevOps gesprochen wird, wird das Thema eher allgemeiner betrachtet, was auch Security beinhalten sollte.

Was ist nun DevSecOps genau? Es geht darum, das Security-Team frühzeitig in den vollständigen Entwicklungs-Lifecycle einzubinden. Auch hierbei ist es nicht das Ziel,

Persönliches Exemplar für Thomas Bartholomäus

das Security-Team vollständig zu ersetzen und aufzulösen, sondern seine Tätigkeiten und Kenntnisse schon so früh wie möglich in den Entwicklungslebenszyklus einzubinden, damit es natürlicher Bestandteil der Entwicklungsarbeit wird. Die Verantwortung dafür, ein sicheres Produkt zu entwickeln, liegt daher nicht mehr in einem Prozess ganz am Ende des Weges, sondern wird auf alle Beteiligten verteilt. Sicherheit spielt damit von der Planungsphase über die Umsetzung bis zum Deployment immer eine Rolle.

In den Architektur- und Planungsphasen sollte also stets auf das Security-Team gehört werden, und es sollte bei der Umsetzung von neuen Features anwesend sein. Dazu gehört dann selbstverständlich, dass das Security-Team während der Entwicklung frühzeitig und ohne hohe Hürden zurate gezogen werden kann.

Wie so häufig im Rahmen der DevOps-Transformation gilt es dazu im ersten Schritt die Zusammenarbeit zwischen dem Security-Team und den übrigen Teams stark zu verbessern. Der bisherige Eindruck, dass das Security-Team stets blockiert und mehr Arbeit für die einzelnen Teams verursacht, als es hilft, muss aufgehoben werden.

11.3.1 Die DevSecOps-Teamstruktur

Wie so häufig gibt es beim Aufbau des cross-funktionalen Teams nicht nur eine Wahrheit. Abbildung 11.5 zeigt *eine* Möglichkeit: In diesem Beispiel werden die Security-Teammitglieder auf die cross-funktionalen Teams aufgeteilt, jedes neue Projektteam bekommt also seinen eigenen Security-Experten.

Es lohnt sich hier aber, ein wenig genauer hinzuschauen. In Kapitel 7, in dem es um die Qualitätssicherung ging, wurde schon beschrieben, dass es eine *doppelte* Teamstruktur für die Qualitätssicherung geben kann: Einmal ist eine Person mit Fokus auf Qualitätssicherung Teil des cross-funktionalen Teams für die täglichen Arbeiten, zudem existiert aber das reine Qualitätssicherungsteam weiter, das für die methodischen Verbesserungen der Organisation verantwortlich ist.

Das Konzept lässt sich auch in Bezug auf die Sicherheit umsetzen: Das Security-Team bleibt bestehen, um organisationsweit die Leitlinien in Sicherheitsfragen festzulegen, die dann von den einzelnen Produktteams umgesetzt werden sollen. Zudem arbeiten Sicherheitsfachleute in allen Teams am Tagesgeschäft mit und sind von der Planung bis zum Betrieb eingebunden.

Ob dieser Ansatz reibungslos funktioniert, hängt von verschiedenen Faktoren ab. Ein maßgeblicher Einflussfaktor ist die Unternehmensgröße. In kleineren Unternehmen gibt es oft nur wenige Teams und meistens kaum dedizierte Sicherheitsexperten. Ein häufiges Problem in solchen Unternehmen ist die begrenzte Anzahl von Sicherheitsexperten. In größeren Unternehmen sieht dies oft anders aus, da es dort mehr Manpower (oder Womanpower) gibt und leichter Stellen mit Experten besetzt werden

© Rheinwerk Verlag, Bonn 2024

können. Diese dürfen sich dann allerdings meist mit mehr Bürokratie und längeren Entscheidungswegen herumschlagen.

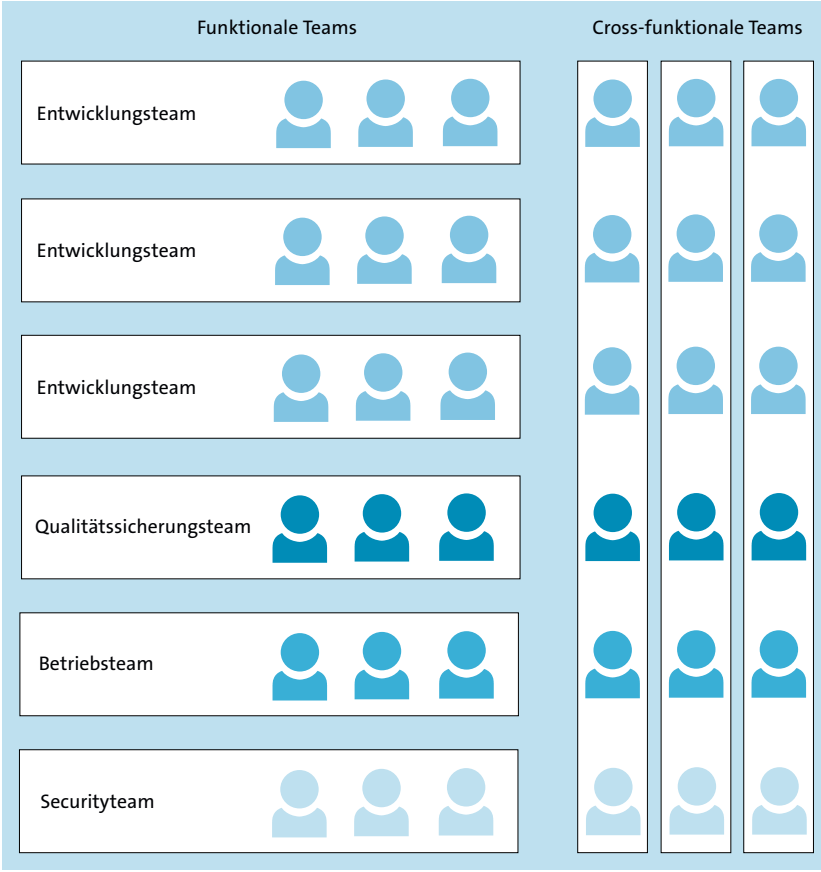

Abbildung 11.5 Das cross-funktionale Team wird um Security-Mitglieder ergänzt.

Ein alternatives Konzept ist, dass ein oder mehrere Teams mit Security-Experten existieren, die für diverse Anwendungen und Produkte zuständig sind. Anstatt also ein voll integriertes Team zu haben, wird hier versucht, die Wände zwischen Security-Team und dem Produktteam niedrig zu halten.

Wichtig ist, dass die Hürden für die Zusammenarbeit nicht nur auf dem Papier niedrig sind, sondern auch effektiv niedrig gehalten werden. Das bedeutet unter anderem, dass der Zugriff auf alle Daten – sei es das Repository, das Projekt-Management oder eben die Security-Dashboards – für alle gleichermaßen gegeben ist.

Die bisher besprochenen Teamstrukturen setzen voraus, dass die für die Sicherheit zuständigen Personen einen starken Schwerpunkt auf Security legen – zu Recht. Ergänzend dazu können DevSecOps-Coaches eine wertvolle Rolle spielen. Ihr Haupt-

Persönliches Exemplar für Thomas Bartholomäus

augenmerk liegt darauf, Teams dabei zu unterstützen, frühzeitig Security-Scans in den Pipelines zu implementieren und effizient mit den entdeckten Sicherheitslücken umzugehen. Diese Lücken sollten automatisch identifiziert und zeitnah geschlossen werden.

11.3.2 Shift-Left: Fehler früher finden

Dieser Ansatz wird *Shift-Left* genannt. Die Prüfungen, die vorher ganz weit rechts, also spät, im Software-Delivery-Lifecycle stattfanden, sollen stattdessen möglichst weit nach links rutschen, also früher in den Prozess integriert werden. Je früher, desto besser!

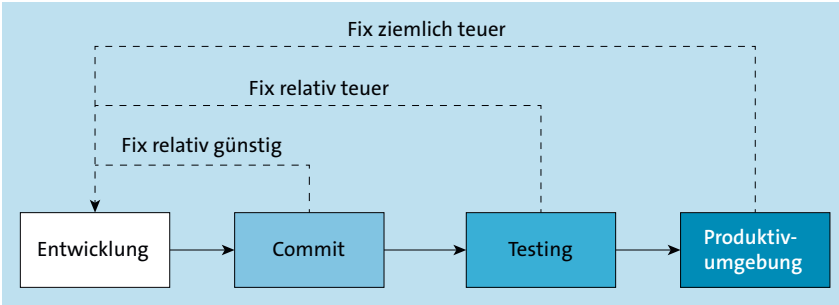

Abbildung 11.6 Shift-Left: Je später ein Fehler korrigiert wird, desto aufwendiger und somit teurer ist der Fix. Bei Security-Vulnerabilitys gilt das umso mehr.

Wenn Security-Lücken in der Produktivumgebung gefunden werden, ist das problematisch: Zum einen kann offensichtlich die Sicherheitslücke ausgenutzt werden, und zum anderen dauert der ganze Prozess, bis die Korrektur eingepflegt, getestet und ausgerollt ist, ein wenig. Je mehr Lücken man in der Produktivumgebung findet, desto aufwendiger und somit teurer ist der Fix.

Besser ist es, wenn die Lücke schon früher gefunden wird. Wenn sie in einer Test- oder Staging-Umgebung gefunden wird, dann ist das schon mal ein wenig besser. Aber auch hier ist solch ein Fix in dem Sinne »teuer«, da auch hier der Kontext der Entwicklung verloren gegangen ist und alle Änderungen erst wieder durch den Review-Prozess müssen.

Idealerweise wird eine Sicherheitslücke also spätestens beim Commit gefunden, also dann, wenn die Pipeline läuft, bevor ein Review stattfindet. Das Tooling sollte den Prozess dann so unterstützen, dass bei jeder Änderung, die man beitragen möchte, die Auswirkungen im Security-Kontext aufgezeigt werden. Wenn keine neuen Lücken hinzugefügt werden, dann ist alles gut. Wenn doch, dann muss hier gegengesteuert werden.

© Rheinwerk Verlag, Bonn 2024

Aber es geht auch noch einen Schritt früher: Einige Security-Checks lassen sich bereits in der Entwicklungsumgebung des Entwicklers ausführen. So gibt es idealerweise ein Feedback bereits dann, wenn die Zeilen des Quellcodes getippt werden.

11.3.3 Inner Sourcing sorgt für formelle Sicherheit

Die in Abschnitt 5.5 beschriebene Öffnung der Repositorys und die gemeinsame Projektplanung für alle Teams innerhalb der Organisation helfen auch dem Security-Team. Dieses kann frühzeitig in die transparente Planung eingreifen und Feedback geben. Für die Entwickler wird die Arbeit einfacher, denn sie können schnell das Security-Team anschreiben und Einschätzungen der Experten einholen.

Voraussetzung hierfür ist natürlich, dass das ganze Team mit offenen Karten spielt und die Planung transparent verläuft. Je mehr Probleme und Herausforderungen asynchron in Tickets oder in einer Knowledge-Database dokumentiert werden, desto einfacher ist es möglich, Hilfestellungen zu geben.

Bleiben wir beim Beispiel einer Abhängigkeit, die bereits vom Upstream-Projekt abgekündigt worden ist und nicht mehr gewartet wird. Durch die Öffnung und die Sichtbarkeit innerhalb der Firma können die Security-Teammitglieder, die nun Teil der bisher reinen Entwicklungsteams geworden sind, direkt erkennen, was passiert. Sie können entweder direkt ein Veto einlegen oder zumindest im Team diskutieren, ob eine veraltete Version an dieser Stelle des Webshops genutzt werden darf.

11.3.4 Security als fester Bestandteil des Entwicklungsprozesses

Wenn die DevOps-Prinzipien gut umgesetzt werden und eine kurze Lead-Time für Änderungen existiert, dann können und sollten Security-Fixes schnell ausgerollt werden, um mögliche Risiken zu vermeiden. Abbildung 11.7 fasst zusammen, wie das aussehen sollte.

Der Workflow sieht schon deutlich besser aus: Es gibt nach dem Lauf der CI-Pipeline schon einen Security-Report, der für die aktuelle Entwicklung genutzt wird, bevor ein Deployment auf die Produktivumgebung angestoßen wird. Das bringt zwei offensichtliche Vorteile mit sich:

- **Für die Entwickler**, die für die Umsetzung zuständig sind, hat es den großen Vorteil, dass sie schon früh Feedback bekommen. Schon zu Beginn der Arbeit wird deutlich, dass es eine schlechte Idee wäre, das Deployment wie geplant umzusetzen.

  Das mag jetzt etwas negativ klingen – so, als ob man als Entwickler die Sicherheitslücken bewusst einbauen würde. Tatsächlich passieren aber die meisten Fehler unbeabsichtigt, meistens aus Unwissenheit. Je früher man darauf hingewiesen wird, desto einfacher kann unnötige Arbeit vermieden werden. Bei der richtigen

Zusammenarbeit profitieren also alle davon, dass frühzeitig interveniert wird statt erst kurz vor dem Deployment.

- **Das Security-Team** kann Aufgaben automatisieren und deutlich agiler agieren. In einer Arbeitswelt voller Silos waren die Sicherheitsleute ständig überlastet, weil die Prüfung eines Deployments einem einzigen, riesigen Big-Bang-Release gleichkam. Zwangsläufig musste man sich auf das Wichtigste konzentrieren, und dabei blieben viele Probleme unbehandelt liegen.

  Durch den frühen Einsatz von Security-Scannern im Entwicklungsprozess wird die Arbeit nun jedoch in viele kleine Schritte aufgeteilt, sodass die Prüfung des Releases wesentlich schneller und einfacher geschieht. Wenn alles optimal umgesetzt wird, verringert sich die Last des Security-Teams: Die einfachen Sachen, die sich automatisieren lassen, blockieren nicht mehr seine Arbeit, sondern werden vom Entwicklungsteam mithilfe der Scanner gelöst. Das ist natürlich für ein chronisch unterbesetztes Team sehr hilfreich.

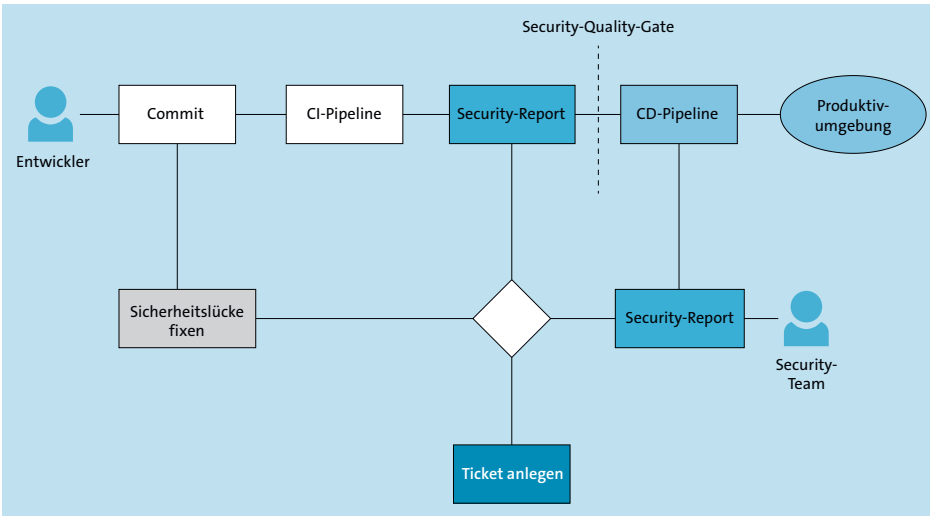

Abbildung 11.7 DevSecOps: Frühes Feedback hilft allen Teams.

Zu den einfacheren Workflows gehört auch, dass Handarbeit wie das Prüfen auf veraltete Abhängigkeiten entfällt. Das Subscriben und Verteilen von Security-Advisorys kann stattdessen automatisiert über den Security-Scanner erfolgen. Wenn neue Lücken in den Abhängigkeiten gefunden werden, sollte also direkt beim entsprechenden Team ein Ticket anlegt und gegebenenfalls priorisiert werden, damit der Fehler möglichst zeitnah korrigiert wird.

Das Security-Team kann sich daher stärker um die Produktivsysteme kümmern, auf denen ebenfalls noch separate Scanner laufen, die die Prod-Umgebung regelmäßig prüfen.

© Rheinwerk Verlag, Bonn 2024

11.3.5 Mit Fehlern umgehen

Wenn Sie der Sicherheit in Ihrem Workflow einen höheren Stellenwert zuweisen und sie bereits früh in die Entwicklungsarbeit integrieren, werden zwangsläufig viele Fehler gefunden. Natürlich mag niemand Fehler und Sicherheitsprobleme, aber es ist gut und wichtig, dass sie gefunden werden. Es wirft jedoch das Problem auf, wie effektiv mit ihnen umgegangen wird.

Quality-Gates

Fangen wir vorn an: Was tun mit Security-Vulnerabilitys, die auftauchen, bevor diese in den Produktionszweig gemergt werden? Das ist der Idealfall, denn ganz zu Beginn lassen sich Fehler ja am einfachsten korrigieren und beheben. Dazu gibt es *Quality-Gates*, also festgelegte Kontrollpunkte oder Kriterien innerhalb des Entwicklungsprozesses, die sicherstellen sollen, dass der Code bestimmte Sicherheitsanforderungen erfüllt, bevor er weiter in den nächsten Schritt des Entwicklungsprozesses übergeht. Wenn der Code ein Security-Gate nicht besteht – zum Beispiel durch das Aufdecken einer Schwachstelle bei einem automatisierten Sicherheitsscan –, dann wird seine Fortführung im Entwicklungsprozess gestoppt, bis das identifizierte Problem behoben wurde.

Das hört sich in der Theorie natürlich sinnvoll an, in der Praxis lauten die Fragen aber:

- Wie sollen die Kriterien aussehen, die den Entwicklungsprozess stoppen?
- Wer legt fest, was ein richtiger *Showstopper* ist?
- Wie konfiguriert man das Quality-Gate?
- Welches Level von Vulnerabilitys stoppt das Deployment?
- Wer darf Änderungen genehmigen, auch wenn diese zu neuen Vulnerabilitys führen?

Auf diese Fragen lassen sich kaum allgemeingültige Antworten geben – die perfekte Lösung hängt natürlich von Ihren individuellen Anforderungen ab. Sie wird für die Sicherheit eines Atomkraftwerks offensichtlich ganz anders aussehen als für ein kleines Online-Spiel. Für den Anfang ist es überhaupt erst mal wichtig, dass die Vulnerabilitys durch die Quality-Gates überhaupt sichtbar werden. Idealerweise erkennen die zuständigen Personen nach dem Erstellen des Merge bzw. Pull Requests direkt, dass bereits während der Entwicklungsarbeit Änderungen nötig sind.

> **Compliance-Policys**
>
> Compliance-Policys lassen sich sowohl in GitLab als auch in GitHub anlegen. Grundsätzlich geht es also darum, dass Policys definiert werden, die durchsetzen, dass Quality-Gates durchlaufen werden. Die typischen Compliance-Funktionalitäten sind in der kostenpflichtigen Enterprise-Variante enthalten.

Im zweiten Schritt muss ein Überblick über die vorhandenen Vulnerabilitys geschaffen werden. Hierfür sind also die Dashboards nötig und hilfreich. Gute Scanner liefern zudem nützliche Informationen zu der gefundenen Vulnerability und dazu, wie diese korrigiert werden kann.

Wenn dieser Überblick besteht, folgen Änderungen, die die gefundenen Vulnerabilitys reduzieren. Es ist die Aufgabe des Entwicklerteams, die Schwachstellen systematisch zu beheben und alle gefundenen Probleme zu bearbeiten.

Das Durchsetzen von Approval-Rules bei der Einführung neuer Vulnerabilitys sollte dann in einem dritten Schritt erfolgen. Das bedeutet, dass Probleme nur unter bestimmten Voraussetzungen akzeptiert werden – dass also Ausnahmen definiert werden, unter denen doch einmal eine bekannte Sicherheitslücke oder eine veraltete Abhängigkeit deployt werden darf. Gründe für eine solche Genehmigung können zum einen darin bestehen, dass das Beheben des Problems sehr aufwendig ist und dadurch wahrscheinlich neue Probleme entstehen. In diesem Fall muss geklärt werden, ob das Risiko tragbar ist. Durch einen transparenten Prozess sollte zudem festgehalten werden, wer für die Genehmigung verantwortlich ist und mit welchen Informationen die Entscheidung getroffen wurde.

Offensichtlich sollte diese Entscheidung niemals allein vom Entwickler getroffen werden, der für den Code verantwortlich ist. Stattdessen muss das Security-Team in diese Entscheidung eingebunden sein, damit es zusammen mit den Entwicklern abwägen kann, wie weiter vorgegangen wird. Zumindest muss die Möglichkeit bestehen, dass die Security-Experten intervenieren können, bevor die Änderungen gemergt werden, und nicht erst, bevor das ganze Produkt deployt werden soll.

Neben dieser schwerwiegenden Entscheidung kommt es in der Praxis jedoch viel öfter vor, dass *False Positives* durchgewunken werden müssen. Das bedeutet, dass der Scanner einen Alarm ausgibt oder eine Schwachstelle markiert, obwohl in Wirklichkeit kein Sicherheitsrisiko besteht. False Positives können für Entwicklungs- und Sicherheitsteams problematisch sein, da sie zusätzliche Ressourcen für die Überprüfung und Behebung vermeintlicher Sicherheitsprobleme erfordern, die in Wahrheit nicht existieren.

False Positives

False Positives wird es immer geben. Erst recht viele False Positives wird man haben, wenn man in einem Projekt von jetzt auf gleich alle Scans anwirft. Das *Tuning* der Scanner ist daher wichtig; gehen Sie am besten Schritt für Schritt vor, wenn Sie die Scanner in Betrieb nehmen, und prüfen Sie, ob die Resultate sinnvoll sind. Etwaige False Positives müssen ermittelt und ausgeräumt werden.

Noch etwas tückischer als echte False Positives ist der Fall, dass zwar eine Security-Lücke gemeldet wird, diese aber für Ihren Use-Case gar nicht relevant ist, da sie nur

© Rheinwerk Verlag, Bonn 2024

unter bestimmten Randbedingungen gefährlich ist. Mit einem solchen Problem kann man leben, allerdings weiß niemand, was die Zukunft bringt – vielleicht wird genau dieses Problem doch irgendwann einmal relevant. Hier hilft es nur, den Fall möglichst transparent zu dokumentieren, damit Ihre zukünftigen Kollegen sofort wissen, was zu tun ist, falls doch einmal Handlungsbedarf besteht.

11.4 Werkzeuge für mehr Sicherheit

Bisher haben wir nur allgemein von Security-Scannern, Dashboards und Co. gesprochen, ohne uns genauer anzuschauen, wie sie arbeiten. Wie bereits erwähnt, sollen die Security-Scanner in der CI/CD-Pipeline ausgeführt werden, in der der Code frühzeitig und automatisiert geprüft wird. Die Grundidee ist, dass auf diese Weise das Feedback natürlicher Bestandteil des Entwicklungsprozesses wird.

Dahinter steht also der Gedanke, dass ein vollständiger Scan mit den Tools automatisiert erfolgen kann. Dadurch sollte ein Gesamtbild entstehen, das über Einzelanalysen bestimmter Produkte oder des Workflows hinausgeht, denn erfahrungsgemäß ist das Zusammentragen dieser Erkenntnisse so aufwendig und fehleranfällig, dass nur der Gesamtüberblick sinnvoll ist.

Wie immer gilt: Die besten Tools nützen nichts, wenn ihre Anwender sie nicht richtig einsetzen können. Und da niemand mehr die Komplexität einer gewachsenen Software komplett überschauen kann, sollten die verschiedenen Tools genau diese Aufgabe übernehmen und übersichtlich Sicherheitsprobleme darstellen und zusammenfassen.

Wichtig im Rahmen der Implementierung von DevSecOps ist daher, dass die Security-Tools möglichst einfach zu nutzen und einzubinden sein sollten. Dabei ist das Security-Team in der Pflicht, die richtigen Tools mit den richtigen Daten und aussagekräftigen Dashboards auszuwählen. Sie sind der Weg, über den die Ergebnisse der Scanner genutzt werden.

11.4.1 Dashboards und Reporting

Security-Dashboards und -Reports sind die einfachste Möglichkeit, wie die Erkenntnisse der Security-Scans mit allen Teams geteilt werden können. In ihnen werden Abhängigkeiten visualisiert und bekannte Sicherheitsprobleme markiert. Idealerweise können Sie so auf einen Blick sehen, ob eine Lücke nur einen kleinen, unbedeutenden Teil Ihrer Anwendung betrifft oder ob gleich der ganze Monitor rot leuchtet.

Wichtig für die erfolgreiche Umsetzung von DevSecOps ist, dass alle Personen auf die gleichen Dashboards schauen können. Ein häufiges Problem, das zu Silos führt, besteht darin, dass nur wenige Personen aus dem Security-Team einen Blick auf die Dashboards haben. Das ist zum einen ein technisches Problem, aber auch ein strukturelles Problem.

Persönliches Exemplar für Thomas Bartholomäus

Wenn nicht alle relevanten Personen – sowohl Entwickler als auch Security und auch Führungskräfte – auf die gleichen Dashboards mit den gefundenen Sicherheitslücken schauen können, dann birgt das erneut das Risiko, dass Sicherheitslücken nicht gefixt oder wegdiskutiert werden, eben weil man nicht vom gleichen Stand aus argumentiert. Es kommt erneut zu Konflikten, weil nicht alle alles sehen und darauf basierend Entscheidungen treffen können.

Dadurch, dass das Security-Team die gleichen Dashboards wie das DevOps-Team sieht, können die Teams anhand der Sicherheitslücken, die dort gelistet sind, verschiedene Aktionen durchführen.

11.4.2 Pull und Merge Requests

Wenn Sicherheitsprobleme identifiziert werden, gibt es grundsätzlich zwei Reaktionen darauf: Bei kleineren Problemen kann es ausreichend sein und für schnelle Lösungen sorgen, wenn das Security-Team Fixes direkt in den Quellcode implementiert, indem es die nötige Änderung über Pull bzw. Merge Request beiträgt, die dann den Weg durch die normalen Reviews gehen – was natürlich nur funktioniert, wenn es einen einfachen Zugriff auf die Repositorys gibt.

Bei größeren Problemen, die eventuell sogar die Architektur der Anwendung betreffen, sollte ein Ticket im System geöffnet werden, damit die Lücken, die von den Scannern gefunden wurden, in die Planung für den nächsten Sprint aufgenommen werden. Aber auch dann, wenn nur kleinere Änderungen notwendig sind, muss das Feedback über Pull bzw. Merge Requests ersichtlich sein. Selbst ohne Review von einer anderen Person erfahren die Entwickler so direkt, was in ihrem Code geändert werden muss.

Die Intention ist erneut, dass viele Änderungen möglichst früh und zeitnah eingebunden werden. Wenn etwa neue Abhängigkeiten eingebunden werden, sollten diese sofort durch die Security-Tools gescannt werden. Wenn zu lange gewartet und zu viel entwickelt wird, ohne dass die Security-Checks durchlaufen, dann hat man immer noch nichts gewonnen.

Also gilt deswegen auch hier: möglichst viele kleine Änderungen beitragen, sodass Feedback früh eintrudelt. Das ist ganz unabhängig davon, ob das Review automatisiert durch einen Scanner passiert oder durch einen Menschen.

11.4.3 Die Security-Scanner im Detail

Bisher haben wir vor allem betrachtet, wie der Workflow von DevSecOps aussieht, welche Probleme damit behoben werden sollen und wann das Ganze stattfinden soll. Dass automatisierte Security-Scanner dabei sehr wichtig sind, dürfte offensichtlich

© Rheinwerk Verlag, Bonn 2024

gewesen sein. Sie sind Teil der CI/CD-Pipeline und führen dort Security-Checks aus. Grob unterteilen lassen sie sich in folgende Aufgabenbereiche:

- Static Application Security Testing (SAST)
- Dependency-Scanning
- Container-Image-Scanning
- Dynamic Application Security Testing (DAST)
- Interactive Application Security Testing (IAST)
- Secret-Detection
- Fuzz-Testing

Je nach Security-Scanner wird auf einer anderen Ebene nach Sicherheitslücken gesucht. Im Grunde gibt es drei verschiedene Ebenen: den Sourcecode, das Binary sowie den Auslieferungsmechanismus. Abbildung 11.8 zeigt, worauf sich die meisten der genannten Scanner konzentrieren.

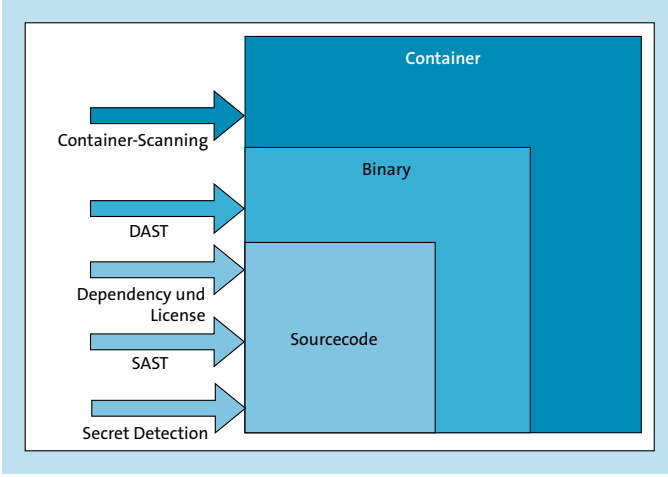

Abbildung 11.8 Security auf allen Leveln: Übliche Security-Scanner scannen auf verschiedenen Ebenen nach Sicherheitslücken.

Alle Security-Tools finden bekannte Security-Vulnerabilitys. Nicht umsonst steht hinter jedem Tool auch eine kommerzielle Firma mit einem kommerziellen Produkt, da die Verwaltung und die Aktualisierung der Vulnerability-Datenbanken einiges an Arbeit benötigen. Als Vorteil bringen viele der kommerziellen Angebote zudem eigene Dashboards und Reports mit. Das ist zwar grundsätzlich löblich, allerdings bringt es nichts, wenn man zwar die Security-Scans laufen lässt, aber niemand die Resultate anschaut.

Wichtig ist daher, dass die Integration für die Teams einfach gehalten wird und dass die Hürde, in die Vulnerabilitys hineinzuschauen, so niedrig wie möglich gehalten

wird. Das Gegenteil ist ein Fehler, der häufig dann passiert, wenn zu viele verschiedene Security-Tools im Einsatz sind und jedes Tool sein eigenes Dashboard mitbringt. Einem vermeintlich faulen Teammitglied, das nur seine Arbeit machen will, macht das ständige Umschalten nicht viel Spaß, und es fängt an, die verschiedenen Dashboards zu ignorieren.

Für die entdeckten Vulnerabilitys sollte es eine zentrale Stelle geben. Idealerweise nutzen Sie direkt die eingebauten Features in GitLab oder GitHub, wo beides in den kommerziellen Angeboten verpackt ist. Das ist die einfachste Möglichkeit, da auf diese Weise die Visibilität und die Integration in den Entwicklungsworkflow am besten gegeben sind. Alternativ gibt es noch diverse andere Tools für das Vulnerability-Management; dann müssen die gefundenen Vulnerabilitys aber auch dahin gemeldet werden. Eine Möglichkeit ist etwa die Nutzung von *DefectDojo* (*https://github.com/DefectDojo/django-DefectDojo*).

Dependency-Scanning

In diesem Kapitel war das Dependency-Scanning unser Parade-Beispiel. Viel zu oft gibt es etliche Abhängigkeiten, die veraltet sind: entweder weil sie bereits »alt« eingebaut wurden oder weil der Zahn der Zeit an ihnen nagt. Beim Dependency-Scanning wird nicht viel mehr gemacht, als die Abhängigkeiten, die eingebunden sind, zu durchsuchen und den ganzen Dependency-Baum aufzubauen (siehe Abbildung 11.4).

Denn nicht nur die direkt eingebundenen Dependencys können Lücken aufweisen, sondern auch die Dependencys von den Dependencys von den Dependencys. *Transitive Abhängigkeitsauflösung* lautet der Fachbegriff dafür. Der Scanner sucht dabei den gesamten Depencency-Baum ab und prüft in der Vulnerability-Datenbank, ob und welche Lücken bekannt sind.

Eine wesentliche Voraussetzung ist die Nutzung eines gängigen, programmiersprachenunabhängigen Paketmanagementsystems, um dieses Verfahren effizient anwenden zu können. Benutzerfreundliche Dependency-Scanner sollten automatisch erkennen, welche Programmiersprache und welches Paketmanagement-Tool verwendet wird. Dies erleichtert das Aufbauen des Abhängigkeitsbaums, das Nachschlagen von Sicherheitslücken und die anschließende Darstellung im Sicherheitsbericht.

Das Entwicklungsteam muss dafür sorgen, dass standardmäßig ein Paketmanager wie Maven in Java genutzt wird und dass der Scanner in die Pipeline integriert wird, um die Sicherheitslücken in den Abhängigkeiten zu identifizieren.

Einige Dependency-Scanner gehen jedoch noch weiter und bieten die Möglichkeit, Abhängigkeiten automatisch zu aktualisieren. Hier könnte entweder automatisch ein Commit erstellt werden, der die Abhängigkeit aktualisiert, oder es wird ein Pull bzw. Merge Request generiert. Dies minimiert den manuellen Aufwand, sodass nur noch der Merge durchgeführt werden muss, um die Aktualisierung abzuschließen.

© Rheinwerk Verlag, Bonn 2024

Die Aktualisierung der Abhängigkeiten aus der Vollautomation sieht zwar auf den ersten Blick sehr hilfreich aus, sollte aber unter Umständen vermieden werden. Damit meine ich das automatische Mergen der Aktualisierung der Abhängigkeit. Denn das kann zu unschönen Nebeneffekten führen, da durch die Abhängigkeitsaktualisierung Inkompatibilitäten auftreten können. Das automatische Erstellen eines Pull bzw. Merge Requests ist dagegen sehr sinnvoll: So kann noch mal separat reviewt werden, ob alles passt und ob es nicht vielleicht doch zu Inkompatibilitäten kommt und etwas per Hand gefixt werden muss.

Tools, die Sie dazu evaluieren sollten, sind *Dependabot* aus dem Hause GitHub (*https://github.com/dependabot*) sowie *Renovate Bot* von Mend (*https://github.com/renovatebot/renovate*). Während Dependabot hervorragend mit GitHub funktioniert, lässt sich Renovate auch auf anderen Plattformen gut verbinden und einsetzen. Praktisch ist dabei, dass es in die Pipeline eingebunden werden kann, sodass Sicherheitslücken frühzeitig auffallen. Durch die verfügbare Automatisierung können beide Tools bei Bedarf Pull bzw. Merge Requests erzeugen, sodass diese dann angenommen werden können.

Container-Scanning

Wenn Sie für ihre Deployments Container nutzen, sollten auch diese gescannt werden. Häufig ist dann von »Container-Scanning« die Rede, gemeint ist aber eher »Container-Image-Scanning«, da nicht der laufende Container gescannt wird, sondern nur in das Abbild, also das Image, hineingeschaut wird, um nach potenziellen Lücken zu suchen.

Zudem gibt es gleich mehrere Arten von Scanning-Mechanismen, die sich um Container drehen. Das ist zum einen das *Container-Image-Scanning in der Pipeline* und zum anderen die *Container-Runtime-Security*, wenn der Container in der Ausführungsumgebung läuft.

Container-Image-Scanning

Die Grundidee von Containern ist ja bekanntlich, dass sie auf Images beruhen, in denen nicht nur die Anwendung enthalten ist, sondern auch die benötigten Abhängigkeiten. Dazu zähle ich an dieser Stelle auch das Basis-Betriebssystem.

Beim reinen Image-Scanning wird das Container-Image auf mögliche Lücken im darunter liegenden Basis-Betriebssystem gescannt. Es ploppen in dem Fall also häufig Sicherheitslücken auf, die beispielsweise vom darunter liegenden Ubuntu-Image stammen. In diesem Fall muss das Basis-Betriebssystem aktualisiert und ein neues Image gebaut werden.

In der Praxis dürfte die Priorität dieser Vulnerabilitys allerdings hinter das SAST und Dependency-Scanning fallen, da Sie keinen Einfluss darauf haben, welche Images für Ihre Container genutzt werden. Wenn Sie das Betriebssystem und die grundlegenden

Persönliches Exemplar für Thomas Bartholomäus

Abhängigkeiten nicht aktualisieren können oder das Image nicht wechseln können, dann müssen Sie wohl oder übel mit diesen Meldungen leben.

Mehr als nur das Scanning des Images

Einige Container-Scanner scannen nicht ausschließlich das zugrunde liegende Betriebssystem, sondern führen auch ein Dependency-Scanning und/oder SAST durch. Hier kann es also vorkommen, dass sich einige Funktionen von verschiedenen Scannern überlappen.

Das Scannen des Container-Images sollte immer und regelmäßig in der Pipeline geschehen, sodass man etwaige Fehler früh und regelmäßig findet. Einige Container-Scanner beschränken sich hingegen auf das Scannen der Container-Registry, in der die Container-Images gespeichert sind. Wenn man das nicht richtig umsetzt, dann können Lücken übersehen werden.

Container-Runtime-Security

Neben der Überprüfung der Images muss man sich auch um die *Runtime-Security* kümmern. In diesem Fall geht es darum, dass der Container in einer Test-Umgebung läuft und entsprechend gescannt wird. Es gibt gleich mehrere Angriffsvektoren, die mit dieser Prüfung abgedeckt werden sollen.

Üblich ist, dass man immer wissen möchte, ob es bekannte Sicherheitslücken in den laufenden Containern gibt. Das Image-Scanning fand schließlich nur in der Entwicklungsphase statt, aber eigentlich kommt es ja darauf an, wie sich die Anwendung in der laufenden Umgebung verhält.

Streng genommen ist das auch noch ein reines Image-Scanning. Als zusätzlicher Sicherheitsmechanismus ist angedacht, dass Sicherheitslücken erkennbar sein sollten, wenn böswilliger Code von außen in den Container injiziert wird, um das Verhalten anzupassen oder Daten abzugreifen. Nah verwandt ist dazu auch die *Drift Detection*, bei der überprüft wird, ob es während der Laufzeit Änderungen am Container gab.

Für das Scannen von Container-Images gibt es eine große Anzahl verschiedener Open-Source-Tools, aber auch etliche kommerzielle Tools. Ein freies Tool ist *Trivy* von der Firma Aqua Security (*https://github.com/aquasecurity/trivy*). Wie auch bei den übrigen genannten Security-Tools gibt es auch für Trivy ein kommerzielles Angebot mit mehr Features. Eine Alternative ist *Grype* von Anchore (*https://github.com/anchore/grype*).

Außerdem lohnt sich ein Blick auf das Tool *kics*, das zu Checkmarx gehört (*https://kics.io/*). Es kann neben den Konfigurationsdateien von Terraform, Kubernetes und Ansible auch Dockerfiles lesen und Security-Hinweise geben.

© Rheinwerk Verlag, Bonn 2024

Static Application Security Testing (SAST)

Beim SAST handelt es sich um die statische Analyse von Sourcecode mit dem Fokus auf dem Finden von potenziellen Vulnerabilitys. Der Sourcecode wird hierzu einmal statisch gescannt. Es erfolgt also in der Regel keine Ausführung des Codes. Je nach Programmiersprache gibt es einige Lücken, die sich schnell und einfach über solche statischen Checks finden lassen. Vor allem Probleme mit der Speicherverwaltung, wie sie bei C vorkommen, lassen sich so frühzeitig aufdecken. Auch SQL-Injections und ähnliche Probleme werden zumeist schnell und automatisch erkannt.

Hier geht es also grundsätzlich um die Frage, ob der geschriebene Code sicher ist. SAST-Scanner sind daher stark abhängig von der verwendeten Programmiersprache. Einige Scanner konzentrieren sich auf einzelne Programmiersprachen, einige auf eine breite Auswahl. Sie sollten in jeder Pipeline ausgeführt werden, um früh zu erkennen, ob neue Sicherheitslücken eingeführt werden.

Bei SAST handelt es sich um White-Box-Tests, denn es wird direkt in den Code hineingeschaut. So können SAST-Scans theoretisch auch schon ausgeführt werden, wenn das Projekt sich nicht bauen lässt. Nicht nur in der Pipeline ist das sinnvoll, denn Sie können grundsätzlich auch innerhalb der Entwicklungsumgebung SAST-Scans laufen lassen, um eben sehr früh in der Implementierung etwaige Sicherheitslücken aufzuspüren.

SAST hat allerdings auch einige kleinere Nachteile. So findet keine direkte Validierung statt, ob die Sicherheitslücke auch wirklich ausnutzbar ist. Intelligente SAST-Scanner könnten zwar erkennen, dass eine potenzielle Sicherheitslücke zwar vorhanden ist, diese aber nicht aufgerufen wird und daher unkritisch ist. Trotzdem wäre es hier auch sinnvoll, den Fehler zu sehen, um zukünftige Aufrufe sicher zu handhaben.

Weiterhin muss eigentlich unterschieden werden, ob der Code tatsächlich produktiv genutzt wird oder ob es sich nicht nur um Testcode handelt, der für die echte Sicherheitseinschätzung irrelevant ist.

Ein freier SAST-Scanner, der viele Programmiersprachen unterstützt, ist *Semgrep* (*https://semgrep.dev/*). Semgrep steht in der Basis-Ausstattung unter einer Open-Source-Lizenz, mehr Features sind in der kommerziellen Ausgabe enthalten. Das Praktische an Semgrep ist, dass es relativ schlank gehalten ist, wie gesagt viele Programmiersprachen unterstützt und dabei auch schnell ist. Es wird auch als Standard für viele Programmiersprachen in den GitLab-Security-Scannern verwendet.

Eine häufig verwendete Alternative ist *SonarQube*. Diese wurde bereits in Kapitel 7 erwähnt. SonarQube ermöglicht allerdings auch das Scannen nach Vulnerabilitys. Auch hier gibt es wieder Unterschiede zwischen der freien und der kommerziellen Variante.

Dynamic Application Security Testing (DAST)

Während es bei SAST um die statische Ausführung ging, geht es bei DAST um die dynamische Ausführung. Es handelt sich um einen Blackbox-Test, da Ihre Anwendung ausgeführt werden muss, um das DAST-Tool gegen sie laufen zu lassen, damit es tatsächlich ausführbare Sicherheitslücken finden kann. Im Gegensatz zu SAST ist DAST programmiersprachenunabhängig, denn schließlich wird der Code bei DAST nicht direkt angeschaut, sondern der Zugriff erfolgt über die Schnittstellen.

Bei DAST sieht die Ausführung im Vergleich zu SAST in der Pipeline etwas anders aus. Hier wird die Anwendung auf einer Review-Umgebung deployt, anschließend werden dynamische Security-Tests direkt gegen die laufende Applikation ausgeführt. Während bei SAST etwa in der Theorie geschaut wird, ob SQL-Injections möglich sein könnten, wird es hier mit dem entsprechenden Tooling direkt ausprobiert.

Je nach verwendetem DAST-Tool ist die Lernkurve relativ hoch, denn Sie müssen das Tool umfangreich konfigurieren, damit es gegen Ihre eigene Anwendung laufen kann. Ein häufiger Anwendungsfall ist etwa, dass zunächst ein Login in die Anwendung über das DAST-Tool erfolgen muss, damit danach diversen Eingabefehlern auf Sicherheitslücken geprüft werden kann.

Eine ordentliche und vollständige Ausführung von DAST kann etliche Stunden dauern. Diese Tests sind meistens nicht dafür geeignet, in einer CI/CD-Pipeline zu laufen – weder im Main-Development-Branch noch in den Feature-Branches. Der Run eines DAST-Tools sollte zwar automatisch und regelmäßig erfolgen, allerdings nicht als Teil der gängigen Entwicklungspipeline.

Es kann sich allerdings trotzdem anbieten, einen schnellen DAST-Scanning-Durchlauf in jeder Pipeline auszuführen. Dies muss dann auf einfache und schnell auffindbare Probleme reduziert werden etwa Probleme mit HTTP-Security-Headern.

Ein Tool für das DAST-Scanning ist das OWASP *Zed Attack Proxy* (kurz: ZAP, *https://www.zaproxy.org/*).

Interactive Application Security Testing (IAST)

Ein Mix aus SAST und DAST ist das *Interactive Application Security Testing*. Wie auch bei DAST muss für IAST die Anwendung auf ein Zielsystem deployt werden, damit die Software untersucht werden kann.

Der Nachteil bei DAST ist, dass die Verknüpfung zwischen der laufenden Anwendung und dem Sourcecode fehlt. Der Nachteil von SAST ist, dass nur die Theorie untersucht wird und nicht, ob ein Exploit tatsächlich funktionieren würde.

Der Ansatz bei IAST ist, dass man diese beiden Nachteile nicht mehr hat. Um bei den vorangegangenen Beispielen mit der SQL-Injection zu bleiben: Während bei SAST nur aufgezeigt werden würde, dass an einer bestimmten Zeile in einer Datei eine

© Rheinwerk Verlag, Bonn 2024

SQL-Injection möglich wäre, würde bei DAST entsprechend gemeldet, um welches Formularfeld auf welcher Seite es geht. IAST verbindet diese beide Informationen und informiert Sie darüber, dass die relevante Zeile in einer Datei auf der ermittelten Seite zu der möglichen SQL-Injection führen kann.

Für die Nutzung von IAST wird eine gewisse Instrumentierung im Code benötigt, damit verifiziert werden kann, dass die Lücken, die gefunden werden, mit dem Code verknüpft werden können. Eine weitere Eigenschaft von IAST ist, dass die Anzahl der False Positives deutlich geringer ausfällt, als es bei SAST oder DAST der Fall ist, da mit realistischeren Szenarien gearbeitet wird.

In der Praxis sieht es also so aus, dass für die Ausführung von IAST die Anwendung auf eine Test- oder Staging-Umgebung deployt werden muss. Währenddessen können reguläre, automatische und manuelle Tests laufen. Etwaige Security-Lücken werden vom IAST-Tool während dieser Tests aufgespürt. Daher rührt auch die Bezeichnung *Interactive* bei IAST, denn durch die Interaktivität werden potenzielle Lücken aufgedeckt.

Secret-Detection

Bei der Secret-Detection geht es nicht um das Finden von Vulnerabilitys, sondern um das Aufdecken von Credentials. Geheime Credentials – dazu gehören Passwörter, API-Keys und sonstige Zugangsschlüssel – sollten grundsätzlich nie direkt in ein Repository committet werden.

Passwörter und Git

Der Grund ist simpel: Bei einem Git-Repository handelt es sich um ein verteiltes Versionsverwaltungssystem. Wenn ein Passwort in ein Repository gepusht wird – ganz unabhängig auf welchem Branch –, dann sollte man davon ausgehen, dass dieses Passwort »öffentlich« ist, da zumindest jeder mit Zugang zu dem Repository es sehen kann. Es bringt dann auch nichts mehr, das Passwort wieder aus dem Code zu löschen, denn die Git-Historie speichert es auch nach dem Entfernen. (Es ist zwar theoretisch möglich, die Git-Historie neu zu schreiben, in der Praxis ist das aber irgendwo zwischen »unheimlich aufwendig« und »fast unmöglich«.)

Stattdessen sollten Passwörter, API-Keys und andere Credentials nicht in einem Repository verwaltet werden, sondern in einem separat dafür vorgesehenen Secrets-Management-Tool, in dem die Credentials sicher und mit einem fein-granularen Rollenkonzept verwaltet werden können. Dadurch ergeben sich auch noch weitere Vorteile, zum Beispiel ist das komplette Durchrotieren von Passwörtern viel einfacher möglich.

Zurück zur Secret-Detection: Wenn auf einem Repository initial ein Secret-Detection-Scan ausgeführt wird, muss die gesamte Historie durchsucht werden. Je nach Größe des Repositorys kann das eine Zeitlang dauern. Wenn Secrets gefunden wer-

Persönliches Exemplar für Thomas Bartholomäus

den, sollten diese als kompromittiert betrachtet und entsprechend ausgetauscht werden. Ein Neuschreiben der Historie ist nicht sinnvoll, schließlich liegen die Passwörter geklont irgendwo auf den verschiedenen Rechnern herum.

Innerhalb einer Pipeline sollte daher ebenfalls nach Secrets gesucht werden, um das Problem frühzeitig anzugehen. Lassen Sie den Scan in jedem Feature-Branch mitlaufen, damit Commits mit Secrets frühzeitig erkannt werden. Je nach Dienst können Sie auch bestimmen, dass gefundene Schlüssel automatisch widerrufen werden, der Zugang also direkt gesperrt wird. Das ist sinnvoll, um die Teammitglieder entsprechend zu trainieren: Wenn ihre Passwörter zu Diensten in Repositorys früh gefunden und automatisch widerrufen werden, dann lernen sie automatisch, so etwas nicht zu tun.

Fuzz-Testing

Eine weitere Methode, um Sicherheitslücken zu finden, ist das *Fuzz-Testing*. Das Konzept beim Fuzz-Testing sieht vor, dass man die Anwendung mit etlichen Eingabeparametern »bewirft«, um zu schauen, ob sie abstürzt. Solche Abstürze können Indikatoren dafür sein, dass sich in der Software Sicherheitslücken verbergen, etwa Speicherüberläufe, die man ausnutzen kann.

Es gibt verschiedene Arten von Fuzz-Tests: *Coverage-guided Fuzzing* sowie *Behavioral Fuzzing*. Bei Ersterem liegt der Fokus auf dem Quellcode und man probiert, die Anwendung zum Absturz zu bringen, indem der Test verschiedene Input-Parameter generiert.

Beim Behavioral Fuzzing ist das Konzept etwas anders. Hier wird zunächst festgelegt, wie sich die Anwendung tatsächlich verhalten soll. Erst dann wird sie mit etlichen Input-Parametern gefüttert. Der Unterschied zwischen Soll- und Ist-Werten zeigt mögliche Sicherheitslücken auf.

Reflexion

Als Techniker haben Sie innerhalb Ihres Teams relativ niedrige Hürden, um erste Sicherheitsscanner in Projekte zu integrieren. Es stehen zahlreiche Open-Source-Sicherheitsscanner zur Verfügung, die in Ihre Pipelines eingebunden werden können.

Es ist von großer Bedeutung, ein Verständnis für Sicherheit sowohl innerhalb Ihres eigenen Teams als auch in anderen Teams aufzubauen, um Sicherheitslücken effektiv angehen zu können. Obwohl Open-Source-Scanner bereits einen guten Fortschritt ermöglichen, führt kaum ein Weg an einer kommerziellen Lösung vorbei, wenn Ihr Team Sicherheitslücken bequem und schnell finden und beheben möchte. Dies betrifft insbesondere Workflows und das Vulnerability-Management.

Darüber hinaus ist es wichtig, die Lead-Time für Änderungen so gering wie möglich zu halten. Sicherheitslücken sollten nicht nur im Projekt behoben werden, sondern auch für die Endnutzer umgesetzt werden. In vielen Organisationen liegt der Fokus oft nur auf einem Aspekt.

© Rheinwerk Verlag, Bonn 2024

Es ist ebenso entscheidend, die Barrieren zwischen den Entwicklungsteams und dem Sicherheitsteam abzubauen. Das Sicherheitsteam sollte die Entwicklungsteams bei der Behebung von Sicherheitslücken unterstützen, um eine sichere Anwendung zu gewährleisten. Auf diese Weise wird die Wahrscheinlichkeit minimiert, dass das Sicherheitsteam die Entwicklung blockiert.

11.5 Supply-Chain-Security

Die *Supply-Chain* – auf Deutsch: Lieferkette – ist in der Wirtschaft ein mehrstufiger Teil der Wertschöpfung, um ein Produkt auszuliefern. Wenn für den Shop von *schick-gekleidet.de* etwa ein Anzug produziert wird, besteht die Lieferkette unter anderem aus den verschiedenen Stoffen und Knöpfen sowie aus dem Nähen des Anzugs, bevor er in den Lagern von *schick-gekleidet.de* ankommt und von dort verkauft werden kann.

Eine Supply-Chain gibt es auch in der Software-Entwicklung, schließlich wird mit vielen Tools und Abhängigkeiten gearbeitet, die im gesamten Software-Delivery-Lifecycle benötigt werden.

Abbildung 11.9 stellt dar, welche Angriffsvektoren es auf dem Weg vom Schreiben des Codes bis zur Ausführung gibt. Fehler beim Coden sind also nur eine mögliche Ursache für Sicherheitslücken. Auch danach kann noch viel schiefgehen. Das fällt unter den Begriff *Build Integrity*, also unter die Frage, ob man den genutzten Werkzeugen und den Abhängigkeiten vertrauen kann.

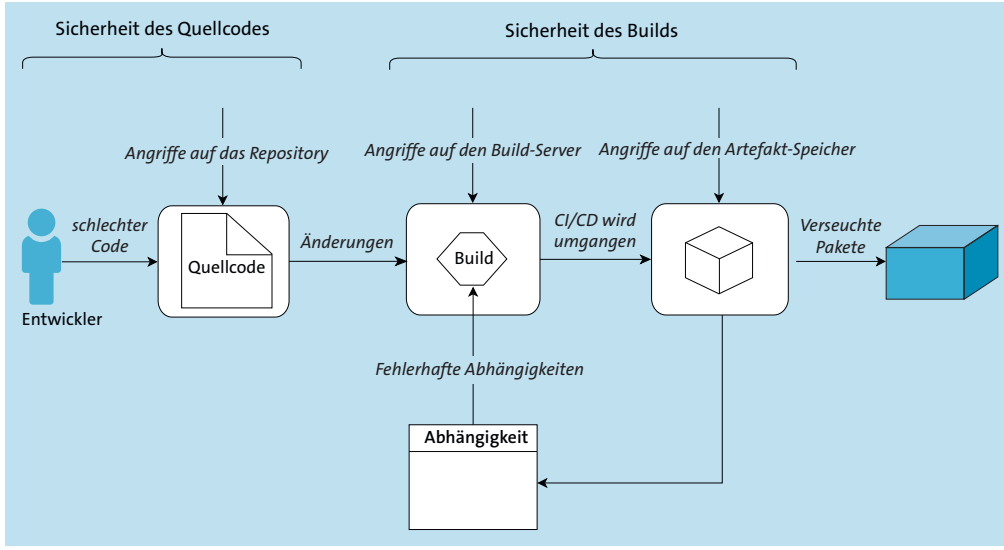

Abbildung 11.9 In der Software Supply-Chain gibt es an vielen Stellen Angriffsvektoren.

11.5.1 Angriffe auf die Supply-Chain

Worum geht es genau bei der Supply-Chain-Security? Anschaulich lässt sich das an zwei Supply-Chain-Problemen beschreiben, die aus gutem Grund groß durch die Presse gingen.

Solarwinds

Orion von Solarwinds ist eine Software, die Netzwerkperformance im IT-Stack misst und überwacht. Eine Monitoring-Software muss per Definition in alle »Ecken« des Systems schauen können und besitzt daher meistens sehr hohe Zugriffsrechte. Die Software ist in vielen großen und komplexen Netzwerken im Einsatz, darunter nicht nur bei großen Firmen, sondern auch bei Behörden und staatlichen Organisationen. Daher war dieser Vektor überaus interessant für Angreifer. Wenn man es also schafft, eine Backdoor in diese Software zu implementieren, dann erhält man Zugriff auf die Netzwerke von allen Kunden, in diesem Fall also fast auf die ganze Welt.

Und genau das ist passiert: Hacker infiltrierten nicht die Netzwerke ihrer eigentlichen Ziele, sondern die Repositorys von Solarwinds (was sehr wahrscheinlich auf äußerst schlampige Sicherheitsregeln zurückging: *https://www.theregister.com/2020/12/16/solarwinds_github_password/*). Dort wurde die Software manipuliert. Dieser Zugriff blieb zunächst unbemerkt, da die eingebaute Backdoor bewusst nicht sofort ausgenutzt wurde – man war ja nicht an der Solarwinds-Infrastruktur interessiert, sondern an den Kunden der Firma. Solarwinds veröffentlichte dann eine verseuchte neue Version, die von den Kunden fleißig aktualisiert wurde.

Grundsätzlich ist das zügige Einspielen von Aktualisierungen eine gute, sinnvolle und auch wichtige Sache. Denn in der Regel werden etliche Sicherheitslücken und Fehler korrigiert, sodass Patches zeitnah eingespielt werden sollten.

In diesem Fall ging damit jedoch das Verderben los, denn über den Update-Mechanismus von Solarwinds wurde die Backdoor auf eine Vielzahl von IT-Systemen großer Organisationen ausgespielt. Unter anderem war Microsoft betroffen, dessen Systeme über diesen Vektor angegriffen wurden:

https://blogs.microsoft.com/on-the-issues/2020/12/13/customers-protect-nation-state-cyberattacks/

log4j

Fast noch schlimmer war das log4j-Sicherheitsproblem, das ich zu Beginn des Kapitels bereits kurz angesprochen habe. Es betraf die weit verbreitete Logging-Bibliothek *log4j*, die in quasi allen Java-Projekten verwendet wird. Eine Schwachstelle (CVE-2021-44228) konnte von Angreifern ausgenutzt werden, um Schadcode in Anwendungen einzuschleusen und so Zugriff auf die betroffenen Systeme zu erlangen. Nachdem diese Lücke bekannt wurde, waren IT-Spezialisten auf der ganzen Welt damit beschäf-

© Rheinwerk Verlag, Bonn 2024

tigt, zu prüfen, ob log4j in ihren Umgebungen eingesetzt wurde (das war fast immer der Fall), und dann entsprechende Updates einzuspielen. Wer keinen guten Überblick über die verwendete Software hatte und betroffene Systeme nicht rasch aktualisieren konnte, hatte große Probleme.

Fazit

In beiden Fällen wurden vermeintlich vertrauenswürdige Softwarekomponenten als Angriffsvektoren benutzt. Ob es das große, renommierte Technologieunternehmen Solarwinds war, das mit seiner professionellen Arbeit wirbt, oder ob es um ein Open-Source-Projekt geht, bei dem der Code und damit das Sicherheitsproblem zur Prüfung für alle Welt offen lag: Sowohl der Einsatz der kommerziellen Profi-Software als auch der quelloffenen Bibliothek führten dazu, dass IT-Infrastrukturen angreifbar wurden, auch wenn die eigentlichen Betreiber eigentlich alles richtig gemacht hatten. Wenn also Teile der Infrastruktur, auf die Sie vertrauen, mit einer Lücke daherkommen, haben Sie leider schlechte Karten.

Was heißt das nun in Hinblick auf DevOps? Klar, Application-Security ist wichtig. Aber es muss ein Blick auf den gesamten *Value-Stream* geworfen werden, um zu schauen, ob Teile der Supply-Chain angreifbar sind.

Dazu haben wir uns bereits zwei Scanner-Typen angeschaut: Dependency-Scanning und Container-Scanning. Beide Teile sind relevant für die Supply-Chain-Security, da fremde Quellen herangezogen und im eigenen Projekt implementiert werden.

Im Hinblick auf das Dependency-Scanning wurde allerdings nur ein Teilaspekt betrachtet. So wurde dort nur geschaut, welche Abhängigkeit in welcher Version definiert ist. Durch den Check in der Vulnerability-Datenbank wurde auf Sicherheitslücken geprüft.

11.5.2 Software Bill Of Materials (SBOM)

Wenn man von Supply-Chain-Security spricht, dann ist die SBOM bald auch ein Thema. SBOM steht für *Software Bill of Materials*. Es handelt sich um eine Liste aller Materialien, die für den Bau der Software genutzt wurden. Diese Liste ist im Wesentlichen das Ergebnis des Dependency-Scannings, bei dem notiert ist, welche Abhängigkeiten ersten Grades in der Software genutzt werden und welche transitiven Abhängigkeiten diese Abhängigkeiten hinter sich herziehen. Dazu gesellen sich die Versionsnummern und die dazugehörigen Lizenzen.

Aus der SBOM lassen sich so die Sicherheitslücken anhand der gelisteten Abhängigkeiten und deren Versionsnummern ableiten. Dadurch soll sichergestellt werden, dass alle Abhängigkeiten bekannt sind und dass bei einer möglichen Attacke auf Komponenten der Supply-Chain zügig reagiert werden kann. Dies ist insbesondere

Persönliches Exemplar für Thomas Bartholomäus

im Hinblick auf Fremdapplikationen wie beispielsweise einer Monitoring-Software wichtig.

Schon bei kleineren Projekten ist es fast unmöglich, Abhängigkeiten »per Hand« zu verwalten, also indem man sich darauf verlässt, dass die Entwicklerteams alle verwendeten Librarys und Module mit der jeweiligen Versionsnummer dokumentieren. In einer DevOps-Umgebung mit automatisierten Builds ist dies auch nicht notwendig, da es durch das Dependency-Scanning stets aktuelle Listen gibt. Wichtig ist, dass diese Listen zentral verwaltet und geprüft werden, d. h., dass diese Informationen auch leicht auffindbar sind und sinnvoll genutzt werden können. Dabei helfen automatisierte CVE-Scanner (*Common Vulnerabilities and Exposures*), die Ihnen basierend auf diesen Infos einen Überblick über alle Sicherheitsprobleme geben.

Weiterhin helfen Compliance-Regeln, die in vielen Branchen ohnehin schon Pflicht sind. Auf diesem Weg werden Regeln festgeschrieben, um sicheres Entwickeln von Software zu gewährleisten und es auch überprüfbar zu machen – darum geht es gleich in Abschnitt 11.6.

Wenn branchenspezifische Compliance-Regeln fehlen, tritt inzwischen zunehmend der Gesetzgeber auf den Plan und erlässt Richtlinien, die ein sorgfältiges Management der Abhängigkeiten vorschreiben:

- Die US-Regierung erließ beispielsweise im Jahr 2021 die *Executive Order 14028* (*https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/*), die besagt, dass jede Software, die in Behörden zum Einsatz kommen soll, eine SBOM liefern muss.
- Im Juli 2023 veröffentlichte hierzulande das BSI die technische *Richtlinie TR-03183* (*https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03183/BSI-TR-03183-2.pdf?__blob=publicationFile&v=3*), in der es um SBOMs geht.
- Auch die EU arbeitet am *Cyber Resilience Act* (*https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act*), der Regeln definiert.

11.5.3 Sicherheit der Build- und Deployment-Server

Gehen wir davon aus, dass Sie durch das Dependency-Scanning einen Überblick über alle Abhängigkeiten Ihrer Software haben und schnell reagieren können, falls dort Probleme bekannt werden. Darüber hinaus müssen Sie noch sicherstellen, dass die Pipeline, die Sourcecode zu fertigen Artefakten baut und ausliefert, sicher ist. Denn wenn Sie Ihrer Infrastruktur nicht vertrauen können, können Sie nie sicher sein, dass sich keine Backdoors in den Paketen verbergen, die Sie ausliefern. Genau das ist ja beim Hack der Solarwinds-Server passiert, und es lassen sich auch noch andere Beispiele finden.

© Rheinwerk Verlag, Bonn 2024

Dass Sie grundlegende Sicherheitshausaufgaben in diesem Bereich erledigen sollten, wird Sie nicht überraschen:

- Patches und Sicherheitsaktualisierungen müssen zügig eingespielt werden,
- Firewalls müssen auf der Build-Infrastruktur ggf. den Netzwerkzugriff einschränken (sowohl auf Betriebssystemebene als auch ggf. als *Web Application Firewall*)
- umfassendes Monitoring sollte Sie über Zugriffe informieren,
- die Kommunikation zwischen den Servern sollte komplett zertifikatsbasiert verschlüsselt sein usw.

Das sind alles wichtige Fleißaufgaben, die in einer modernen Infrastruktur selbstverständlich sein sollten.

11.5.4 Nutzerkonten absichern

Einige fragen sich jetzt vielleicht: »Ist da nicht noch ein Problem der Zugangsbeschränkungen vorhanden?«

Einerseits sollten Sie den Zugriff auf Ihre Server einschränken – je weniger Personen Zugriff auf die Server haben, auf denen die Artefakte gebaut werden, desto besser. Andererseits habe ich ja schon in Abschnitt 5.1 geschrieben, dass ein Build-Server nicht hinter verschlossenen Türen stehen sollte. Für schnelles Feedback und den direkten Austausch im Team und zwischen den Teams ist es viel besser, wenn alle den Code sehen und Änderungen beisteuern können.

Beides schließt sich mit der richtigen technischen Umsetzung gar nicht aus: Die Build-Infrastruktur sollte so weit technisch abgeschottet sein, dass sie immer auf einer sauberen Umgebung die Pakete baut und die Tests ausführt. Gleichzeitig können alle Teams weiterhin die Teile der CI/CD-Pipeline anpassen. Als Quality-Gate dient das Code-Review, über das sichergestellt wird, dass keine Fehler eingepflegt werden.

Das hilft aber natürlich nicht, wenn Konten übernommen werden und das Review umgangen wird. Die Nutzerkonten und eventuell benötigte Service-Accounts müssen auch abgesichert werden, indem Sie ein durchdachtes Rollenkonzept nutzen, das mit starken Authentifizierungsmethoden wie der Zwei-Faktor-Authentifizierung (2FA) eingeschränkt wird. In der Praxis sieht es meist ohnehin so aus, dass das Single-Sign-on des Unternehmens verwendet wird, wo hoffentlich direkt auf eine Mehrfaktorauthentifizierung gesetzt wird. Das ist auch grundsätzlich eher empfehlenswert, um Benutzerkonten automatisch anlegen und sperren zu lassen, wenn neue Mitarbeitende in die Firma einsteigen bzw. Kollegen die Firma verlassen.

RBAC (*Role Based Access Control*, also rollenbasierte Zugriffskontrolle) bedeutet, dass der Zugriff auf wichtige Ressourcen an die Mitgliedschaft in vorher festgelegten Grup-

pen gebunden wird. Zugriffsrechte sollten nicht individuell vergeben werden – wenn Sie einmal damit anfangen, Rechte individuell zu setzen, verlieren Sie in kürzester Zeit den Überblick und können kaum noch nachvollziehen, wer Zugriff worauf hat. Stattdessen gilt grundsätzlich: Zugriffsrechte werden über Rollen verwaltet, einzelne Personen oder Dienste können sich über ein zentrales Management der Identitäten (*Identity Management*, IDM) authentifizieren und erhalten dann entsprechende Rechte. Das sollte stets über das Least-Privilege-Prinzip passieren, also immer möglichst geringe Zugriffsrechte einräumen. Die Visibilität sollte offen sein, Änderungen dürfen nur nach einem Review erfolgen.

Weiterhin ist der Zugang zu den Git-Repositorys relevant. Hier sollte tendenziell eher auf SSH gesetzt werden statt auf HTTPS. Hier gibt es aber häufig Ausnahmen, da primär in größeren Firmen speziellere Regelungen existieren, die es zu befolgen gibt.

[!]

Service Accounts

Ein schwieriges Thema sind Service-Accounts bzw. Bot-Accounts. Häufig werden zusätzliche Accounts benötigt, um Automatisierungen zu steuern. Es muss Governance-Regeln geben, um sicherzustellen, dass diese Bots nicht mit zu vielen Rechten laufen. Wenn diese Zugänge in falsche Hände gelangen und das nicht auffällt, hat man ein Problem.

Ideal ist es, wenn diese Zugänge, die meist ohnehin als Access-Tokens bereitstehen, regelmäßig rotiert werden, damit Tokens invalidiert werden.

11.5.5 Kein Code ist guter Code

Zu guter Letzt gilt: Die wenigsten Probleme machen Abhängigkeiten, die gar nicht existieren. Je weniger Verweise auf fremde Software in Ihren Projekten existieren, desto gelassener können Sie auf Meldungen über Sicherheitsprobleme reagieren.

Ganz so einfach ist diese Empfehlung allerdings nicht: Oftmals ist es besser, bewährte Librarys zu verwenden, als das Rad neu zu erfinden. Wenn es beispielsweise um kryptografische Methoden oder andere sicherheitskritische Bestandteile geht, sollten Sie stets erprobte Tools einsetzen.

Den richtigen Mittelweg werden Sie nur durch Erfahrung finden. Nicht zwangsläufig durch Ihre eigene Erfahrung, sondern auch durch die Ihrer Kollegen und anderer Teams. Nehmen Sie sich daher in Code-Reviews und beim Pair-Programming genug Zeit, um zu diskutieren, ob es sinnvoll und notwendig ist, eine Abhängigkeit einzubinden. Lässt sich das Problem auch mit ein paar eigenen Zeilen Code selbst lösen? Haben Ihre Kollegen vielleicht schon eine Funktion parat, die eine externe Abhängigkeit ersetzen kann, Stichwort *Inner Sourcing*? Oder holt man sich mit einer Eigenentwicklung nur mehr Probleme ins Haus?

© Rheinwerk Verlag, Bonn 2024

Reflexion

Die Software-Supply-Chain ist ein relativ frisches Thema, auch wenn die Grundlagen dazu relativ lange bekannt sind. Der wichtigste Bestandteil für ein Anwendungsprojekt ist das Scannen nach den verwendeten Abhängigkeiten.

Einige Aspekte (wie die Zwei-Faktor-Authentifizierung) erfordern die generelle Schulung aller Mitarbeitenden. Ein Dependency-Scanning wiederum geht mehr auf die Security mit ein. Ob indessen ein SBOM wirklich benötigt wird, hängt von den rechtlichen Grundlagen ab. Helfen wird es auf jeden Fall, insbesondere damit bei neuen schwerwiegenden Lücken schnell geprüft werden kann, ob die eigene Umgebung betroffen ist.

Setzen Sie sich mit den verschiedenen Teilaspekten der Supply-Chain-Security auseinander, und sorgen Sie dafür, dass diese möglichst sicher eingerichtet werden.

11.5.6 Security bei schick-gekleidet.de

Die Auswahl der Security-Tools für die verschiedenen Kategorien war bei *schick-gekleidet.de* auf den ersten Blick nicht so einfach: So lieferten die reinen Security-Tools hervorragende Ergebnisse sowie ein ausgezeichnetes Management von Vulnerabilitys.

Problematisch an diesen Tools war hingegen, dass diese zwar in den Pipelines liefen, aber sich nie so richtig in die Lösungen eingefügt haben. Ein Dashboard war zwar gut und wichtig, allerdings half das alles nicht, weil schon bei einfachen Tests festgestellt wurde, dass die Entwicklungsteams die Dashboards kaum angeschaut haben. Die Einbindung war nur halbgar gelöst, und immer wieder führte das dazu, dass die Ergebnisse ignoriert wurden.

Auch hier orientierten sich die Entwicklungsteams also in Richtung GitHub und GitLab. Damit ist sowohl die Einbindung in die Pipelines einfach als auch die Konfiguration von Merge-Einschränkungen, wenn versucht wird, neue Vulnerabilitys einzuführen.

Reflexion

Sicherheitslücken müssen möglichst früh auffallen, damit sie zeitnah korrigiert werden können, bevor sie in einen Hauptentwicklungsbranch gemergt werden. Entsprechend ist es wichtig, dass Sie dafür sorgen, dass Security-Scanner möglichst früh und regelmäßig in die Entwicklung eingebunden werden. Denn nur so können Lücken während der Entwicklung auffallen.

Wenn Sie verschiedene Scanner gleichzeitig implementieren wollen, dann schauen Sie lieber, dass ein Scanner-Typ nach dem anderen eingeführt wird, um die Teams

Persönliches Exemplar für Thomas Bartholomäus

nicht zu überfordern. Zum Einstieg ist Dependency-Scanning gut und überschaubar umsetzbar. Danach folgt schon SAST.

Ich kann nur empfehlen, dass das Security-Team und die Produktteams auf die gleichen Dashboards und Reports schauen, um die Zusammenarbeit zu fördern. Weiterhin ist es ratsam, dass Pipelines bei gefundenen Sicherheitslücken *nicht* abbrechen, sondern eine Warnung abgeben. Denn False Positives wird es immer wieder geben, und abbrechende Pipelines sind ein unschönes Mittel und können störend sein, ganz unabhängig davon, dass das Vulnerability-Management-Tool den Prozess unterstützen müsste. Hier habe ich schon viele krude Workarounds gesehen, die eher hinderlich als hilfreich waren.

Wichtiges Mantra: *Das beste Security-Tool bringt Ihnen nichts, wenn es nicht genutzt wird, weil es umständlich zu nutzen ist.*

11.6 Compliance

Compliance ist im Rahmen der Software-Entwicklung ein Thema, das noch weniger Begeisterung auslöst als die Security. Während es bei der Security noch um die Abwehr von Hackern und die Absicherung der eigenen Anwendung geht, beschäftigt man sich bei der Compliance »nur« damit, dass Regeln eingehalten werden.

Ist das in einer gelebten DevOps-Kultur überhaupt noch passend? Nicht umsonst rief Mark Zuckerberg einst das Motto »Move fast and break things« für die Entwicklung von Facebook aus: Erst mal machen, um die Konsequenzen kümmern wir uns später.

Dieser Ansatz kann selbstverständlich für ein professionell agierendes Team nicht gelten, und auch Facebook muss sich mit einer Menge Bestimmungen und Regeln auseinandersetzen.

Es gibt im Wesentlichen zwei Gründe für Compliance-Anforderungen:

- Der wichtigste Grund ist, dass es regulatorische Vorgaben gibt. Das sind allgemeine Gesetze, aber auch spezifische Vorgaben beispielsweise für die Automobil- oder Finanzbranche. Hinzu kommen auch konkrete Vorgaben, wie sie beispielsweise in der DSGVO gemacht werden oder sich aus Vorschriften zur Barrierefreiheit ableiten lassen.
- Der zweite Grund leitet sich aus der Größe der Organisation ab. Gerade in großen Konzernen gibt es viele verschiedene Tools und Prozesse, die standardisiert werden müssen und deren Einhaltung durchgesetzt werden muss. Auch wenn das in vielen Einzelfällen nervig und umständlich ist, ist es auch hier wichtig, das große Ganze in den Blick zu nehmen: Gute, durchdachte Regularien machen die Zusammenarbeit einfacher und sorgen für ein besseres Produkt. So bringen die besten

© Rheinwerk Verlag, Bonn 2024

Security-Scanner nichts, wenn die Teams sie umgehen können, weil sie keine Lust auf die Zusatzarbeit haben. Hier trifft die Autonomie der Teams auf die Verpflichtung der Organisation.

Compliance und DevOps sind kein Widerspruch

Kritische, aber nicht fachkundige Stimmen sagen häufig, dass die DevOps-Kultur gar nicht in allen Branchen funktionieren könne, da dann ja jedes Team einfach das machen würde, was es für richtig hält. Es wird also argumentiert, dass die wichtige Autonomie der Teams im Widerspruch zu den Regularien und Gesetzen stehe, die durch einen Compliance-Prozess durchgesetzt werden.

Das ist so natürlich überspitzt und nicht wahr. Richtig ist, dass sich schwerfällige Compliance-Regeln nicht gut mit der DevOps-Idee verstehen: Wenn vor einem Deployment erst vier verschiedene Unterschriften aus dem Vorstand per Fax eingeholt werden müssen, lässt sich nicht agil arbeiten. Das heißt aber nicht, dass es nicht Wege gibt, um Regeln so umzusetzen, dass das Ergebnis schneller, einfacher *und* sicherer wird.

Grundsätzlich ist das Ziel, dass ein sicheres Deployment in die Produktivumgebung möglich ist und dabei die Regulatorik eingehalten wird. Sichergestellt wird dies durch Jobs in der Pipeline, die automatisiert Checks erzwingen. Denn es ist zwar schön und gut, wenn es Security- und Compliance-Regeln gibt; falls allerdings die Teams so autonom sind, dass sie diese einfach abschalten können, dann ist niemandem geholfen.

11.6.1 Compliance-Richtlinien definieren

Woher kommen die Regeln? Es wird in den meisten Firmen ein Team geben, das die Compliance-Regeln für die ganze Organisation und für spezifische Projekte definiert. Dieses Team nutzt Gesetze, allgemeine Bestimmungen und die branchenspezifischen Compliance-Regeln und reichert diese mit den eigenen firmeninternen Regelungen an.

Wichtig ist dabei, dass der *Outcome* im Fokus steht und nicht die Regeln an sich. Häufig sieht man leider ein »Das machen wir schon immer so!«, ohne dass das Regelwerk viel Sinn ergibt. Orientieren Sie sich besser an harten Zahlen und Metriken, und überprüfen Sie stets, dass die formulierte Anforderung im Einklang mit dem Value-Stream steht. Dazu gehört, dass zunächst die Owner der Compliance-Anforderungen definiert werden. Dann müssen der Prozess sowie das *Threat Model* als Ganzes und im Zusammenhang betrachtet werden: Welche Bedrohung gilt es einzudämmen, welche Richtlinien sind notwendig und sinnvoll?

Die Richtlinien können vielfältig sein und betreffen nicht nur die reine Anwendungsentwicklung, sondern auch den Betrieb der Anwendung. In der Software-Entwicklung sieht man häufig folgende Compliance-Regeln:

- Die Test-Coverage muss mindestens bei 75 % liegen.
- Der Coding-Style wird eingehalten und durch Linter überprüft.
- Die Security-Scanner dürfen keine Sicherheitslücken finden, die als `Critical` oder `High` eingestuft werden. Ansonsten darf nicht deployt werden.
- Änderungen dürfen erst gemergt werden, wenn eine weitere Person sie einem Review unterzogen hat. Code, der ins fertige Produkt gelangt oder auf der Produktionsumgebung landet, muss von mindestens vier, besser noch mehr Augen geprüft werden. Wer für die Prüfung zuständig war, muss genauso dokumentiert werden wie der eigentliche Urheber.

Auch im Betrieb sind Compliance-Regeln gängig:

- Alle Datenverbindungen zwischen den Diensten müssten verschlüsselt sein.
- Alle HTTP-Endpunkte müssen per TLS verschlüsselt sein.
- Die laufenden Container müssen regelmäßig auf bekannte Sicherheitslücken geprüft werden.
- Sicherheitslücken, die als `Critical` und `High` eingestuft sind, müssen innerhalb von einer Woche gepatcht werden.

Das sind natürlich nur einige Beispiele. Wichtig ist auch hier: Das Compliance-Team sollte eng mit den anderen Teams zusammenarbeiten und keinesfalls nach Gutherrenart Regeln aufstellen, die sich im Alltag nicht umsetzen lassen.

Policys mit Bedacht einführen

Bedenken Sie bei der Definition der Policys den Entwicklungsstand der Projekte. Wenn man Security-Regeln in einem alten Projekt einführt und es vorher keinerlei Security-Checks gab, dürften der Code und die Abhängigkeiten nur so von gefundenen Sicherheitslücken wimmeln.

Wird gleichzeitig die Compliance-Policy formuliert, dass keine Sicherheitslücken im Projekt enthalten sein dürfen, steht erst mal auf absehbare Zeit alles still. Weder kann es mit der Entwicklung weitergehen, noch können neue Deployments durchgeführt werden. Sinnvoller kann es sein, wenn Sie sich zu Beginn erst mal darauf konzentrieren, dass keine weiteren *neuen* Sicherheitslücken eingeführt werden. Das kann eine pragmatische Lösung sein, wobei der Berg an gefundenen Problemen natürlich nach und nach abgetragen werden muss.

Definieren Sie in den Policys neben der Anzahl und dem Level der Sicherheitslücken auch noch, wer bei gefundenen Lücken trotzdem einen Merge oder ein Deployment genehmigen kann. So lassen sich False Positives oder Sicherheitslücken, die für das Projekt nicht ausnutzbar sind, nach sorgfältiger Prüfung durchwinken, damit die Entwicklung und das Deployment vorangehen können.

© Rheinwerk Verlag, Bonn 2024

Compliance als Prozess

Das Aufstellen der Richtlinien ist nur ein erster Schritt; viel wichtiger ist, dass Compliance als ein Prozess verstanden wird, der das ganze System betrifft. Neben der reinen Dokumentation der Compliance-Regeln muss ein Prozess definiert werden, wie und wann diese zu prüfen sind. Dazu gehört auch, dass die Ergebnisse dokumentiert und festgehalten werden. Es muss immer nachprüfbar sein, dass die Regeln eingehalten wurden – Ihre Rechtsabteilung wird es Ihnen danken.

Die tatsächliche Handhabung von Compliance-Regeln in der Praxis hängt stark von der Art der Implementierung ab. In der »alten« Welt wird viel manuell geprüft, während in der DevOps-Welt viel mehr automatisiert wird. Zu beiden Vorgehensweisen folgt in den nächsten Abschnitten eine nähere Betrachtung, wie ein solcher Prozess aussehen kann und was zu beachten ist.

11.6.2 Manuelle Compliance

In der alten Welt stellte das Compliance-Team eine Checkliste bereit, an die sich die Entwicklungsteams halten mussten. Da nur alle sechs Monate deployt wurde, wurde diese Liste von den Entwicklern erst kurz vor Übergabe an das Betriebsteam abgearbeitet. Für das Betriebsteam gab es kurz vor dem Deployment eine weitere Checkliste, die ebenso abgearbeitet werden musste.

Falls das Compliance-Team an einer Stelle ein Veto einlegte, musste nachgebessert werden. So entstand noch mehr Konfliktpotenzial, denn bei jedem Übergang zwischen den Teams können Compliance-Probleme auffallen, die dafür sorgen, dass der ganze Prozess von vorn durchlaufen werden muss. Das verzögerte die Entwicklung eines Features oder sein Deployment stark und konnte dazu führen, dass mehrmonatige Arbeit vergebens war, wenn am Ende festgestellt wurde, dass etwas nicht stimmte.

Die Teams füllten also diese Checklisten aus und gaben jeweils an, ob sie die Regeln befolgt hatten, inklusive möglicher Kommentare, warum vielleicht etwas nicht umsetzbar war. Das Compliance-Team schaute über die Änderungen und gab diese dann frei.

Bei kleineren Organisationen ist ein solcher Prozess wie in Abbildung 11.10 vielleicht noch durchführbar, da es kurze Wege zum Compliance-Team gibt und die Liste abzuarbeitender Compliance-Punkte überschaubar ist. Problematisch wird es allerdings, wenn in größeren Organisationen eine zeitnahe Abarbeitung unmöglich ist, da das Compliance-Team hoch ausgelastet ist.

Nehmen Sie zum Beispiel die (sinnvolle) Forderung, dass nur Code in der Produktion landen darf, der durch ein Review gegangen ist. Wenn Sie diese Compliance-Regel ernst nehmen, dann müssten Sie eigentlich alle Code-Changes seit dem letzten Deployment überprüfen und schauen, ob es auch wirklich ein Review gab.

Persönliches Exemplar für Thomas Bartholomäus

Das ist offensichtlich vollkommen unmöglich. Und auch in abgeschwächter Form verlängert ein solcher Compliance-Check die Lead-Time potenziell endlos. Zudem skaliert ein manueller Ablauf schlecht. Je größer die Organisation ist und je mehr Regeln eingehalten werden müssen, desto länger dauern solche Prozesse und desto mühseliger wird es auch für die Teams.

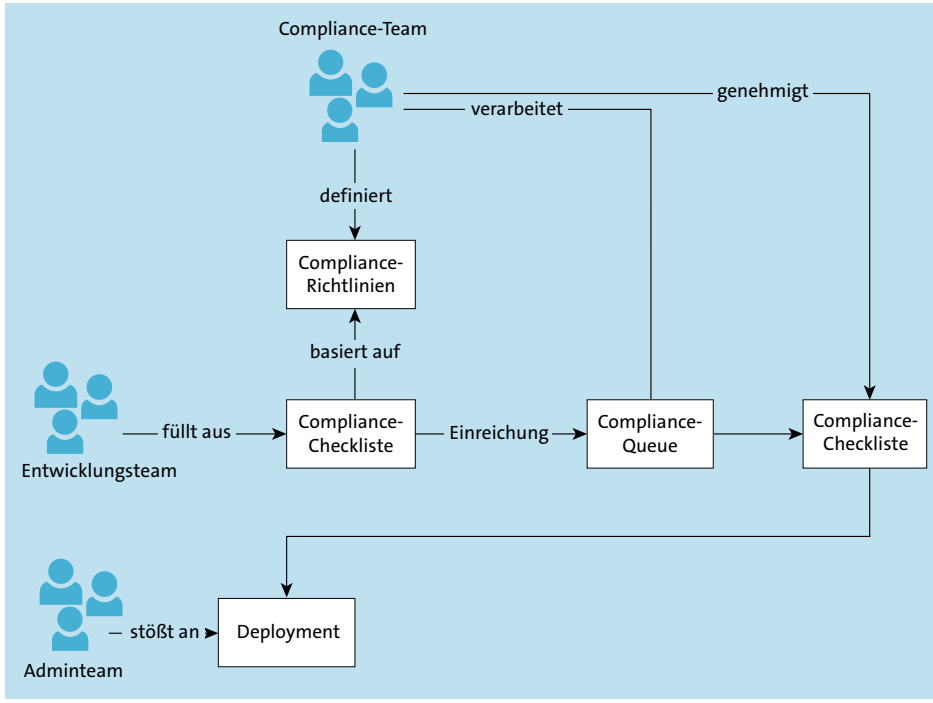

Abbildung 11.10 Die manuelle Abarbeitung ist zeitaufwendig.

11.6.3 Vollautomatische Compliance

Wenig überraschend ist also, dass auf eine mögliche vollautomatische Compliance gesetzt werden sollte. Wie beim manuellen Workflow stellt das Compliance-Team auch weiterhin die Regeln auf, ihre Einhaltung wird nun aber vollautomatisch erzwungen und dokumentiert.

Compliance-Pipelines: Freiheit geben und Regeln erzwingen

Der erste Schritt ist die Definition von Compliance-Pipelines. Wichtig ist genauso wie auch bei allgemeinen Security-Scans, dass ein Shift-Left-Ansatz verfolgt wird: Auch die Compliance-Checks sollten so früh wie möglich im Software-Delivery-Lifecycle erfolgen, damit Fehler und Verstöße dann entdeckt werden, wenn sie entstehen.

© Rheinwerk Verlag, Bonn 2024

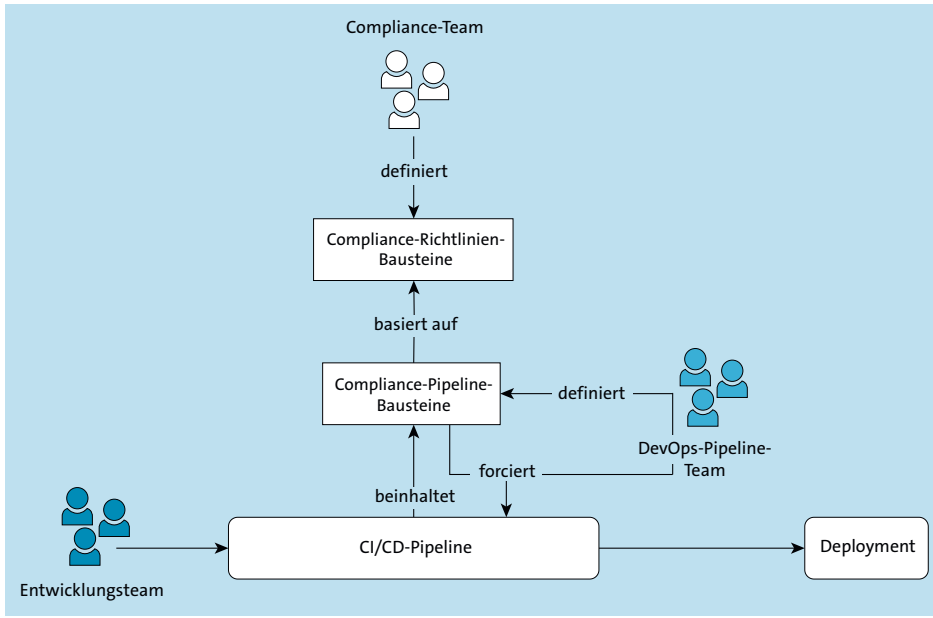

Abbildung 11.11 Das Pipeline-Team entwickelt und forciert die Ausführung der Bausteine für die Compliance-Pipeline.

Neben dem Compliance- und den Entwicklungsteams – die natürlich nach DevOps-Prinzipien arbeiten – kommt nun noch das Team zum Zuge, das die CI-Pipeline betreut (siehe Abbildung 11.11). Das ist besonders für größere Firmen und Organisationen relevant, deren Compliance-Richtlinien genauso skalierbar sein müssen wie die restlichen Systeme.

Das Pipeline-Team stellt grundsätzlich für die verschiedenen Teams die Bausteine bereit, um mehrfache Implementierungen in den einzelnen Teams zu vermeiden. Das gilt nicht nur für reguläre Teile der Pipeline, sondern auch für die Teile, die relevant für die Compliance sind.

Schwerpunkt sind die Security-Scanner, die bei jedem Run ausgeführt werden müssen. Hier gibt es zwei Dinge zu beachten:

- Es muss den Teams so einfach wie möglich gemacht werden, diese Security-Scanner einzubinden.
- Die Teams dürfen die Security-Scanner nicht einfach abschalten können.

Das Grundprinzip ist klar: Die Teams sollten die Compliance-Checks so früh wie möglich im Software-Delivery-Lifecycle durchführen. Und das geht nun mal am besten in der Pipeline, sodass die Checks in jedem Branch regelmäßig ausgeführt werden und Teil des Entwicklungsprozesses werden. Die Teams nutzen dann die bereitgestellten Funktionen des DevOps-Teams. Die Compliance-relevanten Jobs werden dann von

Persönliches Exemplar für Thomas Bartholomäus

der Pipeline automatisch und unumgänglich injiziert. Durch die vollständige Automation wird allen Arbeit abgenommen.

Das Schwierige ist das Erzwingen der Regeln in der Compliance-Pipeline. Hier ist die Wahl des richtigen Tools und der richtigen Pipeline-Architektur essenziell.

Einige Firmen setzen als Lösung auf eine zentrale Pipeline, die vom Pipeline-Team verwaltet wird. Dieser Ansatz führt allerdings zu vielen Problemen, da viele verschiedene Teams verschiedene Software-Stacks nutzen: Das Modell ist also auf Dauer kaum wartbar. Ein solcher zentralisierter Ansatz würde nur zu Frust führen, weshalb er vermieden werden sollte.

Besser ist es, wenn entsprechende Compliance-Bausteine für die Pipeline vorbereitet werden. Die einzelnen Teams haben dann immer noch die Möglichkeit, eigene Teile der Pipeline selbst zu schreiben und zu modifizieren. Das richtige Werkzeug erzwingt aber, dass fest vorgegebene, sicherheitsrelevante Bausteine durchlaufen werden. So wird beispielsweise bei jedem Merge zum Main-Branch geprüft, ob neue Abhängigkeiten Sicherheitslücken mitbringen. Auch kurz vor dem Deployment dienen die Compliance-Bausteine als Quality-Gate: Es können automatisch die Lizenzen der verwendeten Abhängigkeiten gescannt werden. Wird eine problematische Lizenz gefunden, wird kein Deployment durchgeführt.

Ein guter Kompromiss sieht also so aus: Ein Teil der Pipeline muss abgeriegelt werden, ein anderer Teil der Pipeline sollte von den Teams selbst definiert werden können. Abbildung 11.12 zeigt, wie so etwas aussehen kann.

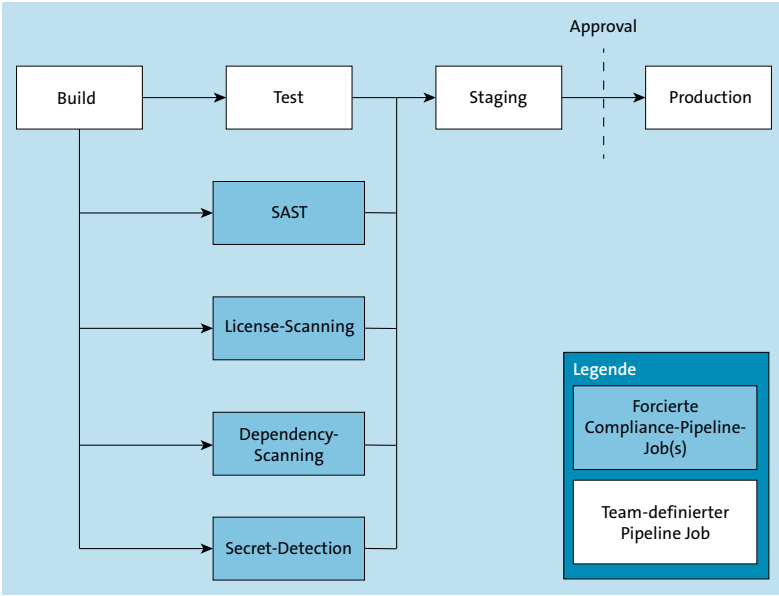

Abbildung 11.12 In der Praxis sieht es dann so aus, dass diverse Compliance-relevante Pipeline-Jobs forciert werden, während das Team eigene Jobs weiterhin definieren kann.

© Rheinwerk Verlag, Bonn 2024

Compliance-Pipelines schreiben

Ich habe bereits angeführt, dass die Wahl des richtigen Tools an dieser Stelle sehr wichtig ist. Während es bei anderen Aufgaben meist unterschiedliche Optionen gibt, die alle die Arbeit gut erledigen, kann ich für das Überprüfen von Compliance-Regeln nur die Pipelines von GitHub und GitLab empfehlen.

Compliance mit Jenkins

Bei *Jenkins* sieht es leider anders aus, da es keine zentrale Plattform aus einem Guss ist. Jenkins erlaubt im Standard sehr viel und ist sehr modular aufgebaut. Alles abzuschotten, ist irgendwo zwischen nicht trivial und unmöglich, was auch damit zusammenhängt, dass das Sourcecode-Management und die CI getrennt voneinander gehandhabt werden.

Die Policys, die in den Pipelines geprüft werden, liegen je nach verwendeter Plattform als Konfigurationsdateien vor. So lassen sich die Inhalte einfach für die Teammitglieder nachvollziehen. Die Policys werden in der Regel in einem separaten Repository gespeichert, also auch versioniert. Die Teams können so selbst im Sinne des Inner Sourcings über Merge und Pull Requests Änderungen vorschlagen, die von den Entscheidern geprüft werden.

Sie können Verfahren wie *Static Code Analysis* oder Security-Scanner einbinden oder benutzerdefinierte Skripte verwenden, um sicherzustellen, dass Ihr Code und Ihre Entwicklungspraktiken den spezifischen Compliance-Anforderungen entsprechen.

Beim Schreiben von Compliance-Pipelines sollten Sie nicht nur darauf achten, dass Sie sich auf das Wichtige konzentrieren, sondern Sie müssen auch noch sehr genau darauf schauen, *wie* sie implementiert werden müssen.

In vielen Firmen gibt es mehrere unterschiedliche Compliance-Pipelines, denn nicht jedes Projekt benötigt jeden Scanner oder muss jede Regel einhalten. Ein Projekt, in dem eine App für Smartphones geschrieben wird, wird kein Container-Scanning benötigen, das Backend für die App wiederum schon. Die Compliance-Pipeline sollte entweder so definiert sein, dass sie automatisch erkennt, ob der Scanner für das Projekt relevant ist, oder Sie müssen mehrere Pipelines definieren, je nachdem, welches Featureset für das entsprechende Projekt gebraucht wird. Das führt allerdings zu Problemen, wenn es mehrere Jobs mit dem gleichen Namen gibt und das CI/CD-Tool nicht weiß, welcher Job nun relevant ist.

Ein automatisches Mergen klingt zwar gut, ist aber in der Praxis nicht realistisch. Stattdessen ist es eher notwendig, dass man eine Hierarchie innerhalb der Compliance-Pipelines selbst aufbaut, indem man die Vererbung selbst nachbaut.

Grundsätzlich gilt hier aber das Gleiche wie auch bei allen anderen Pipeline-Schritten: Die Compliance-Definitionen sollten für die Teams einfach lesbar und nachvoll-

ziehbar sein. Wenn der Compliance-Teil der Pipeline länger ist als die eigentlichen Build-Vorgaben, führt das schnell zu Frust.

Nicht für alle Projekte geeignet

Ich kann nur davon abraten, zu viel festzuschreiben, denn mit jedem Zwang wird die Freiheit der Teams eingeschränkt, was im besten Fall zu Workarounds führt und im schlechtesten Fall zu einem unzufriedenen, genervten und ineffizient arbeitenden Team.

Oft sieht man leider, dass einfach alle Compliance-relevanten Pipeline-Bausteine für alle Projekte der DevOps-Plattform aktiviert sind. So stellt man vielleicht sicher, dass kein Projekt vergessen wird, sorgt aber für viel unnötigen Aufwand. Besser ist es, genauer und differenzierter hinzuschauen: Nicht jeder Code ist für die Produktion gedacht. Nicht jedes Projekt ist ein Software-Entwicklungsprojekt. Nicht jedes Projekt muss streng überwacht werden.

Wenn man die Checks für Projekte erzwingt, für die sie unnötig sind, dann führt das am Ende nur dazu, dass sich eine Schatten-IT bildet, die von den Teams selbst verwaltet wird, weil zu viele Einschränkungen vorhanden sind. Das sollte tunlichst vermieden werden.

Compliance-Komponenten hinzufügen

Compliance-Pipelines sind ein großer und wichtiger Bestandteil der Umsetzung von Compliance-Richtlinien in DevOps-Teams. Aber nicht alle Probleme lassen sich mit selbst geschriebenen Regeln in der Compliance-Pipeline lösen. Dafür müssen Tools von Drittanbietern in die Pipeline eingebunden werden, beispielsweise die in Abschnitt 11.4.3 angesprochenen Security-Scanner, die eigene Vulnerability-Datenbanken mitbringen. Der Container-Image-Scanner kommt zwar von einer fremden Firma, und das Compliance-Team legt nur fest, dass der besagte Scanner genutzt wird, und erzwingt die Ausführung über die Compliance-Pipeline.

Dabei stellt sich nun allerdings die Frage, wer für diese Scanner, Datenbanken usw. zuständig ist? Und was passiert, wenn beim Scan eines Container-Images eine schwerwiegende Lücke gefunden wird?

In kleineren Umgebungen kann es sinnvoll sein, dass sich das Team um diese Aufgaben kümmert, das thematisch am nächsten an diesen Aufgaben dran ist. Das Security-Team ist also für die Scanner zuständig, während die Entwicklungsteams dafür verantwortlich sind, dass stets aktuelle Container-Images verwendet werden. In größeren Umgebungen führt das allerdings wieder zu einem Skalierungsproblem und es ist sinnvoller, wenn ein eigenes Team damit beauftragt wird, die Security-Scanner im Blick zu behalten und bei Problemen dafür zu sorgen, dass die betroffenen Con-

© Rheinwerk Verlag, Bonn 2024

tainer-Images neu gebaut und ausgetauscht werden. Die Entwicklungsteams können sich ganz auf ihre Arbeit konzentrieren, und das Image muss nicht in mehreren Teams neu gebaut werden.

Dieses Komponenten-Team muss dann dafür sorgen, dass in der Pipeline geprüft wird, dass nur die genehmigten Basis-Container-Images verwendet werden (siehe Abbildung 11.13). Solche kleinen, separaten und agilen Teams können ähnlich wie die Plattform-Teams bei Spezialaufgaben helfen, die alle anderen Teams betreffen.

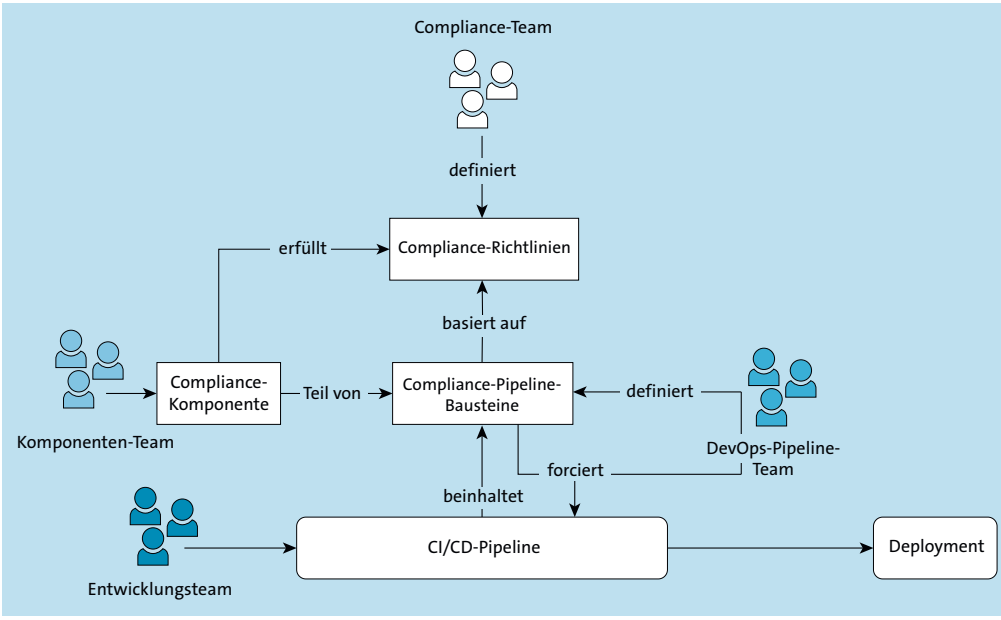

Abbildung 11.13 Das Komponenten-Team stellt Teilkomponenten bereit, die die Compliance-Richtlinien erfüllen, um die Arbeit für die anderen Teams zu reduzieren.

Approvals und Audit-Trail

Indem Sie die Compliance-Richtlinien als Pipeline-Bausteine fest in den Entwicklungsablauf integrieren, erhalten Sie eine einfache Möglichkeit, Genehmigungen (*Approvals*) zu verwalten. Denn natürlich kann niemand manuell überprüfen, ob jede Code-Änderung durch einen Reviewer genehmigt wurde.

Da aber alle gängigen SCM-Tools überprüfen können, dass Merges nur akzeptiert werden, wenn sie von mehreren Personen genehmigt werden, können Sie diese Anforderung vollständig automatisieren. Es ist kein gesonderter Check mehr notwendig, denn der Prozess und die Tools überwachen bereits, dass kein ungeprüfter Code in den Main-Branch gelangt. Das gilt auch für Deployments. Das Approval ist Teil der Prozesse rund um das Deployment und kein gesonderter Schritt davor, sodass nach der Genehmigung automatisch ein Deployment erfolgen kann.

Persönliches Exemplar für Thomas Bartholomäus

Wichtig ist, dass diese Approvals von dem entsprechenden Tool geloggt werden, dass also ein Audit-Trail entsteht, der darüber informiert, was wann geprüft wurde. Die Bausteine der CI-Pipeline werden dazu versioniert, die einzelnen Runs durch die Pipeline protokolliert. Für die Code-Reviews hält das SCM-System direkt in der Git-Historie und in den dazugehörigen Merge bzw. Pull Requests fest, von wem welche Code-Änderung stammt und wer sie akzeptiert hat.

Wenn die Security-Scanner der Pipeline Alarm schlagen, obliegt es zunächst den Entwicklern, die entsprechende Stelle zu prüfen. Wenn Entwickler entscheiden, dass der Treffer nicht relevant ist, muss dies vom Security-Team gegengeprüft werden. Wird gemeinsam ein Approval gegeben, d. h., wird der Code trotzdem in den Main-Branch gemergt, dann muss dies ebenfalls direkt im SCM-System dokumentiert werden. Falls es nun doch auf Produktivsystemen zu Problemen kommt, gibt das Audit-Log Auskunft, seit wann die Lücke besteht und was die Hintergründe der Implementierung waren. Im Rahmen eines Blameless Post-Mortems sollten diese Informationen ausgewertet und verstanden werden.

11.6.4 Compliance bei schick-gekleidet.de

Bisher haben bei *schick-gekleidet.de* alle nach Gutdünken gearbeitet: Irgendwie wurden Änderungen vorgenommen, aber es wurde nie wirklich überwacht, ob diese Änderungen auch gut sind und den eigenen Anforderungen gerecht werden.

Bei der Auswahl der Tools sollte sichergestellt werden, dass die Konfiguration für alle transparent und ohne potenzielle Sicherheitslücken durchgeführt werden kann. Ein entscheidender Faktor bei *schick-gekleidet.de* war, dass neu entdeckte Sicherheitslücken nur nach einer Ausnahmegenehmigung durch das Sicherheitsteam integriert werden durften. Viele Tools erfüllten diese Anforderungen nicht, da sie in der Regel nur das Deployment in Produktionsumgebungen verhinderten.

Daher lag der Hauptfokus erneut auf GitHub und GitLab, da mit ihnen der Shift-Left-Ansatz besonders konsequent umgesetzt werden konnte.

Reflexion

Compliance-Regularien dürfen nicht vernachlässigt werden. Und damit meine ich sowohl die rechtlichen Richtlinien als auch firmeninterne Regeln.

Das Hauptaugenmerk liegt auf der Definition und der Ausführung der Compliance-Pipeline. Man möchte also erzwingen, dass bestimmte Teile in der Pipeline des jeweiligen Projekts ausgeführt werden müssen. Das sind in der Regel Security-Scans. Achten Sie darauf, dass die Compliance-Richtlinien einfach einzuhalten sind und dass die betroffenen Teams dabei unterstützt werden. Vollständige Blockaden durch unnötige Zwischenschritte sollten aufgedeckt und entfernt werden.

© Rheinwerk Verlag, Bonn 2024

Gerade die vermeintlich einfachen Dinge wie die Anzahl der Reviewer im Review-Prozess oder die Einhaltung von Ansprüchen an die Code-Quality und Security bieten den Teams eine gesunde Leitlinie, die das Projekt verbessert, aber die Entwicklung nicht nennenswert einschränkt.

11.7 Fazit

Security ist ein enorm wichtiger Bestandteil im Software-Delivery-Lifecycle. Leider sieht man in vielen Organisationen, dass Security keinen großen Stellenwert genießt. Wenn man sich die Gründe anschaut, dann entdeckt man viele verschiedene Aspekte, die einen nur den Kopf schütteln lassen. Meist fehlt es an Budget und es wird nach dem Motto »Hat ja bisher auch ohne gut funktioniert« gehandelt.

Man neigt oft dazu, zu sagen, dass Security immer wichtiger wird. Das ist natürlich falsch, denn Security war schon immer wichtig! Was man allerdings schon sagen kann, ist, dass die Einbindung von Security-Checks früh im Sofware-Delivery-Lifecycle noch nie einfacher war als heute.

Wenn dann noch die Kultur zwischen Dev, Sec und Ops gut und förderlich ist, können nicht nur Sicherheitslücken frühzeitig gefunden werden, sondern auch zeitnah korrigiert werden, um dann die Patches auszurollen. Und genau darauf kommt es letztlich ja an: Security muss integraler Bestandteil des Software-Delivery-Lifecycles sein, um sichere Software-Entwicklung zu ermöglichen.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 12
Die DevOps-Transformation erfolgreich umsetzen

Die bisherigen Kapitel gaben einen Einblick in die technischen und nicht technischen Faktoren, die beim Arbeiten nach DevOps-Prinzipien eine Rolle spielen. Der Fokus lag allerdings nahezu komplett auf den einzelnen Stages des Software-Development-Lifecycles.

Für den Einstieg in die einzelnen Aspekte war die Unterteilung in die Stages hilfreich, denn so konnten wir die Teamstrukturen und die Techniken isoliert betrachten. An vielen Stellen habe ich immer wieder angemerkt, dass stets gilt: *Menschen über Prozesse über Tools*.

Die besten Tools bringen letztlich nicht viel, wenn die Prozesse Mist sind. Und die besten Prozesse helfen nicht, wenn die Menschen in der Firma ihren Sinn und Zweck nicht verstehen. Und genau das ist das Hauptproblem.

In diesem Kapitel liegt der Fokus auf drei Aspekten: Im ersten Schritt schauen wir uns tiefer und allgemeiner an, welche Methodiken verwendet werden können, um die DevOps-Kultur einzuführen. Denn vor allem zu Beginn der DevOps-Transformation müssen den Teams die Sinnhaftigkeit und die Ziele nähergebracht werden.

Im zweiten Teil des Kapitels zeige ich, wie der Erfolg von DevOps anhand der DORA-Metriken gemessen werden kann. Im dritten Teil gehe ich abschließend darauf ein, was *Value Stream Mapping* ist und wie es genutzt werden kann, um den Value Stream zu optimieren.

12.1 Die DevOps-Kultur einführen

Die DevOps-Kultur einzuführen, ist keine einfache Sache. Schon bei vielen Transformationen von dem Wasserfall-Modell zur agilen Software-Entwicklung sah man immer wieder die gleichen Probleme: Die Menschen wurden nicht richtig abgeholt. Nicht umsonst hört man immer wieder mit starkem ironischem Unterton:

»Wir arbeiten jetzt agil! Bei uns wurde Jira eingeführt!«

Das Gleiche hört man leider auch immer wieder von DevOps-Initiativen:

»Wir machen jetzt DevOps, wir haben jetzt eine CI/CD-Pipeline und Kubernetes!«

Und dann ist diese CI/CD-Pipeline nicht mal vollständig automatisiert, da keine Deployments auf Produktivumgebungen durchgeführt werden. Es gibt noch sehr viel mehr Punkte, aber es ist an der Zeit, das Thema einmal konzeptionell anzuschauen – als Wegweiser für Führungskräfte und als Ideengeber für reine Techniker, die so etwas nach »weiter oben« geben könnten.

Der Weg ist das Ziel

Eine DevOps-Kultur aufzubauen, ist nicht einfach. Ich habe einige Beispiele angebracht, die Sie nutzen können. Nicht jedes ist für Sie womöglich relevant. Zudem ist die Wahrscheinlichkeit hoch, dass Sie in Ihrem Unternehmen andere Paradigmen kennengelernt haben.

Wenn Sie konkrete Aspekte haben, die bei Ihnen funktioniert oder auch nicht funktioniert haben, dann berichten Sie mir doch gern von Ihren Erfahrungen mit einer E-Mail an *mail@svij.org*!

12.1.1 Bottom-up oder Top-down?

Als Erstes stellt sich die grundsätzliche Frage, ob eine DevOps-Transformation einen Bottom-up- oder einen Top-down-Ansatz haben sollte. Bei einem Bottom-up-Ansatz würde die Initiative für die DevOps-Transformation hauptsächlich von denjenigen Leuten ausgehen, die in den Teams sitzen und arbeiten. Bei ihnen handelt es sich explizit nicht um Personen mit Management-Aufgaben.

Bei einem Top-down-Ansatz hingegen würde die Transformation von den Führungskräften initiiert und geleitet werden. Die einzelnen Teammitglieder müssen dann mitziehen. Dabei ist es enorm wichtig, dass möglichst viele Personen in der Organisation richtig abgeholt werden und die geplante Transformation nicht in einen Zwang ausartet. Denn das hilft am Ende niemandem.

Es gibt keine klare Antwort auf die Frage, ob nun Bottom-up oder Top-down der beste Ansatz ist. Das hängt von vielen verschiedenen Faktoren innerhalb der jeweiligen Organisation ab. Tendenziell ist aber erkennbar, dass es bei kleineren Firmen grundsätzlich einfacher ist, einen Bottom-up-Ansatz zu verfolgen. Allerdings müssen in diesem Fall die Mitarbeitenden die Initiative ergreifen und ihre Führungskräfte überzeugen. Je mehr Menschen in der Führungsetage mit einer »Aber wir machen das doch schon immer so«-Mentalität arbeiten, desto schwieriger wird das.

Je größer die Firma ist, desto unumgänglicher ist ein Top-down-Ansatz. In größeren Organisationen können einzelne Individuen meist nicht sehr viel bewegen, da schon sehr viele Strukturen und Prozesse vorhanden sind, die große Initiativen nicht wirklich ermöglichen. Häufig sind Initiativen auch gar nicht erwünscht.

© Rheinwerk Verlag, Bonn 2024

Die größte Herausforderung besteht generell darin, dass Führungskräfte DevOps-Initiativen nicht unterstützen. Es ist schwierig zu sagen, ob eine DevOps-Transformation gelingen kann, wenn der Druck nur von unten kommt. Wenn Führungskräfte keine Probleme im Status quo sehen, dann hilft am Ende alle Begeisterung der Mitarbeiter für eine Transformation nichts.

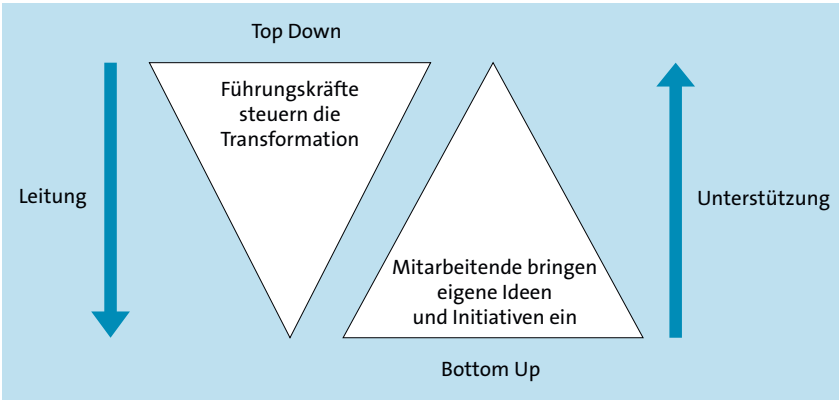

Abbildung 12.1 Erfolgreiche DevOps-Transformationen funktionieren nur, wenn beide Seiten zusammenarbeiten.

In Wahrheit wird sich letztlich weder ein ausschließlicher Bottom-up-Ansatz noch sein Top-down-Gegenstück umsetzen lassen, sondern *alle* müssen an einem Strang ziehen. Einige Mitarbeiter wie Führungskräfte müssen überzeugt werden, andere müssen zwangsläufig mitgezogen werden, damit die Transformation gelingen kann. Idealerweise kommen Ideen und Initiativen von allen Seiten.

Viele Punkte sind schon im Laufe des Buches angeführt worden. Die benötigten Änderungen an der Organisations- und Teamstruktur sind wohl das Auffälligste. Tatsächlich ist das aber der zweite Schritt. Der erste Schritt besteht darin, dass die Führungskräfte zunächst grundsätzlich den Schmerz erkennen und handeln wollen. Wenn Sie eine Führungskraft sind, die etwas verändern möchte, dann ist es schon mal hervorragend, dass Sie dieses Buch lesen.

Wenn in der Führungsriege allerdings bislang kein Verständnis für die Notwendigkeit einer Transformation vorhanden ist, dann hilft es prinzipiell nur, wenn auch die Schmerzen, die es gibt, weiter nach oben getragen werden. Falls es kaum oder nur wenige Möglichkeiten gibt, Feedback weiter nach oben zu geben, dann sehen die Aussichten für eine Transformation schon mal miserabel aus, schließlich ist eine offene Feedback-Kultur auch ein Grundbaustein von DevOps.

Wenn nun Feedback nach oben gegeben wird, dann ist es auch immer wichtig, dieselbe Sprache zu sprechen wie die Führungskräfte. Häufig sieht man leider, dass fal-

Persönliches Exemplar für Thomas Bartholomäus

sche Ansätze gewählt werden: dass nämlich nur einzelne Probleme hervorgehoben werden, ohne das große Ganze zu betrachten.

So höre ich häufig von den Technikern folgende Aussage: »Wir wollen mehr Automatisierung haben, damit wir uns auf die wichtigen Arbeiten konzentrieren können.« Diese Aussage klingt zunächst gut, denn eine Fokussierung auf wichtige Arbeit ist ja nicht unerheblich.

Je nachdem, wie sie von weiter oben in der Hierarchie angenommen wird, kann diese Aussage allerdings ganz anders verstanden werden: »Die Leute sind faul und wollen weniger arbeiten.« Das Empfinden hängt natürlich stark von der Firmenkultur ab.

Wichtiger wäre also, dass die Techniker primär betonen, wie Effizienzsteigerungen umgesetzt werden können. Ich habe im Laufe der Zeit gelernt, dass es sinnvoller ist, die Aussage so umzubauen, dass auf Effizienzsteigerungen hingewiesen wird, die letztlich die Projekte schneller ans Ziel bringen, was am Ende dann auch günstiger für die Firma ist.

Fakt ist allerdings auch, dass nicht jeder im Team die gleichen Probleme sieht. Menschen sind Gewohnheitstiere, und viele wollen nun einmal keine Veränderung: Schließlich arbeitet man schon immer so, und das hat doch funktioniert. Auch mehr Verantwortung für die Teams klingt erst einmal gefährlich (was ich übrigens nachvollziehen kann). Daher gilt es, auch zögerliche Personen entsprechend abzuholen.

12.1.2 Erste Schritte in der DevOps-Transformation

Im ersten Schritt – nachdem feststeht, dass eine Transformation in Richtung DevOps geschehen soll – müssen die Leute in der Firma abgeholt werden. Es dürfte Sie an dieser Stelle nicht mehr überraschen, dass damit nicht nur Dev und Ops gemeint sind, sondern auch Mitglieder aus den Security- und Qualitätssicherungsteams. Und das geht nicht von heute auf morgen, sondern ist ein mehrstufiger Prozess.

Die DevOps-Transformation lässt sich im Wesentlichen in den folgenden fünf Schritten zusammenfassen:

- **Schritt 1:** Definition der DevOps-Vision und -Strategie
- **Schritt 2:** Maßnahmen zur Akzeptanz schaffen
- **Schritt 3:** Ein DevOps-Transformationsteam schaffen
- **Schritt 4:** Mit dem ersten Value Stream starten
- **Schritt 5:** Messen und iterieren
- **Und, ganz wichtig:** Jedes Umfeld ist anders.

Betrachten Sie diese fünf Schritte, auf die ich im Folgenden näher eingehe, *nicht* als unveränderliche, harte Schritt-für-Schritt-Anleitung! Es gibt zu viele Faktoren, die den Erfolg oder Misserfolg beeinflussen, und diese sind bei vielen Firmen unter-

© Rheinwerk Verlag, Bonn 2024

schiedlich. Mein Rat daher: Nutzen Sie diese Schritte als Inspirationsvorlage, und passen Sie sie entsprechend den eigenen Ideen, Erfahrungen und Gegebenheiten an.

Schritt 1: Definition der DevOps-Vision und -Strategie

Im ersten Schritt müssen die Führungskräfte klar definieren, was die genaue DevOps-Vision ist und wie die generelle Strategie sein soll, mit der sie umgesetzt wird. Ohne eine klare Definition kann eine Transformation Richtung DevOps nicht sinnvoll gelingen.

Dafür ist es essenziell, die Mitarbeiter und Mitarbeiterinnen in der Organisation abzuholen. Ohne eine klare Vision und ohne Strategie werden die Mitarbeiter nicht verstehen, warum sie nun anders arbeiten sollen, und die Begeisterung für Veränderung wird gering sein.

Zunächst sollte also klargemacht werden, was DevOps ist und was nicht. So ähnlich, wie ich es auch hier im Buch getan habe. Wichtig ist es, den Fokus auf die Mentalität zu setzen und auch immer zu erklären, was für Vorteile für die Mitarbeitenden der Organisation entstehen.

Als Führungskraft klingen »kurze Lead Times, häufige Deployments und schnellere Auslieferung von Änderungen an die Endnutzer« natürlich erst mal gut, schließlich erreicht die Arbeit schneller die Nutzer, was diese meistens gut finden.

Wenn allerdings die Mitarbeitenden hören, dass Beschleunigung das Ziel ist, bekommen viele zunächst Angst. Wenig verwunderlich: Sie kennen die neue Arbeitsweise nicht und sie klingt erst mal nach viel mehr Arbeit und viel mehr Stress. Und am Ende hat nur der Chef mehr Geld in der Tasche. Super! Da würde auch ich Angst bekommen.

Daher ist es wichtig, dass der Kollaborationsaspekt, die geteilte Verantwortung und das Knowledge-Sharing betont werden und dass man sich langsam dort hinbewegt. Ohne eine gewisse Empathie wird es sonst schwierig. Dennoch muss Ihnen klar sein, dass man nicht alle abholen kann, denn einige Personen muss man letztlich zu ihrem Glück zwingen. Die Zahl der Unwilligen gilt es natürlich so klein wie möglich zu halten.

Zur Vision und zur Strategie gehört es auch, mitzuteilen, wie die Teamstruktur zukünftig aussehen soll. Wichtig ist hier, dass es sich eben nicht nur um Dev und Ops handelt, sondern dass auch QA und Security einen Anteil haben werden.

In diesem Zusammenhang muss ebenfalls noch deutlich gemacht werden, *warum* wir das Ganze machen und *was* man sich davon erhofft. Effizienzgewinne und höhere Produktivität sind immer wichtig – allerdings wenig nachvollziehbar für die ausführenden Mitarbeitenden, denn diese wollen in der Regel einfach nur ihre Arbeit erledigen.

Persönliches Exemplar für Thomas Bartholomäus

Und genau das ist ein Ansatzpunkt, den es zu thematisieren gilt, damit Sie Ihre Kollegen und Mitarbeiter für die Idee begeistern. Die *Software-Development- und -Delivery-Experience* wird sich durch eine solche DevOps-Transformation erheblich verbessern, sofern diese ordentlich gestaltet wird. Den Begriff *Software-Development- und -Delivery-Experience* lehne ich an »Developer-Experience« an, bei der es darum geht, wie angenehm die Arbeitsumgebung für einen Developer ist.

Da unser Ansatz aber nicht auf reine Developer limitiert ist, sondern auch Betriebs-, Qualitätssicherungs- und Security-Teams erfasst, spreche ich hier von »Software-Development- und -Delivery-Experience«.

Wenn man es also schafft, den Ansatz so darzustellen, dass durch den offeneren Umgang, durch wenig sinnlos eingeschränkte Umgebungen und Tools das Arbeiten allgemein einfacher wird, dann können viel mehr Leute etwas mit ihm anfangen.

Feedback-Kanal

Dazu kommen ganz viele Punkte, die auffallen werden, wenn man darüber nachdenkt. Deswegen ist es wichtig, einen einfach zu nutzenden Feedback-Kanal einzurichten, um regelmäßig Stimmungsbilder einzuholen: Was denken die Mitarbeiter? Halten sie die Maßnahmen für gut?

Es sollte nicht nur der Eindruck vermittelt werden, dass man zuhört, man sollte auch *aktiv* zuhören. Denn es werden einige Sorgen und Fragen aufkommen, die man ernst nehmen sollte.

Es gibt verschiedene Möglichkeiten, einen Feedback-Kanal einzurichten. Je nachdem, wie bisher in der Firma gearbeitet wurde, traut sich nicht jede Person, offen zu kommunizieren. Nicht jeder möchte namentlich erkennbar sein. Daher können anonyme Kanäle hilfreich sein, aber auch nicht anonyme, wo alle miteinander diskutieren können.

Hilfreich kann es etwa sein, dass das Management der jeweiligen Teams damit beauftragt wird, die Stimmungsbilder in den Teams abzuholen. So wird deutlich klarer, wo es »hakt«, und auch eine Managerin kann besser einschätzen, was mögliche Kritikpunkte innerhalb des eigenen Teams sind.

Auch sind konkrete Umfragen möglich. So kann aktiv nachgefragt und bewertet werden, wie das Stimmungsbild ist. Reine textbasierte Feedback-Formulare mögen für die eine oder andere Person vielleicht nicht sonderlich hilfreich sein. Meistens melden sich ohnehin nur diejenigen zurück, die etwas zu bemängeln haben.

Solche Umfragen sollten Sie während der Transformation regelmäßig durchführen. So ist der Einblick in das Stimmungsbild immer aktuell, und es ist zu erkennen, ob sich die DevOps-Transformation in die gewünschte Richtung entwickelt.

Das Feedback, das über die Kanäle hereinkommt, sollte daher wie Daten behandelt werden. Datengetrieben lässt sich etwas viel besser umsetzen, wenn auch etwas

© Rheinwerk Verlag, Bonn 2024

messbar wird. Das ist nicht so ganz einfach, denn eine stabile Datenlage muss auch erst aufgebaut werden.

Fehlerkultur

Ein wichtiger Punkt, der direkt zu Anfang klargemacht werden sollte, ist, dass sich auch an der Fehlerkultur etwas ändern muss. *Blameless Post-Mortems*, die in Abschnitt 9.4 behandelt werden, werden zu Beginn nicht immer *blameless* sein, und oft wird man immer noch versuchen, Fehler zu vertuschen.

Eine Idee, um direkt zu Beginn einen soliden Start bezüglich der Fehlerkultur hinzulegen, besteht darin, dass auch Führungskräfte mal berichten, was für Fehler sie gemacht haben und wie sie mit ihnen umgegangen sind.

In vielen Firmen sieht man leider immer wieder, dass Führungskräfte sich so darstellen, als würden sie nie Fehler machen. Aber natürlich unterlaufen auch ihnen Fehler, und aus diesen wird auch gelernt – hoffe ich jedenfalls. Eine offene Fehlerkultur sollte also nicht nur für die Mitarbeitenden gelten, sondern auch für die Führungskräfte, die nun mit gutem Beispiel vorangehen können.

Mitarbeitende ohne Führungsverantwortung in den Teams bekommen auf diese Weise einen viel besseren Einblick, wie die eigenen Führungskräfte arbeiten, was bedauerlicherweise häufig fehlt. Und durch solch eine offene Kultur werden Probleme und Herausforderungen eher berichtet, als wenn die Führungskräfte unerreichbar wirken.

Damit noch einmal deutlicher wird, wohin eine gute Fehlerkultur führen kann und wie diese auch aufgebaut hat, möchte ich Ihnen vom folgenden realen Beispiel erzählen. Ein neuer Mitarbeiter fängt in einem Unternehmen an. In der ersten Woche geht er mit einer Arbeitskollegin zum Mittagessen in die Kantine, wo die beiden auf den CTO treffen. Die Arbeitskollegin stellt den Neuen beim CTO vor. Dieser heißt ihn willkommen und fragt direkt: »Hast du schon die Produktionsumgebung einmal kaputt gemacht?«

Die Reaktion des neuen Mitarbeiters war wenig überraschend: »Was? Ich? Nein, natürlich nicht!«

Und was antwortete der CTO? »Warum denn nicht?«

Dieses Beispiel zeigt, wie so etwas laufen muss. Der CTO macht direkt zu Beginn der Arbeit dem neuen Mitarbeiter deutlich, dass es völlig in Ordnung ist, dass er Fehler begeht. Zudem legt es nahe, dass er sich schnell die Hände »schmutzig« machen soll, um die Infrastruktur besser kennenzulernen. Natürlich fehlen hier noch einige Details, etwa dass es ein Error-Budget gibt, das der neue Kollege nutzen und daher mit Bedacht arbeiten sollte.

Und genau so müssen alle arbeiten und handeln: Fehler zu machen ist okay, wenn denn verantwortungsvoll gearbeitet wird.

Persönliches Exemplar für Thomas Bartholomäus

Im Rahmen der Präsentation der Vision und Strategie lässt sich viel anbringen. Die bisher vorgestellten Punkte sind nur eine kleine Auswahl an Beispielen, die man wählen kann. Es gibt sicherlich noch einige weitere Methoden, die man nicht außer acht lassen. Aber das ist dann bekanntlich nur der erste Schritt.

Fazit

Bevor man nach der Präsentation von Vision und Strategie mit der Transformation fortfährt, sollte ohnehin noch ein bisschen Zeit vergehen, in der man das gesammelte Feedback auswertet. Erst dann sollte es zum nächsten Schritt gehen.

[!]

Zusammenfassung des ersten Schritts

- Kommunikation: Was ist DevOps? Was ist es nicht?
- Vision und Ziel für die Organisation definieren und präsentieren
- Änderungen der Teamstruktur beispielhaft darstellen
- Hervorheben, dass sich die Developer-Experience verbessern wird
- erstes Voranschreiten bei der Fehlerkultur
- Feedback-Kanal schaffen

Schritt 2: Maßnahmen zur Akzeptanz schaffen

Nachdem man im ersten Schritt die Vision und die Strategie mitgeteilt hat, ist es an der Zeit, mit Maßnahmen zu starten, um die Silowände zwischen den Teams aufzubrechen. Allerdings sollte man an dieser Stelle *nicht* direkt mit der Umstrukturierung der Teams beginnen! Solch ein Vorgehen wäre in diesem früheren Stadium eher kontraproduktiv, da zu wenig Einblick in die Aufgaben und Tätigkeiten des jeweils anderen Team existiert.

Der Fokus sollte also zunächst darauf gelegt werden, dass bei allen Mitarbeitenden ein möglichst gutes Verständnis davon aufgebaut werden muss, wie die jeweils anderen Rollen arbeiten. Denn erst danach ist es sinnvoll, die Teamstrukturen aufzubrechen.

Die Maßnahmen müssen daher wohlüberlegt sein, damit dieser Lernprozess nicht abschreckend ist. Man muss sich überlegen, was man alles machen kann, damit die Leute miteinander reden. Dazu existieren sehr viele Teambildungsmaßnahmen, die nicht nur an DevOps gerichtet sind und sicherlich auch hier helfen können.

In diesem Abschnitt fokussiere ich mich daher auf zwei Maßnahmen, die man durchführen kann, und zwar mit konkreterem DevOps-Blick. Die erste Maßnahme ist eine Art Retrospektive und die zweite Maßnahme ist ein »Teamtausch«, bei dem über einen gewissen Zeitraum einzelne Personen zwischen verschiedenen Teams, die bisher getrennt voneinander gearbeitet haben, getauscht werden.

© Rheinwerk Verlag, Bonn 2024

Teamübergreifende Retrospektiven

Retrospektiven kennen viele von Scrum und dem agilen Projektmanagement. Diese finden am Ende des Sprints statt. Das Problem bei Retrospektiven ist allerdings, dass diese häufig nur innerhalb des Teams selbst abgehalten werden und nicht zwischen den Teams. Da wir die Zusammenarbeit zwischen den verschiedenen Teams und Rollen fördern wollen, kann eine zweite Retrospektive ebenfalls helfen, während der alle Teams am Ende des Sprints zusammenkommen und ihren Teil präsentieren.

Das setzt voraus, dass die Sprints innerhalb der Organisation zur selben Zeit enden. Wenn dies nicht der Fall ist, dann ist das eine Maßnahme, die man vorziehen sollte und die sich ohne großen Umbruch umsetzen lässt.

Die Rolle des Moderators ist für solche Retrospektiven essenziell. Denn die Punkte, die im Folgenden genannt werden, müssen strukturiert und gesteuert untergebracht werden. Jedes Team und jede Person sollte genug Zeit haben, um seine bzw. ihre Ergebnisse und Erfahrungen zu präsentieren. Je nach Gruppe können sich sonst Gruppendynamiken einschleichen, die nicht produktiv sind.

Neben der Präsentation des Fortschritts der jeweiligen Teams ist es ebenfalls wichtig hervorzuheben, auf welche Fehler, Probleme oder Unzulänglichkeiten sie gestoßen sind. Wichtig ist, dass man stets darauf achtet, dass hier keine Vorwürfe aufkommen. Das sieht man leider nicht zu selten, und es ist stellenweise auch menschlich. Das ist jedoch ein essenzieller Punkt, an dem der Moderator der Retrospektive bei Bedarf rechtzeitig eingreifen muss.

Die Präsentation der Probleme und Herausforderungen hilft den anderen Teams, im Idealfall ebenfalls das ganze System zu verstehen, was Teil des ersten Weges von »The Three Ways« ist, was in Kapitel 1 ausführlich betrachtet wurde. Dazu gesellt sich ebenfalls noch der Fakt, dass die Teams immer anbringen können, dass sie die eine oder andere Sache zukünftig noch ausprobieren wollen, unabhängig davon, ob bisher gute Kenntnisse dazu vorhanden sind oder nicht.

Häufig beobachtet man dann, dass sich nach dieser Ansage schon die ersten Personen melden und berichten, dass sie die gleiche Sache auch bereits in ihrem Team angeschaut haben – oder auch privat angeschaut haben. Hier lassen sich also direkt Synergien verwenden, wenn solche Personen dann im Anschluss das entsprechende Team unterstützen.

So lässt sich mit einfachen Mitteln die Kommunikation fördern: Alle Beteiligten bekommen nicht nur einen besseren Einblick in die Arbeitsweise der anderen Teams, sondern können nun auch einschätzen, warum einige Hürden so gebaut sind, wie sie sind. Und das hilft enorm.

Persönliches Exemplar für Thomas Bartholomäus

Temporärer Teamtausch

Die zweite Maßnahme ist der sogenannte »Teamtausch«. Andere Begriffe, die ebenfalls passen könnten, die mir aber aus diversen Gründen nicht sonderlich gefallen, sind »Inspektion«, »Hospitation« oder »Praktikum«.

Die Idee ist, dass ein Rollentausch einzelner Personen zwischen den verschiedenen Teams stattfindet. Das ist ebenfalls eine Möglichkeit, um die Kommunikation zwischen den Personen im Team zu stärken. Das ist schon weitestgehend ein Schritt weiter, im Vergleich zu den teamübergreifenden Retrospektiven.

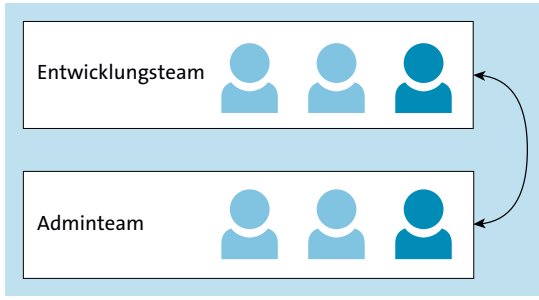

Abbildung 12.2 Einzelne Teammitglieder wechseln temporär das Team.

Konkreter Gedanke ist hier, dass eine Person (zum Beispiel aus dem Entwicklungsteam) in das noch getrennt arbeitende Betriebsteam inkludiert wird, um dort einen Einblick in die Aufgaben und Arbeitsweisen zu erhalten.

Für diesen Tausch reichen nur ein, zwei Stunden sicherlich nicht aus. Stattdessen wird ein längeres Zeitcommitment benötigt. So kann eine Besuchseinheit gut und gerne mal vier Stunden dauern, und sie sollte häufiger über einen gewissen Zeitraum wiederholt werden. Der Gedanke dahinter ist, dass ein Entwickler über einen längeren Zeitraum viel besser mitbekommt, wie das Betriebsteam arbeitet. So kann der Entwickler ihnen während regulärer Arbeiten über die Schulter schauen, aber kann auch mit dabei sein, wenn die Anwendung deployt wird.

Das Über-die-Schulter-Schauen ist natürlich symbolisch zu verstehen. Es sollte schon aktiv kommuniziert werden. Problematisch könnte nur werden, dass das einigen Leuten nicht gefällt. Es bietet sich an, die ersten Personen und Teams auf Basis der Ergebnisse einer Feedback-Umfrage explizit auszuwählen, um mit änderungswilligen Personen zu starten.

Netter Nebengewinn ist, dass durch diese Kommunikation implizit eine Feedback-Schleife eingeführt wird, da der Entwickler vom Betriebsteam immer wieder Informationen mitbekommt, die er sonst nicht bekommen würde und die er in den nächsten Sprint mitnehmen kann. Nicht selten habe ich beobachtet, dass eine Person aus dem Betriebsteam dadurch erst verstanden hat, wie und warum das Entwicklungsteam gewisse Übergangslösungen gebaut hat und wie dies mit den Arbeiten

© Rheinwerk Verlag, Bonn 2024

des Betriebsteams zusammenhängt. Das ist natürlich auch andersherum immer wieder der Fall.

Häufig nutzen die Teammitglieder direkt die Chance, um darüber zu reden, was sie am meisten stört. Hierdurch kann man, selbst ohne DevOps, schon schnell einen großen Gewinn erzielen, noch bevor man überhaupt richtig angefangen hat. Wenn die Änderungen dann auch relativ zeitnah umgesetzt werden, ist das häufig schon ein erster guter Schritt, um die Silos abzubauen. Schließlich hat hier jemand einem anderen Team direkt geholfen.

Die Informationen, wie die Teams arbeiten, müssen allerdings nicht immer nur durch diesen Mitarbeiteraustausch zwischen den Teams fließen. Einige Punkte, die immer wieder auffallen oder die ein oder gar mehrere Teams betreffen, lassen sich in Form von Vorträgen oder Workshops erörtern. Dabei dient der Teamtausch als einfacher Trigger.

Die größte Herausforderung ist allerdings auch, dass der Teamtausch kurzfristig einen gewissen Produktivitätsverlust mit sich bringt. Wer allerdings die ganze Initiative zur DevOps-Transformation ernst nimmt, der wird sehen, dass der mittel- und langfristige Produktivitätsgewinn sehr viel höher ist.

Auch hier ist eine gewisse Iteration nötig, um zu schauen, wie oft und mit welcher Intensität das Tauschen der Teams umgesetzt werden soll. Nach einigen Tauschvorgängen dürfte schon deutlich klarer sein, was die jeweiligen Herausforderungen des jeweils anderen Teams sind, und dann wird diese Initiative auch nach einer Weile auslaufen müssen.

Wichtig ist zudem, dass dieser Austausch kontrolliert erfolgt. Wenn man den Teams es einfach selbst überlässt, dann passieren zwar ein paar Teamtausche, allerdings schläft das rasch ein. Hier sollten Abteilungs- und Teamleiter ein genaues Auge darauf haben, dass hier Ziele gesetzt werden, etwa mittels OKRs, und dass diese regelmäßig getrackt werden. Unter Umständen muss nämlich nachgesteuert werden.

OKR

Objectives and Key Results (kurz: OKR) können eine gute Methode sein, um eine Strategie agil umzusetzen. Mit dem »Objective« wird angegeben, was man erreichen möchte, und mit dem »Key Result«, wie man Ergebnisse oder den Fortschritt misst. Diese Ergebnisse lassen sich für solche Umsetzungen gut nutzen. Über den effektiven Einsatz von OKRs gibt es eigene Literatur.

Der ganze Tausch sollte dann noch mit den verschiedenen Teams erfolgen, also nicht nur zwischen Dev und Ops stattfinden, sondern er sollte einmal quer durch die verschiedenen Rollen geschehen: neben Dev, Ops und QA auch Security. Wie im letzten Kapitel erwähnt, sind die Wände zwischen dem Security-Team und den übrigen Teams besonders dick.

Persönliches Exemplar für Thomas Bartholomäus

Zusammenfassung des zweiten Schritts

- teamübergreifende Retrospektiven durchführen
- Teamtausch von Mitgliedern durchführen

Schritt 3: Ein Team für die DevOps-Transformation schaffen

Zu Beginn des dritten Schrittes sollten die wichtigsten Grundlagen schon absolviert sein: Die Teams sollten so weit abgeholt sein und die Ziele, Aufgaben und Herausforderungen der verschiedenen Rollen im Wesentlichen verstanden haben. Bis das der Fall ist, dürften einige Monate vergangen sein. Ein üblicher Fehler ist, dass das Ganze mit viel Ambitionen zu schnell durchgeführt wird, was für die Initiative nicht förderlich ist.

Im dritten Schritt soll nun das DevOps-Transformationsteam gegründet werden. Dieses Team hat zwei Aufgaben. Die erste Aufgabe ist die Bestimmung und die Betreuung der DevOps-Toolchain. Obwohl ich sie hier erst im dritten Schritt benenne, startet diese Aufgabe schon viel früher, nämlich ebenfalls zu Beginn des ersten Schritts.

Aber warum wird das Team dann erst im dritten Schritt gegründet? Nun, das Team ist in der zweiten Aufgabe auch dafür zuständig, in die Teams zu gehen, um bei der Adaption der DevOps-Prinzipien zu helfen. Dabei geht es nicht nur um die technischen Aspekte der eingesetzten Tools, sondern auch um die Prozesse und die Kultur drumherum. In der Regel besteht ein solches Team sowohl aus Personen mit sehr technischem Hintergrund als auch aus Personen mit einem starken Fokus auf die Etablierung der DevOps-Kultur.

Idealerweise gibt es einige Mitarbeitende in der Organisation, die sich hierfür selbst melden, motiviert sind und vielleicht sogar schon einmal nach DevOps-Prinzipien gearbeitet haben. So haben diese Personen eine solide Basis, um die DevOps-Transformation in den Teams einzuleiten und zu begleiten.

Die Auswahl der Tools sollte ordentlich durchdacht werden, damit eine saubere und schlanke DevOps-Toolchain entstehen kann.

DevOps-Team-Topologie

Das DevOps-Transformationsteam ist eines der wenigen Teams, das ich tatsächlich »DevOps-Team« nennen würde. Das Team ist schließlich dazu da, DevOps innerhalb der Organisation umzusetzen und die Kultur einzuführen.

Viele Firmen nennen ihre Teams, die eigentlich nur aus Entwicklern bestehen, auch DevOps-Teams, weil am Ende des Entwicklungsprozesses Container aus einer CI-Pipeline fallen, die sie dann nicht mal selbst betreiben. Genau so etwas ist hier nicht gemeint!

© Rheinwerk Verlag, Bonn 2024

Diversität ist gut

Bei der Formung solcher DevOps-Teams – es können mehrere sein – sollte idealerweise eine gewisse Diversität gegeben sein. Denn ich kann nur immer wieder wiederholen: DevOps ist in erste Linie eine Kultur und Arbeitsweise, und da sind die Menschen wichtig.

Bei der Diversität geht es nicht nur um Geschlechter oder Nationalitäten, sondern auch um verschiedene Alters- und Erfahrungsstufen und Jobrollen. Auch generell kann ein diverser beruflicher Hintergrund – mit tiefen technischen Aufgaben und eben nicht ganz so technischem Aufgabenfeld – hilfreich sein. Denn so können verschiedene zwischenmenschliche Probleme nachvollziehbarer gemacht werden, als wenn das Team eine homogene Gruppe ist.

Mit dem Blick auf den technischen Hintergrund werden umfassende Kenntnisse benötigt, um effizient den Teams bei der Transformation helfen zu können. Es sind also gegebenenfalls Trainings und Schulungen nicht zu vernachlässigen. Hier wird viel Wissen benötigt, da CI, CD, Infrastruktur und auch Monitoring schon große Themen für sich sind.

Das Ziel des Teams ist, sich selbst wieder abzuschaffen. Nur dann war es erfolgreich. Das heißt wiederum nicht, dass die Teammitglieder danach nicht mehr gebraucht werden: Ihr Kontakt sollte auch zukünftig eng sein, um Änderungen in der Toolchain oder Probleme ansprechen zu können.

Die konkreten Aufgaben des DevOps-Teams variieren über die Zeit und sind stark davon abhängig, wie fähig die zu unterstützenden Teams letztlich sind, um die kulturellen und technischen Aspekte in der Praxis umzusetzen. Idealerweise bringt das DevOps-Transformationsteam möglichst viele der hier genannten Punkte den Teams bei. Dazu sind teamübergreifende Trainings und Schulungen üblich, sodass hier im Idealfall ein Wissenstransfer stattfinden sollte sowie auch Hilfe zur Eigeninitiative gegeben wird.

Nach den reinen Schulungen sollte stets die Implementierung aktiv begleitet werden, damit die Mitglieder des DevOps-Teams bei Problemen als kompetente Ansprechpartner zur Verfügung stehen können. Auch hier wird eine gewisse Zeit gebraucht, um Vertrauen aufzubauen. Das Profil des DevOps-Teams sollte also nicht nur rein technisch sein, sondern auch zwischenmenschlich sollte ein gewisses Fingerspitzengefühl vorhanden sein, um konstruktiv mit Konflikten umgehen zu können.

Zusammenfassung des dritten Schritts

- DevOps-Transformationsteam einführen
- DevOps-Team auf die Teams verteilen, um diese bei der Umsetzung der Kultur und Technik zu unterstützen
- Schulungen und Trainings durchführen
- Begleitung bei der Implementierung

Persönliches Exemplar für Thomas Bartholomäus

Schritt 4: Mit dem ersten Value Stream starten

Nachdem das DevOps-Team gebildet und startklar ist, wird es Zeit, mit der tatsächlichen Umsetzung der DevOps-Transformation zu starten. Wichtig dafür ist, dass mit einem Bereich begonnen wird, in dem die beteiligten Teams für die Veränderung brennen. Je stärker die beteiligten Personen motiviert sind, desto wahrscheinlicher ist eine erfolgreiche Transformation, da durch die intrinsische Motivation viel in Bewegung kommen wird. Wenn Sie die vorherigen Schritte sinnvoll durchgeführt haben, ist der Grundstein gelegt, um die intrinsische Motivation der Teammitglieder zu steigern.

Aber Achtung: Das kann auch nach hinten losgehen! Hier muss zusätzlich eine Risikoabschätzung in Richtung *Business Continuity* getätigt werden. Motivierte Mitarbeitende sind essenziell, aber wenn die Transformation früh scheitert, da ein Business-kritischer Value Stream verwendet wird, dann wird die DevOps-Idee womöglich nicht lange im Unternehmen bestehen.

Die Teamstrukturierung sollte dann umgestellt werden. Darauf werde ich nicht tiefer eingehen, schließlich war es bereits immer wieder Thema. Je nach Firmengröße ist es sinnvoll, ein cross-funktionales Team aufzubauen oder, weil es sonst vielleicht nicht anders geht, gleich mehrere Teams.

Allerdings sollten nicht gleich alle verschiedenen Rollen gleichzeitig gestartet werden, denn das ist nicht zwangsläufig sinnvoll. Ich bin ein großer Freund davon, die Zusammenführung in mehreren Iterationen durchzuführen und nicht alles auf einem Schlag. Der Hauptgrund ist, dass die Umstellung dann deutlich weniger disruptiv ist und bei Problemen schneller gegengelenkt werden kann. Außerdem muss schließlich das Tagesgeschäft weiterlaufen.

Es gibt allerdings noch einen Faktor, den man bei der Bildung des ersten Teams beachten sollte. Motivierte Personen sind zwar ein wichtiger Faktor, allerdings sollte man auch einen *Value Stream* auswählen, der für die technischen Aspekte der DevOps-Transformation geeignet ist.

Prädestiniert sind dafür in der Regel junge Projekte, denn diese setzen häufig schon auf ein modernes Sourcecode-Management und haben sogar Pipelines, die für CI definiert sind. Vielleicht sind sogar schon erste Vorbereitungen getätigt worden, die das Deployment vereinfachen. Gut eignet sich etwa eine Webanwendung, die sich stetig aktualisieren lässt und vielleicht auch einmal ein wenig Downtime verzeiht.

Solche Projekte eignen sich für den Start besser als uralte Anwendungen, die auf exotischer Hardware oder Industriegeräten laufen müssen. Wenn es dann noch spezielle Anforderungen an die Verfügbarkeit gibt oder komplizierte Compliance-Regeln eingehalten werden müssen, ist das Risiko ziemlich groß, dass schon die ersten Gehversuche scheitern.

© Rheinwerk Verlag, Bonn 2024

Ist erst mal die erste Anwendung ausgewählt und das Team entsprechend aufgebaut, dann sollten Sie langsam anfangen, die verschiedenen Aspekte umzusetzen. Wichtig ist, dass alle Aspekte des CALMS-Modells etwa gleichzeitig etabliert werden. Häufigster Fehler ist, dass sich überwiegend auf das **A** von CALMS konzentriert wird, nämlich auf die Automatisierung. Mittlerweile sollte klar sein, dass DevOps mehr als nur eine CI/CD-Pipeline ist.

Wenn man den Verlauf des Buches anschaut, dann kann es als erster Schritt sinnvoll sein, das Sourcecode-Management gerade zu ziehen. Zuerst sollten also die Repositorys der verschiedenen Teams geöffnet und – idealerweise – zusammengeführt werden.

Das hat direkte Auswirkungen auf die Automatisierungen, die bisher genutzt wurden, denn diese sind meist mit den Repositorys verdrahtet, auch wenn diese vielleicht getrennt voneinander laufen. Gleichzeitig müssen Sie sich also um den Build-Prozess kümmern, damit die Arbeitsabläufe im Git dazu passen.

Denn diese Teile gehen Hand in Hand: Die Einführung von Code-Review ist etwa erst dann sinnvoll, wenn es auch eine CI-Pipeline gibt, die das Projekt einmal baut, auch wenn sonst noch keine Tests laufen. Denn Code-Review ohne frühes Feedback durch ein CI-System ist zwar möglich, allerdings etwas mühseliger. Vergessen Sie nicht: Hier arbeiten immer noch Menschen, und die sehen, wenn man es unklug anstellt, zunächst keine Vorteile und winken erst einmal alles durch. Das sollte man durch die richtige Vorgehensweise tunlichst vermeiden!

Nachdem der Build-Prozess lauffähig ist, sollten auch die Tests so weit adaptiert und automatisiert werden. Auch hier gilt: Führen Sie die Tests stetig ein und nicht alle auf einmal. Die Tests sollten erst in der Pipeline landen, wenn sie auch erfolgreich durchlaufen, sonst provoziert man nur weitere Blockaden. Das gilt auch für die Einbindung von Security-Scannern.

Die Einbindung des Ops-Teams sollte meiner Erfahrung nach erst dann erfolgen, wenn die vorherigen Stages des DevOps-Lifecycles gut umgesetzt und etabliert sind und die Zusammenarbeit funktioniert. Dazu gehören eben nicht nur die technischen Aspekte, sondern auch die angepasste, offenere Fehlerkultur. Wenn das verschlafen wird, dann sind wieder zu viele Baustellen gleichzeitig offen.

Continuous Delivery einzuführen ist ebenfalls kein einfacher Schritt. Nicht nur kulturell, sondern auch technisch. Der erste Schritt sollte sein, eine Staging-Umgebung aufzubauen, auf die automatisiert deployt werden kann.

Bis man da hinkommt, dauert es meist, da viele technische Grundlagen geschaffen werden müssen. Ein ungefährlicher Schritt zu Beginn ist, wenn mit den Review-Umgebungen begonnen wird. Dann geht es weiter zu der Staging-Umgebung, auf die man zu Beginn ohnehin nicht jeden Tag ausrollt. Schrittweise können dann die Abstände zwischen den Deployments verringert werden. Gleiches gilt anschließend auch für die Deployments auf die Produktivumgebung.

Wichtig für alle Teile ist: Vertrauen schaffen! Auf allen Ebenen.

Persönliches Exemplar für Thomas Bartholomäus

[!]

Zusammenfassung des vierten Schritts

- geeigneten Value Stream finden
- Umstellung der Teamstruktur
- Sourcecode-Management gerade ziehen
- Continuous Integration mit Code-Review einführen
- erste Schritte in Richtung Continuous Delivery auf Staging-Umgebungen

Schritt 5: Messen und iterieren

Der vierte und der fünfte Schritt müssen eigentlich zusammen betrachtet werden: Denn Iterationen gelten nicht nur innerhalb der jeweiligen Projekte, sondern auch innerhalb der Organisationen. Die im Buch angesprochenen Maßnahmen wie Retrospektiven, Post-Mortems und andere Möglichkeiten zum offenen Austausch, was gut und was schlecht läuft, sollten regelmäßig wiederholt werden, um aus Fehlern zu lernen und Verbesserungen zu implementieren.

Im vorherigen Schritt erwähnte ich bereits, dass Continuous Delivery eher relativ spät eingeführt werden sollte. Das gilt auch für viele weitere technische Aspekte, die in Abschnitt 8.4 genannt werden, wie Feature-Flags und A/B-Tests. Diese lassen sich erst sinnvoll umsetzen, wenn die CI/CD-Pipeline erfolgreich umgesetzt wurde. Daher wird das wohl eher ein Thema in einer späteren Iteration.

Security ist ebenfalls ein wichtiges Thema, aber erfahrungsgemäß wird es auch hier etwas länger dauern, bis das Security-Team in die laufende Transformation eingebunden wird, um auch da die Silos zu durchbrechen. Hierbei ist es primär wichtig, dass man nicht zu viel blockiert, denn sonst staut sich wieder zu viel auf, und das führt zu Frust. Die allgemeine Sicherheit sollte man allerdings trotzdem nicht vernachlässigen.

Nachdem man einige Iterationen durch die DevOps-Schleife vollzogen hat, lernen alle Teams immer wieder neue Dinge dazu. Diese Lernkultur und die Arbeitsstruktur, die geschaffen worden sein muss, sollte langfristig weiter existieren, damit nicht erneut Silos aufgebaut werden.

[!]

Zusammenfassung des fünften Schritts

- Continuous Delivery durchgehend einführen, sofern das möglich ist
- die Möglichkeit und das Vertrauen haben, um Feature-Flags nutzen und A/B-Tests durchführen zu können
- Iterationen über den ganzen Lifecycle und stets Verbesserungen implementieren
- den Erfolg mit Metriken messbar machen

© Rheinwerk Verlag, Bonn 2024

Reflexion

Die DevOps-Transformation erfolgreich durchzuführen, ist wirklich nicht einfach. Fassen Sie die vorgestellte Vorgehensweise nicht als abzuarbeitende Liste auf, denn dafür sind viele Firmen viel zu unterschiedlich.

Beachten Sie, dass viele Aspekte zudem keine Schritte sind, die irgendwann »einfach« fertig sind. Sie müssen langfristig gelebt werden. Wenn etwas nicht so funktioniert, wie Sie erwartet haben, versuchen Sie, die Probelme zu verstehen, und nehmen Sie Anpassungen vor.

12.2 Mit DORA-Metriken den DevOps-Erfolg messbar machen

In diesem Buch lernen Sie viele verschiedene Aspekte der DevOps-Idee kennen. Die Frage ist nun, wie man den Erfolg in der Organisation messen kann, denn die Bewertung des Gesamtbilds ist eine Herausforderung bei der DevOps-Transformation. Aber auch dafür gibt es diverse Metriken.

Maßgeblich für die Erfolgsmessung sind die *DORA-Metriken*. DORA steht für »DevOps Reasearch and Assessment«; das ist eine Organisation, die mittlerweile zu Google gehört. Diese Organisation stellt jedes Jahr den »State of DevOps Report« vor, in dem die aktuellen Entwicklungen und Untersuchungen von DevOps in den Unternehmen betrachtet werden. Ein Blick lohnt sich definitiv. DORA hat im Rahmen ihrer Untersuchungen insgesamt vier Metriken entwickelt, die daher auch als DORA4-Metriken bezeichnet werden. Bei den vier Metriken handelt es sich um:

1. Deployment Frequency
2. Lead Time
3. Change Failure Rate
4. Time to Restore Service

Diese Metriken schauen besonders auf die Performance der Software-Delivery. Aber allein auf diese zu blicken, reicht nicht, denn es gibt viel Kontext, der beachtet werden muss. Ein Irrglaube wäre, dass sich rein anhand dieser Metriken beurteilen lässt, wie gut DevOps – als Kultur! – umgesetzt wird. Vor allem in Deutschland gibt es sehr viel Industrie, deren Software sehr nah an der Hardware ist, wodurch sich solche Metriken nur bedingt eignen. Letztlich bilden sie *einen* Aspekt ab und der Kontext muss betrachtet werden.

Neben den genannten vier Metriken, auf die ich im Folgenden noch näher eingehe, gibt es noch eine weitere Metrik, die das Bild mit dem Blick auf *Operational Performance* vervollständigt:

5. Reliability

Kontext

Es kommt ganz auf den Kontext der Anwendung, der Teams und der Industrie an, in der Sie arbeiten. Nicht jede Firma und nicht jedes Team kann regelmäßig und in sehr kurzen Zeiträumen deployen – hauptsächlich eignet sich dieser Ansatz für Software, die im Web läuft und bei der Sie Kontrolle über die Plattform haben. Aber auch dann müssen Sie genau hinschauen, wie gut das Ergebnis für die Endnutzer ist: Wird eine stetige Änderung der Anwendung überhaupt gewünscht? Gibt es Feedback, das Aktualisierungen fordert, oder sind die Kunden eher an Stabilität interessiert? Ist vielleicht sogar die Verfügbarkeit für die Benutzer eingeschränkt? Werden vielleicht nur halbgare Features ausgerollt, was den Nutzern gar nicht hilft?

Auch das Ziel des Deployments ist essenziell dabei zu betrachten: Nicht jede Organisation entwickelt eine Webanwendung, die einfach in einer Cloud deployt werden kann. Wenn Sie in einem Industriebetrieb sehr nah an der Hardware arbeiten, also *Embedded Software Development* betreiben, können Sie niemals so oft und so schnell Änderungen durchführen wie an einem Online-Shop oder einer App. Hier gelten zwangsläufig andere Maßstäbe. Und dass Projekte, die im Wartungsmodus laufen und keine größere Weiterentwicklung mehr erhalten, ebenfalls anderes bewertet werden müssen, liegt auf der Hand.

Bis zu einem gewissen Grad können die genannten Metriken jedoch auch helfen, die Arbeit in »traditionellen« Industrien zu bewerten. Aber nicht jede Metrik ist aussagekräftig, nicht jedes Ergebnis direkt vergleichbar. Je nach Branche gibt es zudem noch konkrete Regularien, die auch eingehalten werden müssen.

So kann man zwar regelmäßig Software auf Autos deployen – man schaue sich da nur Tesla an –, aber ohne den genauen Kontext sagt das nicht viel aus: Geht es bloß um eine Änderung am Infotainment-System oder wird das Steuergerät der Bremsen umprogrammiert? Die Auswirkungen bei Fehlern sind völlig anders, dementsprechend lassen sich auch die Metriken nicht gleichermaßen anwenden.

12.2.1 DORA-Metrik 1: Deployment Frequency

Bei der *Deployment Frequency* geht es darum, wie oft ein Deployment ausgeführt wird. Besonders wichtig sind die Deployments auf die Produktivumgebung. Je höher dieser Wert ist, desto besser ist es unter DevOps-Gesichtspunkten. Komplett allein sollte man diese Metrik allerdings nicht betrachten, sondern immer zusammen mit den übrigen drei Metriken.

Je geringer die Frequenz der Deployments ist, desto weiter weg ist die Organisation von DevOps-Prinzipien. Der Grund sollte mittlerweile klar sein: Der erste Weg der Drei Wege besagt, dass die Verbesserungen stetig und regelmäßig zu den Endnutzern laufen sollen, und das geht nur mit einer hohen Frequenz von Deployments.

© Rheinwerk Verlag, Bonn 2024

Je höher der Automatisierungsgrad und je schlanker der gesamte Entwicklungszyklus ist, desto eher ist es möglich, eine hohe Anzahl an Deployments auszurollen. Ein Nebeneffekt davon ist, dass bei einer hohen Deploymentfrequenz die Teams und die Organisation hohes Vertrauen in die Software, das Team und die Prozesse besitzen. Zwangsläufig traut sich das Team oft, Änderungen auszurollen, wenn auch wirklich die Qualität auf allen Ebenen stimmt.

So viel zur Theorie. Aber wie sieht es jetzt in der Praxis aus? Welche Werte sind gut, welche sind schlecht und wo liegt die Mitte? Die Zahlen, sowohl in dieser Metrik als auch in den folgenden Metriken, sind dem »State of DevOps Report 2022« entnommen (*https://cloud.google.com/devops/state-of-devops*).

Wenig überraschend ist, dass ein Deployment pro Monat bis zu einem Deployment alle sechs Monate als niedrige Deployment-Frequenz angesehen wird. So häufig deployen Organisationen, die nicht nach DevOps-Prinzipien arbeiten.

Die Mitte liegt bei mindestens einem Deployment pro Woche bis zu einmal im Monat. Wenn man diese Häufigkeit erreicht hat, dann befindet man sich auf gutem Wege zu einer hohen Deployment Frequency, bei der man je nach Bedarf deployt, was mehrmals pro Tag sein kann.

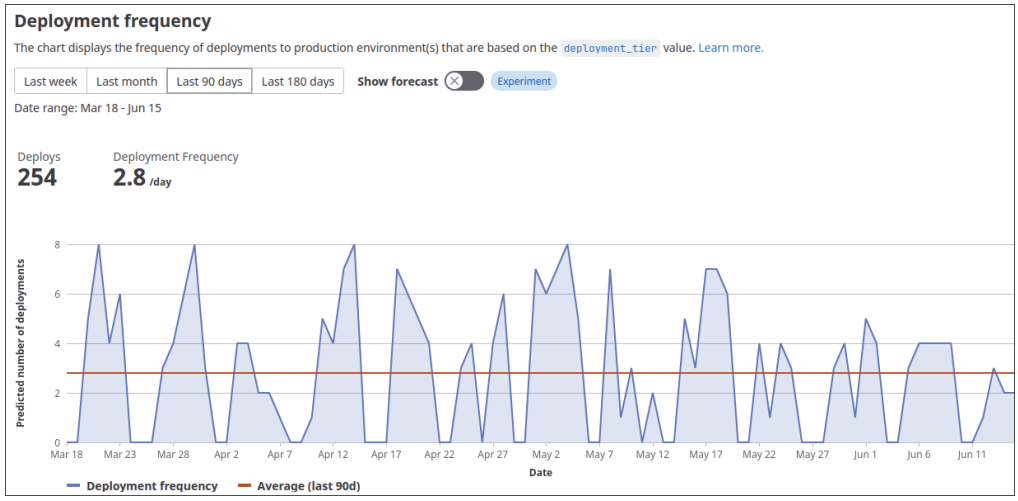

Abbildung 12.3 Die »Deployment Frequency« in der Praxis: Nicht jeden Tag wird deployt.

Dabei kommt es darauf an, dass wirklich an jedem Tag Deployments ausgerollt werden. Abbildung 12.3 gibt die Frequenz der letzten 90 Tage wieder, in der es im Mittel eine Deployment-Frequency von 2,8 Tagen gab. Die tatsächliche Frequenz an Arbeitstagen war allerdings wesentlich höher, da meist zwischen vier und acht Deployments durchgeführt wurden. Ab Freitagnachmittag wurde etwa nicht mehr deployt, denn am Wochenende wurde nicht gearbeitet.

12.2.2 DORA-Metrik 2: Lead Time

Hand in Hand mit einer hohen Deployment-Frequenz geht die zweite Metrik: die *Lead Time*. Die Lead Time ist die Durchlaufzeit, also die Zeit, die es braucht, bis eine Änderung ausgerollt wird. Sie beginnt mit dem Erstellen des Commits, also nach dem Abschluss der Implementierung der Änderung, und endet, wenn diese Änderung auf Produktivsystemen ausgerollt ist. Hier ist zwar eine nahe Verwandtschaft zur Deployment-Frequenz gegeben, allerdings können viele Deployments auch mit vielen älteren Änderungen gemacht werden.

Die Lead Time sollte möglichst kurz sein. Auch hier geht es um den ersten der Drei Wege: die Änderung zeitnah den Endbenutzern zur Verfügung zu stellen.

Eine Lead Time, die zwischen einem Monat und sechs Monaten liegt, ist kein guter Wert. Besser sieht es hingegen aus, wenn es zwischen einer Woche und einem Monat dauert, bis Änderungen ausgerollt werden. Ausgezeichnet unterwegs ist man, wenn die Lead Time zwischen einem Tag und einer Woche liegt, denn so kommen die Änderungen auch wirklich zügig an.

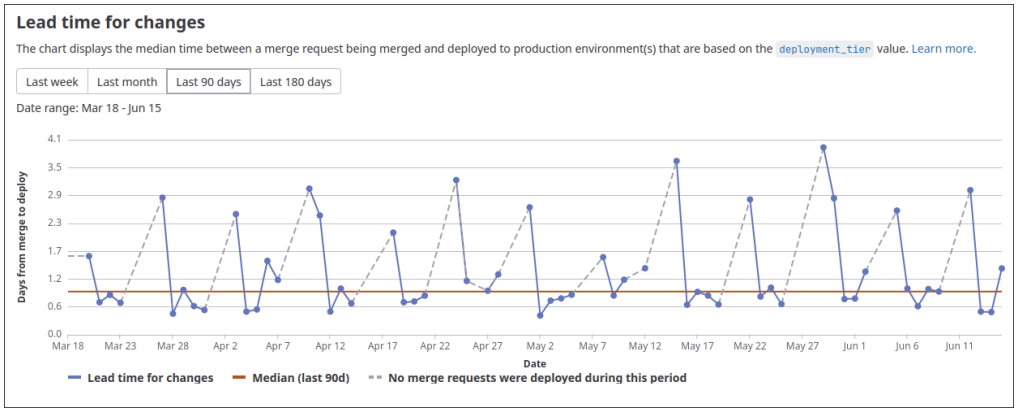

Abbildung 12.4 Auch bei der »Lead Time« sind große Unterschiede zwischen Wochentagen und Wochenenden üblich.

Wie auch bei der Deployment-Frequenz ist die Lead Time am Wochenende häufig deutlich länger, wie Abbildung 12.4 zeigt. Wichtig sind auch hier die Tendenz über einen längeren Zeitraum sowie der Median.

12.2.3 DORA-Metrik 3: Change Failure Rate

Die dritte Metrik schaut im Gegensatz zu den ersten beiden Metriken weniger auf die Häufigkeit und Schnelligkeit der Änderungen, sondern darauf, wie oft Fehler passieren. Ein schnelles und häufiges Ausrollen von Änderungen bringt nämlich für die Nutzer am Ende nicht viel, wenn ständig Fehler passieren.

© Rheinwerk Verlag, Bonn 2024

Die *Change Failure Rate* gibt an, wie viele Fehler pro ausgerollter Änderung auftreten. Je geringer dieser Wert ist, desto besser. Auch hier geht es um die Produktivumgebung, um damit auch die Qualität und Produktivität des Teams zu beurteilen.

Wenn dieser Wert in der Praxis bei 0 bis 15 % liegt, dann ist das schon mal sehr gut. Je näher Sie an 0 kommen, desto besser. Ein mittlerer Wert liegt bei 16 bis 30 %; alles darunter kann man wohl als schlechten Wert ansehen. Wichtig ist aber nochmals: Die Werte sind immer nur in Kombination mit den anderen Metriken sinnvoll!

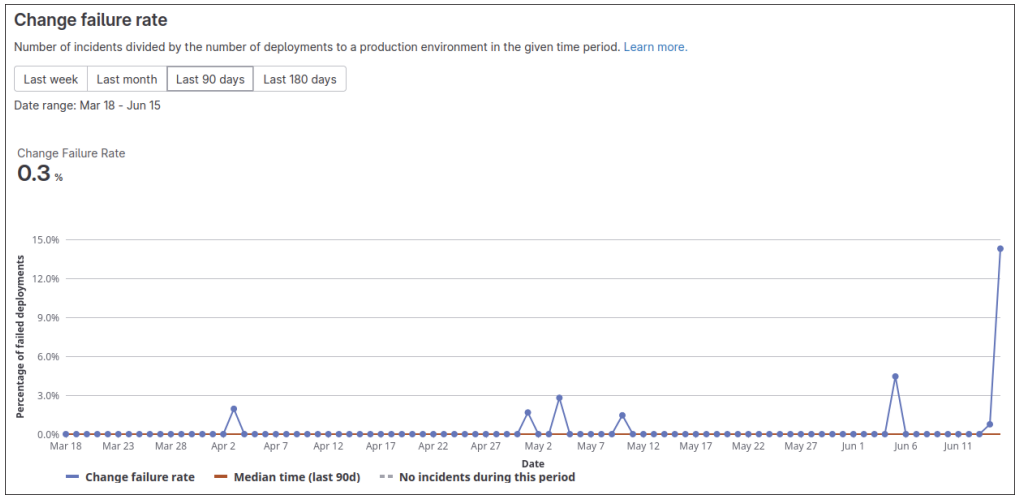

Abbildung 12.5 In der Praxis sollte die »Change Failure Rate« möglichst niedrig sein, es gibt aber immer wieder Ausnahmen.

In Abbildung 12.5 sieht man etwa, wie die Change Failure Rate auch aussehen kann: Meistens gibt es keine Probleme, selten gibt es sehr weniger Fehler – bis es dann mal ein größeres Problem gibt, das die Failure Rate in die Höhe treibt. Diese Metrik sagt daher relativ wenig über die eigentliche Qualität aus, denn einige größere Probleme können schlimmer sein als mehrere kleinere über die Zeit verteilt. Daher gilt auch hier: Die Tendenz ist wichtig, damit Sie frühzeitig gegensteuern können, wenn sich Probleme häufen.

12.2.4 DORA-Metrik 4: Time to Restore Service

Richtig schlimme Fehler führen dazu, dass es zu einer Downtime kommt. Die interessante Frage ist dann, wie schnell eine solche Downtime wieder zu Ende ist.

Genau das gibt diese vierte Metrik an: Wie lange dauert es, bis der Service im Falle eines Ausfalls wieder zur Verfügung steht? Je geringer der Wert ist, desto besser. Damit soll allerdings nicht ausgesagt werden, dass der komplette Dienst offline ist. Incidents, die etwa zu Performance-Einbußen führen oder nur eine kleinere Gruppe betreffen, fallen ebenfalls unter diese Metrik.

Persönliches Exemplar für Thomas Bartholomäus

Idealerweise ist innerhalb von wenigen Stunden – also noch am selben Tag – alles wieder vollständig lauffähig. Das wird als guter Wert angesehen. Alles, was über einen Tag hinaus bis zu einer Woche geht, ist statistisch noch okay. Wirklich schlecht sieht es wohl aus, wenn es länger als eine Woche dauert, bis die Dienste wieder vollständig laufen.

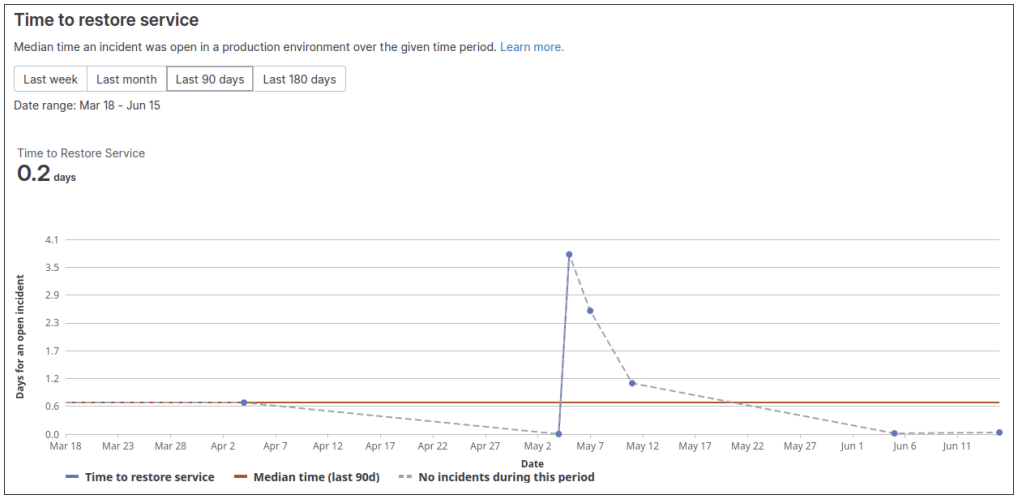

Abbildung 12.6 Idealerweise gibt es wenn überhaupt, dann nur kurze Downtimes, die für die »Time to Restore Service« relevant sind.

Wie auch bei den vorherigen Beispielen gibt es – hoffentlich – nicht so häufig Fehler, dass der Dienst nur eingeschränkt läuft. Nicht jeder Incident ist schnell gelöst, aber jeder Incident verändert den Medianwert, wie lange es dauert, bis der Dienst wieder läuft. Auch hier gilt: Betrachten Sie die langfristigen Tendenzen, lernen Sie aus Fehlern und leiten Sie gegebenenfalls Maßnahmen ein.

12.2.5 DORA-Metrik 5: Operational Performance: Reliability

Die fünfte Metrik ist vergleichsweise neu und wurde 2021 eingeführt. In vielen Quellen liest man daher weiterhin von den DORA4-Metriken, also wundern Sie sich nicht, heutzutage sind es fünf Metriken.

In der fünften Metrik wird die *Reliability* betrachtet. Es geht also darum, wie zuverlässig der Dienst für die Nutzer funktioniert und ob er den Erwartungen gerecht wird. Im »State of DevOps Report 2022« schreibt die DORA-Organisation selbst, dass der vorherige Fokus primär auf der *Verfügbarkeit* lag – und nicht auf der *Zuverlässigkeit*. Diese beiden Begriffe beschreiben aber zwei unterschiedliche Aspekte und sagen nur auf den ersten Blick das Gleiche aus.

Wer jetzt verwirrt ist und keinen direkten Unterschied erkennt: Die *Verfügbarkeit* sagt lediglich aus, ob ein Dienst läuft oder nicht. Bei einer Availability, also Verfügbar-

© Rheinwerk Verlag, Bonn 2024

keit, eines Dienstes von 99,99 % ist eine Downtime von 52,60 Minuten pro Jahr erlaubt.

Damit ist allerdings nicht ausgesagt, ob der Dienst in den restlichen 525.547,64 Minuten des Jahres ordentlich arbeitet. Ein Dienst, der zwar immer »da« ist, aber unbenutzbar langsam ist oder bei jedem zweiten Klick Fehler produziert, ist für die Nutzer nicht zu gebrauchen. Hier hängt die Bewertung und Messung an definierten Performance-Standards.

Diese Metrik hat entsprechend nichts mit der Software-Delivery selbst zu tun, weshalb sie getrennt von den vier bisherigen Metriken betrachtet wird. Bewertet wird die *Reliability* anhand von Verfügbarkeit, Latenz, Performance und Skalierbarkeit.

12.2.6 Erkenntnisse aus dem »State of DevOps Report«

Wenn Sie tiefer in die Zahlen und Fakten rund um die aktuellen DevOps-Entwicklungen eintauchen möchten, sollten Sie den State of DevOps Report aufmerksam lesen. Einmal im Jahr führt das Team große Umfragen durch. Die Ergebnisse werden jedes Jahr im September zusammenfassend veröffentlicht.

Der Download ist zwar kostenlos, allerdings nicht frei im klassischen Sinne, da für den Download eigentlich die eigenen Kontaktdaten, inklusive Mobilfunknummer, Arbeitgeber und Jobtitel angegeben werden müssen. Praktischerweise findet man allerdings den Report auch ohne diese Angaben, wenn man bei Google nach »DORA State of DevOps Report 2023/24/25 pdf« sucht.

Im Report wird stets auf die neuesten Entwicklungen, die neuesten Zahlen und die Veränderungen eingegangen. So kann man gut und einfach überblicken, wie sich die Zahlen über die Jahre verändert haben und wohin sich die Industrie entwickelt – zumindest bei denen, die an der Umfrage teilgenommen haben. (Sie können gerne auch selbst teilnehmen.)

Der Knackpunkt ist allerdings, wie so häufig bei solchen Umfragen, die Streuung der Daten über Länder und Branchen hinweg. Wenig überraschend liegt ein starker Fokus auf den USA, die zumindest im Report von 2022 mit 29 % vertreten sind, gefolgt von Indien mit 21 %. Immerhin stammen noch 4 % der Teilnehmer aus Deutschland. Eine Überbewertung bestimmter Industrien und Firmengrößen ist ebenfalls zu sehen.

Aus dem Report gehen viele Punkte hervor, die ziemlich offensichtlich sind. Sie geben aber einen guten, kompakten Einblick, wie andere Firmen arbeiten, obwohl nicht nach einzelnen Industrien gefiltert wird. Viele Teilnehmer setzen auf die Cloud und nutzen Tools zur Automatisierung der Deployments. Außerdem lässt sich sagen: Je mehr Cloud und je mehr Automatisierung im Spiel sind, desto erfolgreicher verläuft auf technischer Ebene die Umsetzung der DevOps-Idee.

Persönliches Exemplar für Thomas Bartholomäus

Aber auch andere Themen werden betrachtet, die nicht zu vernachlässigen sind. Im Report von 2022 steht beispielsweise, dass eine lose gekoppelte IT-Architektur bei einer DevOps-Transformation zum Burnout im Team beiträgt. Im Report werden genauere Hintergründe der Untersuchung behandelt.

Reflexion: DORA

Die DORA-Metriken geben einen guten – aber nicht alleinigen – Überblick über den Erfolg oder Misserfolg von DevOps in der Organisation. Mithilfe dieser Metrik kann gemessen und verglichen werden, wie gut DevOps-Prinzipien umgesetzt werden.

Eines dürfen Sie allerdings nicht außer Acht lassen: Es handelt sich um Metriken, die die Deployment-Häufigkeit und -Fehler beschreiben. Sie geben keinen Einblick in die tatsächliche Zufriedenheit des Teams oder der Endnutzer. Darum sollte es bei der DevOps-Transformation aber hauptsächlich gehen: Die Menschen stehen im Vordergrund, erst danach geht es um die Prozesse und die Tools. Die DORA-Metriken behandeln jedoch die Prozesse, schließlich lassen sich diese gut messen.

Um die DORA-Metriken zu erfassen, ist eine recht komplexe Tool-Landschaft notwendig, die die unterschiedlichen Werte sammelt. Das ist oftmals kontraproduktiv, denn je mehr Tools man nutzt, desto schwieriger wird es, alle Daten zuverlässig zu erfassen, da jeder Übergang zwischen den Tools nicht nur Zeit kostet, sondern auch den Ablauf komplexer macht. Abhilfe können DevOps-Plattformen schaffen (siehe Kapitel 13), die das Ganze vereinfachen sollen. Wenn Sie etwa auf GitLab setzen und dort die relevanten Funktionen nutzen, dann werden die DORA-Metriken automatisch erfasst und dargestellt. Dafür benötigen Sie allerdings die *GitLab Enterprise Edition* mit der Ultimate-Lizenz.

Sinnvoll nutzen Sie die DORA-Metriken, indem Sie sie über einen Zeitraum anschauen, um ein Gefühl dafür zu bekommen, wie gut die Prozesse und die technische Umsetzung aussehen. Probieren Sie diverse Änderungen aus, und schauen Sie, wie es sich auf die Metriken auswirkt. Steuern Sie nach, wenn Sie sich in die falsche Richtung bewegen. Versteifen Sie sich allerdings auch nicht zu sehr auf die Metriken, ohne den Kontext und die Hintergründe zu betrachten: Eine hohe Deployment-Frequency ist zwar gut und wichtig; wenn sich im Gegenzug jedoch niemand mehr traut, größere Änderungen auszurollen, passt es auch wieder nicht.

12.3 Value Stream Mapping

Bisher ging es in diesem Kapitel vor allem um die kulturelle Adaption und den Einsatz von sinnvollen Metriken. Was allerdings noch fehlt, ist der Blick auf einen *Value Stream*. Denn idealerweise wird vor allem zu Beginn der DevOps-Transformation ein passender Value Stream gewählt und ein *Value Stream Mapping* durchgeführt.

© Rheinwerk Verlag, Bonn 2024

Aber was ist genau Value Stream Mapping? Dazu muss man sich zunächst anschauen, was ein Value Stream selbst genau ist. Ein Value Stream ist eine Abfolge von Aktivitäten, die in einer Organisation durchgeführt werden müssen, um eine Ware oder einen Service an die Endnutzer zu bringen.

Das hat eigentlich nichts mit DevOps zu tun und hat seine Ursprünge auch gar nicht in der IT. Das Konzept stammt nämlich schon aus den 90er-Jahren und ist durch das Buch »Lean Thinking« von James Womack und Daniel Jones populär geworden, das 1996 erschien.

Value Stream Mapping wird auf Deutsch auch *Wertstromanalyse* genannt. Da ich diesen Begriff allerdings scheußlich finde und man in der IT eigentlich stets von Value Stream Mapping spricht, bleibe ich beim englischen Begriff.

Ein Value Stream ist also eine ganze Kette an Prozessen und auch Personen, die beteiligt sind, um einen bestimmten Wert oder Service auszuliefern oder bereitzustellen. Das ist immer cross-funktional, da viele verschiedene Personen mit ihren verschiedenen Rollen beteiligt sind, wie die folgenden Beispiele auch zeigen. Für den Kunden oder Nutzer des Systems gibt es meist nur einen Value Stream, auch wenn es intern viele verschiedene Value-Streams gibt, die das Business stützen.

Prozesse sind Teil des Value Stream Mappings

Wichtig ist, dass es bei einem Value Stream Mapping nicht um die Visualisierung eines Prozesses geht. Prozesse sind Teil eines Value Streams, ein Value Stream besteht also aus vielen verschiedenen Prozessen.

Nachvollziehbar wird es, wenn man sich zwei Value-Streams anschaut, einmal nahezu völlig außerhalb der IT und einmal ein konkretes Beispiel von *schick-gekleidet.de*. Am einfachsten beginnen wir mit der Beschreibung eines Value Streams, die auch nicht technikaffine Personen verstehen. Als Beispiel dient hier dabei eine Pizzeria.

12.3.1 Der Value Stream einer Pizza

Der Value Stream für den Kunden ist, dass dieser in der Regel eine Bestellung aufgibt und die Pizza nach einer Weile erhält. Wenn man sich nun diesen Value Stream vollständig anschaut, besteht er für den Pizza-Bäcker aus vielen kleinen Schritten. So geht im ersten Schritt die Bestellung ein. Diese kann per Anruf erfolgen, persönlich vor Ort oder auch per Online-Bestellung.

Diese Bestellung wird dann bearbeitet: der Pizzateig mit dem gewünschten Belag belegt und dann in den Ofen geschoben, damit die Pizza gebacken wird. Nach dem Backvorgang wird die Pizza entweder direkt auf einem Teller serviert oder sie wird in einen Karton verpackt zwecks Auslieferung oder Abholung.

Persönliches Exemplar für Thomas Bartholomäus

Für den Kunden sind letztlich im Wesentlichen drei Punkte wichtig: Im ersten Schritt soll die Bestellung einfach vonstattengehen, denn niemand möchte eine Pizza per Fax bestellen, ausgenommen vielleicht deutsche Behörden. Die viel wichtigeren Punkte sind aber, dass die Pizza für den Verzehr zügig und warm ankommt und dass sie hervorragend schmeckt.

Bei einer kleinen Pizzeria, in der nur die Inhaber arbeiten, ist das häufig kein Problem, da der komplette Value Stream von ein oder zwei Personen abgefertigt wird, ohne Bottlenecks oder Verzögerungen. Spannend wird es, wenn die Arbeit skaliert werden muss: Zu den typischen Essenszeiten kommen deutlich mehr Bestellungen herein, per Online-Bestellung, per Telefon und auch vor Ort. Es gibt eine begrenzte Anzahl von Öfen, und auch der Belag für die Pizza muss regelmäßig vorbereitet werden, damit die Wartezeiten möglichst gering sind. Außerdem sind die Fahrzeuge und Fahrer für die Auslieferungen begrenzt. Bei einigen Bestellungen dauert es dann an vielen Stellen sehr lange: Zunächst dauert es lange, bis die Pizza überhaupt im Ofen ist, und dann dauert es noch, bis ein Fahrer verfügbar ist, um die Pizza auszuliefern, sodass sie häufig kalt beim Besteller ankommt.

Genau hier kann ein Value Stream Mapping sinnvoll sein. Es ist kein Tool und kein Prozess, sondern eine Methode, um den zuvor definierten und ausgewählten Value Stream zu visualisieren. Das Ziel ist es dabei, den Wertfluss eines Systems oder Prozesses zu identifizieren, um anschließend diesen ganzen Prozess optimieren zu können, damit nicht benötigte Schritte entfernt werden und der Ablauf somit verbessert werden kann.

12.3.2 Der Value Stream von schick-gekleidet.de

Auch bei *schick-gekleidet.de* gibt es für die Kunden einen wesentlichen Value Stream: Statt einer Pizza möchte man eine Bestellung für Kleidungsstücke aufgeben, die möglichst zeitnah geliefert werden sollen. Der Start ist demnach die Online-Bestellung, und das Ende des Value Streams ist die Auslieferung über einen Paketdienst.

Hier fällt Ihnen vielleicht schon auf, dass nicht alles vollständig in eigener Hand liegt, denn einen eigenen Lieferdienst kann sich außer vielleicht Amazon niemand leisten. Es gilt also, den Value Stream mitsamt externen Abhängigkeiten zu visualisieren.

Für *schick-gekleidet.de* heißt es, dass der Kunde eine Online-Bestellung aufgibt, indem er die Artikel in den Warenkorb legt und den Bestellvorgang abschließt. Je nach Bezahlart bezahlt der Kunde sofort oder per Rechnung erst nach ein paar Tagen. Hier kommt es also unter Umständen zu Wartezeiten.

Nachdem der Bezahlvorgang abgeschlossen wurde, läuft jemand im Lager los und packt die Ware ein, die anschließend an den Paketdienst übergeben wird, damit sie ausgeliefert werden kann.

© Rheinwerk Verlag, Bonn 2024

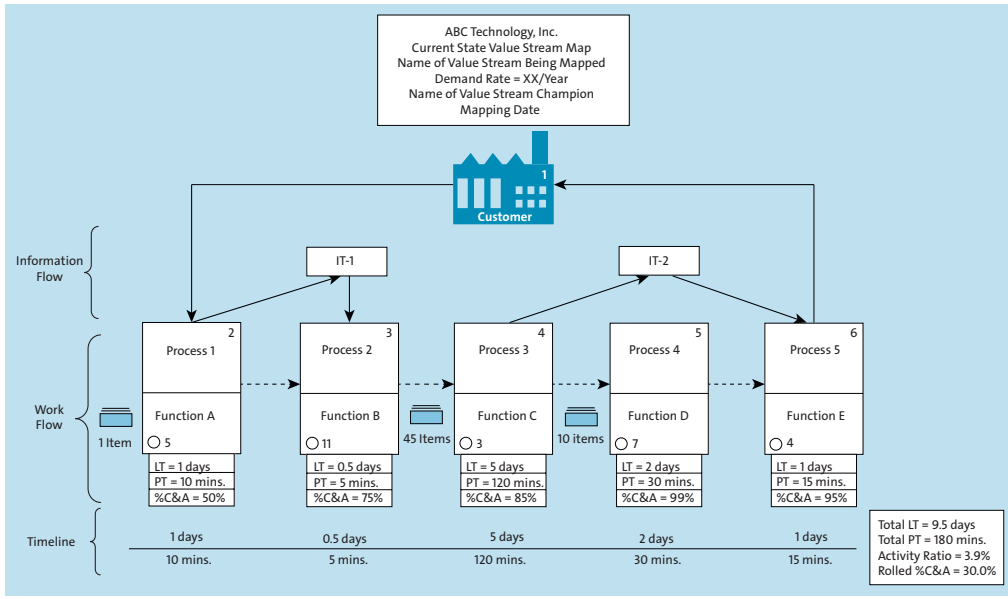

Abbildung 12.7 Ein Beispiel für ein Value Stream Mapping

Bei jedem Übergang können Wartezeiten und Bottlenecks entstehen: Der Kunde überweist die Rechnung nicht rechtzeitig, beim Bezahlvorgang geht etwas schief, das Lager ist unterbesetzt ... Je nach Bestelllage wird die Ware mal schneller oder langsamer im Lager eingesammelt, eingepackt und versandfertig gemacht. Da der Paketdienst die Pakete nur zu bestimmten Uhrzeiten abholt, muss auch dabei eine Koordination erfolgen, damit die Übergaben stimmen.

Schauen wir uns das Beispiel noch konkreter an: Das Bestellsystem liefert dem Lager für jede Bestellung eine Liste, die die Personen im Lager verwenden, um die Pakete zu packen. Problematisch ist, dass für jede Bestellung eine einzelne Liste generiert wird, sodass jeder Auftrag einzeln abgearbeitet wird. Im Rahmen des Value Stream Mappings fällt auf, dass es effizienter wäre, wenn die Laufwege optimiert werden: Anstatt jede Bestellung für sich zu sehen, wäre es besser, wenn ein Laufweg für mehrere Bestellungen berechnet würde. Damit lässt sich die Lead Time für den Prozess des Einpackens deutlich verkürzen, da mehrere Bestellungen parallelisiert betrachtet werden: Mit der gleichen Anzahl an Personen lassen sich mehr Bestellungen verarbeiten.

Ist das offensichtlich? Vielleicht. Meiner Erfahrung nach werden solche Ideen jedoch erst klar, wenn alle Personen im gleichen Raum sitzen und ihre Arbeitsweisen und Herausforderungen erklären.

Aber was hat das nun mit DevOps zu tun? Grundsätzlich in den hier gezeigten Beispielen gar nicht so viel. Es ging hier mehr um die Lean-Methode, die auch in der IT

angewandt werden kann. Nutzen Sie dieses Werkzeug, um den Value Stream innerhalb der Firma zu visualisieren und dann zu optimieren.

Ein solcher Value Stream kann etwa die Auslieferung eines neuen Features für den Online-Shop von *schick-gekleidet.de* sein. Wie der ganze schwerfällige Prozess bisher bei *schick-gekleidet.de* aussieht, erläutere ich zu Beginn der einzelnen Kapitel. Offensichtlich ist dabei, dass die Kollaboration zwischen den verschiedenen Teams gestärkt werden muss, um effizienter und schneller Mehrwert an den Kunden ausliefern zu können.

Dabei geht es um jeden Übergang: vom Kundenwunsch und der Anforderungsanalyse über die Umsetzung des Entwicklungsteam, die Tests des QA-Teams, die Übergabe an das Betriebsteam und schließlich an das Security-Team, das kurz vor dem Deployment die Handbremse ziehen kann.

Value Stream Mapping ist ein Werkzeug, um diese Silos zu durchbrechen und alle gemeinsam an einen Tisch zu holen, um das ganze Produkt und den ganzen Dienst im Blick zu haben. Es müssen alle Informationsflüsse zwischen den Teams betrachtet werden, daher werden beim Value Stream Mapping alle Stakeholder benötigt, die für den Value Stream relevant sind.

Für die größere Pizzeria sind diese Stakeholder

- der Inhaber,
- die Personen, die die Bestellungen annehmen,
- die Pizza-Bäcker
- sowie die Auslieferer.

Alle sind hier relevant. Bei der Auslieferung von neuen Features für den Online-Shop von *schick-gekleidet.de* sind dies dementsprechend nicht nur die Entwickler, sondern auch das Betriebsteam, das Security-Team, aber auch die Führungskräfte, die Entscheidungen treffen müssen. Es liegt an allen, den kompletten Value Stream zu verbessern.

12.3.3 Durchführung eines Value Stream Mappings

Ein Value Stream Mapping ist keine Aufgabe, die man mal eben zwischendurch erledigen kann. Es benötigt eine sorgfältige Vorbereitung, während der zunächst die nötigen Informationen zusammengetragen werden. Auch für das Mapping selbst sollten Sie einige Tage einplanen.

Value Stream Mapping ist kein Verfahren, das nur einmalig ausgeführt wird und für immer gültige Ergebnisse liefert. Es bietet sich an, die Analyse entweder regelmäßig oder zu bestimmten Anlässen zu wiederholen, um zu überprüfen, ob man noch optimal arbeitet.

© Rheinwerk Verlag, Bonn 2024

Ich möchte an dieser Stelle nur anreißen, welche Schritte für die Absolvierung eines Value Stream Mappings relevant sind. Eine komplette Beschreibung würde zu weit vom Thema dieses Buch wegführen und dessen Rahmen sprengen. Falls Sie sich für dieses Thema interessieren, empfehle ich das Buch »Value Stream Mapping« von Karen Martin und Mike Osterling.

1. **Vorbereitung**

   Vor dem eigentlichen Mapping wird ein *Value Stream Mapping Charter* (VSM-Charter) angelegt, in dem die ersten Planungen notiert werden.

   In diesem Meeting wird unter anderem definiert, welcher Value Stream behandelt werden soll, was der erste und der letzte Schritt des Value Streams ist und welche verantwortlichen Personen beteiligt sind. Ein Value Stream Mapping ohne eine Unterstützung von Führungskräften ist sinnlos; sparen Sie sich den Aufwand, wenn diese Personen keine Zeit oder Lust haben. Ohne Input und Unterstützung von oben kann nichts grundlegend verändert werden.

2. **Das VSM-Charter**

   Im VSM-Charter muss festgehalten werden, welche aktuellen Probleme existieren und mit welcher Metrik diese gemessen werden können. Beispielsweise heißt es für die Auslieferung neuer Features für den Online-Shop, dass die Lead Time die wichtigste Metrik ist und dass die Zeit halbiert werden soll.

Value Stream Mapping Charter

Scope		Accountable Parties		Logistics	
Value Stream		Executive Sponsor		Event Dates & Times	
Specific Conditions		Value Stream Champion			
Demand Rate		Facilitator		Base-camp Location	
Trigger					
First Step		Logistics Coordinator		Meals Provided	
Last Step					
Boundaries & Limitations		Briefing Attendees ** required *optional		Briefing Dates &Times	
Improvement Timeframe					

	Current State Problems & Business Needs
1	
2	
3	
4	
5	

	Measurable Target Condition
1	
2	
3	
4	
5	

	Benefits to Customers & Business
1	
2	
3	
4	
5	

	Relevant Data
1	
2	
3	
4	

	Mapping Team		
	Function	Name	Contact Information
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

	On-Call Support		
	Function	Name	Contact Information
1			
2			
3			
4			

	Agreement		
	Executive Sponsor	Value Stream Champion	Facilitator
1			
2	Signature:	Signature:	Signature:
3	Date:	Date:	Date:

Abbildung 12.8 Das Value Stream Mapping Charter wird verwendet, um das eigentliche Mapping zu planen. (Quelle: https://tkmg.com/wp-content/files/Value-Stream-Mapping-Charter-KMMO-v14.xlsx)

3. **Der Workshop**

   Erst nachdem das VSM-Charter vollständig ausgefüllt wurde, sollte das tatsächliche Mapping ausgeführt werden. Dort wird im ersten Schritt der Status quo visualisiert. Alle relevanten Stakeholder müssen sich dafür unter Umständen mehrere Tage Zeit nehmen, um voll konzentriert an einem Value-Stream-Mapping-Workshop teilzunehmen.

   In dem Workshop werden alle Personen interviewt, deren Gruppen an dem Value Stream beteiligt sind, um die einzelnen Prozesse aufzuschlüsseln. Für jeden Prozess gibt es mehrere Informationen und Metriken, die bekannt sein müssen. Für unser Beispiel sind das im Wesentlichen:

   – Lead Time
   – Process Time
   – %C/A (Percentage Complete and Accurate)
   – beteiligte Personen

   Die *Lead Time* beginnt, sobald Arbeit für eine beteiligte Person verfügbar ist, und endet, sobald ihr Teil an der Arbeit abgeschlossen ist und die Aufgabe an die nächste Person im Prozess weitergeht. Ein Teil davon ist die *Process Time*, was die tatsächliche Arbeit abdeckt. Die Lead Time ist also die Zeit für die tatsächliche Arbeit plus mögliche Wartezeiten. Das ist schon ein guter Indikator, um Optimierungspotenzial zu finden.

   %C/A steht für *Percentage Complete and Accurate*, was auf den ersten Blick ein wenig kryptisch wirkt. Hierbei handelt es sich um die Qualität der Arbeit, die in Prozent angegeben wird. Wenn fehlerfreie Arbeit zu Beginn des Prozesses ankommt, dann entspricht das 100 %.

   Häufig sind aber Nachfragen und Korrekturen notwendig, sodass immer wieder ein Schritt zurückgegangen werden muss. Das erhöht dann die Lead Time und ist ein weiterer Punkt, an dem signifikante Verbesserungen erzielt werden können – besonders dann, wenn es mehrere Personen betrifft, wenn also ein Bottleneck mehrere Prozesse blockiert.

   Diese Parameter sind für unser Beispiel relevant, aber es gibt selbstverständlich noch viele weitere Metriken und Informationen, die ich hier der Übersichtlichkeit und Einfachheit halber weggelassen habe. Herauszufinden, was für Ihren Value Stream relevant ist, ist eine zentrale Aufgabe in diesem Schritt.

4. **Visualisierung des Value Streams und Optimierung der Prozesse**

   Nachdem Sie alle Prozesse des Value Streams visualisiert haben und die genannten Metriken aufgeschlüsselt wurden, sollten die vollständige Lead Time, die Process Time, die gesamte %C/A sowie die Activity Ratio ausgerechnet werden. Diese sagen Ihnen, wie der aktuelle Status gerade ist.

© Rheinwerk Verlag, Bonn 2024

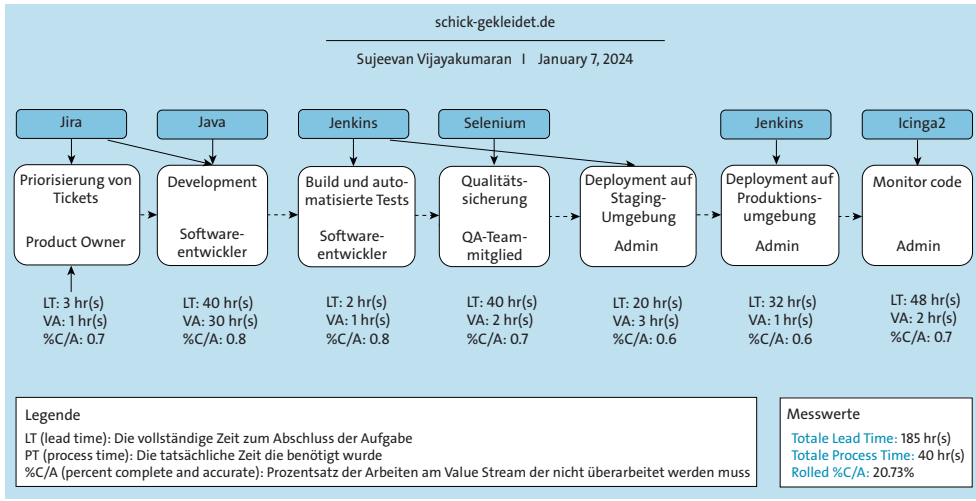

Abbildung 12.9 Ein Beispiel-VSM für die SDLC von schick-gekleidet.de

Ein wesentlicher Vorteil dieser Übung ist, dass über die Teamgrenzen hinweg alle relevanten Personen beteiligt werden und gemeinsam an der Verbesserung arbeiten. Ohne das Value Stream Mapping werden nur einzelne Prozesse betrachtet, was dazu führt, dass einzelne Prozessschritte optimiert werden, ungeachtet der Auswirkungen auf andere Prozesse.

Beim Mapping des Value Streams sollte kein Finger-Pointing betrieben werden, da dies nicht zielführend ist. Wenn Sie sich mit Ihren Kollegen Zeit nehmen, hilft die gemeinsame Arbeit dabei, Probleme auszuräumen und Missverständnisse zu klären.

5. **Die Zukunft planen**

   Im letzten Schritt geht es um die Frage, wie der Prozess in Zukunft aussehen soll. Dafür war es essenziell, den aktuellen Stand zu erfassen, denn nur dadurch lassen sich relevante Verbesserungen erkennen und beheben.

   Wenn Sie nun einen Überblick über alle Parameter und Metriken haben, fällt es erfahrungsgemäß relativ leicht, den gesamten Value Stream zu optimieren. Wenn ein Value Stream erst einmal ordentlich beschrieben und modelliert wurde, fällt allen Beteiligten meist direkt auf, wo zum Wohle der Mitarbeiter und der Kunden Optimierungspotenzial besteht.

6. **Das ganze System verstehen**

   Wenn Sie den oder die zentralen Value Streams Ihres Projekts beschreiben können, sind Sie auf gutem Wege, auch das ganze System zu verstehen. Das ist ganz im Sinne der Three Ways.

Persönliches Exemplar für Thomas Bartholomäus

In unserem Beispiel war die Optimierung der Lead Time und der Process Time zwar das Ziel des Mappings, aber erst, wenn Sie das ganze System im Blick haben, können Sie sagen, ob die Änderungen wirklich einen Mehrwert bieten.

Bei vielen Firmen sieht man in CI/CD-Pipelines nämlich komplexe Genehmigungsstrukturen, die durchlaufen werden müssen, um ein Deployment durchzuführen. Erst wenn alle Stakeholder anwesend sind, kann man hier die Frage stellen: Benötigt man das? Schließlich ist die Lead Time mit ein paar Tagen Wartezeit sehr hoch, während die Process Time nur wenige Sekunden beträgt.

Reflexion

Ein Value Stream Mapping ist ein hervorragendes Werkzeug, um Ineffizienzen aufzudecken. Ich selbst habe schon mit Kunden das ein oder andere Value Stream Mapping durchgeführt. Durch das gemeinsame Notieren und Diskutieren des Value Streams mit den verschiedenen Rollen werden Ineffizienzen und Verbesserungen schnell klar. Es ist daher oft hilfreich, externe Hilfe mit heranzuziehen, die schon Erfahrung in der Durchführung von Value Stream Mapping hat. Ein frischer Blick legt dabei oft (aber nicht immer) eingerostete Verhaltensmuster offen.

Ich kann Ihnen daher sehr dazu raten, sich tiefer mit der Materie zu beschäftigen. Value Stream Mapping hilft, Probleme verschiedenster Arten im Value Stream aufzudecken, die dann in einem separaten Plan angegangen werden können. Und erst nachdem man das Value Stream Mapping durchgeführt hat, sollte man auf die verwendeten Tools schauen. Denn auch hier gilt: *People over Processes over Tools*.

© Rheinwerk Verlag, Bonn 2024

Kapitel 13
DevOps-Plattformen

In den vergangenen Jahren wurde immer deutlicher, dass die Komplexität eines Tool-Stacks zu erheblichen Problemen in der Wartung und im langfristigen Betrieb führt. Über lange Zeit wurden mit viel Heißkleber verschiedene Tools zusammengebastelt, was man vor allem, aber nicht nur, bei historisch gewachsenen Jenkins-Infrastrukturen sieht, die sich nur mit sehr hohem Engineering-Aufwand beherrschen lassen.

In diesem Kapitel befassen wir uns sowohl mit der Historie von DevOps-Tool-Landschaften, die aus unterschiedlichen CI-Tools entstanden sind, als auch mit der Frage, wohin die Reise mit DevOps-Plattformen geht.

Bleiben Sie up-to-date!

Beachten Sie, dass die Informationen aus diesem Kapitel im Gesamtkontext am schnellsten veralten, da der Markt der DevOps-Tools ständig in Bewegung ist. Einige Schwächen, die beim Schreiben dieses Buches im Winter 2023 noch relevant waren, könnten in den Lösungen ein Jahr später schon behoben sein.

Evaluieren Sie also konkret, welche Plattform für Ihre Ansprüche am besten geeignet ist. Auch wenn es natürlich bequem ist, sich auf Erfahrungen anderer Menschen zu verlassen – was für Sie am besten funktioniert, können Sie nur selbst sagen.

Der Begriff »DevOps-Plattform« wird von diversen Herstellern in Anspruch genommen, um die eigenen Produkte damit zu bewerben. Das Marktforschungsunternehmen Gartner, hauptsächlich bekannt durch seinen *Magic Quadrant*, veröffentlicht außerdem den *Gartner Magic Quadrant for DevOps Platforms* (*https://www.gartner.com/en/documents/4416199*). Ursprünglich hieß der Begriff bei Gartner »Value Stream Delivery Platform« und wurde im Laufe der Zeit dann in »DevOps Platforms« umbenannt.

Auch Forrester, ein weiteres Marktforschungsunternehmen, hat sich die verschiedenen DevOps-Plattformen angeschaut. Dort werden sie bislang als »Integrated Software Delivery Platforms« bezeichnet.

Persönliches Exemplar für Thomas Bartholomäus

Gartner und Forrester

Gartners Magic Quadrant für DevOps-Plattformen steckt hinter einer Paywall (*https://www.gartner.com/en/documents/4416199*), daher fasse ich die Ergebnisse kurz zusammen. Der Magic Quadrant von Gartner teilt die Tools in vier Kategorien ein: *Leaders*, *Visionairies*, *Challengers* und *Niche Players*.

Die Angebote von Atlassian, Microsoft und GitLab sind im Report von 2023 die einzigen Vertreter der Kategorie »Leaders«. Bei den Visionären sind die Angebote von Harness, Jet Brains sowie Amazon Web Services vertreten; bei den Challengern findet man Red Hat, VMWare, JFrog, CloudBees und CircleCI. In der Nische findet man noch Codefresh, Bitrise und die Google Cloud Platform.

Das Marktforschungsunternehmen Forrester hat ebenfalls einen »Buyers Guide« zu DevOps-Plattformen veröffentlicht (*https://www.forrester.com/report/the-forrester-wave-tm-integrated-software-delivery-platforms-q2-2023/RES178494*).

Dort werden die Kategorien *Leaders*, *Strong Performers*, *Contenders* und *Challengers* verwendet. Im Bericht aus dem zweiten Quartal 2023 ist GitLab der einzige Leader, gefolgt von CloudBees, der Firma hinter Jenkins. »Starke Performer« sind zudem auch Atlassian, Microsoft, Amazon Web Services sowie Harness und CodeFresh. Der Rest steckt in der Contenders-Kategorie.

Sowohl bei Gartner als auch bei Forrester gilt: Das sind Momentaufnahmen, die sich immer wieder verändern.

13.1 Toolchain-Komplexität

Wenn man sich die Historie der DevOps-Toolchains anschaut, erkennt man diverse Muster. So wie DevOps immer im Wandel ist, so unterliegen auch die Toolchains stets Veränderungen.

Die Toolchain und ihre Komplexität hängen davon ab, wie groß die Firma ist und wann die ersten Bestrebungen nach Automatisierung gestartet wurden. Je nachdem, wie weit die jeweilige Organisation in der DevOps-Kultur und in der DevOps-Toolchain fortgeschritten ist, desto unterschiedlichere Muster sind zu erkennen.

Ich habe in den einzelnen Kapiteln versucht, die Aspekte der DevOps-Idee abzugrenzen, indem ich sie einzelnen Stages zugeordnet habe. Das vereinfacht den Ablauf natürlich, aber Sie werden schon an vielen Stellen gemerkt haben, dass keine harte Abgrenzung möglich ist. Weder die Aufgaben noch die Tools lassen sich strikt den konzeptionellen Schritten der DevOps-Schleife zuordnen.

Teilaspekte der heutigen DevOps-Toolchain sah man schon in Organisationen, die bereits in Richtung Automation gedacht haben, als es DevOps weder als Kultur noch als Begriff gab. Es ging einfach mit der Notwendigkeit los, dass Aufgaben automatisiert werden mussten. Das war Schritt 0.

© Rheinwerk Verlag, Bonn 2024

13.1.1 Schritt 0: Toolchains wachsen historisch

Schritt 0 ist ganz bewusst der »nullte« Schritt, da dieser einfach entstanden ist. Es gab keinen Plan, sondern vieles wuchs über die Zeit historisch. Sie kennen es vielleicht: Immer wenn Sie vor einer neuen Herausforderung standen, haben Sie ein neues Tool eingeführt … bis der Stack kaum noch wartbar war.

So lag der Sourcecode früher häufig in einem Subversion- oder CVS-Repository, später kam immer mehr Git hinzu, aber die Systeme standen immer für sich alleine. Jedes Team setzte seine eigene Infrastruktur auf, sodass es keine zentrale Verwaltung des Quellcodes gab.

Für Automatisierungen kam häufig Jenkins zum Einsatz, dessen Vor- und Nachteile schon in Abschnitt 6.6.1 behandelt wurden. Im Laufe der Zeit hat Jenkins sich in vielen Firmen zum Automatisierungs-Hub entwickelt.

Jenkins wurde häufig erst mal nur an die SCM-Systeme angeschlossen, also an verschiedene CVS-, SVN- oder Git-Server. Für jede Verbindung mussten sowohl ein Plugin installiert werden als auch Zugangsberechtigungen zu den einzelnen Systemen konfiguriert und verwaltet werden. Jede Person und jedes Team, das Zugriff bekommen sollte, brauchte Zugangsdaten, die händisch erstellt werden mussten. Also gab es halt für die meisten Interessenten schlicht keinen Zugang oder man nutzte Standardpasswörter, die extrem unsicher waren. Wenn man es richtig machen wollte, waren die Konfigurationsaufgaben sehr aufwendig und vervielfachten sich pro Team und pro Tool, das angebunden werden musste.

Die Verbindung zwischen SCM-System und CI-Server ist bereits komplex, aber noch komplizierter wird es, wenn auch Server für die Speicherung der Artefakte integriert werden müssen. Oftmals kamen dazu Tools wie *JFrog Artifactory* oder *Sonatype Nexus* zum Einsatz. Auch hier stand man vor denselben Herausforderungen: Die Integrationen mussten konfiguriert werden, und dies erfordert Plugins, Zugangsdaten und die richtigen Berechtigungen für die Tools.

SCM, CI, Package- bzw. Container-Registry sind im Idealfall gerade einmal drei Tools. Es kommen jedoch oft noch viele weitere Tools hinzu, die die Toolchain noch komplexer gestalten: ein eigenes Tool für Continuous-Delivery, verschiedene Sicherheits-Tools für unterschiedliche Programmiersprachen, Helfer für das Requirements-Management und Test-Management sowie ein herkömmliches Ticketsystem wie Jira. Auch all diese Tools mussten konfiguriert und miteinander integriert werden.

Je größer und komplexer die Umgebung war, desto herausfordernder gestaltete sich die Wartung des gesamten Systems. Jenkins ist beispielsweise architektonisch nicht dafür ausgelegt, horizontal zu skalieren. Dies betrifft sowohl die Anzahl der Projekte als auch die steigende Anzahl von Pipelines aufgrund immer größerer Teams. Hinzu kommt, dass die zahlreichen verschiedenen Tools regelmäßig aktualisiert werden mussten, und bei jeder Aktualisierung musste überprüft werden, ob die Verbindung

Persönliches Exemplar für Thomas Bartholomäus

zu den verschiedenen Tools immer noch ordnungsgemäß funktionierte. Probleme waren an der Tagesordnung, sodass eine Aktualisierung eines Tools eine sorgfältige Planung und umfangreiche Tests erforderte. Updates wurden ausgesetzt oder verschoben, neue Funktionen konnten daraufhin nicht genutzt werden. Und Sicherheitsupdates braucht man ja eh nicht, oder?

13.1.2 Schritt 1: Mehrere Toolchains zur Erhöhung der Wartbarkeit

Sie sehen vermutlich, worauf ich hinaus will: Je mehr Tools in der Toolchain stecken, desto komplexer wird es und desto teurer wird auch der ganze Spaß. In vielen Firmen ist so eine Toolchain historisch gewachsen und somit unwartbar geworden, sodass Neuerungen kaum einsetzbar waren.

Für die Mitarbeitenden war dieser Zustand nicht schön, und das führte zu Unzufriedenheit bei tagtäglicher Arbeit. Immer wieder funktionierte etwas nicht, was für den täglichen Arbeitsablauf nötig war. Immer wieder wurde etwas vergessen, weil die einfache Visibilität in benötigte Daten fehlte.

Je mehr Automatisierungen als Lösungen eingeführt wurden, desto höher wurde die Last auf den Systemen. Irgendwann entschloss man sich, die Toolchain aufzubrechen: Jedes Team oder jede Abteilung bekam seine bzw. ihre eigene Infrastruktur für die Toolchain. Dadurch sollte es nicht mehr viele große Monolithen geben, sondern viele kleine.

Das brachte einige Vorteile, aber auch diverse Nachteile mit sich:

- Ein wesentlicher Vorteil ist, dass jeder große Block deutlich verkleinert werden kann. Potenzielle Deadlocks für Updates können so reduziert werden, und ein Ausfall legt nicht die ganze Firma lahm, sondern nur einzelne Teams und Abteilungen. Anstelle eines zentralen Jenkins-Servers, der häufig als Erstes unwartbar wurde, werden hier viele kleinere Server aufgesetzt, die separat verwaltet werden müssen. Bei Wartungsarbeiten können die Systeme Stück für Stück aktualisiert werden. Das reduziert die Einschränkungen durch Downtimes bei Upgrades auf eine deutlich kleinere Gruppe.
- Weiterhin lassen sich bei so einem Verfahren frühzeitig Fehler finden, die bei den Systemen der anderen Abteilungen ebenfalls auftreten würden und die man somit verhindern kann.
- Außerdem kann nun jedes Team sein eigenes Tool wählen. Das sah man auch in vielen Firmen: Je nach Anforderung, persönlichem Geschmack des Teamleiters oder schlicht Zufall wurden neue Tools eingeführt und Insellösungen geschaffen, die nicht dem Standard des Unternehmens entsprachen. So gab es neben Jenkins auch noch Teamcity, CircleCI oder sonstige Lösungen, die mal mehr, mal weniger gewartet und genutzt wurden.

© Rheinwerk Verlag, Bonn 2024

Die Nachteile wurden jedoch rasch deutlich:

- Wenn bei jedem Team ein eigener Toolstack im Einsatz ist, dann kann man schlecht in andere Teams wechseln, denn die Zugriffe müssen nicht nur zwischen den Tools konfiguriert werden, sondern auch für die einzelnen Personen. Eine direkte Kommunikation oder gar ein schnelles Aushelfen ist nicht möglich, da man lange auf Zugriffsrechte warten und dann erst mal den gesamten Toolstack kennenlernen muss.
- Außerdem fehlt der Überblick innerhalb der Organisation völlig: Was funktioniert? Was funktioniert nicht? Wie sieht die Produktivität aus? Welche Librarys werden von welchen Projekten verwendet?
- Je größer die Firma ist – und je nachlässiger man das Thema angegangen war –, desto mehr SCM-Systeme wuchern durch die Infrastruktur. Einen Überblick über die Codebasis zu bekommen, ist so unmöglich, denn niemand kann überblicken, was in diesen ganzen Systemen gespeichert ist. Wenn Sie einmal vor der Aufgabe stehen, einen solchen Wust zu entwirren: Sie brauchen zuerst Zugriffsrechte auf allen Systemen (Wer ist überhaupt Ihr Ansprechpartner?) und müssen dann schauen, welche Projekte noch aktiv weiterentwickelt und betrieben werden. Und was macht man, wenn man für das gleiche Projekt unterschiedliche Codebasen findet?

Ich war einmal dabei, als ein großes Firmennetzwerk nach GitLab-Instanzen durchsucht wurde. Gefunden wurde eine dreistellige Anzahl an Instanzen, größtenteils mit veralteten Versionen und jeweils wenigen Benutzern. Allein durch die Zentralisierung dieser Instanzen wurde die Arbeit ungleich effektiver.

Das Schlimmste an diesem Ansatz ist, dass er die eigentlichen Probleme nicht behebt: Die Konfiguration ist immer noch kleinteilig und nicht automatisierbar, Informationsaustausch und Überblick gibt es nicht. Wer in ein anderes Projekt wechselt, darf sich zum Start mit einem ganz neuen Zoo an Einzelwerkzeugen auseinandersetzen: Alles wurde mit digitalem Gaffertape zusammengeklebt. Das große Ganze betrachtete niemand.

13.1.3 Schritt 2: Standardisierte Tools, aber weiterhin mit viel »Klebeband«

Ich habe zwar bisher Schritt 0 und 1 als aufeinanderfolgende Schritte dargestellt, allerdings ist das eine Generalisierung. Nicht jede Firma fing direkt mit Schritt 0 an, nicht jede hat den Weg von Schritt 0 auf 1 gemacht, viele sind direkt zu 2 gesprungen.

In Schritt 2 wurde vor allem eine Sache angegangen: Der Wildwuchs zwischen den verschiedenen Lösungen wurde aufgelöst. Es sollten nicht mehr mehrere Lösungen für das Sourcecode-Management parallel existieren, sondern nur noch eine einzelne Lösung. Das Gleiche gilt natürlich auch für den CI-Server, auch wenn es weiterhin mehrere Instanzen geben konnte, etwa von Jenkins.

Persönliches Exemplar für Thomas Bartholomäus

Der Unterschied zwischen Schritt 1 und 2 zeigt sich primär an zwei Dingen: Es sollte fortan explizit das jeweils beste Tool für jede DevOps-Stage ausgewählt werden und es sollte ein zentrales Team aufgebaut werden, das diese Tools und deren Integration betreut.

Solche Teams hatten häufig zwei Aufgaben: Sie sollten in erster Linie sowohl die Tools als auch deren Infrastruktur verwalten. Wenig überraschend gilt auch hier: Je mehr Tools es gab, desto komplexer war das Setup. Statt Wildwuchs wie in Schritt 0 und 1 sah man dann einen gewissen Engineering-Aufwand, um die Infrastruktur zu verwalten. Zum Beispiel wurde zwar weiterhin für jedes Team ein Jenkins-Server betrieben, es wurde aber deutlich zentralisierter gearbeitet. Anstatt den Teams die völlige Macht über solche Systeme zu überlassen, wurde eher geschaut, dass diese Arbeit den Teams abgenommen wurde.

Die Auswahl der Tools sollte also innerhalb der Firma standardisiert werden, damit alle effizienter arbeiten konnten. Die Software-Entwicklungsteams sollten schließlich Software entwickeln und nicht ihre DevOps-Toolchain verwalten.

Bei der Auswahl der Tools stellte sich die Frage, welchen Fokus man setzt. Achtet man auf *Best of Breed*, *Best in Suite* oder doch auf was anderes?

Best of Breed

Beim Best-of-Breed-Ansatz wird für jede Problemstellung ein eigenes Tool verwendet: Jira für das Projektmanagement, Bitbucket für das Sourcecode-Management, Jenkins für CI/CD, Artifactory als Package-Registry, Harbor als Container-Registry und noch etliche weitere Tools.

Die Grundidee ist eigentlich nicht verkehrt: Man sucht sich das beste Tool für den jeweiligen Einsatzzweck aus. Nur kommen dann die ganzen Nachteile zum Tragen, die schon in den vorherigen Abschnitten ausführlich betrachtet wurden.

Best in Suite

Der Best-in-Suite-Ansatz steht in direkter Konkurrenz mit dem Best-of-Breed-Ansatz. Hier wird nach einer ganzen Suite an verschiedenen Tools gesucht, die die Aufgaben abdecken. Es kommt nicht mehr darauf an, welches Einzelwerkzeug am besten ist, sondern man achtet auf das Gesamtergebnis.

Beide Varianten haben ihre Vor- und Nachteile. Generell bin ich der Meinung, dass eine integrierte Lösung die bessere Wahl ist. Denken Sie beispielsweise an modere Office-Suites oder an so simple Dinge wie Mails. Rund um E-Mail-Server gibt es schon seit geraumer Zeit eine Vielzahl von Tools, die nahtlos miteinander verknüpft sind. Neben E-Mails gehören auch ein Kontaktbuch und ein Kalender dazu. Erst diese Gesamtheit und der reibungslose Fluss der Informationen zwischen den einzelnen

© Rheinwerk Verlag, Bonn 2024

Teilen macht den Nutzen aus; den besten Kalender und das beste Kontaktbuch zu benutzen, bringt nicht viel, wenn alles per Hand abgeglichen werden muss.

Vielleicht erinnern Sie sich noch an die Präsentation des ersten iPhones durch Steve Jobs. Er kündigte damals an, dass es drei separate Produkte geben würde: einen Widescreen-iPod mit Touchdisplay, ein Telefon und ein Internetkommunikationsgerät. Letztendlich entpuppte sich jedoch alles als ein und dasselbe Gerät: das iPhone.

Das war der richtige Ansatz: Besser alles aus einer Hand anbieten und Funktionen sinnvoll integrieren. Und trotzdem haben natürlich Spiegelreflexkameras weiterhin ihre Existenzberechtigung: Dass es eine Suite gibt, die für die meisten Leute passt, heißt nicht, dass Spezialaufgaben nicht doch besser mit speziellem Werkzeug erledigt werden.

13.1.4 Schritt 3: DevOps-Plattformen

Damit kommen wir auch zu Schritt 3: den DevOps-Plattformen.

Die Hauptaufgabe moderner DevOps-Plattformen ist es, die Komplexität der Toolchain zu beherrschen. Tools müssen zusammenspielen, sie müssen konfiguriert und gewartet werden. Die Lösung für viele dieser Probleme sind DevOps-Plattformen. Die Betonung liegt hier auf »viele«, ich spreche bewusst nicht von »alle«. Das sollte zwar das Ziel der Reise sein, allerdings muss noch viel Entwicklungsarbeit bei den Plattformen geschehen, was noch Jahre dauern wird.

Was sind überhaupt DevOps-Plattformen? Die Grundidee ist, dass auf einer Plattform der komplette DevOps-SDLC abgedeckt wird. Anstatt von einem Tool zum nächsten zu springen, gibt es eine Plattform, in der alles enthalten ist.

Reduzierte Komplexität

Eines sollte hiermit schon klar sein: Je weniger Tools eingesetzt werden, desto weniger Tools müssen konfiguriert und verwaltet werden und desto geringer ist auch die Komplexität der gesamten Toolchain.

Aber auch für die Nutzer – sei es eine Entwicklerin oder ein Admin – wird die Arbeit einfacher. Die Entwicklerin kann ihren Code über mehrere Arbeitsschritte und Tools hinweg verfolgen, ohne den Kontext zu verlieren. Der Admin kann bei einem fehlgeschlagenen Deployment jede Log-Meldung mit einer Code-Zeile und dem entsprechenden Commit in Verbindung bringen. Aus mehr Sichtbarkeit folgt mehr Durchblick.

Einheitliche Oberfläche

Das bedeutet, dass eine einheitliche Oberfläche vorhanden ist, damit sich niemand mehr in etlichen Tools einloggen muss, um alles Relevante zu sehen.

Persönliches Exemplar für Thomas Bartholomäus

Was genau relevant ist, hängt auch von der Position ab. Der Product Owner arbeitet hauptsächlich im Projektmanagement, während die Entwicklerin die meiste Zeit im Sourcecode-Management verbringt. Trotzdem sind alle Informationen stets verfügbar und für alle sichtbar; anders als in getrennten Tools, bei denen niemand wusste, was geplant ist und was schon erledigt wurde. Da in einer Plattform alles innerhalb einer Oberfläche für alle Rollen sichtbar ist, ist die Kollaboration einfacher möglich.

In der Praxis sehe ich häufig, dass sich die Silos gerade durch uneinheitliche Tools mit komplizierten Oberflächen und fehlenden Zugriffsrechten bilden und verfestigen – zu einem gewissen Grad trägt aber natürlich auch Faulheit dazu bei: Vielleicht sind schon Devs und Ops gemeinsam mit denselben Tools unterwegs, aber ein Einblick in das Security-Dashboard hat nur das Security-Team – nicht sonderlich hilfreich!

Reflexion

Wichtig ist, dass Sie und Ihr Unternehmen die Komplexität der DevOps-Toolchain erkennen und diese auch aufbrechen wollen. Viele Firmen haben das Problem schon erkannt, viele allerdings auch nicht.

Gerade in Deutschland sind viele Firmen in der verarbeitenden Industrie tätig sind und arbeiten mit großen Maschinen. In den großen Hallen mit langen Fließbändern und aufwendigen Produktionswegen wird jede kleinste Optimierung durchgeführt, die Zeit bei den Mitarbeitenden sparen kann. Das ist gut und wichtig. Leider fehlt häufig aber die Transferleistung auf die Software-Produktionsstraße: Dort laufen leider viel zu oft die Mitarbeitenden dreimal im Kreis, bevor es einen Schritt weitergeht. Und genau da benötigen diese Firmen Führung von oben, damit der Fortschritt auch bei der Software auch umgesetzt werden kann.

13.2 DevOps-Plattformen im Überblick

Im Wesentlichen gibt es vier verschiedene DevOps-Plattformen auf dem Markt, die ihre volle Leistung erst entfalten, wenn man eine Enterprise-Lizenz für die Vollausstattung bezieht. Namentlich sind es GitLab, dessen Entwickler den Namen »DevOps-Plattform« eingeführt, geprägt und zu einem gewissen Rahmen auch erfunden haben, GitHub und Azure DevOps aus dem Hause Microsoft sowie Atlassian mit seinem Toolstack. Alle diese Plattformen möchten vollständige DevOps-Plattformen sein, allerdings unterstützt keine wirklich die ganze Toolchain so, wie man es in einem Unternehmen erwarten würde. Es ist einfach eine riesige Aufgabe mit vielen Fallstricken.

Eines vornweg: Mit allen vier Plattformen kommen Sie ans Ziel. An allen Plattformen wird aktiv entwickelt, sie verändern sich ständig. (Deshalb habe ich in diesem Buch

© Rheinwerk Verlag, Bonn 2024

auch möglichst auf Screenshots verzichtet und war sparsam mit Schritt-für-Schritt-Anleitungen. Wie etwas konkret funktioniert, erklären Ihnen die inzwischen sehr guten Dokumentationen.)

Der kurze Überblick, der gleich folgt, sollte nicht als Entscheidungskriterium genutzt werden; auch hier kann ich Ihnen wieder nicht abnehmen, die Angebote auf Basis Ihrer konkreten Ansprüche zu evaluieren.

13.2.1 GitLab

GitLab hat den Namen »DevOps-Plattform« geprägt, der von den anderen Plattformen übernommen wurde.

GitLab ist primär für zwei Dinge bekannt: Sourcecode-Management und ein wenig CI. Das sind auch die Funktionen, mit denen das Projekt gestartet wurde, wobei die CI-Komponente zunächst ein separates Projekt war. Schon in frühen Projekttagen wurden beide Teile zusammengelegt, was dazu führte, dass GitLab heute zu der DevOps-Plattform geworden ist, die es heute ist.

Mittlerweile bietet GitLab für alle Stages des DevOps-Lifecycles Lösungen an. Das fängt beim Projektmanagement an, geht über die SCM und CI und das Paket-Management hin zu Continuous-Delivery- und Security- und Monitoring-Lösungen. Wichtig ist, dass man nicht gezwungen wird, alle Funktionen zu nutzen, da einzelne Aspekte problemlos durch Alternativen ersetzt werden können.

Nicht alle Lösungen aus den Stages sind perfekt, aber fast überall werden die Basics abgedeckt. Besondere Stärke liegt auf SCM, CI, CD und Security.

Das Schöne an GitLab ist, dass dahinter eine eher unabhängige Firma steckt. Das Produkt ist Open Core und sehr viele Funktionen sind in der freien Variante verfügbar, die unter einer Open-Source-Lizenz steht. Jeden Monat wird ein neues Release mit etlichen Funktionen veröffentlicht. Dieses kann einfach selbst in der eigenen Infrastruktur gehostet werden.

Insgesamt arbeitet GitLab sehr transparent. Einzelne Funktionsblöcke, aufgeteilt auf die einzelnen DevOps-Stages, sind transparent auf der Maturity-Page verfügbar, inklusive der zukünftig geplanten Features (*https://about.gitlab.com/direction/maturity/*).

13.2.2 GitHub

Obwohl GitHub einige Jahre älter als GitLab ist, ist man dort erst viel später auf den ganzheitlichen Plattform-Ansatz aufgesprungen. Ähnlich wie GitLab begann GitHub ebenfalls mit reinem Sourcecode-Management und einfachen Projektmanagement-Features. Die Komponenten für CI und CD (*GitHub Actions*) kamen erst sehr viel später hinzu und sind noch vergleichsweise jung.

Persönliches Exemplar für Thomas Bartholomäus

GitHub ist am bekanntesten als Plattform für das Hosten von Open-Source-Projekten, die auf *GitHub.com* verfügbar sind. Im Gegensatz zu GitLab ist GitHub selbst jedoch nicht Open Source. Ende 2018 wurde GitHub von Microsoft übernommen und wird seitdem als mehr oder weniger eigenständige Plattform betrieben.

Lange Zeit musste man sich selbst um die eigene CI-Infrastruktur kümmern, die an GitHub angedockt wurde. Dort sah man dann häufig CI-Server wie Travis-CI oder auch Jenkins. Erst 2018 erfolgte die Einführung von GitHub Actions, die CI/CD-Funktionen direkt in GitHub integriert.

GitHub setzt auf einen großen Marketplace. Das ist Fluch und Segen zugleich, so wie es bei Jenkins auch schon war. Fehlende Funktionen können zwar hinzugefügt werden, dafür werden allerdings wieder neue externe Abhängigkeiten geschaffen und es muss viel Zeit in die Konfiguration gesteckt werden, da die Marketplace-Funktionen ja nicht direkt aus der Hand des Herstellers kommen. Eigentlich ist das genau der Punkt, der durch den Einsatz einer DevOps-Plattform vermieden werden sollte.

Um in den Genuss aller Funktionen zu gelangen, wird auch bei GitHub eine Enterprise-Subscription benötigt. GitHub bietet neben GitHub.com auch Enterprise-Lösungen zum Selbsthosten an, die allerdings einen anderen Funktionsumfang besitzen als GitHub.com. Die Features von GitHub.com erscheinen in aller Regel erst deutlich später auf dem GitHub-Enterprise-Server.

Der Funktionsumfang von GitLab und GitHub ist sehr ähnlich. Es ist ein wenig so wie mit Autos: Alle haben vier Räder und man wird schon ans Ziel kommen. Die Frage ist eher: Wie schnell und wie komfortabel ist das möglich und wen kann man auf der Reise mitnehmen?

GitHub versteht sich hauptsächlich als Developer-Plattform. Von DevOps ist in der Selbstbeschreibung eigentlich keine Rede. Das merkt man auch beim Funktionsumfang: Typische Informationen aus dem Betrieb sind in GitHub nicht zu finden.

13.2.3 Azure DevOps

Im Hause Microsoft gibt es neben GitHub auch noch Azure DevOps. Vielleicht kennen Sie es noch unter dem Namen *Visual Studio Team Services*, unter dem es bis 2018 geführt wurde.

Azure DevOps ist ein Angebot aus dem und für das Cloud-Angebot von Microsoft. Entsprechend gut lässt sich die Toolchain in die Microsoft-Azure-Welt einbinden.

Im Gegensatz zu GitHub oder GitLab ist Azure DevOps nur im Firmenumfeld relevant. Es bietet grundsätzlich einen ähnlichen Umfang wie GitHub und GitLab, allerdings mit einem stärkeren Fokus auf Microsoft-Technologien.

© Rheinwerk Verlag, Bonn 2024

Es wird spannend zu beobachten sein, wie die Zukunft von Azure DevOps aussehen wird, da nun zwei Produkte aus dem Hause Microsoft im Wesentlichen auf dasselbe abzielen. Es gibt einige Überschneidungen zwischen beiden Plattformen, beispielsweise ist *GitHub Actions* aus *Azure Pipelines* entstanden und beide haben die gleiche technologische Basis. Dementsprechend gibt es immer wieder Gerüchte, dass Azure DevOps zugunsten von GitHub abgekündigt wird. Aber solange noch nichts von Microsoft direkt kommuniziert wurde, ist das erst mal nur Spekulation.

13.2.4 Atlassian

Atlassian wird zwar von Gartner als »Leader« in seinem Magic Quadrant geführt, allerdings muss man bei dieser Bewertung meiner Meinung nach ein bisschen vorsichtig sein. Die Einstufung ist zwar berechtigt, denn die Tools von Atlassian sind sehr wichtig und in ihren Bereichen oft Marktführer. Im Gegensatz zu GitLab, GitHub und Azure DevOps handelt es sich aber um eine Tool-Sammlung: Sie müssen also alle Tools separat installieren und verwalten. Die Konfiguration für das Zusammenspiel der Tools ist allerdings vernachlässigbar.

Der Atlassian-Toolstack ist besonders für das Projektmanagement-Tool *Jira* und das Wiki *Confluence* von Bedeutung, die weitverbreitet sind und auch stetig weiterentwickelt werden. Das SCM-Tool *Bitbucket* ist relativ ausgereift, große Änderungen sind dort nicht zu erwarten. Die meisten Firmen migrieren aktuell von Bitbucket weg.

Auch die weiteren Tools wie *Opsgenie* für das Incident-Management bei Ausfällen oder *Bamboo* und *Bitbucket Pipelines* für CI/CD-Pipelines spielen in der Praxis eher eine untergeordnete Rolle. Sie alle müssen extra installiert und verwaltet werden, man kann also nicht so recht von einer umfassenden Suite sprechen.

Aber auch Atlassian sieht die Vorteile dieses Ansatzes und bietet seit Kurzem mit *Open DevOps* eine eigene Platform an (*https://www.atlassian.com/solutions/devops*). Dieses Paket wird hauptsächlich von Jira gesteuert, bietet aber andere Tools zur Einbindung mit an: GitHub, GitLab oder Bitbucket für SCM, Jenkins, JFrog, Bamboo und weitere für CI/CD sowie Opsgenie und Confluence.

13.2.5 Sonstige

Neben GitLab, GitHub, Azure DevOps und dem Atlassian-Stack gibt es noch einige Lösungen, die in eine ähnliche Richtung gehen, im deutschsprachigen Raum allerdings kaum verbreitet sind. Dazu gehören die großen Cloud-Provider AWS und GCP, die jeweils eigene Tools im Angebot haben. Hier ist allerdings große Vorsicht geboten, da es sich um wenig genutzte Insellösungen handelt, die stark auf die jeweilige Cloud ausgerichtet sind. Sie binden sich dabei sehr, sehr fest an den jeweiligen Anbieter.

Persönliches Exemplar für Thomas Bartholomäus

Auch JFrog, die Firma hinter Artifactory, hat eine eigene Plattform (*https://jfrog.com/de/platform/*). Es ist eine Verknüpfung mehrerer Tools aus dem Hause JFrog. Große Verbreitung hat diese Plattform nicht.

Reflexion

Der Weg führt heutzutage kaum an GitHub oder GitLab vorbei. Beide sind gute Plattformen für die Entwicklung und das Ausrollen von Produkten und Projekten. Ich sage hier allerdings ganz bewusst nicht »für den Betrieb der Anwendung«, weil es dort noch deutliches Verbesserungspotenzial gibt.

In meinen täglichen Gesprächen mit Kunden ist die Ausrichtung klar: Viele wollen von Bitbucket und Jenkins weg und hin zu GitLab oder GitHub migrieren, um ihre Entwicklungsinfrastruktur zu modernisieren und zu vereinfachen. Für den gesamten DevOps-Lifecycle sind GitHub und GitLab zwar bislang nicht überall vollumfänglich nutzbar, allerdings zeigt der Weg in genau diese Richtung.

13.3 Fazit

Realistisch betrachtet geht der Weg für alle in Richtung DevOps-Plattformen, auch wenn es einige noch nicht erkannt haben. Obwohl GitLab und GitHub schon einige Jahre auf dem Markt sind, merkt man allerdings, dass wir hier noch am Anfang einer langen Reise stehen. Viele Features, die über SCM, CI/CD und Security hinaus gehen, fehlen und sind bislang nur halb gar umgesetzt. Alle haben aber den Fokus, die Kollaboration im Team deutlich zu verbessern und das mit DevOps-Prinzipien.

Wichtig auch hier ist, dass mit dem Einsatz einer der genannten DevOps-Plattformen die DevOps-Transformation nicht plötzlich zum Erfolg wird. Es gilt auch weiterhin: Erst die Menschen auf die geänderten Gegebenheiten einstellen, dann die Prozesse anpassen und erst dann die richtigenn Tools wählen, damit die DevOps-Idee ihre volle Kraft entfalten kann.

© Rheinwerk Verlag, Bonn 2024

Kapitel 14
Jenseits von Kultur und Tools

Die DevOps-Idee wirkt sich auch auf andere Bereiche aus und sorgt für neue Buzzwords, die in diesem Kapitel adressiert werden. So gibt es mittlerweile viele Begriffe, die an DevOps angelehnt sind, oder neue technische Funktionen, die Einfluss auf DevOps haben.

Zunächst geht es um die Rolle von AI in DevOps: Was ist hier wichtig? Was ändert sich? Und: Ist es alles nur ein Hype? Neben AI in DevOps gibt es noch zahlreiche weitere Themen wie DataOps, MLOps oder AIOps. Hier stellen sich weitere, ziemlich ähnliche Fragen: Was ist das und wie hängt es mit DevOps zusammen?

Spannend ist es zudem noch, sich anzuschauen, wie man DevOps in Job-Beschreibungen und im Hiring-Prozess betrachten sollte: Hier gab und gibt es immer wieder Unstimmigkeiten, die es aufzuklären gilt.

14.1 Die Rolle von AI in DevOps

In der IT gibt es immer wieder Hypes, über die zwar alle reden, die sich aber letztlich in der Praxis nicht so etablierten, wie anfangs vermutet wurde. Nachdem in den vergangenen Jahren beispielsweise den Crypto-Währungen oder der Blockchain viel Aufmerksamkeit gewidmet wurde, ohne dass sich Weltbewegendes verändert hat, ging Ende 2022 der Hype um AI los.

Lassen Sie uns die Begriffe ohne Hype anschauen. AI steht für *Artificial Intelligence*; im Deutschen spricht man von KI, also *künstlicher Intelligenz*. Ein Teil davon ist immer auch *Machine Learning*, also ML (siehe Abbildung 14.1).

Beim Machine Learning wird – wie der Name schon andeutet – die »Maschine« trainiert, damit sie Muster erkennt und Ausgaben voraussagen kann, wenn man ihr Eingabeparameter übergibt. Dafür werden Trainingsdaten benötigt, im Hintergrund werkeln neuronale Netze, und für das Training wird sehr viel Mathematik und Statistik benötigt.

Persönliches Exemplar für Thomas Bartholomäus

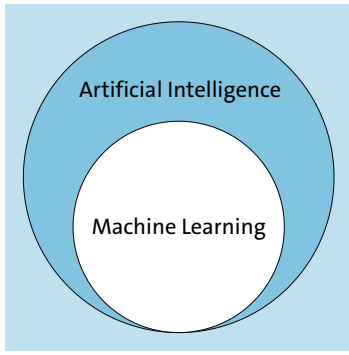

Abbildung 14.1 Machine Learning ist ein Teil von Artificial Intelligence.

Machine Learning löst spezifische Probleme und versucht nicht, eine allumfassende AI zu sein. Die Suche nach einer solchen *General AI* ist ein anderes Thema, aber die Begriffe werden in den Medien gerne durcheinandergeworfen – ich werde im Folgenden auch einfach oft von AI sprechen.

Dass künstliche Intelligenz unglaublich viel kann, steht außer Frage. Dass der Hype um sie (zu) groß ist, leider auch. Man merkt ständig, dass nicht sachlich und nüchtern auf Probleme und Herausforderungen geschaut wird, sondern dass nach einer Lösung geschrien wird, ohne dass das konkrete Problem genau begutachtet wurde.

Sie werden sicherlich schon einmal ChatGPT ausprobiert haben – wie ich auch.

ChatGPT ist ein AI-Chatbot der Firma OpenAI, der mithilfe von etlichen Daten trainiert wurde, um in gewisser Weise intelligent zu sein. Es spricht viele (Programmier-) Sprachen und hat auf fast jede Frage eine Antwort.

Generative AIs wie ChatGPT greifen auf *Large Language Models* (LLM) zurück. Vereinfacht ausgedrückt wird hier mit einer riesigen Anzahl an Parametern und Datensätzen ein Sprachmodell trainiert. Damit lassen sich Sätze erstellen, die mit einer sehr hohen Wahrscheinlichkeit zum Input passen. Diese Modelle können nicht nur für menschliche Sprachen genutzt werden, sondern auch für das Schreiben von Quellcode.

14.1.1 Arbeitserleichterung durch AI-gestützte Code-Generierung?

Wie die meisten war auch ich von den Fähigkeiten von ChatGPT erstaunt. »Das Ding kann doch tatsächlich bei der Entwicklung von Software hilfreich sein!« Vor allem von vielen Führungskräften hörte ich diese Aussagen. Und mehr noch: Wenn man schneller Software entwickeln kann, dann kann auch viel schneller ein Mehrwert für die Nutzer geschaffen und somit mehr Geld gescheffelt werden – vielleicht werden sogar weniger Software-Entwickler gebraucht!

© Rheinwerk Verlag, Bonn 2024

Nach ein wenig Herumexperimentieren wurden schnell einige Stärken und Schwächen sichtbar. Ein einfaches Beispiel ist die Erstellung eines Skriptes, das Daten von einer API abholt und in einem bestimmten Datenformat wieder ausgibt. Wenn man die richtigen Fragen stellt und Aufforderungen eingibt und auch die API öffentlich ist, dann sieht das Ergebnis auf den ersten Blick häufig schon gut funktionsfähig aus:

```
import requests

api_key = 'Ihr-API-Schlüssel'
url = 'https://api.weather.com/v3/weather/forecast'
params = {
    'api_key': api_key,
    'location': 'Berlin,DE',
    'format': 'json'
}

response = requests.get(url, params=params)

if response.status_code == 200:
    weather_data = response.json()
    # Hier können Sie die Daten verarbeiten
else:
    print('Fehler beim Abrufen der Daten: ', response.status_code)
```

Listing 14.1 ChatGPT soll die API von »weather.com« aufrufen.

Wenn man dann aber versucht, das Skript auszuführen, merkt man oft, dass sich das Beispiel gut liest, aber in der Praxis gar nicht funktioniert.

Mir passierte es jedenfalls mehrfach, dass es viele Felder einer API, die vermeintlich existieren sollten, gar nicht gab. ChatGPT halluzinierte Funktionen herbei und machte Vorschläge, die nicht den Anforderungen entsprachen. Nach einigen Iterationen war das Skript dann irgendwann doch lauffähig. In der Zeit hätte ich aber das Skript auch selbst schreiben können.

Prompt Engineering

Für die richtige Arbeit mit den AI-Tools ist es essenziell, dass man weiß, wie man die richtigen Arbeitsanforderungen stellt, damit der passende Code generiert wird. Dafür gibt es schon einen Begriff, nämlich das *Prompt Engineering*.

Die richtige Kommunikation mit der KI ist genauso eine Kunst wie das Requirements Engineering. Auch mit echten Menschen ist es ja schon schwierig genug herauszufinden, was tatsächlich gewünscht und gebraucht wird. Alle, die in der Software-Entwicklung aktiv sind, können ein Lied davon singen, wie Anforderungen der Kunden

falsch verstanden oder nicht richtig dokumentiert werden. So passiert es häufig, dass Anforderungen vom Entwicklungsteam falsch implementiert werden, weil Informationen fehlten oder fehlerhaft waren. Wenn nun noch eine zusätzliche Kommunikationsebene mit der KI hinzukommt, wächst das Potenzial für Missverständnisse.

Dass die AI nicht auf Anhieb perfekten Code generiert, ist aber das kleinere Problem. Für die Software-Entwicklung ist wichtiger, dass die AI erstmal *nur* beim Schreiben des Codes hilft. Software-Entwicklung als Ganzes ist, wie Sie hoffentlich aus diesem Buch gelernt haben, aber viel mehr als nur das Produzieren von Codezeilen. Viel mehr Zeit wird für das Lesen und Verstehen des Codes benötigt. Daneben gibt es auch Aufwand für die Kommunikation im Team, für das Testen und die ganze Toolchain drumherum.

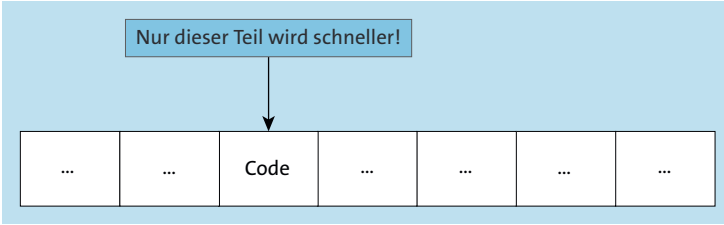

Abbildung 14.2 Durch KI wird das Programmieren tendenziell schneller – aber das ist nur ein kleiner Teil des Value Streams!

Automatisch Tests erstellen

Welchen Einfluss Werkzeuge wie ChatGPT auf die Software-Entwicklung haben werden, vermag aktuell niemand vorauszusagen. Ich halte es für wahrscheinlich, dass die Auswirkungen zunächst weniger beim Schreiben und Erstellen von Code oder gar bei der Software-Architektur zu spüren sein werden, sondern bei den vielen Fleißaufgaben rundherum: beim Erstellen von Dokumentation und Kommentieren von Code sowie besonders beim Testen selbst. Das Schreiben von Unit-Tests lässt sich recht gut an die die KI auslagern, und Computer sind sehr gut darin, sich wiederholende Aufgaben immer wieder exakt aufzuführen. (Einen guten Überblick dazu finden Sie unter *https://www.lambdatest.com/blog/using-chatgpt-for-test-automation/*.)

Das ändert aber nichts daran, dass immer noch geprüft werden muss, ob das daraus resultierende Produkt auch wirklich den Anforderungen entspricht. Integrations- und Funktionstests sollten weiterhin in der Verantwortung der »echten« Intelligenz liegen. Jeder Gedankengang, der als Teil des Code-Schreibens benötigt wird, muss in jedem Fall erfolgen, ganz egal, ob der Code von einer KI oder von einem Menschen geschrieben wurde. Entsprechend sind Reviews noch wichtiger, wenn man gewissenhaft an einem Projekt arbeitet.

© Rheinwerk Verlag, Bonn 2024

Nehmen wir an dieser Stelle also an, dass das Schreiben von Programmcode nur noch halb so lange dauern sollte. Ich nenne hier bewusst das »Schreiben von Programmcode« und nicht die Software-Entwicklung im Allgemeinen, da Letztere viel mehr als nur das Schreiben von Code ist.

Die Frage, die sich nun stellt, lautet: »Selbst wenn man annimmt, dass man 50 % schneller Code schreibt: Hat man dann wirklich so viel mehr Zeit, um noch mehr Code zu schreiben?«

Was häufig fehlt – und was ich auch immer wieder hier im Buch erwähnt habe –, ist ein holistischer Blick auf den gesamten Software-Entwicklungsprozess. Die Tatsache, dass mithilfe von KI schneller Code geschrieben oder generiert wird, hat nur wenig Auswirkungen auf die restlichen Aspekte der Software-Entwicklung. Und genau dieser Blick fehlt denjenigen, die bloß hinter dem Hype hinterherrennen, anstatt einmal kurz innezuhalten, um sich den ganzen Prozess anzuschauen.

14.1.2 Mehr Code führt zu höherem Review-Bedarf!

Hand aufs Herz: Würden Sie Code in der Produktion nutzen, der von ChatGPT erstellt wurde? Wenn ja, dann müssen Sie erklären, warum Sie der KI mehr vertrauen als menschlichen Entwicklern.

Betrachten wir als ersten und wohl wichtigsten Punkt das Testen und den Review: Gerade bei Code, der generiert wurde, muss tendenziell eher noch mehr Zeit in das ausführliche Testen und das Review investiert werden. Und dazu gehört nicht nur, dass man generiertem Code tendenziell misstrauen sollte, sondern auch, dass mehr Code generiert wurde. Es wird also zwangsläufig mehr Zeit für das Testen und Reviewen benötigt. Denn wie gesagt: ChatGPT erfindet Sachen, die zwar auf den ersten Blick schlüssig erscheinen, aber nicht das machen, was sie sollen. ChatGPT ist kein Programmierprofi oder gar eine Faktenmaschine, sondern »nur« ein statistisches Modell, das sehr gut Dinge erstellen kann, die mit einer hohen Wahrscheinlichkeit plausibel aussehen. Ob das Ergebnis inhaltlich stimmt, müssen Sie überprüfen. Und Sie müssen noch mehr prüfen: Ist der generierte Code überhaupt performant oder führt er bei einer bestimmten Datengröße zu Performanceproblemen? Und was ist eigentlich mit Sicherheitslücken? Gibt es subtile Logikfehler, die nicht durch die Tests gefunden werden?

Und selbst dann, wenn die Algorithmen des Modells richtig arbeiten, kann vulnerabler und fehlerhafter Code generiert werden, denn schließlich wurden die Modelle mit Code trainiert, der von Menschen geschrieben wurde. Und dieser Code wurde natürlich nicht immer nach Best Practices entwickelt und ist absolut sicher. *Garbage in, Garbage out* wird auf absehbare Zeit ein großes Problem aller ML-Modelle sein.

Persönliches Exemplar für Thomas Bartholomäus

Wenn Sie im erstellten Code Probleme und Sicherheitslücken finden, fängt die Arbeit eigentlich erst an. Während es bei einem »normalen« Programm meistens eine Entstehungsgeschichte gibt und man nachvollziehen kann, woher der Fehler kommt, steht man bei KI-generiertem Code erst einmal vor einem Rätsel. Es muss viel Zeit investiert werden, um die Probleme zu verstehen und zu korrigieren. Dafür sind mindestens genauso tiefe Kenntnisse in der verwendeten Programmiersprache, den Bibliotheken und dem Quellcode des Projekts nötig, als wenn man den Code von Grund auf selbst schreiben würde. Dieses Wissen ist auch weiterhin unabdingbar. Erschwert wird das Ganze auch noch durch das ständige Wechseln des Kontextes, der für die Reviews nötig ist. Auch das kostet einiges an Zeit.

14.1.3 AI-unterstützende Features

Für mich steht fest, dass auch weiterhin fundierte Kenntnisse in der Software-Entwicklung und im Einsatz der Programmiersprachen gebraucht werden. Dieses Wissen und diese Erfahrung werden auf absehbare Zeit nicht vollständig von KI-Werkzeugen ersetzt werden können. Das heißt aber nicht, dass diese KI-Tools nicht eine große Arbeitserleichterung bedeuten können und Prozesse und Arbeitsabläufe in den nächsten Jahren deutlich verändern werden.

Das Jahr 2023 war voller Neuigkeiten und Ankündigungen. Man sah überall neue Projekte und Start-ups aus dem Boden sprießen, die AI nutzen wollen, um die Software-Entwicklung zu revolutionieren. Für eine Einschätzung ist es noch zu früh, aber man kann schon ein paar Trends erkennen und erahnen, wohin die Reise geht. Ich denke, dass es für jede Stage des DevOps-Lifecycles die eine oder andere AI-unterstützte Lösung geben wird. Beispielsweise wird es in absehbarer Zeit Features geben, die das Schreiben von Pipelines vereinfachen, indem der Quellcode des Projekts gelesen und entsprechend automatisch eine passende Pipeline generiert wird.

Ebenso kann AI helfen, indem die Fehlschläge in Pipelines automatisch analysiert und Root-Cause-Analysen erstellt werden. Die AI könnte dann sogar Ratschläge für die Lösung bereitstellen oder Vorschläge machen, wie die Pipeline optimiert werden könnte.

Im Projektmanagement könnten Zusammenfassungen erstellt werden, die die Arbeit erleichtern, oder es könnten auch Widersprüche in den Beschreibungen entdeckt werden. Kleine Hinweise, dass etwas nicht in Ordnung ist, könnten dabei bereits sehr hilfreich sein.

Das Team-Onboarding sowie die Umstellung auf neue Technologien können mithilfe von AI einfacher gemeistert werden, indem man schneller Code ausprobieren und mit ihm herumspielen kann. Eine neue Programmiersprache und ihre praktischen Anwendungen lernt man am besten in Pair-Programming-Sessions. Mit der Unterstützung der AI kann auch hier effizienter gearbeitet werden.

© Rheinwerk Verlag, Bonn 2024

AI kann auch dabei helfen, Sicherheitslücken und andere sicherheitsrelevante Fragen besser zu verstehen, um das »Sec« in DevSecOps noch stärker zu machen. Oftmals stehen Entwickler und DevOps-Teams vor dem Problem, dass ein Sicherheitsproblem gemeldet wurde, man aber überhaupt nicht weiß, was nun zu tun ist. Worum geht es beim eröffneten CVE überhaupt? Der betroffene Code ist eigentlich veraltet, Tests gibt es nur vereinzelt. Wird die Fehlerkorrektur für alle Szenarien funktionieren? Oder fliegt uns dann die Produktionsumgebung um die Ohren? In diesem Fall kann eine automatisierte AI-Lösung helfen, die die Sicherheitslücke kontextualisiert und erklärt, wie sie ausgenutzt wird. Vielleicht kann es in einigen Jahren möglich sein, dass auf der Basis von statistischen Modellen Sicherheitsprobleme nicht nur gefunden, sondern auch die Gefahren bewertet und minimiert werden können. Gerade bei großen, komplexen Code-Basen, bei denen wortwörtlich kein Mensch den Überblick behalten kann, könnte ein solcher stochastischer Ansatz helfen, die Risiken besser einzuschätzen und zu minimieren.

AI-Tools in der Praxis

Während ich mich mit diesem Buch beschäftigt habe, steckten die meisten AI-Tools in einer sehr frühen Phase. Es lohnt sich an dieser Stelle noch nicht, den konkreten Funktionsumfang zu beschreiben; hier wird sich in den nächsten Jahren viel tun, und ich wage keine Prognose, welche dieser Projekte mittel- und langfristig erfolgreich sein werden.

Neben ChatGPT dürften *Copilot* und *Copilot X* von GitHub sowie *GitLab Duo* am bekanntesten sein. Bei diesen Angeboten wird es darauf ankommen, wie gut sie sich in die Plattformen integrieren und sich über den gesamten Lifecycle erstrecken.

Schaut man ein wenig über den Tellerrand von GitHub und GitLab, finden sich auch noch kleinere Firmen. Sourcegraph bietet etwa mit *Cody* einen AI-Coding-Assistenten an: *https://about.sourcegraph.com/cody*. Spannend ist auch das israelische Start-up *Tabnine*, dessen Software sich auch in eigener Infrastruktur hosten lässt: *https://www.tabnine.com/*. Auch die großen Cloud-Provider bietet eigene Tools an, etwa den *CodeWhisperer* von Amazon oder *Vertex* AI von Google.

Eines haben alle Tools allerdings gemeinsam: Sie verursachen zusätzliche Kosten; ihr Einsatz muss also gut geplant und evaluiert werden.

14.1.4 Datenschutz und Privacy

Bei allen Fragen rund um das technisch Machbare dürfen Sie den Datenschutz und die Vertraulichkeit der Daten nicht vernachlässigen. Werden Ihre Eingaben direkt genutzt, um das Modell weiter zu optimieren und zu trainieren? Fließt der Code, den Sie als Kontext für Prompts liefern, direkt ins Trainingsmaterial ein? (Das sind mindestens die Zeilen darüber und darunter, wenn nicht gar sogar das ganze Projekt,

damit relevanter Code generiert werden kann und für die Autovervollständigung vorgeschlagen wird.) Sind diese Kontexttextinformationen dann abrufbar, indem passende Prompts gestellt werden? Woher stammen überhaupt die Informationen, auf denen die Ausgaben der Tools beruhen?

Die Verantwortlichen von GitHub wurden beispielsweise heftig dafür kritisiert, dass sie für das Anlernen von Copilot die Lizenzen der Projekte ignoriert haben (siehe: *https://www.heise.de/news/Microsoft-GitHub-und-OpenAI-verklagt-KI-Programmierhilfe-Copilot-kopiert-Code-7331566.html*). Dabei gibt es die Open-Source-Lizenzen, die über die Wiederverwendbarkeit des Projekts und somit des Codes entscheiden, gerade aus diesem Grund. In Zukunft wird sich zeigen, was dies für rechtliche Konsequenzen haben kann. Verfolgen Sie beispielsweise dem Rechtsstreit gegen GitHub, der vom amerikanischen Anwalt Matthew Butterick angestrengt wurde: *https://githubcopilotlitigation.com/*. Ähnliche Verfahren wird es auch in der EU geben.

[!]

Sicherheitsrisiken durch AI-Tools

Ich möchte anhand von zwei Beispiele demonstrieren, was schiefgehen kann:

Dass man mit ChatGPT nicht nur neuen Code erstellen kann, sondern dass damit auch alter Code optimiert werden kann, ist beispielsweise den Entwicklern von Samsung nicht entgangen. Sie luden zahlreiche Codezeilen hoch, die ChatGPT überarbeiten und verbessern sollte. Da ChatGPT zumindest in der kostenfreien Variante die Eingaben der Nutzer verwendet, um das System weiterzuentwickeln, konnte man durch geschicktes Fragen an den Code kommen, den die Samsung Engineers in ChatGPT gekippt hatten:

https://www.techradar.com/news/samsung-workers-leaked-company-secrets-by-using-chatgpt

Die meistens Firmen (beispielsweise Apple, Amazon und auch Samsung) haben daher entschieden, dass ChatGPT nicht mehr für Firmenaufgaben genutzt werden darf, da die Befürchtung besteht, dass vertrauliche Daten über die ChatGPT-Firma OpenAI an deren größten Investor Microsoft geschickt werden. Bei anderen AI-Tools ist der Datenschutz (zumindest angeblich) restriktiver geregelt, und auch ChatGPT verspricht, dass die Eingaben nicht weiterverwendet werden, wenn die kostenpflichtige Pro-Version genutzt wird.

Aber auch, wenn gar nicht aktiv Daten geteilt werden, kann es zu Problemen kommen, da die Anbieter der AI-Helfer das gesamte Internet nach Trainingsdaten durchforsten. Im November 2023 wurde bekannt, dass Kundeninformationen der Baumarktkette Bauhaus über das AI-Tool *Bing Chat* (heute: Microsoft Copilot) abgerufen werden konnten. Der Hintergrund war zunächst ein recht simples Sicherheitsproblem: Die Informationen standen für einige Zeit ungeschützt im Internet. Dadurch wurden sie von der Suchmaschine Bing indiziert. Obwohl der Datenabfluss schnell behoben wurde, konnten die Informationen weiterhin über geschickte Fragen an das Chat-Tool abgerufen werden.

© Rheinwerk Verlag, Bonn 2024

Den genauen Fall hat Günter Born auf seiner Seite geschildert: *https://www.born-city.com/blog/2023/11/22/datenleck-beim-bauhaus-shop-verriet-bestellungen-von-plus-card-inhabern/*

Ein grundsätzliches Problem dabei ist, dass die meisten AI-Dienste ausschließlich als *Software as a Service* verfügbar sind. Wenn man sie nutzen möchte, muss man zwangsläufig die Daten an sie schicken. Alternativen für das Selbsthosting gibt es kaum, da sowohl für das Training als auch für die Nutzung von AI-Systemen hochperformante Systeme mit sehr viel GPU-Power benötigt werden.

Firmen in Deutschland und in der EU achten besonders darauf, wo die Daten eines Cloud-Dienstes gespeichert werden. Stichworte sind hier im Besonderen der *CLOUD-Act* und die *Data Residency*. Das löst das Problem allerdings nicht, dass private, firmeninterne Daten weitergegeben und verarbeitet werden müssen.

Auch hier dürfte sich in der Zukunft sicherlich einiges ändern. Bis dahin sollten Sie derartige Probleme allerdings im Hinterkopf behalten.

14.1.5 Das Gesamtkonzept macht's!

Die große Stärke der DevOps-Idee ist, dass sie versucht, den ganzen Value Stream der Software-Entwicklung zu betrachten. Das ist auch bei der Beschäftigung mit den neuen (und vielleicht revolutionären) AI-Helfern ein großer Vorteil: Ja, das Schreiben des Codes wird sich verändern, in Zukunft wird man besonders einfache Funktionen kaum noch selbst schreiben, sondern diese sich direkt generieren lassen können. Das ist aber nur ein kleiner Teil des Value Streams. Haben Sie immer das große Ganze im Blick, und fragen Sie sich, ob automatisch generierter Code wirklich Probleme löst. Drückt der Schuh an anderen Stellen nicht viel mehr? Müssen nicht eher andere Prozesse und Arbeitsabläufe beschleunigt werden? Wie kann künstliche Intelligenz dabei helfen?

Es wird in Zukunft viele kleinere und größere Helferlein im gesamten Prozess geben (siehe dazu auch Abschnitt 14.2.3, »AIOps«, zu dem wir gleich kommen), deren Zusammenspiel den Erfolg ausmachen wird. In diese Richtung entwickeln sich jedenfalls *GitLab Duo* und *Copilot X* von GitHub: umfassende Angebote, die den ganzen Lifecycle in den Blick nehmen.

In der Realität sieht es leider so aus, dass sehr viele Firmen in Deutschland bisher kaum echtes DevOps machen. Damit meine ich nicht nur die Teamstruktur und die Kultur der Zusammenarbeit, sondern auch, wie auf der technischen Ebene gehandelt wird.

Häufig haben die Führungskräfte kein großes Vertrauen in das regelmäßige Ausrollen von Änderungen auf Produktivsysteme. Außerdem wird oft den Menschen, die die Software geschrieben haben, nicht vertraut. Und in diesen Fällen hilft es auch nicht, wenn Code, der von einer AI geschrieben wurde, ebenfalls ausgerollt werden soll.

Persönliches Exemplar für Thomas Bartholomäus

Was ich vor allem damit sagen will: Bevor man anfängt und AI-unterstütztes Programmieren als Lösung vieler Probleme betrachtet, müssen erst alle anderen Punkte im Value Stream korrigiert werden. Wenn jedes Ausrollen einer Änderung ein Krampf mit Fehlern und Downtimes ist, weil nichts automatisiert wird, dann bringt es nichts, wenn Code blitzschnell durch die AI erstellt wird. Code-Änderungen stauen sich letztlich nur noch weiter auf und sorgen für mehr Probleme und Frust. Wenn man schon heute nicht dem Code vertraut, den Menschen geschrieben haben, dann wird es mit der AI auch nicht besser – im Gegenteil!

Verstehen Sie mich nicht falsch: Ein Vorteil von AI-unterstütztem Programmieren ist unleugbar. Der Vorteil im Gesamtkontext ist allerdings deutlich kleiner, als man erwartet. Und das gilt besonders, wenn der Rest des Value Streams noch sehr altbacken ist. Das Fazit lautet also: AI kann bei der Software-Entwicklung helfen. Es braucht aber viel mehr, als den Entwicklern einfach nur einen ChatGPT-Zugang zu geben und sich automatisiert Code erstellen zu lassen. Der Blick muss auf das große Ganze gerichtet sein, um mögliche Verbesserungen zu erkennen und zu nutzen. Wenn die DevOps-Idee richtig in der Organisation gelebt wird, ist der Grundstein dafür gelegt.

Reflexion

Je nachdem, wann genau Sie diese Zeilen lesen, hat sich vermutlich schon vieles wieder geändert. AI in DevOps wird immer wichtiger, allerdings sollten Sie nicht sofort dem Hype hinterherlaufen. Insbesondere Fragen des geistigen Eigentums und des Datenschutzes gilt es zu beachten. Hier ist also noch sehr viel in Bewegung! Wichtig ist, dass Sie nüchtern an das Thema herangehen und immer das große ganze Bild im Blick haben.

14.2 DataOps, MLOps – was es sonst noch alles gibt

Der Erfolg der DevOps-Idee hat dafür gesorgt, dass sich noch viele weitere Begriffe etabliert haben, die ebenfalls auf »Ops« enden. Wenn Sie sich zum ersten Mal damit beschäftigen, kann dies sehr verwirrend sein, denn es ist nicht klar, ob man auf diese Weise DevOps ersetzen, ergänzen oder völlig umkrempeln will.

Werfen wir daher einen kurzen Blick auf *DataOps*, *MLOps* und *AIOps*, die alle einen anderen Fokus setzen. Nichts von ihnen ersetzt DevOps, sondern es geht darum, die DevOps-Prinzipien auf Aufgaben jenseits der reinen Software-Entwicklung auszuweiten.

14.2.1 DataOps

Bei *DataOps* werden die DevOps-Prinzipien auf datengetriebene Lösungen angewendet, und zwar sowohl die kulturellen als auch die technischen Aspekte. Es geht dabei

© Rheinwerk Verlag, Bonn 2024

um die Arbeit von Mathematikerinnen und Statistikern, die sich um die Analyse und Verarbeitung der Daten kümmern, die in einem Unternehmen gesammelt werden.

Die Lehren und Erfahrungen der Software-Entwicklung können angewendet werden, um die Datenanalyse zu beschleunigen und die Abläufe wie das Extrahieren von Daten, die Vorbereitung, die eigentliche Berechnung und die Illustration in Dashboards und Reporten zu automatisieren. Abbildung 14.3 zeigt die einzelnen Schritte in Form einer Pipeline. Automatisierung und ein Fokus auf die Reproduzierbarkeit der Aufgaben sollten dabei eine genauso große Rolle wie bei der Software-Entwicklung spielen.

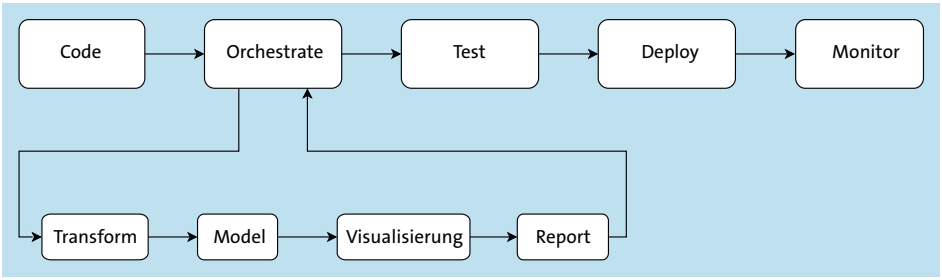

Abbildung 14.3 Diese exemplarische DataOps-Pipeline enthält zusätzliche Aspekte für die Datentransformierung, Modellierung, Visualisierung und für das Erstellen von Reports.

Wenn kein DataOps angewandt wird, werden viele dieser Aktionen manuell durchgeführt, also etwa das Hin- und Herschieben von Daten, um im nächsten Schritt die Analyse zu fahren. Das ist nicht nur mühselig, sondern auch fehleranfällig und teuer, da bei solchen Ansätzen riesengroße Datensätze enthalten sind.

Und auch die fehlende Kommunikation kann sich genauso wie in Entwicklungsprojekten bemerkbar machen: Die Validierung der Daten, die Modellierung und die Integration in den Workflow werden im Allgemeinen von unterschiedlichen Teams erledigt. Wenn diese ihre Ergebnisse einfach nur über den Zaun werfen, hat man genauso schlechte Prozesse wie in den schlimmsten Entwicklungsprojekten. Die Lösung ist eine ordentliche DataOps-Pipeline, die genauso genutzt wird wie eine CI/CD-Pipeline: Die Arbeit mit den Daten wird in Code formuliert, der in Repos versioniert wird. Alle Änderungen müssen ein Code-Review durchlaufen, ständige Tests sorgen für die Qualitätssicherung – also genau so, wie man es aus einem guten Software-Projekt kennt.

14.2.2 MLOps

Das ML in *MLOps* steht für *Machine Learning*. Machine Learning selbst ist der Fachbegriff, der weniger mit Marketing-Hype belastet ist als AI, da meist ein konkretes Problem existiert, wofür die Maschine trainiert wird, um es mithilfe von Machine-Learning-Algorithmen zu lösen.

Machine Learning setzt sich im Wesentlichen aus drei Bestandteilen zusammen: aus den Daten, dem Modell und dem Quellcode. Ein Machine-Learning-Algorithmus ist im Code definiert. Es werden dann die Daten verwendet, um das Modell zu trainieren, das am Ende dafür verwendet wird, um über statistische Auswertungen Prognosen zu treffen oder neues Material zu generieren.

MLOps ist eine technische Ergänzung des DevOps-Prinzips, auch wenn der Zusatz »Dev« nicht Teil des Namens ist. Bei diesem Begriff geht es nicht darum, wie künstliche Intelligenz die DevOps-Arbeit einfacher machen kann, sondern darum, wie der Prozess des Machine Learnings selbst organisiert wird.

Das betrifft beispielsweise das Anlernen der Modelle und die Verwaltung der Trainingsdaten, denn ein häufiges Problem beim Machine Learning ist, dass die Modelle nur einmal trainiert und dann in der Software irgendwie paketiert und ausgerollt werden. Stattdessen sollte das Modell immer wieder mit neuen Daten verbessert werden. Zudem muss stetig ausgewertet werden, wie gut oder schlecht die Experimente performen. Ohne gescheite Automatisierung ist das offensichtlich nicht möglich. Es geht also auch hier wieder darum, DevOps-Prinzipien auf andere Bereiche anzuwenden.

Wenn Sie ML-Techniken in Ihren Projekten nutzen wollen, geht es zunächst darum, die KI-Spezialisten abzuholen, die meist einen anderen Toolstack nutzen, und sie mit den Aufgaben und Prinzipien der DevOps-Kultur vertraut zu machen.

Schwieriger ist es, die technischen Voraussetzungen entsprechend zu integrieren, denn die Tool-Chain des ML-Modells und die riesigen Datenmengen, die benötigt werden, lassen sich nicht ohne Weiteres in »klassischen« CI/CD-Pipelines abdecken. Da Machine-Learning-Modelle aus Binärdateien bestehen, kann damit nicht einfach in Review-Umgebungen experimentiert werden, wie man es von reinem Sourcecode gewohnt ist. Sie können das Modell auch nicht beliebig neu anlernen, denn das würde nicht nur viel zu lange dauern, sondern auch unglaublich teuer werden.

Daher entstehen gerade neue Plattformen und Werkzeuge, die speziell auf die Arbeit eines ML-Projekts ausgerichtet sind. Schauen Sie sich beispielsweise *MLFlow* (*https://mlflow.org/*) an, das den ganzen Lifecycle abdeckt.

14.2.3 AIOps

Beim Arbeiten an diesem Buch bin ich erstmals auf den Begriff *AIOps* gestoßen und fragte mich direkt, ob das nun so etwas wie MLOps ist.

Beim näheren Hineinlesen fällt aber schnell auf: Nein, hier geht es »nur« darum, dass Artificial Intelligence für den reinen IT-Betrieb genutzt wird, also dass man sich die Alltagsaufgaben rund um den Betrieb der IT-Intrastruktur durch intelligente Werkzeuge einfacher macht. Fast immer ist die Idee dahinter, dass mithilfe vieler verschiedener Datenquellen Probleme frühzeitig erkannt werden. Zu diesen Daten gehören

© Rheinwerk Verlag, Bonn 2024

etwa Events, Logs oder Metriken, mit denen das System trainiert wird. Wenn ein ML-Modell beispielsweise gelernt hat, wie der »normale« Traffic eines Netzwerks aussieht, kann es bei Anomalien schneller Alarm schlagen, als dies selbst der aufmerksamste Admin könnte. So können Sicherheitsprobleme erkannt werden, wenn es Zugriffe im Netzwerk gibt, die nicht den bisherigen Mustern entsprechen.

Besonders komplexe Infrastrukturen werden in der Zukunft kaum noch ohne solche Werkzeuge verwaltet werden können – ganz ähnlich, wie in der Industrie kaum etwas ohne *Predictive Maintenance* läuft. Welche kommerziellen Produkte (besonders aus dem Observability-Bereich) langfristig Erfolg haben werden, lässt sich allerdings schwer sagen. Prüfen Sie, was Sie wirklich brauchen und was den Betrieb tatsächlich einfacher und sicherer macht.

Reflexion

DataOps und MLOps sind Themen, die sich in Zukunft noch weiter entwickeln werden. Inwiefern sie für Sie nützlich werden, hängt natürlich stark davon ab, in welcher Rolle Sie sind. Die Wahrscheinlichkeit ist hoch, dass weitere Funktionen in bestehende Plattformen wie GitHub und GitLab einfließen werden, damit man auch noch weitere Use-Cases abdecken kann, auch wenn eigenständige Firmen und Tools wie *Databricks* (*https://www.databricks.com/*) und *Hugging Face* (*https://huggingface.com*) existieren.

14.3 DevOps als Job

In Kapitel 1 habe ich schon kurz angerissen, dass der Begriff »DevOps Engineer« eigentlich falsch ist. »DevOps« im Jobtitel kann zwar durchaus Sinn ergeben, wird aber häufig falsch verwendet.

Wenn Sie dieses Buch aufmerksam gelesen haben, werden Sie verstanden haben, dass DevOps im Wesentlichen eine Kultur ist, die nur funktionieren kann, wenn die beteiligten Menschen sie akzeptieren. Erst danach kommen die Prozesse, die das Ganze unterstützen, und danach folgen erst die Tools, die das Ganze ermöglichen.

Dass DevOps keine Rolle ist, sollte also mittlerweile klar sein. Trotzdem findet man (leider) häufig Stellenbeschreibungen, in denen nach »DevOps« gesucht wird. Häufig hört man auch von Personen, wenn man nach ihrem Job fragt, dass sie »DevOps machen«. Ich frage hier immer kritisch nach, da eben jeder etwas anderes unter dem Begriff versteht.

14.3.1 Die Frage nach den DevOps Engineers

Sich selbst als »DevOps Engineer« zu bezeichnen, kann je nach Rollenbeschreibung manchmal korrekt sein und manchmal nicht.

Häufig wird diejenige Person als DevOps Engineer bezeichnet, die die CI/CD-Pipeline und den ganzen DevOps-Toolstack verantwortlich betreut. Das halte ich in vielen Fällen für einen Fehler. Grundsätzlich sollte jedes Teammitglied in der Lage sein, die CI/CD-Pipeline anzupassen und auch die nötigen Ändeurngen am Rest der Toolchain vorzunehmen. Was nicht passieren sollte, ist, dass nur eine Person alles aufbaut und niemand sonst das Setup versteht. Das sieht man leider häufig, aber es steht komplett konträr zum eigentlichen DevOps-Gedanken, bei dem es um die Kollaboration der verschiedenen Rollen geht.

In größeren Firmen kann es allerdings sinnvoll sein, »DevOps Engineers« zu haben. Ihr Use-Case ist, dass sie teamübergreifend die Ansprechpartner für alle Themen rund um das große Thema DevOps sind. Dabei geht es selbstverständlich nicht nur um die kulturellen Themen, sondern vor allem um den Support für den technischen Toolstack.

Solche Mitarbeiter und Mitarbeiterinnen sollten *Enabler* für die übrigen Teams sein. Ihre Aufgaben können darin bestehen, den Teams helfend zur Seite zu stehen, wenn Spezialwissen benötigt wird oder konkrete, sehr komplexe Probleme gelöst werden müssen. Grundsätzlich sollte der DevOps Engineer dafür sorgen, dass die Teams autark arbeiten können und *nicht* auf ihn angewiesen sind. Ein guter DevOps Engineer schafft sich selbst ab.

Das gilt nicht nur für die einzelne Person, sondern auch für das ganze Team-Pattern. Ein »DevOps-Team« wie in Abbildung 14.4 macht zwischen Teams aus Devs und Ops keinen Sinn – es sei denn, die Aufgabe dieses Teams ist, sich selbst abzuschaffen.

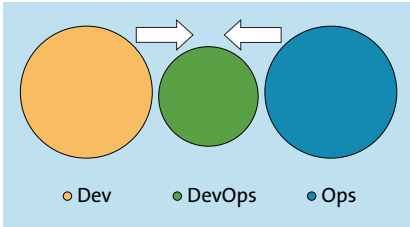

Abbildung 14.4 DevOps-Team-Topologie

Normal und nachvollziehbar ist es hingegen, dass es mehrere Personen gibt, die sich mal mehr und mal weniger mit den Pipelines und dem Setup auskennen. Aber das Gleiche gilt natürlich auch für die Entwicklungsarbeiten, wie auch für die Betriebsaufgaben: Alles zu kennen, ist nahezu unmöglich. Es sollte allerdings immer versucht werden, eine fundierte Basis im gesamten Team zu haben.

Außer für die »richtigen« DevOps-Teams und die »richtigen« DevOps Engineers kann die Bezeichnung »DevOps« auch für Consultants oder Trainer passend sein, die zu diesen Themen beraten und schulen.

© Rheinwerk Verlag, Bonn 2024

In Abschnitt 14.3.3 geht es um den technischen DevOps-Lernpfad. Das ist meist das, was als Kernaufgabe eines DevOps Engineers angesehen wird. Es kommt allerdings immer auf den genauen Kontext an. Ich halte oft nach Anzeigen Ausschau, die »(Senior) DevOps Engineers« suchen, und finde beide Arten: die Rolle, wie von mir beschrieben und gewünscht, sowie die »falschen« DevOps Engineers als Teil eines nahezu reinen Entwicklungsteams.

Wenn das der Fall ist, dann ist das ein guter Indikator, dass einiges nicht so laufen wird, wie man es erwartet. Dabei wird meistens erwartet, dass man zusätzlich noch Ops-Aufgaben im Dev-Team übernehmen soll. Daher mein Appell: Beobachten Sie den Gebrauch des Begriffs schön kritisch, und haken Sie bei einem Jobinterview genau und notfalls mehrfach nach, wie die Stelle nun gedacht ist.

Eine Sache ist allerdings noch einmal wichtig hervorzuheben: Spezialisierungen auf einige Themengebiete sind völlig normal, in Ordnung und häufig auch gar nicht vermeidbar. Nicht jeder ist der geborene Entwickler und nicht jeder interessiert sich für alle Details des Betriebs. Die gesunde Mischung macht es aus, und das muss auch die Firma sehen und es entsprechend steuern.

14.3.2 Soft Skills

Bevor es um die technischen Aspekte des DevOps-Lernpfades geht, ist es noch mal enorm wichtig, über die Soft Skills zu sprechen. Entgegen ihrem Namen sind Soft Skills ganz und gar nicht »soft«, sondern ziemlich hart zu lernen und anzuwenden.

Probleme verstehen

Einer der wichtigsten Punkte ist das richtige Verstehen von Problemen. Damit ist gemeint, dass Sie Zusammenhänge und das große Ganze erkennen müssen, damit Probleme an der Wurzel gepackt werden können. Bauen Sie keine Übergangslösungen, verrennen Sie sich nicht in Details, betreiben Sie kein Overengineering.

Häufig ist stattdessen zu sehen, dass an einer Lösung gearbeitet wird, ohne genau die Problemstellung verstanden zu haben. Gerade wenn Sie als DevOps Engineer unterschiedliche Teams und Anforderungen im Blick behalten müssen, ist die saubere Analyse eines Problems sehr wichtig, da Sie sonst nur viele Kleinigkeiten korrigieren und anpassen, statt wirklich voranzukommen.

Kommunikation

Das Verstehen von Problemen ist eng mit der Kommunikation verzahnt. Gerade im Zusammenspiel mit diversen weiteren Mitarbeitenden ist eine effiziente und effektive Kommunikation essenziell für den Erfolg.

Persönliches Exemplar für Thomas Bartholomäus

Als DevOps Engineer sind Sie Bindeglied zwischen verschiedenen Teams und Rollen. Hier müssen also verschiedene Personen mit ihren unterschiedlichen Zielen zusammengebracht werden. Das betrifft dann auch das Erklären von möglichen Lösungen. Versuchen Sie stets, den Blickwinkel Ihrer Gesprächspartner zu verstehen, und hören Sie idealerweise mehr zu, als selbst zu reden.

Kollaboration

Gute Zusammenarbeit bedeutet gegenseitigen Respekt, Geduld und Vertrauen. Man muss auch bei Konflikten und unterschiedlichen Zielen produktiv zusammenarbeiten und gemeinsam nach Lösungen suchen.

Und somit laufen auch die drei Soft Skills zusammen: Das Zusammenbringen der verschiedenen Rollen ist wichtig, um Probleme zu verstehen, was wiederum nur mit einer gescheiten Kommunikation geht.

14.3.3 Der technische DevOps-Lernpfad

Obwohl mir die DevOps-Kultur wichtiger als die Werkzeuge ist, kann der richtige Umgang mit der DevOps-Toolchain nicht ignoriert werden. In diesem Abschnitt geht es mir darum, Ihnen einen Einblick zu geben, was Sie wissen und kennen müssen. Abbildung 14.5 gibt einen Überblick über den Lernpfad.

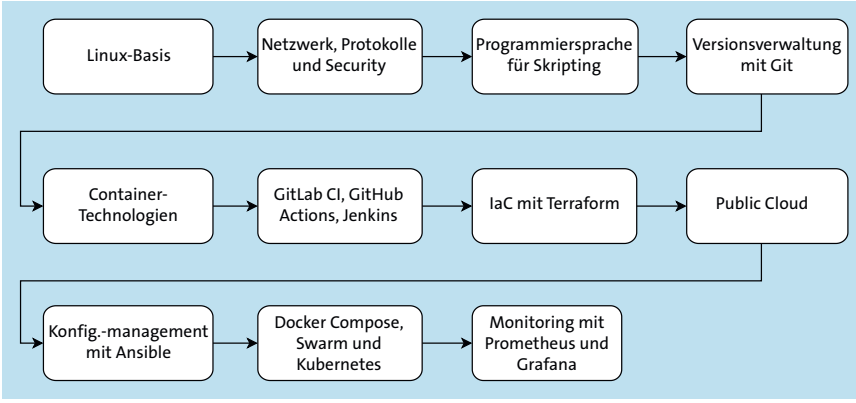

Abbildung 14.5 Der technische DevOps-Lernpfad besteht aus elf fachlichen Vertiefungen.

Grundsätzlich wäre es natürlich toll, wenn Sie in allen Tools Expertenkenntnisse mit Praxiserfahrung aufbauen. Das ist aber nicht nur unrealistisch, sondern quasi unmöglich, da es schlicht zu viele Werkzeuge und Techniken sind. Und sie verändern sich ständig. Was gestern noch Best Practice war, ist vielleicht morgen schon ein schlimmer Hack. Die Marktführer von heute können morgen schon in einer Sackgasse stecken.

© Rheinwerk Verlag, Bonn 2024

Ihr Fokus sollte also eher darauf liegen, welche fundierten Grundlagen nützlich sind und womit Sie anfangen sollten, um möglichst viel in der Praxis umsetzen zu können. Hier geht es weniger darum, jede einzelne Nische mit abzudecken, sondern mehr darum, den breiten Bedarf an Kenntnissen im Jobmarkt zu treffen, damit Sie solide Kenntnisse vorweisen können.

Linux ist die Basis

Um einen umfassenden Lernpfad zu verfolgen, sind solide Linux-Kenntnisse von grundlegender Bedeutung. Diese bilden die Basis für alle nachfolgenden Schritte und sind nahezu unverzichtbar, da fast alles, was im Anschluss kommt, auf diesen Kenntnissen aufbaut. Im Kontext von DevOps und insbesondere im Bereich Cloud Native sind Linux-Kenntnisse in vielerlei Hinsicht unerlässlich. Im Gegensatz dazu ist Windows als Betriebssystem in diesem Bereich eher die Nische.

Der erste Schritt besteht darin, das grundlegende Funktionsprinzip des Linux-Systems zu verstehen, da dieses Wissen später für das Erstellen von Container-Images und für den Betrieb von Containern unerlässlich ist. Besonders wichtig sind dabei umfassende Kenntnisse in der Verwendung der Shell. Dies betrifft sowohl die interaktive Nutzung als auch das Beherrschen von Skriptsprachen. Tatsächlich spielt es dabei eine untergeordnete Rolle, welche spezifische Shell verwendet wird, da die grundlegenden Funktionen, die relevant sind, in den meisten Fällen sehr ähnlich sind. In der Regel ist die Bash-Shell die gängigste und empfohlene Wahl.

Die konkrete Linux-Distribution ist eher von geringerer Bedeutung. Viel wichtiger ist, dass Sie sich mit Paketmanagern auskennen und wissen, wie sie im Gesamtsystem interagieren. Es ist auch ratsam, zwischen Linux-Distributionen für Host-Systeme und solchen für das Erstellen von Container-Images zu unterscheiden.

Bei Host-Systemen handelt es sich um die Systeme, die Container hosten. Hier sind Debian-basierte Systeme wie Ubuntu oder Distributionen, die mit Red Hat Enterprise Linux (RHEL) kompatibel sind, von Bedeutung. Jedoch ist DevOps nicht nur auf Container beschränkt. Wenn Sie solide Linux-Kenntnisse besitzen, ist es nicht allzu schwierig, sich auch mit den Grundlagen von Alpine Linux oder anderen Minimal-Distributionen vertraut zu machen.

Netzwerke, Protokolle und Security

In der Welt von DevOps sind allgemeine Netzwerk- und Infrastrukturthemen eng mit den Betriebssystemen verknüpft. Zusätzlich sollten Sie grundlegende Konzepte in den Bereichen Virtualisierungslösungen, Dateisysteme sowie CPU- und Speichermanagement umfassend verstanden haben. Letztere sind besonders relevant, wenn es darum geht, Anwendungen zu betreiben, insbesondere wenn Sie mit CPU- und Speicherbegrenzungen konfrontiert sind.

Persönliches Exemplar für Thomas Bartholomäus

Ein solides Verständnis von Prozessen, Sockets und Netzwerken ist ebenfalls unerlässlich. Man könnte manchmal denken, dass diese Themen in der Ära von Cloud und Container-Infrastrukturen irrelevant geworden sind, aber auch in dieser Umgebung spielen sie eine wichtige Rolle. Beispiele sind Security Groups und Netzwerkregeln, die bei der Sicherheit und Kommunikation in der Cloud entscheidend sind.

Zu diesem Bereich gehört auch ein fundiertes Wissen über Protokolle. SSH ist unverzichtbar und gehört zu den Grundkenntnissen in der Welt von Linux. Darüber hinaus sollten Sie auch die Grundlagen von HTTP und SSL/TLS verstehen. Besonders relevant ist der Umgang mit REST-Schnittstellen im Allgemeinen, da es oft erforderlich ist, Daten von Diensten über REST-Schnittstellen abzurufen. Dieses Wissen ist auch für das Scripting und die Automatisierung von großer Bedeutung.

Programmiersprachen für das Scripting

Es ist von entscheidender Bedeutung, eine Programmiersprache zu beherrschen, insbesondere wenn Sie in einer Rolle arbeiten, die sich hauptsächlich um die Unterstützung der gesamten CI/CD-Infrastruktur für verschiedene Teams kümmert. Eine Skriptsprache kann Ihnen in dieser Hinsicht enorm weiterhelfen. Der erste Schritt ist oft das Shell-Scripting, wie ich zuvor bereits erwähnt habe. Allerdings stoßen Sie schnell an Grenzen, wenn die Aufgaben komplexer werden.

Shell-Scripting ist zweifellos wichtig, aber Sie sollten niemals versuchen, alles mit einem Shell-Skript zu lösen, nur weil es möglich ist. Gute Kenntnisse in Shell-Scripting zeigen sich darin, dass man weiß, wann man es besser nicht mehr einsetzen sollte.

Der nächste sinnvolle Schritt ist das Erlernen von Python. Bash-Scripting wird hauptsächlich für das Pipelining, das Debugging von Systemen und das Erstellen von Container-Images benötigt. Sobald Ihre Aufgaben komplexer werden, wird Python eine große Hilfe sein. Typischerweise liegen die Anwendungsfälle von Python im Bereich der Infrastruktur und weniger in der Unterstützung der Anwendungen selbst.

Python ist besonders nützlich, wenn Sie auf APIs zugreifen müssen, um Daten für die Erstellung von Berichten zu sammeln. Neue Herausforderungen tauchen regelmäßig auf, und grundlegende Kenntnisse in einer Programmiersprache wie Python können Ihnen bei deren Bewältigung helfen. Python eignet sich gut, da die Einstiegshürde relativ niedrig ist und die Wartung vergleichsweise einfach ist.

In Hinblick auf den Continuous-Delivery-(CD-)Teil von DevOps können fundierte Kenntnisse in Go (Golang) hilfreich sein. Go ist eine weit verbreitete Sprache in der Welt von Cloud Native, und viele Anwendungen sind in Go geschrieben. Dies ist ein weiterer Schritt, der über Python hinausgeht, da Sie hier tiefer in das System eintauchen können.

© Rheinwerk Verlag, Bonn 2024

Versionsverwaltung mit Git

Bei der Versionsverwaltung ist die Auswahl für das Lernen eines Tools sehr einfach: Man braucht eigentlich nur Git. Andere Tools sind nahezu irrelevant. Zwar findet man an der einen oder anderen Stelle noch Subversion, aber das sind meist eher ältere Systeme, die ohnehin ersetzt werden müssen. Andere moderne Alternativen zu Git (wie Mercurial) findet man eher selten und sie sind im Großen und Ganzen vernachlässigbar.

Ihre Kenntnisse in Git sollten nicht nur die grundlegenden Befehle umfassen, sondern auch die Fähigkeit, Probleme zu beheben, wenn sie auftreten. Ein guter Indikator dafür, dass Sie Git effektiv nutzen, ist, dass Sie in der Lage sind, Probleme zu diagnostizieren und zu lösen, ohne dass ein erneutes Klonen des Repositorys erforderlich ist. Stattdessen sollten Sie in der Lage sein, den fehlerhaften Zustand gezielt zu korrigieren.

Darüber hinaus sind Kenntnisse verschiedener Workflows von Bedeutung. Es reicht nicht aus, die Git-Befehle zu kennen; Sie sollten auch wissen, wann welcher Workflow am besten geeignet ist und wann nicht. Das ist besonders für alle relevant, die mit Code arbeiten – sei es mit dem Quellcode der Anwendung oder mit den Konfigurationsdateien in IaC-Setups.

Wenn es um das Hosting von Git-Repositorys geht, gibt es mittlerweile relativ wenige Unterschiede zwischen den reinen Git-Repository-Hosting-Plattformen wie GitHub und GitLab – zumindest in Bezug auf die Verwaltung des Quellcodes. Die Struktur und die Organisation dieser Plattformen mögen sich unterscheiden, aber die grundlegenden Funktionen sind ähnlich.

Zusammenfassend sollten folgende Kenntnisse im Umgang von Git bekannt sein:

- das Erstellen und Pushen von Commits
- das Erstellen und Mergen von Branches
- die Behebung von Merge-Konflikten
- das Arbeiten mit einem und mehreren Remote-Repositorys
- das Beitragen zu »fremden« Repositorys
- das Rebasing von Änderungen auf Feature-Branches
- das sichere Verstehen und Anwenden von diversen Workflows

Container

Es ist äußerst spannend und wichtig, sich mit dem Thema Container vertraut zu machen. Ein erster Gedanke könnte sein: »Nun ja, es geht nur um Docker, oder?«

Container und Docker werden oft miteinander gleichgesetzt und verwechselt. Tatsächlich gibt es einige Unterschiede und Besonderheiten, bei denen auch ich immer wieder ins Stolpern komme.

Persönliches Exemplar für Thomas Bartholomäus

Es ist entscheidend, dass Sie folgende Konzepte verstanden haben und die Unterschiede kennen:

- Open Container Initiative (OCI)
- Container Runtime Interface (CRI)
- Container Engines

Es ist erstaunlich, wie begrenzt oft das Verständnis dieser Konzepte ist – selbst in Umgebungen, die Container produktiv einsetzen. Hand in Hand damit geht auch das Erstellen von Container-Images und die damit verbundenen Sicherheitsaspekte: Sie sollten verstehen, was Layer in Container-Images sind, wie Sie sie schlank halten können und wie Sie Sicherheitslücken vermeiden.

Die allgemeinen Konzepte, die auch in Kubernetes wiederzufinden sind, werden bereits auf der Container-Ebene sichtbar. Dies beinhaltet das Handling von Stateless Containern, die Verwaltung von Storage und damit verbundene Aspekte.

Es gibt im Wesentlichen zwei Anwendungsfälle zu betrachten: Container für CI/CD-Pipelines und Container für produktive Deployments. Obwohl die Grundprinzipien ähnlich sind, haben sie unterschiedliche Anforderungen, insbesondere in Bezug auf die Lebensdauer.

Es ist also entscheidend, die Grundlagen von Container-Infrastrukturen zu erlernen. Dies umfasst nicht nur Docker, sondern auch andere Tools wie Podman, um die Konzepte zu verstehen und anzuwenden. Ohne dieses Wissen braucht man sich gar nicht erst Kubernetes anzuschauen.

CI/CD mit GitHub, GitLab und Jenkins

Wenn Sie die CI/CD-Landschaft betrachten, gibt es besonders drei Plattformen, die Sie im Auge behalten sollten: *GitHub Actions*, *GitLab CI* und *Jenkins*. Natürlich gibt es noch viele andere, wie *TeamCity*, *Bamboo* und *Travis CI*. Mit zumindest einem Tool sollten Sie sich gut auskennen.

Jenkins wird hier eher aus historischen Gründen aufgeführt. Wenn Sie neu anfangen und wenig Erfahrung mit allen drei Systemen haben, sollten Sie sich eher GitHub Actions und/oder GitLab CI genauer anschauen. Jenkins befindet sich im Abwärtstrend, und viele Unternehmen migrieren davon weg, wie ich bereits in Abschnitt 6.6.1 beschrieben habe.

Für den Einstieg sollten Sie also einen Blick auf GitHub Actions und GitLab CI werfen. Beide Plattformen veröffentlichen regelmäßig Verbesserungen, mit denen Abläufe beschleunigt und optimiert werden.

Da sowohl GitHub als auch GitLab DevOps-Plattformen sind, sollten Sie sich nicht nur auf das Schreiben effizienter Pipelines beschränken, sondern einen ganzheitlichen Ansatz verfolgen, der zeigt, wie alles zusammenhängt. Dazu gehören Themen wie die

© Rheinwerk Verlag, Bonn 2024

Wiederverwendung von Pipelines, die Umsetzung von Compliance-Anforderungen und die Integration der Pipeline in Bereiche wie Deployment und Monitoring.

Ein weiterer wichtiger Punkt ist ein vertieftes Verständnis für Sicherheit. Dies gilt auf zwei Ebenen. Zum einen geht es um die Sicherheit des Pipeline-Betriebs und der Infrastruktur, die im Wesentlichen durch die verwendete Plattform gewährleistet ist. Aus Sicht der Anwendungsentwicklung sind die Security-Features zur Überprüfung und Analyse der Anwendungen von praktischer Bedeutung, wie in Abschnitt 11.4 behandelt wurde.

Wenn Sie sich Stellenausschreibungen ansehen, werden Sie feststellen, dass genau diese Fähigkeiten oft gefragt sind. Das ist nicht überraschend, da sie ein integraler Bestandteil der technischen Umsetzung von DevOps sind, insbesondere auf technischer Ebene.

Infrastructure as Code mit Terraform

Infrastructure as Code (*IaC*) sehe ich als optionales Thema an. Die bisherigen Punkte, die ich genannt habe, konzentrieren sich auf den Development-Lifecycle, der sich recht wenig für das Fundament interessiert, auf dem die Anwendungen laufen. Da in der Cloud-Native-Welt ständig neue Infrastrukturen entstehen, sollten Sie jedoch auch bei diesem Thema Grundkenntnisse mitbringen.

Obwohl es verschiedene IaC-Tools gibt, wie in Abschnitt 9.5.1 beleuchtet, konzentriert sich alles auf Terraform. Um Terraform und die Cloud sinnvoll nutzen zu können, sind Linux-Basiskenntnisse, Netzwerk-Kenntnisse sowie umfangreiche Erfahrungen im Umgang mit CI/CD-Tools hilfreich und auch notwendig.

Die Cloud

Es gibt viele verschiedene Cloud-Provider, und alle haben ihre Vor- und Nachteile. Die bekanntesten und verbreitetsten Public Clouds sind eindeutig *Amazon Web Services* (AWS), *Microsoft Azure* sowie *Google Cloud Platform* (GCP). Ich bin kein großer Fan der Behauptung, dass man alle Funktionen von einem oder gar von allen Cloud-Providern kennen muss.

Die Funktionen ändern sich ständig: Es kommen immer wieder neue Services dazu, und andere werden wieder abgekündigt. Echte Multi-Cloud-Setups werden zwar von vielen Firmen gewollt, es wird aber faktisch nie richtig umgesetzt, da ein solches Setup aufwendig und teuer ist.

Am Ende werden häufig ohnehin mehr virtuelle Maschinen in der Cloud automatisiert aufgesetzt und mit Software bestückt, wenn nicht ohnehin Kubernetes verwendet wird. Es gibt zwar einige Unterschiede zwischen den Cloud-Providern, doch am Ende kommt man mit allen ans Ziel.

Persönliches Exemplar für Thomas Bartholomäus

Spannender ist es, sich anzuschauen, welche Managed Services man nutzen kann – und von welchen man lieber die Finger lassen sollte. Das Aufsetzen eines echten hochverfügbaren Datenbankclusters ist ziemlich aufwendig und erfordert sehr tiefe Datenbankkenntnisse. Hier kann es sich also anbieten, den Dienst einer Cloud zu verwenden, aber die Nachteile dieser Entscheidung sollten ebenfalls bekannt sein. Teilweise fehlen dann nämlich Migrationspfade bei Versions-Upgrades bei den angesprochenen Datenbank-Services.

Für die Cloud gilt also immer: Sie sollten wissen, was man nutzen sollte und was nicht. Manchmal ist es in Ordnung, in den Vendor Lock-In zu gehen, manchmal nicht.

Konfigurationsmanagement mit Ansible (oder Puppet)

Nachdem idealerweise die Infrastruktur auf einem Cloud-System mithilfe von IaC-Tools wie Terraform bereitgestellt wurde, ist es nun wichtig, die Fähigkeiten zu haben, diese Infrastruktur mit Software auszustatten. Kubernetes ist nicht überall im Einsatz, und oft werden klassische virtuelle Maschinen oder sogar physische Hardware verwendet. In solchen Fällen kommen Konfigurationsmanagement-Tools wie Puppet oder Ansible in der Toolchain zum Einsatz.

Falls Sie noch keine Erfahrung mit Konfigurationsmanagement-Tools haben, empfehle ich Ihnen, Ansible zu lernen. Die Lernkurve ist relativ flach und der Einstieg ist recht einfach. Linux-Kenntnisse sind hier ebenfalls erforderlich, um die Einstiegshürde niedrig zu halten.

Natürlich wird auch die Arbeit mit Ansible komplexer, je komplexer die Probleme sind, die gelöst werden müssen. Obwohl Puppet in vielen Unternehmen eingesetzt wird, würde ich es für Anfänger eher in den Hintergrund stellen, da Ansible sowohl wichtiger als auch einfacher ist.

Kubernetes

In den vorherigen Abschnitten wurden Container-Technologien angesprochen, aber auch Kubernetes wurde kurz angerissen. Kubernetes ist selbst sehr umfangreich; hinzu gesellen sich zudem noch sehr viel mehr Technologien aus dem Kubernetes-Toolstack. Dabei gibt es zwei Blickwinkel zu betrachten: den Betrieb von Kubernetes-Clustern sowie die Nutzung von Kubernetes-Clustern nicht nur für produktive Umgebungen.

Der Weg zu Kubernetes sollte nicht direkt mit Kubernetes beginnen. Das kann nämlich sehr schnell frustrierend werden! Nachdem Sie die Konzepte von Containern und den Bau der Images verstanden haben und diese auch anwenden können, kann es ratsam sein, erst mal einen Zwischenstopp bei *Docker Compose* einzulegen und sich mit der Orchestrierung von Containern zu beschäftigen. Ein weiterer Zwischenschritt kann zudem *Docker Swarm* sein, was einen Schritt weiter als Docker Compose geht.

© Rheinwerk Verlag, Bonn 2024

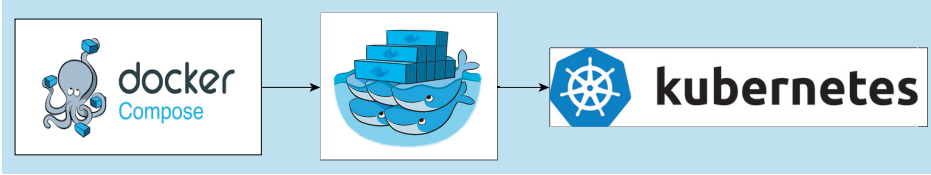

Abbildung 14.6 Docker Compose, Docker Swarm und Kubernetes

Mit Docker Compose lassen sich mehrere Container starten, die miteinander »reden« können. Die Anforderungen an die Infrastruktur und die Kenntnisse, die dafür benötigt werden, sind die Vorstufe zu einem »echten« Cluster-Setup.

Durch den Einsatz von Docker Compose erkennt man schnell, wie Container-Orchestrierungs-Tools funktionieren. Dazu gehören Punkte, die für den Betrieb der Container relevant sind, wie Health-Checks, das Mounten von Volumes und die generelle Kurzlebigkeit von Containern, die immer wieder deployt werden.

Wenn Sie sich eingehender mit Kubernetes beschäftigen, müssen Sie sich fast zwangsweise spezialisieren: Interessieren Sie sich eher für den Op- oder für den Dev-Teil, also für den Einsatz der Container-Orchestrierung im Betrieb oder für den Bau von Containern bei der Entwicklung?

Der Ops-Teil und besonders die Kubernetes-Administration liegen meist in der Hand der Spezialisten für das Plattform-Engineering. Dies trifft primär bei größeren Firmen und größeren Clustern zu, da dies ein Vollzeitjob ist.

Es macht allerdings einen großen Unterschied, ob ein Managed Kubernetes aus der Public Cloud verwendet wird oder ob ein Kubernetes-Cluster in der eigenen Infrastruktur selbst verwaltet wird, etwa über `kubeadm`. Letzteres ist deutlich aufwendiger in der Pflege.

K3s als Spielwiese

Wenn Sie den Umgang mit Kubernetes auf der Administrationsseite lernen möchten, empfehle ich Ihnen K3s (*https://k3s.io/*). Das ist eine leichtgewichtige Kubernetes-Distribution, die sich auch auf »schmalbrüstigen« Geräten installieren und verwalten lässt. Darauf kann man gut das Deployment von Anwendungen üben. So habe ich in den vergangenen Jahren meine Kubernetes-Kenntnisse aufgebaut.

Ebenfalls weitverbreitet ist die Container-Orchestierungsplattform *OpenShift* von Red Hat. Die findet man primär im deutschen Raum vielfach, da hier häufiger auf OpenShift als auf »normales« Kubernetes in eigenen Rechenzentren gesetzt wird.

Wenn Sie eher aus der Entwicklungsecke kommen und sich nicht für die Administration und die Skalierung von Kubernetes interessieren, dann ist die Rolle von Kubernetes in der CI/CD-Pipeline wesentlich spannender.

Persönliches Exemplar für Thomas Bartholomäus

Idealerweise wissen Sie, wie man auf einen Kubernetes-Cluster Anwendungen deployt und wie Configmaps, Services, Ingress, Secrets und so weiter konfiguriert und genutzt werden. Fangen Sie zu Beginn mit einfachen Use-Cases an, um Anwendungen zu deployen und sich dabei mit den ganzen Funktionen vertraut zu machen. Dabei müssen Sie zwangsläufig etwas debuggen, wobei Sie sehr viel lernen.

Die Lernkurve von Kubernetes ist sehr, sehr steil. Gute Linux- und Container-System-Kenntnisse sind hier unumgänglich. Schnell landet man auch auf dem zweiten Layer an Tools und Techniken aus dem Kubernetes-Umfeld, die das Ganze verkomplizieren – vor allem zu Beginn. Das sind vorwiegend Tools wie *Helm* oder *Kustomize*. Danach tauchen auch schon GitOps-Tools wie *ArgoCD* oder *Flux* auf.

Monitoring mit Prometheus und Grafana

Das Thema Monitoring ist grundsätzlich eine Wissenschaft für sich. Wenn Sie von der Dev-Seite kommen, sollten hier grundlegende Kenntnisse reichen. Vielfach findet man klassische Monitoring-Tools wie *Icinga* oder *Check MK*.

In modernen Infrastrukturen ist es hilfreich, die Grundkonzepte von *Prometheus* zu kennen und nutzen zu können. Dazu gehört auch, dass Sie wissen, wie man die Standard-Metriken nutzen kann und wie man bei Bedarf eigene Metriken bereitstellen kann.

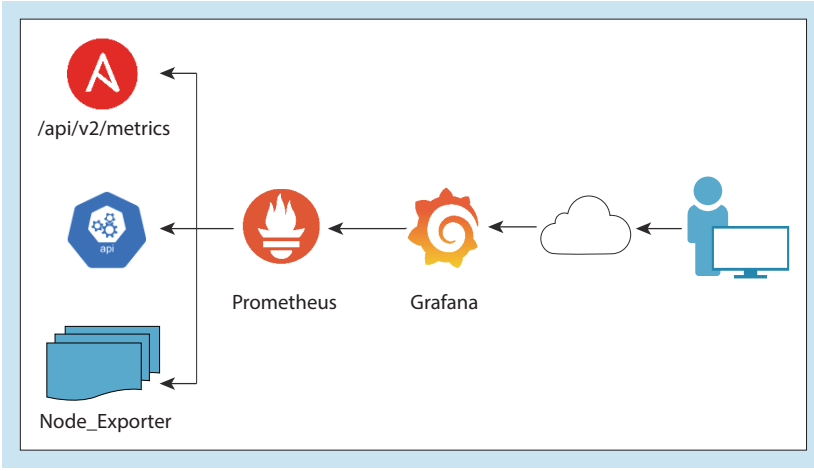

Abbildung 14.7 Prometheus und Grafana

Zur Visualisierung hilft Grafana schnell weiter, da Sie damit bei flacher Lernkurve schnell einfache Dashboards anlegen können. Besonders wichtig ist das Monitoring von komplexen Container-Infrastruktnre, also die Anbindung an Kubernetes. Ihnen sollte also bekannt sein, wie man die Metriken etwa von Kubernetes-Deployments über die Labels definiert und abholen lässt.

© Rheinwerk Verlag, Bonn 2024

Observability für Kubernetes

Wenn Sie dies vertiefen möchten, kann ich Ihnen den Kubernetes-Observability-Workshop meines Kollegen Michael Friedrich empfehlen. In diesem 3,5-stündigen (englischsprachigen) Workshop geht er auf genau diese Aspekte ein:

https://o11y.love/learning-resources/

Als Alternative zu Prometheus findet man auch *Influx* – auch in Kombination mit Grafana. Teilweise sind auch Influx und Prometheus mit verschiedenen Use-Cases im Parallelbetrieb zu finden.

Was sonst noch fehlt

Kommen wir nun zum Ende eines möglichen Lernpfades. Es gibt noch sehr viel mehr. Am wichtigsten ist, dass die Grundlagen solide sind. Hervorragende Linux-Kenntnisse, sowohl vom System selbst als auch auf der Kommandozeile, Grundlagen von Netzwerktechnik sowie Git helfen bei allen Tools weiter.

Und vergessen Sie nicht: Jedes Tool ist zu einem gewissen Maße vergänglich. In ein paar Jahren kann eine bevorzugte Toolchain schon wieder ganz anders aussehen. Daher sind solide technische Grundlagen sehr wichtig, denn so sind auch andere Themen wie das Log-Management oder das Secret-Management mit einfachen Handgriffen lösbar. Das gilt zumindest dann, wenn man nicht immer wieder die Grundlagen lernen muss.

Reflexion

Mit »DevOps« im Jobtitel oder in einer Job-Bezeichnung kann man immer für Verwirrung sorgen. Wenn Sie vielleicht selbst in einer Rolle sind, um Einfluss auf die Verwendung des Begriffs zu nehmen, sorgen Sie bitte zum Wohle aller dafür, dass er bedacht verwendet wird. Das erspart dann einige Verwirrung. Gleichzeitig können Sie die Verwendung des Begriffs auch gut als Filtermechanismus nutzen, um zu schauen, ob Kandidaten und Kandidatinnen in einem Vorstellungsgespräch wissen, woran Sie arbeiten.

Wenn Sie sich mehr für die Technik interessieren, dann dürfte die Fülle aller Tools in diesem Lernpfad Sie vermutlich überfordern. Sie können schlicht nicht in jedem Bereich Experte werden. Sobald Sie in dem einen oder anderen Tool ausgezeichnete Kenntnisse besitzen und sich dann dem nächsten Tool widmen, veralten die verschiedenen Kenntnisse auch schon wieder.

Sehen Sie also zu, dass Sie sich möglichst breit aufstellen. Die technischen Grundlagen sind sehr wichtig – diese ändern sich nicht. Genauso sind grundlegende Konzepte wichtiger als die tatsächliche Implementierung.

Persönliches Exemplar für Thomas Bartholomäus

Wenn Sie »DevOps« in einer Job-Bezeichnung sehen, auf die Sie sich bewerben, fragen Sie doch mal vorsichtig nach, was die Firma unter DevOps versteht. Merkwürdige Erklärungen könnten schnell Red Flags zum Vorschein bringen, an denen Sie erkennen, dass Sie besser eine andere Stelle suchen sollten.

14.4 Fazit

Die DevOps-Welt bewegt sich auch nach über 10 Jahren nach der Geburt des Begriffes »DevOps« weiter. Das ist gut und nötig, denn es muss immer wieder iteriert, angepasst und beobachtet werden, was gebraucht wird und was vielleicht auch nicht.

© Rheinwerk Verlag, Bonn 2024

Index

A

B

C

Persönliches Exemplar für Thomas Bartholomäus

D

E

F

G

© Rheinwerk Verlag, Bonn 2024

H

I

J

K

L

M

Persönliches Exemplar für Thomas Bartholomäus

N

O

P

Q

R

S

© Rheinwerk Verlag, Bonn 2024

T

U

V

W

Z

Persönliches Exemplar für Thomas Bartholomäus

Die Serviceseiten

Im Folgenden finden Sie Hinweise, wie Sie Kontakt zu uns aufnehmen können.

Lob und Tadel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Wenn Sie zufrieden waren, empfehlen Sie das Buch bitte weiter. Wenn Sie meinen, es gebe doch etwas zu verbessern, schreiben Sie direkt an den Lektor dieses Buches: *christoph.meister@rheinwerk-verlag.de*. Wir freuen uns über jeden Verbesserungsvorschlag, aber über ein Lob freuen wir uns natürlich auch!

Auch auf unserer Webkatalogseite zu diesem Buch haben Sie die Möglichkeit, Ihr Feedback an uns zu senden oder Ihre Leseerfahrung per Facebook, Twitter oder E-Mail mit anderen zu teilen. Folgen Sie einfach diesem Link: *https://www.rheinwerk-verlag.de/5562*.

Zusatzmaterialien

Falls Zusatzmaterialien (Beispielcode, Übungsmaterial, Listen usw.) für dieses Buch verfügbar sind, finden Sie sie in Ihrer Online-Bibliothek sowie auf der Webkatalogseite zu diesem Buch: *https://www.rheinwerk-verlag.de/5562*. Wenn uns sinnentstellende Tippfehler oder inhaltliche Mängel bekannt werden, stellen wir Ihnen dort auch eine Liste mit Korrekturen zur Verfügung.

Technische Probleme

Im Falle von technischen Schwierigkeiten mit dem E-Book oder Ihrem E-Book-Konto beim Rheinwerk Verlag steht Ihnen gerne unser Leserservice zur Verfügung: *ebooks@rheinwerk-verlag.de*.

Über uns und unser Programm

Informationen zu unserem Verlag und weitere Kontaktmöglichkeiten bieten wir Ihnen auf unserer Verlagswebsite *https://www.rheinwerk-verlag.de*. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher und E-Books schnell und komfortabel bestellen. Alle Buchbestellungen sind für Sie versandkostenfrei.

Rechtliche Hinweise

In diesem Abschnitt finden Sie die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen für dieses E-Book.

Copyright-Vermerk

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen bei den Autor*innen und beim Rheinwerk Verlag, insbesondere das Recht der Vervielfältigung und Verbreitung, sei es in gedruckter oder in elektronischer Form.
© Rheinwerk Verlag GmbH, Bonn 2024

Nutzungs- und Verwertungsrechte

Sie sind berechtigt, dieses E-Book ausschließlich für persönliche Zwecke zu nutzen. Insbesondere sind Sie berechtigt, das E-Book für Ihren eigenen Gebrauch auszudrucken oder eine Kopie herzustellen, sofern Sie diese Kopie auf einem von Ihnen alleine und persönlich genutzten Endgerät speichern. Zu anderen oder weitergehenden Nutzungen und Verwertungen sind Sie nicht berechtigt.

So ist es insbesondere unzulässig, eine elektronische oder gedruckte Kopie an Dritte weiterzugeben. Unzulässig und nicht erlaubt ist des Weiteren, das E-Book im Internet, in Intranets oder auf andere Weise zu verbreiten oder Dritten zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und jegliche den persönlichen Gebrauch übersteigende Vervielfältigung des E-Books ist ausdrücklich untersagt. Das vorstehend Gesagte gilt nicht nur für das E-Book insgesamt, sondern auch für seine Teile (z. B. Grafiken, Fotos, Tabellen, Textabschnitte).

Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte dürfen aus dem E-Book nicht entfernt werden, auch nicht das digitale Wasserzeichen.

Die automatisierte Analyse des Werkes, um daraus Informationen insbesondere über Muster, Trends und Korrelationen gemäß § 44b UrhG (»Text und Data Mining«) zu gewinnen, ist untersagt.

Digitales Wasserzeichen

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf. Wenn Sie diese Person nicht sind, liegt ein Verstoß gegen das Urheberrecht vor, und wir bitten Sie freundlich, das E-Book nicht weiter zu nutzen und uns diesen Verstoß zu melden. Eine kurze E-Mail an *service@rheinwerk-verlag.de* reicht schon. Vielen Dank!

Markenschutz

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Haftungsausschluss

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor*innen, Herausgeber*innen oder Übersetzer*innen für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

Über den Autor

Sujeevan Vijayakumaran arbeitet als Senior Solutions Architect bei GitLab. Seit einigen Jahren unterstützt er große Konzerne aus Deutschland, Österreich und der Schweiz beim Ein- und Umstieg in die DevOps-Welt. Er steckt hinter dem Podcast TILpod und hält gerne Vorträge auf Open-Source-Konferenzen – nicht nur zu technischen Themen, sondern auch zu gutem Teamwork, effizienter Kommunikation und allem, was sonst zur DevOps-Kultur gehört.