DevOps

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

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.

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*.

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.

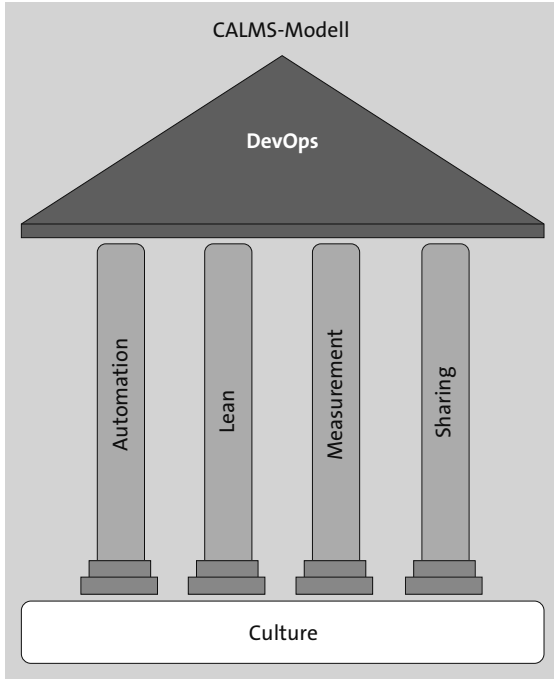

Abbildung 2.1 Das CALMS-Modell

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.

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.

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

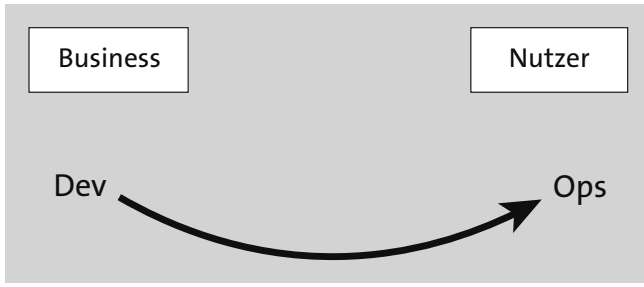

Abbildung 2.2 The First Way

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?

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.

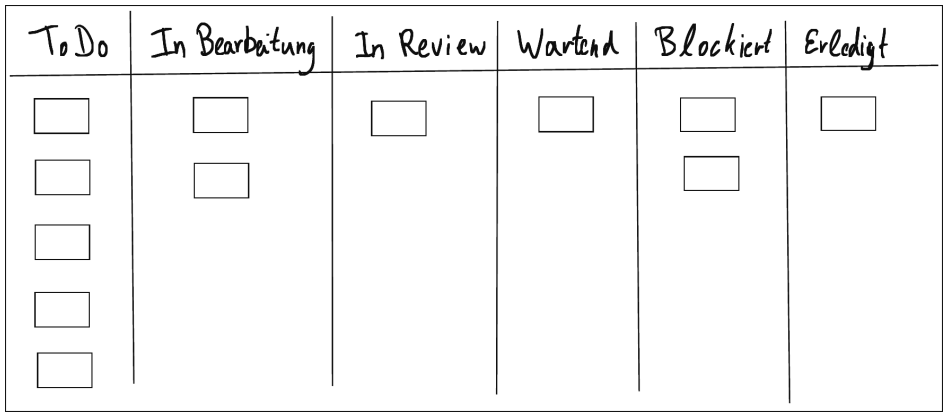

Abbildung 2.3 Dieses beispielhafte und vereinfachte Kanban-Board zeigt übersichtlich die Arbeitspakete in ihren verschiedenen Status an.

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.

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,

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

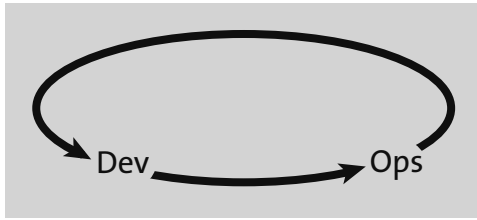

Abbildung 2.4 The Second Way

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.

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.

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

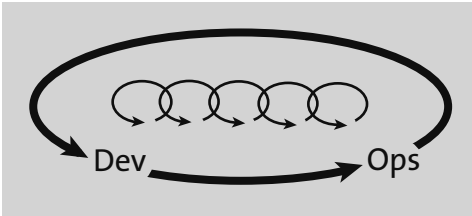

Abbildung 2.5 The Third Way

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-

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

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.

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.

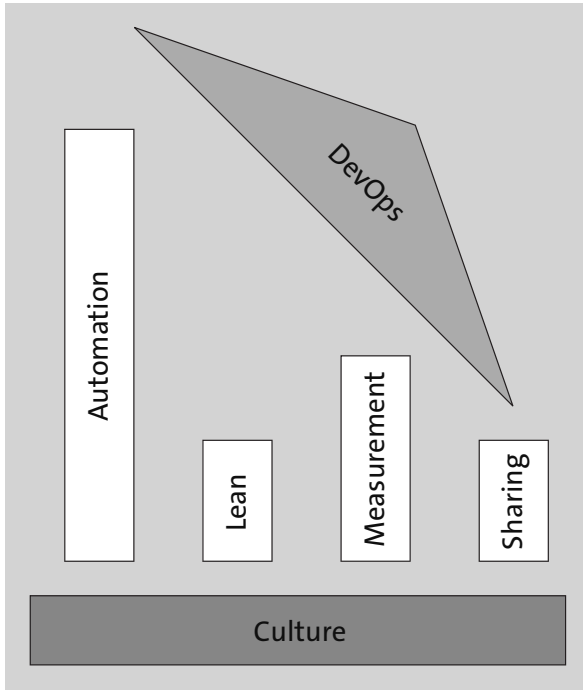

Abbildung 2.6 Eine starke Automatisierung in DevOps ist zwar wichtig – aber definitiv nicht das Einzige, denn sonst führt das Ungleichgewicht zu weiteren Problemen.

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.

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).

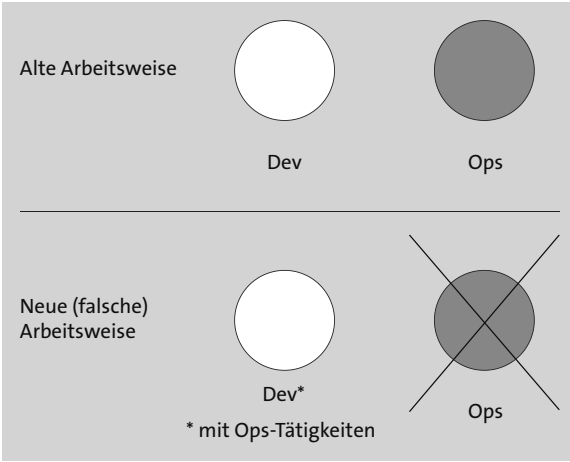

Abbildung 2.7 DevOps heißt eben nicht, dass die Entwickler nun die Betriebstätigkeiten übernehmen und somit Admins völlig unnötig werden.

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«.

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.

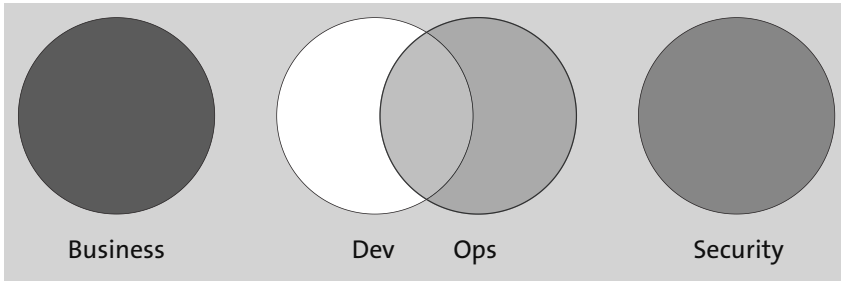

Abbildung 2.8 Die Säulen zwischen Dev und Ops wurden zwar niedergerissen, aber was ist mit den übrigen Teams?

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

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

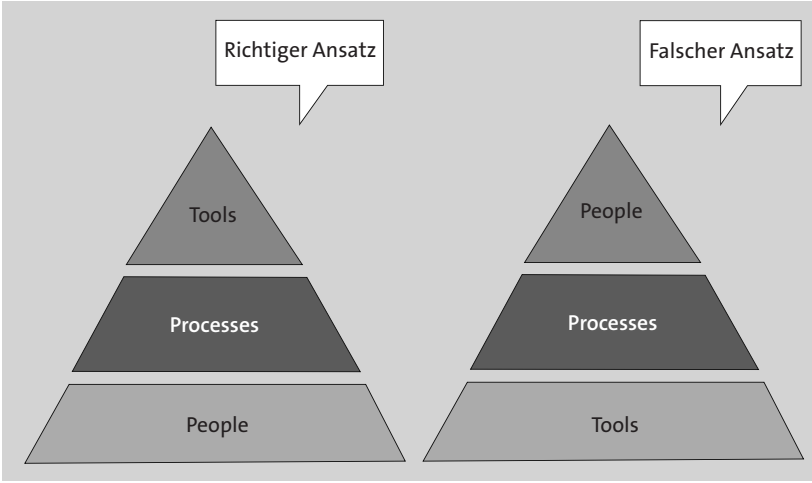

Abbildung 2.9 Die Menschen im Team bilden die wichtigste Grundlage. Erst danach kommen die Prozesse und die Tools – und eben nicht umgekehrt!

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.

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.

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

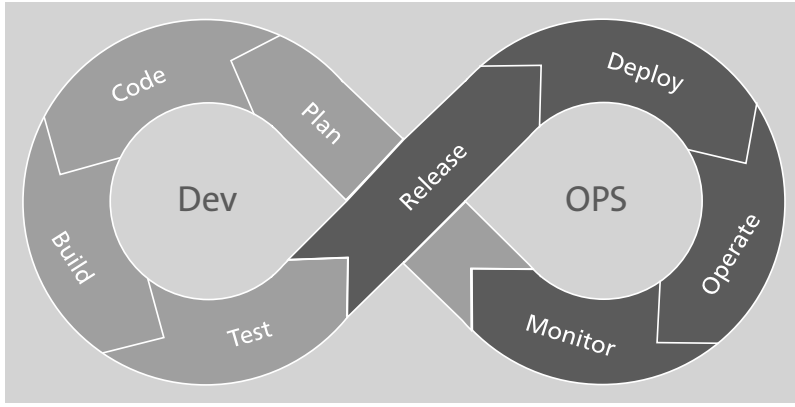

Abbildung 2.10 DevOps ist ein unendlicher Vorgang.

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.

- **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.

- **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.

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.

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.

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.

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.

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-

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.

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.

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.

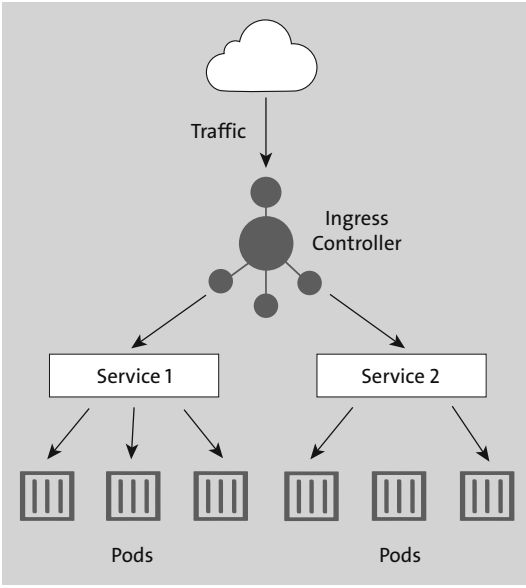

Abbildung 8.11 Dieses Ablaufdiagramm zeigt stark vereinfacht, wohin der Traffic im Kubernetes-Cluster fließt.

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

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).

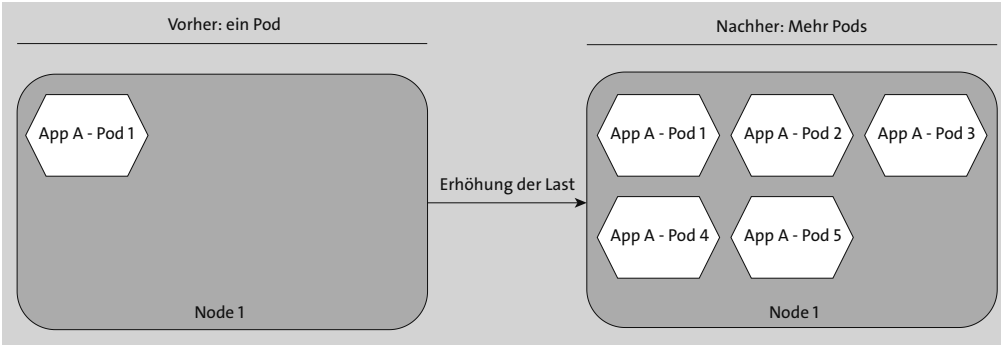

Abbildung 8.12 Bei höherer Last wird die Anzahl der Pods erhöht und später wieder reduziert.

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.

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.

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,

- 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.

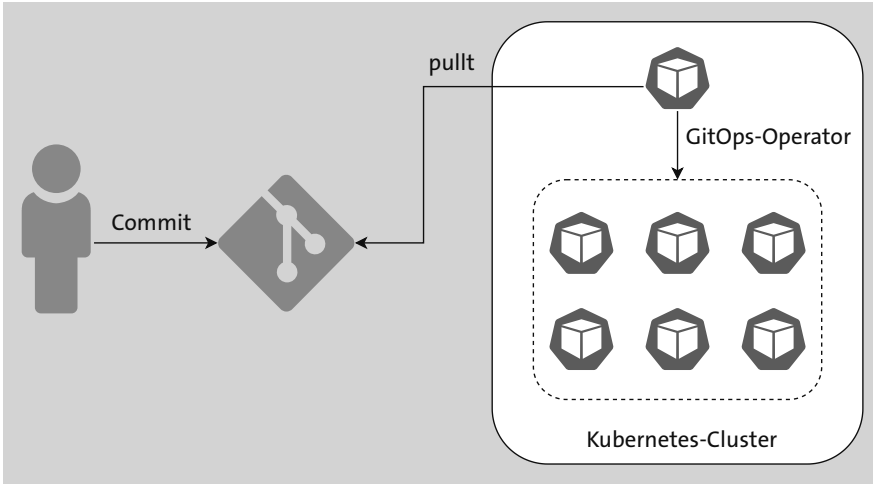

Abbildung 8.13 Das GitOps-Pattern: Der GitOps-Operator arbeitet im Kubernetes-Cluster. Er holt sich die Konfiguration aus einem Git-Repository und wendet sie innerhalb des Clusters an.

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.

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.

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.

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.

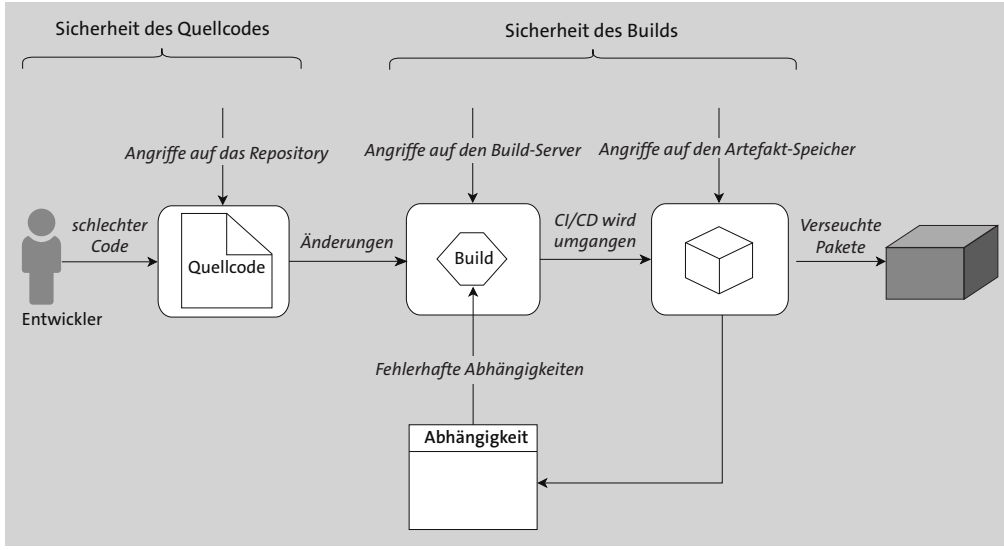

Abbildung 11.9 In der Software Supply-Chain gibt es an vielen Stellen Angriffsvektoren.

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-

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

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.

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?

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

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