KAPITEL 9

Die Grundlagen für unsere Deployment-Pipeline legen

Um einen schnellen und zuverlässigen Flow von Dev nach Ops zu schaffen, müssen wir sicherstellen, dass wir an jeder Stelle unserer Wertkette produktivähnliche Umgebungen nutzen. Zudem müssen diese Umgebungen automatisch erstellt werden – idealerweise auf Anforderung über Skripte und Konfigurationsinformationen, die in der Versionsverwaltung abgelegt sind. Das Ganze sollte als Self-Service aufgebaut sein, sodass Operations keinerlei manuelle Arbeit damit hat. Wir wollen so sicherstellen, dass wir die gesamte Produktivumgebung immer wieder neu aus den Informationen in der Versionsverwaltung aufbauen können.

Die Enterprise Data Warehouse Story (2009)

Allzu häufig erkennen wir erst dann, wie sich unsere Anwendungen in so etwas wie produktivähnlichen Umgebungen verhalten, wenn das Produktiv-Deployment durchgeführt wird – viel zu spät, um Probleme zu beheben, ohne die Kunden zu beeinträchtigen. Ein schönes Beispiel für das Spektrum an Problemen, das durch inkonsistent aufgebaute Anwendungen und Umgebungen entstehen kann, ist das Enterprise Data Warehouse Program, das 2009 von Em Campbell-Pretty bei einem großen australischen Telekommunikationsunternehmen geleitet wurde. Campbell-Pretty wurde Geschäftsführerin und Business-Sponsor für dieses 200-Millionen-Dollar-Programm und übernahm damit die Verantwortung für alle strategischen Ziele, die auf dieser Plattform aufbauten. [1]

In ihrer Präsentation auf dem *DevOps Enterprise Summit 2014* beschrieb Campbell-Pretty:

> *Damals gab es zehn Workstreams, die alle Wasserfall-Prozesse einsetzten, und alle zehn Streams lagen deutlich hinter ihrem Zeitplan. Nur einer der zehn Streams hatte pünktlich die Phase des User Acceptance Testing [UAT]*

1 Campbell-Pretty, »DOES14 – Em Campbell-Pretty – How a Business Exec Led Agile, Lead, CI/CD«

erreicht, aber es dauerte weitere sechs Monate, um diese Phase auch erfolgreich abzuschließen, wobei die Ergebnisse ziemlich weit hinter den Erwartungen der Business-Abteilungen zurücklagen. Diese schlechte Leistung war der größte Antrieb für die Agile Transformation der Abteilung.[2]

Nachdem sie Agile fast ein Jahr im Einsatz hatten, sahen sie jedoch nur wenige Verbesserungen und erreichten immer noch nicht die notwendigen Ergebnisse.

Campbell-Pretty führte eine programmweite Retrospektive durch und fragte: »Was können wir angesichts der Erfahrungen aus dem letzten Release tun, um unsere Produktivität zu verdoppeln?«[3]

Während des Projekts gab es Nörgeleien über fehlendes Business-Engagement, während der Retrospektive jedoch schaffte es »Verbesserte Verfügbarkeit von Umgebungen« ganz nach oben auf die Liste.[4] Im Nachhinein war das offensichtlich – die Entwicklungsteams brauchten fertige Umgebungen, um mit ihrer Arbeit beginnen zu können, aber sie warteten oft bis zu acht Wochen darauf.

Also setzten sie ein neues Integration- und Build-Team auf, das dafür verantwortlich war, »Qualität in unsere Prozesse einzubauen, statt zu versuchen, sie nach dem Schaffen von Tatsachen zu untersuchen«.[5] Es wurde zunächst mit Datenbankadministratoren (DBAs) und Automatisierungsspezialisten besetzt, die die Aufgabe hatten, den Prozess zum Erstellen der Umgebungen zu automatisieren. Das Team machte schnell eine überraschende Entdeckung: Nur 50 % des Quellcodes in den Entwicklungs- und Testumgebungen stimmten mit dem überein, der in der Produktivumgebung lief.[6]

Campbell-Pretty: »Plötzlich verstanden wir, warum wir beim Deployen unseres Codes in neue Umgebungen so viele Probleme hatten. In jeder Umgebung behoben wir die Fehler, die Änderungen wanderten aber nicht zurück in die Versionsverwaltung.«[7]

Das Team führte ein sorgfältiges Reverse-Engineering aller Änderungen in den verschiedenen Umgebungen durch und führte sie wieder der Versionsverwaltung zu. Zudem automatisierten sie den Prozess zum Erzeugen von Umgebungen, sodass sie wiederholt und reproduzierbar Umgebungen aufsetzen konnten.

Campbell-Pretty beschrieb die Ergebnisse und merkte an: »Die Zeit, die zum Erstellen einer sauberen Umgebung notwendig war, reduzierte sich von acht Wochen auf einen Tag. Das war einer der Schlüsselfaktoren beim Erreichen unserer Ziele bezüg-

2 Campbell-Pretty, »DOES14 – Em Campbell-Pretty – How a Business Exec Led Agile, Lead, CI/CD«
3 Ebd.
4 Ebd.
5 Ebd.
6 Ebd.
7 Ebd.

lich der Durchlaufzeit, der Cost to Deliver und der Anzahl an Fehlern, die es in die Produktivumgebung schafften.«[8]

Diese Geschichte zeigt die Vielzahl an Problemen, die aufgrund von inkonsistent aufgesetzten Umgebungen und von Änderungen, die nicht zurück in die Versionsverwaltung gebracht werden, entstehen können.

Im Rest dieses Kapitels werden wir darüber sprechen, wie wir Mechanismen aufbauen, die es ermöglichen, Umgebungen auf Anforderung hin zu erstellen, eine Versionsverwaltung von jedem in der Wertkette nutzen zu lassen, die Infrastruktur leichter neu bauen als reparieren zu lassen und sicherzustellen, dass Entwickler ihren Code in jeder Phase des Softwareentwicklungs-Lebenszyklus in produktivähnlichen Umgebungen laufen lassen.

Dev-, Test- und Produktivumgebungen auf Anforderung erstellen können

Wie im Beispiel des Enterprise Data Warehouse weiter oben ist einer der Hauptgründe für chaotische, disruptive und manchmal sogar katastrophale Software-Releases, unsere Anwendung erstmals beim Release mit realistischen Load- und Produktivdatenmengen zu konfrontieren und ihr dortiges Verhalten zu beobachten.[9] In vielen Fällen haben Entwicklungsteams eventuell Testumgebungen in früheren Stadien des Projekts angefordert.

Aber wenn es bei Operations lange Durchlaufzeiten für das Bereitstellen von Testumgebungen gibt, erhalten die Teams sie eventuell nicht früh genug, um adäquate Tests durchführen zu können. Schlimmer noch: Die Testumgebungen sind häufig falsch konfiguriert oder unterscheiden sich so sehr von den Produktivumgebungen, dass wir in Letzteren trotzdem Probleme bekommen, obwohl wir vor dem Deployment Tests durchgeführt haben.

In diesem Schritt wollen wir, dass die Entwickler produktivähnliche Umgebungen auf ihren eigenen Arbeitsplatzrechnern laufen lassen, die per Self-Service auf Anforderung hin erstellt werden. So können Entwickler ihren Code als Teil ihrer täglichen Arbeit in produktivähnlichen Umgebungen laufen lassen und testen und so konstant Feedback zur Qualität ihrer Arbeit erhalten.

Statt die Spezifikationen der Produktivumgebung nur in einem Dokument oder auf einer Wiki-Seite abzulegen, schaffen wir einen allgemeinen Build-Mechanismus, der alle unsere Umgebungen baut – für Entwicklung, Test und Produktion. Da-

8 Ebd.

9 In diesem Zusammenhang ist eine *Umgebung* definiert als alles im Anwendungs-Stack außer der Anwendung – also auch Datenbanken, Betriebssysteme, Netzwerk, Virtualisierung und alle entsprechenden Konfigurationsdaten.

durch kann jeder produktivähnliche Umgebungen in Minuten erhalten, ohne ein Ticket aufmachen oder gar Wochen warten zu müssen.[10]

Dazu müssen wir das Erstellen unserer bekannten, guten Umgebungen definieren und automatisieren. So bekommen wir einen stabilen, sicheren und risikoarmen Status und können das Wissen der gesamten Firma darin einfließen lassen. Alle unsere Anforderungen sind integriert – nicht in Dokumenten oder als Wissen im Kopf von irgendjemandem, sondern kodifiziert in unseren automatisierten Umgebungs-Build-Prozessen.

Statt Operations die Umgebung manuell bauen und konfigurieren zu lassen, können wir die Automatisierung für einen Teil des Folgenden (oder für alles) nutzen:

- Eine virtualisierte Umgebung kopieren (zum Beispiel ein VMware-Image, ein Vagrant-Skript laufen lassen oder ein Amazon Machine Image File in EC2 starten).
- Einen automatisierten Prozess zum Erstellen von Umgebungen aufbauen, der von »Bare Metal« ausgeht (zum Beispiel ein PXE-Install aus einem Baseline-Image).
- »Infrastructure as Code«-Konfigurationsmanagement-Tools verwenden (zum Beispiel Puppet, Chef, Ansible, Salt oder CFEngine).
- Automatisierte Konfigurationstools für Betriebssysteme einsetzen (zum Beispiel Solaris Jumpstart, Red Hat Kickstart oder Debian Preseed).
- Eine Umgebung aus einem Satz virtueller Images oder Container zusammenstellen (zum Beispiel Docker oder Kubernetes).
- Eine neue Umgebung in einer öffentlichen Cloud (zum Beispiel Amazon Web Services, Google App Engine oder Microsoft Azure), einer privaten Cloud (zum Beispiel mit einem Stack, der auf Kubernetes basiert) oder einer anderen PaaS (*Platform as a Service*, wie zum Beispiel OpenStack oder Cloud Foundry) starten.

Weil wir sorgfältig alle Aspekte der Umgebung im Voraus definiert haben, können wir nicht nur neue Umgebungen schnell erstellen, sondern auch sicherstellen, dass diese Umgebungen stabil, zuverlässig, konsistent und sicher sind. Dadurch gewinnt jeder.

Operations profitiert davon, neue Umgebungen schnell erstellen zu können, weil das Automatisieren dieses Prozesses Konsistenz sicherstellt und langweilige, fehleranfällige manuelle Arbeit verringert. Und die Entwicklung profitiert davon, weil sie alle notwendigen Teile der Produktivumgebung auf ihren Arbeitsplatzrechnern nachbauen kann, um den Code zu bauen, auszuführen und zu testen. Dadurch können Entwickler viele Probleme finden und beheben – sogar schon in einem sehr

10 Die meisten Entwickler wollen ihren Code testen und müssen häufig ziemlich viel Aufwand treiben, um entsprechende Testumgebungen zu erhalten. Entwickler nutzen dann gern alte Testumgebungen aus früheren Projekten (häufig viele Jahre alt) oder fragen jemanden, der dafür bekannt ist, eine zu finden – häufig wollen sie auch gar nicht wissen, woher sie stammen, denn irgendjemand anderem wird nun ein Server fehlen.

frühen Stadium des Projekts statt erst bei den Integrationstests oder gar in der Produktion.

Indem die Entwickler eine Umgebung erhalten, über die sie die volle Kontrolle besitzen, ermöglichen wir es ihnen, Fehler schnell reproduzieren, ergründen und beheben zu können – sicher getrennt von produktiven Services und anderen gemeinsam genutzten Ressourcen. Sie können zudem mit Änderungen an den Umgebungen und am diese erzeugenden Infrastrukturcode (zum Beispiel Konfigurationsmanagementskripten) experimentieren und so mehr gemeinsames Wissen zwischen Entwicklung und Operations aufbauen.[11]

Ein Single Repository of Truth für das gesamte System schaffen

Mit dem vorherigen Schritt haben wir es ermöglicht, Entwicklungs-, Test- und Produktivumgebungen auf Anforderung hin erstellen zu können. Nun müssen wir noch sicherstellen, dass alle Komponenten unseres Softwaresystems mit einer Source of Truth konfiguriert und gemanagt werden können, die einer Versionskontrolle unterliegt.

Seit Jahrzehnten findet der umfassende Einsatz von Versionsverwaltungssystemen eine immer weitere Verbreitung als verpflichtende Praxis einzelner Entwickler und Entwicklungsteams.[12] Ein Versionsverwaltungssystem zeichnet die Änderungen an Dateien auf, die darin gespeichert werden.[13] Dabei kann es sich um Quellcode, Assets oder andere Dokumente handeln, die eventuell Teil eines Softwareentwicklungsprojekts sind. Wir nehmen Änderungen in Gruppen vor, die als Commits oder Revisionen bezeichnet werden. Jede Revision wird zusammen mit Metadaten wie dem Namen der ändernden Person und dem Änderungsdatum auf die eine oder andere Art und Weise im System gespeichert, sodass wir alte Versionen von Objekten im Repository auslesen, vergleichen, zusammenführen und committen können. Auch wird das Risiko verringert, weil wir so die Möglichkeit haben, Objekte im Produktivumfeld auf frühere Versionen zurückzusetzen.[14]

11 Idealerweise sollten wir Fehler finden, bevor die Integrationstests laufen und es im Testzyklus zu spät ist, den Entwicklern schnelles Feedback geben zu können. Ist das nicht möglich, haben wir vermutlich ein Problem mit der Architektur, das wir angehen müssen. Ein entscheidender Teil beim Schaffen einer Architektur für schnellen Flow und schnelles Feedback ist, unsere Systeme auf Testbarkeit hin zu entwerfen, um die Fähigkeit einzubauen, die meisten Fehler schon in einer nicht integrierten virtuellen Umgebung auf einem Entwicklerrechner zu finden.

12 Das erste Versionsverwaltungssystem war sehr wahrscheinlich UPDATE auf dem CDC6600 (1969). Später kamen SCCS (1972), CMS auf VMS (1978), RCS (1982) und so weiter hinzu. – »Version Control History«, Plastic SCM (Website)

13 Davis und Daniels, *Effective DevOps*, 37

14 Für manche erfüllt die Versionsverwaltung damit einige der ITIL-Konstrukte der *Definitive Media Library* (DML) und der *Configuration Management Database* (CMDB), in denen alles gelagert wird, das zum Wiederherstellen der Produktivumgebung notwendig ist.

Legen Entwickler alle Quell- und Konfigurationsdateien der Anwendung in der Versionsverwaltung ab, wird diese zum Single Repository of Truth und enthält genau den gewünschten Stand des Systems. Aber um dem Kunden Werte anzubieten, brauchen wir nicht nur den Code, sondern auch die richtigen Umgebungen, daher müssen sich diese ebenfalls in der Versionsverwaltung befinden. Mit anderen Worten: Die Versionsverwaltung wird von allen in unserer Wertkette genutzt, einschließlich QA, Operations, Infosec und (natürlich) der Entwicklung.

Bringen wir alle Produktionsartefakte in der Versionsverwaltung unter, können wir mit dem Repository wiederholt und zuverlässig alle Komponenten unseres funktionierenden Softwaresystems reproduzieren – dazu gehören die Anwendungen und die Produktivumgebung, aber auch unsere Pre-Production-Umgebungen.

Um sicherzustellen, dass wir die Produktivservices wiederholt und planbar (idealerweise auch schnell) wiederherstellen können, auch wenn ein katastrophales Ereignis eingetreten ist, müssen wir die folgenden Assets in unser gemeinsam genutztes Versionsverwaltungs-Repository einchecken:

- Den gesamten Anwendungscode und die Abhängigkeiten (zum Beispiel Bibliotheken, statische Inhalte usw.).
- Alle Skripte zum Erstellen von Datenbankschemata, Anwendungsreferenzdaten usw.
- Alle Tools und Artefakte zum Erstellen von Umgebungen, die im vorherigen Schritt beschrieben wurden (zum Beispiel VMware- oder AMI-Images, Puppet-, Chef- oder Ansible-Rezepte usw.).
- Alle Dateien zum Erstellen von Containern (zum Beispiel Docker-, Rocket- oder Kubernetes-Definitions- oder Kompositionsdateien).
- Alle unterstützenden automatischen Tests und alle manuellen Testskripten.
- Alle Skripte, die Code-Packaging, Deployment, Datenbankmigration und Ausstatten von Umgebungen unterstützen.
- Alle Projektartefakte (zum Beispiel Anforderungsdokumente, Deployment-Prozeduren, Release-Hinweise usw.).
- Alle Cloud-Konfigurationsdateien (zum Beispiel AWS Cloudformation Templates, Microsoft Azure Stack DSC Files oder OpenStack HEAT).
- Alle anderen Skripte oder Konfigurationsinformationen, die erforderlich sind, um die Infrastruktur für diverse Services zu erstellen (zum Beispiel Enterprise Service Buses, Datenbankverwaltungssysteme, DNS-Zone-Dateien, Konfigurationsregeln für Firewalls und andere Netzwerkschnittstellen).[15]

15 Später werden wir in die Versionsverwaltung auch die ganze unterstützende Infrastruktur einchecken, zum Beispiel die automatisierten Testsuiten und unsere Pipelining-Infrastruktur für Continuous Integration und Deployment.

Wir können mehrere Repositories für die unterschiedlichen Arten von Objekten und Services nutzen, wenn diese entsprechend unserem Quellcode getaggt und bezeichnet werden. So kann man zum Beispiel große VM-Images, ISO-Dateien, kompilierte Binaries und so weiter in Artefakt-Repositories ablegen (zum Beispiel Nexus oder Artifactory). Alternativ lassen sie sich auch in Blob-Stores speichern (zum Beispiel Amazon S3 Buckets), Docker Images können in Docker-Registries untergebracht werden und so weiter. Wir werden zudem einen kryptografischen Hash dieser Objekte beim Bauen erstellen und ablegen und diesen dann beim Deployen überprüfen, um sicherzugehen, dass mit den Objekten kein Schindluder getrieben wurde.

Es reicht nicht aus, nur einen beliebigen Zustand der Produktivumgebung wiederherstellen zu können, es muss auch möglich sein, die gesamten Pre-Production- und Build-Prozesse wiederzuerschaffen. Konsequenterweise müssen wir in der Versionsverwaltung alles ablegen, was wir für unsere Build-Prozesse benötigen, auch unsere Tools (zum Beispiel Compiler und Testtools) und die Umgebungen, von denen sie abhängen.

Die Forschung zeigt die Wichtigkeit einer Versionsverwaltung. In den *State of DevOps Reports* von 2014 bis 2019, geleitet von der Koautorin Dr. Nicole Forsgren, war die Verwendung einer Versionsverwaltung für alle Produktivartefakte der deutlichste Parameter für die Leistung bei der Softwareauslieferung, was wiederum ein Indikator für die Firmenperformance darstellte.

Die Ergebnisse dieser Reports unterstreichen die so wichtige Rolle der Versionsverwaltung im Softwareentwicklungsprozess. Wenn alle Änderungen der Anwendung und der Umgebungen in der Versionsverwaltung dokumentiert sind, können wir nicht nur schnell herausfinden, welche Änderungen eventuell zu einem Problem beigetragen haben, sondern wir können auch in einen bekannten lauffähigen Zustand zurückkehren und schneller aus einer fehlerhaften Situation entkommen.

Aber warum hat der Einsatz einer Versionskontrolle für unsere Umgebungen einen so viel größeren Einfluss auf die Leistungen bei der Softwareauslieferung und der ganzen Firma als beim Quellcode?

Weil bei den Umgebungen fast immer um Größenordnungen mehr Konfigurationseinstellungen möglich sind als beim Code. Daher muss die Umgebung am ehesten unter Versionsverwaltung gestellt werden.[16]

Die Versionsverwaltung stellt zudem für jeden in der Wertkette eine Kommunikationsmöglichkeit dar – dadurch, dass Entwicklung, QA, Infosec und Operations die Änderungen der anderen sehen können, werden Überraschungen vermieden,

16 Jeder, der schon einmal eine Codemigration für ein ERP-System (zum Beispiel SAP oder Oracle Financials) durchgeführt hat, wird die folgende Situation kennen: Schlägt die Migration fehl, liegt das selten an einem Codefehler. Stattdessen ist die Wahrscheinlichkeit viel größer, dass es an einem Unterschied der Umgebungen liegt, zum Beispiel zwischen Entwicklung und QA oder QA und Produktion.

die Arbeit aller wird sichtbarer, und es hilft dabei, Vertrauen aufzubauen und zu verstärken (siehe Anhang G). Das heißt natürlich, dass alle Teams das gleiche System zur Versionsverwaltung nutzen müssen.

Infrastruktur einfacher neu bauen, statt zu reparieren

Können wir unsere Anwendungen und Umgebungen schnell auf Anforderung neu bauen, können wir sie auch schnell neu erschaffen, wenn etwas schiefgeht, statt sie zu reparieren. Das ist zwar etwas, das so gut wie alle großen Web-Operations schon so machen (also bei mehr als 1.000 Servern), aber wir sollten diese Praxis auch dann übernehmen, wenn wir nur einen Server produktiv haben.

Bill Baker, angesehener Entwickler bei Microsoft, hat scherzhaft gesagt, dass wir Server wie Haustiere behandelt haben: »Wir gaben ihnen Namen, und wenn sie krank wurden, haben wir uns liebevoll um sie gekümmert. Heutzutage werden Server wie Nutzvieh behandelt. Sie erhalten Nummern, und wenn sie krank werden, erschießt man sie.«[17]

Durch die Systeme, mit denen sich Umgebungen reproduzierbar erstellen lassen, können wir die Kapazitäten leicht erhöhen, indem wir neue Server dazustellen (horizontales Skalieren genannt). Zudem vermeiden wir Desaster, die unweigerlich eintreten, wenn wir Services nach einem katastrophalen Fehler wiederherstellen müssen, die Infrastruktur aber nicht reproduzierbar ist, weil seit Jahren undokumentierte und manuelle Änderungen an den Produktivumgebungen durchgeführt werden.

Um die Konsistenz unserer Umgebungen sicherzustellen, müssen alle Änderungen an der Produktivumgebung (Konfigurationsänderungen, Patches, Upgrades usw.) auch an allen anderen Produktiv- und Pre-Production-Umgebungen durchgeführt werden, und neu erstellte Umgebungen müssen diese ebenfalls enthalten.

Statt uns manuell an den Servern anzumelden und Änderungen vorzunehmen, müssen wir sie so durchführen, dass sie automatisch überall repliziert werden, und sie müssen in die Versionsverwaltung wandern.

Abhängig vom Lebenszyklus der fraglichen Konfiguration können wir auf automatische Konfigurationssysteme zurückgreifen, um diese Konsistenz zu erhalten (zum Beispiel Puppet, Chef, Ansible, Salt, Bosh oder andere), ein Service Mesh oder einen Configuration Management Service zum Verteilen der Runtime-Konfiguration nutzen (Istio, AWS Systems Manager Parameter Store und andere), oder wir erzeugen neue virtuelle Maschinen oder Container über automatische Build-Mechanismen und deployen diese in die Produktion, um dann die alten zu zerstören oder aus der Rotation zu nehmen.[18]

17 Sharwood, »Are Your Servers PETS or CATTLE?«

18 Bei Netflix ist eine AWS-Instanz durchschnittlich 24 Tage alt, wobei 60 % weniger als eine Woche überleben. – Chan, »OWASP AppSecUSA 2012«

Das eben beschriebene Muster hat den Namen *Immutable Infrastructure* – manuelle Änderungen an der Produktivumgebung sind nicht mehr erlaubt, und die einzige Möglichkeit, dort Änderungen vorzunehmen, ist, sie in die Versionsverwaltung einzuchecken und den Code und die Umgebungen von Grund auf neu zu erstellen.[19] Dadurch kann sich in die Produktivumgebung keine Abweichung zu den anderen Umgebungen einschleichen.

Um unkontrollierte Konfigurationsabweichungen zu verhindern, können wir eventuell Remote-Log-ins für Produktivserver deaktivieren[20] oder routinemäßig Produktivinstanzen abschießen und ersetzen,[21] um sicherzustellen, dass manuell vorgenommene Produktivänderungen entfernt werden. Damit wird jeder dazu motiviert, seine Änderungen auf dem korrekten Weg in die Versionsverwaltung einzubringen. Durch das Umsetzen solcher Maßnahmen reduzieren wir systematisch die Gefahr, dass unsere Infrastruktur von einem bekannten, funktionierenden Zustand wegdriftet (zum Beispiel durch kleine Konfigurationsänderungen, fragile Artefakte, Kunstwerke oder »Schneeflocken«).

Zudem müssen wir unsere Pre-Production-Umgebungen aktuell halten – insbesondere müssen Entwickler immer auf der aktuellsten Umgebung arbeiten. Die Entwickler bevorzugen häufig ältere Umgebungen, weil sie Angst haben, dass Updates bestehende Funktionalität zerstören. Aber wir wollen, dass sie häufig updaten, um Probleme so früh wie möglich zu entdecken,[22] und GitHub hat im *2020 State of Octoverse Report* gezeigt, dass Sie Ihre Codebasis am besten absichern, indem Sie Ihre Software aktuell halten.[23]

Neue Fallstudie: Wie eine Hotelkette 30 Milliarden Dollar Umsatz in Containern generierte (2020)

Als Mitarbeiter einer der größten Hotelketten haben Dwayne Holmes, Senior Director of DevSecOps and Enterprise Platforms, und sein Team alle umsatzgenerierenden Systeme containerisiert, was das Generieren von über 30 Milliarden Dollar Jahresumsatz unterstützt.[24]

Dwayne kam ursprünglich aus dem Finanzsektor. Er hatte Probleme damit, weitere Elemente zu finden, die sich automatisieren ließen, um die Produktivi-

19 Fowler, »Trash Your Servers and Burn Your Code«

20 Oder wir erlauben dies nur in Notfällen, wobei wir dafür sorgen, dass eine Kopie der über die Konsole eingegebenen Befehle an das Operations-Team gemailt wird.

21 Kelly Shortridge hat darüber einiges in ihrem Buch *Security Chaos Engineering* geschrieben.

22 Der gesamte Anwendungs-Stack und die Umgebung kann in Containern zusammengefasst werden, was zu einer beispiellosen Einfachheit und Geschwindigkeit in der gesamten Deployment-Pipeline führen kann. – Willis, »Docker and the Three Ways of DevOps Part 1«

23 Forsgren et al., *2020 State oft he Octoverse*

24 Holmes, »How A Hotel Company Ran $30B of Revenue in Containers«

tät zu erhöhen. Bei einem lokalen Treffen zu Ruby on Rails stolperte er über Container. Für Dwayne waren Container eine klare Lösung zum Beschleunigen des Business-Werts und zum Erhöhen der Produktivität.

Container helfen bei drei Aspekten: Sie abstrahieren Infrastruktur (das Wähltonprinzip – Sie nehmen den Hörer von einem Telefon ab, und es funktioniert, ohne dass man wissen muss, wie), und sie ermöglichen Spezialisierung (Operations kann Container erstellen, die von der Entwicklung immer wieder genutzt werden kann) und Automatisierung (Container können immer wieder gebaut werden, und trotzdem funktioniert alles).[25]

Nachdem seine Liebe zu Containern nun aufgeblüht war, ergriff Dwayne eine Chance, ließ seinen bequemen Posten zurück und wurde freier Mitarbeiter für eine der größten Hotelketten, die dazu bereit war, komplett auf Container zu setzen.[26]

Mit einem kleinen interdisziplinären Team aus drei Entwicklerinnen und Entwicklern sowie drei Mitgliedern der Infrastrukturabteilung bestand das Ziel nun darin, zu klären, ob man evolutionär oder revolutionär vorgehen wollte und damit eventuell die Arbeitsweise im Unternehmen komplett umkrempeln würde.[27]

Auf dem Weg gab es viel zu lernen, wie Dwayne in seiner Präsentation auf dem *DevOps Enterprise Summit 2020* erklärte, aber schließlich wurde das Projekt ein Erfolg.[28]

Für Dwayne und die Hotelkette sind Container der richtige Weg. Sie sind Cloudportabel. Sie sind skalierbar. Es gibt eingebaute Health Checks. Sie können auf Latenz versus CPU testen, und Zertifikate sind nicht mehr länger in die Anwendung eingebaut oder werden von der Entwicklung verwaltet. Zudem ist man nun dazu in der Lage, sich auf Circuit Breaking zu konzentrieren, es gibt eingebautes APM, man setzt auf Zero Trust, und Images sind sehr klein – dank einer guten Container-Hygiene und dem Einsatz von Sidecars für alles Mögliche.[29]

Während seiner Zeit bei der Hotelkette haben Dwayne und sein Team über 3.000 Entwicklerinnen und Entwickler bei verschiedenen Service-Providern unterstützt. 2016 liefen Microservices und Container produktiv. 2017 wurde eine Milliarde Dollar in Containern verarbeitet, 90 % der neuen Anwendungen liefen in Containern, und Kubernetes lief in der Produktivumgebung. 2018 waren sie einer der fünf größten produktiven Kubernetes-Cluster (nach Umsatz).

25 Holmes, »How A Hotel Company Ran $30B of Revenue in Containers«
26 Ebd.
27 Ebd.
28 Ebd.
29 Ebd.

Und 2020 wurden Tausende Builds und Deployments pro Tag ausgeführt, und Kubernetes lief bei fünf Cloud-Providern.[30]

Container wurden zu einer schnell wachsenden Methode, Infrastruktur leichter neu zu bauen und wiederzuverwenden, als sie zu reparieren, was schließlich das Bereitstellen von Business-Wert beschleunigt und die Entwicklungsproduktivität steigert.

Die Definition des Entwicklungs-»Done« so anpassen, dass der Code in produktivähnlichen Umgebungen läuft

Nachdem nun unsere Umgebungen auf Anforderung hin erstellt werden können und sich alles unter Versionsverwaltung befindet, ist es unser Ziel, sicherzustellen, dass diese Umgebungen auch in der täglichen Arbeit der Entwickler genutzt werden. Wir müssen prüfen, ob die Anwendung in einer produktivähnlichen Umgebung wie erwartet läuft – und zwar lange vor dem Ende des Projekts oder vor dem ersten Produktiv-Deployment.

Die meisten modernen Softwareentwicklungsmethoden geben kurze und iterative Entwicklungsintervalle vor, die im Gegensatz zum Big-Bang-Ansatz stehen (zum Beispiel dem Wasserfall-Modell). Je länger die Intervalle bei der Entwicklung sind, desto schlechter ist im Allgemeinen das Ergebnis. So ist zum Beispiel bei der Scrum-Methode ein *Sprint* ein zeitlich begrenztes Entwicklungsintervall (meist ein Monat oder weniger), in dem wir fertig – »done« – werden sollen, was im Allgemeinen definiert ist als »funktionierender und potenziell auslieferbarer Code«.

Wir wollen sicherstellen, dass Entwicklung und QA im Projekt den Code routinemäßig und mit steigender Frequenz in produktivähnliche Umgebungen integrieren.[31] Dazu erweitern wir die Definition von »Done« über die reine Korrektheit der Codefunktionalität hinaus (hervorgehoben im nächsten Satz): Am Ende jedes Entwicklungsintervalls (oder häufiger) haben wir integrierten, getesteten, lauffähigen und potenziell auslieferbaren Code, *der in einer produktivähnlichen Umgebung demonstriert wird*.

Mit anderen Worten: Wir akzeptieren nur Entwicklungsarbeit als »Done«, wenn sie erfolgreich gebaut und deployt ist und wie erwartet in einer produktivähnlichen Umgebung laufen kann, statt uns darauf zu beschränken, was ein Entwickler für

30 Ebd.

31 Der Begriff *Integration* hat in Entwicklung und Operations viele leicht unterschiedliche Bedeutungen. Bei der Entwicklung bedeutet Integration meist eine *Codeintegration*, also das Zusammenführen mehrerer Code-Branches in der Versionsverwaltung in den Trunk. Beim Continuous Delivery und in DevOps bezieht sich das *Integrationstesten* auf das Testen der Anwendung in einer produktivähnlichen Umgebung oder in einer Integrationstestumgebung.

seinen Code als »Done« ansieht. Idealerweise läuft der Code unter produktivähnlicher Last mit produktivähnlichen Daten, und zwar lange vor dem Ende eines Sprints. Damit werden Situationen vermieden, bei denen ein Feature als »Done« abgehakt wird, nur weil ein Entwickler es erfolgreich auf seinem Laptop laufen lässt (aber nirgendwo sonst).

Indem Entwickler ihren eigenen Code in einer produktivähnlichen Umgebung schreiben, testen und ausführen, geschieht ein Großteil der Arbeit für eine erfolgreiche Integration des Codes in unseren Umgebungen schon während des Tagesgeschäfts und nicht erst am Ende des Release. Bei Abschluss des ersten Intervalls kann unsere Anwendung in einer produktivähnlichen Umgebung als erfolgreich laufend vorgeführt werden, wobei Code und Umgebung schon viele Male gemeinsam integriert wurden – idealerweise mit rein automatischen Schritten (ohne manuelles Eingreifen).

Hinzu kommt, dass wir bis zum Ende des Projekts unseren Code hundert- oder gar tausendfach in produktivähnliche Umgebungen deployt haben und sicher sein können, dass die meisten Probleme beim Deployment in die Produktivumgebung gefunden und behoben wurden.

Idealerweise nutzen wir in den Pre-Production-Umgebungen die gleichen Tools zum Monitoren, Loggen und Deployen wie im Produktivumfeld. Dadurch sind wir mit ihnen vertraut und haben schon Erfahrung darin, unseren Service im Produktivumfeld ohne Probleme laufen zu lassen oder diese schnell erkennen und beheben zu können.

Indem wir Entwicklung und Operations gemeinsam erfahren lassen, wie Code und Umgebung interagieren, und Deployments rechtzeitig und häufig üben lassen, reduzieren wir die mit Produktiv-Releases verbundenen Deployment-Risiken deutlich. So können wir auch eine gesamte Klasse von operationellen und Sicherheitsfehlern sowie architektonischen Problemen ausmerzen, die im Allgemeinen in einem Projekt erst zu spät erkannt werden und dann nicht mehr behoben werden können.

Zusammenfassung

Der schnelle Arbeitsflow von der Entwicklung zu Operations erfordert, dass jeder auf Anforderung produktivähnliche Umgebungen erhalten kann. Indem Entwickler solche Umgebungen schon sehr frühzeitig im Projekt nutzen, verringern wir das Risiko von Produktivproblemen deutlich. Das ist eine der vielen Vorgehensweisen, die zeigen, wie Operations die Entwickler viel produktiver machen kann. Wir erzwingen das Ausführen von Code in produktivähnlichen Umgebungen, indem wir es in die Definition von »Done« einbinden.

Zudem bringen wir alle Produktivartefakte in die Versionsverwaltung und haben damit eine »Single Source of Truth«, die es uns ermöglicht, die gesamte Produktiv-

umgebung schnell, wiederholbar und dokumentiert neu zu erstellen, wobei wir die gleichen Entwicklungspraktiken für die Operations-Arbeit nutzen wie für die Entwicklungsarbeit. Und wenn sich Produktivinfrastruktur leichter neu bauen als reparieren lässt, werden auch Probleme einfacher und schneller gelöst, und wir können unsere Kapazität besser erweitern. Sind diese Maßnahmen umgesetzt, gehen wir zum nächsten Schritt, in dem wir die Tests umfassend automatisieren – das Thema des nächsten Kapitels.