Eberhard Wolff

Continuous Delivery

Der pragmatische Einstieg

dpunkt.verlag

Eberhard Wolff beschäftigt sich seit vielen Jahren mit Softwareentwicklung und -architektur. Er ist Autor zahlreicher Fachartikel und Bücher, regelmäßiger Sprecher auf internationalen Konferenzen und im Programmkomitee verschiedener Konferenzen vertreten. Er ist Fellow bei der innoQ.

Continuous Delivery und die Auswirkungen hat er in verschiedenen Projekten in unterschiedlichen Rollen kennengelernt. Der Ansatz verspricht, die Produktivität der IT-Projekte erheblich zu erhöhen, und hat Auswirkungen auf das Vorgehen, aber auch die Architektur und die Technologien. Daher lag es für ihn auf der Hand, dieses Buch zu schreiben.

Eberhard Wolff

Continuous Delivery

Der pragmatische Einstieg

2., aktualisierte und erweiterte Auflage

dpunkt.verlag

Eberhard Wolff
eberhard.wolff@gmail.com

Lektorat: René Schönfeldt
Copy-Editing: Petra Kienle, Fürstenfeldbruck
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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

ISBN
Print 978-3-86490-371-7
PDF 978-3-86491-930-5
ePub 978-3-86491-931-2
mobi 978-3-86491-932-9

2., aktualisierte und erweiterte Auflage 2016

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Inhaltsverzeichnis

1 Einleitung 1

1.1 Überblick über Continuous Delivery und das Buch 1
1.2 Warum überhaupt Continuous Delivery? 2
1.3 Für wen ist das Buch? 5
1.4 Neu in der 2. Auflage 5
1.5 Übersicht über die Kapitel 7
1.6 Pfade durch das Buch 8
1.7 Danksagung 10

2 Continuous Delivery: Was und wie? 13

2.1 Was ist Continuous Delivery? 13
2.2 Warum Software-Releases so kompliziert sind 13
2.3 Werte von Continuous Delivery 14
2.4 Vorteile von Continuous Delivery 17
2.4.1 Continuous Delivery für Time-to-Market 17
2.4.2 Continuous Delivery zur Risikominimierung 20
2.4.3 Schnelleres Feedback und Lean 23
2.5 Aufbau und Struktur einer Continuous-Delivery-Pipeline . 24
2.6 Links & Literatur 28

3 Infrastruktur bereitstellen 29

3.1 Einleitung 29
3.2 Installationsskripte 31
3.3 Chef 34
3.3.1 Technische Grundlagen 37
3.3.2 Chef Solo 44

3.3.3 Chef Solo: Fazit 46
3.3.4 Knife und Chef Server 46
3.3.5 Chef Server: Fazit 51
3.4 Vagrant 51
3.4.1 Ein Beispiel mit Chef und Vagrant 53
3.4.2 Vagrant: Fazit 55
3.5 Docker 55
3.5.1 Dockers Lösung 56
3.5.2 Docker-Container erstellen 59
3.5.3 Beispielanwendung mit Docker betreiben 61
3.5.4 Docker und Vagrant 63
3.5.5 Docker Machine 66
3.5.6 Komplexe Konfigurationen mit Docker 68
3.5.7 Docker Compose 70
3.6 Immutable Server 73
3.7 Infrastructure as Code 74
3.8 Platform as a Service (PaaS) 77
3.9 Umgang mit Daten und Datenbanken 79
3.10 Fazit 82
3.11 Links & Literatur 83

4 Build-Automatisierung und Continuous Integration 87

4.1 Überblick 87
4.2 Build-Automatisierung und Build-Tools 88
4.2.1 Ant 90
4.2.2 Maven 90
4.2.3 Gradle 95
4.2.4 Weitere Build-Tools 98
4.2.5 Das geeignete Tool auswählen 99
4.2.6 Links und Literatur 100
4.2.7 Experimente und selber ausprobieren 100
4.3 Unit-Tests 101
4.3.1 »Gute« Unit-Tests schreiben 103
4.3.2 TDD – Test-driven Development 105
4.3.3 Clean Code und Software Craftsmanship 106
4.3.4 Links und Literatur 106
4.3.5 Experimente und selber ausprobieren 107
4.4 Continuous Integration 107
4.4.1 Jenkins 108
4.4.2 Continuous-Integration-Infrastruktur 114
4.4.3 Fazit 115

4.4.4 Links und Literatur 116
4.4.5 Experimente und selber ausprobieren 116
4.5 Codequalität messen 118
4.5.1 SonarQube 120
4.5.2 Links und Literatur 122
4.5.3 Experimente und selber ausprobieren 122
4.6 Artefakte managen 123
4.6.1 Integration in den Build 125
4.6.2 Weiterreichende Funktionen von Repositories . 127
4.6.3 Links und Literatur 127
4.6.4 Experimente und selber ausprobieren 127
4.7 Fazit 128

5 Akzeptanztests 131

5.1 Einführung 131
5.2 Die Test-Pyramide 131
5.3 Was sind Akzeptanztests? 135
5.4 GUI-basierte Akzeptanztests 139
5.5 Alternative Werkzeuge für GUI-Tests 145
5.6 Textuelle Akzeptanztests 147
5.7 Alternative Frameworks 150
5.8 Strategien für Akzeptanztests 152
5.9 Fazit 154
5.10 Links & Literatur 154

6 Kapazitätstests 157

6.1 Einführung 157
6.2 Kapazitätstests – wie? 158
6.3 Kapazitätstests implementieren 163
6.4 Kapazitätstests mit Gatling 164
6.5 Alternativen zu Gatling 169
6.6 Fazit 171
6.7 Links & Literatur 172

7 Exploratives Testen 173

7.1 Einleitung 173
7.2 Warum explorative Tests? 173
7.3 Wie vorgehen? 175

7.4 Fazit 179
7.5 Links & Literatur 180

8 Deploy – der Rollout in Produktion 181

8.1 Einleitung 181
8.2 Rollout und Rollback 182
8.3 Roll Forward 183
8.4 Blue/Green Deployment 185
8.5 Canary Releasing 186
8.6 Continuous Deployment 188
8.7 Virtualisierung 190
8.8 Jenseits der Webanwendungen 192
8.9 Fazit 193
8.10 Links und Literatur 194

9 Operate – Produktionsbetrieb der Anwendungen 195

9.1 Einleitung 195
9.2 Herausforderungen im Betrieb 196
9.3 Log-Dateien 198
9.3.1 Werkzeuge zum Verarbeiten von Log-Dateien . . 200
9.3.2 Logging in der Beispielanwendung 202
9.4 Logs der Beispielanwendung analysieren 203
9.4.1 Experimente und selber ausprobieren 208
9.5 Andere Technologien für Logs 211
9.6 Fortgeschrittene Log-Techniken 212
9.7 Monitoring 213
9.8 Metriken mit Graphite 214
9.9 Metriken in der Beispielanwendung 216
9.9.1 Experimente und selber ausprobieren 217
9.10 Andere Monitoring-Lösungen 219
9.11 Weitere Herausforderungen beim Betrieb der Anwendung 220
9.12 Fazit 221
9.13 Links & Literatur 222

10 Continuous Delivery im Unternehmen einführen 225

10.1 Einleitung 225
10.2 Continuous Delivery von Anfang an 225
10.3 Value Stream Mapping 226
10.4 Weitere Optimierungsmaßnahmen 229
10.5 Zusammenfassung 233
10.6 Links & Literatur 233

11 Continuous Delivery und DevOps 235

11.1 Einführung 235
11.2 Was ist DevOps? 235
11.3 Continuous Delivery und DevOps 239
11.4 Continuous Delivery ohne DevOps? 243
11.5 Fazit 245
11.6 Links & Literatur 246

12 Continuous Delivery, DevOps und Softwarearchitektur 247

12.1 Einleitung 247
12.2 Softwarearchitektur 247
12.3 Komponentenaufteilung für Continuous Delivery optimieren 250
12.4 Schnittstellen 252
12.5 Datenbanken 255
12.6 Microservices 258
12.7 Umgang mit neuen Features 261
12.8 Fazit 264
12.9 Links & Literatur 265

13 Fazit: Was bringt's? 267

13.1 Links & Literatur 268

Index 269

1 Einleitung

1.1 Überblick über Continuous Delivery und das Buch

Continuous Delivery ermöglicht es, Software schneller und mit wesentlich höherer Zuverlässigkeit in Produktion zu bringen als bisher. Grundlage dafür ist eine Continuous-Delivery-Pipeline, die das Ausrollen der Software weitgehend automatisiert und so einen reproduzierbaren, risikoarmen Prozess für die Bereitstellung neuer Releases darstellt.

Woher kommt der Begriff Continuous Delivery?

Das agile Manifest *(http://agilemanifesto.org)* definiert als wichtigstes Ziel:

»Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.«

Die deutsche Übersetzung findet sich ebenfalls auf der Website:

»Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen.«

Also ist Continuous Delivery eine Technik aus dem agilen Umfeld.

Dieses Buch erläutert, wie eine solche Pipeline praktisch aufgebaut werden kann und welche Technologien dazu eingesetzt werden können. Dabei geht es nicht nur um das Kompilieren und die Installation der Software, sondern auch um verschiedene Tests, die dazu dienen, die Qualität der Software abzusichern.

Das Buch zeigt außerdem, welche Auswirkungen Continuous Delivery auf das Zusammenspiel zwischen Entwicklung (Development) und Betrieb (Operations) im Rahmen von DevOps hat. Schließlich werden die Auswirkungen auf die Softwarearchitektur beschrieben. Neben der Theorie wird ein möglicher Technologie-Stack vorgestellt, der Build, Continuous Integration, Lasttests, Akzeptanztests und Monitoring abdeckt. Für die einzelnen Bestandteile des Technologie-Stacks gibt es jeweils ein Beispielprojekt, mit dem der Leser praktische

Erfahrungen sammeln kann. Das Buch bietet einen Einstieg in den Technologie-Stack und zeigt außerdem auf, wie man sich mit den Themen tiefergehend beschäftigen kann.

Durch Experimente und Vorschläge zum selber Ausprobieren lädt es zur weiteren praktischen Vertiefung ein. Die Leser erhalten so Anregungen, wie sie sich weiter in die Themen vertiefen und eigene Erfahrungen sammeln können. So können die Beispielprojekte Basis für eigene Experimente oder für den Aufbau einer eigenen Continuous-Deployment-Pipeline sein.

Unter *http://continuous-delivery-buch.de* steht die Website zum Buch bereit mit Informationen, Errata und Links zu dem Beispiel.

1.2 Warum überhaupt Continuous Delivery?

Warum sollte man überhaupt Continuous Delivery einsetzen? Eine kleine Geschichte soll diese Frage beantworten – ob sie wahr ist oder nicht, bleibt offen.

Eine kleine Geschichte

Das Marketing eines Unternehmens – nennen wir es Raffzahn Online Commerce GmbH – hatte entschieden, den Registrierungsprozess auf der E-Commerce-Website zu überarbeiten. Dadurch sollten mehr Kunden gewonnen und der Umsatz erhöht werden. Also machte sich ein Entwicklerteam an die Arbeit. Nach nicht allzu langer Zeit waren sie fertig.

Zunächst mussten die Änderungen getestet werden. Dazu hatte das Team der Raffzahn Online Commerce GmbH in einem aufwendigen Prozess eine Testumgebung aufgebaut, auf der die Software manuell getestet wurde. Und es wurden tatsächlich Fehler gefunden. Mittlerweile waren die Entwickler aber schon bei dem nächsten Projekt und mussten sich wieder einarbeiten, bevor sie die Fehler mit einem Fix beheben konnten. Und wegen der manuellen Tests stellte sich bei einigen »Fehlern« heraus, dass die Tester nicht richtig getestet hatten oder die Fehler aus irgendwelchen Gründen nicht reproduzierbar waren.

Nun galt es, den Code in Produktion zu bringen. Der Prozess dazu war aufwendig – denn die E-Commerce-Website der Raffzahn Online Commerce GmbH war über die Jahre gewachsen und daher sehr komplex. Nur dieses eine Feature auszuliefern, rechtfertigte den Aufwand nicht. Daher wurde nur einmal pro Monat deployt. Schließlich konnte die Änderung zusammen mit den anderen Änderungen aus dem letzten Monat in Produktion gehen. Dazu war eine Nacht reserviert. Leider gab es beim Rollout einen Fehler. Das Team machte sich an die Arbeit, das Problem zu analysieren. Aber das war so schwierig, dass das System am nächsten Morgen nicht zur Verfügung stand. Zu diesem Zeit-

punkt waren die Mitarbeiter übernächtigt und standen unter großem Stress – jede Minute Ausfall kostete bares Geld. Und zurück zur alten Version ging es nicht, weil einige Änderungen im Deployment nicht ohne Weiteres rückgängig gemacht werden konnten. Erst im Laufe des Tages, nach einer ausführlichen Fehleranalyse, konnte eine Task Force das Problem beheben und die Website stand wieder zur Verfügung. Der Fehler war eine Konfigurationsänderung gewesen, die in der Testumgebung vorgenommen, aber bei der Produktionseinführung vergessen worden war.

Also schien alles in Ordnung zu sein – aber es gab einen weiteren Fehler, der zunächst nicht entdeckt wurde. Dieser Fehler hätte eigentlich durch die manuellen Tests gefunden werden sollen. Der Test, der den Fehler gefunden hätte, wurde auch erfolgreich durchgeführt. Aber in der Testphase wurden auch einige Fehler gefixt und dieser Test wurde nur vor den Fixes durchgeführt. Der Fehler wurde erst durch einen der Fixes eingeführt. Nach den Fixes wurde der Test nicht noch einmal durchgeführt – daher konnte der Fehler es bis in die Produktion schaffen.

Am nächsten Tag stellte sich also mehr zufällig heraus, dass die Registrierung für die Website der Raffzahn Online Commerce GmbH gar nicht mehr funktionierte. Das war niemandem aufgefallen und erst, nachdem der erste potenzielle Kunde sich bei der Hotline meldete, wurde das Problem erkannt. Wie viele Registrierungen dieser Ausfall gekostet hatte, konnte leider niemand sagen – dazu fehlten Informationen über die Nutzung der Website. Wie schnell die optimierte Registrierung diesen Nachteil ausgleichen konnte, ist fraglich. So konnte es gut sein, dass die Änderung nicht wie ursprünglich geplant zu mehr Registrierungen, sondern zu weniger geführt hatte. Und außerdem war das neue Release wesentlich langsamer – ein Umstand, mit dem vorher auch niemand gerechnet hatte.

Und so begann die Raffzahn Online Commerce GmbH, die nächsten Features zu implementieren, um in einem Monat wiederum ein Update der Website auszurollen. Was wohl dieses Mal passieren würde?

Continuous Delivery hilft.

Continuous Delivery löst solche Probleme durch verschiedene Maßnahmen:

- Es wird öfter deployt – bis hin zu mehreren Malen pro Tag. Dadurch wird die Zeit, bis ein neues Feature genutzt werden kann, verringert.
- Durch häufige Deployments ist auch das Feedback auf neue Features und Code-Änderungen schneller. Die Entwickler müssen sich

nicht erst wieder darauf besinnen, was sie vor einem Monat implementiert haben.

- Um schneller zu deployen, müssen der Aufbau von Umgebungen und die Tests weitgehend automatisiert werden, da der Aufwand sonst zu hoch ist.
- Die Automatisierung führt zu Reproduzierbarkeit: Wenn die Testumgebung erfolgreich aufgebaut werden kann, dann lässt sich mit demselben Automatismus auch die Produktion aufbauen – und zwar mit derselben Konfiguration. Das Problem durch die Fehlkonfiguration der Produktionsumgebung wäre also nicht aufgetreten.
- Außerdem führt die Automatisierung zu mehr Flexibilität. Testumgebungen können On-Demand aufgesetzt werden. So kann es z.B. bei einem Redesign der Oberflächen zeitlich begrenzt eine separate Testumgebung für Marketing geben. Oder für großangelegte Lasttests können zusätzliche Umgebungen aufgesetzt werden, um eine produktionsnahe Umgebung zu haben, die nach den Tests wieder abgerissen werden, so dass keine dauerhaften Investitionen in Hardware notwendig sind – wenn beispielsweise eine Cloud genutzt wird.
- Automatisierte Tests führen dazu, dass Fehler leichter reproduziert werden können. Da die exakt gleichen Schritte bei jedem Test ausgeführt werden, gibt es auch keine Fehler bei der Testdurchführung.
- Wenn Tests automatisiert sind, können sie öfter ausgeführt werden. Also wäre der Fix durch den gesamten Testprozess gegangen und dieser Fehler nicht erst in Produktion aufgefallen.
- Das Risiko eines neuen Release wird weiter reduziert, indem das Deployment in Produktion so aufgesetzt wird, dass es einen Weg zurück zur alten Version gibt. So wird der Produktionsausfall aus dem Beispiel verhindert.
- Und schließlich sollten die Anwendungen auch ein fachliches Monitoring haben, so dass die Registrierung nicht ausfallen kann, ohne dass es jemand merkt.

Durch Continuous Delivery gewinnt das Business eine schnellere Verfügbarkeit neuer Features und eine zuverlässigere IT. Die Zuverlässigkeit ist auch für die IT nützlich. Nachts oder an Wochenenden unter hohem Stress neue Releases auszurollen und Fehler zu beheben, macht

eben keinen Spaß. Und es ist sicher auch für die IT besser, wenn Fehler durch Tests auffallen und nicht erst in Produktion.

Um Continuous Delivery umzusetzen, gibt es eine Vielzahl an Technologien und Techniken. Continuous Delivery hat Auswirkungen bis hin zur Architektur der Anwendung. Genau um diese Themen geht es in diesem Buch. Am Ende steht ein schnellerer und zuverlässigerer Prozess, um Software in Produktion zu bringen.

1.3 Für wen ist das Buch?

Das Buch wendet sich an Manager, Architekten, Entwickler und Administratoren, die Continuous Delivery als Methodik und/oder DevOps als Organisationsform einführen wollen:

- Manager lernen durch den theoretischen Teil Prozess, Erfordernisse und Vorteile von Continuous Delivery für ihr Unternehmen kennen. Außerdem können sie die technischen Konsequenzen von Continuous Delivery abschätzen.
- Entwickler und Administratoren erhalten eine umfassende Einleitung in die technischen Aspekte und können damit die notwendigen Fähigkeiten erlernen, um Continuous Delivery umzusetzen und eine entsprechende Pipeline aufzubauen.
- Architekten können neben den technischen Aspekten auch die Auswirkung auf die Softwarearchitektur kennenlernen – siehe dazu Kapitel 12.

Das Buch stellt verschiedene Technologien für die Umsetzung von Continuous Delivery vor. Als Beispiel dient ein Java-Projekt. Für einige technologische Bereiche – zum Beispiel für die Umsetzung von Akzeptanztests – sind für andere Programmiersprachen andere Technologien etabliert. Das Buch zeigt an den jeweiligen Stellen Alternativen auf, fokussiert aber auf Java. Technologien zur automatisierten Bereitstellung von Infrastrukturen sind unabhängig von der genutzten Programmiersprache. Das Buch ist besonders gut für Leser geeignet, die im Java-Umfeld aktiv sind – für andere Technologien müssen die Ansätze teilweise vom Leser selber transferiert werden.

1.4 Neu in der 2. Auflage

Die Neuauflage wurde in Bezug auf Werkzeuge wie Docker, Jenkins, Graphite und den ELK-Stack aktualisiert. An neuen Themen sind Docker Compose, Docker Machine, Immutable Server, Microservices

und die Einführung von Continuous Delivery ohne DevOps hinzugekommen. Im Einzelnen:

- Der neue Abschnitt 3.6 beschreibt das Konzept »Immutable Server«, bei dem Server niemals mit Updates versehen werden, sondern unveränderlich sind. Das macht Installationen einfacher reproduzierbar.
- Docker ist ein sehr interessantes Werkzeug für das Deployment von Software. Docker Machine vereinfacht die Installation von Docker-Servern. Der neue Abschnitt 3.5.5 beschreibt Docker Machine. Mit Docker Compose können mehrere Docker Container zusammen installiert werden – das beschreibt der neue Abschnitt 3.5.7.
- Continuous Delivery und DevOps haben viele Synergien – aber Continuous Delivery ist auch ohne DevOps möglich. Das beschreibt der neue Abschnitt 11.4.
- Der neue Abschnitt 12.6 erläutert Microservices-Architekturen und die Beziehung zu Continuous Delivery.
- Der ELK-Stack zum Speichern und zur Analyse von Logs in Abschnitt 9.3 nutzt die aktuelle Version von Elasticsearch (2.1), Logstash (2.1) und Kibana (4.3). Dabei hat sich die Installation, aber auch die Oberfläche geändert – deswegen waren auch Änderungen im Abschnitt »Experimente und selber ausprobieren« notwendig. Die Installation ist nun nicht nur mit Vagrant, sondern auch Docker Machine möglich.
- Das Monitoring mit Graphite in Abschnitt 9.8 ist jetzt wesentlich besser modularisiert: Ein Docker-Container nimmt die Metriken entgegen, ein anderer bietet die Webschnittstelle zur Analyse der Daten. Auch hier kann das Beispiel nun mit Docker Machine statt Vagrant genutzt werden.

In der 2. Auflage wurden die Beispiele auf aktuelle Versionen der Werkzeuge umgestellt. Konkret:

- Die Beispiel-VMs benutzen jetzt Ubuntu 15.04.
- Die Beispielanwendung verwendet Spring Boot 1.3.0.
- Das Chef-Beispiel in Abschnitt 3.3 wurde aktualisiert auf Chef 12, Java 1.8 und Tomcat 7
- Docker aus Abschnitt 3.5 basiert jetzt auf Docker 1.10.
- Selenium für GUI-Tests in Abschnitt 5.4 wird in der Version 2.48.2 verwendet.

- JBehave für textuelle Akzeptanztests in Abschnitt 5.7 bezieht sich auf Version 3.9.5.

1.5 Übersicht über die Kapitel

Das Buch umfasst drei Teile. Der erste Teil legt die Grundlagen für ein Verständnis von Continuous Delivery:

- Kapitel 2 führt den Begriff Continuous Delivery ein und zeigt auf, welche Probleme Continuous Delivery wie löst. Dabei wird eine Einführung in die Continuous-Delivery-Pipeline gegeben.
- Für Continuous Delivery muss Infrastruktur automatisiert bereitgestellt werden – es muss Software auf Servern installiert werden. Kapitel 3 stellt dazu einige Ansätze vor. Chef dient zur Automatisierung von Installationen. Mit Vagrant können Testumgebungen auf Entwicklerrechnern eingerichtet werden. Docker ist nicht nur eine sehr effiziente Virtualisierungslösung, sondern kann auch zur automatisierten Installation von Software dienen. Schließlich wird noch ein Überblick über die Nutzung von PaaS-Cloud-Lösungen (Platform as a Service) für Continuous Delivery gegeben.

Es schließen sich im zweiten Teil Kapitel an, die einzelne Bestandteile einer Continuous-Delivery-Pipeline konkret beschreiben. Neben einer konzeptionellen Erläuterung werden jeweils beispielhaft konkrete Technologien dargestellt, mit denen dieser Teil der Pipeline umgesetzt werden kann:

- Im Mittelpunkt von Kapitel 4 von Bastian Spannberg steht alles, was beim Commit einer neuer Softwareversion geschieht. Build-Werkzeuge wie Gradle oder Maven werden vorgestellt, ein Überblick über Unit-Tests wird gegeben und Continuous Integration mit Jenkins wird beleuchtet. Daran schließen sich statische Code Reviews mit SonarQube und Repositories wie Nexus oder Artifactory an.
- Kapitel 5 zeigt mit JBehave und Selenium einen Ansatz für automatisierte GUI-basierte Akzeptanztests und für textuelle Akzeptanztests.
- Performance deckt das Kapitel 6 durch Kapazitätstests ab. Als Beispieltechnologie wird Gatling genutzt.
- Das explorative Testen (Kap. 7) dient dazu, manuell neue Features und generell Probleme in der Anwendung zu überprüfen.

Diese Kapitel betrachten den Anfang der Continuous-Delivery-Pipeline. Diese Phasen beeinflussen hauptsächlich die Softwareentwicklung. Die weiteren Kapitel stellen vor allem Techniken und Technologien vor, die bei den betriebsnahen Bereichen von Continuous Delivery nützlich sind:

- Kapitel 8 zeigt Vorgehensweisen, die beim Rollout der Software in Produktion nützlich sind, um Risiken zu reduzieren.
- Aus dem Betrieb der Anwendung können zahlreiche Daten erhoben werden, um so Feedback zu bekommen. Kapitel 9 stellt dazu Technologien vor: den ELK-Stack (Elasticsearch – Logstash – Kibana) für Log-Dateien-Analyse und Graphite für Monitoring.

Zu den Technologien aus diesen Kapiteln gibt es jeweils Beispiele zum selber Experimentieren und Nachvollziehen am eigenen Rechner. Dadurch können Leser sehr einfach eigene Erfahrungen sammeln. Dank Infrastrukturautomatisierung sind diese Beispiele auch auf dem eigenen Rechner recht einfach zum Laufen zu bringen.

Schließlich stellt sich die Frage, wie Continuous Delivery eingeführt werden kann und welche Auswirkungen Continuous Delivery hat. Das zeigt der dritte Teil des Buchs:

- Kapitel 10 zeigt, wie Continuous Delivery in einer Organisation eingeführt werden kann.
- DevOps beschreibt das Zusammenwachsen von Betrieb (Ops) und Entwicklung (Dev) zu einer Organisationseinheit (Kap. 11).
- Continuous Delivery hat auch Auswirkungen auf die Architektur der Anwendungen. Diese Herausforderungen diskutiert Kapitel 12.
- Am Ende steht das Fazit in Kapitel 13.

1.6 Pfade durch das Buch

Dieser Abschnitt erläutert die möglichen Lesepfade durch das Buch für die verschiedenen Zielgruppen – also welche Kapitel in welcher Reihenfolge gelesen werden sollten. Die Einführung in Kapitel 2 ist für alle Leser interessant – das Kapitel klärt grundlegende Begriffe und zeigt die Motivation für Continuous Delivery auf.

Die weiteren Kapitel haben unterschiedliche Ausrichtungen:

- Für Techniker, die sich vor allem für die Entwicklung interessieren, sind die Kapitel rund um Commit und Tests interessant. In diesen

Kapiteln geht es neben Entwicklung auch um Qualitätssicherung und Build. Die Kapitel zeigen, wie diese Aufgaben durch Continuous Delivery beeinflusst werden, und geben konkrete Code- und Technologiebeispiele aus dem Java-Bereich.

- Administratoren und Betriebsmitarbeiter sollten sich mit Themen wie dem Deployment, dem Bereitstellen von Infrastrukturen und dem Betrieb auseinandersetzen, die ebenfalls durch Continuous Delivery beeinflusst werden.
- Aus der Managementsicht sind die Einführung von Continuous Delivery und der Zusammenhang zwischen DevOps und Continuous Delivery interessant. Diese beiden Kapitel zeigen die Auswirkungen von Continuous Delivery auf die Organisation.
- Schließlich ist der Architektur und dem Fazit jeweils noch ein Extra-Kapitel gewidmet.

Architekten sind eher breit aufgestellt. Sie werden sicher das Kapitel 12 über Architektur lesen, aber meistens interessieren sich Architekten auch für die technischen Details und die Managementsicht. Daher werden Architekten vermutlich mindestens ausgewählte Kapitel aus diesen Bereichen lesen.

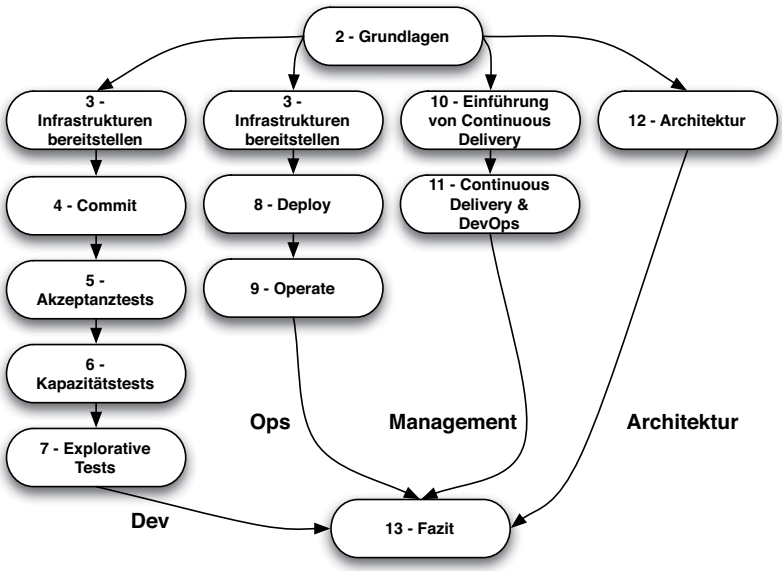

Abb. 1–1
Pfade durch das Buch

1.7 Danksagung

Bedanken möchte ich mich bei Bastian Spannberg für seinen Beitrag zu diesem Buch. Außerdem gilt mein Dank den Reviewern, die mit ihren Kommentaren das Buch wesentlich beeinflusst haben: Marcel Birkner, Lars Gentsch, Halil-Cem Gürsoy, Felix Müller, Sascha Möllering und Alexander Papaspyrou. Bedanken möchte ich mich auch für die vielen Diskussionen – viele Gedanken und Ideen sind in solchen Dialogen entstanden.

Bedanken möchte ich mich auch bei meinem Arbeitgeber, der innoQ.

Schließlich habe ich meinen Freunden, Eltern und Verwandten zu danken, die ich für das Buch oft vernachlässigt habe – insbesondere meiner Frau.

Und natürlich gilt mein Dank all jenen, die an den in diesem Buch erwähnten Technologien gearbeitet haben und so die Grundlagen für Continuous Delivery gelegt haben.

Last but not least möchte ich dem dpunkt.verlag und René Schönfeldt danken, der mich sehr professionell bei der Erstellung des Buchs unterstützt hat.

Teil 1: Grundlagen

In diesem Teil des Buchs werden in Kapitel 2 die Grundlagen für ein Verständnis von Continuous Delivery gelegt. Die technischen Grundlagen von Continuous Delivery zeigt Kapitel 3: Hier geht es um den automatisierten Aufbau von Infrastrukturen und die automatische Installation von Software, ohne die Continuous Delivery nicht denkbar ist.

2 Continuous Delivery: Was und wie?

2.1 Was ist Continuous Delivery?

Diese Frage ist nicht so einfach zu beantworten. Die Erfinder des Begriffs geben in [6] keine echte Definition. Martin Fowler [7] stellt in den Mittelpunkt, dass die Software jederzeit in Produktion ausgeliefert werden kann. Dazu ist eine Automatisierung der Prozesse zur Installation der Software notwendig und Feedback über die Qualität der Software. Wikipedia hingegen spricht von einer Optimierung und Automatisierung des Software-Release-Prozesses [8].

Letztendlich geht es bei Continuous Delivery darum, den Prozess bis zum Release der Software zu betrachten und zu optimieren. Genau dieser Prozess wird oft bei der Entwicklung ausgeblendet.

2.2 Warum Software-Releases so kompliziert sind

Software-Releases sind eine Herausforderung – jede IT-Abteilung hat wohl schon ein Wochenende in der Firma verbracht, um ein Release in Produktion zu bringen. Oft enden solche Aktionen damit, dass die Software irgendwie in Produktion gebracht wird – weil ab einem bestimmten Punkt in dem Prozess der Weg zurück zur alten Version noch gefährlicher und schwieriger ist, als der Weg nach vorne. An die Installation des Release schließt sich häufig eine lange Phase an, in der das Release stabilisiert werden muss.

Continuous Integration macht Hoffnung.

Heutzutage ist das Release in Produktion ein Problem. Vor nicht allzu langer Zeit setzten die Probleme schon wesentlich früher an: Die einzelnen Teams arbeiteten an ihren Modulen und vor dem Release mussten die verschiedenen Versionen zunächst integriert werden. Wenn die Module das erste Mal zusammen genutzt werden sollten, kompilierte das System oft noch nicht einmal. Tage oder gar Wochen später waren dann erst alle Änderungen integriert und kompilierten erfolgreich. Dann erst konnten die Deployments beginnen. Diese Prob-

leme sind heute meistens gelöst: Alle Teams arbeiten an einem gemeinsamen Versionsstand, der ständig automatisiert gemeinsam kompiliert und getestet wird. Dieses Vorgehen nennt man Continuous Integration. Die dafür notwendige Infrastruktur wird Kapitel 4 noch detailliert darstellen. Dass die Probleme dieser Phase gelöst sind, macht Hoffnung, dass die Probleme in den anderen Phasen hin zur Produktion ebenfalls lösbar sind.

Langsame und risikoreiche Prozesse

Die Prozesse in den späteren Phasen sind oft sehr komplex und aufwendig. Durch manuelle Prozesse sind sie außerdem langwierig und fehleranfällig. Das betrifft das Release in Produktion, aber auch die Phasen davor – also beispielsweise die verschiedenen Tests. Schließlich können gerade bei einem manuellen Prozess, der zudem nur ein paar Mal im Jahr ausgeführt wird, viele Fehler passieren – und das trägt zum Risiko bei.

Wegen des hohen Risikos und Aufwands werden Releases nicht besonders häufig in Produktion gebracht. Dadurch dauern die Prozesse noch länger, weil es keinen Übungseffekt gibt. Ebenso ist es schwierig, die Prozesse zu optimieren.

Schnell geht auch.

Auf der anderen Seite gibt es immer Möglichkeiten, im Notfall ein Release sehr schnell in Produktion zu bringen – wenn beispielsweise dringend ein Fehler behoben werden muss. Diese Prozesse umgehen aber jegliche Tests und damit die Sicherheitsnetze, die im normalen Prozess genutzt werden. Letztendlich ist das eine Risikomaximierung – die Tests werden ja aus einem Grund durchgeführt.

Also ist der normale Weg in die Produktion langsam und risikoreich – und im Notfall ist der Weg schnell, aber dafür noch risikoreicher.

2.3 Werte von Continuous Delivery

Mit der Motivation und den Ansätzen aus der Continuous Integration wollen wir den Weg für Releases in die Produktion optimieren.

Ein wesentliches Prinzip von Continuous Integration ist »If it hurts do it more often and bring the pain forward« – etwa: »Wenn es weh tut, tu es öfter und verlagere den Schmerz nach vorne.« Was sich wie Masochismus anhört, ist in Wirklichkeit ein Ansatz zur Problemlösung. Statt die Probleme bei den Releases zu umgehen, indem man möglichst wenig Releases in Produktion bringt, sollen die Prozesse so oft und so früh wie möglich durchgeführt werden, um sie möglichst schnell zu optimieren – in Bezug auf Geschwindigkeit und auf die Zuverlässigkeit. Continuous Delivery zwingt so die Organisation, sich zu ändern und eine andere Arbeitsweise zu adaptieren.

Überraschend ist der Ansatz eigentlich nicht: Wie schon erwähnt, kann jede IT-Organisation in kurzer Zeit einen Fix in Produktion bringen – und dabei werden meistens nur ein Teil der sonst üblichen Tests und Sicherheitsvorkehrungen ausgeführt. Das ist möglich, weil die Änderung nur klein ist und daher ein geringes Risiko hat. Hier zeigt sich ein anderer Ansatz für die Risikominimierung: Statt einer Absicherung über komplexe Prozesse und seltene Releases, kann man auch kleine Änderungen in Produktion bringen. Dieser Ansatz ist genau derselbe wie bei Continuous Integration (CI): Bei CI werden die Änderungen an der Software durch die einzelnen Entwickler und das Team ständig integriert und so häufig kleine Änderungen integriert, statt die Teams und Entwickler tage- oder wochenlang getrennt arbeiten zu lassen und die Änderungen am Ende zusammenzuführen – was meistens zu erheblichen Problemen führt, manchmal so sehr, dass die Software noch nicht einmal kompiliert werden kann.

Continuous Delivery ist aber mehr als »schnell und klein«. Continuous Delivery liegen verschiedene Werte zugrunde. Aus diesen Werten lassen sich die konkreten technischen Maßnahmen ableiten.

Regelmäßigkeit

Regelmäßigkeit bedeutet, Prozesse öfter durchzuführen. Alle Prozesse, die zur Auslieferung von Software notwendig sind, sollten regelmäßig durchgeführt werden – und nicht nur, wenn ein Release in Produktion gebracht werden muss. Beispielsweise ist es notwendig, Test- und Staging-Umgebungen aufzubauen. Die Testumgebungen können für fachliche oder technische Tests genutzt werden. Die Staging-Umgebung kann vom Endkunden genutzt werden, um die Features eines neuen Release auszuprobieren und zu evaluieren. Durch die Bereitstellung dieser Umgebungen kann der Prozess für den Aufbau einer Umgebung zu einem regelmäßigen Prozess werden, der nicht erst ausgeführt wird, wenn die Produktionsumgebung aufgebaut wird. Um diese Vielzahl von Umgebungen ohne allzu großen Aufwand zu erstellen, müssen die Prozesse weitgehend automatisiert werden. Regelmäßigkeit führt meistens zur Automatisierung. Ähnliches gilt für Tests: Es ist nicht sinnvoll, die notwendigen Tests bis kurz vor das Release zu verschieben – sie sollten lieber regelmäßig ausgeführt werden. Auch in diesem Fall erzwingt der Ansatz praktisch eine Automatisierung, um die Aufwände in Zaum zu halten. Regelmäßigkeit führt auch zu einer hohen Zuverlässigkeit – was ständig durchgeführt wird, kann zuverlässig wiederholt und durchgeführt werden.

Nachvollziehbarkeit

Alle Änderungen an der auszuliefernden Software und der dafür notwendigen Infrastruktur müssen nachvollziehbar sein. Es muss möglich sein, jeden Stand der Software und Infrastruktur zu rekonstruieren. Das führt zu einer Versionierung, die nicht nur die Software, sondern auch die notwendigen Umgebungen erfasst. Idealerweise ist es möglich, jeden Stand der Software zusammen mit der für den Betrieb notwendigen Umgebung in der richtigen Konfiguration zu erzeugen. Dadurch können alle Änderungen an der Software und an den Umgebungen nachvollzogen werden. Ebenso ist es einfach möglich, für die Fehleranalyse ein passendes System aufzubauen. Und schließlich können so Änderungen dokumentiert oder auditiert werden.

Eine mögliche Lösung des Problems ist es, dass Produktions- und Staging-Umgebung nur für bestimmte Personen zugänglich sind. Dadurch sollen Änderungen »kurz zwischendurch« vermieden werden, die nicht dokumentiert werden und nicht mehr nachvollziehbar sind. Außerdem sprechen Sicherheitsanforderungen und Datenschutz gegen den Zugriff auf Produktionsumgebungen.

Mit Continuous Delivery sind Eingriffe an einer Umgebung nur möglich, wenn ein Installationsskript geändert wird. Die Änderungen an den Skripten sind nachvollziehbar, wenn sie in einer Versionskontrolle liegen. Die Entwickler der Skripte haben auch keinen Zugriff auf die Produktionsdaten, so dass es mit Datenschutz ebenfalls keine Probleme gibt.

Regression

Um das Risiko bei der Auslieferung der Software zu minimieren, muss die Software getestet werden. Natürlich muss dabei die korrekte Funktion neuer Features sichergestellt werden. Aber viel Aufwand entsteht, um Regressionen zu vermeiden – also Fehler in eigentlich schon getesteten Softwareteilen, die durch Modifikationen eingeführt worden sind. Dazu müssen eigentlich alle Tests bei jeder Modifikation noch einmal ausgeführt werden – schließlich kann eine Modifikation an einer Stelle des Systems irgendwo anders einen Fehler erzeugen. Dazu sind automatisierte Tests notwendig, da sonst der Aufwand für die Ausführung der Tests viel zu hoch wird. Sollte ein Fehler es dennoch bis in die Produktion schaffe, so kann er immer noch durch Monitoring entdeckt werden. Idealerweise gibt es die Möglichkeit, auf dem Produktionssystem möglichst einfach eine ältere Version ohne den Fehler zu installieren (Rollback) oder einen Fix schnell in Produktion zu bringen (Roll forward). Eigentlich geht es also um eine Art

Frühwarnsystem, das über verschiedene Phasen wie Test und Produktion Maßnahmen ergreift, um Regressionen zu entdecken und zu lösen.

2.4 Vorteile von Continuous Delivery

Continuous Delivery bietet zahlreiche Vorteile. Je nach Szenario können die Vorteile unterschiedlich wichtig sein – und damit auch die Nutzung von Continuous Delivery beeinflussen.

2.4.1 Continuous Delivery für Time-to-Market

Continuous Delivery verringert die Zeit, die benötigt wird, um Änderungen in Produktion zu bringen. Dadurch ergibt sich auf der Geschäftsseite ein wesentlicher Vorteil: Es ist viel einfacher, auf Änderungen am Markt zu reagieren. Also verbessert Continuous Delivery das Time-to-Market.

Aber die Vorteile gehen weiter: Moderne Ansätze wie Lean Startup [1] propagieren einen Ansatz, der von der erhöhten Geschwindigkeit noch mehr profitiert. Im Wesentlichen geht es darum, am Markt Produkte zu positionieren und die Marktchancen auszuwerten und dabei möglichst wenig Aufwand zu investieren. Ganz wie bei wissenschaftlichen Experimenten wird definiert, wie der Erfolg des Produkts am Markt gemessen werden kann. Dann wird das Experiment durchgeführt und am Ende der Erfolg oder Misserfolg gemessen.

Ein Beispiel

Nehmen wir als konkretes Beispiel: In einem Webshop soll die Möglichkeit geschaffen werden, Bestellungen zu einem bestimmten Termin auszuliefern. Als erstes Experiment kann das Feature beworben werden. Als Indikator für den Erfolg dieses Experiments kann beispielsweise die Anzahl der Klicks auf einen Link in der Werbung genutzt werden. Zu diesem Zeitpunkt ist noch keine Software entwickelt – das Feature ist also noch nicht implementiert. Wenn das Experiment zu keinem erfolgversprechenden Ergebnis geführt hat, ist das Feature wohl nicht sinnvoll und es können andere Features priorisiert werden – ohne dass viel Aufwand investiert worden ist.

Feature implementieren und ausliefern

Wenn das Experiment erfolgreich war, wird das Feature implementiert und ausgeliefert. Auch dieser Schritt kann als Experiment durchgeführt werden: Metriken können helfen, um den Erfolg des Fea-

tures zu kontrollieren. Beispielsweise kann die Anzahl der Bestellungen mit einem festen Liefertermin gemessen werden.

Auf zum nächsten Feature!

Bei der Analyse der Metriken stellt sich heraus, dass die Anzahl der Bestellungen hoch genug ist – und interessanterweise die meisten Bestellungen nicht direkt zum Kunden, sondern zu einer dritten Person geschickt werden. Weitere Messungen ergeben, dass es sich offensichtlich um Geburtstagsgeschenke handelt. Basierend auf diesen Informationen kann nun das Feature weiter ausgebaut werden – beispielsweise mit einem Geburtstagskalender und Empfehlungen für passende Geschenke. Auch dazu müssen natürlich entsprechende Features geplant, implementiert, ausgeliefert und schließlich der Erfolg gemessen werden. Oder vielleicht gibt es auch Möglichkeiten, die Marktchancen dieser Features ganz ohne Implementierung zu evaluieren – durch Werbung, Kundeninterviews, Umfragen oder andere Ansätze.

Continuous Delivery führt zu Wettbewerbsvorteilen.

Continuous Delivery ermöglicht es, die notwendigen Änderungen an der Software schneller auszuliefern. Dadurch kann eine Firma schneller verschiedene Ideen ausprobieren und das Geschäftsmodell weiterentwickeln. Das führt zu einem Wettbewerbsvorteil: Da mehr Ideen ausgewertet werden können, ist es einfacher, die richtigen Ideen herauszufiltern – und zwar nicht aufgrund subjektiver Abschätzungen über die Markchancen, sondern anhand objektiv gemessener Daten.

Ohne Continuous Delivery

Ohne Continuous Delivery wäre das Feature für die festen Liefertermine geplant und beim nächsten Release ausgeliefert worden – das kann durchaus einige Monate dauern. Vorab hätte man wohl kaum gewagt, das Feature bereits zu bewerben – denn die lange Zeit, bis das Feature ausgeliefert wird, macht solche Werbung sinnlos. Wenn das Feature kein Erfolg geworden wäre, hätte man hohe Kosten bei der Implementierung gehabt, ohne dadurch einen Nutzen zu erzielen. Das Messen des Erfolgs wäre sicher auch in einem klassischen Modell möglich, aber die Reaktion würde deutlich länger dauern. Weiterentwicklungen wie die Unterstützung für Geburtstage wären noch später am Markt, denn dafür muss die Software noch einmal ausgeliefert werden und der langwierige Release-Prozess hätte noch ein zweites Mal durchlaufen werden müssen. Außerdem ist es fraglich, ob der Erfolg des Features ausreichend detailliert gemessen wird, um das Potenzial für die Features rund um Geburtstage zu erkennen.

Continuous Delivery und Lean Startup

Dank Continuous Delivery können also die Optimierungszyklen viel schneller durchlaufen werden, weil Features praktisch jederzeit in Produktion gebracht werden können. Das ermöglicht Ansätze wie Lean Startup. Das hat Auswirkungen auf die Geschäftsseite: Sie muss

schneller neue Features definieren und sie muss nicht mehr langfristig planen, sondern kann auf die Ergebnisse der aktuellen Experimente reagieren. Das ist in Start-ups besonders einfach, aber auch in klassischen Organisationen können solche Strukturen aufgebaut werden. Der Lean-Startup-Ansatz hat leider einen irreführenden Namen: Er beschreibt einen Ansatz, bei dem neue Produkte durch eine Serie von Experimenten am Markt positioniert werden – und dieser Ansatz ist natürlich auch in klassischen Unternehmen umsetzbar, nicht nur in Start-ups. Er kann auch genutzt werden, wenn Produkte klassisch ausgeliefert werden müssen – beispielsweise auf Datenträgern, mit anderen komplexen Installationsprozeduren oder als Teil eines anderen Produkts wie einer Maschine. Dann muss die Installation der Software vereinfacht oder idealerweise automatisiert werden. Außerdem muss ein Kundenkreis identifiziert werden, der gerne neue Versionen der Software ausprobieren will und dazu Feedback geben kann – also klassische Beta-Tester oder Power-User.

Auswirkungen auf den Entwicklungsprozess

Continuous Delivery hat Auswirkungen auf den Softwareentwicklungsprozess: Wenn einzelne Features in Produktion gebracht werden sollen, muss der Prozess das unterstützen. Einige Prozesse nutzen Iterationen von einem oder mehreren Wochen Länge. Am Ende jeder Iteration wird ein neues Release mit mehreren Features ausgeliefert. Das ist kein idealer Ansatz für Continuous Delivery, denn so können Features nicht alleine durch die Pipeline gebracht werden. Auch der Lean-Startup-Ansatz wird so erschwert: Wenn mehrere Features gleichzeitig ausgerollt werden, ist es nicht offensichtlich, welche Änderung die Messwerte beeinflusst. Nehmen wir an, dass die Auslieferung zu einem festen Liefertermin parallel mit einer Änderung der Versandpreise einher geht – welche der beiden Maßnahmen mehr Einfluss auf die höheren Verkaufszahlen hatte, ist nicht zweifelsfrei zu klären.

Also sind Prozesse wie Scrum, XP (Extreme Programming) und natürlich der Wasserfall hinderlich, denn die Prozesse liefern immer mehrere Features gemeinsam aus. Kanban [2] hingegen fokussiert darauf, ein einzelnes Feature durch die verschiedenen Phasen schließlich in Produktion zu bringen. Das passt ideal zu Continuous Delivery. Natürlich kann man die anderen Prozesse auch so modifizieren, dass sie die Auslieferung einzelner Features unterstützen – dann sind die Prozesse aber abgewandelt und zumindest nicht mehr nach Lehrbuch umgesetzt. Eine weitere Möglichkeit ist es, Features zunächst zu deaktivieren, um so zwar mehrere Features in einem Release gemeinsam auszuliefern, aber einzeln messbar zu machen.