Seminararbeit

Thema:

Continuous Delivery

Stefan Kruk
geboren am 14.08.1992
Matr.-Nr.: xxxxxxx

An der Fachhochschule Dortmund im Fachbereich Informatik erstellte
Seminararbeit
im Studiengang Softwaretechnik (Dual)

Betreuer: Dr. Kim Lauenroth

Fachbereich Informatik

Dortmund, 28. Juni 2016

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel werden zunächst die Grundlagen erläutert, welche für das Verständnis dieser Arbeit notwendig sind. Außerdem werden in den Grundlagen alle wichtigen Begriffe erklärt, die zum Verständnis des Themas beitragen und notwendig sind. Anschließend wird auf die zugrundeliegende Problemstellung eingegangen und darauf aufbauend auf das Ziel der Arbeit.

1.1 Grundlagen

Grundsätzlich ist das in dieser Arbeit behandelnde Thema für jede Person mit einer allgemeinen Informatikausbildung ohne weiteres zu verstehen. Es kann bei dieser Personengruppe, die Kenntnisse über grundsätzlichen Prozess einer Softwareentwicklung vorausgesetzt werden. Zudem kann vorausgesetzt werden, dass jede Person dieser Gruppe, der englischen Sprache mächtig ist. Zudem wird vorausgesetzt, das diese Personen wissen, was *Git* und *SVN*, bzw. andere Versionierungs-Werkzeuge sind und wie sie funktionieren. Trotzdem soll im weiteren Verlauf einige Begriffe genauer erklärt werden.

Time-to-Market

Unter dem Begriff Time-to-Market wird die Zeit von der Produktentwicklung bis zur Auslieferung auf dem Markt verstanden.[1] In dieser Zeit müssen Kosten für die Erstellung/Entwicklung aufgebracht werden, es wird in dieser Zeit jedoch keine Umsätze erzeugt. Daher strebt jedes Unternehmen eine möglichst geringe Time-to-Market Zeit an. Insbesondere wenn es um Wettbewerb geht, muss diese Zeit kurz gehalten werden.

[1] Vergleich mit [2]

Delivery

Delivery (zu deutsch Ausliefern) beschreibt das Ausliefern (das Verteilen) von Artefakten. Dabei kann das Artefakt eine ganze Applikation oder nur ein Service in einer Service Orientierten Architektur sein.

Deployment

Unter Deployment (zu deutsch Softwareverteilung) versteht man das installieren, eines Artefaktes. Auch hier kann ein Artefakt eine ganze Applikation oder nur ein Service sein.

Software-as-a-Service (SaaS) „Das Modell der Software-as-a-Service (SaaS) ist Teil des großen Konzeptes „Cloud-Computing“. SaaS ist dem Konzept des Application-Service-Provider (ASP) sehr ähnlich ist und funktioniert, indem der Kunde eine bereITgestellten Software-Anwendungen online nutzen kann, wie eine Art Dienstleistung. Für die Nutzung der Anwendung zahlt er Gebühren an den Provider, der die Software für ihn bereitstellt. Der Kunde kann hierbei einen monatlichen Betrag wählen oder die Software on Demand, also je nach Bedarf, nutzen und zahlen.“[7]

1.2 Problemstellung

Durch fortschreitende Technologien, werden Arbeitsabläufe immer automatisierter. Dadurch wird die Zeitspanne für Time-to-Market immer relevanter und kürzer, wodurch der Wettbewerbsdruck wächst. Daher ist es wichtig eine kurze TTM zu haben.

fiktives Beispiel

Eberhard Wolff beschreibt in [13, S. 2 ff.] einen Fall eines fiktiven E-Commerce Unternehmens. Das Unternehmen hatte nur eine große Software, den E-Commerce Shop. Durch neue Angebote und das dauerhaft ändernde Interesse der Kunden mussten neue Funktionen regelmäßig und in möglichst kurzen abständen dem Kunden zugänglich gemacht werden. Dies wurde jedoch durch die Tatsache behindert, dass die Software über die Jahre gewachsen ist und das erneute Ausliefern der Software für eine Funktion sich nicht lohnte. Daher wurde nur einmal im Monat neu Deployed. Der Prozess wurde außerdem dadurch behindert, dass die Qualitätssicherung zwar ein Teil der Softwareentwicklung war, jedoch Tests nur manuell ausgeführt worden sind, wodurch regelmäßig Fehler übersehen wurden.

die Software wurde schließlich mit Fehlern ausgeliefert und es stellte sich erst am nächsten Tag, oder schlimmer nach einer Woche, heraus, dass sie nicht einwandfrei funktionierte. Entwickler mussten ihre Arbeit unterbrechen und den Fehler finden und beheben. Da jedoch ein wenig Zeit vergangen ist, seit dem die Entwickler an diesem Teil des Codes gearbeitet haben, müssen sie sich erst wieder einarbeitet, bis sie den Fehler finden und beheben können.

Das Unternehmen hatte eine große TTM-Zeit und dadurch hohe kosten. Zusätzlich entstehen fehlerhafte Releases wodurch zusätzliche Kosten bzw. Einbußen entstehen.

Reales Beispiel

Als Reales Beispiel dient das Unternehmen *Rally Software*.

> "When Rally Software was founded in April 2002 the engineering strategy was to ship code early and often. The company founders had ben successfully practicing Agile and Scrum, with its well-known patterns of story planning, sprints, daily stand-ups and retrospectives. They adopted an eight week release cadence that propelled the online software as a service (SaaS) product forward for more than seven years."[11]

Weiter steht in [11]: Rally Software deployed alle acht Wochen Code in Ihre SaaS Umgebung. Sieben der acht Wochen wird für die Ausführung des planning-sprint cycles genutzt und die verbleibende Woche für das 'härten' des Codes. Hiermit ist die Testphase gemeint. Dabei klicken sich alle Angestellten durch die Anwendung auf der suche nach Fehlern. Wurde ein Fehler gefunden, wird dieser direkt an die Entwicklungsabteilung gegeben. Die wiederum versuchten den Fehler so schnell wie möglich zu beheben. Ist der Release Zeitpunkt gekommen, werden die Datenbanken manuell migriert und die komprimierten WAR Dateien durch das Netzwerk kopiert.

> "If anything went wrong we could lose a [day] to fix the failure."[11]

Am nächsten Tag standen die Entwickler früh im Büro, noch bevor der erste user-traffic-spike, war um die Software zu überwachen.

> "After a successful release Rally would celebrate. The event was recorded for history with a prized release sticker and the eight week cycle began again."[11]

Dieses Beispiel verdeutlicht sehr stark, was das ausführen von manuellen Tests bewirken kann und wie glücklich die Mitarbeiter sind, wenn ein Realese ohne Fehler durchgeführt wurde.

1.3 Ziel der Arbeit

In dieser Arbeit soll Continuous Delivery genauer erläutert und dabei folgende Leitfragen beantwortet werden:

1. Was ist Continuous Delivery?
2. Warum sollte man Continuous Delivery einsetzten?
3. Welche Voraussetzungen müssen gegeben sein, um Continuous Delivery einzusetzen?
4. Was ist Continuous Integration und wie gehört es zu Continuous Delivery?
5. Was ist eine Continuous Delivery Pipeline?
6. Welche Werkzeuge werden benötigt für die Verwendung von Continuous Delivery?
7. Wie kann man Continuous Delivery in ein bestehenden Entwicklungsprozess einbinden?

Die erste Leitfrage wird den Begriff Continuous Delivery erläutern. In diesem Zusammenhang wird zunächst einmal der Begriff *Continuous Integration* eingeführt und mit der vierten Leitfrage genauer erläutert. Zudem wird erläutert welche Ziele mit Continuous Delivery verfolgt werden.

Darauf aufbauend wird geklärt warum Continuous Delivery eingesetzt werden soll. Konkret werden die Vorteile von Continuous Delivery anhand von Realen Situationen dargelegt und erläutert. Zudem wird erläutert, welche Vorteile sowohl Entwickler, als auch das Unternehmen selbst, von dem Schritt, den Entwicklungsprozess auf Continuous Delivery umzustellen, haben.

Anschließend wird die dritte Leitfrage geklärt werden. Es wird erläutert, welche Voraussetzungen gegeben sein müssen, bzw. welche Schritte notwendig sind, damit Continuous Delivery eingeführt werden kann.

Nachdem erläutert wurde was Continuous Delivery ist, warum ein Unternehmen die Softwareentwicklung auf diesen Prozess umstellen sollte und welche Voraussetzungen dafür gegeben sein müssen, wird schließlich der Begriff *Continuous Integration* weiter erläutert. Zudem wird genauer dargelegt, wie *Continuous Integration* mit Continuous Delivery zusammenhängt. Es wird außerdem erläutert, in wie weit Tests verändert werden müssen, damit sie in den Continuous Integration Prozess mit aufgenommen werden können.

Mit der fünften Leitfrage wird der Begriff *Continuous Delivery Pipeline* eingeführt und erläutert. Dies wird schließlich an ein Beispiel weiter verdeutlicht.

Da nun ein Verständnis von *Continuous Delivery* vorhanden ist, werden Werkzeuge eingeführt, welche zum Beispiel die Frage genauer erklären: “Was ist ein *Continuous Integration Server*“. Zudem wird noch einmal darauf eingegangen wie welche Werkzeuge eingesetzt werden können, um Tests in den Continuous Integration Prozess aufnehmen zu können.

Abschließend wird ein bestehender Entwicklungsprozess zu Continuous Delivery überführt. Dabei werden die zuvor erklärten Werkzeuge eingesetzt, um eine Continuous Delivery Pipeline aufzubauen.

Kapitel 2

Systematische Literaturrecherche

Die Systematische Literaturrecherche stellt die Basis der Quellen und Informationen, des in Kapitel 3 Continuous Delivery vorgestellten Inhalts dar.
In diesem Kapitel wird daher aufgezeigt, wie die herangezogenen Quellen und Informationen ermittelt, welche Auswahl- und Ausschlusskriterien festgelegt wurden und was für Ergebnisse die entsprechenden Suchanfragen gebracht haben.

2.1 Auswahlkriterien und Suchbegriffe

Um die Auswahl der Literaturen zu Filtern, wird zunächst allgemeine Auswahl- und Ausschlusskriterien definiert, mit denen die im Rechercheprotokoll angegebenen Suchergebnisse begründet werden. Anschließend wird der Zugrundlegende Ansatz (P.I.C.O.C.) genauer erläutert und darauf aufbauend Suchbegriffe in Deutsch und Englisch definiert.

2.1.1 Inhaltliche Auswahlkriterien

Folgende Inhaltliche Auswahlkriterien wurden für die Recherche festgelegt: spacing

P1 Dokument ist über oder hat direkten Bezug zu Continuous Delivery

P2 Dokument beschreibt die Einsatzmöglichkeiten von Continuous Delivery

P3 Dokument beschreibt wichtige Technologien für den Einsatz von Continuous Delivery

P4 Dokument kann zum beantworten der Leitfragen genutzt werden.

Mit Hilfe der Auswahlkriterien wird im Rechercheprotokoll die Relevanz der gefundenen Materialien begründet. Sie werden über die Buchstaben a) bis c) referenziert.

2.1.2 Inhaltliche Ausschlusskriterien

Folgende Inhaltliche Ausschlusskriterien wurden für die Recherche festgelegt. spacing

N1 Dokument ist zu allgemein und/oder hat nur am Rande etwas mit dem Thema zu tun (Bsp.: Enthält den Begriff nur in Referenzen)

N2 Dokument ist unvollständig.

N3 Inhalt des Dokumentes muss kostenpflichtig erworben werden oder ist aus anderen Gründen nicht einsehbar

N4 Inhaltsangabe, Einleitung, Fazit oder Abstract sind nicht Aussagekräftig bzw. lassen keine Hinweise auf den Einsatz oder der Beschreibung von Continuous Delivery zu

N5 Inhalt trägt nicht zur Beantwortung der Leitfragen bei. (Bsp.: Das Dokument ist ein Erfahrungsbericht, enthält jedoch keine konkreten Erläuterungen oder Verweise auf Dokumente, die den Prozess genauer beschreiben.)

N6 Dokument ist nicht in Deutsch oder Englisch (Material kann aufgrund der Sprachbarriere nicht verwendet werden)

N7 Inhalt des Dokumentes muss kostenpflichtig erworben werden oder ist aus anderen Gründen nicht einsehbar.

N8 Dokument beschreibt wie ein Werkzeug und/oder Framework aufgebaut ist und funktioniert.

Mit Hilfe der Ausschlusskriterien wird im Rechercheprotokoll die Irrelevanz der gefundenen Materialien begründet. Sie werden über die Buchstaben d) bis j) referenziert.

2.1.3 P.I.C.O.C.

Die Suchbegriffe werden mit Hilfe des PICOC-Ansatzes (siehe [10]) ermittelt.

Population

Die Population, zu Deutsch etwa "Bevölkerung", beschreibt eine Teilmenge von relevanten Personen, wie Tester, Manager, Novizen oder Experten. Aber auch Applikationsfelder wie IT-Systeme, Command und Control Systeme oder Industrielle Gruppen wie Telekommunikations- oder kleine IT-Unternehmen.

Bezüglich der Population werden hier die sogenannten SStackeholder"weggelassen. Da das Thema jedoch stark mit dem Thema DevOp-Teams tangiert, wird hier zusätzlich zu diesem Thema Suchanfragen gestellt. Eine Einschränkung des Thema auf DevOps besteht jedoch nicht.

Intervention

Bei der Intervention handelt es sich um die eingesetzten Methoden, Werkzeugen, Technologien oder Prozeduren für ein bestimmtes Problem. Im Rahmen von Continuous Delivery bedeutet dies, die Werkzeuge die nötig sind um eine Continuous Delivery Pipeline aufbauen und durchführen zu können. Grundlegend sollen Werkzeuge erläutert werden, welche zur Verwaltung, Bauen, Testen und Ausliefern dienen.

Comparison / Vergleich

Der Vergleich (Comparison) beschreibt die Werkzeuge einer Kontrollgruppe, welche mit denen aus der Intervention verglichen werden. Konkret werden die Gruppen aufgrund der TTM verglichen. Anzumerken sei, dass die Vergleichsgruppe ohne einen automatisierten Prozess Softwareprojekte durchführt.

Outcomes / Auswirkung

Bei der Auswirkung soll mit Hilfe von Zahlen und Faktoren der Vergleich erläutert werden zwischen der Intervention und der Kontrollgruppe. Hier kann zum Beispiel die Zeitspanne des Time-to-Market verglichen werden, was wiederum einen Einblick in die Kosten für die Implementierung/Auslieferung eines Produktes/einer Funktion gibt.

Context / Kontext

Der Kontext ist hier die Software-Entwicklung in Bezug auf Aufwand und Kosten der Produktion. Bei Continuous Delivery spielt ebenfalls der Aufbau des Teams eine Rolle. Wie bei der Population spielen hier DevOp-Teams eine zentrale Rolle. Zusätzlich sei hier auf Intervention und die dortigen Werkzeuge verwiesen.

2.1.4 Suchbegriffe

Im Nachfolgenden sind für jeden, der in Abschnitt 2.1.3 P.I.C.O.C genannten Aspekte Synonyme in Deutsch und Englisch festgehalten, welche die Basis für die verwendeten Suchanfragen bilden.

Kategorie	**Deutsche Begriffe**	**Englische Begriffe**
Population	DevOps	
Intervention	• Continuous Delivery • Docker • Jenkins • Deployment • pipeline • DevOPs	
Comparison	• Manuel (Deployment)	
Outcomes	• Kosten • Tests • Zeit	• costs • tests • time / duration
Context	• Technik • Prinzipien • Praxis • Anwendung	• techniques • principles • practice • usage

Der Begriff Deployment is sowohl in Intervention und Comparison, da dieser ein zentraler Begriff in beiden Mengen ist und unterschiedlich verwendet werden kann.

2.1.5 Zusätzliche Anmerkungen zur Recherche

Für diese Arbeit wurden zusätzlich folgenden Einschränkungen, für die Durchführung der systematischen Suche festgelegt:

Zugänglichkeit Materialien müssen entweder öffentlich oder für den Personenkreis, für die diese Arbeit angefertigt wird, ohne weitere Einschränkungen, wie ein notwendiges Login, zugänglich sein. da ansonsten der Beschaffungsaufwand zu hoch ist und die Materialien nicht für andere Studenten bzw. den Dozenten zugänglich wären.

Auswahl der Materialien Materialien werden anhand der Inhaltsübersicht, Einleitung, Fazit oder eines Abstracts ausgewählt, da ein einlesen in einzelne Kapitel zu viel Zeit beanspruchen würde.

Suchergebnisse Bei auffinden von Großen Mengen bei der Suche, wird zunächst versucht

durch eventuelle Filtermöglichkeiten, die Relevanz der Materialien zu ordnen und die ersten 20 Resultate begutachtet. Dabei wird die Qualität der Ordnung und die Effizienz des zugrunde liegenden Algorithmus der Suchmaschine überlassen. Sollten keine Filtermöglichkeiten vorhanden sein, wird anhand der Kurzbeschreibungen und der Titel die ersten 20 besten Treffer ausgewählt.

2.2 Quellen und Suchanfragen

2.2.1 Suchstring

In Abschnitt 2.1.4 Suchbegriffe wurden Suchbegriffe festgelegt, auf denen die Literaturrecherche basiert. Diese werden zu nächst mit einem “ODER"bzw. “OR“ verknüpft. Im Zweiten Schritt werden die Suchbegriffe mit einem “UND‘"bzw. ÄND“ verknüpft.

Da bestimmte Begriffe mit verschiedenen Kontexte in Verbindung gebracht werden können. Wird zunächst ein Suchstring aufgebaut, der als Erstes Element (“Continuous Delivery“ OR “Deployment“) besitzt. Nachfolgend werden nun die einzelnen Suchstrings aufgelistet, die verwendet wurden, um die systematische Literaturrecherche durchzuführen. Da Suchmaschinen einen komplexen Algorithmus aufweisen, wird ebenfalls davon ausgegangen, dass auch eine aneinander reihen von den in 2.2.1 Suchstring Ausdrücken (immer mit dem Führenden “Continuous Delivery“ Oder “Deployment‘") zum gewünschten Ergebnis führt. Dies beruht auf den Eigenschaften moderner Suchalgorithmen. Nach einer ersten suche, hat sich ergeben, dass einige Begriffe irreführend für die Suchmaschinen sind, wodurch Materialien gefunden wurde, welche absolut nichts mit dem Thema zu tun haben. Daher sind nur die folgenden Suchanfragen von Bedeutung.

$S_{Komplex}$ = ("’Continuous Delivery"‘ OR "’Deployment"‘)
AND
(“SSH“ OR “FTP“ OR “Deployment“ OR“Kosten“ OR “costs“ OR “Tests“ OR “Zeit“ OR “time“ OR “duration“ OR “Fehler“ OR “error“ OR “Qualitätssicherung“ OR “quality assurance“ OR “Technik“ OR “techniques“ OR “Prinzipien“ OR “principles“ OR “Praxis“ OR “practice“ OR “Anwendung“ OR “usage")

$S_{deployment}$ = ("’Continuous Delivery"‘ OR "’Deployment"‘)
AND
(“Deployment“)

$S_{pipeline}$ = ("'Continuous Delivery"‘ OR "'Deployment"‘)
AND
(“Pipeline“)

S_{devops} ("'Continuous Delivery"‘ OR "'Deployment"‘)
AND
(“DevOps“)

Da nach [10, vgl. S. 26] auch ein einfacher Suchstring effektiv sein kann, wurde neben den bereits genannten Suchstrings noch ein weiterer abgeleitet, welcher lediglich aus dem Thema dieser Arbeit besteht.

$S_{einfach}$ "Continuous Delivery"

Dieser Suchstring wird ausschließlich dazu verwendet, bei einer Suche, den Dokumenten bzw. den Publikations Titel, nach vorkommen dieser Begriffe zu durchsuchen.

2.2.2 Quellen

In [10] wurden einige elektronische Standartquellen der Informatik genannt. Nachfolgend, werden diese noch einmal aufgelistet und kurz begründet, warum jeweilige Quellen gewählt bzw. nicht gewählt wurde.

ACM Digital Library

Dokumente aus der *ACM Digital Library* sind nicht frei zugänglich und müssen zunächst erworben werden, bevor sie gelesen werden können.

Citeseer Library

Eine erste Literaturrecherche in der *Citeseer Library* ergab keine nennenswerten Einträge. Es wurden Dokumente zu Themen zurückgeliefert, welche nur ansatzweise etwas mit dem Thema zu tun haben und nach **N5** ausscheiden.

EI Compendix

Bei *EI Compendix* ist zunächst eine Registrierung erforderlich, bevor Dokumente angesehen werden können.

IEEExplore

Bei IEEExplore handelt es sich um eine Digitale Datenbank für wissenschaftliche Literaturen. Sie ist für jeden Informatiker, bzw. für jedes Informatik-Thema eine wichtige Ressource für die Literaturrecherche. Sie ist einer der wichtigsten Datenbanken für wissenschaftliche Arbeiten im Bereich Informatik, Elektrotechnik und Elektronik.

Informationen bezogen von: http://ieeexplore.ieee.org/xpl/aboutUs.jsp

Inspect

Bei *Inspect* ist zunächst eine Registrierung erforderlich, bevor Dokumente angesehen werden können.

Google Scholar

Google Scholar hat ein großes Angebot und die meisten PDFs sind frei zugänglich. Jedoch ist hier zu beachten, dass *Google Scholar* auch auf Dokumente verweist, die zum Beispiel im *IEEEXplore* liegen.

ScienceDirect

Die Dokumente in *ScienceDirect* sind nur teilweise frei zugänglich.

2.2.3 Gewählte Quellen

Nachstehende Tabelle zeigt noch einmal die gewählten und nicht gewählten Quellen:

Quelle	gewählt
ACM Digital Library	✘
Citeseer Library (citiseer.ist.psu.edu)	✘
EI Compendex (www.engineeringvillage2.org)	✘
IEEExplore	✔
Inspec (www.iee.org/Publish/INSPEC/)	✘
Google scholar (scholar.google.com)	✔
ScienceDirect (www.sciencedirect.com)	✘

2.3 Ergebnisse der Recherche

In der Nachfolgenden Tabelle sind, nach Suchmaschine geordnet, die Ergebnisse der systematischen Literaturrecherche. Es wird der Suchstring, die Anzahl der gefundenen Ergebnisse, die Anzahl er betrachteten Ergebnisse, die Anzahl der relevanten Ergebnisse, die Anzahl der gewählten Ergebnisse und das Datum an dem die Suche durchgeführt worden ist. Das dazugehörige Rechercheprotokoll ist im Anhang zu finden.

IEEExplore					
Suchstring	gesamt	betrachtet	relevant	gewählt	Datum
$S_{komplex}$	201.863	50	0	0	23.06.2016
$S_{deployment}$	12.092	50	1	1	23.06.2016
$S_{pipeline}$	10	10	6	6	23.06.2016
S_{devops}	11	11	4	1	23.06.2016
$S_{einfach}$	40	40	10	6	23.06.2016

Google scholar					
Suchstring	gesamt	betrachtet	relevant	gewählt	Datum
$S_{komplex}$	1.580	50	0	0	23.06.2016
$S_{deployment}$	224.00	50	1	1	23.06.2016
$S_{pipeline}$	80.800	50	2	1	23.06.2016
S_{devops}	1.230	50	5	1	23.06.2016
$S_{einfach}$	3.180.000	50	0	0	23.06.2016

Der Begriff *Continuous Delivery* ist stark mit dem Bereich Biologie verbunden. Gibt man diesen Begriff in Google Scholar ein, so sind die meisten Einträge aus der Biologie.

Kapitel 3

Continuous Delivery

Continuous Delivery beschreibt ein Prozess, der mit Hilfe verschiedener Werkzeuge, den Softwareauslieferungsprozess verbessern soll. Durch Techniken wie Continuous Integration, automatisieren von Tests und kontinuierlichen Installationen soll qualitativ hochwertige Software erstellt werden.

> "Officially, we describe continuous delivery as the ability to release software whenerver we want. This could be weekly or daily deployments to production; it could mean every check-in goes straight to production"[11]

Es soll nicht nur qualitativ hochwertige Software erstellt werden, sondern auch die Möglichkeit geboten werden, Software zu jederzeit zu veröffentlichen (releasen).

Auch Martin Fowler schreibt [5]:

> "Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time. [..] You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running automated tests on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software will work in production. To do this you use a Deployment Pipeline."

Continuous Delivery soll den Softwareauslieferungsprozess verbessern, welche Vorteile dieses im einzelnen mit bringt, wird als nächstes beschrieben.

3.1 Warum Continuous Delivery?

Der Softwareentwicklungsprozess unterliegt einem ständigen Wandel. Viele Unternehmen arbeiten nach agilen Vorgehensmodelle wie Scrum, um Software zu entwickeln. Andere Unternehmen verwenden ggf. andere Vorgehensmodelle wie das Wasserfallmodell, jedoch alle Unternehmen arbeiten nach dem gleichen Prinzip: *Sie wollen Software möglichst qualitativ Hochwertig und Preis günstig Entwickeln*. Diese beiden Aussagen widersprechen sich jedoch oft, denn damit Software qualitativ Hochwertig erstellt werden kann, müssen ausführliche und gründliche Tests durchgeführt werden, welche die Qualität der Software sicherstellt. Oft werden diese Test manuell durchgeführt und nehmen viel Zeit in Anspruch, weshalb Unternehmen diesen Teil der Softwareentwicklung meist verkürzen wollen um Geld zu Sparen. In den meisten Unternehmen existiert zudem ein fester Zeitpunkt, an dem die Software fertig gestellt sein muss.

Ein weiteres Problem besteht darin, wenn Software nur jeden Monat ausgeliefert wird. Enthält die Produktive Software einen Fehler, dauert es im schlimmsten Fall einen Monat, bis dieser behoben wird.

Continuous Delivery ermöglicht durch automatisierte Tests zum einen die Zeit, die für die Durchführung der Tests, benötigt wird zu reduzieren und zum anderen die Tests zu standardisieren. Das heißt, ein Tests kann beliebig oft ausgeführt und immer das selbe Ergebnis erwartet werden. Dadurch ist es möglich die selben Tests nach jeder Änderung durchzuführen und zu überprüfen ob Fehler aufgetreten sind oder nicht.

Jedoch spielt auch der Erfolg des Produktes eine große Rolle. Es wurde zuvor erwähnt, dass Unternehmen oft Geld einsparen wollen, in dem sie den Testzeitraum verkürzen, wodurch die Qualität leidet.

> "The Time to Market of a product critically affects its success especially when talkng about technologies, which have to be delivered while they are still new. In such environemnt, then, what really differentiates the product on the market is not only the product itself, or its quality, but also the speed at which it can evolve: for a company, taking the product to market fast means to win over the competitors and being always aligned with new tendencies."[1]

Besitzt ein Produkt eine zulange Time-to-Market Zeitspanne ist die Wahrscheinlichkeit hö-

her, dass der Erfolg eines Produktes nicht so hoch ist, wie wenn die Zeitspanne kürzer wäre. Zudem spielen Konkurrenzunternehmen eine wichtige Rolle. ist die Time-to-Market Zeitspanne zu groß, kann ein anderes Unternehmen ein gleichwertiges Produkt auf den Markt bringen und so den Erfolg des eigenen Produktes noch weiter verringern. Durch Continuous Delivery kann die Time-to-Market Zeitspanne verkürzt und neue Produkte/Funktionen schneller auf dem Markt etabliert werden.

3.1.1 Die Entwickler

> "Selling continuous delivery to our development team was relatively easy. The Software engieerns [..] are eager to experiment with new technologies and methodologies. They understood that smaller batch size would lead to fewer defects-in production as we limited the size of code changes and garnered fast feedback from our systems." [11]

Durch Continuous Delivery erhält der Entwickler zum einen die Möglichkeit mit neuen Technologien zu experimentieren, zum anderen erhalten sie direktes Feedback vom System ob die durchgeführten Änderungen funktionieren oder den Code zerbrechen.

3.1.2 Das Unternehmen

Es gibt verschiedene Gründe für die Einführung von Continuous Delivery. Rally Software zum Beispiel hatte nur alle zwei Monate realeased.

> "Realeasing every two months is painful for a number of reasons: (1) when you miss a release you potentially have to wait another eight weeks to dliver your features to customers; [..]"[11]

Eines der größten Probleme ist das verpassen von Release-Zeitpunkten. Es muss 4 Monate gewartet werden, bis eine Funktion in Produktion geht. Durch Continuous Delivery wird diese Zeit erheblich verringert. Funktionen müssen nicht zu einem fixen Zeitpunkt fertig gestellt werden, sondern können, sobald diese fertiggestellt, direkt im Continuous Integration Prozess getestet und released werden.

3.2 Voraussetzungen für Continuous Delivery

Bisher wurde beschrieben welche Vorteile Continuous Delivery bietet, jedoch nicht welche Voraussetzungen gegeben sein müssen, damit es eingeführt werden kann.

> "The improved test coverage, rapid feedback cylces, scrutiny of monitoring systems, and fast rollback mechanisms result in a far safer environment for shipping code. But it is not free; it is not pailess. It is important to have the right level of sponsorship before you begin."[11]

Konkret heißt das, bevor Continuous Delivery eingeführt werden kann, muss zunächst einmal geklärt werden, warum man diesen Schritt durchführen möchte und welche Ziele damit verfolgt werden sollen. Argumente wie: *Es ermöglicht schnellere releases.* oder *Continuous Delivery bringt uns mehr Geld.* sind jedoch ein falscher Ansatz und sollten vermieden werden.

Vorallem müssen die beteiligten Personen überzeugt werden können. Dafür ist es umso wichtiger zu wissen, wohin man möchte und mit welchen mitteln.

> "When you present the vision to these stakeholders it is important to have the mission clarified. [..] You must set clear objectives and keep progress steering toward the rigth direction. There are many side paths. experiments and "shiny" things to distract you on the journey towards continuous delivery. It is easy to become distracted or waylaid by these."[11]

Zudem darf man sich nicht vom Ziel ablenken lassen, sondern sollte zunächst einmal auf dieses zusteuern. Hat man das Ziel, welches durch Continuous Delivery erreicht werden soll, erreicht, kann man versuchen den Prozess stetig zu verbessern. Jedoch ist auch hier Vorsicht geboten.

Continuous Delivery kann beliebig verändert und erweitert werden, jedoch sollten nie Änderungen durchgeführt werden, die keinen Nutzen haben bzw. den eigentlichen Prozess behindern. Dies würde den Prozess unübersichtlich und ggf. langsamer machen.

3.3 Continuous Integration

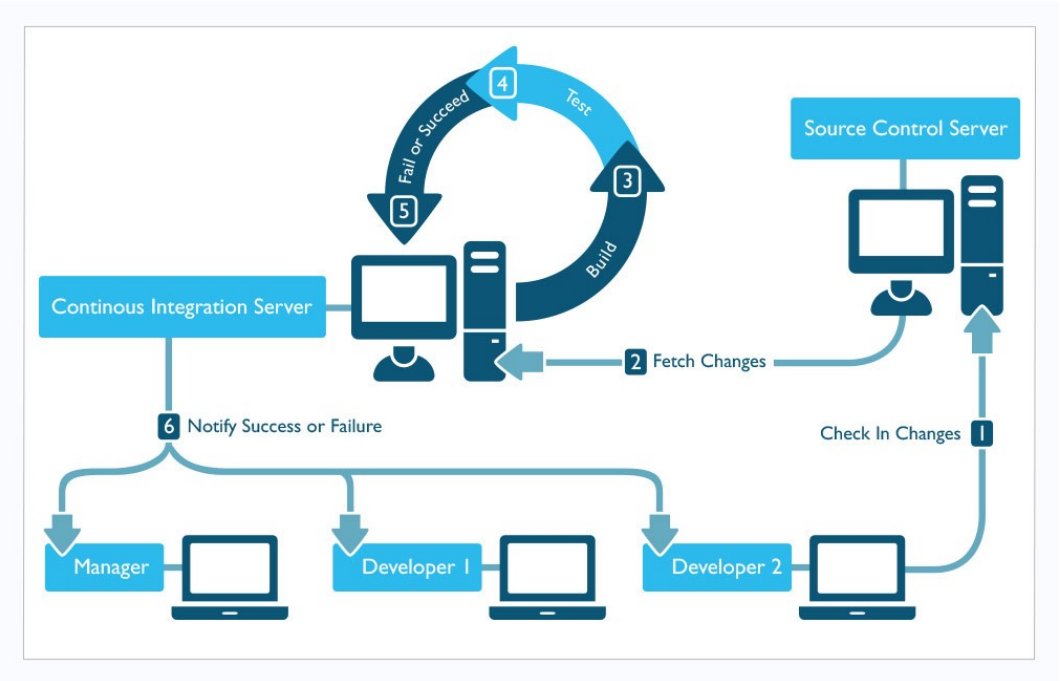
> "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced

> integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage. [..] At all times you know where you are, what works, what doesn't, the outstanding bugs you have in your system "[4].

Das ist die Aufgabe von Continuous Integration. Um dies zu erreichen ist es notwendig jede Teilaufgabe innerhalb der Entwicklung in den Prozess mit einzubinden.

> "CI literally started to change how companies looked at Build Managment, Release Management, Deployment Automation, and Test Orchestration [..]."[1]

Das Ziel ist es so früh wie möglich Bugs zu erkennen und diese zu beheben. Dafür muss jeder Prozess automatisiert werden. Vor allem muss es möglich sein, Tests automatisiert durchführen zu können. Eine einfache Continuous Integration zeigt Abbildung 3.1 Continuous Integration:



Quelle: https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html

Abbildung 3.1: Continuous Integration

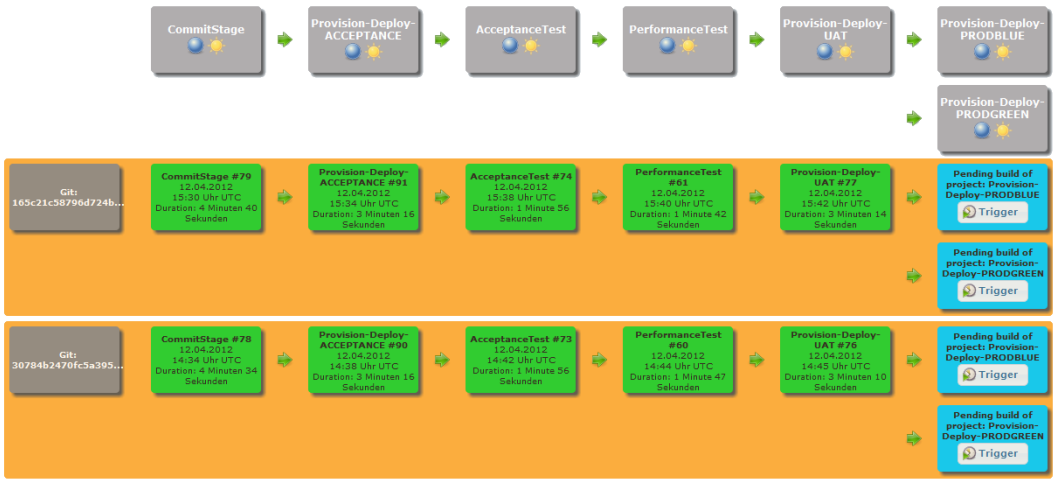
Wie in Abbildung 3.1 Continuous Integration zu erkennen ist, sorgt ein zentraler “Continuous Integration Server“ für das Bauen und Testen der Software und informiert die zuständigen Entwickler über den Status. In der Regel wird dies bei jeder Code Änderung durchgeführt, sodass direkt erkannt wird, ob eine Änderung des Codes zu einem Erfolg oder einem Fehlschlag führt.

> “The concept of Continuous Integration (CI) was a first step that significantly sped up the lifecycle of a product, pushing developers to commit/integrate more frequently to a shared repository, triggering automated unit-tests after each commit; as direct consequence, this helped to detect problems richt after a bad commit and reduced the necessity of back-tracking to individuate the issue in changes happend far away in time.“[1, in Introduction]

Continuous Delivery ist eine natürliche Erweiterung von Continuous Integration. Trotzdem unterscheiden sich die beiden Begriffe nicht wirklich von einander. Während bei der Continuous Integration wert darauf gelegt wird, Software möglichst Fehlerfrei zu erzeugen, wird bei Continuous Delivery darauf wert gelegt, Software möglichst regelmäßig zu deployen. Continuous Delivery beinhaltet Continuous Integration und erweitert diese um das ausliefern.

3.4 Continuous Delivery Pipeline

Es wurden bisher die Vorteile von Continuous Delivery erläutert, wie funktioniert jedoch Continuous Delivery im einzelnen? Die einzelnen Schritte werden durch die *Continuous Delivery Pipeline* beschrieben. Dabei wird der Durchlauf der Pipeline automatisch durchgeführt. Die Continuous Delivery Pipeline integriert zum einen Die Folgende Abbildung zeigt eine mögliche Pipeline. Das Deployen in die Produktionsumgebungen “PRODBLUE“ und “PRODGREEN“ muss in diesem Beispiel jedoch manuel, durch klicken auf Trigger, erfolgen.
Wie man in der Abbildung sehen kann beinhaltet die Pipeline alle nötigen schritte, welche zuvor in Continuous Integration besprochen wurden. Hier sei noch einmal erwähnt, dass Continuous Integration alle Prozesse, bis auf den letzten (das Deployment), beinhaltet. Continuous Integration und Delivery unterscheiden sich, wie schon zu vor erwähnt, nur beim letzten Schritt, dem Deployen. Dieser wird jedoch bei Continuous Delivery Manuel ausgeführt. Unter ausgeführt ist hierbei zu verstehen, dass ein Prozess, hier durch einen



Quelle: https://blog.codecentric.de/en/2012/04/continuous-delivery-in-the-cloud-part1-overview/

Abbildung 3.2: Continuous Delivery Pipeline

klick, gestartet wird, welcher die Software, bzw. das Artefakt, automatisch in die Produktion bringt.

3.5 Werkzeuge für Continuous Delivery

Es wurde bisher erläutert was Continuous Delivery ist, welches Vorteile es hat und welche Voraussetzungen gegeben sein müssen, damit Continuous Delivery eingeführt werden kann. Es wurde ebenfalls mit Abschnitt 3.3 Continuous Integration und Abbildung 3.1 Continuous Integration der Begriff *Continuous Integration Server* eingeführt. Es wurde jedoch noch nicht geklärt was dieser Server genau ist und welche konkreten Aufgaben dieser besitzt. Dies soll in diesem Abschnitt geklärt werden. Zudem sollen weitere Werkzeuge vorgestellt werden, der für den *Continuous Delivery Prozess* nützlich sein können. Es soll jedoch nicht die konkrete Funktionsweise von Werkzeugen erläutert werden. Einige Werkzeuge werden zum besseren Verständnis genauer erklärt als andere. Es sei hier erwähnt, dass nicht alle Werkzeuge eingesetzt werden müssen, damit *Continuous Delivery* durchgeführt werden kann, jedoch können die Werkzeuge den Prozess vereinfachen.

Continous Integration Server: Jenkins

Der Begriff *Continuous Integration* wurde bereits genau erläutert. Ein *Continuous Delivery Server* ist nichts anderes, als eine Applikation die auf einem Server läuft, welche die nötigen Teilprozesse von *Continuous Integration* ausführt. Wird noch einmal die Abbildung 3.1 Continuous Integration betrachtet, ist der Server, auf dem die Applikation läuft,

direkt neben der Box, in der "Continuous Integration Serverßteht abgebildet. Ein Beispiel "Continuous Integration Server"könnte *Jenkins* sein. *Jenkins* ist ein sogenanntes “Build and Management“ Werkzeug.

> “Jenkins is an Open Source (OSS) CI Platform, whose initial objective has been the automation of the and test process. The build system is completely written in Java and is easily extensible thanks to its plugins architecture and to extension points left into its object model. this makes of jenkins a highly customizable and flexible tool, able to cover many possible scenarios and requirements thatnks to thousands of plugins developed by its huge Open Source Community.“[1]

Ein “Continuous Integration Server“ wie *Jenkins* ist die zentrale applikation, wenn es um *Continuous Integration/Delivery* geht.

> “[..] If CI startet as automation of the development phase only, pretty soon the revolution embraced the test-phase of the QA team and the deployment into various envirnment of the Ops team, involving the whole lifecycle of a product and introducing a new concept: Continuous Delivery (CD)“[1]

Jenkins hat dafür gesorgt, dass *Continuous Delivery* leichter wurde. Mit der Applikation war es nun möglich, in wenigen Schritten eine eigene *Continuous Delivery Pipeline* aufzubauen.

Build-Management-Tool

Build-Management-Tool, nicht zu verwechsel mit einem “Build and Management“ Werkzeug, dient in erster Linie dazu, das bauen einer Software zu standardisieren. Einige bekannte Vertreter für Java sind: Maven, Ant, Gradle. Jeder dieser drei *Build-Management-Tools* arbeitet auf Basis einer zentralen Datei in der die Anweisungen stehen, wie die Software gebaut werden soll. Das beinhaltet unter anderem die nötigen Abhängigkeiten (auch Dependency-Management genannt), sowie die Pfade zu den Dateien, welche zusätzlich eingebunden werden sollen.

Durch *Build-Management-Tools* können zudem zusätzliche Aktionen durchgeführt werden, wie zum Beispiel das ausführen von Unit-Tests, sowie die Steuerung darüber, was bei fehlgeschlagenen Tests passieren soll.

Unter anderem nutzt *Jenkins* solche *Build-Management-Tools* zum bauen der jeweiligen Projekte.

Automatischer Aufbau der Infrastruktur

Nachdem Werkzeuge eingeführt worden sind, mit deren Hilfe standardisierte Software innerhalb und außerhalb von *Continuous Integration*, gebaut werden kann. Muss, damit *Continuous Delivery* durchgeführt und ebenfalls standardisiert werden kann, eine bzw. zwei neue Werkzeuge eingeführt werden.

Damit Software automatisch, im Rahmen von *Continuous Delivery*, ausgeliefert werden kann, muss neben den Tests auch das aufbauen der Infrastruktur automatisiert werden. Zwei Werkzeuge um dies zu erreichen sind *Chef* und *Puppet*.

> "Puppet and Chef are the most commonly used IT Automation Tools used by DevOps to set up the infrastructure, speeding up the process of installing the required software, moddleware and various dependencies."[1]

Damit Software ohne Probleme automatisch deployed werden kann, müssen ggf. Änderungen an der Infrastruktur durchgeführt werden bzw. eine neue Infrastruktur aufgesetzt werden. Letzteres ist zum Beispiel bei der Skalierung notwendig. Damit jede Instanz gleich, bzw. jede Änderung der vorhandenen Instanzen nachvollzogen werden kann, müssen alle Schritte dokumentiert werden. Die technische Dokumentation geschieht mit Hilfe von *Chef/Puppet*. Nach dieser Dokumentation kann das jeweilige Werkzeug gestartet werden und eine standardisierte Infrastruktur wird aufgebaut.

Binary Repository Manager

Zuletzt muss noch eine Möglichkeit gefunden werden, um die einzelnen Versionen von Artefakten zu Versionieren und sicher zu lagern. Hier können sogenannte *Binary Repository Manager* eingestzt werden.

> "It's a single gateway through which you access external artifacts, and store your own build artifacts. By centralizing the management of all binary artifacts, it overcomes the complexity arising from the diversity of binary types, their position in the workflow and the dependencies between them."[8]

In Zusammenarbeit mit *Build-Management-Tools*, kann eine sichere und einfache Möglichkeit des *Dependency Managements* aufgebaut werden.

Nachdem die *Continuous Delivery Pipeline* durchlaufen wurde und alle Tests erfolgreich bestanden wurden, wird das Artefakt im *Binary Repository Manager* gespeichert.

3.6 Einführung von Continuous Delivery

In Kapitel 1.2 Problemstellung wurden bereits zwei Fälle beschrieben, in denen kein Continuous Delivery eingesetzt wurde. Darauf aufbauend wird nun Schritt für Schritt der bestehende Entwicklungsprozess, des fiktive Beispiels von Eberhard Wolff, zu Continuous Delivery überführt. Für die Erstellung der *Continuous Delivery Pipeline* werden die zuvor erläuterten Werkzeuge eingesetzt. Des besseren Verständnis wegen, wird davon ausgegangen, dass in der Programmiersprache Java programmiert wird. Außerdem wird davon ausgegangen, dass eine Versionsverwaltung wie *Git* eingesetzt wird, um Code-Änderungen zu verwalten und nach zu vollziehen.

Bevor eine *Continuous Delivery Pipeline* aufgebaut werden kann, muss zunächst einmal geklärt werden, welche Ziele erreicht werden sollen. Eines der Hauptziele ist es den Deployment-Prozess zu verkürzen. Anstatt nur einmal im Monat zu deployen, soll langfristig gesehen, so oft wie möglich deployed werden. Im Zuge dessen, ist das nächste Hauptziel, die Qualitätssicherung zu verbessern, in dem Tests nicht mehr manuell ausgeführt werden, sondern innerhalb der *Continuous Delivery Pipeline* automatisch ausgeführt werden.
Als ersten Schritt wird ein *Continuous Integration Server* aufgesetzt. Dafür wird das bereits erwähnte Werkzeug *Jenkins* eingesetzt. Jenkins wird zunächst dafür verwende, die in Abbildung 3.1 Continuous Integration gezeigten Prozesse, abzubilden. Mit diesem Schritt wird die “Commit Stage“ abgebildet.

> “The goal of the commit stage is to compile the source code written by the development teams into executable binaries.“[9]

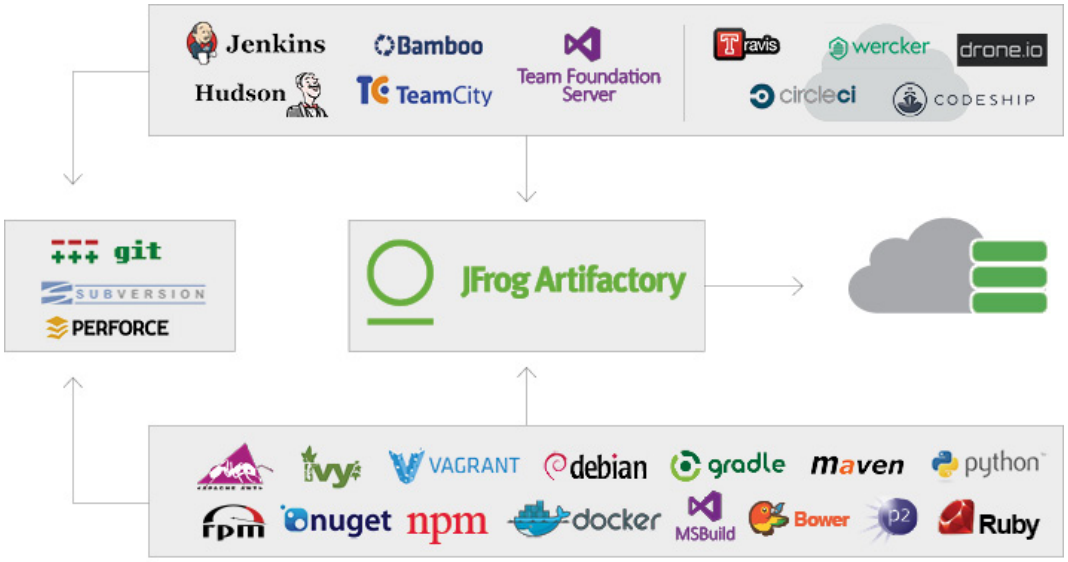
Jenkins wird dabei so eingestellt, dass bei jeder Code-Änderung in der Versionsverwaltung dieser Prozess durchlaufen wird. Damit ist der Anfang der *Continuous Delivery Pipeline* aufgebaut und kann durchlaufen werden.

Damit die aus dem ersten Schritt entstehende Artefakt gespeichert und wiederverwendet werden können wird ein zentrales *Binary Repository Manager* wie Nexus oder Artifactory aufgesetzt. Damit wird, wie schon im Abschnitt 3.5 Werkzeuge für Continuous Delivery

erläutert, sichergestellt, dass alle Versionen eines Artefaktes archiviert werden und durch andere Projekte wiederverwendet werden können. Damit dies geschieht, muss Jenkins noch so eingestellt werden, dass nach einem erfolgreichen durchlaufen der *Pipeline*, das entstandene Artefakt in das zentrale *Binary Repository Manager* archiviert wird.

Als nächsten Schritt wird das Softwareprojekt um ein *Build-Management-Tool* wie Maven erweitert. Mit diesem Schritt soll sichergestellt werden, dass die Software nach einem standardisierten Verfahren gebaut wird, wodurch bei mehrmaligen ausführen der Commit Stage, stets das selbe Resultat erwartet werden kann. Mit dem *Build-Management-Tool* wird außerdem das Problem, der Abhängigkeiten gelöst. Damit ist gemeint, das mit Hilfe des Werkzeugs Abhängigkeiten zu anderen Artefakten automatisch aufgelöst werden können und nicht mehr manuell zum Projekt hinzugefügt werden müssen. Die Abhängigkeiten werden dabei aus dem zentralen *Binary Repository Manager* geladen.

Nach diesem Schritt sieht die aufgebaute Infrastruktur des Entwicklungsprozesses wie folgt aus:



Quelle: https://www.jfrog.com/article/jenkinshudson/

Abbildung 3.3: Continuous Integration Setup. Oben die möglichen „Continuous Integration Server“, links die möglichen Code-Versionsverwaltungen, unten die möglichen „Build-management-Tools“

Die Abbildung zeigt ein Continuous Integration Setup. Oben sind die möglichen “Continuous Integration Server“. Diese greife zum einen auf die Code-Versionsverwaltung zu, um den Code zu laden, mit dem ein Artefakt gebaut werden soll. Zum anderen speichert er das erfolgreich entstandene Artefakt mit Hilfe von Artifactory. Mit einen der unten aufgeführten Build-management-Tools können diese Artefakte in Projekte übernommen werden, ohne sie manuell hinzufügen zu müssen.

Es wurde bisher dafür gesorgt, das Artefakte standardisiert gebaut werden können und das erfolgreiche Resultat, mit Hilfe eines *Binary Repository Managers*, archiviert wird. Jedoch ist der Prozess der Überführung zu *Continuous Delivery* noch nicht abgeschlossen. Ein weiterer wichtiger Schritt ist die Automatisierung der Tests. Zu nächst einmal sollten Unit-Tests geschrieben werden, die den Code Überprüfen. Zusätzlich sollten Akzeptanz Tests geschrieben werden, womit die “Acceptance Test Stage“ abgebildet wird.

> “The goal of the acceptance test stage to verify that the software produced in the commit stage meets the business requirements using automated tests.“[9]

Neben den Akzeptanz Test, sollten noch weitere Tests erstellt werden, wie zum Beispiel

Performance Tests, auch genannt “Capacity Tests“. Mit diesem Schritt wurde ein weiterer Schritt, der *Continuous Delivery Pipeline* hinzugefügt.

> “The goal of the capacity test stage is to make sure that the capacity requirements of the software are met. this stage also includes tests for scalability and endurance“[9]

Mit Hilfe von Jenkins können diese Tests automatisch, innerhalb der *Continuous Delivery Pipeline* durchlaufen werden.

Es können noch weitere Schritte wie zum Beispiel Oberflächen Tests durchgeführt werden. Oft ist es dabei nötig das Artefakt in eine Tests Umgebung zu deployen. Dabei können mit Hilfe von Chef/Puppet ganze Systeme innerhalb der Pipeline aufgebaut werden. Somit kann zum Beispiel eine ganze Produktiv Infrastruktur, nur zum Testen, nachgebaut werden. Hierbei sei jedoch zu beachten, dass das Produktiv System nicht zu groß sein sollte, da der Prozess der Infrastruktur Erstellung zu lange dauern würde.

Es wurden jetzt alle Wichtigen Schritte durchgeführt, die das Bauen und Testen abbilden. Die dadurch entstandene *Continuous Delivery Pipeline* wird, sobald eine Code-Änderung durchgeführt wurde, komplett durchlaufen. Entsteht in einem Schritt ein Problem, wird die Pipeline abgebrochen und der Entwickler erhält eine direkte Rückmeldung über den Fehler. Wird hingegen die Pipeline ohne weitere Probleme durchlaufen, wird das entstandene Artefakt in einem *Repository Manager* archiviert. Abschließend kann unter anderem der Deployment-Prozess ebenfalls automatisiert werden und bei jedem erfolgreichen Durchlauf durchgeführt werden. Der Deployment-Prozess kann jedoch auch soweit automatisiert werden, das mit einem klick das Artefakt ausgeliefert wird, jedoch nicht nach jedem erfolgreichen Durchlauf der Pipeline ausgeführt wird.

Abschließend sei noch erwähnt, dass es nicht nötig ist, von „alle vier Wochen“ zu „wann immer man möchte“ , über Nacht zu springen.
Steve nelly und Steve Stolt von Rally Software schreiben dazu folgendes[11] :

> “Jumping directly from [four] week releases to a pus-to-production strategy is clearly not a sensible approach. We began by shrinking our release cycles down to fortnightly, weekly, semiweekly and finally at-will. These steps took weeks or even monts of preparatory work to get our deploy process streamlined and automated“[11, You do not have to go from eight to zero overnight]

Kapitel 4

Zusammenfassung und Ausblick

4.1 Zusammenfassung

Continuous Delivery ist ein großes Thema und durchaus eine Methode um die Produktivität zu verbessern. Produkte oder Funktionen nach einem bestimmten Zeitplan zu releasen ist nicht mehr möglich. Durch den sehr schnell wandelnden Markt gewinnt bei Unternehmen der Begriff Time-to-Market eine immer größere Bedeutung. Vor allem wenn es sich um Software- bzw. Internet-Unternehmen handelt. Durch stark fortschreitende Technik, können neue Produkte oder Funktionen deutlich schneller released werden, als je zu vor, wodurch bei den Unternehmen der Wettbewerbsdruck immer größer wird. Nur mit einer geringen Time-to-Market Zeitspanne ist es möglich, als Unternehmen dabei zu bleiben, denn auch Nutzer können sich meistens durch die große Menge an Angeboten deutlich schneller für ein anderes Produkt entscheiden, als je zu vor. Ein fester Release-Zeitpunkt für alle neuen Produkte bzw. Funktionen kann nicht nur dafür sorgen, dass Konkurrenzunternehmen ein Produkt schneller auf den Markt bringen als man selbst, es kann auch dazu führen, dass Funktionen im schlimmsten Fall, bei verpassen eines Release-Termins erst mit dem nächsten Release ausgerollt werden. Zudem ist es schwierig Fehler zu beheben, welche im Produktivsystem aufgetaucht sind.

Continuous Delivery kann die Time-to-Market Zeitspanne deutlich verkürzen, wenn das Verfahren richtig angewendet wird, sowie eine frühere Erkennung der Fehler gewährleisten. Dazu ist es jedoch erforderlich einige Vorkehrungen zu treffen. Zum Beispiel müssen Tests möglichst weitgehend automatisiert werden und wiederholbar gemacht werden. Dadurch kann auch bei Änderungen am Code überprüft werden, ob diese Änderungen das Produkt beschädigen oder alle Funktionalitäten wie gewünscht weiterhin funktionieren. Zudem be-

kommt der Entwickler dadurch ein schnelles Feedback über seine Änderungen.

Ziel von Continuous Delivery ist es, alle Schritte während der Entwicklung in eine sogenannte *Continuous Delivery Pipeline* zu bringen, damit sie gemeinsam durchlaufen werden können. Dadurch ist es schließlich möglich von einem Festen Release-Termin zu *"Wann immer man möchte"* Release zu wechseln, da immer ein Releasefähiges Artefakt entsteht; sofern die Pipeline erfolgreich durchlaufen wurde. Wird die Pipeline nicht erfolgreich durchlaufen, enthält der Entwickler ein direktes Feedback über den Fehler, da die Pipeline beim ersten Fehler unterbrochen wird. Dadurch können Fehler schon frühzeitig erkannt und behoben werden.

4.2 Kritische Reflektion

Zu Beginn dieser Arbeit, war zwar klar in welche Richtung sich die Arbeit entwickeln sollte, jedoch wurden die Leitfragen noch nicht gründlich genug herausgearbeitet. Die Systematische Literaturrecherche hingegen bereitete weniger Probleme. Durch die in [10] angegebenen Quellen (siehe 2.2.2 Quellen) war es schnell möglich die einzelnen Quellen zu bewerten und einige relevante für die Recherche zu nehmen. Auch ergaben die Begriffe nach denen gesucht wurden und den Suchstrings keinerlei Probleme. Es stellte sich jedoch nach erster suche in den ausgewählten Quellen heraus, das nicht alle dieser Quellen geeignet waren, um dieses Thema zu bearbeiten. Der Begriff Continuous Delivery wird auch oft mit Themen aus der Biologie verbunden und sowohl der Suchzusatz *„IT"* oder *„Software"* ergaben keinerlei Treffer für das Gebiet der Informatik, sondern für spezielle Software, die das Biologische Continuous Delivery betrachtet. Daher mussten die Suchmaschinen auf *IEEEXplore* und *Google Scholar* beschränkt werden. Jedoch verwies auch *Google Scholar* zum Teil auf Dokumente, die der Biologie angehören.

Wie bereits erwähnt waren die Leitfrage nicht von Anfang an klar definiert, sodass der Hauptteil (Kapitel 3) zunächst einmal nur eine grobe Fassung über Continuous Delivery war. Erst mit der Zeit entwickelte sich ein klares Bild, in welche Richtung sich die Arbeit wirklich entwickeln sollte. Ein Aspekt der hier mit spielt, ist die Tatsache das die Begriffe Continuous Integration/Delivery/Deployment nicht eindeutig definiert sind und sich je nach Definition nicht von einander unterscheiden. Lediglich der Begriff Continuous Integration unterscheidet sich im geringen von Continuous Delivery/Deployment und hat dadurch die Möglichkeit geboten, die zugrunde legende Strategie genauer zu erläutern und abzugren-

zen. Aus diesen Gründen wurde der Begriff explizit genannt und erläutert. Der Begriff bzw. der Vergleich mit Continuous Deployment hingegen wurde weggelassen, da, wie bereits geschrieben, die Begriffe sich nicht eindeutig von einander unterscheiden.

Da das Thema Continuous Delivery noch nicht so verbreitet ist und auch wenig Material, im Vergleich zu anderen Themen, bietet, wurden leider im Rahmen der durchgeführten Systematischen Literaturrecherche keine nennenswerte Studien über dieses Thema gefunden. Eine normale Google Suche ergab leider auch keine nennenswerten Studien zu diesem Thema.

4.3 Ausblick

Diese Arbeit bietet einen kleinen Einblick in das Thema Continuous Delivery. Darauf aufbauend können zum Beispiel die Kosten für die Einführung von Continuous Delivery, sowie die Kostenunterschiede zwischen Entwicklungsprozesse mit und ohne Continuous Delivery ermittelt werden. Zudem kann explizit auf die Umstellung eines bestehenden Entwicklungsprozesses zu Continuous Delivery eingegangen werden. Dazu könnte zum Beispiel ein Prozess analysiert und bewertet werden und Schritt für Schritt in detaillierter Form festgehalten werden. Eine weitere Möglichkeit wäre es die Werkzeuge für Continuous Delivery genauer zu erläutern. Insbesondere das *Continuous Integration Werkzeug Jenkins* könnte weiter erläutert werden. Es könnte jedoch auch ein Plugin für dieses Werkzeug geschrieben werden, wodurch die Abbildung einer Continuous Delivery Pipeline vereinfacht wird.

Literaturverzeichnis

[1] ARMENISE valentina: *Continuous Delivery with Jenkins*. IEEE/ACM. 2015

[2] BUSINESSDICTIONARY.COM: *time to market*. – URL http://www.businessdictionary.com/definition/time-to-market.html

[3] CHEN, Lianping: *Continuous Delivery Huge Benefits, but Challenges Too*. IEEE. 2015

[4] FOWLER, Martin: *Continuous Integration*. 01 May 2006. – URL http://martinfowler.com/articles/continuousIntegration.html. – visited: 08.06.2016

[5] FOWLER, Martin: *ContinuousDelivery*. 30 May 2013. – visited: 08.06.2016

[6] FOWLER, Martin: *DeploymentPipeline*. 30 May 2013. – visited: 08.06.2016

[7] GRUENDERSZENE.DE: *Software-as-a-Service (SaaS)*. – URL http://www.gruenderszene.de/lexikon/begriffe/software-as-a-service-saas. – visited: 27.06.2016

[8] JFROG.COM: *What is a Binary Repository Manager*. – URL https://www.jfrog.com/binary-repository/. – visited: 27.06.2016

[9] JOHANNES GMEINER, Julian H.: *Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company*. IEEE. 2015

[10] KITCHENHAM: *Guidelines for performing Systematic Literature Reviews in Software Engineering*. 2007

[11] NEELY, Steve ; SOFTWARE), Steve Stolt (.: *Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy*. IEEE. 2013

[12] WIESMANN, Prof. Dr. D.: *Skript der Veranstalltung SWT D*. FH-Dortmund. 2014

[13] WOLFF, Eberhard: *Continuous Delivery - Der pragmatishe Einstieg*. 1. Auflage. dpunkt.verlag, 2015. – ISBN 978-3-86490-208-6

Anhang A

Anhang

A.1 Rechercheprotokoll

Kriterien der Kategorie **P** sind Inhaltliche Auswahlkriterien.

Kriterien der Kategorie **N** sind Inhaltliche Ausschlusskriterien. Doppelte Einträge wurden Grau hinterlegt.

Tabelle A.1: IEEEXplore

IEEExplore			
Suchstring	**Titel**	**Author**	**Kriterium**
$S_{deployment}$	System dynamics Modeling of Agile Continuous Delivery Process	Olumide Akerele, Muthu Ramachandran, Mark Dixon	P1/P2
$S_{pipeline}$	End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery	Miteshi Soni	P2/P3
$S_{pipeline}$	Test orchestration a framework for Continuous Integration and Continuous Delivery	Nikhil Rathod, Anil Surve	P3
$S_{pipeline}$	Towards Architecture for Continuous Delivery	Lianping Chen	P1
$S_{pipeline}$	The Highways and Country Roads to Continuous Deployment	Marko Leppänen, Simon Mäkinien, Max Pagels, Veli-Pekka Eloranta, Juhu Itkonen, Mika V. Mäntylä, Tomi Männistö	P1

Continued on next page

Tabelle A.1 – continued from previous page

Suchstring	Titel	Author	
$S_{pipeline}$	Continuous Delivery: Huge Benefits, but Challenges Too	Lianping Chen	P1
$S_{pipeline}$	Automated testing in the continuous delivery pipeline: A case study of an online company	Johannes Gmeiner, Rudolf Ramler, Julian Haslinger	P1
S_{devops}	Delivering Software with agility and quality in a cloud environment	F. Oliveira, T. Elam, P. Nagapurkar, C. Isci	N3
S_{devops}	DevOps and Its Practices	Liming Zhu, Len Bass, George Champlin-Scharff	N1/N5
S_{devops}	Research Opportunities in Continuous Delivery: Reflections from Two Years' Experiences in a Large Bookmaking Company	Lianping Chen	N2
S_{devops}	DevOps: Making It Easy to Do the Right Thing	Matt Callanan, Alexandra spillane	P4
$S_{einfach}$	Continuous Delivery with Jenkins: Jenkins Solutions to Implement	Valentina Armenise (CloudBees)	P3
$S_{einfach}$	Understanding DevOps & bridging the gap from continuous integration to continuous delivery	Manish Virmani	P1
$S_{einfach}$	Microfabricated silicon nanopore membranes provide continuous delivery of biopharmaceuticals	P. Gardner	N2
$S_{einfach}$	Introducing Continuous Delivery of Mobile Apps in a Corporate Environment: A Case Study	Sebastian Klepper, Stephan Krusche, Sebastian Peters, Bernd Bruegge, Lukas Alperowitz	P1

Continued on next page

Tabelle A.1 – continued from previous page

Suchstring	Titel	Author	
$S_{einfach}$	Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That easy)	Steve Neely, Steve Stolt	P1
$S_{einfach}$	An Experimental Continuous Delivery Framework for SmartX-mini IoT-Cloud playground	Jeongju Bae, JongWon Kim	P4
$S_{einfach}$	Toward Design Decisions to Enable Deployability: Empirical Study of Three projects Reaching for the Continuous Delivery Holy Grail	Stephany Bellomo, Neil Ernst, Robert Nord, Rick Kazman	P1
$S_{einfach}$	Social testing: A frameowkr to support adoption of continuous delivery by small medium enterprises	Jonathan dunne, David Malone, Jason Flood	N1
$S_{einfach}$	Full exploitation of process variation space for continuous delivery of optimal delay test quality	Baris Arslan, Alex Orailoglu	N1
$S_{einfach}$	Continuous Delivery Message dissemination Problems under the Multicasting Communication Mode	Teofilo F. Gonzalez	N1

Tabelle A.2: Google Scholar

Google Scholar			
Suchstring	**Titel**	**Author**	**Kriterium**
$S_{deployment}$	Continuous Integration	Martin Fowler	P1
$S_{pipeline}$	Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery	Valentina Armenise	P3
$S_{pipeline}$	(IEEExplore) Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)	Steve Neely	P2
S_{devops}	Continuous Delivery	Martin Fowler	P1 / N3
S_{devops}	Continuous Delivery and DevOps - A Quickstart Guide	Paul Swartout	N3
S_{devops}	Breaking down barriers and reducing cycle times with devops and continuous delivery	Paul Duvall	P1
S_{devops}	Synthesizing Continuous Deployment Practices Used in Software Development (gefunden in IEEEXplore)	Akond Ashfaque ur Rahman, Eric Helms, Laurie Wiliams, Chris Parnin	N1/N4
S_{devops}	Continuous Integration and Automation for Devops	Adnreas Schaefer, Marc Reichenbach, Dietmar Fey	N2/N3