fachhochschule stralsund
university of applied sciences

adesso business. people. technology.

Master-Thesis

Evaluierung von ausgewählten Werkzeugen zur Realisierung von Continuous Delivery im Vergleich zur aktuellen Auslieferungspraxis von Software der adesso AG

Verfasser:	André Gärtner
	Fährhofstraße 28a
	18439 Stralsund
Erstkorrektor:	Prof. Dr. rer. nat. Gerold Blakowski
Zweitkorrektor:	Dipl. Ing.(FH) Dieter Gaikowski

Datum	12. Oktober 2012

Kontakt E-Mail:	swangaer@gmail.com

Inhaltsverzeichnis

II. Anhänge

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

BGB	Bürgerliches Gesetzbuch	IaaS	Infrastructure as a Service

Teil I.

Master-Thesis

1. Einleitung und Zielstellung

Diese Arbeit setzt sich mit dem Themenkomplex von Continuous-Delivery auseinander. Wolf Schlegel formulierte die Zielstellung von Continuous-Delivery in einem Artikel folgendermaßen: "*Software ist erst dann fertig, wenn sie produktiv eingesetzt wird.*"[1] Damit wird der Schritt, die Anwendung in die Produktivumgebung zu heben, in einen besonderen Fokus gesetzt.

In den späten 90er Jahren arbeitete Kent Beck für ein Versicherungsunternehmen in der Schweiz, bei der jede Nacht in das Produktivsystem ausgeliefert wurde. Dieses Verfahren war für den Versicherer von Vorteil, da fertige und produktiv einsetzbare Software nicht erst auf das nächste Release in drei Monaten warten musste, wie in anderen Projekten üblich, sondern schon am nächsten Tag gewinnbringend eingesetzt werden konnte.[2]

Das erste der 2001 verfassten 12 Prinzipien aus dem Manifest für Agile Softwareentwicklung, nachfolgend kurz *Agiles Manifest* genannt, priorisiert eine frühe und kontinuierliche Auslieferung von wertvoller Software.[3] Mit Continuous-Delivery wird ein Prinzip des *Agilen Manifests* umgesetzt. Daraus kann abgeleitet werden, dass aus der Sicht des Kunden, durch jeden Entwicklungsschritt der Wert der Software erhöht wird und der daraus resultierende Nutzen frühestmöglich auch produktiv eingesetzt werden sollte. Ein Ziel, das Kent Beck in dem zuvor genannten Versicherungsprojekt bereits verwirklichen konnte.

In dem agilen Vorgehensmodell Scrum wird der Prozess der Softwareentwicklung in Iterationen mit einer Dauer von 2 bis 4 Wochen unterteilt. Jede Iteration soll eine Version der Software liefern, die mehr für den Kunden nützliche Funktionalitäten bereitstellt, als die Version der vorhergehenden Iteration.[4]

Anforderungen erfüllt, ist eine Inbetriebnahme dieser Version vom Kunden durchaus wünschenswert. Bei einem Kunden-Lieferanten-Verhältnis kann dem Kunden dann die Möglichkeit eingeräumt werden, die aktuelle Version zu testen, abzunehmen und die so bereitgestellten Funktionalitäten schon produktiv einzusetzen. Die Entwicklung ist zu

[1] Vgl. [Sch12]
[2] Vgl. Vorwort von Martin Fowler in [HF11, S. xxi]
[3] Vgl. [B+01]
[4] Vgl. [Wir09]

diesem Zeitpunkt noch nicht abgeschlossen und weitere geplante Funktionalitäten können folgen. Nachfolgende Entwicklungsschritte erfordern dann aber wiederum weitere Abnahmetests und Inbetriebnahmen der neuen Version. Bis zum Ende der Projektlaufzeit könnte diese mehrere Wiederholungen des Abnahmeprozesses nach sich ziehen. Bei der Forderung eine Software kontinuierlich in kleinen Schritten auszuliefern, wird die dahinterstehende Komplexität nicht unmittelbar ersichtlich, die sich aus der Bereitstellung einer Software in der spezifizierten Produktivumgebung verbirgt.

Nach Steinweg umfasst die Einführung einer Software die Phasen der Werkabnahme, des Pilotbetriebs, der offiziellen Abnahme, dem Rollout, dem Training der Anwender und dem Going-Live.[5] Bei der Werkabnahme ist der Besteller nach § 640 Absatz 1 Satz 1 BGB verpflichtet, das vertragsmäßige Werk abzunehmen. Dieser Schritt ist notwendig, damit Mängel beim Hersteller des Werkes fristgerecht angezeigt werden können und der Kunde seine in § 634 Nr. 1 bis 4 BGB aufgeführten Rechte auf die Nacherfüllung, den Ersatz für erforderliche Aufwendungen, den Vertragsrücktritt, die Minderung oder den Schadensersatz wahrnehmen kann. Damit sind aber auf Kundenseite auch nicht zu unterschätzende Aufwände für die Abnahme der Software in den geregelten Betrieb einzuplanen.

Continuous-Delivery verspricht durch die Automatisierung und eine klare Prozessgestaltung die Komplexität des Liefervorgangs zu reduzieren. Es soll die Basis eines schnellen und zuverlässigen Software-Lieferprozess bereitstellen.[6] Soll für ein Projekt ein teil- bzw. vollautomatisierter Prozess umgesetzt werden, müssen zuvor nicht nur eine Reihe technischer Probleme, sondern auch vielmehr vertragliche und organisatorische Implikationen auf einen automatisierten Lieferprozess betrachtet werden. Continuous-Delivery wird aber erst durch Technologie ermöglicht.

Continuous-Delivery steht in einem engen Zusammenhang mit der DevOps-Bewegung. DevOps fordert die engere Zusammenarbeit von Softwareentwicklern und IT-Betrieb als die Weiterentwicklung der agilen Vorgehensmodelle. Durch die konsequente Verwirklichung, neue Funktionalitäten dem Kunden schnellstmöglich bereitstellen zu können, sind neue Strukturen in der Organisation von Entwicklungsteams notwendig. Continuous-Delivery ist in diesem Zusammenhang die technologische Verwirklichung von DevOps, da es hilft, Kommunikationsbarrieren zwischen den Teilbereichen Betrieb und Entwicklung abzubauen. DevOps selbst fokussiert keinen spezifischen Technologie-Stack. Die Ideen zu DevOps und Continuous-Delivery stammen aber von Personen, die besonders aktiv im Umfeld von Web 2.0-Anwendungen und Cloud-Computing sind.

Die Anforderung, eine neue Funktionalität den Anwendern möglichst vor der Konkur-

[5] Vgl. [Ste04, S. 172 ff]
[6] Vgl. [Sch12]

renz bereitstellen zu können, ist in diesem Umfeld besonders hoch. Der Prozess, eine neue Funktionalität in die Produktionsumgebung auszuliefern, muss deshalb schnell und ohne zusätzlichen Aufwand ablaufen können. Beim Ausliefern einer neuen Softwareversion darf aber nicht das Risiko des Scheiterns des Prozesses missachtet werden. Ein Webshop mit vielen tausend aktiven Nutzern, der nach einem Update plötzlich nicht mehr verfügbar ist oder schwere Fehler z. B. bei der Kalkulation des Preises macht, kann einen hohen wirtschaftlichen Verlust verursachen. Automatisierte Tests und das stückweise Aktualisieren der Anwendung sowie ein sofort verfügbares Roll-Back Konzept sind daher auch Komponenten von Continuous-Delivery.

In seinem Blog-Eintrag aus dem Jahr 2009 ging Timothy Fitz auf das Thema Continuous-Delivery ein. In diesem schrieb er, dass längere Releasezyklen in Projekten problematisch sein. Er sah, dass auch kleinere Fehler, die erst in der Produktionsumgebung durch hervortreten, z. B. durch eine andere Konfiguration oder andere Systembedingungen, sehr schwierig sind, zu beheben. Er beschrieb, wie durch ein einfaches Refactoring einer Anwendung kleinere Fehler in die Quellcode-Basis gebracht wurden. Der Fehler trat auf, als die Anwendung zwei Wochen später in Produktion ausgeliefert wurde. Da es ein Konfigurationsfehler war, der das Produktivsystem betraf, konnte der Fehler durch die Qualitätssicherung nicht entdeckt werden. Erst nach mehrstündiger intensiver Inspektion konnte der Fehler entdeckt werden und Anwendung wieder produktiv gehen. Die Erinnerung an die durchgeführten Änderungen waren nur noch ungenau vorhanden, da die Änderung schon zwei Wochen her war. Fitz empfahl nach diesem Vorfall täglich in die Produktionsumgebung auszuliefern. Er ging davon aus, dass dieser Fehler auch nicht durch die Steigerung von automatisierten Tests zu vermeiden gewesen wäre. Der Prozess selbst war das Problem. Wäre die Anwendung direkt nach dem Refactoring und dem Durchlauf der Tests in die Produktivumgebung ausgeliefert worden, hätte die Auswirkung schneller mit der durchgeführten Änderung in Verbindung gebracht werden können.[7]

In den nachfolgenden Kommentaren zu seinem Vorschlag fanden sich viele Aussagen wie "*unrealistisch*", "*funktioniert nicht*", "*geht nur mit kleineren Anwendungen*", "*nette Theorie, aber kann nicht klappen*". Ein weiterer Verfasser eines Kommentars hatte auf die Anforderungen eines komplexen Finanztransaktionssystem hingewiesen, in dem alle Transaktionen einer fehlerhaften Version möglicherweise wieder rückgängig gemacht werden müssten. Für den Verfasser kann ein direktes Ausliefern der Anwendung in die Produktivumgebung nur für kleinere Anwendungen mit geringem Funktionsumfang möglich sein. Die Nutzung dieses Konzeptes für Testumgebungen hält er jedoch für lohnenswert.[8]

[7] Vgl. [Fit09]
[8] Vgl. [Fit09]

Auf Ebene der Betriebssysteme ist es möglich wiederkehrende Vorgänge, wie Programmaufrufe, Softwareinstallationen oder Dateibewegungen, in Shell-Skripten zu verfassen und vom System jederzeit wieder ausführen zu lassen. So könnte das Kompilieren des Quellcodes oder die Bereitstellung einer Software auf einem Testsystem in einem derartigen Kleinstprogramm beschrieben werden. Über die Systemkonsole ließe sich dieses jederzeit auf die gleiche Art und Weise ausführen. Eine Teilautomatisierung des existierenden Auslieferungsprozess würde so schon realisiert werden können. Dieser würde allerdings nur aus einer losen Sammlung von Skripten bestehen, die auf eine konsistente Aufrufreihenfolge angewiesen sind. Wenn die Prozesse der Softwareerstellung, des Tests und der Lieferung auf entfernten Systemen durchgeführt werden, ist die Durchführung möglicherweise nur auf einen bestimmten Personenkreis beschränkt, welcher Zugriffsrechte auf die entfernten Systeme besitzt.

Der Problematik, den Lieferprozess von jedem Mitglied des Entwicklungsteam durchführen lassen zu können ohne aber die Systemrechte an alle zu verteilen, standen auch eine Reihe von Projekten im Bereich der Webanwendungen gegenüber, die in eigene Lösungen mündeten. Einige dieser Lösungen, wie *Deployinator* und *Dreadnot* wurden unter eine quelloffene Lizenz gestellt und einer breiten Öffentlichkeit zur Nutzung und Weiterentwicklung zugänglich gemacht.[9] Daneben haben auch Hersteller proprietärer Software den Bedarf an derartigen Lösungen entdeckt und bieten Systeme, wie *Go* von Thoughtworks, unter dem Schlagwort Continuous-Delivery an.[10]

Die adesso AG, nachfolgend adesso genannt, ist ein unabhängiger IT-Dienstleister, der mit Beratung und Softwareentwicklung die Kerngeschäftsprozesse seiner Kunden optimiert.[11] adesso besitzt dabei, neben anderen Technologien, eine Kompetenz im Java-Technologiespektrum. Dies umfasst auch besonders das Komponentenmodell der Java Enterprise Edition, kruz JEE. Auf dieser Basis verwirklicht adesso regelmäßig Projekte.[12]

Für die Realisierung von neuen Softwareprojekten hat adesso ein eigenes agiles Vorgehensmodell, *adVANTAGE*, entwickelt. *adVANTAGE* soll die Entwicklung auf Basis eines grob kalkulierten Budgets ermöglichen. Anforderungen werden hierbei, ähnlich dem Vorgehen in Scrum, priorisiert und für Sprints ausgewählt und entwickelt. Bei der Auswahl stehen die für das Geschäft kritischen Anforderungen im Vordergrund, die für einen frühestmöglichen Start der Anwendung notwendig sind.[13]

Hieraus ergibt sich für adesso ein Interesse am automatisierten Auslieferungsprozess und Continuous-Delivery. Im Unternehmen werden deshalb verschiedene Aktivitäten

[9] Vgl. [Kas10]
[10] Vgl. [Tho12c]
[11] Vgl. [ade12a]
[12] Vgl. [ade12e]
[13] Vgl. [ade12b]

rund um das Thema vorangetrieben. Dabei hat adesso drei Werkzeuge identifiziert, die einen Fortschritt für den derzeitigen Auslieferungsprozess bedeuten könnten. Folgende Werkzeuge sind zu untersuchen:

- *Go* von Thoughtworks Studios[14],
- *Deployinator* von Etsy[15] und
- *Dreadnot* von Rackspace[16].

Der genaue Funktionsumfang der Werkzeuge ist nicht bekannt. Bekannt ist nur, dass diese in Verbindung mit Continuous-Delivery stehen. adesso möchte aus diesem Grund einen Überblick über die Funktionalitäten und Ansätze dieser Werkzeuge gewinnen. Ziel dieser Arbeit soll deshalb die Evaluierung dieser sein. Dabei soll geprüft werden, ob eines der untersuchten Werkzeuge geeignet ist, um den derzeitigen Auslieferungsprozess sinnvoll zu ergänzen. Aus diesem Grund müssen nicht nur die angeführten Werkzeuge untersucht, sondern auch der bestehende Auslieferungsprozess betrachtet werden. Einführend beschäftigt sich diese Arbeit mit der Kernfrage von Continuous-Delivery und beschreibt die entsprechenden Konzepte, die zum Verständnis der Werkzeuge notwendig sind.

[14] Informationen zu *Go* unter `http://www.thoughtworks-studios.com/go-agile-release-management`
[15] Informationen zu *Deployinator* unter `https://github.com/etsy/deployinator`
[16] Informationen zu *Dreadnot* unter `https://github.com/racker/dreadnot/`

2. Entstehung und Konzepte von Continuous-Delivery

Um die Funktionalitäten der zu untersuchenden Werkzeuge verstehen zu können, ist eine Betrachtung der Konzepte Continuous-Delivery notwendig. Dieser Abschnitt behandelt eingehend die Problemstellung von Continuous-Delivery, denen mit den hier vorgestellten Konzepten begegnet werden soll. Hinter den Ideen von Continuous-Delivery steht die DevOps-Bewegung. Aus DevOps leiten sich technische und organisatorische Implikationen wie z. B. die *Definition of Done*, der Release-Candidate, die Prozessautomatisierung und das Fast-Feedback an das Entwicklungsteam ab. Wesentliche Beförderer dieses Themenkomplexes sind Jez Humble und David Farley, die in ihrem Buch Continuous-Delivery und die Deployment-Pipeline zusammenhängend darstellen und Lösungsstrategien für technische Fragestellungen anbieten.[1] Continuous-Delivery berührt durch das automatisierte Auslieferungsverfahren auch die Domaine der ITIL Service Transitions gerade im Bereich von Service- und Release-Management. Am Ende des Abschnitts wird mit einem Exkurs auf die gegenseitigen Implikationen von Continuous-Delivery und ITIL eingegangen.

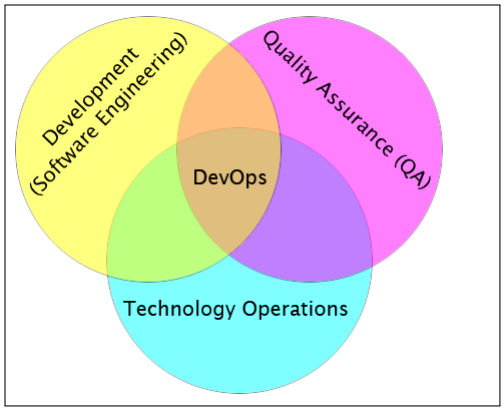
2.1. Die DevOps-Bewegung

In der belgischen Stadt Genth fand unter dem Titel DevOps-Days im Jahr 2009 eine Konferenz zu den Themen Build[2], Softwaretest und Deployment[3] statt.[4] Die Seite `www.DevOps.com` begrüßt seine Besucher mit dem Slogan "*Helping finish what Agile development started*"[5]. Dieser Slogan beschreibt das eigene Verständnis der DevOps-Bewegung. Chris Read schreibt in einem Gastbeitrag über DevOps wie folgt: "*At its heart it is the integration of Agile principles into Operations practices*". Dies ermöglichte

[1] Vgl. [HF11]
[2] Build bezeichnet den das Kompilieren und Paketieren von Quellcode.
[3] Deployment bezeichnet das Ausliefern von Softwarepaketen in eine Ausführungsumgebung. Dies kann z. B. eine Webanwendung sein, die von einem Webserver ausgeführt wird
[4] Vgl. [NS10]
[5] [dev12]

nach seiner Ansicht die Clound- und Web-2.0-Giganten.[6] DevOps versucht im Kern die Kommunikation zwischen Systemadministratoren und den Entwicklern der Systeme zu verbessern. Die agilen Praktiken, wie sie zwischen Entwicklern und ihren Kunden schon flächendeckend betrieben werden, sollen auch zwischen IT-Betrieb und Softwareentwicklern implementiert werden. Auch Humble und Farley sind in ihrem Buch der Ansicht, dass agile Techniken zum Managen von Infrastrukturen sehr gut geeignet sind.[7]

Nach Ansicht von Stephen Nelson-Smith bereitet vornehmlich die Angst der Unternehmensführung und des IT-Managements, Änderungen an einer laufenden Anwendung vorzunehmen, Probleme bei der Entwicklung von Software mit kurzen Releasezyklen. Zudem werden häufig bürokratische Prozesse wie das Change-Management von ITIL, welche für die Compliance eines Unternehmens wichtig sind, von der Unternehmensführung als Argument genutzt, eine Änderung von laufenden Systemen an hohe Anforderungen zu knüpfen. Nach Einschätzung von Nelson-Smith kommt es hierdurch zu einem merklichen Zeitverlust, wenn ein neues Feature oder ein Bug-Fix eingespielt werden soll.[8]



Quelle: [Pan10] .

Abbildung 2.1.: DevOps als Schnittmenge von Entwicklung, IT-Betrieb und Qualitätssicherung.

Bei der Auslieferung von Software in die Produktivumgebung treten erhebliche Risiken auf, die den fehlerfreien Betrieb der Anwendung gefährden können. Dabei gibt es keine vollständige Sicherheit, ob die Software in der geplanten Umgebung wie erwartet läuft oder z. B. für Webanwendungen die Lastanforderungen durch steigende Nutzerzahlen erfüllt werden können. Nelson-Smith zielt dabei auf Projektszenarien ab, bei der die Lauffähigkeit einer Anwendung, durch Build-Prozess und Testdurchlauf, nur auf den Arbeitsgeräten der Entwickler bewiesen wird. Ein Test, in einer dem Produktivsystem angelehnten Umgebung, wird aus Zeit- und Kostengründen nicht durchgeführt.[9]

Bei Webanwendungen kommen häufig mehrere Komponenten wie Web- und Applikationscontainer, Frameworks, Datenbankserver oder ein ganzer Server-Cluster zum Einsatz. Ein Testsystem, welches der späteren Produktionsumgebung entspricht, verursacht zusätzliche Kosten, da weitere Infrastruktur und Wartungsaufwände für die Test-Systeme hinzukommen. Hat ein Entwickler Bedarf eine neue Komponente zu testen, können

[6] Vgl. [Rea10]
[7] Vgl. [HF11, S. 279]
[8] Vgl. [NS10]
[9] Vgl. [NS10]

je nach Organisation des IT-Betriebs mehrere Tage vergehen, bis die notwendige Infrastruktur für einen Kapazitätstest zur Verfügung steht. In den kurzen Entwicklungszyklen agiler Methoden bedeutet dies einen empfindlichen Zeitverlust, bis die Reife der neuen Komponente bewiesen und im Produktionssystem laufen kann.[10]

Smith-Nelson führt weiter an, dass zwischen IT-Betrieb und Entwicklern häufig ein "*Bunkerdenken*" existiert, in der jeder Beteiligter in seiner fachlichen Domaine eines Entwicklers, Testers, Release-Managers oder Systemadministrators denkt, als produktiv an der Verwirklichung neuer gewinnbringender Funktionalitäten zu wirken. Bei Fehlern werden häufig Aufgaben oder Schuldzuweisungen zwischen den geschaffenen Domainen hin und her geschoben.[11]

Humble sieht in der generellen Aufteilung von Entwicklung und IT-Betrieb als auch durch das Governance-Framework ITIL und Cobit den Wunsch nach stabil laufenden Systemen. Seiner Ansicht nach soll hierdurch verhindert werden, dass die Entwickler einen zu großen Schaden durch neue und noch nicht vollständig ausgereifte Funktionalitäten am Produktivsystem verursachen könnten. Die Entwicklung und Einführung neuer Funktionalitäten wird so verlangsamt.[12]

Damon Edwards schreibt, dass es sich bei DevOps nicht um ein technologisches sondern viel mehr um ein betriebswirtschaftliches Problem handelt. Technologie nimmt aber die Schlüsselrolle ein, wenn es darum geht, dieses betriebswirtschaftliche Problem auch zu lösen. Bei der Entwicklung einer neuen Software steht immer das betriebswirtschaftliche Problem im Vordergrund. Dies könnte z. B. die Erschließung eines neues Marktes über das Internet sein. Der Nutzen, der aus der Entwicklung eines neuen Online-Shops gezogen werden soll, ist die Erwirtschaftung von Gewinn. Um dieses Ziel zu ermöglichen, müssen alle Beteiligten im Prozess wie Entwickler, Qualitätssicherung und IT-Betrieb, eng zusammenarbeiten. Für Edwards steht folgende Frage bei DevOps im Vordergrund: "*How to enable a business to react to market forces as quickly as possible?*"[13]

Nach Nelson-Smith müssen Entwickler, Tester, Manager, Administratoren von Datenbanken, Netzwerktechniker und Systemadministratoren das selbe Ziel verfolgen, gute Software liefern. Dabei fordert er Entwicklungsteams die die Kompetenz eines "*sysadmin coders*" besitzen. Dies ist ein Team, welches die Fähigkeiten, Kompetenzen und Rechte des IT-Betriebs und der Softwareentwicklung auf sich vereint.[14]

Für Read besteht eine Definitionslücke zwischen "*dev complete*", also der Fertigstellung der Entwicklung einer Anwendung oder einzelnen Features und "*live, in production, sta-*

[10] Vgl. [NS10]
[11] Vgl. [NS10]
[12] Vgl. [Hum11]
[13] Vgl. [Edw10]
[14] Vgl. [NS10]

ble, making money". Typischerweise sind Systemadministratoren in der Verantwortung und der Pflicht eine neue oder aktualisierte Anwendung, deren genaue Funktionsweise sie nicht kennen, in die Produktivumgebung auszurollen und deren Betrieb sicherzustellen. Dementsprechend ist auf dieser Seite mit Vorbehalten und Vorsicht zu rechnen, wenn es darum geht, neue Software einzuführen. Wie Nelson-Smith ist auch Read der Ansicht, dass Entwicklungsteams alle Disziplinen besetzen sollten, die notwendig sind, um eine Anwendung nicht nur entwickeln sondern auch betreiben zu können.[15]

Chad Dickerson, CEO bei Etsy, berichtete, dass bei Etsy Designer, Produktmanager, Entwickler und Administratoren sehr eng an der Verwirklichung neuer Funktionalitäten zusammenarbeiten. Etsy erreicht durch DevOps und der Automatisierung bis zu 517 Deployments in das Produktivsystem im Monat. Diese werden von bis 63 unterschiedlichen Mitarbeitern angestoßen. Möglich wurde diese auch durch den Einsatz von *Deployinator*.[16]

Für Read helfen folgende Ansätze, DevOps zu ermöglichen:[17]

- Die Auflösung verwurzelten Funktionsstrukturen im Unternehmen die verhindern, dass Entwicklungsteams sich aus Programmierern, Testern, Business-Analysten, Administratoren zusammensetzen können. Falls dies nicht möglich sein sollten, ist eine enge Verbindung zwischen IT-Betrieb und Entwicklern sowie allen sonstigen Beteiligten herzustellen und agile Praktiken im IT-Betrieb einzuführen.

- Investition in die Automatisierung von Administrationsvorgängen, wie das Ausrollen einer neuen Softwareversion. Hier kann eine *Tool-Chain*, die Aneinanderreihung von geeigneten Werkzeugen, einen Mehrgewinn bedeuten.

- Das operative Risiko durch Softwareaktualisierungen kann gesenkt werden, wenn nur kleine dafür aber kontinuierliche Änderungen am System vorgenommen werden.

- Eine Cloud-Strategie im Unternehmen ermöglicht es dem Entwicklungsteam sich schnell und einfach mit Test- und Produktivsystemen selbst zu versorgen.

- Allen an der Entwicklung eines Dienstes oder einer Anwendung Beteiligten sollte die gemeinsame Verantwortung für eine qualitative Lieferkette bewusst sein.

Die Einführung einer DevOps-Kultur ist für Read die Grundlage, um Continuous-Delivery zu betreiben. Continuous-Delivery wiederum löst die Probleme von DevOps. Die

[15] Vgl. [Rea10]
[16] Vgl. [Dic11]
[17] Vgl. [Rea10]

Umsetzung von Continuous-Delivery für ein spezifisches Projekt wird dementsprechend auch die Probleme von DevOps berühren.[18]

2.2. Problemstellung von Continuous-Delivery

Für Continuous-Delivery können vier verschieden Problemfelder ausgemacht werden. Die im nachfolgenden Abschnitt vorgestellten Konzepte versuchen diese zu lösen.

Ausnutzung des Wertzuwachses durch iterative Softwareentwicklung: Bei Web-2.0-Anwendungen ist der Erfolg auch davon abhängig, neue Funktionalitäten möglichst schnell und vor der Konkurrenz bereitstellen zu können. Diese Fähigkeit kann zu einem hohen Maße den wirtschaftlichen Erfolg positiv beeinflussen.

Der iterative Ansatz der agilen Softwareentwicklung produziert, wenn entsprechend organisiert, die Funktionalitäten mit dem höchsten Kundennutzen an erster Stelle. Nützliche und funktionierende Software sollte nach Humble und Farley den Anwendern auch schnellstmöglich zur Verfügung gestellt werden.[19]

Einen Gegenentwurf zu diesem Vorgehen, dem sich Dienstleister in einem typischen Kunden-Lieferanten-Verhältnis gegenübersehen, stellt der reguläre Einführungsprozess von Software dar. Hier fließen Werkabnahme, Pilotbetrieb, Abnahme und Erfüllung aller Anforderungen, Roll-Out und Going-Live sequenziell ineinander.[20] Ein neues Feature schnell und einfach in die Produktion zu bringen, auch wenn nach agilen Methoden vorgegangen wird, ist mit einem derartigen Prozess schwer möglich. Der Abnahmeprozess für eine neue Software ist aufwendig aber notwendig, da der Kunde verpflichtet ist, das Werk abzunehmen. Jede Lieferung in die Produktivumgebung würde auch zu einer notwendigen Abnahme des Werkes führen. In einem Dienstleitungsverhältnis können diese Begleitumstände aber abgemildert werden.

Der Softwarelieferprozess muss darauf ausgerichtet werden, die Vorteile die sich aus den iterativen Vorgehen agiler Modelle ergeben, auch gewinnbringend ausnutzen zu können. Dieser Softwarelieferprozess sollte unabhängig davon sein, ob es sich um eine Web-Anwendung mit tausenden anonymen Nutzern oder um eine Anwendung im Back-Office einer Versicherung mit nur 100 Nutzern zu gewöhnlichen Bürozeiten handelt.

Einspielen eines Hot-Fix: Ein kritischer Systemfehler in einer Banking-Software, der erst einige Wochen nach der Abnahme auffällt, erfordert einen hohen Personalaufwand und verursacht unnötige Kosten. Diese entstehen vorwiegend bei der Fehlersuche, da die

[18] Vgl. [Rea10]
[19] Vgl. [HF11, S. 11]
[20] Vgl. [Ste04, S. 169 ff]

Implementierung der fehlerhaften Stelle möglicherweise schon Monate zurückliegt und die Erinnerung der Entwickler nicht mehr frisch ist. Zudem kommen Aufwände für die Vorbereitung des Release sowie für das Einspielen des Bugfixes hinzu. Die Aufwände, die hierdurch entstehen, sind unabhängig davon, ob sich es bei der fehlerhaften Stelle nur um ein einzelnes Zeichen oder eine größere Änderung handelt.

Ist eine Hot-Fix einzuspielen, geht dem häufig die Situation voraus, dass die Produktivumgebung nicht ordnungsgemäß arbeitet. Möglicherweise ist z. B. ein Web-Shop nicht mehr zu erreichen oder berechnet Preise falsch. Der dringende Bedarf, dieses Fehlverhalten schnellstmöglich zu beseitigten, führt dann zu einem Umgehen des regulären Lieferprozesses und der Qualitätssicherung. Neue kritische Fehler und Sicherheitslücken können so schnell in die Software einschleichen.

Bei einer kontinuierlichen Auslieferung der Software durch einen automatisierten Prozess könnten derartige Risiken minimiert werden. Eine fehlerhafte Implementierung, die auch durch eine sorgfältige Qualitätssicherung nicht erkannt wird und erst in der Produktivumgebung erkannt wird, ist schnell auf die letzte Änderung einzugrenzen. Die Fehleridentifizierung kann hierdurch beschleunigt werden und das fehlerhafte Verhalten schnell beseitigt werden.

Der Softwarelieferprozess muss auch die Lücke schließen können, die durch die schnelle Beseitigung eines Fehlers auftritt. Die qualitative Auslieferung von Software muss auch für schnelle Hot-Fixes gelten. Das Umgehen des regulären und auf Qualitätssicherung gestützten Lieferprozesses sollte auch für das schnelle Einspielen eines Notfallpatches nicht hingenommen werden. Ein automatisierter Prozess garantiert auch für Hot-Fixes eine gleichbleibende Qualität.

Komplexität der Softwarelieferung: Der Softwarelieferprozess ist das Ausrollen einer neuen Software in die Produktivumgebung. Dieser Prozess ist mit einem hohen Aufwand an Vorbereitung und personellen Ressourcen verbunden. Ein Beispiel aus dem Projektkontext von adesso ist das Ausrollen einer neuen Softwareversion bei einem großen Versicherer. Dieser Vorgang konnte nur als "*Night-Session*" durchgeführt werden nachdem das reguläre Personal gegangen war. Um die neue Version der Anwendung zu installieren sind Server nacheinander herunter gefahren, anschließend aktualisiert und wieder hochgefahren worden. Ein *Smoke-Test* zeigte die grundlegende Funktionalität der Anwendung. In dieser Situation war das gesamte Team hochgradig angespannt, da nicht endgültig zweifelsfrei vorher geprüft werden konnte, ob der Aktualisierungsvorgang gelingt und das System anschließend wieder seinen regulären Dienst aufnimmt.

Ein manueller Lieferprozess kann nach Ansicht von Humble und Farley vielschichtige Fehler verursachen, deren Gründe sich nicht immer unmittelbar identifizieren lassen.

Der manuelle Lieferprozess ist fragil, da er auf das korrekte und disziplinierte Vorgehen der Ausführenden angewiesen ist. Ein automatisierter Prozess kann diese Komplexität mit einem einmaligen Aufwand in die Automatisierung einfangen. Der Prozess ist dann immer in gleichbleibender Qualität ausführbar.[21]

Spätes Feedback: In dem von Steinweg beschriebenen Einführungsprozess für neue Software ist ein Pilotbetrieb dieser vorgesehen. Beim Pilotbetrieb wird eine Software durch einen ausgewählten Nutzerkreis zukünftiger Anwender auf Performanz und Stabilität geprüft. Der Pilotbetrieb folgt auf die vorläufige Werkabnahme, was die Fertigstellung der Anwendung aus Sicht des Herstellers impliziert. Werden im Pilotbetrieb Fehler ausfindig gemacht, ist der Hersteller zur Nachbesserung verpflichtet.[22]

Werden Fehler in dieser Phase noch ausfindig gemacht, sind diese vom Entwicklungsteam nur noch mit hohem Aufwand zu beheben. Die Lokalisierung der Fehler ist zeitaufwändig, da zwischen Implementierung und Auslieferung mehrere Monate vergangen sein können. Dieses Vorgehen bezeichnen Humble und Farley als ein "*common release antipattern*" für den Softwarelieferprozess.[23] Zudem ist in dieser Phase der durchaus nützliche Input des Testteams der Pilotphase kaum noch verwertbar. Das Agile Manifest stellt die schnelle Bereitstellung von funktionierender Software sowie ein schnelles Feedback an das Entwicklungsteam in den Mittelpunkt des Softwareentwicklungsprozesses.[24]

Der Softwarelieferprozess selbst sollte dem Entwicklungsteam ein frühestmögliches Feedback über den Zustand der Quellcode-Basis und damit der Software ermöglichen. Je früher ein Fehlverhalten aufgedeckt wird, desto einfacher und effizienter wird deren Beseitigung.

2.3. Konzepte von Continuous-Delivery

Die nachfolgend beschriebenen Konzepte basieren auf der Arbeit von Humble und Farley, die diese in ihrem Buch über Continuous-Delivery geben. Diese Konzepte stellen, dem Erachten von Humble und Farley nach, die notwendige Basis dar, um eine zentrale und automatisierte Deployment-Pipeline für den Softwarelieferprozess zu implementieren.[25]

[21] Vgl. [HF11, S. 5-10]
[22] Vgl.[Ste04, S. 179 ff]
[23] Vgl. [HF11, S. 7 ff]
[24] Vgl. [B+01]
[25] Vgl. [HF11, S. 3 ff]

2.3.1. Feedback-Prozess

Software kann in vier Komponenten zerlegt werden: der ausführbare Code, die Konfiguration der Software, die Laufzeitumgebung und Daten mit denen die Software arbeitet. Das Verhalten der Anwendung wird von allen vier Komponenten beeinflusst, was es erfordert, alle Komponenten unter Kontrolle zu halten. Eine Möglichkeit diese Kontrolle auszuüben ist jede Änderung an der Quellcode-Basis in ausführbare Software zu wandeln, also zu kompilieren, und in ihrer geplanten Ausführungsumgebung zu testen. Das Entwicklungsteam erhält so ein schnelles Feedback, welches hilft, die Qualität des aktuellen Softwarestandes zu beurteilen. Kommt es zu einem Fehler, kann dieser schnell identifiziert und behoben werden.

Unabhängig davon, ob es sich um ein Testsystem oder das Produktivsystem handelt, sollte es keine Änderung der erstellten Softwarepakte, im weiteren Verlauf allgemein als Binaries bezeichnet, mehr geben. Werden die Binaries vor Auslieferung in die Produktivumgebung neu kompiliert und es kommt zu unerwarteten Fehlern im Produktivsystem, kann nicht sicher ausgeschlossen werden, dass diese Fehler erst durch das erneute Kompilieren auftraten. Einmal kompilierte Binaries sollten während des gesamten Lieferprozesses nicht mehr verändert werden. Ein dann auftretendes Fehlverhalten im Produktivsystem kann so z. B. auf eine veränderte Konfiguration eingegrenzt werden.

Unterschiedliche Umgebungen erfordern auch häufig unterschiedliche Konfiguration der Software. Konfigurationsparameter für unterschiedliche Ausführungsumgebungen sollten, Humble und Farley zufolge, die einzigen veränderbaren Werte sein. Ändern sich diese Daten, muss die Anwendung auf die neuen Einstellungen hin getestet werden.

Zum Feedback über den Zustand der Quellcode-Basis gehört die Qualitätssicherung durch Softwaretests. Komponenten- bzw. Unit-Tests gehören zur ersten Stufe des Feedback-Prozesses. Dieser soll zeigen, dass sich alle Komponenten, losgelöst betrachtet, der Erwartung entsprechend verhalten.

Der Test von einzelnen Komponenten kann nur eine Aussage über das Verhalten einzelner Methoden und Klassen der Anwendung liefern. Ob die Komponenten und Module einer Anwendung wie erwartet zusammenarbeiten, müssen Integrationstests ermitteln. Diese bilden die zweite wichtige Stufe im Feedback-Prozess und erfordern die Ausführung der Anwendung in einer dem Produktionssystem angelehnten Umgebung. Nimmt ein Entwickler eine Anpassung einer Funktionalität vor, bei der ein auf dem Entwicklungssystem ausgeführter Komponententest keine Fehler anzeigt, ist noch nicht sichergestellt, dass die Änderung keinen negativen Einfluss auf z. B. abhängige Komponenten hat. Das korrekte Zusammenspiel aller Komponenten wird durch Integrationstests sichergestellt. Scheitert dieser an der neuen Version, obwohl die vorherige Version stabil lief, kann die Ursache schnell aus dem Änderungsprotokoll der Versionsverwaltung ermittelt und die

fehlerhafte Stelle angepasst werden.

Die dritte Stufe des Feedback-Prozesses, die eine Aussage zum Zustand der Quellcode-Basis ermöglicht, ist die Prüfung der Akzeptanzkriterien. Dabei wird die Erfüllung der für die Abnahme der Software relevanten funktionalen und nicht-funktionalen Anforderungen geprüft. Können diese kontinuierlich mit jeder Änderung der Quellcode-Basis geprüft werden, kann eine Aussage zur Lieferfähigkeit einer jeden Version gemacht werden.[26]

Ein guter Ansatz, um in einem neuen Projekt frühzeitig zu einer Aussage über den Erfüllungsgrad der Akzeptanzkriterien zu kommen, ist die Umsetzung von Behavior-Driven-Development, kurz BDD. BDD ist eine Weiterentwicklung des Test-Driven-Development, kurz TDD. Bei TDD werden Komponententests einer neuen Funktionalität vor deren Implementierung erstellt. Nach Dan North besteht ein wesentliches Problem bei der Umsetzung von TDD darin, dass Entwickler am Anfang nicht genau wissen, wo sie beim Testen beginnen bzw. was konkret sie testen sollen und wie viel. Dan North versuchte diese Probleme zu lösen und dabei die guten Seiten von *TDD* noch mehr mit den agilen Methoden zu verknüpfen. Beim BDD-Ansatz werden aus den Akzeptanzkriterien, mit den Schlüsselwörtern *given*, *when* und *then*, Testfälle generiert. Akzeptanzkriterien können so vor dem Beginn der Entwicklungsarbeit, bzw. auch parallel zu diesen, in automatisierte Testfälle umgesetzt werden.[27]

2.3.2. Zugewinn für das Entwicklungsteam

Der Nutzen von Continuous-Delivery liegt in der Vereinfachung des Deployment-Prozesses. Das Entwicklungsteam, das Test-Team, der IT-Betrieb und der Support soll in die Lage versetzt werden, einen bestimmten Softwarestand in eine beliebige Ausführungsumgebung deployen zu können. Kern dieser Ideen ist ein Self-Service-Portal. Durch dieses Portal könnte sich das Test-Team die aktuelle Version der Anwendung in eine Testumgebung selbst deployen, um z. B. explorativ zu testen oder Design und Layout zu überprüfen. Das Support-Team kann einen aus dem Testdurchlauf und vom *Quality-Gate* als lieferfähig eingestuften Hot-Fix in die Produktivumgebung ausrollen lassen.

Durch die Möglichkeit, jede Version einer Software, auch ältere, in eine bestimmte Umgebung bereitstellen zu können, können plötzlich auftretende und bisher unbekannte Verhaltensweisen einer verifiziert werden. So kann geprüft werden, ob ein Verhalten auch schon in vorhergehenden Versionen aufgetreten ist und bisher nur nicht entdeckt wurde oder es sich um ein neues Fehlverhalten handelt. Auf diese Weise kann das Auftreten ein Fehlverhalten auf eine bestimmte Änderung zurückgeführt werden. Zudem kann bei Bedarf jederzeit auf eine ältere Version mit Knopfdruck zurück gewechselt werden die

[26] Vgl. [HF11, S. 13 ff]
[27] Vgl. [Nor06]

als stabil bekannt ist.

Humble und Farley weisen besonders auf den zunehmenden Druck vor einem geplanten Release hin, der auf dem Entwicklungsteam lastet. Die Motivation des Entwicklungsteams fassen sie wie folgt zusammen: "*Just get something working*". Die Möglichkeit am Tag des Release, dieses einfach mit einem einfachen Knopfdruck ausführen zu können, schafft Freiräume für die eigentliche Tätigkeit, qualitative Software zu erstellen. Geht beim Deployment in die Produktivumgebung doch etwas schief, kann die vorherige Version schnell wieder hergestellt werden.

Humble und Farley sind der Ansicht, je häufiger ein Release durchgeführt wird, desto kleiner ist das Delta zwischen neuer und alter Version und so geringer das Risiko unerwarteter Fehler. Der Softwarelieferprozess wird nach ihrer Ansicht damit stabiler und verlässlicher.[28]

2.3.3. Der Release-Candidate

In einem automatisierten Lieferprozess ist es erforderlich, die Eigenschaften einer lieferfähigen Software zu definieren. Hierfür führen Humble und Farley das Konzept des Release-Candidate ein.

Jede Änderung an der Code-Basis führt möglicherweise zu lieferfähiger Software, einem Release-Candidate. Ob eine bestimmte Version der Software auch die Eigenschaften eines Release-Candidate besitzt, muss zuvor validiert und getestet werden. Ein Release-Candidate ist eine bestimmte Version, in der keine Fehler in den Komponenten- und Integrationstests gefunden wurden, die Metriken wie Code-Abdeckung sowie alle funktionalen und nicht-funktionalen Akzeptanzkriterien erfüllt werden konnten.

Jede Änderung an der Quellcode-Basis fügt der Software einen Wert hinzu. Im Gegensatz hierzu können sich mit jeder Änderung auch Fehler in das System einschleichen. Eine Überprüfung erfordert die Ausführung des Systems in einer an das Produktivsystem angelehnten Umgebung. Dies ist ein Prinzip von Continuous-Integration, kurz CI. Continuous-Integration stellt einen Grundpfeiler für die Deployment-Pipeline in Continuous-Delivery bereit.

Humble und Farley führen an, dass bei Projekten, die nicht kontinuierlich ein Release in eine an die Produktivumgebung angelehnte Testumgebung durchführen, die Integration der Anwendung mit allen zugehörigen Komponenten gerne in eine spätere Projektphase verschoben wird. Nach der Erfahrung von Humble und Farley ist die Integration eines Softwaresystems unvorhersehbar und schwer zu handhaben. Je später die Integration des Gesamtsystems erfolgt, desto aufwendiger wird die Zusammenführung sowie die Besei-

[28] Vgl. [HF11, S. 17 ff]

tigung von Fehlern, die erst zu diesem Zusammenhang sichtbar werden. Je früher die Integration des Gesamtsystems betrieben wird, desto geringer der Aufwand. Der Release-Candidate integriert frühestmöglich alle Komponenten miteinander und stellt die kontinuierliche Lieferfähigkeit der Software her.[29]

2.3.4. Implementierung eines wiederholbaren und verlässlichen Lieferprozesses

Um Continuous-Delivery verwirklichen zu können, müssen, nach Ansicht von Humble und Farley, die nachfolgend beschriebenen Prinzipien umgesetzt werden.[30]

Wenn Software gut getestet wird, sollte es einfach sein, diese in die Produktivumgebung ausliefern zu können. Das Deployment einer neuen Version sollte nur einen Knopfdruck bedeuten. Automatisiert werden sollen alle Teilschritte im Softwarelieferprozess wie Build, Test und Deployment. Ein automatisiertes Deployment setzt voraus, dass die Zielumgebungen mittels Programm oder Konfiguration verwaltet und gesteuert werden können. Zur Laufzeit der Deployment-Pipeline müssen möglicherweise benötigte Softwarekomponenten und Infrastrukturdienste installiert oder angepasst werden.

Manuelle Stufen im Softwarelieferprozess bleiben das explorative Testen, die Demonstration der Funktionsfähigkeit und die Compliance. Compliance beschreibt die Einhaltung von Regelungen, die für eine IT-Landschaft in einem Unternehmen getroffen wurden oder durch gesetzliche Bestimmungen eingehalten werden müssen, um einen bestimmten IT-Service bereitstellen zu können.[31]

Humble und Farley gehen davon aus, dass die Schritte, die in einer automatisierten Deployment-Pipeline ablaufen, am Anfang einfacher sind, diese manuell auszuführen. Nach der zehnten manuellen Ausführung ist es aber ein typisches menschliches Verhalten, diese Vorgänge ungenauer und weniger konzentriert als beim ersten Mal durchzuführen. Auf diese Weise wird der Prozess fragil und ist von der Disziplin der Beteiligten abhängig, die aber nicht erwarten werden darf. Deshalb plädieren sie, zu Beginn eines Projektes in die Automatisierung des Deployments zu investieren.

Der Versionsverwaltung wird in diesem Prozess eine zentrale Bedeutung zugeschrieben. Neben der Verwaltung der Quellcode-Basis müssen die Konfigurationselemente der Ausführungsumgebungen verwaltet werden. Führt die Änderung der Einstellung zu einem Fehler im Test- oder Produktivsystem, ist es leicht, über die Versionsverwaltung die fehlerhaften Einstellungen zu identifizieren. Solche Konfigurationselemente können z. B. Test-Skripte, Netzwerkeinstellungen, Deployment-Skripte, Datenbank-Skripte, In-

[29] Vgl. [HF11, S. 22]
[30] Vgl. [HF11, S. 24]
[31] Vgl. [Teu11]

stallationsskripte für Infrastrukturkomponenten, benötigte Bibliotheken und technische Dokumentationen sein.

Ein zentraler Lieferprozess ist neben der technischen Komponente auch eine organisatorische Aufgabe. In kleineren Teams besteht zumeist die vollständige Kontrolle über die benötigten Ressourcen. Größere Unternehmen bauen durch ihre Organisationsstrukturen möglicherweise auch Kommunikationsbarrieren zwischen den Beteiligten im Entwicklungs- und Lieferprozess ein. Entwickler, Tester und Systemadministratoren sind dann unterschiedlichen Organisationseinheiten unterstellt und räumlich voneinander getrennt. Die Ziele von DevOps geben hier einen guten Ansatzpunkt, um Strukturen zu schaffen, in der eine Kultur der ungehinderten Kommunikation mit einer gemeinsamen Verantwortlichkeit für Qualität und Nutzen eines IT-Dienstes entsteht.

Ein weiterer zentraler Gedanke, der auf die DevOps-Idee aufbaut, ist die *Definition of Done*. Nach Humble und Farley ist ein neues Feature erst dann als fertig anzusehen, wenn es fehlerfrei in der Produktionsumgebung ausgeführt werden kann. *Done* bedeutet, dass eine Funktionalität einer Software erfolgreich einer repräsentativen Nutzergruppe demonstriert und durch diese ausprobiert wurde. Diese führt zurück zur Notwendigkeit des Pilotbetriebs mit einem eingeschränkten Benutzerkreis, dieser unterliegt hier aber dem iterativen Vorgehen agiler Softwareentwicklung und wäre dann für einzelne Funktionalitäten durchzuführen.[32]

Bei Humble und Farley ist der Lieferprozess kein statisches System, welches implementiert und dann unverändert betrieben werden kann. Viel mehr sehen sie die Notwendigkeit, diesen Prozess in einem "*continuous improvement*" zu entwickeln. Die erste Implementierung der Deployment-Pipeline steht am Anfang eines Projektes, mit den minimal möglichen Schritten, das geplante System in die Produktivumgebung auszuliefern. Mit der Umsetzung der geplanten Funktionalitäten nimmt das System an Größe und Komplexität zu. Die Deployment-Pipeline muss den geänderten Anforderungen kontinuierlich angepasst werden. Hierzu gehören neue Maßnahmen der Qualitätssicherung, die Implementierung von Quality-Gates oder die Berücksichtigung der Datenbank in der Deployment-Pipeline.

2.4. Exkurs: ITIL und Continuous-Delivery

ITIL ist ein Referenzmodell für die Compliance eines Unternehmens und stellt einen Leitfaden für das IT-Service-Management bereit. Continuous-Delivery berührt mit dem Konzept einer automatisierten Deployment-Pipeline besonders den Bereich der Service-Transition. ITIL Service-Transition beschreibt Prozesse und Verfahren, um neue oder

[32] Vgl. [Ste04, S. 178]

auch geänderte IT-Services in den operativen Betrieb zu heben. Die Service-Transition folgt auf das Service-Design und mündet dann in Service-Operation. Die Kernprozesse von Service-Transition sind Transition-Planning & Support, Change-Management, Service-Asset & Configuration-Management, Release-Management, Release & Deployment-Management.[33]

Das Change- und Release Management ist bei der Implementierung einer Deployment-Pipeline zu beachten, wenn z. B. der Kunde, für den ein neuer IT-Dienst umgesetzt werden soll, intern nach dem ITIL Compliance-Framework vorgeht.[34]

Erik Minck ist der Ansicht, dass das Change- und Release-Management von ITIL Synergien mit DevOps besitzt und die gleichen Ziele verfolgt, schnell nutzbringende und qualitative IT-Dienste und Anwendungen bereitstellen zu können.[35]

Ein wichtiges Merkmal im Release Management ist der Erhalt der Systemintegrität der bestehenden Infrastruktur. Unvorhergesehene Beeinträchtigungen müssen vor der Einführung neuer oder geänderter Dienste vermieden werden.[36]

ITIL hat vornehmlich in größeren Unternehmen Rückhalt, da bei diesen eine risikoscheue Haltung im IT-Betrieb zu beobachten ist. Bei kleineren Unternehmen und Start-Ups sind hingegen häufiger die Konzepte von DevOps anzufinden.[37]

Zu den Aktivitäten im Change-Management, der Ablaufsteuerung für Veränderungsmaßnahmen, gehört die Dokumentation aller Änderungsanfragen, der *R*equest *F*or *C*hange, das Zulassen und die Beurteilung der RFCs, die Autorisierung, die Koordinierung der Implementierung und das Prüfen des Ergebnisses. Aktivitäten im Release-Management sind die Erstellung von Release-Richtlinien, die Planung eines Release, die Erstellung von Testfällen, die Steuerung der Implementierung, der Support bei der Einführung und der Abschluss des Projektes. Als Methodiken empfiehlt ITIL die Trennung der Umgebungen für Entwicklung, Test und Produktion sowie die Nutzung von Verteilungswerkzeugen für die neue Software.[38]

Die Change- und Release-Richtlinien von ITTL weisen in Richtung Continuous-Delivery. Continuous-Delivery schlägt die automatisierte Verteilung von Software in die Produktivumgebung vor sowie ein sequenzielles Roll-Out und die Implementierung einer Push-Funktionalität. Die Verifizierung eines Release erfolgt in einer möglichst nahen Abbildung der Produktivumgebung durch verschiedene Testverfahren. Verifiziert werden dabei die Fähigkeiten einer Software in die geplante Umgebung installiert werden zu können, der Test der Integration mit den vorhandenen Komponenten sowie eine Un-

[33] Vgl. [Böt08]
[34] Vgl. [Böt08]
[35] Vgl. [Min12]
[36] Vgl. [Böt08, S. 108]
[37] Vgl. [Min12]
[38] Vgl. [Böt08, S. 89 ff]

tersuchung der Auswirkung auf die Systemstabilität als auch auf das Systemverhalten. Ein neues System muss beweisen, dass es sich wie erwartet verhält.[39]

Die Deployment-Pipeline unterstützt das Change- und Release-Management effektiv und kann unter folgenden Umständen als ein Revisions- und Compliance-Werkzeug angesehen werden:

- Der Prozess oder das Werkzeug muss die Version einer Software verwalten, die in einer Umgebung ausgeführt wird.
- Die Ergebnisse der Zwischenschritte, die innerhalb der Pipeline ablaufen, müssen dokumentiert und ausgewertet werden. Hierzu gehören z. B. die Testprotokolle der automatisierten Testdurchläufe und die Protokolle der Systemänderungen, wenn automatische Änderungen an der Infrastruktur vorgenommen werden.
- Es muss ersichtlich sein, welcher Mitarbeiter oder welches System einen Prozess angestoßen hat und wann.
- Ein spezifisches Deployment muss sich auf eine Revisionsnummer im Versionsverwaltungssystem zurückführen lassen.
- Prozessmetriken, wie z. B. die Durchlaufzeit der einzelnen Phasen müssen dokumentiert sowie ein kontinuierliches Monitoring- und Feedback-System implementiert werden.

2.5. Zusammenfassung

DevOps und Continuous-Delivery haben einen starken Fokus auf Web-Technologien. Die Ideen und Konzepte hierzu stammen von Personen, die im Umfeld von Web-2.0-Anwendungen und Cloud-Computing sehr aktiv sind. Bei DevOps geht es um ein gemeinsames und zielorientiertes Zusammenwirken von Entwicklern, Testern und Systemadministratoren. Hierfür werden gemischte Teams vorgeschlagen, die eine gemeinsame Verantwortung für die umzusetzenden Funktionalitäten tragen. Alle im Team leisten dabei einen Betrag, den anvisierten wirtschaftlichen Nutzen zu erzielen. Continuous-Delivery beschreibt einen automatisierten Auslieferungsprozess, der dem Entwicklungsteam wiederkehrende Aufgaben abnimmt und diese präziser und schneller ausführen kann. Dem Entwicklungsteam wird durch das Auslieferungssystem ein schnelles Feedback über den Zustand der Quellcode-Basis gegeben. Im Mittelpunkt stehen automatisierte Komponenten, Integrations- uns Akzeptanztests. Zudem ist jederzeit ersichtlich, welche Version

[39] Vgl. [Böt08, S. 114]

in welcher Umgebung ausgeführt wird. Der automatisierte Auslieferungsprozess reduziert Fehler, die durch manuelles Deployment der Software und manuelles Konfigurieren der Systemumgebung entstehen können. In der zeitkritischen Phase vor einem Release wird durch einen verlässlichen und oft erprobten Lieferprozess, Stress und Druck vom Entwicklungsteam genommen. Dieses kann sich so besser auf die qualitative Umsetzung der Funktionalitäten konzentrieren. Für Unternehmen, die Compliance anstreben, ist DevOps und Continuous-Delivery eine Möglichkeit, die Anforderungen aus dem ITIL Change- und Release-Management in einer leichtgewichtigen Version zu erfüllen.

3. Deployment-Pipeline und Liefersystem

Nach DevOps kann die Deployment-Pipeline als Realisierung eines Liefersystems betrachtet werden. Die Deployment-Pipeline bildet den Auslieferungsprozess auf ein Liefersystem ab. Die Anforderungen aus diesem Abschnitt haben eine Implikation auf die zu untersuchenden Werkzeuge und dienen als Vergleichsbasis für das derzeitige Liefersystem bei adesso. Dieser Abschnitt befasst sich deshalb mit den Verfahren der Automatisierung und den Komponenten sowie dem Aufbau des Liefersystems.

Humble und Farley schreiben über eine häufig anzutreffende Verschwendung von Ressourcen im Auslieferungssystem, die durch Prozessverzögerungen hervortreten. Als Beispiel geben sie an, dass der IT-Betrieb auf Unterlagen des Entwicklungsteams wartet, wie die ausstehende Dokumentation für ein einzuführendes Softwaresystem. Das System kann vorher vom IT-Betrieb nicht in Produktion gebracht werden. Analog hierzu kommt es vor, dass das Testteam auf eine neue Version der Anwendung wartet, um diese zu testen oder umgekehrt das Entwicklungsteam den letzten Bug-Report der Testabteilung benötigt, um Fehler beseitigen zu können. Zudem existieren nach Ansicht von Humble und Farley bei zu später Integration der Softwarekomponenten, die Risiken, dass die Architektur nicht den Anforderungen entspricht. Lange Zyklen im Feedback-Prozess erhöhen die Gefahr, fehlerhafte Software auszuliefern. Dies kann auch die Projektkosten am Ende noch einmal deutlich steigen lassen. Wenn ein Release scheitert oder sich verspätet, kann der betriebswirtschaftliche Nutzen hinter einem IT-Dienst nicht in der anvisierten Höhe ausgeschöpft werden. Ferner drohen hierdurch möglicherweise gar empfindliche Verluste, wenn z. B. in einem umkämpften Markt Kunden einer Plattform wegen Ausfällen und instabilem Verhalten zu einem anderen Anbieter wechseln.[1]

3.1. Continuous-Integration

Continuous-Delivery und das Konzept der Deployment-Pipeline bauen auf Continuous-Integration auf. Im Prozess von Continuous-Integration wird die Software auf die tech-

[1] Vgl. [HF11, S. 105 ff]

nische Funktionsfähigkeit, die Einhaltung von Standards, auf Schwachstellen sowie das Zusammenspiel der Komponenten geprüft. Die Erfüllung von Akzeptanzkriterien ist hierin aber nicht mit eingeschlossen. Das Kriterium lieferfähig ist damit noch nicht erfüllt.[2]

Das Ziel von Continuous-Integration beschreibt Paul M. Duval mit "*build software at every change*". Dies führt dazu, dass auch kleine Änderungen an der Code-Basis in einem Build- und Test-Prozess münden. Diese Prozesse sollen dem Entwicklungsteam ein schnelles Feedback über mögliche Integrationsfehler geben, die bei komplexen und modularen Softwaresystemen auftreten können. Continuous-Integration stellt sicher, dass alle entwickelten Komponenten in der geplanten Weise zusammenarbeiten.[3]

Nach Duval ist die zu späte Integration der Softwarekomponenten ein häufig anzutreffendes Problem. Die zu späte Integration betrifft häufig Projekte, in denen verschiedene Entwicklungsteams parallel an verschiedenen Softwarekomponenten arbeiten. Der Frage, ob die Komponenten auch tatsächlich in der geplanten Art und Weise zusammenarbeiten können, wird erst kurz vor dem Ende der Projektlaufzeit nachgegangen. Da dann fast alle Komponenten fertiggestellt sind, entsteht in der Phase vor dem geplanten Release noch einmal ein großer Aufwand, um die betroffenen Komponenten anzupassen[4]

Continuous-Integration bietet hier eine frühe Identifizierung von Fehlern, Problemen und Schwachstellen, die bei der Zusammenarbeit von Softwarekomponenten auftreten können und betrachtet in diesem Zusammenhang auch die Komplexität des Quellcodes durch statische Codeanalyse. Hierunter fallen z. B. die Umsetzung von Programmierstandard und die Abdeckung von Quellcodes durch Testfälle. Über jede Änderung an der Quellcode-Basis kann mit Continuous-Integration eine Aussage über die Auswirkung auf die Qualität getroffen werden. Das Verhalten der Integrierbarkeit der Komponenten wird so kontinuierlich geprüft und es können negative Einflussfaktoren durch verschiedene Metriken gemessen werden. Sollte die fehlerfreie Funktion der Anwendung nach einer Änderung an der Quellcode-Basis nicht mehr gegeben sein, ist die Ursache durch eine Analyse der zuletzt durchgeführten Änderung leicht auszumachen. Das Entwicklungsteam erhält ein unmittelbares Feedback und kann die notwendigen Änderungen vornehmen, um die Anwendung in einen lieferfähigen Zustand zurück zu setzen.[5]

Humble und Farley sehen Continuous-Integration als sehr gut geeignet und als erste Stufe im Softwarelieferprozess an. Continuous-Integration ist ihrer Ansicht nach aber nicht ausreichend. Die alleinige Fokussierung auf das Entwicklungsteam im Lieferprozess geht ihnen nicht weit genug.[6]

---

2 Vgl. [Duv07]
3 Vgl. [Duv07, S. 4-6]
4 Vgl. [Duv07, S. 4-6]
5 Vgl. [Duv07, S. 4-6]
6 Vgl. [HF11, S. 105]

Das Ende der Prozesskette der Integrationssysteme ist der Eintrittspunkt des IT-Betriebes in den Lieferprozess. Für die Installation und den korrekten Betrieb der Test- und Produktivumgebung ist in einer regulären Aufteilung der IT-Betrieb verantwortlich. Dies ist der Anknüpfungspunkt der DevOps -Bewegung. Diese betrachtet eine Anwendung nicht nur als kompilierten Code, sondern vielmehr auch im Zusammenhang mit dem Management der Umgebung in der diese ausgeführt wird. Die Entwicklung und der Betrieb einer Anwendung müssen für die Umsetzung von Continuous-Delivery ineinandergreifen. Dieses wird durch die Deployment-Pipeline verwirklicht. Das Liefersystem besteht in seiner ersten Stufe aus einem CI-System. Für die zweite Stufe werden werden eine Reihe geeigneter Werkzeuge benötigt, welche die anfallenden Aufgaben während der Auslieferung einer Anwendung automatisieren können. Mit Continuous-Delivery wird das Konzept von Continuous-Integration um automatisierte Akzeptanztests sowie das automatisierte Ausliefern einer Anwendung in die Produktivumgebung erweitert.[7]

3.2. Erste Lösungsstrategie für die Deployment-Pipeline

Die Deployment-Pipeline ist eine Kombination aus technischen und organisatorischen Lösungen und manifestiert den Lieferprozess. Ausgangspunkt ist eine Wertstromanalyse, "*from conzept to cash*". Um eine Deployment-Pipeline implementieren zu können, müssen alle Schritte in einem Auslieferungsprozess analysiert werden. Ausgangspunkt der Wertstromanalyse ist damit das *commit* von Änderungen der Quellcode-Basis in die Versionsverwaltung, die ein Entwickler vorgenommen hat. Der Weg, den der Quellcode von diesem Punkt aus bis hin zur Ausführung in einem Produktivsystem nimmt, muss vorweg genau untersucht werden. Im Zentrum stehen die Fragen: wer macht was, was wird gemacht und wie wird es gemacht? Diese Fragen geben einen Anhaltspunkt bei der Modellierung eines automatisierten Prozesses.

In einem kleinen internen Projekt wurden folgende Beobachtungen gemacht: Freitag 14:00, *Entwickler 1* hat eine neue Funktionalität implementiert und seine Komponenten auf dem Entwicklungssystem mit Unit-Tests getestet. Nach dem erfolgreichen Verlauf des Tests werden die Änderungen an der Quellcode-Basis in das Versionsverwaltungssystem commited. *Entwickler 1* möchte gerne wissen, ob die neuen Funktionalitäten auch mit der benötigten Infrastruktur zusammenspielen. Hierfür möchte er die Version auf dem dafür bereitgestellten Testserver installieren, da nur auf diesem alle notwendigen Infrastrukturkomponenten verfügbar sind. Leider hat nur *Entwickler 2* einen Remote-

[7] Vgl. [HF11, S. 105]

Zugriff auf den Testserver. *Entwickler 2* führt ein Update der lokalen Dateien durch und muss dabei einige Konflikte lösen, da *Entwickler 1* und *Entwickler 2* Änderungen an derselben Komponente durchgeführt haben. Nachdem alle Konflikte aufgelöst werden konnten, kompiliert *Entwickler 2* den Quellcode und packt die Anwendung für die Ausführung. Anschließend loggt sich *Entwickler 2* in die Konsole des Test-Servers ein und stoppt die Ausführung der älteren Softwareversion. Dabei muss er noch alle Log-Dateien und Anwendungszustände der älteren Version sowie auch das ältere Softwarepaket selbst löschen, da es sonst zu unerwarteten Fehlern kommen kann. Das Löschen übernimmt ein selbstgeschriebenes Shell-Skript, das im Ausführungsverzeichnis der Anwendung bereit steht. *Entwickler 2* kopiert das erstellte Paket der neuen Softwareversion in das Ausführungsverzeichnis des Test-Servers und startet die Anwendung über ein Startskript. Es kommt zu einem Fehler. Die neue Softwareversion benötigt noch eine weitere Bibliothek, die auf dem System installiert werden muss. Da *Entwickler 2* sich zwar in den Server einloggen kann, aber leider keine Administrationsrechte besitzt, um die benötigte Bibliothek selbst zu installieren, wendet er sich an den *Helpdesk* des IT-Betriebes. Da der *Helpdesk* ein Ticketsystem benutzt, sind die Mitarbeiter angehalten, dieses Ticketsystem zu nutzen und ihre Anfragen als E-Mail zu formulieren. *Entwickler 2* schreibt die E-Mail mit der Bitte, die benötigte Bibliothek zu installieren. Leider ist es Freitag 15:00 und alle Mitarbeiter sind ausgelastet. Das Ticket kann erst am Montag-Vormittag bearbeitet werden. Um 16:00 kommt der Abteilungsleiter auf das Entwicklungsteam zu, er hätte jetzt 15 Minuten Zeit, um sich den aktuellen Stand anzusehen und bittet um eine kurze Vorführung. Zu diesem Zeitpunkt gibt es aber bedauerlicherweise keine ausführbare Version der Anwendung, das Backup der alten Version wurde leider vergessen.

Für dieses Szenario können folgende wiederkehrende Schritte ausgemacht werden:

- Kompilieren des Quellcodes.
- Durchlaufen der Komponententests.
- Erstellen des Softwarepaktes.
- Stoppen der alten Anwendung.
- Archivieren der alten Ausführungsdateien, Anwendungszustände und Log-Dateien.
- Überprüfen und Aktualisieren der Systemkonfiguration.
- Kopieren und Starten der neuen Version.
- Informieren des Entwicklungsteams.

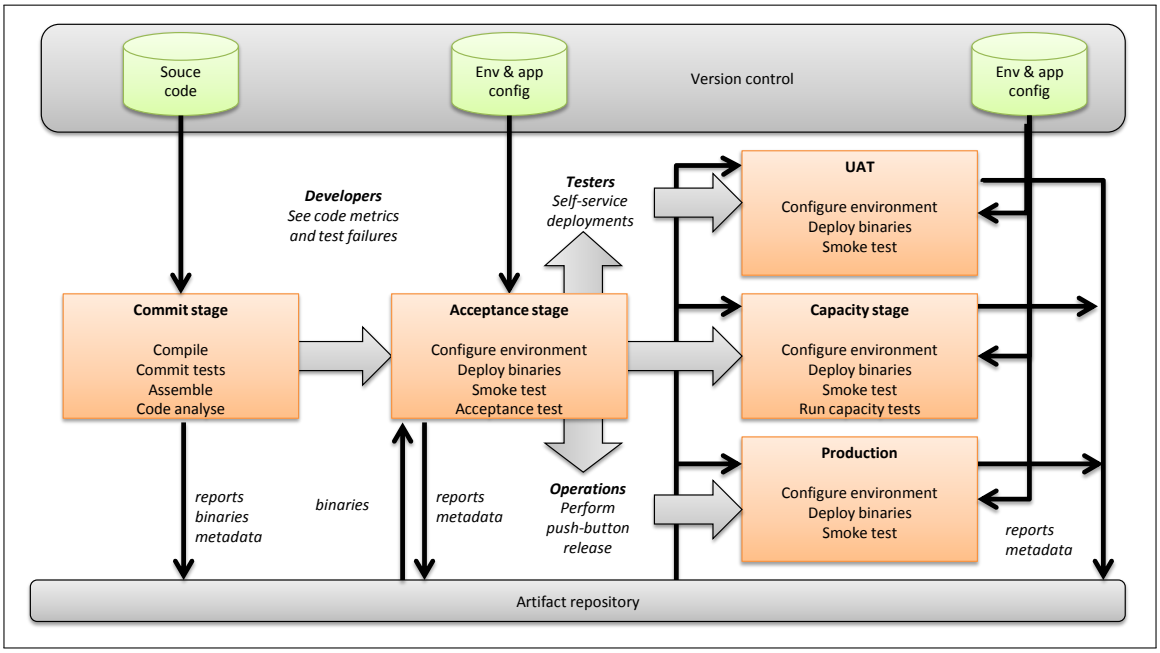
Das Kompilieren des Quellcodes für die Ausführung im Testsystem wird ausschließlich auf der Basis der aktuellen Version im Versionsverwaltungssystem durchgeführt. Dadurch wird vermieden, dass noch Änderungen in den Quellcode einfließen, die eine Fehleranalyse und ein Zurückführen auf bestimmte Änderungen erschweren. Alle Komponententests werden durchlaufen. Die Qualität auf Ebene der Komponenten ist gesichert. Scheitert der Testdurchlauf, scheitert der Prozess. Die kompilierten Dateien werden zu einem lieferbaren Paket zusammengefasst. Das Paket selbst wird mit der Versionsnummer der aktuellen Quellcode-Basis versehen. So lassen sich später verschiedene Versionen einer Anwendung verwalten. Über einen Remote-Zugriff wird ein Stopp-Skript aufgerufen, das die aktuelle Version der Anwendung stoppt. Anschließend wird ein Clean-Up-Skript aufgerufen, das alle Log- und die gespeicherten Anwendungszustände zusammen mit der alten Version für eine spätere Inspektion packt und archiviert. Das Paket wird mit der Versionsnummer des alten Softwarearchivs versehen. Die Deployment-Pipeline prüft die aktuelle Konfiguration des Systems und hat die Rechte, um benötigte Bibliotheken auf dem System installieren zu können. Die Datei, die eine Liste aller Abhängigkeiten enthält, wird aus dem Versionsverwaltungssystem geladen. Die neue Softwareversion wird auf das Testsystem kopiert und gestartet. Das Team wird über das E-Mail-System sowie das interne Nachrichtensystem informiert, dass eine neue Version der Anwendung auf dem Testsystem bereitgestellt wurde.

Die Automatisierung dieser Teilschritte kann durch eine Tool-Chain, eine ineinandergreifende Verkettung von Werkzeugen, erreicht werden. Dabei schließt der Softwarelieferprozess menschliche Interaktion nicht gänzlich aus. So kann das Deployments auf das Test- oder Produktivsystem von einem Mitglied des Entwicklungs-, Test- oder Support-Teams angestoßen werden. Build- und Komponententests werden durch die neue Version in der Versionsverwaltung aber automatisch angestoßen. Das Testteam bzw. das Entwicklungsteam sollte entscheiden können, ob eine bestimmte Version in eine Umgebung deployt wird. Die Schritte hinter diesem Prozess werden automatisch durchgeführt und das Team über den Erfolg oder über aufgetretene Fehler informiert.[8]

Für Humble und Farley stellt dieses Vorgehen einen schnellen, jederzeit wiederholbaren und verlässlichen Prozess dar. Auch dringende Hotfixes, die schnellstmöglich in die Produktivumgebung gebracht werden müssen, können diesen Prozess sicher durchlaufen.[9]

Für ein kleines Projekt, welches noch am Anfang der Entwicklung steht, genügt die geringe Komplexität der hier vorgestellte Deployment-Pipeline zunächst. Wächst die Größe der Anwendung und die Zahl der beteiligten Komponenten, muss die Deployment-

[8] Vgl. [HF11, S. 109]
[9] Vgl. [HF11, S. 109]



Quelle: nach [HF11, S. 111] .

Abbildung 3.1.: Deployment-Pipeline nach Humble und Farley.

Pipeline den gestiegenen Ansprüchen angepasst werden. Dies könnte z. B. bedeuten, dass das verwendete Schema einer Datenbank durch das Deployment-Skript aktualisiert wird oder in einem Server-Cluster einzelne Server gestoppt und wieder gestartet werden müssen.

3.3. Struktur der Deployment-Pipeline

Humble und Farley schlagen bei der Realisierung der Deployment-Pipeline ein phasenweises Vorgehen vor und nutzen hierfür den Begriff des *Staging*. Eine *Stage* bildet eine deutlich abgrenzbare Phase ab. Für die Umsetzung der Deployment-Pipeline sollen diese nur der Orientierung einer möglichen Einteilung dienen. Je nach gestellten Anforderungen an den Lieferprozess, ergeben sich andere Abläufen und Strukturen und die *Stages* müssen anders modelliert werden. Abildung 3.1 zeigt die Deployment-Pipeline, wie Humble und Farley sie vorschlagen. Grundlegende Elemente sind demnach:[10]

- Commit-Stage,
- Acceptance-Stage,
- User-Acceptance-Stage,

[10] Vgl. [HF11, S. 109-110]

- Capacity-Stage und
- Production.

3.3.1. Die Commit-Stage

Die Commit-Stage ist ein Prozess aus Continuous-Integration. Sie stellt sicher, dass das entwickelte System sich auf dem erforderlichen technischen Niveau befindet. Das bedeutet, dass der Quellcode kompiliert und es keine Compiler-Fehler gibt, die die Komponententests fehlerfrei durchlaufen und die statische Analyse des Quellcodes eine hohe Testabdeckung und die Einhaltung von Programmierstandards bescheinigen.[11]

Eine Änderung an der Quellcode-Basis im Versionsverwaltungssystem startet die Commit-Stage. Das CI-System reagiert mit einer neuen Instanz des Integrationsprozesses und kompiliert sowie testet dabei den neuen Softwarestand. Dem Entwicklungsteam sollte ein Feedback über den aktuellen Zustand des Quellcodes nach ca. fünf Minuten geben werden können. Nimmt dieser Prozess mehr Zeit in Anspruch, empfehlen Humble und Farley die Aufteilung der Unteraufgaben. Komponententests können z. B. auch in parallel laufende Tasks aufgeteilt werden und auf verteilten Systemen ausgeführt werden. Die Ergebnisse können anschließend zusammengefasst und vom Entwicklungsteam ausgewertet werden.[12]

Metriken der statischen Codeanalyse, die in dieser Phase getestet werden, sind:[13]

- Testabdeckung,
- die Prüfung auf Codedubletten,
- die zyklomatische Komplexität,
- die Untersuchung nach der afferenten und efferenten Kopplung,
- Warnungen und Programmierrichtlinien (*Code Style*).

Werkzeuge wie Sonar[14] verdichten die Testabdeckung auf einen prozentualen Wert. Dieser wird durch die Menge der im Test durchlaufenden Kontrollflusspfade bzw. der abgedeckten Codezeilen ermittelt. Wenn z. B. bei der Menge der Eingaben, mit denen getestet wird, in einer Fallunterscheidung nie der alternative Zweig der Fallunterscheidung erreicht wird, beträgt die Testabdeckung für dieses Konstrukt 50%.[15]

[11] Vgl. [HF11, S. 109-110]
[12] Vgl. [HF11, S. 105-120]
[13] Vgl. [HF11, S. 121]
[14] Weitere Informationen zu Sonar unter `http://www.sonarsource.org/`.
[15] Vgl. [Man10]

Die zyklomatische Komplexität wird durch die zyklomatische Zahl ausgedrückt. Diese misst die strukturelle Komplexität des Quellcodes durch einen Kontrollflussgraph. Dabei wird die Menge unabhängiger Pfade im Programmablauf betrachtet.[16]

Mit der afferenten Kopplung wird die Anzahl der Pakete beschrieben, die von Klassen innerhalb des untersuchten Paketes abhängen. Efferente Kopplung hingegen beschreibt das Gegenteil, die Anzahl von Klassen eines Paketes die von Klassen außerhalb des untersuchten Paktes abhängig sind. Diese Werte ermöglichen eine Beschreibung der Stabilität der Software. Demnach ist Stabilität $I = \frac{Ce}{Ca+Ce}$. Wobei I im Intervall $[0, 1]$ liegt, Ce für efferente und Ca für afferente Kopplung steht. Ein Wert von 0 deutet auf Stabilität und darauf hin, dass dieses Pakte hauptsächlich von anderen benutzt wird. Dies könnte z. B. für eine Schnittstellenbeschreibung der Fall sein oder auf einen Service hindeuten, der von anderen Paketen verwendet wird. Ein Wert von $0,5$ weist auf eine wechselseitige Beziehung zwischen den Paketen hin. Ein Wert von 1 zeigt eine deutliche Abhängigkeit von anderen Paketen. Ändern sich die verwendeten Pakete, wird dieses auch Auswirkungen auf die Funktionsweise des untersuchten Paketes haben.[17]

Eine von Sun für die Programmiersprache Java empfohlene *Code-Style*, also die Einhaltung von Programmierstandards, macht Vorgaben für die Organisation von Dateien, die Einrückungen im Quellcode, das Setzen von Kommentaren, die Deklaration von Variablen und die Vergabe von Namen. Ein Entwicklungsteam kann dabei weitere Richtlinien definieren. Ein verabredeter Standard unterstützt die Zusammenarbeit und erhöht die Wartungsfreundlichkeit. Deren Einhaltung lässt sich auf der Ebene der Entwicklungsumgebung sowie durch das CI-System in Form der statischen Code-Analyse prüfen.[18]

Ein CI-System sollte Artefakte wie Test-Protokolle und Binaries in einem zentralen Verzeichnis, einem Repository, verwalten können. Das ist notwendig, da nachfolgende Phasen auf diesem Ergebnis aufbauen und die Binaries in der Acceptance-Stage, User-Acceptance-Stage, Capacity-Stage und in Production, dem Produktivsystem, wiederverwenden.

Die in der Commit-Stage erstellten Softwarepakete sollten in einer späteren Phase nicht noch einmal kompiliert werden. Es könnte sonst nicht zweifelsfrei und revisionssicher garantiert werden, dass die dann gelieferte Version auch der getesteten exakt entspricht. Flexibilität und Wartbarkeit werden reduziert, wenn für bestimmte Umgebungen spezielle und angepasste Binaries erstellt werden müssen.[19]

---

16 Vgl. [SL05, S. 100 f]
17 Vgl. [Mar00, S. 23-24]
18 Vgl. [Sun97]
19 Vgl. [HF11, S. 113]

3.3.2. Acceptance-Stage, User-Acceptance-Stage, Capacity-Stage und Production

Ob ein Release-Candidate auch in die Produktivumgebung ausgeliefert werden kann, ist in der Commit-Stage noch nicht endgültig entscheidbar, da hier nur die technische Funktion überprüft wird. Eine Softwareversion in die Produktivumgebung zu bringen, setzt die Abnahme des Werkes und damit die Erfüllung der spezifizierten Akzeptanzkriterien aus den funktionalen und nicht-funktionalen Anforderungen voraus.

Die Acceptance-Stage prüft auf der Seite des Lieferanten, ob diese Anforderungen erfüllt werden und das Verhalten der Anwendung die Bedürfnisse des Nutzers erfüllt und der Spezifikation gerecht wird. Bei den nicht-funktionalen Anforderungen wie z. B. den Kapazitätstests sollte eine Auslagerung der Tests als eigene Stage, der Capacity-Stage, vorgenommen werden. Die Prüfung der Anwendung auf mögliche Auslastungsengpässe läuft gewöhnlich länger und sollte deshalb parallelisiert werden.[20]

Continuous-Delivery setzt voraus, dass alle Testfälle für jede neue Version der Quellcode-Basis durchlaufen werden. Damit kann der Quellcode auch auf Regression getestet werden und eine Verschlechterung der Qualität so schnell erkannt werden. Eine zuvor funktionierende Software, die nach einer Änderung bestimmte Qualitätskriterien nicht mehr erfüllt, kann schnell identifiziert und durch das Entwicklungsteam unverzüglich angepasst werden, damit diese Anforderungen wieder erfüllt werden.

Für die Verifizierung von Akzeptanz- und Kapazitätskriterien muss die Software in einer Umgebung ausgeführt werden, welche der Produktivumgebung entsprechen soll. Wenn Struktur und Konfiguration dieser Umgebungen zu sehr von denen der Produktivumgebung abweichen, kann nicht ausgeschlossen werden, dass es im Produktivsystem zu einem nicht vorhersehbaren Verhalten kommt. Handelt es sich beim Produktivsystem z. B. um ein komplexen und teuren Server-Cluster, wird es wirtschaftlich nicht möglich sein, die Produktivumgebung exakt nach zu bilden. Im Fall des Server-Cluster kann aber ein herunter skaliertes System für den Test bereitgestellt werden. Die Lasttests können entsprechend dem Verhältnis von Test- und Produktivumgebung skaliert werden.

Mit der User-Acceptance-Stage wird für jede Softwareversion ein Pilotbetrieb der Anwendung umgesetzt. Wird die Software durch einen externen Dienstleister als Werk erbracht, führt, sofern nicht anders vereinbart, das Ausliefern der Anwendung in die Produktivumgebung zur Abnahme des Werkes durch den Auftraggeber. Eine direkte Auslieferung nach jedem möglichen Commit eines Entwicklers ist in dieser Konstellation nicht möglich. Werden aber vertragliche Teillieferungen vereinbart, ist auch die Abnahme nur einen Teil, der im Gesamtpakt vereinbarten Funktionalitäten möglich, sodass die

[20] Vgl. [HF11, S. 124-128]

Abnahme des Werkes erst am Ende der Vertragslaufzeit endgültig vollzogen wird.

Die letzte Stufe im Liefersystem bildet die Auslieferung einer gut getesteten und den Anforderungen entsprechenden Version der Software in die Produktivumgebung. Ein Versagen in einer Teilphase verneint die Lieferfähigkeit der inspizierten Version. Humble und Farley schlagen für das Deployment eines Release-Candidate ein Self-Service-Portal vor. Von diesem aus können die Beteiligten im Lieferprozess die Fähigkeit erhalten, den Liefervorgang selbst anstoßen zu können. So kann ein Release-Manager dem Kunden die vereinbarte Version in seine Testumgebung liefern, damit dieser die Werkabnahme vollziehen kann. Die Entscheidung, das abgenommene Werk in das Produktionssystem auszurollen, trägt der Kunde dann selbst. Dieser Vorgang kann ihm durch die Bereitstellung eines speziell vorbereiteten Self-Service-Portal vereinfacht bzw., wenn entsprechend vereinbart, durch den Dienstleister nach vorhergehendem Auftrag durchgeführt werden.

3.3.3. Skripte für Build und Deployment

Eine Grundtechnik der Deployment-Pipeline stellen Skripten und Konfigurationsdateien dar. Je nach verwendetem Betriebssystem, auf dem die Deployment-Pipeline ausgeführt wird, können kleinere Programme in einer vom Betriebssystem unterstützten Skriptsprache verfasst werden, um die anfallenden Aufgaben bei der Systemkonfiguration und dem Deployment auszuführen. Mit Skripten können Programme und Informationen zur Steuerung und Organisation von Kompilier- und Verteilungsvorgängen verfasst werden. Beim Kompilieren von Quellcode stehen z. B. im Bereich der Programmiersprache Java die Build-Tools Ant und Maven zur Verfügung, deren Anweisungen, bei Ant, in der `build.xml` bzw. deren Konfiguration, bei Maven, in der `pom.xml` festgehalten werden. Zu den weiteren Aufgaben von Konfigurations- und Verteilungskripten gehören das Anpassen der Datenbank an neue Erfordernisse, das Deployen der neuen Softwareversion sowie das Installieren und Konfigurieren benötigter Middleware, Diensten und Komponenten. Das Verfassen von Skripten sollte dabei aber eine Aufgabe sein, die von Entwicklern und IT-Betrieb in gemeinsamer Verantwortung durchgeführt wird.[21]

Humble und Farley sehen hier drei Möglichkeiten, um die Konfigurations- und Verteilungsscripte auf dem Zielsystem automatisiert auszuführen:

- Ein Script, welches auf dem System der Deployment-Pipeline aufgerufen wird, sich mit der gewünschten Plattform verbindet und die entsprechenden Kommandos zur Änderung absetzt. Bei Unix-Betriebssystemen kann hier z. B. das SSH-Protokoll genutzt werden, um Befehle über eine gesicherte Verbindung auf einem Remote-System ausführen zu können.

[21] Vgl. [HF11, S. 147-150, 161]

- Ein Konfigurationsskript wird auf einem Verwaltungsserver bereit gestellt. Auf der Zielplattform läuft ein Agent, der sich mit dem Server verbindet und das Konfigurationsskript lädt und ausführt. Ein derartiges Konzept setzt das Werkzeug Chef[22] von Opscode um. Chef kann auf diese Weise einen ganzen Infrastruktur-Cluster verwalten und eine Änderung an der Systemkonfiguration durchführen und benötigte Software installieren.

- Durch das Ausnutzen des Paketsystems der jeweiligen Zielplattform kann z. B.für Debian Paketsystem *dpkg* ein Paket erstellt werden, welches die auszuliefernden Binaries enthält sowie ein *control file* der Informationen über die Pakete enthält, von denen die zu installierende Software abhängt sowie in Konflikte steht. Die Auslösung von Abhängigkeiten übernimmt dann das Debian Package-Tool.[23]

3.4. Kritische Komponenten im Liefersystem

3.4.1. Datenbanken in der Deployment-Pipeline

Datenbanken im Liefersystem benötigen eine besondere Aufmerksamkeit während des Softwarerelease. Dies gilt, sofern die Struktur der verwendeten Datenbank durch veränderte Anforderungen an das Gesamtsystem angepasst werden muss. Kann zu Beginn einer Entwicklung davon ausgegangen werden, dass es keine Änderungen am Datenbankschema geben wird, weil ein System entwickelt wird, welches sich an der Struktur einer bestehenden Datenbank ausrichten soll, bedarf dieses Thema keiner weiteren Beachtung.

Anders verhält es sich, wenn die Datenbank ausschließlich von einer Anwendung genutzt wird und das Schema der Datenbank den neuen Anforderungen und Funktionalitäten angepasst werden muss. So könnte eine neue Funktionalität in einer Webanwendung z. B. das Anlegen einer neuen Tabelle in der Datenbank erfordern oder die Anpassung von bestehenden Datenbankeinträgen. Der Verlust von Daten kann dabei aber nicht hingenommen werden. Der Aktualisierungsvorgang erfordert deshalb die gleiche Aufmerksamkeit, wie die Aktualisierung der Anwendung selbst.

Wird eine Anwendung inkrementell aktualisiert, z. B. Server eines Clusters nacheinander, kann es dabei zu Problemen kommen, wenn in den Clustern unterschiedliche Datenbankversionen von den laufenden Anwendungen genutzt werden. Transaktionen mit der alten Datenbank, die während des Aktualisierungsvorgangs von der alten Softwareversion durchgeführt werden, müssen nach Abschluss des Aktualisierungsvorgangs

[22] Mehr Informationen zu Chef unter: `http://www.opscode.com/chef/`
[23] Vgl. [Bro03]

auch in die neue Datenbank übertragen werden. Ein Datenverlust muss vom Release vermieden werden.

Schwierig kann es aber auch sein, ein Rollback auf die alte Version durchzuführen. Transaktionen, die mit der neuen Softwareversion auf die aktualisierte Datenbank durchgeführt wurden, müssen bei einem Rollback in die alte Datenbankversion übertragen werden. Hier stellt sich die Frage, was mit Daten geschehen soll, die nicht mit dem alten Datenbankschema kompatibel sind.

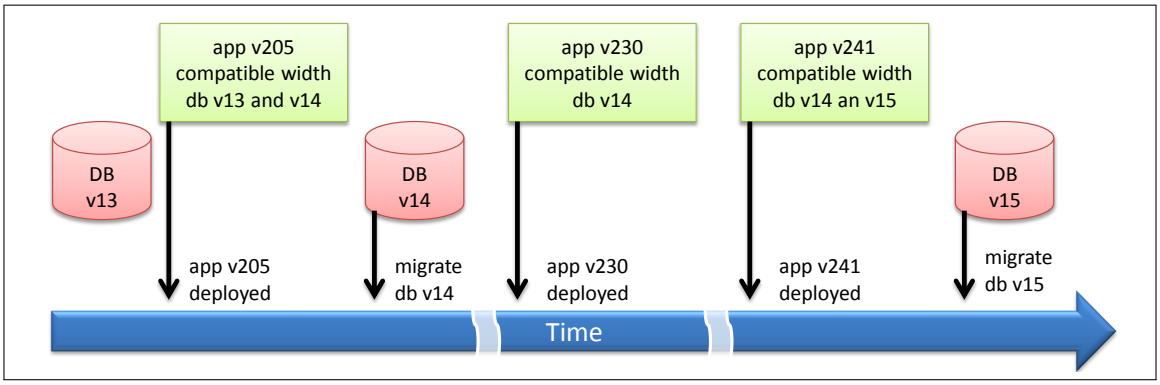
Humble und Farley schlagen hier folgendes Vorgehen vor:[24]

Inkrementelle Änderungen: Zu Beginn der Entwicklung wird eine neue leere Datenbank aufgesetzt. Werden Tabellen benötigt oder müssen angepasst werden, können die benötigten Anweisungen durch ein Skript ausgeführt werden. Das Skript enthält auch Anweisungen für das Rollback der Änderungen. Im Skript selbst sind Informationen enthalten, von welcher Datenbankversion auf welche aktualisiert wird. Die aktuelle Version der Datenbank kann in der Datenbank selbst gespeichert werden. Die Deployment-Pipeline kann so programmatisch prüfen, ob die vorgesehenen Änderungen auch ausgeführt werden dürfen. Im Zweifelsfall bricht der Vorgang ab. Bevor die Deployment-Pipeline Änderungen durchführt, müssen alle Bedingungen geprüft und erfüllt sein.

Datenbank-Rollback: Geht ein Deployment schief, muss die alte Datenbank wieder hergestellt werden können, ohne in der Zwischenzeit durchgeführter Transaktionen verlustig zu werden. Dafür muss vor einem Deployment ein Back-up der Datenbankdateien erstellt und auf einem zweiten Datenbankserver bereitgestellt werden. Anschließend können die Änderungen an an dem Datenbankserver vorgenommen werden, auf den die Anwendung nicht zugreift. Die korrekte Funktionsweise der Datenbank ist anschließend zu testen. Die Anwendung kann nun, z. B. in einem Cluster, stückweise herunter gefahren, aktualisiert und wieder hochgefahren werden. Die neue Version der Software greift auf die neue Datenbank zu, die alte Software auf die alte Datenbank. Um den Vorgang abzuschließen, wird am Ende des Vorgangs ein Delta der alten Datenbank zwischen Update-Beginn und Update-Ende in die neue Datenbank übertragen.

Entkopplung von Anwendungs- und Datenbankupdate: Die Entkopplung von Datenbank- und Anwendungsupdate erfordert ein mehrstufiges Vorgehen. Soll von einer Version auf die andere aktualisiert werden und die neuere Version benötigt z. B. eine neue Tabelle, muss dies vor dem Anwendungsupdate durchgeführt werden. Die

[24] Vgl. [HF11, S. 328-333]



Quelle: nach [HF11, S. 333] .

Abbildung 3.2.: Datenbank-Migration in der Deployment-Pipeline.

neue als auch die alte Anwendungsversion müssen dann mit der geänderten Datenbank kompatibel sein. Werden Tabellen nach einem Anwendungsupdate nicht mehr genutzt und sollen entfernt werden, kann dies erst nach dem erfolgreichen Anwendungsupdate erfolgen. In Abbildung 3.2 wird dieses Vorgehen visualisiert. Sind größere Änderungen des Datenbank-Schemas notwendig, um z. B. ein neues Feature zu realisieren, müssen die Änderungen am Quellcode die Einschränkungen durch die Datenbank berücksichtigen und in kleine Schritte zerlegt werden. Die Auswirkungen auf die aktive Datenbank können dann so gering wie möglich gehalten werden.

Die besonderen Erfordernisse von Datenbanken müssen bei der inkrementellen Anwendungsaktualisierung berücksichtigt werden. Ein Release sollte dabei so gestaltet werden, dass nur marginale Änderungen an einer produktiv eingesetzten Datenbank durchgeführt werden müssen. Die Integrität der Daten darf nicht gefährdet werden. Sind größere Änderungen an der Datenbank notwendig, um eine neue Funktionalität zu ermöglichen, ist es eventuell möglich, aus einem großen Release mehrere kleine Schritte zu machen Die Auswirkung auf die Datenbank kann so reduziert werden.

3.4.2. Strategien für die Versionsverwaltung

Die Funktionalitäten des Versionsverwaltungssystems, *Branch* und *Merge*, nehmen auf die Funktionsfähigkeit einer Deployment-Pipeline Einfluss. Mit einem Branch wird ein paralleler Entwicklungszweig begonnen. Dies ist z. B. notwendig um eine parallele Entwicklung eines Features zu ermöglichen, ohne den Hauptast oder andere Entwicklungen negativ zu beeinflussen, bis das Feature umgesetzt ist.[25]

[25] Vgl. [Pop06, S.154-159]

Eine Folge dessen ist ein erhöhter Integrationsaufwand beim *Merge*, bei dem ein Zweig in den Hauptstamm integriert wird. Je größer diese Differenzen, desto größer der Aufwand des Zusammenführens. Humble und Farley empfehlen, einen Zweig nach ein bis maximal zwei Tagen wieder in den Hauptstamm zu integrieren. Zudem wird der Zweig nicht mehr von der Deployment-Pipeline erfasst. Ein Feedback über den Zustand des Quellcodes ist für den Zweig nicht möglich.[26]

3.5. Implikationen für das Liefersystem

Aus den Konzepten von Continuous-Delivery und Vorschlägen für die Umsetzung einer Deployment-Pipeline ergeben sich folgende Implikationen für ein Liefersystem:

- Das Liefersystem besteht aus Infrastruktur und Systemumgebungen. Es hat Auswirkung auf das Entwicklungsteam und auf den IT-Betrieb. Beide Gruppen müssen eng zusammenarbeiten, um eine automatisierte, Deployment-Pipeline realisieren zu können. DevOps schlägt hierfür gemischte Entwicklungsteams vor, die Kompetenzen aus beiden Bereichen besetzen können.

- Das Liefersystem gibt allen Beteiligten ein Feedback. Hierzu gehören der aktuelle und der historische Zustand der Quellcode-Basis, welche Version in welcher Umgebung ausgeführt wird sowie aktuelle Vorgänge in der Deployment-Pipeline.

- Das Liefersystem wird in Phasen unterteilt. Die Qualitätssicherung wird weitgehend automatisiert. Das Scheitern einer Phase stoppt die Instanz der Deployment-Pipeline und damit alle nachfolgenden Phasen.

- Das Liefersystem baut auf Continuous-Integration auf und erweitert das Konzept um ein automatisiertes Deployment.

- Das Liefersystem muss sich um die Konfiguration der Infrastruktur kümmern und diese überwachen. Hierfür sind Skripte zu erstellen, welche die notwendigen Änderungen an der Ausführungsumgebung vornehmen können.

- Das Liefersystem stellt eine Oberfläche bereit, welche das Deployment einer Version in eine Testumgebung oder in die Produktionsumgebung auf eine einfache Weise anstoßen kann. Anstoßen bedeutet, das Deployment-Skript auszuführen. Als Methode genügt eine schlichte Oberfläche mit einem einfachen Push-Button dessen Betätigung zur Ausführung des Skripts führt.

[26] Vgl. [HF11, S. 390-393]

- Das Liefersystem muss für Komponenten wie Datenbanken ein geeignetes Konzept bereithalten, um Änderungen und Rollback ohne Verlust von Daten verwirklichen zu können. Zweige in der Versionsverwaltung umgehen das Liefersystem und sollten vermieden werden.
- Das Liefersystem muss stetig weiter entwickelt werden und sich den Anforderungen anpassen können.

Continuous-Delivery und die Deployment-Pipeline benötigen spezielle Werkzeuge. Dabei ist es für die Ausführung unerheblich, ob es ein einzelnes Werkzeug gibt, mit dem alle Anforderungen erfüllt werden können oder eine Verknüpfung von verschiedenen Werkzeugen und Scripten zu einer *Tool chain* das Gleiche Ergebnis liefert.

4. Entwicklungsstand von Continuous-Delivery bei adesso

Dieser Abschnitt geht auf die derzeitige Situation der Auslieferung von Software bei adesso ein. Hierzu gehören der Auslieferungsprozess und die Infrastruktur, die für die Auslieferung bereitsteht. Dies ist notwendig, um Ansatzpunkte für die noch zu untersuchenden Werkzeuge zu finden.

adesso als unabhängiger IT-Dienstleister berät und entwickelt individuelle Software für Kunden aus den Branchen Versicherungen und Rückversicherungen, Banken und Finanzmarkt, Gesundheitswesen, Lotterie, öffentliche Verwaltung, Telekommunikation und Medien.[1] Jeder Kunde hat dabei bestimmte Anforderungen und Rahmenbedingungen, die er an die Projekte knüpft. Mit einer heterogenen Projektlandschaft hinsichtlich der eingesetzten Verfahren, Technologien oder Prozesse wird nicht zu rechnen sein. Zudem wird angenommen, dass es bei adesso schon parallel Bemühung gibt, den Auslieferungsprozess zu verbessern bzw. Continuous-Delivery zu ermöglichen.

4.1. Ausgangssituation für die Untersuchung des Entwicklungsstandes

Technologisch unterscheidet adesso Projekte selbst nach den Kategorien Java, Microsoft, Mainframe sowie Enterprise Content Management.[2] Bei der weiteren Betrachtung des aktuellen Auslieferungsprozesses sowie von Continuous-Delivery sollen vornehmlich Projekte aus dem Java bzw. JEE-Umfeld und Web-Umfeld betrachtet werden. Das ist der Bereich, bei dem im Unternehmen eine hohe Aktivität und ein besonderes Interesse an neuen Ansätzen wie Continuous-Delivery, Behavior Driven Develpment (kurz BDD), Big-Data und Cloud-Computing herrscht.[3]

Auf technischer Ebene lässt sich ein kontinuierliches Ausliefern von Software mit verschiedenen Plattformen und Softwarearchitekturen verwirklichen, im Zentrum der wei-

[1] Vgl. [ade12c]
[2] Vgl. [ade12g]
[3] Vgl. [ade12f]

teren Betrachtung stehen hier aber Web-Anwendungen. Häufig werden WAR und EAR-Files für Web- und Applikations-Container nach der JEE-Spezifikation entwickelt.[4]

Um Projektrisiken durch späte Integration einzelner Softwarekomponenten zu reduzieren, stellt adesso ein System für Continuous-Integration, kurz CI-System genant, bereit. Dort können Projektteams den aktuellen Stand ihrer Software automatisiert kompilieren und die Komponenten testen lassen. Besonders bei modularen Softwaresystemen, ist die Integration von mehreren Softwareteilen eine mit hohen Aufwänden verbundene Angelegenheit.

Damit die Konfiguration des CI-Systems für ein neues Projekt reibungslos ablaufen kann, stellt adesso den Projektteams ein Support-Team bereit. Dieses ist mit Spezialisten rund um das Thema Continuous-Integration und Build-Management besetzt. Hauptgrund, ein solches Support-Team zu unterhalten, ist die stetige Wiederholung von Aufgaben, die beim Einrichten eines neuen Projektes anfallen. Das adesso Build-Management stellt eine gemeinsame Integrations- und Test-Umgebung bereit. Zudem können sich Projektteams zu Technologien beraten lassen, die das Build-Management betreffen. Ziel einer Beratung ist die geeignete Wahl eines Build-Werkzeugs und die Erarbeitung von Build-Skripten. Vordergründig werden dabei die Werkzeuge Ant und Maven unterstützt.[5]

Wieweit der derzeitige Prozess ausgereift ist und die noch zu untersuchenden Werkzeuge ergänzt werden können, muss die Untersuchung des Istzustandes zeigen. Durch den Einsatz eines Reifegradmodells soll sich der derzeitige Entwicklungsstand abschätzen und beurteilen lassen. Ein Reifegradmodell für Continuous-Integration und Continuous-Delivery wurde von der Firma UrbanCode[6] veröffentlicht, die sich auf die Entwicklung von Werkzeugen zur Unterstützung von DevOps und Continuous-Delivery spezialisiert hat. Da sich UrbanCode auch auf Werkzeuge für Continuous-Delivery anbietet, dient das Modell auch zu einem Teil der Eigenwerbung. In seinem Ansatz kann es aber unabhängig genutzt werden, um den eigenen Zustand im Unternehmen zu beschreiben.

Ein Hauptanliegen von Continuous-Delivery ist die Ausbeutung von Gewinnchancen, die durch die kurzen Iterationen agiler Vorgehensmodelle entstehen. Wenn neu entwickelte Programmfunktionalitäten unverzüglich am Markt angeboten oder in Unternehmensanwendungen eingesetzt werden können, ergeben sich hieraus strategische Vorteile für das Unternehmen gegenüber anderen Wettbewerbersteilnehmern. Damit adesso seinen Kunden einen solchen Vorteil anbieten kann, müsste der Auslieferungsprozess auf diese Zielstellung angepasst sein.

---

4 Weitere Informationen zur JEE-Sepzifikation: Vgl. [Sun09]

5 Vgl. [ade12d]

6 Mehr Informationen zu UrbanCode: `http://www.urbancode.com/`

4.2. Erhebungsmethode zum Entwicklungsstand

Neben der Recherche der unternehmensinternen Quellen besteht durch das CI-Team die Gelegenheit, einzelne Mitarbeiter zur aktuellen Praxis bei der Auslieferung von Software zu befragen. Die Befragung des CI-Teams kann als Einzelinterview oder in Form eines Fragebogens erfolgen. Ein Einzelinterview bietet dabei die Möglichkeit einer individuell geführten Befragung. Im Gegensatz dazu legt ein Fragebogen einen gleichbleibenden Befragungsrahmen fest und kann zudem den Zeitaufwand der Befragung für den Fragesteller sowie für den Befragten auf ein Minimum reduzieren.

Um den Zeitaufwand so gering wie möglich zu halten, wird auf den Fragebogen als Erhebungsinstrument zurückgegriffen. Der Fragebogen kann dabei so gestaltet werden, dass sich der Befragungsaufwand für die Befragten auf einen Zeitaufwand von 10 bis 15 Minuten reduzieren lässt. Der Aufwand, die formulierten Fragen über ein Telefoninterview zu vermitteln und die Antworten zu notieren, wird auf 45 bis 60 Minuten geschätzt.

Zudem kann der Fragebogen zu einer beliebigen Zeit beantwortet werden und so die individuelle Koordination eines Gesprächstermins mit den Mitgliedern des CI-Teams umgangen werden. Der Fragebogen selbst dient keiner statistischen Auswertung. Vielmehr sollen mit ihm die Meinungen und Standpunkte der Befragten eingesammelt werden.

Bei der Gestaltung des Fragebogens werden nur zu einem geringen Teil freie Textfelder verwendet. Mehrheitlich ist der Fragebogen so konzipiert, dass der Befragte bestimmten Aussagen zustimmen oder diese ablehnen kann. Zu jeder Frage gibt es weiterhin die Möglichkeit, freie Anmerkungen zu geben. Hierdurch können auch Ideen zu bestimmten Themen durch die Befragten geäußert werden, wie es auch in einem Telefoninterview möglich wäre.

Als Grundlage des Fragebogens werden das *Enterprise Continuous Integration Maturity Model* von UrbanCode[7] sowie die von Humble und Farley entwickelten Konzepte zu Continuous-Delivery[8] genutzt. Das Maturity-Model für Continuous-Delivery wurde für eine Selbstevaluierung von Unternehmen entwickelt. Dabei werden Zielvorgaben definiert, an denen sich die Reifegrade bestimmen lassen. Untergliedert ist das Modell in die Einheiten Kompilieren, Ausliefern und Testen einer Anwendung sowie die Berichtserstellung und Visualisierung des Prozesses.

Der Fragebogen ist Unterteilt nach Commit-Stage, Acceptance-Stageund *Deployment*. Dieser orientiert sich damit an der von Humble und Farley vorgeschlagenen Struktur der Deployment-Pipeline.[9] Zusätzlich soll der Befragte zu Beginn die Phasen des Auslieferungsprozesses selbst skizzieren. Dies soll das Selbstverständnis über die Phasen der

[7] Vgl. [MF11]
[8] Vgl. [HF11]
[9] Vgl. [HF11, S. 111]

Commit-Stage zeigen.

Im Bereich der Commit-Stage wird der Reifegrad des Build-Prozesses abgebildet und es werden Fragen zum Konfigurationsmanagement sowie zum Umgang mit Artefakten gestellt. Unter die Acceptance-Stage fallen Fragen zu Verfahren und Werkzeugen, die beim Testen von Quellcode verwendet werden. Die letzte und kritische Phase der Deployment-Pipeline ist das Ausrollen der Anwendung in die Produktivumgebung. Hierzu werden Fragen im Bereich des *Deployments* gestellt. Es wird versucht zu ermittelt, welche Bemühungen es bei adesso derzeit gibt, um eine Deployment-Pipeline aufzubauen.

4.3. Auswertung und Darstellung des Entwicklungsstandes

Der Fragebogen wurde von drei Mitarbeitern beantwortet, welche den repräsentativen Kern des CI-Teams darstellen. Die konkreten Antworten können dem Anhang 1 entnommen werden. In diesem Abschnitt werden die gegebenen Antworten ausschließlich zusammengefasst und im Zusammenhang mit Continuous-Delivery betrachtet.

Da sich nicht alle gegebenen Antworten decken, wird hieraus geschlussfolgert, dass es noch kein durchgehendes standardisiertes Vorgehen für den Auslieferungsprozess gibt und es vom jeweiligen Projekt abhängig ist, welche Phasen und Teilprozesse von Continuous-Integration und Continuous-Delivery unterschieden und realisiert werden.

4.3.1. Phasen des derzeitigen Entwicklungs- und Auslieferungsprozesses

Aus dem Fragebogen gehen folgende Phasen des Entwicklungs- und Auslieferungsprozesses hervor:

- Aufsetzen der Projektumgebung und Einrichten der Entwicklungsumgebung,
- Entwicklung der Software,
- Durchführen von Integrationstests und
- Auslieferung der Software.

Die Entwicklung unterteilt sich in die Erstellung des Quellcodes und die anschließenden Phasen von Kompilierung, Tests und Analyse. Dieser Prozess wird durch die Änderung der Quellcode-Basis in der Versionsverwaltung als automatisierter Vorgang angestoßen. Abbildung 4.1 zeigt diesen Prozess. Optional ist ein automatisiertes Deployment in eine

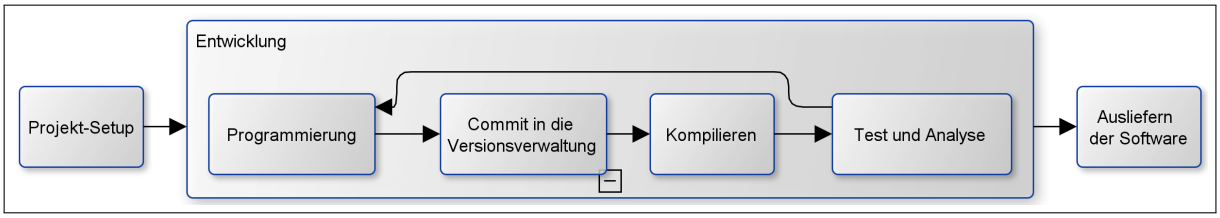


Abbildung 4.1.: Lieferprozess bei adesso.

Testumgebung bzw. auf ein Staging-System. In dieser Umgebung können weitere Tests wie z. B. Akzeptanz- oder Lasttests durchgeführt werden. Sind alle Phasen erfolgreich verlaufen, kann das Release durchgeführt werden und die Software in die Produktivumgebung installiert und ausgeführt werden.

Entwickelt wird zunehmend unter dem TDD-Ansatz, dem Test-Driven-Development. TDD setzt die Erstellung von Test-Fällen vor die Implementierung der Funktionalität. Ziel dieses Vorgehens ist die Fokussierung des Entwicklungsprozesses auf die noch umzusetzenden Funktionalitäten. Deren erwartetes Ergebnis wird zu Beginn der Implementierung einer Funktionalität festgelegt.[10]

Da es oftmals keinen direkten Zugriff auf die Produktivumgebung des Kunden gibt, kann z. B. im JEE-Umfeld nur ein fertiges WAR- oder EAR-File ausgehändigt werden. Die Anwendung wird mit einer Installationsanleitung übergeben. Mitarbeiter des Kunden bzw. ein weiterer Dienstleister in Form eines Rechenzentrums ist dann für die Installation und den reibungslosen Betrieb der Anwendung verantwortlich. Besteht der direkte Zugriff auf die Produktivumgebung, z. B. weil adesso selbst für den Kunden das System betreibt, sind automatisierte Auslieferungen in die Produktivumgebung möglich.

4.3.2. Commit-Stage

Kompilieren des Quellcodes und erstellen der Softwarepakete ist bei adesso ein standardisierter Prozess. Ob dieser auch ausgenutzt werden kann, hängt allerdings von den Rahmenbedingungen eines Projekte ab. Unterstützung gibt es durch den Jenkins CI-Server. Dieser übernimmt die Aufgaben des Build-Prozesses, der Durchführung der Unit-Test sowie der Integrationstest. Dabei wird auf die Build-Tools wie Maven- oder Ant zurückgegriffen, die das Kompilieren des Quellcodes organisieren können. Projektteams, die das System nutzen wollen, sind angehalten, die Vorgänge und Prozesse, die bei der Erstellung der Softwarepakete notwendig sind, in Skripte zu verfassen, die für eine automatisierte Ausführung notwendig sind.

In Abbildung 4.2 wird die Übersicht über alle vom Jenkins kontrollierten Projekte sowie die Detailansicht auf Projektebene gezeigt. Die Statusseite des Jenkins ist für alle

[10] Vgl. [Amb]

Abbildung 4.2.: Jenkins CI-Seve: Übersicht über alle Build-Porzesse.

Mitarbeiter bei adesso zugänglich. Über den konkreten Zustand eines Projektes können sich so alle Beteiligten leicht informieren. Die verwendeten Symbole sind leicht zu interpretieren. In der Detailansicht kann die Historie des Build-Prozesses über alle im Build-Prozess genutzten Revisionen hinweg eingesehen werden. Das rechte Beispiel zeigt die grafische Auswertung der Testtrends. Die letzten drei Änderungen an der Quellcodebasis waren alles Fehlschläge. Obwohl die Identifizierung der Fehlerursache durch dieses System vereinfacht wird, trifft dies in diesem Fall nicht auf die Beseitigung des Fehlverhaltens zu. Der Prozess ermöglicht hier jedoch ein schnelles Feedback an das Entwicklungsteam und alle weiteren Beteiligten.

Dieser Prozess ist zentral und kann jederzeit repliziert werden. Hierdurch können Probleme ausgeschlossen werden, die entstehen, wenn ein derartig gestalteter Prozess auf einem Entwicklersystem durchgeführt wird. Die Umgebung, in der ein Entwickler seinen Quellcode erstellt, kann nicht als stabil und verlässlich angesehen werden. Ein solches System unterliegt zu hohen Seiteneinflüssen durch andere Programme und Systeme, als dass ein verlässlicher und wiederholbarer Prozess möglich wäre. Die Verlässlichkeit sowie die Kontrolle über mögliche Seiteneinflüsse wird jedoch für den Build-Prozess vorausgesetzt. Standards, auf die sich verständigt wurde, können so möglicherweise nicht eingehalten werden. Scheitert der Prozess, lassen sich aufgetretene Fehler so nicht vollständig replizieren und die Ursache nicht ohne einen Restzweifel ergründen. Das vorhandene CI-System ist ein wesentlicher Baustein, solche Fehler zu vermeiden.[11]

Alle erstellten Softwarepakete werden mit einer Versionsnummer versehen und in einem zentralen Repository abgelegt. Nachgeschaltete Tests und Prozesse können so jederzeit auf die unterschiedlichen Versionsstände einer Software zurückgreifen. Zu keinem Zeitpunkt wird eine einmal erstellte Version ein zweites Mal kompiliert. Dies ist ein wichtiger Punkt, um auftretende Fehler im Prozess jederzeit replizieren zu können. Würde ein Paket innerhalb des Prozesses ein zweites Mal erstellt werden, lassen sich die Bedingungen, unter denen die ursprüngliche Version erstellt wurde, möglicherweise nicht mehr herstellen. Der Grund für das Fehlverhalten, z. B. bei einem Testdurchlauf, ist

[11] Vgl. [MF11]

dann nur mit höherem Aufwand zu ermitteln. Es kann nicht ausgeschlossen werden, dass der Build-Prozess durch eine veränderte Konfiguration selbst den Fehler verursacht hat. Wird ein Paket hingegen nur einmal erstellt und kommt es in einem späteren Test zu einem Fehlverhalten, lässt sich die Fehlerursache auf die letzte Änderung der Quellcodebasis zurückführen. Eine Untersuchung sollte dann den Grund des Fehlverhaltens schnell eingrenzen können.

Das Management von Abhängigkeiten, in Bezug auf 3rd-Party-Bibliotheken, wird in Verbindung mit Maven durch den Nexus Repositroy-Sever unterstützt. Der Nexus-Server stellt ein zentrales Repository für alle benötigten 3rd-Party-Bibliotheken bereit, die in entwickelter Software zum Einsatz kommen. Zudem lassen sich auch die selbst entwickelten Komponenten für beliebige andere Prozesse bereitstellen.[12]

Alle Prozesse im Jenkins CI-Server werden automatisch ausgelöst. Übermittelt ein Entwickler seine Änderungen an den bei adesso genutzten Subversion-Server, startet der Jenkins-Server den Build-Prozess. Der Jenkins-Server vergleicht dann die aktuelle Revisionsnummer der Versionsverwaltung mit der zuletzt bekannten und für das letzte Build genutzten Revisionsnummer.

Der Jenkins CI-Server ist ein wichtiger Baustein für die Realisierung Continuous-Integration. Er steht jedem Projektteam bereit, welches Technologien verwendet, die durch den Jenkins abgedeckt werden können. Ob dieses Angebot angenommen werden kann, ist von den Bedingungen abhängig, die der Auftraggeber bestimmt. Damit obliegt die genaue Ausgestaltung der Commit-Stage den einzelnen Projektteams in Abstimmung mit den speziellen Projektvorgaben.

4.3.3. Qualitätssicherung und Acceptance-Stage

Innerhalb der Acceptance-Stage sind die Verfahren zusammengefasst, die sicherstellen, dass eine spezifische Softwareversion den funktionalen sowie nicht-funktionalen Anforderungen genügt. Da sich diese Phase durch den Fragebogen nicht vollständig von den Komponenten- und Integrationstests trennen lässt, werden hier alle Elemente der Qualitätssicherung berücksichtigt.

In adesso-Projekten wird generell eine hohe Testabdeckung angestrebt, ein Grad von 100% jedoch nicht vorausgesetzt. Automatisierte Tests werden zum Teil durch manuelle ergänzt, um die Anwendung hinsichtlich bestimmter Risikoszenarien untersuchen zu können.

Zur statischen Analyse des Quellcodes wird Sonar, eine Plattform um Code-Qualität zu managen, eingesetzt. Sonar bietet die Möglichkeit, Quellcode nach so genanten Coding-

[12] Vgl. [Son12b]

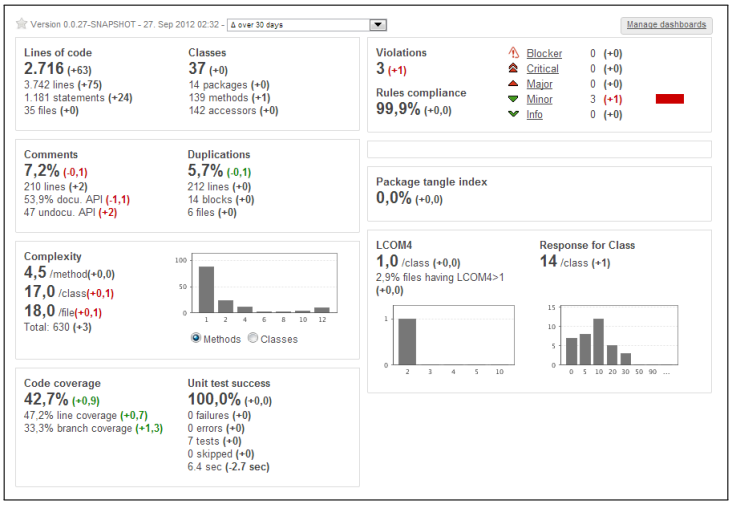


Abbildung 4.3.: Detailansicht eines Projektes im Sonar-Server.

Rules zu analysieren. Hierzu gehören Namenskonventionen, bekannte Anti-Pattern und Metriken wie die Testabdeckung oder die zyklomatische Komplexität von Paketen.[13]

Abbildung 4.3 zeigt die Detailansicht der statischen Code-Analyse eines Projektes. Relevant ist hier die Einhaltung der vereinbarten Regeln, die dieses Projekt mit $99,9\%$ erfüllt. Zudem können Veränderungen zu vorhergehenden Analysen angezeigt werden.

Weitere Werkzeuge zur statischen Code-Analyse sind:

- FindBugs[14],
- PMD[15],
- JaCoCo[16] und
- Cobertura[17].

Neben den Komponenten-Tests und der statischen Analyse des Quellcodes werden breitere Testverfahren auf das Gesamtsystem ausgeführt. Ziel dieser Phase ist, das vollständig integrierte System auf die Erfüllung von funktionalen und nicht-funktionalen Anforderungen zu prüfen.

Automatisierte Akzeptanztests stellen eine Möglichkeit dar, dem Entwicklungsteam eine schnelle Rückmeldung zu geben, welche Anforderungen vom aktuellen System erfüllt werden. Auch für adesso-Projekte sind diese deshalb eine wichtige Grundlage im Entwicklungsprozess und heben den Vorteil des CI-Systems hervor.[18]

13 Vgl. [Son12a]
14 Weitere Informationen zu FindBugs unter: `http://findbugs.sourceforge.net/`
15 Weitere Informationen zu PMD unter:`http://pmd.sourceforge.net/`
16 Weitere Informationen zu JaCoCo unter: `http://www.eclemma.org/jacoco/trunk/index.html`
17 Weitere Informationen zu Cobertura unter:`http://cobertura.sourceforge.net/`
18 Vgl. [HF11, S. 86-87] und [Duv07, S. 15]

Werden Änderungen der Quellcode-Basis an das Subversion-System übermittelt, wird die komplette Anwendung kompiliert und die damit verbundenen Tests ausgeführt. Dass jede Änderung zum Durchlauf aller Teststufen führt, sichert die Qualität zusätzlich durch Regressionstests. Wirkt sich ein neues Feature negativ auf die Gesamtqualität der Anwendung aus, wird dies durch die fortlaufende Qualitätssicherung entdeckt.[19]

4.3.4. Deployment

Mit dem Deplyoment wird die Software in eine bestimmte Umgebung ausgeliefert. Dabei kann die Zielumgebung des Deployments eine für Testzwecke bestimmte Umgebung oder die Produktivumgebung sein. Sind Commit-Stage und Acceptance-Stagerichtig angelegt, wird nach jeder Änderung der Quellcodebasis eine neue Instanz des Softwarelieferprozesses erzeugt, an deren Ende eine reife Software für den Produktivbetrieb steht. Um die erstellte und geprüfte Software auch in einem automatisierten Prozess in Produktion zu bringen, erfordert es einige Vorbereitung und kann durch weitere Werkzeuge wie Chef unterstützt werden. Für den bei adesso verwendeten Jenkins CI-Server gibt es die Möglichkeit, ein Deployment auf eine laufende Serverinstanz durchzuführen. So kann der Jenkins Deployment-Task z. B. eine laufende Tomcat- oder JBoss-Instanz ansprechen und mittels Ant-Tasks oder Maven-Goals ein War-File in diese deployen.

Ein komplexeres Deployment, wie z.B. die Auslieferung in eine Infrastruktur, die gegebenenfalls erst installiert und gestartet werden muss, ist nur über eigene Skripte möglich. Hierfür lassen sich z. B. Shell- oder Ant-Skripte hinterlegen, die derartige Aufgaben und Werkzeuge anstoßen. Zum Teil ist es aber auch üblich, die benötigte Systemumgebung manuell zu konfigurieren und zu installieren. In einigen Projekten konnten auch schon automatische Verfahren in Zusammenhang mit Cloud-Infrastrukturen eingerichtet werden. Das hierbei eingesetzt Werkzeug Chef[20] ermöglicht die Anpassung der Softwarekonfiguration von entfernten Systemen.

4.4. Anknüpfungspunkte für Continuous-Delivery

Der Prozess, die Anwendung zu kompilieren und zu testen, sollte für alle Projekte, bei denen die Bedingungen bestimmt bzw. angeboten werden können, standardisiert und verpflichtend sein. Manuelle, nicht standardisierte Prozesse, bieten den Entwicklern zwar Flexibilität und Individualität, erhöhen aber das Risiko, Fehler und Schwachstellen beim Zusammenspiel der unterschiedlichen Softwareteile erst viel zu spät zu erkennen. Möglicherweise ist dieser Moment erst kurz vor der geplanten Abnahme der Software durch

[19] Vgl. [HF11, S. 87]

[20] Weitere Informationen zu Opscode Chef unter `http://www.opscode.com/chef/`

den Kunden. Dies kann dann zu einer Reihe kurzfristig und nicht ausreichend geprüfter Änderungen an der Anwendung führen. Ein standardisierter Prozess, der bei jeder Änderung auf die gleiche Art und Weise ausgeführt wird, hilft ein Fehlverhalten der Anwendung einzugrenzen.[21]

In Projekten, bei denen adesso als Lieferant von War- bzw. Ear-Files auftritt, ist ein direktes Deployment in die Produktivumgebung, z.B. durch Sicherheitsrichtlinien des Rechenzentrumsbetreibers, nicht möglich. Eine voll automatisierte Deployment-Pipeline wäre in diesen Fällen nicht möglich. Für die Sicherstellung der Qualität sollte der interne CI-Prozess verwendet werden und die Software nach jeder Änderung in eine an das Produktivsystem angelehnte Umgebung installiert und getestet werden.

Projekte mit einem agilen Vorgehen setzen in jeder Iteration neue Funktionalitäten um. Auf der Seite des Kunden kann es wünschenswert sein, ein dringend benötigtes Feature schnellstmöglich produktiv einsetzen zu können. Gerade bei Softwaresystemen, bei denen sich auf die Wartung des Systems oder eine kontinuierliche Weiterentwicklung verständigt wurde, kann das automatisierte Deployment einer gut getesteten Software die Kundenzufriedenheit erhöhen und den Prozess in Summe verbessern.

Das Einrichten von Infrastrukturen und Systemen sollte dabei möglichst nach standardisierten Prozessen ablaufen. Die Schritte für Projekte mit einem gleichen Technologie-Stack ähneln sich. Das CI-Team deckt mit seinem Angebot und dem Jenkins CI-Server schon einen großen Teil der notwendigen Prozesse für Continuous-Delivery ab. Hinsichtlich des Deployments könnte aber eine vorkonfigurierte *Tool-Chain* den Konfigurationsaufwand für neue Projekte deutlich reduzieren. adesso könnte so in die Fähigkeit versetzt werden, seinen Kunden eine Deployment-Pipeline bei bestimmten Projekten mit anbieten zu können. Die Untersuchung der drei Werkzeuge wird weiteren Aufschluss über die möglichen Ansatzpunkte für Continuous-Delivery bieten.

[21] Vgl. [Duv07, S. 29-33]

5. Werkzeuge für Continuous-Delivery

Als Ziel dieser Arbeit wurde die Evaluierung von drei Werkzeugen, die adesso für die Umsetzung von Continuous-Delivery als interessant eingestuft hat, gesetzt. Dabei möchte adesso vornehmlich herausfinden, wie diese Werkzeuge funktionieren und aufgebaut sind und wie sich damit eine Deployment-Pipeline umsetzen lässt. Vor einer genaueren Untersuchung der Eigenschaften ist eine Grobbetrachtung sowie eine Aufstellung von Beurteilungskriterien notwendig. Im Anschluss daran werden die Werkzeuge auf einer Testumgebung installiert und können dann genauer nach ihren Funktionalitäten untersucht werden.

An Continuous-Delivery und den Möglichkeiten einer Deployment-Pipeline besteht bei adesso ein hohes Interesse. Mit dem Jenkins-Server wird bereits ein System für Continuous-Integration verwendet und damit der erste Teil von Continuous-Delivery verwirklicht. Untersucht werden soll Go von Thoughtworks, Deployinator von Etsy und Dreadnot von Rackspace. Mit der Evaluierung von Go, Deployinator und Dreadnot möchte adesso herausfinden, ob ein Lückenschluss zwischen dem CI-System und der Verwaltung von Infrastrukturen mit Werkzeugen wie Chef möglich ist.

5.1. Kriterien für die Evaluierung

Für die Evaluierung der Werkzeuge ist kein spezielles Anwendungsszenario vorgegeben, wie z. B. für ein bestimmtes Projekt eine Deployment-Pipeline mit Hilfe eines dieser Werkzeuge umzusetzen. Aus diesem Grund werden die Fähigkeiten der Werkzeuge nach allgemeinen Kriterien betrachtet und der Rahmen berücksichtigt, in dem sich die adesso Projekte aus dem Umfeld der Web-Technologien bewegen.

adesso sieht in Continuous-Delivery, in Zusammenhang mit Cloud-Techniken, eine mögliche zukünftige Ausrichtung, auf deren Basis sich neuartige Dienstleistungen errichten lassen. Im Zentrum steht hier nicht die Auswahl einer geeigneten Software mit anschließender Einführung, sondern viel mehr eine Betrachtung der sich bietenden Optionen für zukünftige Projekte und Dienstleistungen.

Der genaue Funktionsumfang dieser Werkzeuge als auch die verfolgten Ansätze sind vorerst nicht weiter bekannt. Implikationen für ein Liefersystem ergeben sich aber aus Abschnitt 2 und 3 und wurden unter Punkt 3.5 zusammengefasst. Kriterien für eine Evaluierung können hieraus abgeleitet werden.

5.1.1. Einordnung der Werkzeuge

Eine einführende Grobbetrachtung der zu untersuchenden Werkzeuge zeigt, dass sich die Werkzeuge in unterschiedlichen Klassen befinden und nicht dieselben Ziele verfolgen.

Bei *Go* handelt es sich um ein vollwertiges CI-System in dem die Phasen der Deployment-Pipeline abgebildet werden können. *Dreadnot* und *Deployinator* sind vornehmlich Self-Service-Portale, die das Deployment anstoßen sollen.

Ein direkter Vergleich der Werkzeuge wird nicht möglich sein, da der Funktionsumfang zu unterschiedlich ist, um eine objektive Beurteilung zu ermöglichen. Für eine unabhängige Betrachtung der Werkzeuge spricht auch die Implikation, für neue Dienstleistungen und zukünftige Projekte unterschiedliche Ansätze ableiten zu können.

Die unabhängige Betrachtung schließt aber nicht einen Vergleich von allgemeinen Qualitätsmerkmalen von Software aus sowie die Untersuchung, welche Merkmale einer Deployment-Pipeline konkret abgebildet werden können. Für den derzeitigen Lieferprozess wäre es eine Verbesserung, wenn die automatisierte Auslieferung von Software in die Test- und Produktivumgebung sich über ein Portal besser organisieren lassen würde.

In einem Kunden-Lieferanten-Verhältnis ist ein direktes Liefern in die Produktivumgebung häufig aus vertragsrechtlichen Einschränkungen oder Gründen der Compliance nicht möglich. adesso als Dienstleister für individuelle Softwarelösungen könnte in diesem Fall aber ein Self-Service-Portal als zusätzliche Dienstleistung bereitstellen, in das sich Mitarbeiter des Kunden einloggen können. Die Testabteilung des Kunden oder der IT-Betrieb dann das Deployment der gewünschten Version anstoßen.

5.1.2. Allgemeine Kriterien

Die Qualität von Software lässt sich durch die definierte Qualitätscharakteristika der DIN/ISO-9126 beschreiben. Durch diese Kriterien lässt sich eine Software nach Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Portabilität beurteilen.[1] Diese Kriterien sollen auch für einen allgemeinen Vergleich von *Go*, *Deployinator* und *Dreadnot* genutzt werden, um eine vergleichende Aussage der Werkzeuge zu ermöglichen.

[1] Vgl. [Sta11, S. 57f] und [Bal11, S. 110]

Funktionalität Mit der Funktionalität wird das Vorhandensein von Funktionen einer Anwendung mit festgelegen Eigenschaften und die angemessene Erfüllung der Anforderungen beschrieben. In Bezug auf die Konzepte von Continuous-Delivery bedeutet dies die Möglichkeit, eine Deployment-Pipeline abbilden zu können und die Unterstützung einer weitgehenden Automatisierung der dazugehörigen Prozesse. Die Anforderungen leiten sich aus Continuous-Delivery ab.

Damit die Anwendung mit den vorgegebenen Systemen zusammenarbeiten kann, muss geprüft werden, wie weit die Anwendung sich in die bestehende Systemlandschaft integrieren lässt. Für adesso ist es erforderlich, dass die betrachtete Anwendung mit dem bestehenden CI-System zusammenarbeitet.

Zuverlässigkeit Mit der Zuverlässigkeit wird die Reife des Produktes zum Ausdruck gebracht. Dabei spielen folgende Faktoren eine Rolle:

- Eine geringe Versagenshäufigkeit,
- die Reaktion des Systems auf Fehlerzustände und
- die Möglichkeit den Zustand des Systems nach einem Versagen wiederherstellen zu können.

Eine Aussage über die Reife kann z. B. das Entwicklungsstadium der Anwendung sein. Mit einer Versionsnummer kleiner 1.0 wird signalisiert, dass es sich noch um keine für ein Release freigegebene Version der Software handelt. Zum Vergleich von Reife zwischen Anwendungen ist dies allerdings kein Kriterium.

Benutzbarkeit Die Benutzbarkeit einer Anwendung kann über ihre Verständlichkeit, Erlernbarkeit und Bedienbarkeit beurteilt werden. Bei einer verständlichen Anwendung muss ersichtlich sein, welche Funktionen welche Zustände und Prozesse anstoßen. Der Aufwand, das Konzept der Anwendung zu verstehen, muss gering sein. Zudem muss der Einstieg in die Anwendung leicht und schnell erlernbar sein. Konfiguration und Bedienung sollen leicht sein und in verständlichen Schritten erfolgen.

Effizienz Mit der Effizienz wird das Zeitverhalten einer Anwendung untersucht. Die Anwendung sollte schnell auf Nutzereingaben reagieren können und die Antwortzeit bei einer Funktionsausführung gering sein. Weiterhin wird hier das Verbrauchsverhalten in Form der benötigten Betriebsmittel untersucht, die das System für den Betrieb benötigt. Als Vergleichsmaßstab können die angegebenen Systemvoraussetzungen herangezogen werdend.

Wartbarkeit und Änderbarkeit Der Aufwand, Mängel der Anwendung sowie deren Ursache zu finden, sollte gering sein. Sind Anpassungen der Anwendung notwendig, stellt sich die Frage nach den Voraussetzungen, um eine Anpassung durchführen zu können sowie der damit verbundene Aufwand.

Portabilität und Übertragbarkeit Die Anpassbarkeit beschreibt die Möglichkeit, die Anwendung an eine andere Umgebung anzupassen. Demnach sollte das System auf unterschiedlichen Plattformen und Systemkonfigurationen ausgeführt werden können.

Zudem steht hier die Durchführung und der Aufwand von Installation und Inbetriebnahme der Anwendung auf dem Prüfstand. Um eine Anwendung in den Betrieb zu nehmen, muss die Installation gut dokumentiert, verständlich und vollständig sein.

5.1.3. Anforderungen für Continuous-Delivery

Aus Continuous-Delivery leiten sich Anforderungen ab, die von einer Deployment-Pipeline berücksichtigt werden müssen. Es wird nicht erwartet, dass es ein Werkzeug gibt, welches alle Anforderungen erfüllt. Vielmehr haben Humble und Farley bei der Beschreibung der Deployment-Pipeline auf den Einsatz von spezialisierten Werkzeugen und Shell-Skripten verwiesen.

Go, *Deployinator* und *Dreadnot* sollen gegen die Anforderungen von Continuous-Delivery geprüft werden, damit ersichtlich wird, welche Elemente der Deployment-Pipeline abgebildet werden können und für welche ein ergänzendes Werkzeug gefunden werden muss, um eine vollständige Deployment-Pipeline verwirklichen zu können.

Nach Humble und Farley besteht die Deployment-Pipeline aus einzelnen Stages, die parallel oder sequenziell verlaufen können. Für die Deployment-Pipeline muss es also Möglichkeiten geben, Prozesse abbilden zu können. Eine Oberfläche für das Management der Deployment-Pipeline sollte anzeigen, welche Version in welche Umgebung deployt wurde und welche Revisionsnummer der Code-Basis genutzt wurde.

Um den Lieferprozess anzustoßen, muss dieser getriggert werden können. Dabei sollte der Teil von Continuous-Integration durch eine neue Version der Code-Basis automatisch angestoßen werden, während es für das Deployment auf das Test- und Produktivsystem automatische oder manuelle Möglichkeiten geben sollte. Letzteres müsste dann über eine Oberfläche zugänglich sein und z. B. über einen einfachen Knopfdruck angestoßen werden können.

Weitere Komponente sind die Dokumentation sowie ein schnelles Feedback an den Nutzer. Folgende Elemente sollten dokumentiert und dargestellt werden:

- Die Durchlaufzeit einer Stage,

- die Fehler, die aufgetreten sind sowie deren Gründe,
- die Anzeige von Test-Reports, die eine qualitative Beurteilung des Builds ermöglichen.

Die Commit-Stage und Acceptance-Stage erfordern die Kommunikation und Ausführung verschiedener Systeme und Programme zur Erfüllung der Aufgaben dieser Stage. Hierzu gehören das Laden der aktuellen Quellcode-Basis aus der Versionsverwaltung, das Anstoßen von Build-Tool, die Verwaltung der Binaries in einem Repository, die Ausführung von Test-Skripts und die Vorbereitung der Ausführungsumgebung für Test und Produktion.

Die in diesen Bereichen vorhandene Infrastruktur von adesso muss Berücksichtigung finden. So wird für die Verwaltung der Quellcode-Basis Subversion (SVN) eingesetzt und je nach Projekt Ant oder Maven zum Erstellen der Binaries verwendet.

5.2. Evaluierung der Werkzeuge

5.2.1. *Go* von Thoughtworks

Go wird von ThoughtsWors Studois[2] entwickelt. Der Fokus von ThougthWorks liegt auf Software, die agile Methoden unterstützen. *Go* wird unter dem Slogan "*Agile Release Management*" angeboten.[3]

Konzept von *Go*

Mit *Go* können automatisierte Build-, Tests und Releaseprozesse modelliert werden. Dabei wird das Konzept eines "*push-button deployments*" unterstützt. Das Konzept von *Go* baut auf Continuous-Integration auf und erweitert dieses um Funktionen für Continuous-Delivery. Dabei können verschiedene Pipelines für verschiedene Projekte und Produkte mit unterschiedlichen Nutzerkreisen verwaltet werden. *Go* kann Pipelines parallel verarbeiten, in dem es ein Netz von verteilten Agents nutzt.

In einer Pipeline können verschiedene Stages angelegt werden, die dann nacheinander abgearbeitet werden. Jede Stage kann einen oder mehrere Jobs definieren. Dabei dienen die Jobs der Bündelung von Tasks. Die Jobs einer Stage werden parallel ausgeführt und auf freie Agents verteilt. Die in einem Job gegliederten Tasks werden dann aber sequenziell abgearbeitet.

[2] Informationen zu ThoughtWorks Studios unter `http://www.thoughtworks-studios.com/` .
[3] Vgl. [Tho12a]

	Community	Go 20	Go 50
Maximum Users	10	20	50
Maximum Agents	3	10	25
Maximum Jobs	Unlimited	Unlimited	Unlimited
Maximum Stages	Unlimited	Unlimited	Unlimited
Support	Forums	Support Team	Support Team
Pricing, Annual, USD	Free	$12,500	$24,500

Abbildung 5.1.: Versionen und Preise von *Go*, Quelle: [Tho12b].

Go ist eine auf den Jetty-Server[4] basierende Web-Applikation, welche diesen in den Anwendungskern einbettet. Die Ausführung einer Stage wird von verteilt oder lokal laufenden Agents durchgeführt, die sich mit dem go-Server verbinden.

Lizenzvarianten

Go wird in drei Preisvarianten angeboten, eine kostenfreie Community- und zwei proprietäre Enterprise-Versionen. Der wesentliche Unterschied zwischen den Varianten besteht in der Anzahl der erlaubten User und der verteilt laufenden Agents. Agents, die lokal auf dem System des *Go*-Server ausgeführt werden, können je nach vorhandener Kapazität der Hardware unbegrenzt gestartet werden.

Abbildung 5.1 zeigt die Unterschiede zwischen den Versionen Community und der Enterprise. Bis einschließlich zur Version 12.2 war es mit der kostenfreien Community-Version nicht möglich, ein Cluster mit Agents zu betreiben. Zulässig waren bis dahin nur Agents, die auf der Server-Instanz ausgeführt worden sind. Mit Version 12.3 können in der Community-Version auch bis zu drei verteilt laufende Agents im Go-Server angemeldet werden. In der Enterprise-Version können bis zu 10 oder 25 verteilt laufende Agents betrieben werden. Zudem bietet Thoughtworks hier ein unterstützendes Support-Team, welches bei der Konfiguration des Systems helfen kann.[5]

Testprojekt und Probeversion

Humble und Farley empfehlen für die Umsetzung einer Deployment-Pipeline am Anfang ein funktionierendes Skelett der geplanten Anwendung zu verwenden.[6] Thoughtworks hat für die Erprobung eine Test-Lizenz für 30 Tage bereitgestellt, mit der sich die Vorteile

[4] Der Jetty Server ist seit 2009 ein Teil der Eclipse Foundation. Weitere Informationen unter `http://www.eclipse.org/jetty/about.php` .
[5] Vgl. [Tho12b]
[6] Vgl. [HF11, S. 132]

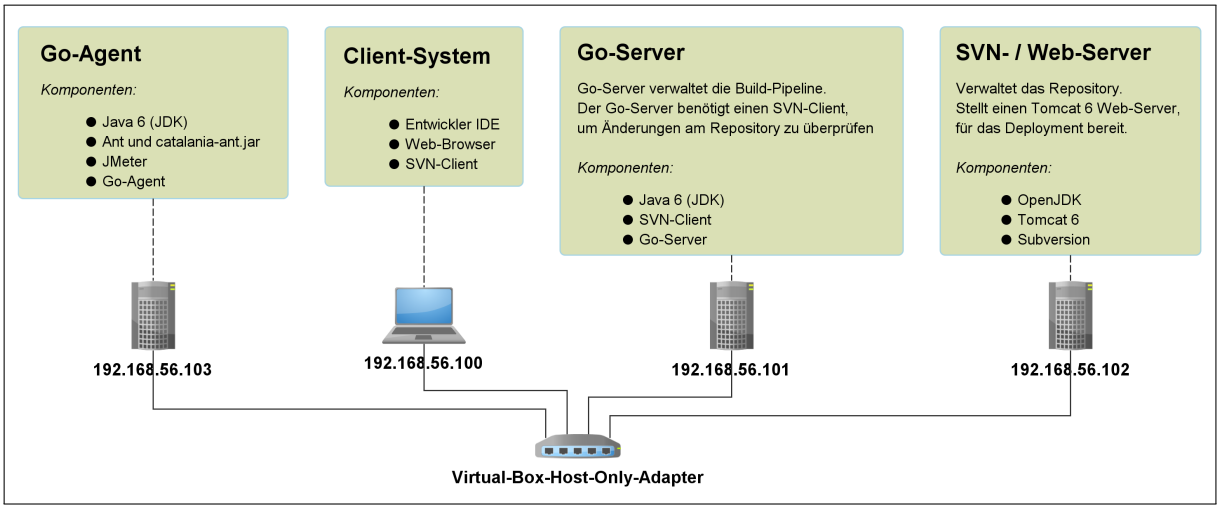


Abbildung 5.2.: Testaufbau mit VirtualBox.

der Enterpriseversion nutzen lassen.

Für den Probebetrieb von *Go* muss eine Netzwerkinfrastruktur mit mehreren Servern simuliert werden. Dies ist dadurch begründet, dass *Go* Funktionalitäten von Continuous-Integration bietet und Subsysteme wie Versionsverwaltung und Testserver für einen für den Probebetrieb benötigt werden sowie selbst als Server-Agent-Architektur realisiert wurde.

Ausgangspunkt der Deployment-Pipeline ist das Entwicklersystem. Dieser übermittelt seine Änderungen am Quellcode in das Versionsverwaltungssystem. Hierfür wird die Eclipse IDE genutzt und eine einfache Hello-World-Applikation, mit dem Projekt-Wizzard von Eclipse, als dynamisches Web-Projekt angelegt. Ziel des Testprojektes ist die Untersuchung der Konfigurationsfähigkeit und der Funktionalität von *Go*. Strukturell betrachtet wird *Go* verteilt ausgeführt. Dabei verbinden sich Agents mit dem Server und teilen ihren Status mit. Der Server verteilt neue Aufträge an freie Agents.

Um einen Test dieser Struktur zu ermöglichen, werden drei virtuelle Systeme mit VirtualBox[7] erzeugt und die Server-Variante von Ubuntu ohne GUI darauf installiert. Diese Struktur wird angelegt, um ein verteiltes System zu simulieren. Abbildung 5.2 zeigt die Netzkonfiguration für die virtuellen Systeme und den Host. Auf Grund von Kapazitätsbeschränkungen des Host-Systems werden die Test-Umgebung und das Versionsverwaltungssystem zusammengelegt.

Auf den virtuellen Systemen wird Ubunut 11.10 für 32 Bit Architekturen. Der *Go*-Server wurde mit 1280 MB Arbeitsspeicher ausgestattet, Systemvoraussetzung sind 1 GB, empfohlen aber 2 GB, die vom Host aber nicht bereitgestellt werden können. Der *Go*-Agent benötigt weniger Ressourcen und ist mit 256 MB ausgestattet. Gleiches gilt für

[7] Mehr Informationen zu VirtualBox unter `https://www.virtualbox.org/` .

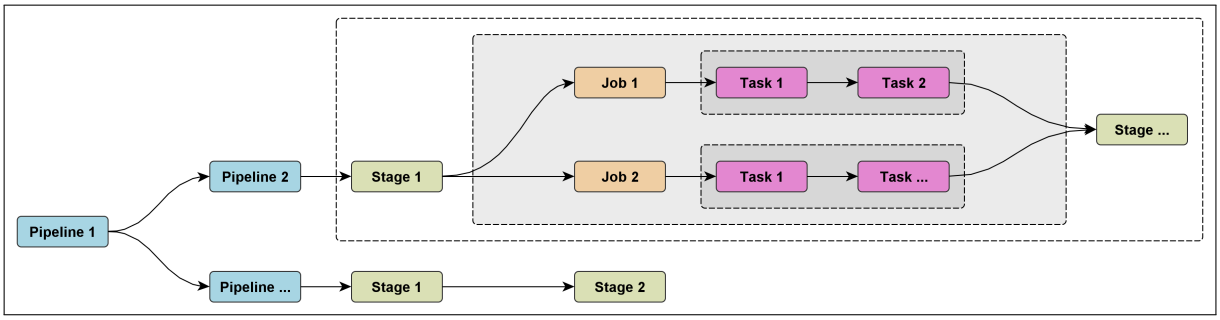


Abbildung 5.3.: Konzept der Pipelines und Stages in *Go*.

die Test-Umgebung.

Auf allen Systemen kommt das Java Development Kit in Version 6 Update 31 für 32-Bit-Architekturen zum Einsatz. Zur Zeit der Erprobung war Verison 12.2 aktuell. Die Abarbeitung der Jobs einer Stage wird durch Agents durchgeführt. Da Jobs parallele Abläufe einer Stage verwirklichen, können Jobs einer aktiven Stage auf verschiedene Agents verteilt werden. Für die Umsetzung der in Tasks vorgesehenen Aufgaben müssen auf dem Agent-System alle Voraussetzungen geschaffen werden. Der Build-Prozess des Testprojektes basiert auf Ant und JUnit-Tests für die Komponenten, was eine Installation von Ant und JUnit notwendig werden lässt. Die Quellcode-Basis muss auf das System geladen werden, wofür Subversion genutzt und ein SVN-Client installiert wird. Im weiteren Verlauf werden Akzeptanz- und Kapazitätstests mit Hilfe von Apache JMeter und JWebUnit durchgeführt. Die hierfür benötigen Bibliotheken müssen auf dem Agent-System vorhanden sein. Die Testumgebung wird auch für die Verwaltung der Code-Basis verwendet, was die Installation von Subversion bedingt. Für die automatisierten Akzeptanz-Tests wird ein Tomcat-Server in Version 6 als Deploymentziel verwendet.

Der Ablauf des Testsystems sieht Commit-Stage und Acceptance-Stage vor. Für die Acceptance-Stage muss die Testanwendung in das Testsystem deployt werden. Die Deployment-Pipeline endet hier für das Versuchsprojekt, da alle komplexeren Vorgänge sich auf gleiche Art und Weise konfigurieren werden und *Go* dabei ausschließlich das Anstoßen dieser Aufgaben in Form von Skripten ermöglicht. Komplexere Szenarien könnten hier z. B. mit Hilfe von Werkzeugen wie Chef umgesetzt werden. Infrastructure-As-A-Service, abgekürzt IaaS, ist ein Ansatz, Rechnerinfrastrukturen bei Bedarf benötigter Serverressourcen mieten zu können, welcher als ein Teil des so bezeichneten Cloud-Compunting verstanden wird.[8] Ein derartiges Szenario ist aber nicht Gegenstand dieser Arbeit und Werkzeuganalyse, sollte mit anderen Arbeiten aber noch weiterführend betrachtet werden.

[8] Vgl.: [HC11]

Funktionen

Pipeline Management Eine Pipeline kann in *Go*, wie es Humble und Farley empfehlen, durch verschiedene Stages abgebildet werden. Pipelines wiederum können Ausgangsmaterial für andere sein, bzw. selbst durch andere angestoßen werden. Das Konzept von Pipeline, Stages, Jobs und Tasks in *Go* zeigt Abbildung 5.3. Mehrere Pipelines lassen sich zu einer Baumstruktur verzweigen. Die Stages einer Pipeline werden, je nach Ergebnis der vorhergehenden Pipeline, sequentiell abgearbeitet. In einer Stage werden dann Jobs an freie Agents verteilt. Dieses Konzept ermöglicht die Abarbeitung unterschiedlich lang laufender Akzeptanztests. Dies ist ein Vorgehen, welches Humble und Farley empfehlen damit Fail-Fast ein frühestmögliches Scheitern der Pipeline bei Fehlern in der Quellcode-Basis ermöglicht wird und ein schnelles Feedback an die Entwickler gegeben werden kann.

Angestoßen werden kann die Pipeline durch verschiedene Trigger. Trigger können entweder die verwendeten Materialien der Pipeline sein oder ein manuelles Anstoßen über die Oberfläche. Als Materialien werden die Versionsverwaltungssysteme und Pipelines bezeichnet, deren aktueller Stand oder letzte Ergebnisse, Ausgangsmaterial einer Stage sind. Wird z. B. das Material Subversion verwendet, reagiert das System auf Veränderungen der Code-Basis in der Versionsverwaltung von Subversion und erzeuge eine neue Instanz der Pipeline mit der aktuellen Revisionsnummer des Materials. Abbildung 5.4 zeigt die Abhängigkeiten für die mittlere Pipeline (CI Pipeline) in Zusammenhang mit Up- (Subversion) und Downstream (Deploy-Pipeline).

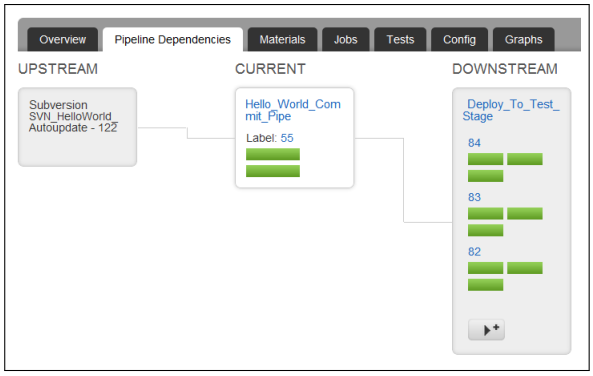


Abbildung 5.4.: Dependencie-Graph der Pipelines.

Die Oberfläche von *Go* gibt dem Entwicklungsteam ein schnelles Feedback über den Zustand der Quellcode-Basis. Abbildung 5.5 zeigt die Einstiegsseite sowie die Historie der Instanzen einer Pipeline.

Go erzeugt beim Durchlauf eines Tasks Log-Dateien, die über die Oberfläche ausgerufen werden können. Werden in einem Task Test-Protokolle im HTML-Format erstellt, können diese in die Oberfläche eingebunden und angezeigt werden. Abbildung 5.6 zeigt auf der rechten Seite das eingebundene Testprotokoll von JUnit, welches während eines Durchlaufs erzeugt wurde. Die Log-Datei ist auf der linken Seite zu sehen. Die Log-Datei hilft Fehler im Durchlauf zu identifizieren.

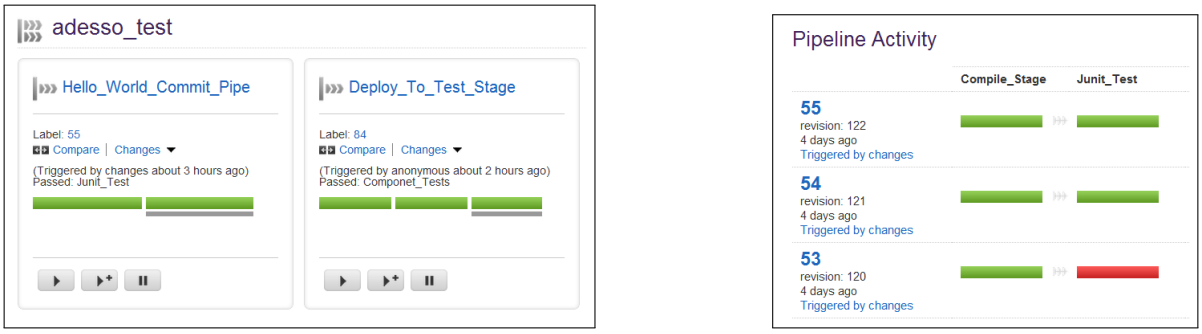


Abbildung 5.5.: Übersicht über die Pipelines und deren Historie.

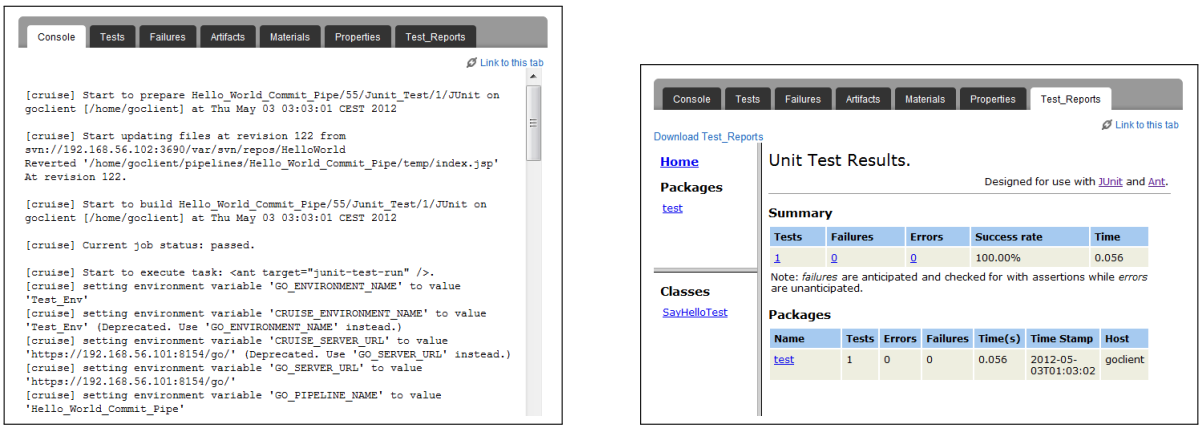


Abbildung 5.6.: Darstellung der Konsolenausgabe und des JUnit Test-Reports.

Stages Für die Commit-Stage, welche Teil von Continuous-Integration ist, können die Versionskontrollsysteme von Apache Subversion, Git, Mercurial, Preforce und der Microsoft Team Foundation Server als Materialien genutzt werden. Im Testprojekt wurde Subversion installiert, um die Code-Basis zu verwalten. Das Material gilt für die gesamte Pipeline, welche mit der aktuellen Versionsnummer instantiiert wird. Jeder Job beginnt mit dem Laden der aktuellen Version der Code-Basis in das lokale Verzeichnis des Systems. Alle Tasks können dann auf die dort hinterlegten Dateien zugreifen.

Für die Test-Pipeline ist der erste Task der Commit-Stage das Kompilieren und Packen der Binaries in Form eines War-Files. Hierfür wird der Ant-Taks genutzt. Dieser erwartet die `build.xml` und das Ant-Target, welches die Ant-Kommandos enthält. Das Target `compile-and-create-war` muss mit dem in der `build.xml` übereinstimmen, wie nachfolgendes Listing zeigt.

Listing 5.1: Ant-Target

```
<target name="compile-and-create-war">
    <.... />
</target>
```

Die in diesem Task auszuführenden Aufgaben wie kompilieren, testen, packen werden von Ant übernommen. Auf diese Weise können in *Go* alle Aufgaben für die Commit-Stage umgesetzt werden. Neben Ant werden auch NAnt und Rake unterstützt. Müssen andere Tools genutzt werden, z. B. weil ein Projekt auf Maven setzt, ist dies über ein *Custom-Command* möglich. Über *Custom-Command* können beliebige Kommandos auf der Systemkonsole ausgeführt werden und so Systemprogramme, Shell-Skripte oder andere Systeme angesprochen werden, die für die Deployment-Pipeline Aufgaben übernehmen.

Für Anwendungen, die auf das Build-Tool Ant setzen, ergeben sich eine ganze Reihe von Möglichkeiten, Tests und Analysen auf der Quellcode-Basis durchführen zu lassen. So kann z. B. ein Sonar-Server über ein Ant-Plugin eingebunden werden, um die statische Code-Analyse zu ermöglichen und Parameter wie Testabdeckung und Code-Style analysieren zu lassen. Die Testabdeckung lässt sich bei Sonar über den Anteil durch Testfälle abgedeckter Codezeilen als auch über die den Anteil der im Test durchlaufenen Kontrollflusspfade berechnen.[9]

Aus den durchgeführten Tests lassen sich so Quality-Gates definieren, welche über die Qualität der Anwendung wachen. Dies spielt eine Rolle bei der Entscheidung, ob der neue Stand der Quellcode-Basis die notwendigen Qualitätskriterien erfüllt, um in die nachfolgende Stage oder in die Produktivumgebung weitergereicht werden zu können. Ein Beispiel für ein Quality-Gate ist die Definition eines Merkmals wie die Testabdeckung. Hier könnte dann, je nach Definition, ein Wert von mindestens 70 % bestimmt werden, der erreicht werden muss, bevor die Pipeline von der Commit-Stage in die Acceptance-Stage übergehen kann. Dieses Kriterium kann aber nur eines von weiteren sein, um den Weiterfluss innerhalb der Deployment-Pipeline zu steuern.

Die Acceptance-Stage wurde im Testprojekt einfach gehalten. Sie diente vornehmlich der Erprobung der Testversion und wurde aus diesem Grunde nicht weiter vertieft. Umgesetzt wurden im JWebUnit ein einfacher Test, der die Anforderung an die Hello-World-Anwendung prüft und testet, ob die Nachricht "*Continuous Delivery Test Web Project*" beim Aufrufen der Seite erscheint. JWebUnit ist ein auf Java basierendes Test-Framework für Web-Anwendungen von SourceForge. JWebUnit vereint dabei die Frameworks HtmlUnit und Selenium durch eine einheitliche Schnittstelle, benötigt aber JUnit zur Ausführung.[10] Für den Aufbau der Deployment-Pipeline besitzt dieser Test aber eine hinreichende Größe, um die Funktionsweise der Pipeline zu testen.

Nachfolgendes Listing zeigt diesen Test:

[9] Vgl. [Man10]
[10] Vgl. [Sou]

Listing 5.2: Smoke-Test mit JWebUnit

```
\begin{lstlisting}
public class IndexPageTest {
  @Before
    public void prepare() {
      setBaseUrl("http://192.168.56.102:8080/HelloWorld");
    }
  @Test
  public void openIndexPage() {
    beginAt("index.jsp");
    assertTitleEquals("Continuous␣Delivery␣Test␣Web␣Project");
    assertElementPresent("sayHelloTitel");
  }
}
```

Die Ausführung wird als Ant-Task in die Deployment-Pipeline von *Go* eingebaut. Auf Ebene von *Go* muss hier lediglich das Ant-Target `jwebunit-test-run` angegeben werden. Um JWebUnit nutzen zu können, müssen Jar-Dateien von JWebUnit auf dem ausführenden Agent-System verfügbar sein und der Pfad zum Classpath, z. B. per `build.xml`, hinzugefügt werden.

Listing 5.3: Einbinden von JWebUnit in Ant

```
...
<property name="jwebunit-home" value="/opt/JWebUnit" />
...
<target name="jwebunit-test-run">
  <javac srcdir="JWebUnit" destdir="JWebUnit">
    <classpath>
      <path refid="junit-jar" /><path refid="jwebunit-classpath" />
    </classpath>
  </javac>
  <mkdir dir="jwebunitTestReports" />
  <junit haltonfailure="true" includeantruntime="true"
      printsummary="true" showoutput="true">
    <classpath>
      <pathelement location="JWebUnit"/>
      <path refid="junit-jar" />
      <path refid="jwebunit-classpath" />
    </classpath>
    <formatter type="xml" />
    <batchtest fork="yes" todir="jwebunitTestReports">
      <fileset dir="JWebUnit"><include name="**/*Test*.class" />
```

```
    </batchtest>
  </junit>
  <junitreport todir="jwebunitTestReports">
    <fileset dir="jwebunitTestReports" includes="TEST*.xml" />
     <report todir="jwebunitTestReports" />
   </junitreport>
</target>
```

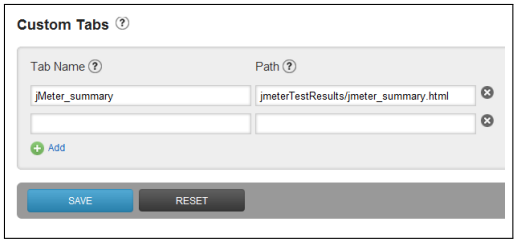
Nach der Kompilierung der Tests wird ein Verzeichnis angelegt, in welches JUnit die generierten Testberichte im XML-Format ablegen kann. Nach dem Durchlauf der Tests werden diese dann von <**junitreport**> in formatierte HTML-Dateien umgewandelt. Abbildung 5.7 zeigt, wie Berichte im HTML-Format in die Oberfläche eingebunden werden können. Berichte sind dabei Artefakte der Deployment-Pipeline, die beim Durchlauf dieser entstehen. Artefakte verbleiben, sofern nicht anders spezifiziert, im Arbeitsverzeichnis des Agents und werden bei der nächsten Aktivierung des Agents gelöscht. Um Artefakte für die weitere Verwendung innerhalb der Pipeline verwenden zu können, müssen diese auf den *Go*-Server übertragen werden. Dies kann über die Konfiguration von *Artifacts* erreicht werden. In der Abbildung werden die Testberichte von JMeter als Test-Artefakte auf den Server übertragen. Über die Konfiguration eines *Custom Tab* kann dann eine beliebige HTML-Seite mit den Testergebnissen eingebunden werden. Das Entwicklerteam hat so einen schnellen Zugriff auf die Testauswertung.

Da jeder so eingefügte Bericht an die Instanz der Pipeline gebunden ist, können auch alle Berichte der vorhergehenden Durchläufe aufgerufen werden. Auf diese Weise werden Regressionen an der Quellcode-Basis festgestellt. Verschlechtern sich die Ergebnisse wie die Anzahl von Fehlschlägen, die Größe der Testabdeckung oder die Durchlaufzeit, kann dies über die Historie näher inspiziert werden und eine bestimmte Änderung an der Code-Basis als Ursache identifiziert werden. *Go* bietet hier eine kontinuierliche Qualitätssicherung, wie sie von Continuous-Integration vorgeschlagen wird.

Ein effektiver Ansatz, Akzeptanztests in die Deployment-Pipeline einzubauen, ist das *B*ehavior *D*riven *D*evelopment (BDD). Bei BDD werden vor Beginn der Entwicklungsarbeit, Akzeptanzkriterien aus verbaler Sprache in mehreren Iterationen, in ausführbarere Testfälle umgeformt.[11] Der Vorteil, Akzeptanzkriterien zu Beginn der Entwicklung einer Anwendung als automatisiert durchführbare Tests zu definieren, kommt bei Continuous-Delivery sehr gut zum Tragen. Die so entwickelten Akzeptanztests können verwendet werden, um innerhalb der Deployment-Pipeline entscheiden zu können, ob alle für ein Release notwendigen Akzeptanzkriterien erfüllt werden. Da die Deployment-Pipeline bei jeder Änderung der Quellcode-Basis aktiviert wird, werden auch alle definierten und als

[11] Vgl. [Nor06]

(a) Log-Datei



(b) JUnit Testbericht

Abbildung 5.7.: JUnit-Testberichte in die OBerfläche von *Go* einbinden.

Test implementierten Akzeptanzkriterien geprüft.

In *Go* lassen sich Akzeptanztests sehr gut in Form der Acceptance-Stage einbauen. Die Entscheidung, ob in die Produktionsumgebung bzw. in die User-Acceptance-Stage geliefert wird, muss in *Go* durch ein Skript realisiert werden. Dieses muss die vorliegenden Test-Berichte analysieren und aggregieren können. Dieser Vorgang sollte einen Qualitätsindex erzeugen können, der der Pipeline einen Messwert für die Entscheidungsfindung liefert. Weniger Komplex kann dies auch als manueller Vorgang gestaltet werden. Nach dem Durchlauf der Akzeptanztests wird dem Entwicklungsteam eine Nachricht über den Abschluss der Stage zugesandt. Ein Teammitglied prüft dann die Testberichte und entscheidet, ob diese Version reif für die Produktion ist.

Deployment Wie das Ausrollen einer Anwendung in eine bestimmte Umgebung organisiert werden kann, gibt *Go* nicht vor. Zur Unterstützung wird aber folgendes geboten:

- *Custom-Command* durch die freie Verwendung der Systemkonsole z. B. in Verbindung mit Shell-Skripten.
- *Ant Taks*, bzw. auch die unterstützten Tools NAnt und Rake, in Verbindung mit Plug-Ins für diese Build-Tools, die das Deployment in bestimmte Server-Umgebungen ermöglichen.
- *Go*-Agent selbst als Ausführungsumgebung der Software.

In Verbindung mit *Custom-Command* können Werkzeuge wie Chef angesprochen werden, um ein Deployement zu organisieren. So bietet das Plugin *knife-ec2*[12] für Chef die Möglichkeit, Instanzen von Amazons EC2 direkt provisionieren zu können[13], wie folgendes Kommando zeigt:

12 Mehr Informationen zu knife-ec2 unter: `https://github.com/opscode/knife-ec2`
13 Vgl. [Sto12], interne Quelle und [Tim12]

Listing 5.4: Knife-Anweisung für EC2 Instanz

```
knife ec2 server create -I ami-3323f25a -S homeKey.pem
  -f t1.micro -i ./.chef/homeKey.pem -x ubuntu -r role[programXYZ]
```

Dabei enthält role[programmXYZ] Anweisungen, die Teile eines Kochbuchs von Chef sind. Innerhalb einer Rolle werden dann alle Anweisungen zusammengefasst, um ein System zu konfigurieren. Auf diese Weise können z. B. benötigte Server und Infrastruktur erstellt werden, um die Anwendung zu testen oder in Produktion zu bringen.[14]

Eine weitere Möglichkeit, die *Go* bietet, aber nicht die Flexibilität von Chef mitbringt, ist die Verwendung eine Plugins für Ant, um eine Web-Anwendung zu deployen. In der Test-Pipeline wird ein Plugin für Tomcat verwendet, um mit dem Tomcat-Manager zu kommunizieren. Nachfolgendes Listing zeigt die Deyploy-Anweisung für Ant, um das War-File der Hello-World-Anwendung in eine Tomcat-Instanz zu deployen:

Listing 5.5: Tomcat-Deployment mit Ant

```
<property name="tomcat-manager-url"
  value="http://${tomcat-server-ip}:8080/manager" />
...
<path id="catalina-ant-classpath">
  <pathelement location="${ant-lib-dir}/catalina-ant.jar"/>
</path>
...
<taskdef name="deploy"
   classpathref="catalina-ant-classpath"
   classname="org.apache.catalina.ant.DeployTask" />
...
<target name="deploy-tomcat-from-compile-pipe" depends="get-war-file">
  <deploy url="${tomcat-manager-url}" username="${tomcat-username}"
    password="${tomcat-password}" path="/HelloWorld"
    war="file:/home/goclient/pipelines/"
      "${go.GO_PIPELINE_NAME}/${deploy-war-folder}"
    update="true">
  </deploy>
</target>
```

Zumindest für eine Testumgebung und für kleinere Anwendungen mit fixer Infrastruktur ist dies eine einfache Möglichkeit, eine Web-Anwendung zu deployen. Die dritte Möglichkeit bedingt die Nutzung des *Go*-Agents als Ausführungsumgebung.

[14] Vgl. [Ghe10]

Alle Anweisungen zur Anpassung der Infrastruktur, an die die Ausführung der Anwendung geknüpft ist, können dann über *Custom-Command* oder Shell-Scripte ausgeführt werden. Ein Shell-Skript sollte dann Teil der Versionskontrolle sein. Änderungen können so auch extern, z. B. in der Entwicklungsumgebung vorgenommen werden. Die aktuelle Version des Shell-Skripts wird durch das Agent-System als Teil des auszuführenden Jobs in das Arbeitsverzeichnis geladen und kann dann von dort aufgerufen werden. Sind alle Anpassungen vorgenommen, können die Binaries ausgeführt oder wie Beispielprojekt in das Arbeitsverzeichnis des Tomcat-Servers kopiert werden. Der Tomcat stoppt die aktuell ausgeführte Version der der Anwendung, entpackt das neue Archiv und startet die Anwendung neu. Um diesen Schritt vollziehen zu können, müssen Binaries, die in einer anderen Stage kompiliert wurden, in das aktuelle Arbeitsverzeichnis kopiert werden. Dieses Szenario ist allerdings nur auf kleinere Projekte mit kleinerer Testinfrastruktur bzw. auf ein paar wenige Projekte begrenzt, da, je nach Lizenz, nur 10 bzw. 25 Agents verwendet werden können. Bei der kostenfreien Communit-Version sind es lediglich drei Agents.

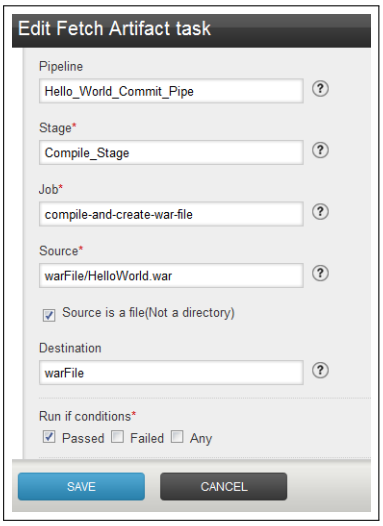


Abbildung 5.8.: Artefakt einer anderen Stage wiederverwenden.

Beurteilung

Funktionalitäten *Go* kann auf den Betriebssystemen Microsoft Windows, Mac OS X, Linux und Solaris ausgeführt werden. Möglich wird dies durch die Plattformunabhängigkeit der Java-Runtime, auf welcher *Go* läuft. *Go* bietet eine umfassende Abbildung einer Deployment-Pipeline mit der Modellierung der Commit-Stage, Acceptance-Stage, User-Acceptance-Stage, Capacity-Stage und Production. Jobs können parallel ausgeführt werden, wodurch das Konzept von Fail-Fast möglich wird. Um die Aufgaben *compile*, *test* und *deploy* verwirklichen zu können, wird Ant, NAnt und Rake unterstützt. Andere Tools oder Shell-Skripte können als Kommando für die Systemkonsole mittels *Custom-Command* angestoßen werden.

Die Stages werden der Reihe nach ausgeführt. Eine Pipeline wird durch die Ereignisse Material oder Pipeline angestoßen. Letzteres ermöglicht den Aufbau von Baumstrukturen für Pipelines und die Abbildung komplexerer Lieferprozesse.

Um komplexe Deployment-Szenarien für Test- oder Produktivumgebungen lösen zu können, bietet *Go* mit dem Ant-Taks sowie dem *Custom-Command* verschiedene Möglichkeiten, dieses durchführen zu können. Dabei können Tools wie Chef in Verbindung

mit der EC2 von Amazon genutzt werden, um sich auf einfache Weise mit benötigten Infrastrukturen für Test- oder Produktivumgebung versorgen und konfigurieren zu können. Auch über Ant ist das Deployment, z. B. über den Catalina-Ant-Task, realisierbar. Eine letzte Möglichkeit ist die Verbindung von Ausführungsumgebung und *Go*-Agent auf einem System. Dabei können die notwendigen Anpassungen als *Custom-Command* in der Pipeline-Konfiguration oder als Shell-Skript in das Versionskontrollsystem abgelegt werden. Der Agent hat dann direkte Kontrolle auf das System und kann Anwendungen installieren, starten oder stoppen.

Go bietet damit ein vollständiges Continuous-Delivery-System an, welches Continuous-Integration integriert und das Konzept so erweitert, dass entweder externe Deployment-Lösungen angesprochen werden können oder aber der *Go*-Agent selbst zu Ausführungsumgebung wird.

Zuverlässigkeit *Go* wird seit Ende September in der Version 12.3 angeboten. Für die Erprobung im Juni und Juli 2012 kam dementsprechend Version 12.2 zum Einsatz. *Go* befindet sich als proprietär vertriebenes Produkt in der ständigen Weiterentwicklung. Es kann davon ausgegangen werden, dass dieses Produkt auch in den nächsten Jahren weiterentwickelt, gewartet und verbessert wird. Treten im Programmablauf Fehler auf, werden diese geloggt. Das Problem mit der Testversion ist vornehmlich durch die geringe Ausstattung des Testsystems mit 1280 MB RAM aufgetreten.

Die Konfigurationseinstellung von *Go* kann durch ein Back-up im Administrationsbereich der Oberfläche gesichert werden. Hierzu gehören dann alle Einstellungen wie Pipelines, Nutzer, Rollen, Umgebungen, registrierte Agent-Systeme und weitere Informationen. Eine Sicherung von Artefakten erfolgt nicht, was auch nicht notwendig erscheint. Alle Artefakte lassen sich durch Aktivierung einer Instanz der Pipeline mit der gesuchten Revisionsnummer des Versionskontrollsystem wiederherstellen.

Benutzbarkeit *Go* orientiert sich eng am Konzept von Continuous-Delivery. Pipelines und Stages werden auch in *Go* an das Konzept angelehnt bezeichnet, was den Einstieg in die Konfiguration erleichtert. Die gesamte Anwendung kann über die Oberfläche bedient und konfiguriert werden. Es ist aber auch möglich, die Systemkonfiguration in Form einer XML-Datei in die Anwendung zu laden. Das ermöglicht die Konfigurationen für *Go* selbst unter die Versionskontrolle zu stellen. Diese Möglichkeit lässt die Deployment-Pipeline noch stabiler werden. Schlägt der Lieferprozess nach einer Änderung der Konfiguration fehl, kann die alte Konfiguration sehr schnell wiederhergestellt werden.

Die Oberfläche selbst zeigt den aktuellen Zustand der Pipeline sowie die Ergebnisse der letzten Durchläufe an. Der Nutzer erhält ein sehr schnelles Feedback über die derzeit

vom System ausgeführten Aufgaben. Symbole wie *Play* und *Pause*, mit denen ein Prozess manuell angestoßen bzw. pausiert werden kann, erleichtern die Bedienung der Pipelines. Statusinformationen werden über die farbigen Elemente mit den Farben grün, rot und gelb. Wird eine Pipeline bzw. ein Job einer Pipeline ausgeführt, wird dies als gelb-animierter Balken in der Oberfläche gezeigt.

Die Oberfläche ist durch das Layout und die Farbgestaltung klar strukturiert. Der Nutzer kann sich jederzeit orientieren und eine intuitive Bedienung ist möglich. So sind Verbindungen zwischen Übersichten und Detailansichten klar und einheitlich durch Anklicken des Titels geregelt.

Die Dokumentation von *Go* ist nach Rollen gegliedert und erleichtert so den Einstieg in die Anwendung. Mit der Formulierung "*As a developer, i want to ...*" werden die typischen Aufgaben eingeleitet, die z. B. ein Entwickler mit *Go* gerne erledigen möchte, wie "*... watch what's currently building*".[15]

Effizienz Die empfohlene Systemkonfiguration für *Go* ist ein 2 GHz Prozessor. Für den Server werden mindestens 1 bis 2 GB RAM und für das Agent-System 128 bis 256 MB vorausgesetzt. Der Probebetrieb erfolgte in einer virtuellen Umgebung. Das Host-System war ein mit 4 GB Arbeitsspeicher ausgestatteter Laptop mit Windows 7. Um das Host-System nicht zu überbeanspruchen, konnten die virtuellen Systeme nicht mit der empfohlenen, sondern nur mit der minimal geforderten Speichergröße versehen werden. Zu Beginn ist der Server mit 1024 MB und der Agent mit 256 MB ausgestattet worden. Da der Server am Anfang der Erprobung instabil lief und bei der Ausführung einer Stage Speicherprobleme auf der Konsole ausgab, wurde der Speicher auf 1280 MB angehoben, was die Probleme beseitigte. Der *Go*-Agent wurde gleich mit der empfohlenen Größe versehen, sodass hier im gesamten Verlauf keine Probleme auftraten.

Ein weiteres Merkmal, welches bei der Planung Berücksichtigung finden sollte, ist der notwendige Plattenplatz, um Artefakte ablegen zu können. Jede Aktivierung der Pipeline produzierte Testberichte, Log-Dateien und Binaries. Je häufiger die Pipeline aktiviert wird, desto mehr Plattenplatz muss zur Verfügung stehen, um die anfallenden Artefakte speichern zu können. Eine Webanwendung, die in ihrer kompilierten Form z. B. 30 bis 40 MB Platz beansprucht, würde bei geschätzten 100 Änderungen der Quellcodebasis 3 bis 4 GB Speicherplatz benötigen. Ein Team, welches aus 5 Entwicklern besteht und täglich einmal die Änderungen commited, erreicht dieses Volumen in 4 Wochen. Laufen 10 oder 20 Projekte parallel, kommen so in nur 4 Wochen 30 bis 80 GB zusammen.

Da die Deployment-Pipeline in *Go* mit einer beliebigen Version der Quellcode-Basis jederzeit auch die gleichen Ergebnisse produziert, sind nur die zuletzt erzeugten Artefak-

[15] Vgl. [Tho]

te von einem besonderen Interesse. Älter Artefakte können gelöscht werden. *Go* bietet die Möglichkeit, den Speicherplatz für Artefakte zu limitieren. Dabei wird regelmäßig geprüft, wie viel Restspeicher auf dem System noch vorhanden ist. Eine entsprechende Konfiguration könnte so beim Erreichen einer Untergrenze von 10 GB Restspeicher alle Artefakte so lange löschen, bis der freie Plattenspeicher wieder mindestens 25 GB beträgt.

Portabilität Eine Erweiterung von *Go*, z. B. durch Plug-ins, ist nicht möglich. Eine Änderung der Anwendung ist nicht möglich. Im Probelauf wurde aber keine Funktionalität vermisst, welche eine Änderbarkeit der Anwendung erwünscht hätte.

Die Installation von *Go* war einfach und unkompliziert. Sofern die notwendige Infrastruktur in Form der Java-Runtime von Oracle und Subversion installiert ist, kann der *Go*-Server starten. Je nach Tasks, die in der Deployment-Pipeline ausgeführter werden sollen, müssen zusätzliche Werkzeuge auf dem Agent-System installiert werden. Im Testprojekt waren diese, zusätzlich zu den schon genanten, JUnit Ant-Plugin, JWebUnit Ant-Plugin, Catalina Ant-Plugin und das JMeter Ant-Plugin.

Go kann als JAR-File direkt von der Konsole mit java −jar go.jar gestartet werden. Der *Go*-Agent muss zusätzlich noch die Adresse des *Go*-Servers als Parameter übergeben bekommen: java −jar agent−bootstrapper.jar <go−server−host>:<port>

Die Dokumentation gibt alle notwendigen Vorgaben zum Systembetrieb, sodass die Installation und die Konfiguration problemlos verlaufen sind. Der *Go*-Server, im Gegensatz zum *Go*-Agent, benötigt reichlich Speicherplatz. Bei entsprechender Hardware-Konfiguration können aber mehrerer Agents auf dem selben System parallel gestartet und am Server angemeldet werden.

5.2.2. *Deployinator* von Etsy

Deployinator wurde von Etsy entwickelt, um Änderungen der eigenen Anwendung, den Etsy Web-Shop[16], schnellstmöglich in die Produktivumgebung zu bringen. Einen maßgeblichen Anteil an der Entwicklung von *Deployinator* hatte Erik Kastner, der *Deployinator* über einen Eintrag vom 29.07.2011 auf dem Entwickler-Blog[17] von Etsy frei zugänglich[18] machte und unter die MIT-Lizenz[19] stellte.[20] Durch diesen Lizenztyp ist es möglich, *Deployinator* z. B. frei zu verwenden, zu kopieren oder zu ändern.[21]

[16] Mehr Information zu Etsy unter: `http://www.etsy.com/`
[17] Entwickler-Blog von Etsy unter: `http://codeascraft.etsy.com/`
[18] *Deployinator* auf GitHub unter: `https://github.com/etsy/deployinator`
[19] Mehr Informationen zur MIT-Lizenz unter: `http://opensource.org/licenses/mit-license.php`
[20] Vgl. [Kas10]
[21] Vgl. [The12]

Kastner fasst *Deployinator* in einem Satz wie folgt zusammen: "*one button web-based deployment app*". Vor *Deployinator* benötigte Etsy für ein Deployment ihrer Web-Anwendung in die Produktivumgebung drei Entwickler und einen Verantwortlichen aus dem IT-Betrieb. Mit *Deployinator*, so Kastner, war es Etsy möglich geworden, das Deployment auf unter zwei Minuten zu reduzieren, das nun praktisch von jedem Teammitglied durchgeführt werden kann. Etsy deployt die Anwendung seit dem mehrmals am Tag.[22]

Die Entwicklungsziele von *Deployinator* fasst Kastner so zusammen:

- Die Anwendung ist Web-basierend,
- loggt Ereignisse,
- lässt sich in die Infrastruktur von Etsy integrieren,
- nutzt IRC und E-Mail, um das Team zu informieren,
- ist transparent im Hinblick auf die ausgeführten Aktionen und
- die Integration in das Monitoring-System von Etsy ist möglich.

Konzept von *Deployinator*

Als Web-Anwendung läuft *Deployinator* als Ruby-Anwendung, welche auf das Sinatra-Framework[23] aufbaut. Deployinator ermöglicht die Ausführung von Shell-Skripten und Kommandos auf der Systemkonsole. Um verschiedene Projekte bzw. Anwendungskomponenten wie Web oder Datenbank parallel verwalten zu können, lassen sich Stacks mit eigenen Konfigurationen definieren. Jeder Stack kann wiederum für verschiedene Umgebungen konfiguriert werden. So ist z. B. die Definition von *Test* und *Produktion* möglich. Technisch ist ein Stack ein Ruby-Modul, für das bestimmte Methoden zu implementieren sind. Durch dieses Modul wird der Stack konfiguriert.

Das Konzept von *Deployinator* unterscheidet sich wesentlich von *Go*. *Deployinator* ergänzt ein bestehendes System für Continuous-Integration. Das Kompilieren und Durchführen automatisierter Tests steht hier nicht im Fokus. Vielmehr soll *Deployinator* das Ausrollen einer Anwendung in Produktivumgebung dahin gehend unterstützen, dass es dem Entwicklungsteam eine Oberfläche bereitstellt, Deployment-Skripte anstoßen zu können.

Funktionen von *Deployinator*

[22] Vgl. [Kas10]

[23] Weiter Informationen zum Sinatra-Framework unter: `http://www.sinatrarb.com/`

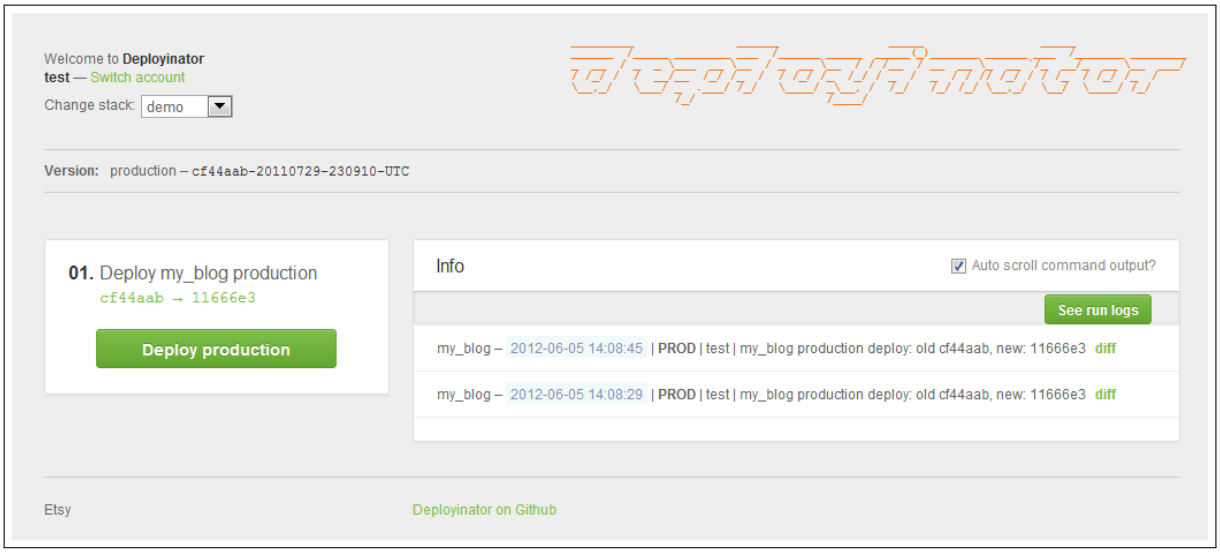


Abbildung 5.9.: Oberfläche von *Deployinator*.

Management der Deployment-Pipeline Die Modellierung von Pipelines, wie es in *Go* möglich ist, kann derart mit *Deployinator* nicht umgesetzt werden. Das bedeutet nicht, dass keine Deployment-Pipeline mit *Deployinator* möglich ist. Ferner ist *Deployinator* die geeignete Erweiterung eines CI-Systems. Die Hauptseite von *Deployinator* ist in Abbildung 5.9 dargestellt. *Deployinator* erzeugt je Umgebung eines Stacks einen Button auf der Web-Oberfläche. Bei Betätigung des Buttons werden die im Modul definierten Aktionen, wie der Aufruf von Kommandos auf der Systemkonsole, durchgeführt. Demnach werden alle Prozesse manuell angestoßen. Hat das CI-System alle Testkriterien in Form der Komponenten- und Akzeptanz-Tests sowie der statischen Code-Analyse durchgeführt, kann dies durchaus gewünscht sein. Im Fall der User-Acceptance-Stage kann sich dann das Testteam die aktuelle Version der Anwendung zum Testen in die Testumgebung deployen. Diese entspricht einem Self-Service-Portal.

Relevante Ausgaben bei der Durchführung einzelner Aktionen können in ein Log-File geschrieben werden. Es hängt aber von der jeweiligen Konfiguration ab, ob Informationen in das Log-File geschrieben werden. *Deployinator* stellt hierfür lediglich Methoden bereit, die dies ermöglichen. Das Log-File selbst wird auf der Weboberfläche eingeblendet. Dabei aktualisiert es sich selbst in kurzen Abständen, wodurch der Eindruck einer aktiven Systemkonsole entsteht.

Obwohl *Deployinator* konzeptionell ein CI-System erweitert, ist es möglich, Build und Test-Prozesse auch mit *Deployinator* umzusetzen. Allerdings ist dabei kein automatisches triggern durch Änderungen an der Code-Basis möglich. Alle definierten Prozesse müssen über die Oberfläche manuell angestoßen werden.

Stages *Deployinator* integriert Git für Aufgaben der Versionsverwaltung. Dabei ist es aber auch ohne Probleme möglich, auch andere Systeme wie Subversion zu verwenden und diese über die Systemkonsole von *Deployinator* ansprechen zu lassen. Im Testprojekt wird die Code-Basis aus Subversion geladen. Nachfolgendes Listing zeigt die Einbindung von Shell-Skripten in das Stack-Modul:

Listing 5.6: Stack-Modul von Deployinator

```
def my_test_build_and_deploy(options={})
  # last deployed version
  old_build = %x{cat $userHome/current_deployed_version}
  # current version on
  cur_build = %x{svn info
    svn://192.168.56.102:3690/var/svn/repos/HelloWorld
    | grep '^Revision:' | awk '{print␣$2}'}
  # run build-script
  run_cmd %Q{$userHome/build_war.sh}
  # log the deploy
  log_and_shout :old_build => old_build, :build => cur_build
end
```

Das Build-Skript, welches hier angestoßen wird, lädt zu erst die aktuelle Quellcode-Basis aus dem Versionsverwaltungssystem. Die derzeit ausgeführte Version wird in einer lokalen Verzeichnisstruktur gesichert. Abildung 5.10 zeigt zwei implementierte Stacks. Jeder dieser Push-Buttons aktiviert eine eigene Methode im Stack-Modul.

Deployment *Deployinator* wurde besonders auf die schnelle Aktivierung von Deployment-Skripten bzw. zur Aktivierung von Werkzeugen, die das Deployment übernehmen, konzipiert. Im Fokus steht hier also viel mehr deren Anstoß, als eine Unterstützung des Deployments selbst. *Deployinator* kann aber deshalb als leichtes Self-Service-Portal angesehen werden.

Perspektivisch betrachtet lässt sich *Deployinator* auch in einer Situation nutzen, in der ein direktes Deployment in die Ausführungsumgebung des Kunden nicht möglich ist. Dabei kann auf Kundenseite ein vorkonfigurierter *Deployinator* installiert werden. Einem Testteam des Kunden kann mit *Deployinator* die Möglichkeit in gegeben

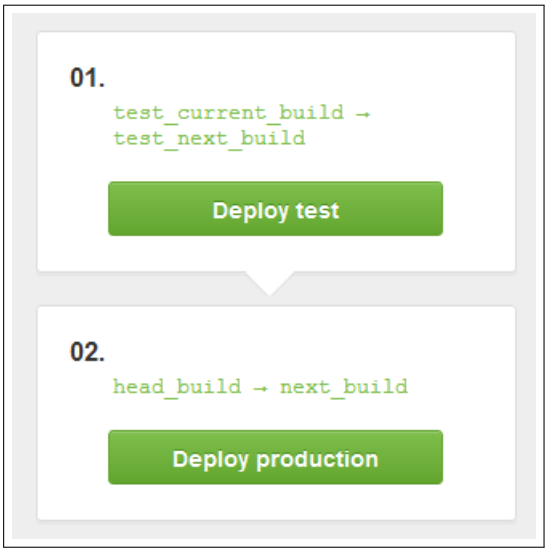


Abbildung 5.10.: Stacks mit Push-Button in *Deployinator*.

werden, sich für Testzwecke selbst mit der aktuellen Version zu versorgen. Die aktuelle Version könnte dann durch *Deployinator* in die Testumgebung zu deployt und ausgeführt werden. Die Anwendung kann dann durch den Kunden explorativ untersucht und abgenommen werden.

Dieses Szenario ist auch für die Produktivumgebung denkbar. So fern ein neues Feature durch den CI-Prozess gegangen ist und alle vereinbarten Akzeptanzkriterien erfüllt sind, wird die neue Version in ein für den Kunden erreichbares Repository ausgeliefert. Der Kunde kann hierüber durch das CI-System informiert werden. Die Testabteilung des Kunden ruft dann die Oberfläche von *Deployinator* auf und kann das Deployment aktivieren und die neue Version testen. Wird die neue Version akzeptiert, kann der Programm-Manager des Kunden bzw. der IT-Betrieb das Deployment in die Produktivumgebung anstoßen. Deployinator kann dann so konfiguriert werden, dass adesso als Lieferant in diesem Szenario über das Ausliefern in die Produktivumgebung durch das System informiert wird. Dies kann z. B. aus vertragsrechtlichen oder abrechnungstechnischen Gründen erforderlich sein.

In der Erprobung von *Deployinator* wurde ein vereinfachtes Deployment mit einer lokalen Testumgebung durchgeführt. Dabei wird das in der ersten Stufe erzeugte War-File aus dateiorientierten Repository in die aktive Tomcat-Instanz kopiert. Der Tomcat-Server registriert diesen Vorgang und führt ein redeploy mit der neuen Anwendung durch. Der Vorgang wird, ähnlich wie das Build-Script, über die Methode run_cmd `%Q{~/deploy_last_stable.sh}` aufgerufen. Folgendes Listing zeigt das Deploy-Skript:

Listing 5.7: Deployment-Skript für lokale Tomcat-Instanz

```
last_stable_file=$USER_HOME/last_stable_version
current_deployed_file=$USER_HOME/current_deployed_version
last=0
current=0
if [ -f $last_stable_file ] && [ -f $current_deployed_file ]
then
  last=`cat $last_stable_file`
  current=`cat $current_deployed_file`
  if ! [ $last -eq  $current ]
  then
    cp $USER_HOME/artefactRepo/$last/HelloWorld_$last.war
      /var/lib/tomcat7/webapps/HelloWorld.war
    echo $last > $current_deployed_file
  fi
fi
```

Werkzeuge wie Chef können das Deployment maßgeblich unterstützen und verwaltete Knoten für eine neue Version rekonfigurieren. Ein aktiven Server-Cluster für ein Update vom Netz nehmen zu können, erfordert eine entsprechende Konfiguration im Vorfeld. Besteht dies aus Apache HTTPD als Load-Balancer und mehreren Tomcat-Instanzen, kann z. B. die Synchronisierung von aktiven Nutzer-Sessions zwischen den Servern ermöglicht werden. Voraussetzung hierfür ist die Definition des Clusters in der server.xml und die Verwendung von *sticky sessions* durch den Load-Balancer. Alle verwendeten Session-Attribute müssen das Interface java.io.Serializable implementieren. Informationen zu Sessions werden dann über ein Multicast über alle Instanzen des Clusters verteilt.[24] Werden einzelne Instanzen nun der Reihe nach herrunter gefahren und die Aktualisierung durchgeführt, gehen keine Informationen aktiver Nutzer verloren.

Qualitative Beurteilung

Funktionalitäten *Deployinator* bietet selbst nicht die Implementierung einer vollständigen Deployment-Pipeline. *Deployinator* ist allerdings geeignet ein bestehendes CI-System um die fehlenden Funktionalitäten, die für die Umsetzung einer Deployment-Pipeline benötigt werden, zu erweitern. Dennoch kann *Deployinator* frei verwendet und konfiguriert werden, sodass auch mit *Deployinator* CI-Prozesse möglich wären. Fehlen würde in dieser Verwendung aber die Fähigkeit, aktiv die Version im Versionskontrollsystem zu überwachen und bei einer Änderung den CI-Prozess anzustoßen. Der Einsatz von *Deployinator* kann sich deshalb nur auf die Funktionalität eines Self-Service-Portals einschränken lassen, in dem das Testteam, der IT-Betrieb oder das Entwicklungsteam in Lage versetzt werden Deployments jederzeit anstoßen können.

Das System kann über verschieden Shell-Kommandos wie z. B. *ssh*, *wget* oder *curl* entfernte Systeme erreichen und Konfigurationen ändern bzw. die Anwendung deployen. So kann die Ermittlung der aktuellen Revisionsnummer der Code-Basis oder der neusten Version in einem zentralen Repository abgefragt werden.

Deployinator ist für eine Integration in die Systemlandschaft von Etsy konzipiert worden. Bei Etsy wird ein Single-Sign-On-Verfahren (SSO) genutzt, welches als Proxy die Attribute HTTP_X_USERNAME und HTTP_X_GROUPS in jedem Request an die Subsysteme setzt. Für einen korrekten Betrieb erwartet *Deployinator* diese Parameter im Request. Für die Testversion konnten diese in der Konfigurationsdatei config.ru durch die Vordefinition dieser Parameter umgangen werden. In einem realen Liefersystem muss hierdurch eine Instanz vorgeschaltet werden, um SSO zu nutzen und unbefugte Nutzer auszuschließen.

[24] Vgl. [The11]

Was in *Deployinator* fehlt, ist die Möglichkeit eine spezifische Version in die Produktivumgebung liefern zu können. *Deployinator* ist so konzipiert, dass entweder die neuste Version zu ausliefert wird oder die letzte stabile Version der Anwendung mit einem Rollback wiederhergestellt werden kann. Für ein Testteam kann es aber durchaus von Interesse sein nicht die neuste Version, sondern eine ältere Version in die Testumgebung zu deployen. Als Beispiel könnte es der Fall sein, dass in der aktiven Testversion ein Verhalten entdeckt wurde, dessen Vorkommen nun bei älteren Versionen verifiziert werden soll. So kann es sein, dass dieses Verhalten auch schon bei vorhergehenden Versionen auftrat, bisher aber nicht entdeckt wurde. Dem Testteam soll die Flexibilität gegeben werden, auch ältere Versionen noch deployen zu können. *Deployinator* bietet, durch die Open-Source-Lizenz, die Möglichkeit den Quellcode anzupassen. Prinzipiell besteht hier also die Möglichkeit *Deployinator* beliebig zu erweitern. Dies soll nachfolgend für das angeführte Szenario eines Auswahlfeldes beschrieben werden.

Um das Auswahlfeld in der Oberfläche zu ermöglichen, muss das Template für Singe-Push angepasst werden und ein Form-Element hinzugefügt werden:

Listing 5.8: Anpassung des Templates

```
templates/generic_single_push.mustache:
<select name="selected_version">{{version_as_html_option}}</select>
```

Eine weitere Anpassung ist im Helper-File von *Deployinator* notwendig, um den Request-Parameter der aktuellen Version auszulesen und in anderen Modulen zugänglich machen zu können.

Listing 5.9: Extrahieren der Versionsnummer aus dem HTTP-Request

```
helpers.rb:
def init(env)
  ...
  @version = form_hash(env, "selected_version")
  ...
end
...
def version_to_deploy @version end
```

Letzte Änderung betrifft die Auswahlmöglichkeiten in der Oberfläche. Die Liste möglicher Versionen könnte z. B. bei Nutzung von Artifactory zur Verwaltung der lieferbaren

Binaries über eine REST-Schnittstelle abgerufen werden.[25] Eine Methode, die die Auswahlelemente im Stack-Modul hinzufügt, sieht dann wie folgt aus:

Listing 5.10: Extrahieren der verfügbaren Versionen über die Artifactory REST-API

```
def version_as_html_option
json_response = x%{curl http://server:port/
  artifactory/api/storage/libs-release-local/org/acme/}
result_hash = JSON.parse(json_response)
children = result_hash["children"]
result = ""
children.each do |ch|
  a = ch.["uri"].[/[^\\]/]
  result << "<option>#{a}</option>"
end
return result
end
```

Artifactory liefert einen String im JSON-Format zurück. Ruby bietet hier eine einfache Möglichkeit, das Ergebnis der Abfrage zu parsen. In der folgenden For-Each-Schleife werden alle Elemente von children durchlaufen und das Element uri in die HTML-Tags gesetzt. Die Rückgabe von version_as_html_option wird dann bei der Verarbeitung des Templates an die dort zuvor definierte Position gesetzt.

Zuverlässigkeit *Deployinator* wird auf GitHub sporadisch gepflegt. Seit der initialen Bereitstellung im Juli 2011 wurden Änderungen an 6 Stellen im Quellcode durchgeführt und Bugs beseitigt. Sofern Fehler bei der Ausführung von Shell-Scripten auftreten, werden diese in die Log-Datei geschrieben. Das Log-File kann über die Oberfläche angezeigt werden. Anwendungsfehler werden je nach betroffener Ebene entweder als Fehlerseite vom Sinatra-Framework angezeigt oder auf der Konsole ausgegeben, wenn es sich um einen Compiler-Fehler handelt.

Es empfiehlt sich, zu Beginn des Aufbaus eines Liefersystems mit *Deployinator*, einen Fork auf GitHub zu erstellen, bzw. *Deployinator* in ein selbstverwaltetes Git-Repository zu überführen. Alle notwendigen Änderungen an der Konfiguration und dem Programmablauf können dann über eine integrierte Entwicklungsumgebung wie z. B. Eclipse oder rudimentärer mit Werkzeugen wie Notepad++ durchgeführt und anschließend in Git versioniert werden. Auf dem Zielsystem kann die Anweisung zum Starten von *Deployinator* wie folgt aussehen, wenn das Repository initial dort ausgecheckt wurde:

[25] Vgl. [JFr12]

Listing 5.11: Deployinator aktualisieren und starten

```
deployinator_home $ git fetch upstream
deployinator_home $ rackup
```

Änderungen am Lieferprozess können so jederzeit rückgängig und überwacht werden.

Benutzbarkeit Um *Deployinator* für eine Deployment-Pipeline konfigurieren zu können sind Sprachkenntnisse von Ruby und der Linux-Shell erforderlich. Zudem sind Kenntnisse des Konzeptes von Ruby-on-Rails hilfreich. Die Bedienung der Anwendung ist klar und verständlich. Werden die Buttons entsprechend betitelt, was sich im Stack-Modul konfigurieren lässt, kann dem Anwender die Folge der Ausführung verständlich gemacht werden. Ausgaben während der Ausführung werden in eine HTML-Datei geschrieben, die so formatiert ist, dass diese einer Systemkonsole optisch ähnelt.

Effizienz Das Zeitverhalten der Anwendung ist unter Umständen abhängig von verschiedenen Subsystemen wie der Abfrage nach der aktuellen Revisionsnummer im Versionskontrollsystem. In der Erprobung antwortete das System schnell und ohne Zeitverzögerung. Eine Speicheranalyse zeigte, dass Ruby und Rack nach dem Start 38 MB Speicherplatz belegen. Dadurch ist eine Ausführung von *Deployinator* auch auf kleineren Systemen möglich, bzw. kann auf anderen Systemen wie der Versionsverwaltung parallel betrieben werden.

Wartbarkeit Ist Know-How in der Kombination Ruby, Ruby-on-Rails und Linux-Shell vorhanden, lassen sich Fehler schnell identifizieren und beseitigen. Eine Anpassung des Systems ist durch die Quellenoffenheit problemlos möglich.

Änderungen an der Anwendung sollten jedoch mit Tests der Funktionalitäten einhergehen. In *Deployinator* selbst sind schon Testmodule hinterlegt. Diese können durch rake **test** aufgerufen werden. Dabei werden alle Testdateien mit dem Pattern *_test.rb unter dem Pfad **test** ausgeführt. Für eigene Erweiterungen sollten hier Testmodule hinterlegt werden.

Portabilität Als in Ruby geschriebene Anwendung lässt sich *Deployinator* in verschiedenen Umgebungen wie Windows, Linux, Mac OS X und Solaris ausführen. Die Installationsanweisungen von *Deployinator* sind mangelhaft und unvollständig. Für den Betrieb erforderliche Pakete und infrastrukturelle Maßnahmen sind nicht angegeben. Die Dokumentation von *Deployinator* bezieht sich lediglich auf einen Verweis auf Rack[26] und

[26] Informationen zu Rack unter: `http://rack.github.com/`

Sinatra als erforderliche Komponenten, ohne jedoch zu erwähnen, in welchem Zusammenhang *Deployinator* zu diesen Komponenten steht.[27] Rack selbst bietet eine API für Ruby und Ruby-Frameworks, welches Web-Server und Frameworks verbindet.[28] Sinatra ist ein Framework mit dem sich Web-Anwendungen schnelle entwickeln lassen sollen. Zum Betrieb benötigt Sinatra Rack für das Loggen, Debugging, URL-Routing, Authentifizierung und Session-Handling.[29]

Folgende Installationsanweisung konnte nach einer zweiwöchigen Recherchephase auf einem Ubuntu-Server mit Version 12.04 erfolgreich durchgeführt werden:

Listing 5.12: Deployinator installieren

```
cd <User-Home>
sudo -su
apt-get install ruby1.8-dev ruby1.8 ri1.8 rdoc1.8 irb1.8
apt-get install libreadline-ruby1.8 libruby1.8 libopenssl-ruby
apt-get install libxslt-dev libxml2-dev
apt-get install rubygems
gem install bundler
wget https://github.com/etsy/deployinator/zipball/master
unzip master
cd etsy-deployinator-<Version>
bundle install
rackup
```

5.2.3. *Dreadnot* von Rackspace

Dreadnot wurde vom Cloud-Monitoring-Team von Racksapce entwickelt und ist stark von *Deployinator* inspiriert worden. Nach dem *Deployinator* öffentlich zugänglich gemacht wurde, begann man bei Rackspace mit der Erprobung. Dabei stellte Rackspace allerdings fest, dass *Deployinator* nicht den speziellen Anforderungen von Rackspace genügt. Ein wesentliches Problem war die konzeptionelle Ausrichtung von *Deployinator*, als System das sich auf ein einzelnes Produkt beschränkt. Die Teams, die *Deployinator* nutzen wollten, mussten regelmäßig umständliche Anpassungen an *Deployinator* vornehmen, um die verschiedenen Umgebung und Regionen von Rackspace nutzen zu können. Aufbauend auf dem Konzept von *Deployinator* entwickelte Rackspace daher ein eigenes Werkzeug, welches sich besser in die Infrastruktur von Rackspace eingliedern lässt und

[27] Vgl. [Kas12]
[28] Vgl. [Neu07]
[29] Vgl. [Miz]

verschiedene Server-Regionen und Produkte unterstützt. *Dreadnot* ist seit dem 5. Januar 2012 unter die Apache License Version 2.0 gestellt und über GitHub[30] frei zugänglich gemacht worden.[31]

Das Konzept der Stacks von *Deployinator* wurde auch in *Dreadnot* aufgegriffen. Der Stack definiert den Ablauf des Deployments. Rackspace definiert bei sich z. B. einen Stack für Monitoring-Dienste und einen anderen für API-Services. Für jeden Stack können verschiedene Regionen angegeben werden, auf denen das Deployment durchgeführt wird. Dabei ist ersichtlich, welche Version der Anwendung in welcher Region deployt ist sowie welche Version auf dem Git-Repository verfügbar ist. *Dreadnot* ist so konzipiert, dass es ausschließlich mit einem Git-Repository zusammenarbeiten kann. Je Region können Log-Files und Diff-Ansichten zweier Versionen, in Verbindung mit dem Git-Repository, betrachtet werden.

Neben Git setzt *Dreadnot* auf weitere Infrastrukturen, deren Schnittstellen bereits integriert sind. So wird das CI-System Buildbot[32] unterstützt, welches die Anwendungen kompilieren, packen und testen kann. Zudem kann *Dreadnot* den Load-Balancer[33] eines Apache HTTPD konfigurieren sowie mit Hilfe von Chef die Anwendung ausliefern oder Anpassungen an der Infrastruktur vornehmen. Wird der Verkehr einer laufenden Anwendung in eine andere Region umgeleitet, kann der Chef-Server für diese Region modifiziert werden. Die Client-Systeme des Chef-Servers, werden in der betroffenen Region getriggert. Das Ausrollen der neuen Chef-Rezeptur wird damit angestoßen. Ist die Infrastruktur angepasst worden, ist die Konfiguration aber noch zu verifizieren, bevor die Region über den Load-Balancer wieder verfügbar gemacht wird. Scheitern die Testfälle, meldet das System über E-Mail und IRC-Dienst den kritischen Zustand, je nach Verantwortlichkeit, sofort an das Entwicklungsteam oder an den IT-Betrieb. Ein Rollback auf die vorhergehende Version ist dann manuell anzustoßen.

Technisch nutzt *Dreadnot* *nodejs*[34], einem auf der Java-Script-Runtime von Chrome aufbauenden Framework, mit sich schnelle und skalierbare Netzwerkanwendungen aufbauen lassen. Eine typische Struktur für die Konfiguration der Stacks ist wie folgt aufgebaut:

Listing 5.13: Stack in Dreadnot

```
# Pfad zum lokalen Git-Repository:
```

[30] *Dreadnot* auf GitHub unter: `https://github.com/racker/dreadnot`
[31] Vgl. [Que12]
[32] Informationen zu Buildbot unter: `http://trac.buildbot.net/`
[33] Informationen zu mod_proxy des Apache 2 unter: `http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html#balancer_manager`
[34] Infromationen zu nodejs unter: `http://nodejs.org/`

```
./data/local_git/<repos_name>/
# Warnhinweis vor dem Deplyomkent:
./data/warnings.txt
# Alle Stacks werden hier abgelegt:
./stacks/
# Pattern für Dateibezeichnung:
./stacks/<my_stack_name>_stack.js
# Passwort für Git-Repository:
./htpasswd
# Lokale Einstellungen:
./local_settings.js
```

In den lokalen Einstellungen werden Angaben zum Projekt bzw. der Produktname, die Umgebung, in die deployt werden soll, der Ort der Passwortdatei für den Zugriff auf Git, für jeden Stack die Git-URL, der genutzte Branch und in welche Region deployt werden soll, angegeben. Installiert werden kann *Dreadnot* über folgendes Kommando:

Listing 5.14: Dreadnot installieren und starten

```
sudo -su
apt-get install git
apt-get install nodejs
apt-get install npm
dreadnot -c ./local_settings.js -s ./stacks -p 8000
```

Die letzte Zeile dient dem Start der Anwendung, was allerdings erst nach der Konfiguration eines Stacks und dem Anlegen eines lokalen Git-Repositories möglich ist.

Funktionen

Dreadnot ist, so wie *Deployinator* auch, für die Ergänzung eines bestehenden CI-Systems konzipiert. Dementsprechend ist die Qualitätssicherung der neuen Version in einer vorhergehenden Stufe durchzuführen. Um den konzeptionellen Aufbau einer Deployment-Pipeline ermöglichen zu können, müssen Binaries nach dem Build-Prozess in einem von der Quellcode-Basis unabhängigen Git-Repository abgelegt werden. Nur auf diese Weise kann *Dreadnot* erkennen, ob eine neue Version für ein Deployment, unabhängig ob in Test- oder Produktionsumgebung, bereit steht. Im Gegensatz zu *Deployinator*, das verschiedene Repository-Systeme über Scripte und Kommandos auf der Systemkonsole ansprechen kann, ist dies für *Dreadnot* nicht möglich, da die aktuelle Revisionsnummer

nur aus Git abgerufen wird. Dies führt zu drei Lösungsszenarien, um eine Deployment-Pipeline mit *Dreadnot* umsetzen zu können:

1. Die Deployment-Pipeline mit Build, Test und Deployment wird ausschließlich von *Dreadnot* angestoßen. Das entspricht einer Umkehrung des Kontrollflusses, wie ihn Humble und Farley vorgeschlagen haben. Build und Test, als Teil des CI-Systems, würden dann nicht automatisch durch eine Veränderung der Quellcode-Basis, angestoßen, sondern auf menschliche Interaktion warten. Ein entscheidender Vorteil von Continuous-Integration, negative Auswirkungen einer Änderung sofort zu erkennen, würde dann allerdings verschwinden.

2. Ein CI-System wird durch eine Veränderung der Code-Basis getriggert. Anschließend wird der Quellcode kompiliert und getestet. War dieser Vorgang erfolgreich, werden Binaries in ein zusätzliches Git-Repository geladen. Die Revisionsnummer des Git-Repository ändert sich und *Dreadnot* kann diese Information durch die eingebauten Routinen abrufen und in der Oberfläche anzeigen. Die aktuelle Version kann anschließend vom Git-Repository geladen und auf das Zielsystem deployt werden.

3. *Dreadnot* wird z. B. nur in Verbindung mit interpretierbaren Sprachen wie Ruby oder PHP verwendet. Test- als auch Produktionsumgebung können dann, ohne Zwischenschritte mit der selben Quellcode-Basis arbeiten. Dieser Ansatz bringt aber den Nachteil, dass es so noch keine Qualitätsschranke im Prozess gibt. Jederzeit könnte auch die ungetestete Version ohne Zwischenprüfung in die Produktion geliefert werden. Um dies zu verhindern, müssten Qualitätsmerkmale einer Version in eine extra Datei gespeichert werden, die vor Ausführung des Deployments in der *Dreadnot*-Routine ausgewertet wird. Alle Versionen, die in die Testumgebung ausgeliefert worden sind und die Qualitätskriterien erfüllt haben, werden in einer solchen Datei erfasst.

Eine Umstellung der Anwendung auf andere Versionskontrollsysteme ist möglich, bedingt aber den Umbau von *Dreadnot*. Durch den Aufsatz von *Dreadnot* auf nodejs können, so wie in *Deployinator* auch, verschiedene Kommandos auf der Systemkonsole abgesetzt werden. Nach einer Entkopplung der Git-Bindung könnte auch Artifactory oder ein anderes System genutzt werden.

Dreadnot integriert Knife, das eine Interaktion mit dem Chef-Server ermöglicht. Mit Knife können die wesentlichen Chef-Komponenten manipuliert werden.[35]

[35] Vgl. [Tim12]

Qualitative Beurteilung

Funktionalität Eine Deployment-Pipeline für Java-Anwendungen ist möglich, bedingt jedoch andere Strukturen. So muss ein Konzept erstellt werden, auf welche Weise in *Dreadnot* neue Versionen markiert werden und anhand welcher Kriterien die Freischaltung in die Produktivumgebung erfolgen kann. Eine Möglichkeit ist die Verwendung eines separaten Git-Repositories für kompilierte Pakete. Sollen andere Tools Verwendung finden, muss *Dreadnot* angepasst werden. Diese ist aber durch die Anwendung einer Open-Soruce-Lizenz möglich.

Zuverlässigkeit Die letzten größeren Aktivität an der Quellcode-Basis von *Dreadnot* waren von Dezember 2011 bis Januar 2012. Danach sind nur noch kleinere Bugs beseitigt.

Fehler bei der Ausführung des Deployments werden in eine Log-Datei geschrieben. Ist ein Stack falsch konfiguriert, bricht das Programm ab. Die Zuverlässigkeit von *Dreadnot* hängt zum großen Teil von den verwendeten Werkzeugen und Shell-Skripten ab.

Benutzbarkeit Das Konzept der Oberfläche ist einfach gehalten. Die Unterteilung nach Produkten und Regionen ist ersichtlich. Der Nutzer erhält auf einen Blick Informationen zum aktuellen Vorgang und Information über den Erfolg vorhergehender Aktivitäten. Die Log-Datei eines Deployments kann leicht aufrufen werden. Diese enthält Informationen zu durchgeführten Aktivitäten sowie die während eines Vorgangs erzeugten Meldungen. Vor der Ausführung eines Deployments kann für die Nutzer des Portals eine Warnmeldung hinterlassen werden. Diese müssen dann bei Durchführung des Deployments die Warnmeldung bestätigen. So ist es z. B. möglich, während eines Datenbank-Updates eine Warnmeldung zu hinterlassen und eine ungewollte Durchführung des Deployments zu verhindern.

Effizienz Die Antwortzeit von *Dreadnot* ist in der Erprobung besonders niedrig gewesen. Alle Abläufe waren schnell und flüssig. Eine Speicheranalyse ergab, dass *Dreadnot* nur 25 MB Arbeitsspeicher belegt. Das nodejs-Framework, als unterliegende Plattform, selbst schläft, so fern keine Anfrage bearbeitet werden muss. Eine Verbindung benötigt nur geringen dynamischen Speicherbereich.[36] *Dreadnot* kann so auch in einer minimalistisch ausgestatteten Systemumgebung ausgeführt werden.

Wartbarkeit und Portabilität Da *Dreadnot* quellenoffen ist, können Anpassungen jederzeit vorgenommen werden. Hilfreich ist dann z. B. das Anlegen eines Forks auf GitHub.

[36] Vgl. [Joy12]

Wie auch *Deployinator* kann *Dreadnot* auf mehreren Umgebungen ausgeführt werden. Anwendungen, die auf nodejs aufbauen, können unter Windows, Mac OS X, Linux und Solaris ausgeführt werden.

5.3. Gegenüberstellung

5.3.1. Modellierung der Deployment-Pipeline

Prozessmodellierung

Mit dem Commit einer Änderung an der Quellcode-Basis bis hin zur abgenommenen und laufenden Anwendung, bietet *Go* dem Entwicklungsteam, für die Modellierung einer Deployment-Pipeline, eine einheitliche Oberfläche an. Der gesamte Prozess kann in *Go* modular untergliedert abgebildet werden. Die Kleinste Einheit in *Go* ist der Tasks. Ein Task kapselt eine einzelne Aufgaben innerhalb der Deployment-Pipeline. Tasks werden in Jobs zusammengefasst, um parallele Pfade in einer Stage abbilden zu können. Eine Stage ist das Hauptelement der Deployment-Pipeline. In einer Pipeline ist nur die sequenzielle Ausführung von Stages möglich. Es ist aber auch möglich, von einer Pipeline mehrere weitere Pipelines zu verzweigen. Über die Oberfläche ist jederzeit ersichtlich, in welchem Zustand sich die Pipeline befindet und welche Tasks ausgeführt werden.

Dreadnot und *Deployinator* bieten hier keine Möglichkeit einer vollständigen Umsetzung einer Deployment-Pipeline. Vielmehr bauen sie auf ein vorhandenes CI-System auf. Dadurch ist keine zentrale Visualisierung der Deployment-Pipeline, wie *Go* es bietet, möglich. *Dreadnot* und *Deployinator* können lediglich den Ausschnitt des Deployments darstellen.

Trigger

Ein Triggern der Prozesse durch eine Änderung der Code-Basis oder durch Aktivitäten einer anderen Pipeline ist nur in *Go* möglich. Dies spielt vornehmlich bei der Realisierung von Continuous-Integration in Form der Commit-Stageund Acceptance-Stage eine Rolle. In *Dreadnot* und *Deployinator*, aber auch in *Go*, können Prozesse über Schaltflächen in der Oberfläche manuell angestoßen werden.

Quality-Gate

Dreadnot als auch *Deployinator* kennen keine Qualitätsschranken. Als Erweiterung eines bestehenden CI-Systems müssen *Dreadnot* und *Deployinator* davon ausgehen, dass nur gut getestet Software in diese Stage weitergereicht wird, welche alle Akzeptanzkriterien

erfüllt. Die vorhergehende Prüfung dieser Kriterien muss vorausgesetzt werden können. Vor einem Einsatz, müssen Strategien gefunden werden, die einen Schutz der Produktivumgebung vor unreifen Deployments schützt. Besonders bei *Dreadnot* aber auch bei *Deployinator* würde der Einsatz unterschiedlicher Git-Repositories einen Lösungsansatz bieten. Gut getestete Version, werden anschließend in ein separates Repository überführt. In einem mehrstufigen Testverfahren, reagiert eine Teststufe auf das Repository der vorhergehenden Stufe und legt bei einem erfolgreichen Durchlauf der Testfälle die geprüfte Version in das zur Stage gehörende Repository ab. Das Repository könnte so als Quality-Gate verwendet werden.

In *Go* können Qualitätsschranken definiert werden, in dem die Tasks einer Pipeline einen Fehler erzeugen. *Go* bietet Continuous-Integration und die Ausführung von Komponenten-, Akzeptanz- und Kapazitätstests an. In *Go* hängt die Durchführung eines Tasks, einer Stage oder einer Pipeline vom Erfolg des vorher aufgerufenen Tasks ab. Kommt es in einem Task zu einem Fehler, bricht die Ausführung unmittelbar ab, wovon auch parallel laufende Jobs betroffen sind. Bei der Verwendung von Ant und JUnit, wird dies z. B. durch einen Fehler in der Prüfung einer Annahme hervorgerufen. Ein als nachfolgend konfiguriertes Deployment wird dann nicht mehr ausgeführt. Die Produktivumgebung wird so vor einem Ausrollen einer unreifen Version geschützt.

Skript-Ausführung

Eine wesentliche Fähigkeit, Aufgaben innerhalb einer Deployment-Pipeline durchführen zu können, ist die Unterstützung anderer Werkzeuge, Systeme oder Programme, die die benötigten Fähigkeiten besitzen. Dies kann durch die Möglichkeit realisiert werden, auf der Systemkonsole Kommandos abzusetzen oder durch eine Erweiterung der Anwendung selbst. In *Go* können das Build-Tool Ant, NAnt und Rake genutzt werden, um Aufgaben wie Build, Test oder Deployment zu realisieren. Zudem besteht bei allen drei Werkzeugen die Möglichkeit, Kommandos auf der Systemkonsole abzusetzen. *Dreadnot* und *Deployinator* werden mittels Java-Script bzw. Ruby konfiguriert. Hier besteht die unmittelbare Möglichkeit, die vorbereitenden Vorgänge für ein Deployment direkt in der Konfiguration zu organisieren.

5.3.2. Staging

Versionskontrolle

Dreadnot bedingt Git. Ohne ein Git-Repository kann *Dreadnot* nicht ausgeführt werden. Um die Funktionalität von *Dreadnot* auch im Zusammenhang mit einer anderen Versionsverwaltung zu ermöglichen, müsste *Dreadnot* im Kern verändert werden. Dies ist

zwar durch die Qpen-Source-Lizenz möglich, jedoch mit Aufwänden für die Anpassung verbunden.

Deployinator unterstützt Subversion, in dem einige Methoden für den Zugriff auf Subversion bereitgestellt werden. Diese müssen allerdings nicht Verwendung finden. In der Konfiguration der Stacks können verschiedene Typen von Versionsverwaltungssystemen eingebunden und genutzt werden, sofern es für diese eine Programmbibliothek für das genutzte Betriebssystem gibt und das Programm über die Konsole ausgeführt werden kann.

Go unterstützt die Versionsverwaltungssysteme Apache Subversion, Git, Mercurial, Perforce und den Microsoft Team-Foundation Server. Andere Verwaltungssysteme können nicht genutzt werden, sofern der Prozess durch eine Änderung der Code-Basis automatisch getriggert werden soll.

Artefakt-Repository

Für *Dreadnot* sollte Git zur Verwaltung von Artefakten genutzt werden, wie schon im Zusammenhang mit dem Quality Gate beschrieben wurde. Artefakte in Form der Log-Dateien werden im Arbeitsverzeichnis von *Dreadnot* gespeichert. *Deployinator* sieht für Log-Dateien Ähnliches vor. Zu deployende Versionen können aus verschiedenen Systemen geladen werden. Eine Möglichkeit, Artifactory sowie das lokale Dateisystem zu nutzen wurde bereits in Abschnitt 5.2.2 beschrieben. *Go* stellt hier ein eigenes Repository für Artefakte bereit. Artefakte, die bei der Ausführung einer Stage entstehen, können dort zentral abgelegt und in anderen Stages oder Pipelines weiterverwendet werden. Zudem bietet *Go* eine REST-Schnittstelle und ermöglicht so die Interoperabilität mit anderen Systemen.

5.3.3. Feedback

Dreadnot und *Deployinator* geben die Ergebnisse einer Aktivität in eine Log-Datei aus, welche auf der Oberfläche angezeigt wird. Durchgeführte Deployments werden in einer Liste aufgeführt. *Dreadnot* zeigt das Ergebnis eines Durchlaufs mit *success* oder *fail* an. Bei *Deployinator* kann dies durch die Konfiguration im Stack-Modul bestimmt werden. Elemente wie das Zeitverhalten des Deployments können für *Deployinator* und *Dreadnot* beim Aufrufen des Deployments gemessen und in die Log-Datei ausgegeben werden, welche in der Oberfläche dargestellt wird. Welche Version in welcher Umgebung ausgeführt wird, ist in der Oberfläche ersichtlich. *Dreadnot* als auch *Deployinator* erwarten die Bereitstellung dieser Informationen durch das Stack-Modul.

Go zeigt alle derzeit ausgeführten Aktivitäten auf der Startseite an. Kam es in einer

Phase der Deployment-Pipeline zu einem Fehler, wird der betroffene Task rot markiert. Die Log-Datei des Tasks, die in der Detailansicht angezeigt wird, listet alle Ausgaben, die während der Bearbeitung auf Konsole ausgegeben wurden, auf. Der Zeitverbrauch für die Abarbeitung einer Stage wird in einem Diagramm aufgetragen. Im Zusammenhang mit den Funktionalitäten eines CI-Systems können in *Go* die Protokolle der Testläufe eingebunden werden. Die Möglichkeit, Testprotokolle bei *Dreadnot* und *Deployinator* anzeigen zu lassen, ist nicht gegeben.

Alle drei System informieren einen bestimmbaren Nutzerkreis über E-Mail und IRC. Fehlerzustände und Abbrüche werden über diese Systeme kommuniziert.

5.3.4. Konfiguration

Einstellungen des Systems sollten gesichert werden können und zu einem späteren Zeitpunkt wiederherstellbar sein. Zudem muss klar sein, welche Parameter welche Auswirkungen haben und welche Möglichkeiten bestehen das System zu konfigurieren.

Dreadnot und *Deployinator* werden auf ähnliche Weise konfiguriert. So müssen bei beiden Werkzeugen Stacks implementiert werden, die der Schnittstellenbeschreibung des Systems genügt. Innerhalb der Stacks sind Methoden bzw. Funktionen zu implementieren, die vom Hauptsystem aufgerufen werden, um notwendige Informationen wie z. B. die Bezeichnung des Stacks, zu letzte deployte Version oder, wie bei *Dreadnot*, die URL zum Git-Repository bereitzustellen. Eine Konfiguration über die Oberfläche ist nicht möglich. Es empfiehlt sich, die Stacks in einer Versionsverwaltung wie Subversion oder Git zu halten. Vor jedem Start der Anwendung können dann die lokalen Dateien aktualisiert werden und *Dreadnot* bzw. *Deployinator* starten mit der aktuellen Konfiguration.

Go bietet den Ansatz, die Konfiguration des Systems in Dateien zu halten. Dabei nutzt *Go* ein XML-Dokument, welches auch der Versionsverwaltung unterliegen sollte. Weiterhin besteht aber die Möglichkeit, *Go* durch das integrierte Admin-Modul über die Oberfläche zu konfigurieren. Hier können alle Informationen schrittweise zusammengetragen werden. Zudem bietet *Go* dem Nutzer Hilfefunktion für die korrekte Konfiguration der Felder an. Über die Oberfläche veränderte Konfigurationseinstellungen werden in die interne XML-Datei übertragen. Über den Admin-Bereich lässt sich diese jederzeit abrufen und sichern. Die Möglichkeit, das System über die Oberfläche zu konfigurieren, fehlt *Dreadnot* und *Deployinator*.

Alle drei System bieten die Möglichkeit, die Systemkonfiguration zu sichern und wiederherstellen zu können. Welche Parameter bei *Dreadnot* und *Deployinator* konfigurierbar sind und wie und wo diese zu konfigurieren sind, ist nur knapp und unvollständig über das Readme-File des Projektes auf Github dokumentiert. Abhilfe können aber die Beispielstacks geben, an denen sich bei der ersten Konfiguration eines Stacks orientiert

werden kann. Mehr Informationen über die Systemkonfiguration können nur durch Inspektion des Quellcodes erlangt werden. Bei *Go* sind alle Parameter und Möglichkeiten ausführlich dokumentiert. Zudem sind alle Einstellungen des Systems sowie die Konfiguration der Deployment-Pipeline über die Oberfläche möglich. Durch die Eingabefelder als auch durch eine Hilfefunktion ist ersichtlich, was eine Einstellung bewirkt und welche Möglichkeiten es gibt.

5.3.5. Deployment

Das Deployment einer Anwendung ist im Gegensatz zu anderen Phasen der Deployment-Pipeline sehr individuell. Dabei hängt es von der Art der Anwendung, von der Ausführungsumgebung, von der genutzten Sprache ab, ob sie interpretiert oder kompiliert wird und auch von den vertraglichen Anforderungen, die sich aus einer Kunden-Lieferanten-Beziehung ergeben können. Eine universelle Lösung, das Deployment zu organisieren, kann daher keine Werkzeuge anbieten. Für bestimmte Konstellationen im Bereich der Web-Anwendungen haben sich spezielle Werkzeuge etabliert, welche die Verwaltung einer verteilten Infrastruktur erleichtern.

Deployinator und *Dreadnot* bieten daher keine tiefere Unterstützung für ein Deployment an, als andere Werkzeuge oder Shell-Skripte anstoßen zu können. *Dreadnot* bietet noch ein Modul, um Knife einfacher ansprechen zu können und die geänderte Konfiguration der Ausführungsumgebung an einen Chef-Server zu übertragen. *Deployinator* und *Go* können diese Werkzeuge aber auch genauso einbinden, in dem die Kommandozeile der Systemkonsole angesteuert werden kann.

Go bietet noch eine weitere Möglichkeit das Ausführungssystem zu konfigurieren und ein Deployment durchzuführen. Das Konzept der Agents eröffnet die Möglichkeit, auch lokale Änderungen an der Infrastruktur vorzunehmen und benötigte Komponenten zu installieren. Dieses Konzept eignet sich gut für die Testumgebung. Die Verwendung der Agents für die Produktivumgebung scheint aber nicht geeignet. Ein Agent ist nur für die Durchführung eines Jobs zeitlich an die Pipeline gebunden. Nach einem Deployment steht dieser dem *Go*-Server für die Vermittlung weiterer Aufgaben zur Verfügung. Zu dem ist die Anzahl der Agents begrenzt. Für die Testdurchführung ergibt sich aus diesem Konzept jedoch eine leicht umzusetzende Alternative zu Chef und Knife.

5.4. Zusammenfassung

Die eingehende Einordnung der zu untersuchenden Werkzeuge zeigte wesentliche Unterschiede im Ansatz und in der Zielausrichtung dieser. Während *Go* ein vollständiges

CI-System bietet, mit dem sich auch die Acceptance-Stage und User-Acceptance-Stage modellieren lassen, stellen *Deployinator* und *Dreadnot* die Ergänzung eines solchen Systems dar. Da adesso bereits ein CI-System betreibt, stellen *Deployinator* und *Dreadnot* daher die generelle Möglichkeit einer Ergänzung des bestehenden Systems dar.

Trotz der unterschiedlichen Ansätze konnten Beurteilungskriterien gefunden werden, die eine Betrachtung der Werkzeuge zulassen. Diese Kriterien waren durch die Anforderungen an Continuous-Delivery, der Benutzbarkeit und der Wartbarkeit ausgerichtet.

Die Anforderungen aus Continuous-Delivery, die ausgewählt wurden um eine Beurteilung der betrachteten Werkzeuge zuzulassen, konzentrierten sich auf die Funktionalität der Prozesssteuerung einer Deployment-Pipeline. Für *Dreadnot* und *Deployinator* bedeutet diese, die Fähigkeit ein Deployment anstoßen und mit anderen Systemen zusammenarbeiten zu können. Für *Go* als CI-System kamen Build-Prozess und Testdurchführung hinzu.

Dreadnot scheint für eine Verwendung bei adesso derzeit weniger geeignet, da es durch die Voraussetzung des Git-Repository nicht in die derzeitige Infrastruktur passt. Es würden zu viele Anpassungen des Quellcodes durchgeführt werden müssen, um diese Anpassung zu erreichen. Für *Go* als auch *Deployinator* gibt es bei adesso Integrations- als auch Verwendungsmöglichkeiten. Beide Ansätze dieser Werkzeuge werden im nachfolgenden Abschnitt noch einmal aufgegriffen, um zwei Verwendungsszenarien darstellen zu können.

Die Untersuchung des Lieferprozesses bei adesso hat gezeigt, dass Continuous-Integration durch den Jenkins-Server bereits auf einer breiten Ebene betrieben wird. Allerdings ist längst nicht jedes Projekt Teil dieser Umgebung. Hier kommen die unterschiedlichen Bedingungen der Projekte zum tragen. Eine Abhängigkeit ist z. B. die Rolle, in der adesso als Dienstleiter auftritt. Als Dienstleister, ist es für adesso oft nicht möglich, eine Integration der Projekte in die eigene Systemlandschaft vorzunehmen. Dies tritt besonders dann auf, wenn die Entwicklungsarbeit in der Umgebung des Kunden durchgeführt wird.

In einer derartigen Projektsituation kann *Go* eine Chance darstellen, ein CI-System nach projektspezifischen Kriterien mit einer kleinen Infrastruktur aufzubauen. Die Community-Version von *Go* mit einem *Go*-Server, speziell für diese Projekte, kann bis zu drei Agents anbinden. Die Agents könnten dann auf der Testumgebung laufen. Dem Testteam des Kunden wird so ein zusätzlicher Nutzen gegeben, sich selbst, die zu testenden Versionen der Anwendung auf die Testumgebung zu deployen und manuelle und explorative Tests durchzuführen.

Für Projekte, die auf dem bestehenden CI-System von adesso verwaltet werden, stellt *Deployinator* eine Erweiterung des Systems um die Funktionalität des automatisierten

Deployments dar. Skripte, mit denen bisher das Ausrollen einer neuen Version in die Testumgebung organisiert wird, kann dann über die Oberfläche von *Deployinator* eingebunden und ausgeführt werden. Für Projekte, bei denen adesso direkt in die Produktivumgebung des Kunden ausliefern kann, sollte das Deployment analog zur Testumgebung angestoßen werden können. Selbst wenn keine Auslieferung der Anwendung nach jeder Änderung der Quellcode-Basis vereinbart ist, stellen diese Werkzeuge die Möglichkeit bereit, ein Liefersystem auf einfache Weise aktivieren zu können.

6. Szenarien eines Auslieferungsprozesses

Nach dem *Deployinator* und *Go* als mögliche Werkzeuge für die Prozessteuerung einer Deployment-Pipeline bei adesso eingesetzt werden könnten, soll dieser Abschnitt zwei mögliche Szenarien für einen solchen Einsatz näher erörtern. Der kritische Bereich, die technische Umsetzung des Deployments in die Test- und Produktivumgebung, wird hier aber nur abstrakt angerissen. Der Fokus soll hier vielmehr auf der Realisierung des Prozessablaufs und der Integration in die bestehende Umgebung liegen.

6.1. Szenario 1

6.1.1. Beschreibung und Projektaufbau

Szenario 1 unterstellt ein internes Software-Projekt. Entwickelt wird ein Portal für Mitarbeiter, in dem ein hier nicht näher definierter Service bereitgestellt wird. Das Portal wird als Web-Anwendung entwickelt. Es ist zu erwarten, dass in den nächsten Monaten und Jahren eine kontinuierliche, wenn auch sporadische Weiterentwicklung betrieben wird. Je nach freien Kapazitäten werden Mitarbeiter und Studenten im Rahmen der Ausbildung die Anwendung weiterentwickeln.

Die Implementierung einer Deployment-Pipeline erscheint für dieses Szenario als geeignet. Die wechselnden Teammitglieder und sporadischen Wartungs- und Entwicklungsarbeiten führen zu einem Know-how Verlust von Build- und Release-Prozess. Continuous-Delivery kann hier mit einer Deployment-Pipeline den einmal aufgebauten Lieferprozess sicher und verlässlich gestalten. Neue Teammitglieder benötigen für ein Deployment von Änderungen zunächst keine tieferen Kenntnisse über den Lieferprozess. Zudem gibt die Deployment-Pipeline durch Quality-Gates die Sicherheit, nur geprüfte Versionen in die Produktion zu bringen. Dieser Prozess läuft für ein normales Release in der gleichen Qualität ab, wie für das kurzfristige Beseitigen eines kritischen Fehlers.

Technisch betrachtet ist das System von Szenario 1 eine JEE-Anwendung, welche als einzelne Instanz auf einem virtuellen System ausgeführt wird. Die Anwendung wird mit

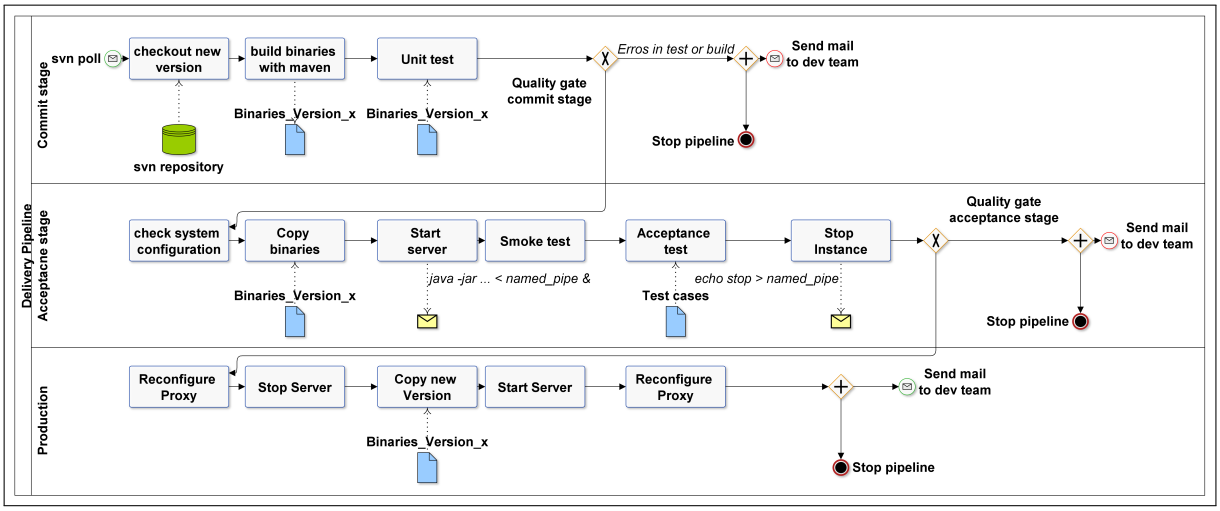


Abbildung 6.1.: Deployment Pipeline für Szenario+1

einem eingebetteten Jetty-Server ausgeliefert, der durch die Main-Methode gestartet wird. Die Konfigurationsdatei des Systems wird beim Start der Anwendung über den Klassenpfad initiiert. Ein Apache 2 wird als Forward-Proxy eingesetzt. Eine Datenbank wird hier nicht benötigt, da es nur wenig Daten gibt, die dauerhaft gehalten werden. Persistenz erhält das System durch eine einfache Textdatei, in der die benötigten Daten mit Hilfe der JavaScript Object-Notation[1] (JSON) gespeichert werden.

Als Versionskontrollsystem wird Subversion eingesetzt. Alle Änderungen an der Quellcode-Basis werden dort verwaltet. Die Umsetzung der Deployment-Pipeline erfolgt auf der Basis von *Go*. Der Prozess von Continuous-Integration und das anschließende Deployment in Test- und Produktivumgebung werden von *Go* gesteuert.

Dem Projekt stehen zwei Linux-Server zur Verfügung. Auf einem System wird das Produktivsystem ausgeführt, auf dem anderen die Testumgebung und die benötigten *Go*-Instanzen für die Deployment-Pipeline. Auf Test- und Produktivsystem wird die Laufzeitumgebung von Java als Vorbedingung installiert. Das Produktivsystem benötigt zusätzlich das Java-Development-Kit, Maven zur Organisation des Build-Prozesses und einen Subversion-Client.

6.1.2. Ablauf der Deployment-Pipeline

Commit-Stage

Ein Entwickler übermittelt die durchgeführten Änderungen am Quellcode der Versionsverwaltung (svn commit). In der Versionsverwaltung wird die Revisionsnummer erhöht. Die letzte Version, die für eine Instanz der Pipeline genutzt wurde, ist *Go* bekannt.

[1] Weite Informationen zu JSON: `http://www.json.org/`

Durch eine Abfrage der Versionsverwaltung prüft *Go* in einem festen Zeitintervall, ob es Änderungen der Revisionsnummer gibt (svn poll). Ist dem so, wird eine neue Instanz der Pipeline mit der aktuellen Revisionsnummer erzeugt.

Obwohl es in *Go* möglich ist, in einer Stage Aufgaben durch Jobs zu parallelisieren, gibt es in dieser Pipeline hierfür keinen Bedarf. Unterhalb der Commit-Stage befindet sich deshalb nur ein einziger Job. Ein freier Agent wird mit der Job-Konfiguration initialisiert. Der Agent beginnt mit dem Checkout der aktuellen Quellcode-Basis aus Subversion heraus. Alle Dateien werden im Arbeitsverzeichnis des Agents unter dem Pfad `/pipelines/<PIPELINE_NAME>/` abgelegt. Anschließend wird Maven über den *Custom Command* mit `mvn install assembly:single` angestoßen. Maven lädt dabei alle Pakete, von denen die Anwendung abhängt, aus dem zentralen Repository, kompiliert den Quellcode, führt die Komponententests durch und pakt alle Abhängigkeiten in ein einziges Jar-File zusammen.

Ein Nachweis der technischen Korrektheit der Version wird dadurch erbracht, dass die Anwendung kompiliert und alle Komponententests durchlaufen. Die Commit-Stage beendet erfolgreich und das erste Quality-Gate wurde passiert.

Acceptance-Stage

Vor der Ausführung der Akzeptanztests wird die Umgebung auf ihre Konfiguration hin untersucht. Jede Änderung der Infrastruktur wird in einem Update-Skript festgehalten. Ein Update-Skript ist durch die Versionsnummern gekennzeichnet, die das System von einer älteren auf eine neuere Version heben. Folgendes Muster sei als Beispiel gegeben: `system_update_0005_0006` hebt die Systemumgebung von Version 5 auf 6. Die letzte Versionsnummer wird auf dem System in einer Datei gehalten. Das Update-Skript enthält alle Änderungen von einer Infrastruktur-Version auf die nächste. Die aktuelle Version wird ausgelesen und geprüft, ob ein Update-Skript für diese Version existiert. Nach der Änderung durch das Skript wird die aktuelle Versionsnummer des Scripts in eine Datei geschrieben, damit spätere Änderungen möglich bleiben.

Da das Testsystem und *Go* sich eine Umgebung teilen, können das neue Jar-File sowie das Startskript direkt in das Ausführungsverzeichnis kopiert werden. Die Anwendung wird von *Go* gestartet. Hierfür wird der *Custom Command* genutzt, um das Startscript aufzurufen und ausführen zu lassen. Alle Ausgaben der Systemkonsole werden an das Log-File weitergeleitet, dieses kann später über die Oberfläche von *Go* eingesehen werden.

Nach dem Start wird ein Smoke-Test durchgeführt, welcher prüft, ob das System grundlegend einsatzbereit ist. Hierfür wird das Kommandozeilenwerkzeug *curl* genutzt, um ein *GET* der Status-Seite über HTTP auszuführen. Als Antwort wird ein Response-Status-Code 200 im HTTP-Header erwartet.

Das System ist nun bereit, auf die Erfüllung aller funktionalen und nicht-funktionalen Anforderungen hin untersucht zu werden. Die Testfälle werden vom Test-Framework geladen und ausgeführt. Reagiert das System wie verlangt, kann die Acceptance-Stage abgeschlossen werden.

Die Anwendung öffnet beim Start einen Eingabekanal auf der Systemkonsole. Über diesen kann die Anwendung gestoppt und Status-Informationen abgerufen werden. Bei Start des Systems wurde der Eingabekanal auf eine Named-Pipe gelegt und die Anwendung im Hintergrund gestartet. Über die Named-Pipe kann *Go* der Anwendung nun das Stopp-Kommando geben. Die Anwendung beendet den Server und schließt. Anschließend wird ausgewertet, ob Fehlermeldungen während der Tests in das Error-Log geschrieben wurden.

An dieser Stelle kann festgestellt werden, dass die neue Version der Anwendungen auf ihre technische Korrektheit und Erfüllung der Akzeptanzkriterien hin überprüft wurde. Werden bestimmte Akzeptanzkriterien nicht erfüllt, verhindert dies die Auslieferung in die Produktivumgebung. Auch die Auswertung der Fehlerprotokolle spielt hier eine Rolle. Gibt es Einträge im Fehler-Log der Anwendung oder in der Fehlerausgabe auf der Systemkonsole, verhindert dies die Auslieferung. *Go* ist so konfiguriert, dass es dann eine Nachricht an das Entwicklerteam sendet.

Der Entwickler, der die letzte Änderung vorgenommen hat, ist in der Verantwortung, die Fehlerprotokolle durchzusehen. Zusammen mit dem Projektleiter wird entschieden, ob das Release in die Produktion dennoch stattfinden sollte. Dies geschieht unter Abwägung aller Risiken.

Production

Wenn alle Tests positiv verlaufen sind, wird die Anwendung in die Produktivumgebung ausgerollt. Je nach Art der Umgebung stellt dies einen mehr oder minder komplexen Prozess dar. Das in diesem Szenario genutzte System ist kein kritischer Geschäftsprozess mit einer sporadischen Nutzung für spezielle Aufgaben. Um einen direkten Systemzugriff zu verhindern, wurde ein Apache HTTP-Server vorgeschaltet, der Anfragen an das System weiterleitet. Der HTTP-Server ist so konfiguriert, dass dieser mit dem Parameter `-DClosedForNow` neu gestartet wird und anschließend Anfragen mit einer Wartungsseite beantwortet. Damit kann das zu aktualisierende System heruntergefahren werden. Der Zugriff auf das Produktivsystem erfolgt über SSH[2]. Ist das System heruntergefahren, wird die neue Version eingespielt. Die alte Version wurde zuvor in das Verzeichnis `Recover` kopiert, um im Notfall das Recover-Skript starten zu können und die vorhergehende Version wiederherzustellen.

[2] SSH: *S*ecure *Sh*ell ermöglicht eine Verbindung zur Kommandozeile entfernter Systeme.

6.2. Szenario 2

6.2.1. Beschreibung und Projektaufbau

Szenario 2 nimmt ein Kundenprojekt als Ausgangspunkt. Dabei handelt es sich um eine Web-Anwendung für Mitarbeiter einer Rückversicherung, die diese nutzen, um Risikokalkulationen durchführen zu können. Das System befindet sich in einer kontinuierlichen, wenn auch nicht ununterbrochenen, Weiterentwicklung. Continuous-Delivery und eine Deployment-Pipeline sollen den Lieferprozess erheblich verkürzen und den Nutzern alle zwei Wochen neu entwickelte Funktionalitäten bereitstellen können. Zudem sollen kurzfristige Fehlerbeseitigungen (Hot-Fix) in die Produktivumgebung eingespielt werden können.

Als Ausführungsumgebung der Anwendung wird ein JBoss Application Server verwendet sowie das JSF 2.0 Framework eingesetzt. Die Testumgebung besteht aus einer einzelnen Serverinstanz mit angegliederter Testdatenbank. Bei der Produktivumgebung werden vier Instanzen zu einem Cluster verbunden und HTTP-Server als Load-Balancer vorgeschaltet.

Die bestehenden Maßnahmen der Qualitätssicherung sind schon als automatisierte Komponenten und Akzeptanztests realisiert worden. Der Build-Prozess wird auf dem CI-System von adesso, dem Jenkins CI-Server, durchgeführt. Jede Änderung der Quellcode-Basis in der Versionsverwaltung (SVN) führt zum Anstoßen des Build-Prozesses und zur Ausführung der Komponententests. Akzeptanztests, die in automatisierter Form vorliegen, wurden bisher manuell angestoßen und gegen das Testsystem ausgeführt. Auf dem Testsystem existiert ein Hilfsskript. Dieses stoppt den Web-Server, kopiert das neue WAR-File vom Arbeitsverzeichnis des Nutzers in das des Servers und startet dieses neu. Voraussetzung ist das Laden der aktuellen Version aus dem unternehmensinternen Repository. Dieser Schritt wurde so weit automatisiert, als das ein anderes Skript nur noch mit der aktuellen Versionsnummer aufgerufen werden muss. Das Laden der aktuellen Version übernimmt dann jenes Skript. All diese Aufgaben müssen aber manuell aufgerufen werden und benötigen die Administrationsrechte für den Testserver. Dies gilt analog auch für das Produktivsystem.

Ein Teil der Deployment-Pipeline wird bereits durch das CI-System umgesetzt. Build-Prozess und Test der Komponenten können dort verbleiben. Mit *Deployinator* wurde ein Werkzeug evaluiert, welches die bestehenden Ansätze aufgreifen und miteinander verbinden kann. *Deployinator* soll hier genutzt werden, um auf Test- und Produktivumgebung zu deployen und die Erfüllung der funktionalen und nicht-funktionalen Anforderungen zu prüfen.

6.2.2. Aufbau der Deployment-Pipeline

Da die Commit-Stage schon durch das bestehende CI-System abgebildet wird, werden hier nur Acceptance-Stage und Production näher beschrieben. Bei jedem Durchlauf des CI-Systems werden die erstellten Softwarepakete in einem Repository abgelegt.

Deployinator als Self-Service-Portal stellt dem Team die Möglichkeit bereit, die aktuelle Version der Anwendung, deren technische Korrektheit mittels Komponententests untersucht wurde, in die Testumgebung zu deployen. Die automatisierten Akzeptanztests können anschließend durchlaufen werden. Nach dem eine Version erfolgreich getestet wurde, kann das Deployment in die Produktivumgebung, am Tag des Release, von einem Mitglied des Entwicklungsteams über die Oberfläche von *Deployinator* angestoßen werden.

Deployinator wurde, wie in Abschnitt 5.2.2 beschrieben, um die Möglichkeit erweitert, eine spezifische Version für das Deployment auswählen zu können.

Acceptance-Stage

Ein Mitglied des Entwicklungsteams ruft die Oberfläche von *Deployinator* auf. *Deployinator* ermittelt dabei die im Repository abgelegten Versionen und stellt diese in der Auswahlliste bereit. Soll nicht die zuletzt erstellte Version deployt werden, kann eine andere im Drop-Down-Menü ausgewählt werden. Mit dem Button *Deploy to test* wird der Deployment-Prozess angestoßen. An der Infrastruktur werden während der Entwicklung keine Änderungen erwartet, sodass dieser Schritt im automatisierten Deployment nicht weiter berücksichtigt wird. Da die Anwendung mit einer Datenbank zusammenarbeitet und hier, im Gegensatz zur Infrastruktur, Änderungen zu erwarten sind, muss der Deployment-Prozess dieses berücksichtigen können. Alle Schritte, die während des Deployments durchgeführt werden, sind im Stack-Modul von *Deployinator* definiert. Dabei werden die hinterlegten Skripte auf der Systemkonsole über `run_cmd` aufgerufen.

Der erste Schritt des Deployments ist die Aktualisierung der Deployment-Skripte, welche in der Versionsverwaltung von Subversion in der aktuellen Version hinterlegt sind. Die Änderungen an der Datenbank werden durch ein Update-Skript organisiert. Auch diese werden in der Versionsverwaltung gehalten und gepflegt.

Update und Rollback der Datenbank werden inkrementell durchgeführt. Die Datenbank speichert hierfür die aktuelle Versionsnummer. Ist die Datenbank von einer Version auf die nächste anzuheben, wird ein Update-Skript erstellt, welches die notwendigen Anweisungen enthält. Auch der inverse Vorgang, ein Rollback, muss berücksichtigt werden. Auch hierfür muss ein Skript existieren, welches die inversen Operationen des Update-Skriptes enthält. Soll eine Softwareversion deployt werden, wird geprüft, welche Version

der Datenbank für die Version der Anwendung benötigt wird. Diese Information kann dem Softwarepaket selbst über die Datei `db.info` durch den Parameter `db_version` entnommen werden. Für das Datenbankupdate existiert ein Skript, welches die Notwendigkeit einer Veränderung prüft und diese dann ausführt.

Verlangt die neue Version z. B. Version 6 der Datenbank, diese aber den Stand der Version 4 hat, muss zuerst das Skript `db_update_4_5.sql` ausgeführt werden, um anschließend das Update `db_update_5_6.sql` ausführen zu können. Wird ein Rollback notwendig, so ist z. B. zuerst das Skript `db_rollback_6_5.sql` und dann das Skript `db_rollback_5_4.sql` ausgeführt werden.

Bevor die Datenbank aktualisiert werden kann, ist zu prüfen, ob die Instanz des Testservers aktiv ist. In diesem Fall wird der Server heruntergefahren. Anschließend wird eine Sicherung der Datenbank durchgeführt. Nach dem Update ist eine Überprüfung des Zustandes der Datenbank notwendig. Der Health-Check und die Sicherung sind Teil des Skriptes.

Nachdem die Überprüfung des Zustandes der Datenbank den erfolgreichen Verlauf des Updates bestätigt hat, kann die Software auf dem Testserver aktualisiert werden. Der Server wird wieder hochgefahren. Alle Befehle werden hierzu über SSH ausgeführt. Die Datenbankaktivitäten können z. B. über *SQL*Plus*[3] ausgeführt werden.

Das Testsystem hat nun den notwendigen Zustand, um die neue Version auf die Einhaltung der Anforderungen zu überprüfen. Die automatisierten Akzeptanztests können nun durchgeführt werden. Die hierfür genutzten Testframeworks sind auf dem System, auf dem auch *Deployinator* ausgeführt wird, installiert. Alle Testfälle sind am Anfang der Acceptance-Stage aus der Versionsverwaltung geladen worden.

Die Testprotokolle, die bei einem Durchlauf der Akzeptanztests entstehen, werden gepackt und für eine spätere Auswertung in das Repository des Servers kopiert. Das Testteam erhält die Testprotokolle ferner als Anhang der E-Mail-Nachricht, die zum Abschluss der Phase versandt wird.

Sofern im Testprozess Fehler auftreten, kann dies erfasst und darauf reagiert werden. Laufen alle Tests fehlerfrei durch, wird die Versionsnummer des geprüften Softwarepaketes in einer Liste mit erfolgreich getesteten Versionen eingetragen.

Production

Die Schritte beim Deployment in die Produktivumgebung verlaufen ähnlich wie beim Deployment in die Testumgebung. Gleich sind das Laden der Skripte aus der Versionsverwaltung und das Update der Datenbank.

[3] Weitere Informationen zu SQL*Plus unter: `http://docs.oracle.com/cd/B19306_01/server.102/b14357/toc.htm`

Während des Aktualisierungsvorgangs steht die Anwendung nicht zur Verfügung. Der Proxy-Server leitet alle Anfragen für den Zeitraum der Aktualisierung auf eine Wartungsseite. Dieses läuft dabei genauso ab, wie in Szenario 1 schon beschrieben.

Nachdem der Proxy konfiguriert wurde, können die Web-Server angehalten werden, damit das Datenbankupdate eingespielt werden kann. Ist dies erfolgreich verlaufen, werden die Web-Server aktualisiert und neu gestartet. Der Proxy-Server wird dann wieder normal konfiguriert.

Da die Anwendung nicht bei jeder Änderung, sondern nur alle zwei Wochen, in die Produktivumgebung ausliefert, ist der Deployment-Prozess weniger komplex gestaltet. Dies sollte z. B. für eine Anwendung, die einem stetigen Zugriff unterliegt, nicht der Fall sein. Das gleichzeitige Herunterfahren aller Produktivinstanzen wäre dann keine Option. Das System von Szenario 2 wird aber nicht mehr am Freitag nach 17 Uhr verwendet, sodass der Aktualisierungsvorgang danach auch mit geringerer Komplexität durchgeführt werden kann.

Alternativ könnte dies in anderen Anwendungssituationen so gestaltet sein, dass die Datenbank vor einem Update auf ein Zweitsystem gespiegelt wird. Alle Änderungen würden dann auf dem Zweitsystem ausgeführt. Das Update der Anwendung wird dann inkrementell auf allen Servern durchgeführt. Die neue Anwendung arbeitet mit der neuen Datenbankversion, der alte Softwarestand mit der alten Version der Datenbank. Nach dem alle Server aktualisiert worden sind, müssen die Änderungen, die in der Zeitspanne des Updates durchgeführt worden sind, auf die neue Datenbank übertragen werden. Eine weitere Voraussetzung ist der Austausch von Informationen aktiver Nutzer-Sessions zwischen den Servern, damit ein Herunterfahren einer Instanz nicht zu einem Informationsverlust führt.

6.3. Zusammenfassung

Grundlage der hier beschriebenen Szenarien waren Projekte, mit denen der Verfasser der Arbeit während seiner Tätigkeit im Unternehmen in Berührung gekommen war. Die Szenarien wurden so angepasst, dass diese als Ausgangspunkt für eine mögliche Umsetzung von Continuous-Delivery durch die untersuchten Werkzeuge aufzuzeigen konnten.

Beide Szenarien stellen Möglichkeiten der Verwendbarkeit der untersuchten Werkzeuge dar und sind noch keine optimalen Lösungen. Diese können nur durch das Zusammenwirken von IT-Betrieb, Entwicklungsteam und Kundenvertretern gemeinsam erarbeitet werden.

Die Umsetzung einer Deployment-Pipeline ist mit *Go* einfacher, da Quality-Gates einfacher in den Prozess einzurichten sind. Kann das CI-System aus bestimmten Gründen

nicht genutzt werden, bietet sich *Go* als eine Alternative an, alle Elemente von Continuous-Delivery in einer Umgebung konfigurieren und steuern zu können. *Deployinator* hingegen ergänzt das CI-System. Die Umsetzung einer Acceptance-Stage ist jedoch nicht sehr komfortabel. In *Deployinator* können keine Testprotokolle angezeigt werden. Das System muss um eine Liste geprüfter Versionen ergänzt werden, um ermitteln zu können, welche Version aus dem Repository die notwendige Reife besitzt, um in die Produktivumgebung ausgeliefert werden zu können.

7. Fazit

Im Rahmen dieser Arbeit sollte sich allgemein mit dem Themenkomplex des Continuous-Delivery auseinandergesetzt werden. Speziell standen in diesem Zusammenhang drei als interessant eingestufte Werkzeuge im Fokus, die näher untersucht werden sollten. Im Vorfeld der Untersuchung wurde auf die Beziehung zwischen Continuous-Delivery und DevOps eingegangen. Anschließend wurden die Konzepte von Continuous-Delivery, wie sie Humble und Farley vorschlagen, beleuchtet. Dies war einführend notwendig, um für adesso der Frage nach der Funktionsweise sowie dem jeweilig verfolgten Ansatz der zu untersuchenden Werkzeuge nachgehen zu können. Zudem sollte nach möglichen Synergien für den derzeitigen Lieferprozess gesucht werden.

7.1. Erkenntnisse

Mit einem Fragebogen an das unternehmenseigene CI-Team wurden die etablierten Lieferstrukturen ermittelt. Eine Erkenntnis für die Umsetzungsmöglichkeiten von Continuous-Delivery, welche aus dem Fragebogen abgeleitet werden konnte, war die der ungleichen Projektstruktur bei adesso. Diese kommt durch die individuelle Softwareentwicklung und deren starke Ausrichtung an das Projektgeschäft zustande. Dabei wird für adesso die Umsetzung eines allgemeingültigen Lieferprozesses verhindert, welcher von allen Projekten auf gleiche Weise genutzt werden kann. Vielmehr sind projektspezifische Faktoren bei der Einrichtung des Lieferprozesses heranzuziehen, die von Projekt zu Projekt variieren können. adesso unterstützt die Projektteams durch einen zentralen Build-Prozess bei der Realisierung von Continuous-Integration. Dies stellt eine zentrale Voraussetzung für die Umsetzung von Continuous-Delivery dar.

Im Verlauf der Arbeit zeigte sich, das Continuous-Delivery nicht durch eine bestimmte Technologie oder ein einzelnes Werkzeug lösbar ist. Es entstand die Annahme, dass die zu untersuchenden Werkzeuge nur die Lösung eines Teilproblems von Continuous-Delivery ermöglichen können. Diese Annahme bestätigte sich bei der Evaluierung. Das Kernproblem von Continuous-Delivery, ein automatisiertes Deployment in verschiedene Umgebungen zu ermöglichen, lösen die Werkzeuge nicht, ermöglichen es aber, die Implementierung eins Lieferprozesses in Bezug auf *Go* zu organisieren bzw., in Bezug auf

Dreadnot und *Deployinator*, anstoßen zu können.

Zum Ende der Arbeit trat die Erkenntnis hervor, dass *Go*, *Dreadnot* und *Deployinator* Continuous-Delivery durch die beschriebenen Ansätze unterstützen, diese aber nicht direkt als Werkzeuge für Continuous-Delivery angesehen werden können, da sie nur eine Kategorie von notwendigen Funktionen und Eigenschaften für die Realisierung von Continuous-Delivery abdecken. Continuous-Delivery kann nur aus einer intelligenten und auf den Bedingungen der zu liefernden Software abgestimmten Kombination verschiedener Werkzeuge und Skripte möglich werden. *Das* Werkzeug für Continuous-Delivery kann nicht erwartet werden.

Den Lieferprozess auf einfache Weise starten zu können ist ein wichtiger Baustein im automatisierten Lieferprozess, hilft aber nicht bei der Lösung der kritischen Fragen von Continuous-Delivery. Dies sind z. B. die sichere und verlustfreie Aktualisierung der Datenbank einer Web-Anwendung bei laufendem Betrieb oder das inkrementelle automatische Update einer Web-Anwendung, ohne das dies eine Auswirkung auf die aktiven Nutzer hat.

Vielmehr unterstützen die untersuchten Werkzeuge die Verwirklichung von DevOps durch die Möglichkeit, eine aus Skripten und Werkzeugketten bestehende automatisierte Deployment-Pipeline anstoßen zu können. IT-Betrieb und Entwicklerteam rücken näher zusammen, sodass der zu realisierende Deployment-Prozess nur in einer engen Zusammenarbeit möglich wird.

Go bietet hier als Teillösung für Continuous-Delivery die am weitesten gehende Implementierung einer Steuerung des Lieferprozesses an. Basierend auf Tasks, Jobs und Stages sowie der einfachen Umsetzung von Quality-Gates können alle Prozesse der Deployment-Pipeline in *Go* organisiert werden. *Dreadnot* und *Deployinator* sind als reine Push-Button-Lösungen einfacher aufgebaut, können aber an bestehende Prozesse des CI-Systems angekoppelt werden.

In der Kombination der Erkenntnisse des Fragebogens und der Untersuchung der Werkzeuge, wurden zwei mögliche Szenarien vorgestellt, wie diese Lösungen in zukünftigen Projekten zum Tragen kommen könnten. Für Projekte, die nicht auf die interne CI-Infrastruktur von adesso zurückgreifen können, ist *Go* ein gutes und schnell verfügbares Werkzeug, um Continuous-Integration und ein Deployment-Pipeline realisieren zu können. Für Projekte, die das CI-System bereits nutzen, automatisierte Akzeptanztests implementiert haben und mit Deployment-Skripten die Umgebung konfigurieren und ausliefern, ist *Deployinator* das richtige Werkzeug, diese Elemente verbinden zu können.

7.2. Ausblick und Anknüpfungspunkte

Auf die dringenden Fragestellungen von Continuous-Delivery, verschiedene Systeme bei laufendem Betrieb aktualisieren zu können, konnte in dieser Arbeit nicht eingegangen werden. Es blieb bei der Betrachtung von Werkzeugen für die Prozessteuerung bzw. für die Aktivierung schon hinterlegter automatisierter Auslieferungsprozesse.

In diesem Zusammenhang ergeben sich weitere interessante Anknüpfungspunkte für die Realisierung von Continuous-Delivery:

- Die Erprobung von *Deployinator* und *Go* in einem realen Projektumfeld. Für eine Vergleichbarkeit der Ansätze sollte dies an einem zu bestehenden Projekt umgesetzt werden können. Dieses sollte dann schon feste Abläufe des Lieferprozesses definiert haben. Daraus könnten weitere Schlussfolgerungen gezogen werden, wie sich die so implementierte Deployment-Pipeline auf Produktivität des Entwicklungs- und Testteam auswirken kann. Zudem sind in einem realen Projektumfeld auch die Bedingungen des Produktivsystems bekannt, die in den beschriebenen Szenarien nur angerissen werden konnten.

- Die Erarbeitung von Ansätzen, *Deployinator* und *Go* mit Werkzeugen wie Chef zu verbinden, um Infrastrukturen automatisch zu konfigurieren. Dieser Punkt konnte in dieser Arbeit nur in einer groben und abstrakten Form angerissen werden. Es sollten hier aber die Stärken und Schwächen einer solchen Verbindung noch näher beleuchtet werden. Continuous-Delivery besitzt an dieser Stelle möglicherweise noch Schwachstellen, die eine Deployment-Pipeline bzw. die mit ihr verbundenen Systeme instabil werden lässt. Dies sollte noch näher untersucht werden.

- Eine mögliche Verbesserung könnte auch das Ausnutzen von Infrastructure-As-A-Service in Verbindung mit einer Umsetzung von Private- bzw. Hybrid-Cloud im Unternehmen bedeuten. Hier wurden schon in Form einer Master-Thesis von Marko Salchow eine entsprechende Beschreibung geliefert, wie der Aufbau einer Hybrid-Cloud mit OpenStack[1] und unter Zunahme einer Cloud-API realisiert werden könnte. Salchow kam zum Schluss dass die Nutzung von Abstraktions-APIs gewinnbringend sein kann, obwohl in einem komplexen Szenario eine Umsetzung mit OpenStack nur durch einen erhöhten Aufwand verwirklicht werden kann.[2] Von seiner Arbeit ausgehend, könnten aber vertiefende Szenarien in Zusammenhang mit Continuous-Delivery entwickelt werden. An deren Ende könnte die Realisierung einer Deployment-Pipeline mit der automatischen Versorgung von Rechnerinstanzen

[1] Weitere Informationen zu OpenStack unter `http://www.openstack.org/`

[2] Vgl. [Sal12]

in einer Private-Cloud stehen. Benötigte Instanzen für die Testdurchführung könnten so auf einfache Weise erzeugt und der Deployment-Pipeline zur Verfügung gestellt werden.

- Ein automatisierter Auslieferungsprozess, der direkt in die Produktivumgebung eines Kunden liefert, muss alle funktionalen und nicht-funktionalen Anforderungen erfüllen können. Der Ansatz von BDD liefert hier einen geeigneten Beitrag, Akzeptanztests schon zu Beginn in die Deployment-Pipeline fest zu implementieren. BDD hat adesso parallel zu Continuous-Delivery in Form einer Master-Thesis evaluieren lassen. Robert Weber kam in seiner Arbeit zur Ansicht, dass BDD zwar einen erhöhten Implementierungsaufwand erfordert, dies aber in Verbindung mit Continuous-Delivery relativiert wird. Der Gesamtaufwand, der durch die erhöhte Häufigkeit des kontinuierlichen Lieferprozesses entsteht, kann durch automatisierte Durchführung von Akzeptanztests deutlich reduziert werden. [3] Dieses sollte bei zukünftigen Projekten Berücksichtigung finden.

- Bei einem automatisierten Auslieferungsverfahren ergibt sich eine weitere Komplexität durch die Aktualisierung des Produktivsystems im laufenden Betrieb, wenn z. B. das Datenbank-Schema aktualisiert werden soll. Ein Datenbank-Update wird in Verbindung mit dokumentenbasierten Datenbanken aber vereinfacht. Zum Thema *Big Data und NoSQL* bietet Robert Zeschkes 2012 verfasste Master-Thesis bei adesso einen Anknüpfungspunkt, der in Verbindung mit Continuous-Delivery noch näher betrachtet werden sollte. Da bei einer dokumentenbasierten Datenbank keine Verwerfungen auftreten, wenn Objekte mit neuen Attributen gespeichert werden, sind diese Systeme durchaus für eine Verwendung im kontinuierlichen Lieferprozess geeignet.[4]

Die Themen Continuous-Deliveryund DevOps, IaaS in einer Hybried-Cloud, BDD und Big Data und NoSQL verbindet adesso unter den Namen *New Shool of IT*. adesso hält hier eine neue strategische Ausrichtung der IT-Branche mit anderen Dienstleistungen und Angeboten für möglich.[5]

[3] Vgl. [Web12]
[4] Vgl. [Zes12]
[5] Vgl. [Wol12]

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel sind angegeben. Die Arbeit hat mit gleichen bzw. in wesentlichen Teilen gleichem Inhalt noch keiner Prüfungsbehörde vorgelegen.

Stralsund, den 12. Oktober 2012

André Gärtner

Literaturverzeichnis

[ade12a] ADESSO AG: *adesso AG - Unternehmen*. WWW. https://www.adesso.de/de/unternehmen/unternehmensprofil/index.html. Version: Aug. 2012

[ade12b] ADESSO AG: *adVANTAGE - Agiles wertorientiertes Vorgehensmodell mit Budgetkontrolle in Softwareentwicklungsprojekten*. WWW. http://www.adesso.de/de/leistungen/softwaredevelopment/advantage/advantage.jsp. Version: Aug. 2012

[ade12c] ADESSO AG: *Branchen - Hohe Branchenkenntnis verbunden mit technologischer Kompetenz*. WWW. http://www.adesso.de/de/branchen/index.html. Version: Aug. 2012

[ade12d] ADESSO AG: *Build Management*. interne Quelle, Juli 2012

[ade12e] ADESSO AG: *Kompetenz - Langjährige Praxis- und Projekterfahrung in der Anwendung von Java und Komponentenmodell JEE*. WWW. https://www.adesso.de/de/technologien/java/kompetenz/index.html. Version: Aug. 2012

[ade12f] ADESSO AG: *Leistungen - Applikationen in der Cloud - Cloudbasierte Softwareentwicklung für mehr Stabilität und Effizienz*. WWW. http://www.adesso.de/de/technologien/cloud_computing/leistungen_2/cloud_computing_1.html. Version: Aug. 2012

[ade12g] ADESSO AG: *Technologien - Technologieführerschaft bedeutet bei adesso vor allem Qualifikation und Erfahrung*. WWW. http://www.adesso.de/de/technologien/index.html. Version: Aug. 2012

[Amb] AMBLER, Scott W.: *Introduction to Test Driven Development (TDD)*. Online. http://www.agiledata.org/essays/tdd.html. – Abgerufen am 01.10.2012

[B+01] BECK, Kent u. a.: *Principle behind the Agile Manifesto*. WWW. http://agilemanifesto.org/principles.html. Version: 2001

[Bal11] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Bd. 3. Spektrum Akademischer Verlag Heidelberg, 2011

[Bro03] Brockmeider, Joe: *Create Debian Linux packages*. WWW. http://www.ibm.com/developerworks/linux/library/l-debpkg/index.html. Version: Juli 2003. – Abgerufen am 19.09.2012

[Böt08] Böttcher, Roland: *IT-Servicemanagement mit ITIL V3*. Heise Zeitschriften Verlag, Hannover, 2008

[dev12] devopsdays.org: *Ghent 2009 program*. WWW. http://www.devopsdays.org/events/2009-ghent/program/. Version: Sep. 2012

[Dic11] Dickerson, Chad: *How does Etsy manage development and operations?* WWW. http://www.quora.com/How-does-Etsy-manage-development-and-operations/answer/Chad-Dickerson?srid=36Z. Version: Feb. 2011

[Duv07] Duvall, Paul M.: *Continuous Integration*. Addison-Wesley, 2007

[Edw10] Edwards, Damon: *DevOps is not a technology problem. DevOps is a business problem.* WWW. http://dev2ops.org/blog/2010/11/7/devops-is-not-a-technology-problem-devops-is-a-business-prob.html. Version: Nov. 2010

[Fit09] Fitz, Timothy: *Continuous Deployment*. WWW. http://timothyfitz.wordpress.com/2009/02/08/continuous-deployment/. Version: Feb. 2009

[Ghe10] Gheorghiu, Grig: *Bootstrapping EC2 instances with Chef*. WWW. http://agiletesting.blogspot.de/2010/07/bootstrapping-ec2-instances-with-chef.html. Version: Juli 2010. – Abgerufen am 22.09.2012

[HC11] Huth, Alexa ; Cebula, James: *The Basics of Cloud Computing*. Online. http://www.us-cert.gov/reading_room/USCERT-CloudComputingHuthCebula.pdf. Version: 2011. – Abgerufen am 28.09.2012

[HF11] Humble, Jez ; Farley, David: *Continuous Delivery*. Upper Saddle River and NJ : Addison-Wesley, 2011

[Hum11] Humble, Jez: *What DevOps Means for Enterprises*. WWW. http://www.agileweboperations.com/what-devops-means-for-enterprises. Version: Jan. 2011

[JFr12] JFrog Inc.: *Artifactory's REST API*. WWW. http://wiki.jfrog.org/confluence/display/RTF/Artifactory%27s+REST+API#ArtifactorysRESTAPI-FolderInfo. Version: Aug. 2012

[Joy12] Joyent Inc.: *Node's goal is to provide an easy way to build scalable network programs*. WWW. http://nodejs.org/about/. Version: Sep. 2012

[Kas10] Kastner, Erik: *Quantum of Deployment*. WWW. http://codeascraft.etsy.com/2010/05/20/quantum-of-deployment/. Version: Mai 2010

[Kas12] Kastner, Erik: *Deployinator Readme*. WWW. https://github.com/etsy/deployinator#readme. Version: Apr. 2012

[Man10] Mandrikov, Evgeny: *Measure Code Coverage by Integration Tests with Sonar*. WWW. http://www.sonarsource.org/measure-code-coverage-by-integration-tests-with-sonar/#more-4229. Version: Okt. 2010

[Mar00] Martin, Robert C.: *Design Principles and Design Patterns*. WWW / PDF. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. Version: Jan. 2000

[MF11] Minick, Eric ; Frederick, Jeffrey: *Enterprise Continuous Delivery Maturity Model*. Whitepaper. http://www.urbancode.com/html/resources/white-papers/Enterprise_Continuous_Delivery_Maturity_Model/. Version: 2011

[Min12] Minick, Eric: *Gartner also observes DevOps mixing with ITIL*. WWW. http://blogs.urbancode.com/uncategorized/gartner-also-observes-devops-mixing-with-itil/. Version: März 2012. – Blog-Eintrag

[Miz] Mizerany, Blake: *Rack Middleware*. WWW. http://www.sinatrarb.com/intro#Rack%20Middleware. – Abgerufen am 23.09.2012

[Neu07] Neukirchen, Christian: *Introducing Rack*. WWW. http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html. Version: Feb. 2007. – Abgerufen am 23.09.2012

[Nor06] North, Dan: *Introducing BDD*. WWW. http://dannorth.net/introducing-bdd/. Version: Mrz. 2006. – Abgerufen am 18.09.2012

[NS10] Nelson-Smith, Stephen: *Just Enough Developed Infrastructure - What Is This Devops Thing, Anyway?* WWW. http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/. Version: Feb. 2010

[Pan10] Pant, Rajiv: *Organizing a Digital Technology Department of Medium Size in a Media Company*. WWW. http://www.rajiv.com/blog/2009/03/17/technology-department/. Version: Sep. 2010

[Pop06] Popp, Gunther: *Konfigurationsmanagement mit Subversion, Ant und Maven*. dpunkt.verlag; Heidelberg, 2006

[Que12] Quera, Pail: *Rackspace Open Sources Dreadnot*. WWW. http://www.rackspace.com/blog/rackspace-open-sources-dreadnot/. Version: Jan. 2012

[Rea10] Read, Chris: *DevOps: State of the Nation*. WWW. http://www.agileweboperations.com/devops-state-of-the-nation-by-chris-read. Version: Nov. 2010

[Sal12] Salchow, Marko: *Auswahl und Einsatz von APIs zur Steuerung einer hybrid Cloud-Infrastruktur*, FH-Stralsund, Master-Thesis, Aug. 2012

[Sch12] Schlegel, Wolf: Continuous Delivery. In: *Javamagazin* 3 (2012), S. 20–28

[SL05] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest*. dpunkt.verlag; Heidelberg, 2005

[Son12a] SonarSource S.A: *sonar - Features*. WWW. http://www.sonarsource.org/features/. Version: Aug. 2012

[Son12b] Sonatype, Inc: *Repository Management with Nexus*. WWW. http://www.sonatype.com/books/nexus-book/reference/. Version: Juli 2012

[Sou] SourceForge: *JWebUnit*. WWW. http://jwebunit.sourceforge.net/. – Abgerufen am 22.09.2012

[Sta11] Starke, Gernot: *Effektive Software-Architekturen : ein praktischer Leitfaden*. Bd. 5. München : Hanser, 2011

[Ste04] Steinweg, Carl: *Management der Software-Entwicklung : Projektkompass für die Erstellung von leistungsfähigen IT-Systemen*. Bd. 5. Wiesbaden : Vieweg, 2004. – 397 S.

[Sto12] Stoyanov, Stoyan: *Continuous Deli*. Juni 2012. – Interner E-Mail-Verkehr vom 01.06.2012

[Sun97] Sun Microsystems ; SUN MICROSYSTEMS (Hrsg.): *Java Code Conventions*. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.: SUN MICROSYSTEMS, Sep. 1997

[Sun09] Sun Microsystems ; SUN MICROSYSTEMS (Hrsg.): *Java Platform, Enterprise Edition (Java EE) Specification, v6*. 4150 Network Circle, Santa Clara, California 95054, USA: SUN MICROSYSTEMS, Juni 2009. `http://download.oracle.com/otndocs/jcp/javaee-6.0-fr-eval-oth-JSpec/`

[Teu11] Teuteberg, Frank: *Compliance*. WWW. `http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/daten-wissen/Grundlagen-der-Informationsversorgung/Compliance/index.html/?searchterm=compliance`. Version: Sep. 2011. – Abgerufen am 18.09.2012

[The11] The Apache Software Foundation: *Apache Tomcat 6.0 - Clustering/Session Replication HOW-TO*. WWW. `http://tomcat.apache.org/tomcat-6.0-doc/cluster-howto.html`. Version: Nov. 2011. – Dokumentation Apache Tomcat 6.0

[The12] The Open Source Initiative: *Open Source Initiative OSI - The MIT License (MIT):Licensing*. WWW. `http://opensource.org/licenses/mit-license.php`. Version: Sep. 2012

[Tho] ThoughtWorks, Inc.: *go Help documentation*. WWW. `http://www.thoughtworks-studios.com/docs/go/12.3/help/`. – Abgerufen am 22.09.2012

[Tho12a] ThoughtWorks, Inc.: *Experience Counts: The Leaders in Agile Software Development*. WWW. `http://www.thoughtworks-studios.com/company`. Version: Sep. 2012

[Tho12b] ThoughtWorks, Inc.: *Go Pricing and Sales*. WWW. `http://www.thoughtworks-studios.com/go-agile-release-management/contact-sales`. Version: Sep. 2012

[Tho12c] ThoughtWorks, Inc.: *One-Click Deployment.* WWW. http://www.thoughtworks-studios.com/go-agile-release-management. Version: Aug. 2012

[Tim12] Timberman, Joshua: *What is Knife?* WWW. http://wiki.opscode.com/display/chef/Knife. Version: Sep. 2012. – Abgerufen am 23.09.2012

[Web12] Weber, Robert: *Analyse von Anforderungstest-Automatisierung mithilfe des Behaviour Driven Development Frameworks Cucumber-JVM im Rahmen des Projektes Zensus Merkmalserver der adesso AG*, FH-Stralsund, Master-Thesis, Okt. 2012. – Zum Zeitpunkt der Verarbeitung noch unveröffentlicht.

[Wir09] Wirdemann, Ralf: *Scrum mit User Stories.* München ; Wien : Hanser, 2009. – 220 S.

[Wol12] Wolff, Eberhard: *Was Enterprises von Startups lernen koennen.* Online Blog. http://blog.adesso.de/was-enterprises-von-startups-lernen-konnen/. Version: Aug. 2012. – Abgerufen am 28.09.2012

[Zes12] Zeschke, Robert: *Big Data und NoSQL - Vergleich von NoSQL-Datenbanksystemen in Bezug auf das Handling und die Analyse-Möglichkeiten von riesigen Datenbestaenden*, FH-Stralsund, Master-Thesis, Okt. 2012. – Zum Zeitpunkt der Verarbeitung noch unveröffentlicht.

Teil II.

Anhänge

1. Zusammenfassung Fragebogen CI-Team

1.1. Skizze / Phasen des derzeitigen Entwicklungs- und Auslieferungsprozesses

---

Bitte skizzieren Sie den derzeitigen und Prozess der Softwareerstellung von der Erstellung bis zur Auslieferung.

Antwort: *Verweis auf das Unternehmens-Wiki von adesso. Hierunter fallen demnach die Phasen Build, Smoke-Test, Integrationstest, Regressionstest und Web-Test*[1]

Antwort:

- Setup Entwicklungsumgebung
- Entwicklung, Subphasen: Checkin $->$ Build & Unittests $->$ Automatisierte Deployments $->$ Automatisierte (Integrations-)Tests
- Integrationstests
- Delivery

Antwort:

1. Entwicklung, meist TDD
2. Build
3. Unit-Tests
4. Integrations Tests

[1] Vgl. `https://www.adesso.de/wiki/index.php/Continuous_Integration#Build-Pipeline`

5. ggf. Deploy auf Testsystem / Staging System
6. ggf manuelles Testen oder exploratives Testen
7. Release bzw. Auslieferung manuell, wenn 1-7 ok

Welche Phasen, von der Erstellung bis zur Auslieferung von Software, werden unterschieden?

Zustimmung:

Kompilierung des Quellcode	2	67%
Unit- und Komponententests / Analye	2	67%
automatisierte Akzeptanztests	1	33%
automatisierte Kapazitätstests	2	67%
manuelle Testsstage	1	33%
Konfigurieren von Test- und Produktivumgebung	1	33%
Installation und der Release der Software	2	67%

Antwort: Integrations-Tests

Antwort: diese sind in jedem Projekt ganz individuell!

1.2. Commit-Stage

Aussagen zur Commit-Stage

Zustimmung:

Gewöhnlich wird die Software auf einem Entwickler-System gebaut und paketiert.	0	0%
Es gibt einen standardisierten Prozess für das Bauen / Paketieren einer Anwendung.	2	67%
Wenn Quellcode kompiliert wurde, wird dieser in allen nachfolgenden Phasen verwendet und nicht noch einmal kompiliert.	3	100%
Erstellte Softwarepakete werden mit einer fortlaufenden Versionsnummer versehen.	2	67%
Alle im Prozess erstellten Softwarepakete werden mit einer fortlaufenden Versionsnummer versehen.	1	33%
Alle Artefakte, die beim Kompilieren und Testen entstehen, werden in einem zentralen Repository aufbewahrt.	3	100%
Dateien und Information zur Konfiguration der erstellten Software werden in einem geschützten Speicher zentral verwaltet.	2	67%
Es werden übersichtliche Berichte erstellt, die eine qualitative Aussage einer bestimmten Version ermöglichen.	1	33%

Anmerkung: auch hier: ist in jedem Projekt ganz individuell!

Wie werden Softwareentwicklungsprojekte beim Konfigurationsmanagement unterstützt?

Antwort: Es gibt Vorlagen für Ant und Maven

Antwort: Das Startup (vormals CI) team unterstützt durch Beratung und Doing in der Startup-Phase eines Projektes. Weiterhin werden Templates für Buildsysteme z.V. gestellt, damit Projekte einheitlich gebaut werden (ANT bereits vorhanden, Maven und Gradle-Templates in Vorbereitung).

Antwort:

- Beratung
- über Ticketsystem, um Einrichten von CI Jobs oder bsp. Sonar zu triggern

Wie wird mit Abhängigkeiten zu externen Bibliotheken bzw. Komponenten von Drittanbietern umgegangen?

Antwort: Über einen eigenen Nexus-Server

Antwort: 3rd-Party Bibliotheken werden auf einem zentralen Nexus in einem dafür vorgesehenen Repository abgelegt. Die Projekte sind verantwortlich für die Einhaltung der Compliance, z.B. dürfen Libs, deren Lizenz eine Auslieferung verhindern (z.B. SUN JDK Bibliotheken, JDBC-Treiber usw.) nicht mit ausgeliefert werden.

Antwort: Nexus als Repository Manager und Proxy zu anderen Repositories

---

Wie wird der Build-/ Test-Prozess ausgelöst?

Zustimmung:

Kompilierung, Paketierung und Test werden jeweils durch einzelne Kommandos auf der Konsole durchgeführt.	0	0%
Für den Prozess existieren Skripten, die das Bauen, Paketieren und Testen der Anwendung durchführen.	0	0%
Wir haben automatisierte Prozesse, die durch Ereignisse, z.B. Entwickler-Commit in der Versionsverwaltung, ausgelöst werden.	2	67%

Anmerkung: individuell

Anmerkung zur Commit-Stage:

Anmerkung: Jedes Projekt handhabt dies ganz individuell!

1.3. Acceptance-Stage

Aussagen zum Testbetrieb:

Zustimmung:

	Zustimmung	Anteil
Es gibt vereinzelt ein paar automatisierte Unit- und Komponententests.	1	33%
Bei einer Änderung des Softwarestandes werden der größte Teil der Tests wiederholt (Regressionstest).	3	100%
Es wird eine statische Code-Analyse durchgeführt.	2	67%
Funktionalen Anforderungen werden automatisiert getestet.	2	67%
Unit- und Komponententest werden mit einer hohen Abdeckung durchgeführt.	3	100%
Wenn manuelle Tests durchgeführt werden, basieren diese auf speziellen Risikoszenarien.	1	33%
Die Anwendung wird nach bestimmten Sicherheitsaspekten analysiert.	1	33%
Es wird auf eine Testabdeckung von 100% Wert gelegt.	0	0%
Test- und Produktivumgebung sind zu großen Teilen identisch.	1	33%
Testberichte können von allen Projektbeteiligten eingesehen werden.	1	33%

Welche Werkzeuge und Methoden werden zur statischen Analyse des Quellcodes angeboten?

Antwort: Sonar und FindBugs

Antwort: Sonar (kapselt Findbugs, Jococo, Checkstyle, PMD, Cobertura)

Antwort: Sonar, Findbugs, PMD

Welche Werkzeuge werden zur Aufdeckung von möglichen Sicherheitslöchern verwendet bzw. angeboten?

Antwort: s. oben (Findbugs hat einige Security-Regeln)

Welche Werkzeuge nutzen Sie für automatisierte Akzeptanz- und Kapazitätstests?

Antwort: JMeter, Selenium

Antwort: JUnit, JBehave

Welche Informationen können aus den Testberichten entnommen werden?

Antwort: Alle wesentlichen ;-) Abdeckung Erfolg, Nichterfolg

Antwort: Fehlerhafte Tests, Testabdeckung

Nach welchen architektonischen Vorgaben / Brüchen wird die erstellte Software hin untersucht?

Antwort: Modularität, Complexität, Kohäsion

Gibt es bestimmte Strategien für den Umgang mit Testdaten?

Antwort: teilweise

Anmerkungen zur Acceptance-Stage

Anmerkung: Auch hier: in jedem Projekt ganz individuell

1.4. Deployment

Welche Verfahren werden genutzt, um eine releasefähige Software in die Produktivumgebung auszuliefern?

Antwort: Releases werden über Jenkins ausgeliefert

Antwort: manuelles Delivery → es ist mir nicht bekannt, dass wir in eine Prod-Umgebung eines Kunden automatisiert zugreifen und deployen könnten. Zudem steht einem automatisierten Deployment in vielen Unternehmen die SOX-Compliance entgegen, die besagt, dass Entwicklung und Produktion strikt zu trennen ist.

Antwort: Skripte (Shell, Ant), Manuell

Wie wird das Software-Deployment durchgeführt?

Zustimmung:

	Zustimmung	Anteil
Es gibt derzeit keine Deployment-Skripte.	0	0%
Deployment-Skripte werden vorwiegend über die Konsole angestoßen.	1	33%
Deployment-Skripte werden automatisch angestoßen, sofern ein bestimmtes Quality-Gate durchschritten wurde.	0	0%
Es gibt eine Oberfläche von der aus das Deployment einer bestimmten Version angestoßen werden kann.	1	33%

Anmerkung: individuell, meist über Ant oder Maven

Werden Projektteams bei der Erstellung eines Deployment-Skriptes unterstützt?

Zustimmung:

	Zustimmung	Anteil
Es gibt keine Unterstützung, da die Projekte dies selbst regeln.	0	0%
Es gibt ein paar Skripte aus anderen Projekten, die als Vorlage genutzt werden können.	1	33%
Es gibt Templates für unsere Standardwerkzeuge.	1	33%
Es gibt Tools, mit denen sich das Deployment konfigurieren lässt.	1	33%

Welche Werkzeuge werden für das Deployment genutzt?

Antwort: ant und Shell-Skripte

Antwort: Ant mit Shellscripten für App-Server, die keine dedizierte Schnittstelle oder Plugin für Ant/Maven bieten. Falls diese Ant-Tasks oder Maven-Plugins anbieten (z.B.

Apache Tomcat) werden diese genutzt

Wie wird mit Konfiguration von Test- / Produktivumgebungen umgegangen?

Zustimmung:

	Zustimmung	Anteil
Abhängige Komponenten werden vor dem Betrieb bzw. Test manuell installiert.	1	33%
Projekte erstellen eigene Skripte, um die benötigten Komponenten zu installieren.	2	67%
Es gibt Template-Skripte, um Standardumgebungen zu installieren und zu konfigurieren.	0	0%

Existieren derzeit Bemühungen eine Delivery-Pipeline in einzelnen Softwareentwicklungsprojekten umzusetzen?

Zustimmung:

	Zustimmung	Anteil
Es gibt noch kein Einsatzszenario bzw. Interesse.	0	0%
Es sind Projekte in Zukunft geplant.	0	0%
Es gab / gibt schon Pilotprojekte.	1	33%
Eine Delivery-Pipeline haben wir schon öfter umgesetzt.	1	33%

Anmerkungen zum Deployment

Anmerkung: Auch hier: alles individuell, insbesondere was die Produktivsetzung angeht. Zur Frage *Wie wird mit Konfiguration von Test- / Produktivumgebungen umgegangen?* Auch hier ganz unterschiedlich, aber meist manuell. Bei der AMS haben wir nun in einer Cloud-Umgebung Chef im Einsatz um Server automatisiert zu provisionieren.