

MASTERARBEIT

DevOps und Continuous Delivery

Evaluierung der Umsetzung von DevOps in kleinen Unternehmen mit geringer Personenanzahl

durchgeführt am
Studiengang Informationstechnik und System-Management
an der
Fachhochschule Salzburg

vorgelegt von:
Michael Haslauer, BSc



Studiengangsleiter: FH-Prof. DI Dr. Gerhard Jöchl
Betreuer: DI Christian Neureiter

Puch-Urstein, Januar 2015

Eidesstattliche Erklärung

Hiermit versichere ich, Michael Haslauer, geboren am 22. Feb. 1991 in Salzburg, dass die vorliegende Masterarbeit von mir selbstständig verfasst wurde. Zur Erstellung wurden von mir keine anderen als die genannten Quellen mit Hilfsmitteln verwendet.

Salzburg, am 21. Juni 2015



Michael Haslauer

1310581019

Matrikelnummer

Kurzzusammenfassung

Die beiden Begriffe DevOps und Continuous Delivery sind im Web-Bereich der IT Branche aktuell sehr stark vertreten. Immer mehr Unternehmen führen DevOps in ihre Prozesse ein und betreiben Continuous Delivery für ihre Online-Plattform. Die genaue Definition dieser beiden Begriffe ist allerdings unklar und nicht genau festgelegt. Die vorliegende Arbeit zeigt auf, was diese beiden Begriffe bedeuten und welche Vorteile bzw. Nachteile die Umsetzung dieser Konzepte für ein Web-Unternehmen hat. Zur Veranschaulichung werden diese Konzepte, im Rahmen einer Fallstudie, für ein Unternehmen diskutiert. Grundsätzlich geht es bei DevOps und Continuous Delivery um die vollständige Automatisierung der Build und Deployment Pipeline von Software Projekten. Damit soll bei gleichbleibender Effektivität die Effizienz erheblich gesteigert werden. Es wird aufgezeigt, welche Änderungen sich dadurch für ein Unternehmen ergeben und wie diese durchgeführt werden können. Insbesondere werden dabei die Bereiche der Prozesse, Infrastruktur und Firmenkultur betrachtet. In vorliegender Arbeit wird dazu eine minimale Infrastruktur aufgebaut, die benötigt wird um DevOps und Continuous Delivery betreiben zu können. Anschließend wird anhand vier ausgewählter Applikationen gezeigt, wie eine Umstellung durchgeführt wird und der dafür notwendige Arbeitsaufwand erfasst. Die positiven Effekte nach der Umstellung werden anhand von definierten Metriken erfasst und mit dem Zustand vor der Einführung verglichen. Im Zuge vorliegender Arbeit wird schlussendlich aufgezeigt, dass die Effizienz erheblich verbessert wird und sich der notwendige Arbeitsaufwand in Grenzen hält. Die Einführung von DevOps und Continuous Delivery ist deshalb für viele Unternehmen erstrebenswert.

Abstract

DevOps and Continuous Delivery are currently two common terms in the web branch of the IT area. More and more companies adopt the principles of DevOps and try to do Continuous Delivery in the development process of their online platforms. Nevertheless the concrete definitions of these two terms are unclear. This master thesis shows in depth what these two terms stand for and what implementation of these principles implies for a web company. In order to achieve this, a case study is setup to show the implementation by means of a concrete company. Basically the adoption of DevOps and Continuous Delivery result in a complete automation of the build and deployment pipeline of software projects. Therefore at constant effectivity the efficiency should rise dramatically. In this case study is shown which changes have to be done in the area of processes, infrastructure and company culture and how these changes can be performed. Therefore in the course of this master thesis, a minimal infrastructure will be implemented that is necessary to achieve DevOps and Continuous Delivery. Afterwards the automation of the build and deployment pipeline is shown on behalf of four selected applications and how much effort is involved. The positive effects after the implementation are measured by defined metrics and compared to the state before the implementation started. This shows how the efficiency increases for these applications and how many effort is necessary to achieve it. As a conclusion the adoption of the principles DevOps and Continuous Delivery is desirable for many companies.

Inhaltsverzeichnis

1	Einleitung	6
1.1	Grundkonzepte von DevOps	7
1.2	Continuous Delivery	8
2	Theoretische Grundlagen	9
2.1	Probleme des klassischen Software-Betriebs	9
2.1.1	Manuelles Deployment einer Software	9
2.1.2	Lange Release- und Feedbackzyklen	11
2.1.3	Kein Konfigurationsmanagement für die Infrastruktur	12
2.1.4	Distanz zwischen Entwicklung und Betrieb	12
2.2	DevOps und Continuous Delivery	13
2.2.1	Grundprinzip	14
2.2.2	Automation von Aufgaben	16
2.2.3	Automatisiertes Testen	18
2.2.4	Infrastruktur Management (Infrastructure as Code)	21
2.2.5	Die Deployment Pipeline	22
2.2.6	Unternehmenskultureller Aspekt von DevOps	26
3	Methodik	28
3.1	Goal-Question-Metric	28
3.1.1	Problemstellung und Zielsetzung	28
3.1.2	Forschungsfragen	29
3.1.3	Definition der Metriken	30
3.2	Beschreibung der Fallstudie	31
3.2.1	Rahmenbedingungen	32
3.2.2	Vorgehensweise	32
3.2.3	Ist-Zustand der Red Bull Media Base	33
3.2.4	Das Projektteam	34
4	Implementierung	36
4.1	Voraussetzungen	36

4.1.1	Logischer Aufbau des Rechenzentrums	37
4.1.2	Virtual Maschine Management	39
4.1.3	Agiles Entwicklungsmodell	40
4.2	Ausgewählte Systeme und Konzepte	41
4.2.1	Konfigurationsmanagement mit Ansible	41
4.2.2	Build und Deployment Pipeline mit Bamboo	46
4.2.3	Lokale Entwicklungsumgebung mit Vagrant	48
4.2.4	Lifecycle der Applikationen im DC2.0	50
4.2.5	Erstellung von virtuellen Maschinen	52
4.2.6	Sonstige Systeme	53
4.3	Basisinfrastruktur	54
4.3.1	Erstellen von VM-Images mit Packer	55
4.3.2	Automatic Service Discovery/Registry mit Consul	57
4.3.3	Loadbalancing mit HAProxy	60
4.3.4	Log-Aggregation mit Logstash	63
4.3.5	Monitoring mit Sensu	68
4.3.6	Zusammenfassung der Basisinfrastruktur	73
4.4	Implementierung der DevOps Konzepte	73
4.4.1	Applikation - Delivery Agent	74
4.4.2	Applikation - OAuth2 Server (OIDC)	77
4.4.3	Applikation - Content Processing Agent (CPAS)	81
4.4.4	Applikation - Outlet Service (APEX Framework)	85
5	Ergebnisse der Fallstudie	89
5.1	Auswertung der Metriken	89
5.2	Diskussion der Ergebnisse	91
5.3	Abschließendes Fazit	96
	Verzeichnisse	97
	Tabellenverzeichnis	97
	Abbildungsverzeichnis	98
	Auflistungsverzeichnis	99
	Referenzen	100

Kapitel 1

Einleitung

In der Softwarebranche ist es sinnvoll standardisierte Prozesse zu schaffen, um die Entwicklung von Software, sowie deren Einsatz möglichst effizient zu gestalten. Dabei müssen viele unterschiedliche Bereiche beachtet werden. Dies umfasst beispielsweise die Entwicklung von Prozessen für die Planung und Umsetzung von Software, sowie Vorgehensmodelle für den zuverlässigen Einsatz der fertigen Applikationen. Auch zwischenmenschliche Aspekte, wie die Zusammenstellung beteiligter Personen zu einem hochperformanten Team, müssen mit einbezogen werden. In der klassischen Software-Entwicklung gibt es bereits viele Modelle die versuchen, diese Aufgaben zu bewältigen. Die im letzten Jahrzehnt eingesetzten Modelle haben aber mehrere Nachteile [8]. So ist es oft üblich, dass Release-Zyklen mehrere Monate dauern, um den erhöhten Verwaltungs- und Kommunikationsaufwand dabei zu vermeiden. Die durch den langen Zeitraum resultierenden vielfältigen Änderungen, können an vielen Stellen Fehler erzeugen. Auch das Testen der Änderungen erfordert einen hohen Aufwand und benötigt entsprechende Zeit. Auf Grund dieser Faktoren entsteht bei einem Release oft erhöhter Stress für die beteiligten Personen, da es unklar ist, ob die Software noch korrekt betrieben werden kann oder ob es unentdeckte Fehler gibt. Eine gängige Praxis ist, dass die Auslieferung und der Betrieb einer fertigen Applikation oft noch mit viel manueller Arbeit verbunden ist. Diese Vorgehensweise ist zeitaufwändig und fehleranfällig. Darüber hinaus kann nicht ausreichend flexibel auf geänderte Anforderungen (z.B. vom Kunden) reagiert werden. Es ergeben sich lange Durchlaufzeiten, bis Änderungen an der Software tatsächlich eingesetzt werden können. Dies beeinflusst die Zufriedenheit des Kunden oft negativ.

Ein weiterer Nachteil ist, dass Entwickler sehr spät Rückmeldung zur Qualität ihrer Arbeit bekommen. Im Bereich der Planung und Durchführung von Software-Entwicklung setzen deswegen immer mehr Unternehmen auf sogenannte *Agile Vorgehensmodelle*. Diese agilen Modelle sprechen viele Probleme von klassischen Vorgehensmodellen an.

Ihr Hauptziel ist es, Durchlaufzeiten so kurz wie möglich zu halten und dadurch möglichst hohe Flexibilität zu erreichen. Durch kurze Release-Zyklen können Verbesserungen rascher an die Kunden geliefert werden. Agile Vorgehensmodelle fokussieren sich hauptsächlich auf die Entwicklung von Software und bieten Methoden, um diese effizienter zu gestalten. Der sogenannte *Operations-Bereich*, der sich mit der Auslieferung (Deployment) sowie dem zuverlässigen Betrieb der Software beschäftigt, wird hierbei meist nicht beachtet. Deswegen ist es für sie oft eine Herausforderung, mit den häufigen Releases der Entwicklung mithalten zu können. Dies führt letztendlich dazu, dass sie nur schwer einen einwandfreien Betrieb gewährleisten können [8, 9].

1.1 Grundkonzepte von DevOps

DevOps, Verschmelzung der Begriffe **D**evelopment und **O**perations, adressiert oben genannte Probleme und bietet Konzepte für deren Lösung. Laut [23] sollte man zwei Hauptbereiche betrachten, bei denen Änderungen notwendig sind. Konkret handelt es sich dabei um die technische Unterstützung und die Unternehmenskultur:

Der erste Teil beschäftigt sich mit der technischen Unterstützung von Prozessen und Aufgaben. Dadurch soll die Auslieferung von Software effizienter und weniger fehleranfällig gemacht werden. Darüber hinaus wird im Betrieb eine Steigerung der Zuverlässigkeit angestrebt. Wie bereits erwähnt, wird das Deployment einer neuen Software oft manuell durchgeführt. Diese manuelle Tätigkeit muss bei jedem Release vom Operations-Team durchgeführt werden. Dabei wird die neue Version in allen vorhandenen Umgebungen (meist Test, Staging und Produktion) deployed und anschließend getestet, ob die Software noch einwandfrei funktioniert. Bei verkürzten Release-Zyklen muss dies häufiger erfolgen, wodurch die Gefahr besteht, dass das Operations-Team diese Aufgabe nicht mehr sorgfältig und gewissenhaft durchführen kann. Dies hat eine höhere Fehleranfälligkeit zur Folge. Der technische Aspekt von DevOps befasst sich damit, manuelle Tätigkeiten zu minimieren und möglichst viele Arbeitsschritte zu automatisieren. Die Automatisierung umfasst dabei nahezu alle Bereiche einer Software-Entwicklung. So können zum Beispiel die Erstellung des fertigen Release-Paketes, das Testen der Software (speziell bei Regressionstests), sowie das Deployment automatisiert werden. Da die automatisierten Prozesse jedes mal genau gleich durchgeführt werden, wird die Möglichkeit eines menschlichen Fehlers minimiert. Des Weiteren können wesentlich kürzere Durchlaufzeiten beim Deployment erzielt werden. Dieser positive Effekt tritt vor allem bei Clustern in Erscheinung, wo eine Software nicht nur auf einer einzigen Maschine, sondern auf mehreren Servern eingesetzt wird. Durch Automatisierung können alle Server gleichzeitig aktualisiert werden.

Der zweite Bereich von DevOps befasst sich mit der Zusammenarbeit der unterschiedlichen Teams, die im Lebenszyklus einer Software beteiligt sind. Wie bereits erwähnt, bedeutet der Begriff DevOps die Zusammenführung von Entwicklung und Betrieb. Diese beiden Teams verfolgen oft unterschiedliche Ziele und bestehen aus Personen mit verschiedenen Backgrounds und Wissen. Grundsätzlich kennt niemand die Software besser, als derjenige, der sie entwickelt hat. Das Operations-Team ist an der Entwicklung nicht beteiligt, muss aber für den fehlerfreien Betrieb der Software sorgen. Dies erfordert einen hohen Aufwand an Wissenstransfer, von der Entwicklung zum Betrieb. Um dieses Problem zu lösen, versucht DevOps die strikte Trennung beider Abteilungen aufzuheben. Die Mitarbeiter beider Teams sollen in allen Phasen beteiligt sein. Operations stellt die nötigen Rahmenbedingungen her, um Software fehlerfrei betreiben zu können. Die Entwicklung selbst ist am Prozess des Software-Betriebs beteiligt und hat auch die nötigen Befähigungen und Berechtigungen, um ihre Software deployen und betreiben zu können. DevOps verfolgt das Ziel, einen hohen Grad an Kollaboration zwischen den einzelnen Teams zu erreichen [22].

1.2 Continuous Delivery

Continuous Delivery ist eine Praxis, die mit Hilfe von DevOps ermöglicht wird. Ist in allen erforderlichen Bereichen ein ausreichend hoher Grad an Automatisierung erreicht, kann jede Änderung an der Software sofort durch den Deployment Prozess geschickt werden. Im Laufe dieses Prozesses wird die geänderte Software gebaut, getestet und deployed, was zu einer minimalen Durchlaufzeit führt. Voraussetzung ist, dass die automatisierten Tests eine ausreichend hohe Qualität der Software sicherstellen, um diese im produktiven System einsetzen zu können. Außerdem muss die Automatisierung entsprechend ausgereift sein, damit vertrauenswürdige Ergebnisse erzielt werden können.

Continuous Delivery ist allerdings nicht in allen Bereichen, in denen Software verwendet wird, einsetzbar. Medizinische Software hat beispielsweise sehr hohe Ansprüche an Qualität und Fehlerfreiheit. Hier wird man in vielen Fällen nicht mit rein automatischen Tests auskommen. Dies ist aber ein Ausnahmefall, der auf viele Applikationen, insbesondere bei Internet Plattformen, nicht zutrifft. Hierbei sprechen viele Gründe für den Einsatz von Continuous Delivery.

Kapitel 2

Theoretische Grundlagen

Das nachfolgende Kapitel beschäftigt sich mit den theoretischen Grundlagen, die für die vorliegende Arbeit benötigt werden. Dies bezieht sich auf die Konzepte von DevOps und Continuous Delivery. Grundsätzlich kann man zwei primäre Phasen im Lebenszyklus einer Software unterscheiden. Zuerst erfolgt die Entwicklung und nachfolgend der Betrieb der Software. DevOps findet hauptsächlich in der Phase des Software-Betriebs statt und liefert Ansätze, um häufig auftretende Probleme zu bewältigen.

2.1 Probleme des klassischen Software-Betriebs

Das primäre Ziel von Software ist, einen Mehrwert für die Benutzer zu schaffen. Von der initialen Idee bis zum produktiven Einsatz gibt es viele Probleme zu überwinden. Seit Ende der Neunziger werden bereits viele Probleme in der Software-Entwicklung, durch den Einsatz von agilen Entwicklungsmethoden, bewältigt. Die Probleme im Bereich des Software-Betriebs wurden dabei aber nicht berücksichtigt. Wie in der Einleitung erwähnt, versucht DevOps diese bekannten Probleme des klassischen Software-Betriebs zu lösen. Aus diesem Grund wird DevOps auch als agiler Software-Betrieb bezeichnet [8, 17]. In den nachfolgenden Abschnitten wird eine Auswahl bekannter Probleme aus [8] angeführt und durch DevOps ermöglichte Lösungen aufgezeigt.

2.1.1 Manuelles Deployment einer Software

Software-Systeme haben meist einen hohen Grad an Komplexität. Bei den meisten Systemen ist es nahezu ausgeschlossen, dass eine einzige Person alle Komponenten überblicken und verstehen kann. Trotzdem führen viele Betriebsteams das Deployment einer

neuen Software Version manuell durch. Laut [8] treten dabei häufig folgende Probleme auf:

Jede Tätigkeit die manuell durchgeführt wird, ist anfällig für menschliche Fehler. Sofern diese Tätigkeiten nicht sehr genau definiert und dokumentiert sind, können unterschiedliche Personen sie unterschiedlich ausführen. Wird der Prozess wiederholt, so ist nicht sichergestellt, dass er genau gleich durchgeführt wird wie zuvor. Das Ergebnis des Prozesses ist somit nicht deterministisch. Dies hat zur Folge, dass der Prozess nicht nachvollziehbar ist und somit die Analyse von Problemen erheblich erschwert wird. Jede einzelne Tätigkeit muss nachvollziehbar sein. Wenn das Problem erst einige Zeit nach der Durchführung auftritt, ist dies meist nicht mehr möglich. Im schlimmsten Fall überblicken im Unternehmen nur einzelne Personen einen gesamten Prozess. Die Durchführung ist dann immer von diesen Personen abhängig.

Ein weiteres Problem des manuellen Deployments ist, die dafür benötigte Zeit. Je mehr Infrastrukturkomponenten bei einem Release betroffen sind, desto aufwändiger wird das Deployment. Vor allem bei verteilten Applikationen sind meist viele Server betroffen. Je nach Art des Deployments (siehe Abschnitt 2.2.5) ist die Applikation während dieses Vorgangs für die Benutzer nicht verfügbar. Zusätzlich zu den Kosten für die Arbeitszeit der Mitarbeiter, kommen noch Wertverluste durch Nicht-Erreichbarkeit der Applikation hinzu. Dies verursacht beispielsweise negative Reputation oder Umsatzeinbußen bei Onlineshops. Durch den nicht deterministischen Ausgang eines manuellen Deployments, ist nicht sichergestellt, dass die neue Version nach dem Deployment korrekt funktioniert.

Folglich beinhaltet manuelles Deployment ein hohes Maß an Risiko und ist fehleranfällig. Aus diesem Grund sind Releases in vielen Unternehmen eine kritische Angelegenheit, mit der oft mehrere Mitarbeiter über einen längeren Zeitraum beschäftigt sind. Deployments werden deswegen weitgehend vermieden, wodurch die Probleme aber eher verstärkt anstatt vermindern werden. DevOps und Continuous Delivery setzen deswegen auf die vollständige Automatisierung dieser Aufgaben (Abschnitt 2.2.2). Wenn alle Tätigkeiten automatisiert erfolgen ist sichergestellt, dass sie jedes mal in genau der gleichen Reihenfolge und gleichen Art durchgeführt werden. Der Ausgang des Prozesses ist somit deterministisch und kann jederzeit nachvollzogen werden. Außerdem sind während eines Deployments keine Mitarbeiter beschäftigt und das Deployment ist in den meisten Fällen schneller abgeschlossen. Die Kosten und Risiken reduzieren sich erheblich und die Effizienz steigt. Als Nachteil ist allerdings anzuführen, dass die Umsetzung einen großen initialen Aufwand erfordert. Auf diesen benötigten Aufwand wird in Kapitel 4 genauer eingegangen.

2.1.2 Lange Release- und Feedbackzyklen

Mit agilen Entwicklungsmethoden können in der Entwicklung kurze Releasezyklen erreicht werden. Ein klassischer Betrieb kann damit aber nicht schritthalten. Hier betragen die Releasezyklen oft einige Monate, bis eine neue Software Version in Produktion deployed ist. Diese langen Zyklen haben laut [8] folgende Auswirkungen:

Je länger die Zyklen sind, desto größer ist das sogenannte Time-to-Market. Wird eine neue Funktion umgesetzt, so verursacht die Entwicklung dieser Funktion zuerst ausschließlich Kosten. Erst wenn die neue Funktion in der Applikation verfügbar ist, ergibt sich ein Wert für die Kunden. Je länger diese Zeit dauert, desto später wird ein Wert generiert. Im schlimmsten Fall wird zum Zeitpunkt des Deployments, die Funktion gar nicht mehr benötigt. Es sind also nur Kosten entstanden aber kein Wert. Ändern sich die Anforderungen der Kunden, kann nicht mit der nötigen Flexibilität reagiert werden. Dies führt zu unzufriedenen Kunden und im schlimmsten Fall zu deren Verlust. Des Weiteren müssen die Kunden entsprechend lange warten, bis geforderte Verbesserungen in der Applikation verfügbar sind. Dieser Nachteil betrifft vor allem die Fehlerbehebung. Wird ein Fehler erkannt, dauert es im schlimmsten Fall einen kompletten Zyklus, bis er behoben ist. Bei kritischen Fehlern ist dies oft nicht möglich und es wird ein außerplanmäßiges Deployment benötigt.

Je früher die Entwickler Feedback zu ihrer Arbeit bekommen, desto mehr sind sie noch mit dem nötigen Kontext vertraut. Fehler können so schneller gefunden und behoben werden. Erfolgt aufgrund langer Zyklen erst spät Rückmeldung, ist für die Fehlerbehebung ein Kontextwechsel notwendig. Dies bedingt meist eine längere Dauer für die Fehlerbehebung.

Gängige Praxis ist es, durch möglichst wenige Releases, das dabei auftretende Risiko beim Deployment zu vermeiden. In der Realität steigt aber das Risiko, je länger die Releasezyklen sind. Bei einem langen Zyklus sind viele Änderungen in einem Release enthalten. Die Änderungen betreffen viele Bereiche der Infrastruktur, die im Zuge des Deployments verändert werden. Treten anschließend Fehler auf, ist es schwer nachzuvollziehen, welche der Änderungen diesen Fehler verursacht. Bei kurzen Releasezyklen wäre die Menge der Änderungen hingegen überschaubar. Die Ursache für einen Fehler kann leicht festgestellt werden. Auch ein Rollback auf eine frühere Version, ist bei wenigen Änderungen meist leichter durchzuführen. Der Stressfaktor verringert sich erheblich für alle beteiligten Personen. Ist der Deploymentprozess automatisiert, kann dieser schnell und oft durchgeführt werden. Kurze Releasezyklen sind daher einfach zu erreichen und entlasten das Personal.

2.1.3 Kein Konfigurationsmanagement für die Infrastruktur

Der Software-Betrieb stellt für eine Applikation die benötigte Infrastruktur zur Verfügung. Die Verwaltung dieser Infrastruktur wird meist manuell durchgeführt. Die manuelle Verwaltung birgt laut [8] folgende vielfältige Gefahren:

Neben der produktiven Infrastruktur, müssen meist noch weitere Testumgebungen verwaltet werden. Alle Umgebungen sollten dabei den gleichen Stand an Konfiguration aufweisen. Ist dies nicht der Fall, so kann ein Deployment zwar auf der Testumgebung, nicht aber auf der produktiven Umgebung erfolgreich sein. Je weiter die Konfigurationen der einzelnen Umgebungen auseinander laufen, desto ineffizienter werden Tests. Anhand der Tests kann keine Aussage mehr getroffen werden, ob die neue Version auf der produktiven Umgebung einwandfrei funktioniert. Bei verteilten Applikationen muss sichergestellt werden, dass alle Maschinen des Clusters dieselbe Konfiguration aufweisen. Ansonsten können Unterschiede in der Konfiguration zu schwer reproduzierbaren Fehlern führen. Die Analyse und Behebung solcher Fehler erfordert demnach viel Zeit.

Wenn Umgebungen manuell verwaltet werden, ist die Menge an Umgebungen meist begrenzt. Spezielle Umgebungen für Tests von experimentellen Entwicklungen, die Änderungen an der Infrastruktur benötigen, sind dabei nicht vorgesehen. Solche Tests müssen entweder auf einer bestehenden Umgebung durchgeführt oder es muss eine neue Umgebung aufgebaut werden. Der Aufbau einer Umgebung ist allerdings mit viel Aufwand und Kosten verbunden. Wird auf bestehenden Umgebungen getestet, so sind die Tests des herkömmlichen Releasezyklus blockiert. Es ist also schwierig, experimentelle und komplexe Änderungen ausreichend zu testen.

Ist die Verwaltung der Infrastruktur ebenfalls automatisiert, so können diese Probleme gelöst werden (siehe Abschnitt 2.2.4). Die Infrastruktur wird dadurch nachvollziehbar und reproduzierbar. Unterschiedliche Konfigurationen werden verhindert, da die Prozesse auf jeder Maschine immer gleich ausgeführt werden. Ebenso können ganze Umgebungen ohne erheblichen Aufwand aufgebaut und wieder abgerissen werden.

2.1.4 Distanz zwischen Entwicklung und Betrieb

Ein Problem das DevOps immer wieder anspricht, ist die Distanz zwischen Software-Entwicklung und Betrieb. Beide Abteilungen sind maßgeblich an der Bereitstellung von Applikationen für die Kunden beteiligt. Herrscht zwischen diesen Abteilungen eine schlechte Zusammenarbeit, kommt es laut [17] meist zu folgenden Problemen:

Während der Entwicklung werden vom Entwicklerteam oft Annahmen über die produk-

tive Umgebung getroffen. Dies kann dazu führen, dass die Applikation zwar lokal auf der Entwickler-Maschine läuft, aber nach einem Deployment nicht in Produktion funktioniert. Hier kommt es oft dazu, dass sich Entwicklung und Betrieb die Schuld gegenseitig zuweisen, anstatt gemeinsam an einer Problemlösung zu arbeiten [17]. Releases sind deshalb oft sehr aufwändig und benötigen viel Zeit, da umfangreiche Kommunikation zwischen Entwicklung und Betrieb nötig ist.

Ein weiteres Problem ist, dass diese Abteilungen unterschiedliche Ziele verfolgen, die sich teilweise gegenseitig widersprechen [17, 21]. Ziel der Entwicklung ist es, möglichst rasch und oft Änderungen zu veröffentlichen, um die Applikation zu verbessern. Der Betrieb verfolgt das Ziel eine stabile Plattform bereitzustellen und wird meist anhand der Verfügbarkeit der Applikationen bewertet. Jede Änderung birgt dabei ein Risiko für dieses Ziel und hat eventuell eine Ausfallzeit zur Folge¹. Daher sollten nur ausreichend getestete Änderungen deployed werden. Dies bewirkt allerdings lange Releasezyklen und spricht gegen das Ziel der Entwicklung.

Deswegen befasst sich DevOps neben den technologischen Themen auch speziell mit der Unternehmenskultur. Es bietet Konzepte wie die Zusammenarbeit verbessert und die Kommunikation vereinfacht werden kann. Auf diese Lösungen wird in Abschnitt 2.2.6 genauer eingegangen.

2.2 DevOps und Continuous Delivery

DevOps nahm seinen Anfang im Jahr 2010, als Patrick Debois eine Konferenz namens DevOpsDays veranstaltete [17]. DevOps bedeutet dabei die Verschmelzung der beiden Bereiche der Entwicklung (**D**evelopment) und des Betriebs (**O**perations) für Applikationen. Allerdings fehlte damals eine genaue Definition des Begriffes und lieferte somit viel Interpretationsspielraum. Bis 2015 hat der Begriff bereits eine genauere Bedeutung erhalten. So versteht man heutzutage unter DevOps vor allem die Automatisierung von Prozessen, um diese effizienter und resistenter gegen Fehler zu gestalten [17, 21]. Ebenso ist das Abbauen von Hürden und die Optimierung von Prozessen² ein Hauptaspekt von DevOps. Ähnliche Ansätze spiegeln sich in der aktuell aufkommenden Lean-Bewegung wieder [9].

Der Begriff Continuous Delivery (CD) wurde im Jahr 2010 maßgeblich von [8] geprägt. Allerdings ist die Definition ebenfalls unpräzise. [5] und [23] versuchen dem Begriff eine genauere Bedeutung zu geben und bezeichnen Continuous Delivery als die Fähigkeit, jederzeit und ohne erheblichen Aufwand Änderungen auf Produktion ausrollen

¹Gemäß des Sprichworts [21]: "Never change a running system"

²Im Englischen oft mit **Reducing Waste** bezeichnet

zu können. CD ist die Weiterführung von Continuous Integration (CI), wie es in der Entwicklung schon gängige Praxis ist. Die Applikation wird dabei nicht nur gebaut, sondern auch sofort getestet und ausgerollt. Jede Änderung hat ein sofortiges Deployment zur Folge und reduziert somit die Release- und Feedbackzyklen auf ein Minimum. Hierfür ist es ebenfalls notwendig, Prozesse zu automatisieren. DevOps ermöglicht also erst die Umsetzung von Continuous Delivery [18]. Das hauptsächliche Anwendungsgebiet von DevOps und Continuous Delivery ist der Web-Bereich, wo Applikationen auf Servern betrieben werden. Allerdings können viele Ansätze und Prinzipien auch auf andere Gebiete der Software-Entwicklung und des Betriebs adaptiert werden. Die vorliegende Arbeit beschränkt sich auf den genannten Hauptbereich.

2.2.1 Grundprinzip

Wie man aus den gängigen Beschreibungen von [17, 21, 23] herauslesen kann, ist das Grundprinzip von DevOps die Steigerung der Effizienz³ bei gleichbleibender Effektivität⁴. Hürden sollen abgebaut und Prozesse optimiert werden, um einerseits die Kosten zu senken und andererseits mehr Ressourcen für die Weiterentwicklung der Plattform zur Verfügung zu haben.

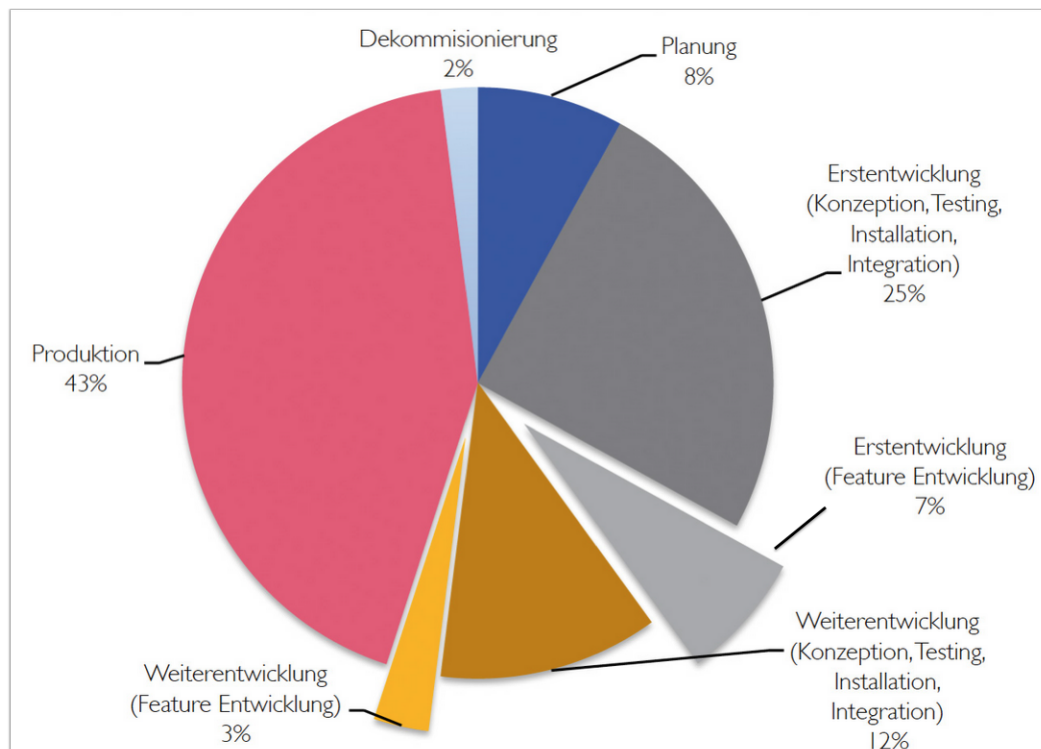


Abbildung 2.1: Verteilung des Budgets einer Applikation auf die unterschiedlichen Bereiche ohne DevOps, während ihres gesamten Lebenszyklus, aus [19]

³Definition: Wirtschaftlichkeit, Maß für den eingesetzten Aufwand zur Erreichung eines Zieles

⁴Definition: Wirksamkeit, Maß für das Erreichen eines definierten Zieles

Folgendes Beispiel führt [19] an, um die Wirtschaftlichkeit von DevOps zu verdeutlichen. Abbildung 2.1 zeigt die Verteilung des Budgets einer Applikation in ihrem Lebenszyklus. Zählt man die Bereiche für Erstentwicklung (graue Bereiche) und Weiterentwicklung (orange Bereiche) zusammen, so beträgt der Anteil für die Entwicklung 47% des Gesamtbudgets. Dabei sind allerdings Konzeption, Testen und Installation inkludiert. Betrachtet man rein den Anteil für die Entwicklung, so beträgt dieser lediglich 10%. Durch DevOps können die Kosten vieler Bereiche reduziert werden. Auf Grund der häufigen Releases und das schnelle Feedback, muss in der Planungsphase nicht mehr jedes Szenario aufs kleinste Detail analysiert werden. Ist eine Fehlplanung erfolgt, so kann diese schnell wieder korrigiert werden. Auf die Lebenszeit einer Applikation gesehen kann, durch die Automatisierung des Deploymentprozesses, viel Budget eingespart werden. Das Testen von Funktionen ist ebenfalls effizienter und benötigt weniger menschliche Ressourcen. Abbildung 2.2 zeigt die Verteilung des Budgets nach der Umsetzung von DevOps.

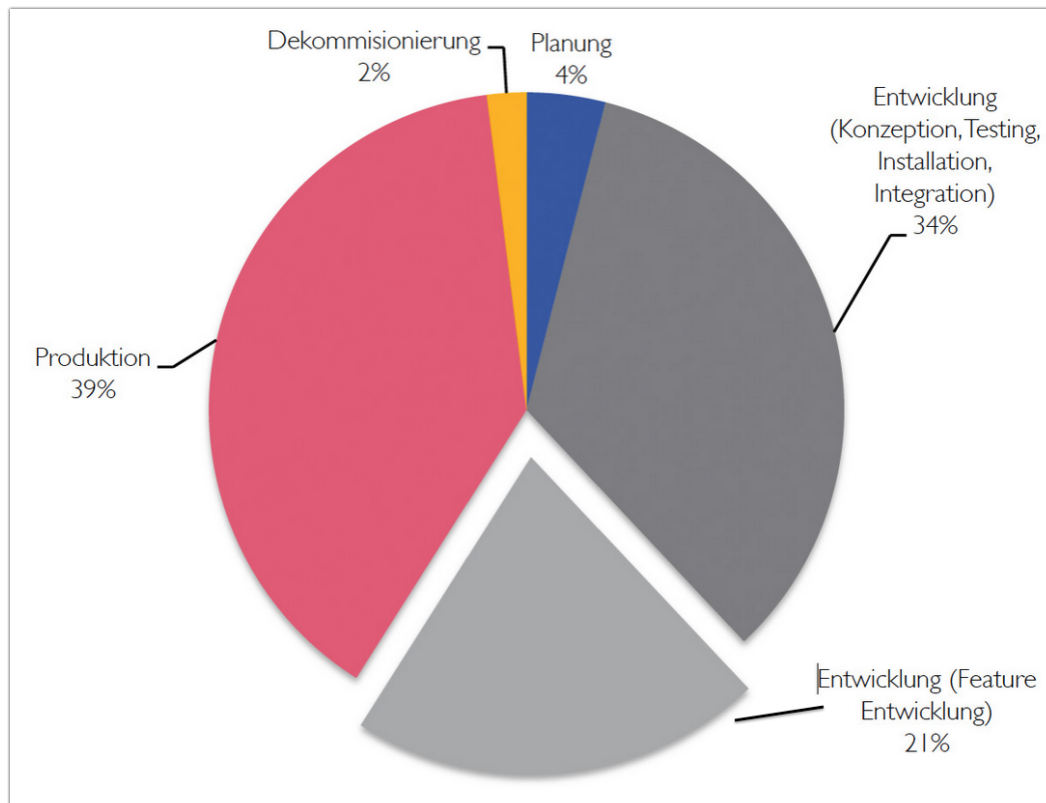


Abbildung 2.2: Verteilung des Budgets einer Applikation auf die unterschiedlichen Bereiche mit DevOps, während ihres gesamten Lebenszyklus, aus [19]

Führt man das eingesparte Budget der Entwicklung zu, so kommt man nun auf einen Anteil von etwa 21%. Dies entspricht einer Steigerung von mehr als 100%. Das hat zur Folge, dass mit demselben Budget mehr als doppelt so viele Weiterentwicklungen einer Applikation möglich sind. Somit kann unter anderem die Kundenzufriedenheit und der Wert der Applikation erheblich gesteigert werden.

In den nachfolgenden Abschnitten wird nun beschrieben, wie die Steigerung der Effizienz erreicht werden kann. Dabei wird einzeln auf die wichtigsten Bereiche eingegangen, die DevOps umfasst und Konzepte beschrieben, wie die Probleme aus Abschnitt 2.1 bewältigt werden können.

2.2.2 Automation von Aufgaben

Wie in den Abschnitten 2.1.1 und 2.1.2 beschrieben, birgt die manuelle Ausführung von Aufgaben ein hohes Risiko und kann in größeren Maßstäben nicht effizient durchgeführt werden. Daher ist die oberste Priorität auf technischer Seite, die Automatisierung von Aufgaben, die bisher manuell durchgeführt wurden. Aus diesem Grund muss der komplette Prozess, von der Änderung der Software bis zum Betrieb der neuen Version, betrachtet werden [23]. Reduziert man den Prozess auf ein Minimum, so umfasst er zumindest zwei Aufgaben: das Bauen der Software und das Deployment in Produktion. Ohne diese beiden Tätigkeiten würde eine Software nicht die produktive Umgebung erreichen und somit nicht für den Kunden verfügbar sein.

Im ersten Schritt kann der Build-Prozess der Software automatisiert werden. Dies ist ebenso Voraussetzung für Continuous Integration (CI) und hat sich in den letzten Jahren bereits in vielen Unternehmen etabliert. Speziell für diese Aufgabe gibt es einige Systeme am Markt wie Gradle⁵ oder Maven⁶, die unter anderem das Abhängigkeitsmanagement erleichtern. Mit ihnen wird der Build-Prozess formalisiert und textuell erfasst. Jede Änderung an diesem Prozess kann in einem Versionsverwaltungssystem (VCS) erfasst werden. Zusätzlich dazu wird noch ein CI-Server benötigt, der den Build-Prozess ausführt und die Ergebnisse verwaltet. Ein verbreitetes Beispiel hierfür ist die freie Software Jenkins⁷. Somit kann jede Änderung am Source Code sofort und reproduzierbar gebaut werden. Jedes Software Artefakt kann ohne manuellen Aufwand genau reproduziert werden.

Als nächstes muss das Deployment des Software Artefaktes automatisiert werden. Hierfür ist es nötig, ein sogenanntes Konfigurationsmanagement-System einzuführen. In diesem System wird der Prozess des Deployments formalisiert und textuell erfasst. Der Prozess wird dadurch jederzeit reproduzierbar. Bei der Umsetzung muss unbedingt beachtet werden, dass das Deployment idempotent gestaltet wird. Das bedeutet, dass kein bestimmter Zustand angenommen werden darf. Nur der Endzustand ist relevant. Ein wiederholtes Ausführen des Prozesses muss als Ergebnis immer den gewünschten Endzustand haben. Damit wird erreicht, dass der Prozess jederzeit ausgeführt werden

⁵<https://gradle.org/docs/current/userguide/userguide>, Build Managment System

⁶<https://maven.apache.org/>, Build Managment System

⁷<https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>, erweiterbarer CI/CD-Server

kann. Es ist kein Vorwissen über den Zustand des Zielsystems nötig und ein Deployment kann deswegen jederzeit durchgeführt werden.

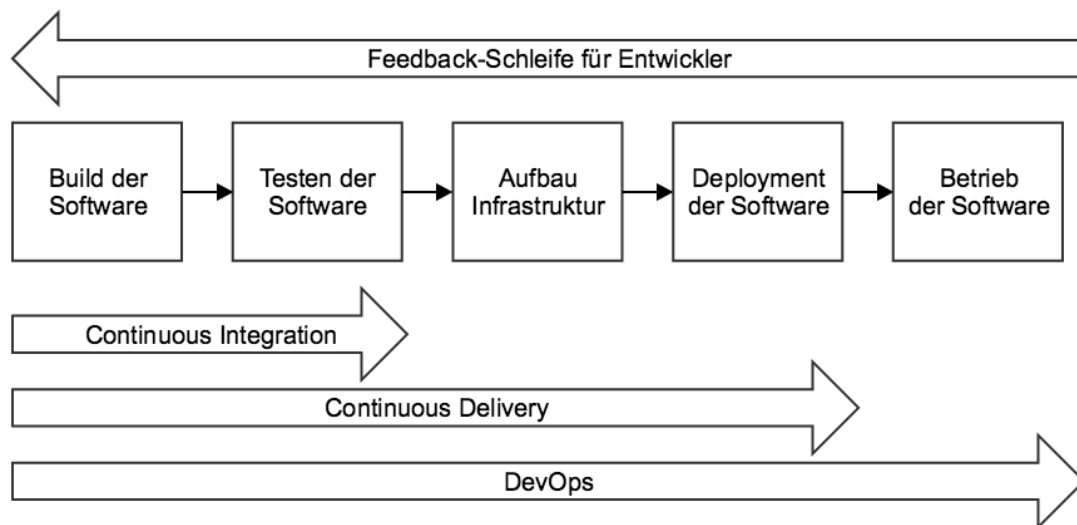


Abbildung 2.3: Grundsätzliche Aufgaben die im Software Releasezyklus automatisiert werden müssen, um die Effizienz zu steigern.

Beispiele für solche Konfigurationsmanagement-Systeme sind Ansible⁸, Puppet⁹ oder Chef¹⁰. Sie bieten bereits viele Konzepte, um die Entwicklung von idempotenten Prozessen zu erleichtern. Wie auch beim Build-Prozess wird ein System benötigt, das diese Prozesse ausführt und die Ergebnisse verwaltet. Ansonsten wäre zwar der Prozess selbst automatisiert, aber man könnte nicht reproduzieren, wann der Prozess ausgeführt hat. Die Nachvollziehbarkeit geht verloren. Für diese Aufgabe kann ebenfalls Jenkins verwendet werden, da dieser durch Plugins von einem CI-Server zu einem Continuous Deployment Server erweitert werden kann.

Dies sind aber nur die zwei grundlegenden Aufgaben, die in jedem Fall automatisiert werden müssen [8, 23]. Es gibt noch viele weitere Aufgaben, die beachtet werden müssen. Abbildung 2.3 zeigt typische Aufgaben, die in vielen Prozessen eines Software-Releasezyklus vorkommen. So muss auch die Infrastruktur für die Software automatisiert bereitgestellt (Abschnitt 2.2.4) oder das Testen der Software effizienter gestaltet werden (Abschnitt 2.2.3). Die Zusammenführung aller automatisierten Aufgaben in einen reproduzierbaren Prozess wird oft als Delivery Pipeline bezeichnet. Auf diese Pipeline wird in Abschnitt 2.2.5 noch genauer eingegangen.

⁸<http://docs.ansible.com/>

⁹<https://puppetlabs.com/puppet/what-is-puppet>

¹⁰<https://docs.chef.io/>

2.2.3 Automatisiertes Testen

Der meiste Aufwand beim Testen eines neuen Releases fällt für Regressionstests an. Je umfangreicher die Applikation, desto mehr Aufwand ist hier pro Release nötig. Das Testen von neuen Funktionen beträgt oft nur einen Bruchteil des Testaufwands. Durch DevOps und Continuous Delivery werden die Releasezyklen verkürzt und die in Abschnitt 2.1.2 beschriebenen Probleme gelöst. Allerdings hat das zur Folge, dass häufiger getestet werden muss. Hierbei ist manuelles Testen, im besonderen für Regressionstest, nicht mehr effizient genug und hat hohe Kosten zur Folge. Außerdem haben manuelle Tests den gravierenden Nachteil, dass sie nicht eindeutig nachvollziehbar sind. Die durchgeführten Schritte können bei der Fehlersuche nicht mehr genau reproduziert werden. Aus diesen Gründen müssen auch die Tests automatisiert werden, um sie schneller und öfter durchführen zu können und gleichzeitig den Aufwand zu reduzieren [23].

Bei DevOps und Continuous Delivery werden bei jeder Änderung der Software die Tests durchgeführt. Das ermöglicht, dass der Entwickler sofort Feedback bekommt, ob seine Änderung einen Fehler verursacht. Ist dies der Fall, kann durch die geringe Menge an Änderungen die Fehlerquelle leicht identifiziert werden. Um eine gute Aussagekraft der Tests zu erreichen, müssen sie möglichst alle Bereiche der Applikation abdecken. Laut [23] sind folgende Typen von Tests maßgeblich zu beachten:

- **Unit Tests:** Sie sind die einfachsten und schnellsten Tests und stellen die grundlegende Funktionalität von Modulen sicher. Mit ihnen wird die interne Implementierung von Modulen überprüft, womit sie zu den White-Box-Tests gehören. Sie prüfen allerdings immer nur ein Modul und nicht das Zusammenspiel mehrerer Module. Dies muss durch Systemtests abgedeckt werden. Ein Beispiel für eine Technologie für den Einsatz von Unit Tests ist JUnit.¹¹
- **Systemtests:** Sie prüfen ein System als Ganzes und somit das Zusammenspiel mehrerer Komponenten. Hier können auch externe Systeme miteinbezogen werden. Da die interne Implementierung irrelevant ist, zählen Systemtests meist zu den Black-Box-Tests. Sie sind eine der wichtigsten Testtypen, da mit ihnen die korrekte Funktionsweise einer Applikation überprüft wird.

Beiden Typen von Tests können laut [23] auf folgende Arten durchgeführt werden. Diese Arten stehen orthogonal zu den Testtypen und ermöglichen kombiniert, in den meisten Fällen, eine gute Testabdeckung einer Applikation.

- **Funktionale Tests:** Sie testen die Applikation auf funktionale Anforderungen und somit auf die Erfüllung bestimmter Anwendungsfälle. Funktionale Tests fungieren oft als Akzeptanztests. Akzeptanztests sind vor allem für die Anwender

¹¹<http://junit.org/>, Unit Test Framework für Java Applikationen

von Relevanz. Sofern sie zusammen mit den Anwendern definiert und akzeptiert wurden, können sie auch als Abnahmekriterium für die Applikation dienen. Sind die Akzeptanztests erfüllt, so hat die Applikation eine ausreichend hohe Qualität für den Einsatz.

Die Durchführung kann über spezielle APIs oder die Benutzeroberfläche (GUI) erfolgen. Tests über die GUI sind allerdings fragiler, als Tests über eine definierte API. Sie können bereits durch einfache Designänderungen der Oberfläche gebrochen werden, obwohl kein Fehler vorliegt. Für GUI Tests kann beispielsweise Selenium¹² und für API Tests SoapUI¹³ verwendet werden.

- **Nicht-Funktionale Tests:** Diese kontrollieren die Applikation auf bestimmte Qualitätskriterien. Kapazitätstests überprüfen beispielsweise die Performanz einer Applikation. Sie gestalten sich allerdings schwierig, da Applikationen meist nicht linear auf Last reagieren. Daher werden Testreihen durchgeführt, um festzustellen, ab welcher Last die Performanz einbricht. Die getestete Umgebung muss für aussagekräftige Resultate sehr ähnlich der produktiven Umgebung sein. Das ist aus wirtschaftlichen Gründen oft nicht möglich. Daher sollte zusätzlich ein entsprechendes Monitoring installiert werden, das die aktuelle Performanz des Systems misst und potentielle Performanzprobleme größtenteils voraussagt. So bleibt ausreichender Handlungsspielraum um darauf reagieren zu können. Für Kapazitätstests können unter anderem FunkLoad¹⁴ oder Gatling¹⁵ verwendet werden.
- **Explorative Tests:** Die bisher genannten Arten von Tests sind vor allem für Regressionstests gebräuchlich. Explorative Tests beschäftigen sich hingegen mit neuen Funktionalitäten und nicht quantitativ messbaren Aspekten, wie die Verwendbarkeit der Applikation. Diese Art von Test lässt sich allerdings nur schwer automatisieren. Daher können sie in vielen Fällen nur manuell durchgeführt werden. Da sich der zeitliche Aufwand für automatisierte Regressionstests auf ein Minimum begrenzt, steht für exploratives Testen ausreichend Zeit zur Verfügung. Neue Funktionen erfordern allerdings nach geraumer Zeit die Überführung explorativer in automatisierte Tests.

Eine von [23] empfohlene Verteilung und Gewichtung der unterschiedlichen Arten von Tests für viele Anwendungsfälle ist in Abbildung 2.4 dargestellt.

Die meisten Tests sollten über Unit-Tests erfolgen. Sie bilden die Basis für eine funktionierende Applikation. Darauf aufbauend sollte es eine große Menge an funktionalen und nicht-funktionalen Tests geben. Für eine höhere Stabilität der Tests sollten diese

¹²<http://www.seleniumhq.org/>, Testing Framework zur Simulation von Benutzerinteraktionen

¹³<http://www.soapui.org/about-soapui/what-is-soapui.html>, Testing Framework für APIs

¹⁴<http://funkload.nuxeo.org/intro.html>, Framework für Funktionale und Performanztests

¹⁵<http://gatling.io/docs/2.1.6/>, Framework spezialisiert auf Performanztests

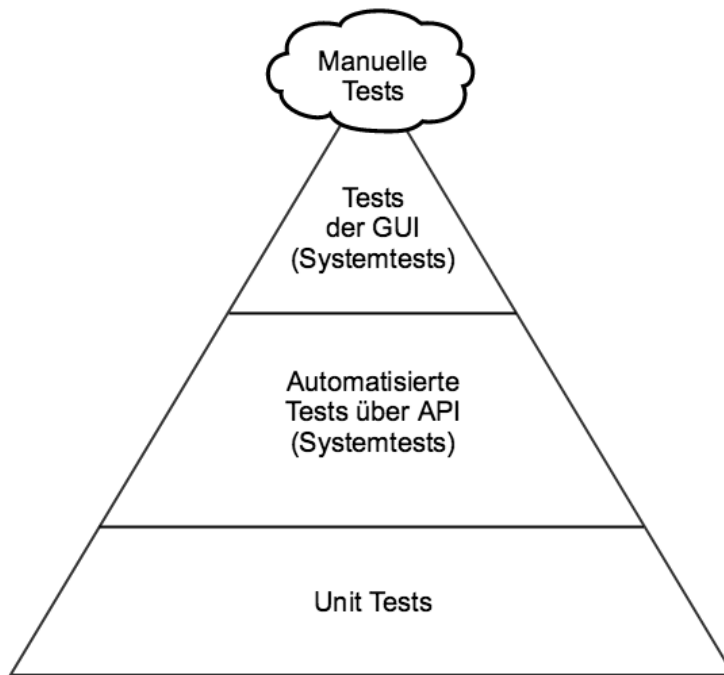


Abbildung 2.4: Empfohlene Verteilung der Tests für viele Applikationen, visualisiert Anhand der Testpyramide, adaptiert aus [23]

eine API der Applikation nutzen. Zusätzlich kann es noch Tests über die GUI geben, um diese mittels automatisierten Tests zu überprüfen. Ansonsten kann das Backend der Applikation fehlerfrei sein, die dazugehörige GUI aber nicht korrekt funktionieren. Den kleinsten Anteil bilden manuelle Tests, zur explorativen Überprüfung der Applikation. Sie sollten in sehr eingeschränktem Ausmaß nötig sein.

Die automatisierten Tests decken den Bereich der Regressionstests ab und testen vor allem bestehende Funktionalitäten. Da sie meist den Großteil des Testaufwands ausmachen, steckt hier viel Optimierungspotential. Ein Sonderfall, bei dem manuelles Testen nötig ist, sind neue Funktionalitäten. Dies kann allerdings bereits in der Entwicklung berücksichtigt werden, indem Test Driven Development¹⁶ betrieben wird. Dabei werden zuerst die Tests geschrieben und die Implementierung ist erst abgeschlossen, wenn die Tests fehlerlos ausgeführt werden. Dies setzt aber voraus, dass die Abnahmekriterien bereits vor der Entwicklung feststehen.

Automatisierte Tests müssen allerdings nicht allumfassend sein. Jeden Grenzfall zu testen würde zuviel Aufwand in Anspruch nehmen, manuell wie auch automatisiert. Daher sollten die Tests die Grundfunktionalitäten der Applikation sicherstellen. Sofern alle Tests erfolgreich ausgeführt werden, sollte die Applikation eine ausreichend hohe Qualität für den Einsatz in Produktion besitzen. Fehler können nie vollständig ausgeschlossen werden, aber durch DevOps und Continuous Delivery ist die Möglichkeit

¹⁶<http://www.infoq.com/presentations/tdd-original>, Einstündige Einführung in TDD

gegeben, schnell auf sie reagieren zu können. Für Software-Bereiche mit besonders hohen Qualitätsanforderungen, wie beispielsweise Medizintechnik-Software, könnte dieser Ansatz nicht ausreichend passabel sein. Hierbei empfiehlt es sich, das Deployment bis zur Testumgebung zu automatisieren. Das Deployment auf die produktive Umgebung kann per manuellen Trigger durchgeführt werden. Automatisierte Tests können aber auch hier den Aufwand des Testens erheblich verringern.

2.2.4 Infrastruktur Management (Infrastructure as Code)

Bei der Bereitstellung der grundlegenden Infrastruktur für den Betrieb einer Applikation können manuelle Fehler passieren (siehe Abschnitt 2.1.3). Das betrifft nicht nur das Betriebssystem einer Maschine, sondern auch etwaige Drittsysteme, die für eine Applikation benötigt werden. Diese Probleme werden gelöst bzw. verringert, wenn die Infrastruktur automatisiert erstellt wird. Das in Abschnitt 2.2.2 beschriebene Konfigurationsmanagementsystem kann dafür eingesetzt werden.

Die Erstellung einer Infrastruktur oder die Installation eines Drittsystems kann als Deployment einer Software angesehen werden [23]. In Zeiten der Virtualisierung ist die Zielmaschine nur mehr Software und kann automatisiert deployed werden. Es müssen also Prozesse im Konfigurationsmanagement-System geschaffen werden, um die Erstellung der virtuellen Maschine, inklusive Installation des Betriebssystems und etwaiger Drittsysteme, zu übernehmen. So ist sichergestellt, dass dieser Zustand der Infrastruktur jederzeit wiederhergestellt werden kann. Bei einem Cluster wird außerdem jede Maschine gleich erstellt. Eine unterschiedliche Konfiguration einzelner Maschinen ist nicht mehr möglich. Zusätzlich hat man die Möglichkeit, komplette Umgebungen ohne manuellen Aufwand und mit geringen Kosten zu erstellen. So können für einzelne komplexe Testszenarien spezielle Umgebungen aufgebaut oder Zero-Downtime-Release (Abschnitt 2.2.5) durchgeführt werden.

Ein wichtiger Aspekt hierbei ist, dass Personen keine Änderungen mehr manuell auf Maschinen durchführen dürfen. Jede Änderung einer Konfiguration des Betriebssystems, einer Drittsoftware oder ähnlichem darf nur mehr über den automatisierten Prozess erfolgen. Ansonsten können sich Konfigurationen unterscheiden bzw. kann eine Änderung nach erneutem Deployment wieder verschwunden sein. Wenn das Konfigurationsmanagementsystem in einem VCS eingchecked ist, hat man den Vorteil, dass jede Änderung anhand der Historie ersichtlich ist. Man kann nachvollziehen wer welche Änderung wann durchgeführt hat, im Gegensatz zu den manuellen Tätigkeiten. Daher die Bezeichnung Infrastructure-as-Code, da die Infrastruktur wie Source Code behandelt wird [23].

Ähnlich wie in Abschnitt 2.2.2 muss auch das Deployment der Infrastruktur idempotent gestaltet werden, damit der Prozess jederzeit ausführbar ist. Bei einer kompletten Infrastruktur müssen wesentlich mehr Bereiche beachtet werden, als beim Deployment einer einzelnen Applikation. Bei einer Applikation sind oft nur wenige Teile und Konfigurationen betroffen, die im Konfigurationsmanagementsystem verwaltet werden. Bei einer Infrastruktur ist die Zahl der Konfigurationen, die verändert werden können, wesentlich höher. Alle Konfigurationen im Prozess zu beachten, würde diesen unnötig verkomplizieren. Um dies zu lösen, hat [4] im Jahr 2012 den sogenannten Phoenix Server beschrieben. Dies bezeichnet einen Server, der jederzeit abgerissen und neu aufgebaut werden kann, um wieder einen genau definierten Zustand zu erreichen. Somit werden auseinander laufende Konfigurationen verhindert. Darauf aufbauend beschrieb [14] den Immutable¹⁷ Server. Ein Server der nach initialer Erstellung nicht mehr verändert werden darf. Sind Änderungen notwendig, so muss der aktuelle Server durch einen Neuen, mit aktualisierter Konfiguration, ersetzt werden. Bei diesem Ansatz werden nachträgliche Fehlkonfigurationen vermieden. Ein weiterer Vorteil ist, dass der Prozess zum Erstellen der Infrastruktur regelmäßig getestet und so zur Routinearbeit wird.

Neben der Bereitstellung der Infrastruktur müssen weitere infrastrukturelle Aufgaben betrachtet werden. Für die Installation des Betriebssystems muss dieses in Form eines Image bereitgestellt werden. Dieses muss genau definiert sein und soll wiederum automatisiert erstellt werden. Darüber hinaus ist es von Vorteil, sich selbst verwaltende Applikationen zu verwenden, um manuelle Arbeit zu vermeiden. Eine Applikation, die beispielsweise oft neu konfiguriert wird, ist der Loadbalancer. Jedes mal wenn eine neue Maschine erstellt bzw. eine gelöscht wird, muss die Konfiguration des Loadbalancers entsprechend angepasst werden. Deshalb sollte die Konfiguration des Loadbalancer automatisch erzeugt werden. Dafür ist eine sogenannte Service Registry notwendig, in der alle vorhandenen Maschinen verzeichnet sind. Diese kann beispielsweise mit Hilfe von Consul¹⁸ aufgebaut werden. Aus der Service Registry können anschließend Konfigurationen vielfältig erzeugt werden.

2.2.5 Die Deployment Pipeline

Die einzelnen Phasen einer Software, von der Fertigstellung der Entwicklung bis zum Deployment in der produktiven Umgebung, werden in einer sogenannten Deployment Pipeline zusammengefasst [8]. Eine Applikation muss die Deployment Pipeline vollständig durchlaufen, bis sie produktiv eingesetzt wird. Die Pipeline kann je nach Unternehmen und Applikation im Detail unterschiedlich aufgebaut sein. Allerdings lässt sich jede

¹⁷deutsch: unveränderbar

¹⁸<https://www.consul.io/intro/>, verteilte Service Registry von Hashicorp

Pipeline auf das in Abbildung 2.5 gezeigte grundsätzliche Schema reduzieren.

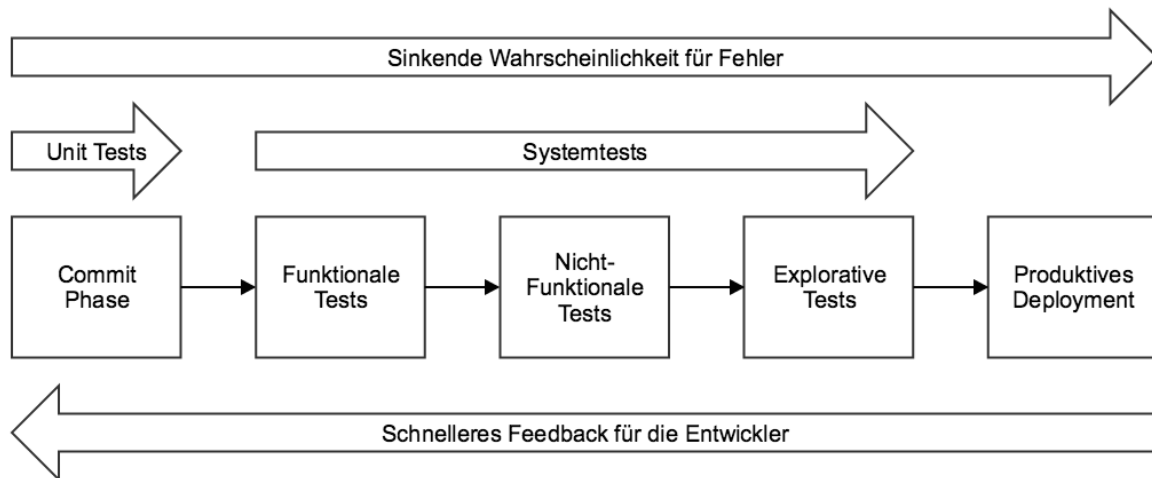


Abbildung 2.5: Grundsätzliches Schema einer Deployment Pipeline. Jede abgeschlossene Phase steigert die Produktionsreife der Applikation. Adaptiert aus [8] und [23].

Durch diesen Aufbau garantiert die Pipeline, dass die Entwickler so schnell wie möglich Feedback zu gemachten Änderungen bekommen. Fehler werden frühestmöglich erkannt und können daher leichter und kosteneffizienter behoben werden. Die einzelnen Phasen erfüllen laut [8] und [23] folgende Aufgaben:

- **Commit Phase:** Diese Phase ist der Start der Deployment Pipeline und wird getriggert, sobald die Entwickler eine Änderung in das VCS einchecken. Jeder Commit löst eine neue Instanz der Pipeline aus. Zuerst wird die Applikation gebaut und ein Artefakt erzeugt. Sobald dies abgeschlossen ist, werden die Unit-Tests ausgeführt. Nur wenn alle diese Prozesse erfolgreich durchgeführt sind, darf die Commit Phase abgeschlossen werden. Das Ergebnis ist ein Software Artefakt, das ein Mindestmaß an Qualität aufweist und keine groben Fehler enthält. Üblicherweise steht das Ergebnis der Commit Phase nach wenigen Minuten fest. Die Entwickler bekommen hierbei schnellstmöglich Feedback.
- **Funktionale Tests:** Anschließend wird das Artefakt in einer möglichst produktionsähnlichen Umgebung deployed und funktionale Systemtests durchgeführt. Sofern dies weitgehend automatisiert ist, kann für jede Instanz der Pipeline eine eigene Testumgebung aufgebaut werden. Sind die Systemtests erfolgreich, so erfüllt das Artefakt jedenfalls die technischen und funktionalen Anforderungen und weist somit bereits eine hohe Qualität auf. Üblicherweise sollte das Ergebnis dieser Phase in weniger als einer Stunde feststehen. Die Entwickler können sich dann weitgehend sicher sein, dass ihre Arbeit eine ausreichende Qualität aufweist. Im Gegensatz dazu erfordert dies bei klassischen Prozessen meist einige Tage bis Wochen.

- **Nicht-Funktionale Tests:** Nachdem die funktionale Qualität sichergestellt ist, sollten auch nicht-funktionale Systemtests durchgeführt werden. Üblicherweise sind dies Performanz- und Kapazitätstests. Diese garantieren beispielsweise, dass durch eine Änderung, die Performanz nicht drastisch beeinträchtigt wurde. Nach Abschluss dieser Phase hat das Artefakt meist eine ausreichend hohe Qualität für ein Deployment in der produktiven Umgebung. Die Wahrscheinlichkeit für einen Fehler ist bereits auf ein Minimum reduziert. Das Ergebnis dieser Phase sollte üblicherweise nach einem Tag zur Verfügung stehen.
- **Explorative Tests:** Diese Phase kann als optional angesehen werden, allerdings ist sie Bestandteil vieler Pipelines. Hier wird das Artefakt manuell getestet, um beispielsweise neue Funktionalitäten zu überprüfen. Bis hierher wurde die Pipeline vollständig automatisch durchlaufen. Diese Phase muss jedoch manuell bestätigt werden um sie erfolgreich abzuschließen. Sind hingegen keine explorativen Tests notwendig, kann die Pipeline bis zum Deployment auf Produktion vollständig automatisch durchlaufen werden.
- **Produktives Deployment:** In der letzten Phase wird das getestete und mittlerweile qualitativ hochwertige Artefakt in der produktiven Umgebung deployed. Oftmals ist der Beginn dieser Phase eine manuelle Bestätigung und nicht der erfolgreiche Abschluss der vorangegangenen Phase. Insbesondere dann, wenn die Pipeline eine Phase für exploratives Testen erfordert. Ist die Pipeline hingegen vollständig automatisiert, sind selbst vielzählige Deployments pro Tag realistisch¹⁹.

Wenn eine der oben genannten Phasen nicht erfolgreich abgeschlossen wird, muss die Deployment Pipeline abgebrochen werden, da es keinen Sinn macht, weitere Phasen zu durchlaufen. Bei Abbruch müssen die zuständigen Entwickler entsprechend benachrichtigt werden, um auf das negative Ergebnis zu reagieren. Je weiter die Pipeline zum Zeitpunkt des Abbruchs durchlaufen wurde, desto länger mussten die Entwickler auf Feedback warten. Allerdings sinkt mit jeder abgeschlossenen Phase die Wahrscheinlichkeit für Fehler. Daher können die Entwickler bereits eine Stunde nach erfolgreichem Commit, mit hoher Wahrscheinlichkeit davon ausgehen, keine Fehler verursacht zu haben. Der Fokus kann also auf die nächste Aufgabe gelegt werden.

Eine Grundbedingung einer Deployment Pipeline ist, dass das gebaute Artefakt aus der Commit Phase in jeder weiteren Phase verwendet wird. Das Artefakt darf pro Instanz der Pipeline nur ein einziges mal gebaut werden. Es darf also nicht für jede Phase ein eigenes Artefakt gebaut werden. In diesem Fall könnte durch den erneuten Buildprozess ein Fehler entstehen, der durch die Tests der Commit Phase aufgefallen

¹⁹<http://www.infoq.com/presentations/Continuous-Deployment-50-Times-a-Day>

wäre, später aber nicht mehr entdeckt wird. Im schlimmsten Fall wird dieser Fehler dann in Produktion deployed.

Durch die Verwendung einer Deployment Pipeline ergeben sich zusammenfassend folgende Vorteile:

- Die Feedback-Zyklen für die Entwickler reduzieren sich auf ein Minimum. Fehler können schneller gefunden und mit weniger Kosten behoben werden.
- Die Deployment-Prozesse werden laufend getestet, da bei der Transition zwischen den Phasen ein Deployment auf Testumgebungen erfolgt. Dadurch erhöht sich dessen Sicherheit und das Fehlerrisiko sinkt.
- Die Qualität der Software wird durch die vielen Testphasen zu einer hohen Wahrscheinlichkeit sichergestellt.

Deployment Möglichkeiten: Kritischste Phase der Pipeline ist das Deployment in die produktive Umgebung. Obwohl der Erfolg eines Deployments durch Automatisierung nahezu sichergestellt wird, können Fehler nie vollständig ausgeschlossen werden. Um das Risiko zu minimieren, gibt es mehrere Möglichkeiten, wie ein Deployment durchgeführt werden kann. Laut [8] werden folgende drei Vorgehensweisen am häufigsten eingesetzt:

- **Rollout:** Dies ist die einfachste Variante für ein Deployment. Hierbei wird das gesamte System mit einer neuen Version aktualisiert. Dies geschieht meist während eines Wartungsfensters, bei dem die Applikation nicht erreichbar ist. Nach dem Rollout ist ausschließlich die neue Version für die Benutzer aktiv. Gibt es Fehler nach dem Deployment kann ein Rollback durchgeführt werden. Dies hat aber wiederum einen Ausfall der Applikation zur Folge.
- **Blue/Green Deployment:** Hierbei wird die neue Version auf einer komplett getrennten Umgebung deployed. Ist das Deployment abgeschlossen, wird auf diese Umgebung umgeschaltet. Ab diesem Zeitpunkt verwenden alle Benutzer die neue Version. Es entsteht also keine Ausfallzeit (Zero-Downtime-Release). Diese Methode bedingt einen hohen Ressourcenverbrauch, da zwei produktive Umgebungen betrieben werden müssen. Wird außerdem eine Datenbank verwendet, so müssen die Datenbanken der beiden Umgebungen synchron gehalten werden.
- **Canary Release:** Bei dieser Vorgehensweise wird die neue Version auf wenige Maschinen im Cluster deployed. Sie wird dadurch nur von einem Bruchteil der Benutzer verwendet und so in Produktion getestet. Ein Rollback ist sehr einfach durchzuführen, indem die aktualisierten Maschinen wieder entfernt werden. Funktioniert die Version, so kann sie ohne Ausfallzeit sukzessive auf alle Maschinen

ausgerollt werden. Diese Vorgehensweise ist jedoch für viele Arten von Applikationen nicht praktikabel, wenn beispielsweise die neue Versionen ein angepasstes Datenbankschema benötigt.

Abgesehen vom klassischen Rollout kann mit Blue/Green Deployments bzw. Canary Releases neben der Risikominimierung auch ein Rollback vereinfacht werden. Allerdings haben diese beiden einen wesentlich höheren Ressourcenverbrauch als der klassische Rollout.

2.2.6 Unternehmenskultureller Aspekt von DevOps

In Abschnitt 2.1.4 wurden Probleme beschrieben, die durch die strikte Trennung von Entwicklung und Betrieb entstehen. Die Konzepte von DevOps beschäftigen sich neben den technologischen Aspekten auch mit diesen Problemen. Laut [22] ist es für den Erfolg von DevOps essentiell, die Zusammenarbeit zwischen Entwicklung und Betrieb zu verbessern. Ein großes Hindernis bei der Zusammenarbeit sind die unterschiedlichen Ziele und Werte der Teams. Sie müssen für eine erfolgreiche Zusammenarbeit einheitlich definiert werden. Hierbei muss ein Kompromiss zwischen Stabilität und flexibler Entwicklung gefunden werden, bei dem der Schwerpunkt entsprechend den Anforderungen des Unternehmens individuell definiert werden kann. Dabei sollte laut [23] für beide Teams der Service immer im Vordergrund stehen, um diesen bestmöglich an die Kunden zu liefern.

In der Literatur zu DevOps, wie beispielsweise [9, 17, 21], wird eine verbesserte Kommunikation als Schlüsselement angeführt. Die Barrieren zwischen Entwicklung und Betrieb müssen abgebaut und die Kommunikation verbessert werden. Bei Problemen ist gegenseitige Schuldzuweisung, die häufig Praxis ist, unter allen Umständen zu vermeiden. Für die Kunden ist es nicht relevant, wer einen Fehler verursacht hat. Deswegen müssen die Mitarbeiter einen lösungsorientierten Ansatz verfolgen. Wichtigstes Ziel ist es, den Fehler gemeinsam so schnell wie möglich zu beheben. Es ist lediglich wichtig, wie der Fehler entstanden ist und wie dieser in Zukunft vermieden werden kann. Das wird auch im serviceorientierten Ansatz von [23] angeführt.

Um eine effiziente Zusammenarbeit zu erreichen, muss neben verbesserter Kommunikation das vorhandene Wissen beider Teams ausgetauscht werden. Dies schafft Synergien, um Applikationen effizienter betreiben zu können. Einerseits sollen die Entwickler in den Betrieb mit eingebunden werden und Verantwortung für erstellte Applikationen übernehmen. Das Betriebsteam schafft die Grundlagen für die Entwickler und unterstützt sie, um ihre Applikationen selbst bestmöglich betreiben können. Andererseits sollen Entwickler das Betriebsteam bereits in der Planungsphase der Applikation mit

einbeziehen. Dadurch kann der Betrieb bereits in der Entstehung wichtige Vorschläge liefern, um die Betriebstauglichkeit der Applikation zu erhöhen. Weitere Vorteile die sich daraus ergeben sind, dass der Betrieb beim Aufbau lokaler Entwicklungsumgebungen behilflich sein kann und somit diese schon mehr an die produktive Umgebung angleichen. Die Entwicklung wird dadurch wesentlich effizienter. Außerdem hat die Entwicklung mehr Freiheiten bei der Wahl von Technologien und dem Aufbau der Infrastruktur. Sie müssen allerdings auch entsprechend die Verantwortung für deren Betrieb übernehmen. Diese Vorgehensweise setzt aber ein hohes Maß an Vertrauen und Respekt untereinander voraus. Die kulturelle Anpassung an DevOps ist deswegen ein langfristiger Prozess, der Zeit und entsprechende Führung durch Vorgesetzte benötigt. Nur wenn diese neue Unternehmenskultur von allen Ebenen unterstützt und gelebt wird, ist ein erfolgreiches Umdenken möglich [22].

Betrachtet man viele Unternehmen, so sind Entwicklung und Betrieb oft getrennt. Eine Praxis die [23] empfiehlt ist, diese beiden Bereiche zu vereinen und in Teams aufzuteilen. So werden kleinere Teams gebildet, die jeweils für die Entwicklung **und** den Betrieb einer Komponente²⁰ verantwortlich sind. Das jeweilige Team hat die Entscheidungskompetenz über eingesetzte Technologien und deren Betrieb. Es muss nur sicherstellen, dass die Komponente bestmöglich bereitgestellt wird. Daher ist es wichtig, die Schnittstellen zwischen den einzelnen Komponenten zu definieren und entsprechend einzuhalten. Dieser Ansatz setzt eine große Eigenverantwortung des jeweiligen Teams voraus. Man sollte besonders darauf achten, dass zwischen den Teams reger Austausch von Wissen und Zusammenarbeit herrschen. Um dies zu erreichen, ist es sinnvoll regelmäßige Wechsel von Personen zwischen den Teams zu ermöglichen bzw. zu forcieren.

Mit den technologischen Aspekten von DevOps kann bereits eine wesentliche Steigerung der Effizienz erreicht werden. Für die Lösung aller in Abschnitt 2.1 genannten Probleme, muss aber auch die Firmenkultur mit einbezogen werden. Nur wenn man diese beiden Komponenten entsprechend verbindet, können die Potenziale von DevOps zur Gänze genutzt werden.

²⁰Je nach Größe des Unternehmens und der Plattform muss eine sinnvolle Trennung gefunden werden. Dieser Ansatz steht vor allem im Bezug auf die Microservice Bewegung, bei der große Monolithen in kleinere Services aufgeteilt werden. Ein Team kümmert sich vollständig um einen Microservice.

Kapitel 3

Methodik

Im folgenden Kapitel wird die, in vorliegender Arbeit angewandte, systematische Vorgehensweise von Goal-Question-Metric (GQM) beschrieben. Zuerst werden die einzelnen Punkte von GQM aufgezeigt. Anschließend wird die durchgeführte Fallstudie, sowie deren Rahmenbedingungen beschrieben. Am Ende des Abschnitts wird der Ist-Zustand aufgezeigt, um ein besseres Bild von der Ausgangslage darzustellen.

3.1 Goal-Question-Metric

Als Erstes wird die gewünschte Zielsetzung der Arbeit beschrieben. Dann wird auf die Fragen, die für das Erreichen des Zieles beantwortet werden müssen, näher eingegangen. Sie stellen die Forschungsfragen der Arbeit dar. Der letzte Punkt behandelt die definierten Metriken, die eine quantitative Beantwortung der Fragen ermöglichen.

3.1.1 Problemstellung und Zielsetzung

Viele große Unternehmen, wie beispielsweise Google [10, 12], Facebook [3] oder Amazon [11], betreiben ihre Infrastruktur bereits erfolgreich nach den Konzepten von DevOps. Sie waren auch maßgeblich bei der Entwicklung dieses Trends beteiligt. Allerdings haben diese Unternehmen eine große Anzahl von Mitarbeitern um DevOps umzusetzen und erfolgreich einzusetzen. Je mehr verschiedene Projekte betrieben werden und je inhomogener die Infrastruktur ist, desto mehr Personen werden benötigt, um die automatisierten Prozesse zu warten. Außerdem sind grundlegende Änderungen an der vorherigen Handhabung der Software, sowie der Infrastruktur notwendig, um die Konzepte von DevOps erfolgreich umzusetzen. Dies ist für kleine Unternehmen eine große

Herausforderung, da meist nicht so viele Mitarbeiter für diese Umstellung zur Verfügung stehen, der initiale Aufwand der Implementierung aber erheblich sein kann.

Das Ziel vorliegender Arbeit ist, eine neue Infrastruktur und Prozesse aufzubauen, die eine vollständige Automatisierung der Build und Deployment Pipeline (Abschnitt 4.2.2) ermöglichen und umsetzen. Die Infrastruktur und die darin enthaltenen Applikationen sollen vollständig nach den Konzepten von DevOps entwickelt und betrieben werden. Damit werden manuelle Tätigkeiten in diesen Prozessen minimiert oder eliminiert, und somit höhere Sicherheit gegenüber Fehlern gewährleistet und Ressourcen einspart. Bei gleichbleibender Effektivität soll daher die Effizienz des Betriebs der Software erheblich gesteigert werden. Somit können entweder Ressourcen eingespart oder auf andere Tätigkeiten verlagert werden. Für die komplette Umsetzung soll ein Team, bestehend aus drei Mitarbeitern, und ein Zeitraum von etwa einem halben Jahr ausreichen.

3.1.2 Forschungsfragen

Um das angestrebte Ziel zu erreichen, werden im Laufe der Arbeit folgende Fragen genauer betrachtet. Sie stellen gleichzeitig die Forschungsfragen der Arbeit dar.

Die grundlegende Frage lautet, wie viel Aufwand die Implementierung der Konzepte von DevOps und Continuous Delivery grundsätzlich benötigt. Die Automatisierung der Build und Deployment Pipeline muss für jede Applikation separat durchgeführt werden. Daher wird ermittelt, ob dieser Aufwand pro Applikation gleich ist oder ob sich dieser mit steigender Anzahl der Applikationen reduziert/erhöht. Außerdem wird ermittelt, ob unterschiedliche Applikationen in etwa den gleichen Aufwand für die Automatisierung benötigen. Dafür werden nach Möglichkeit Charakteristiken herausgefunden, anhand derer sich der erwartete Implementationsaufwand grob abschätzen lässt.

Um Applikationen professionell nach den Konzepten von DevOps zu betreiben, wird eine gewisse Basisinfrastruktur benötigt. Dazu zählen beispielsweise die benötigten Systeme oder entsprechende Konzepte. Die Basisinfrastruktur muss ebenfalls aufgebaut werden. Hier wird ermittelt, welche Infrastruktur grundsätzlich notwendig ist und entsprechende Konzepte erstellt um diese aufzubauen. Darüber hinaus wird aufgezeigt, wie viel Aufwand die Erstellung dieser Infrastruktur benötigt. Weiters wird erfasst, ob der Aufwand für den Aufbau der Basisinfrastruktur einmal initial notwendig ist, oder ob sich dieser mit steigender Anzahl an Applikationen erhöht.

Anhand dieser beiden Fragestellungen soll primär geklärt werden, ob die Einführung der Konzepte von DevOps und die Umstellung der Applikationen von lediglich drei Mitarbeitern, innerhalb eines angemessenen Zeitraums von etwa einem halben Jahr (Beginn 2015 bis Mitte 2015), bewältigt werden kann.

Da DevOps bei gleichbleibender Effektivität vor allem die Effizienz steigert, wird untersucht, wie sich diese nach erfolgter Umstellung für die Applikationen verändert. Dabei wird vor allem die Durchlaufzeit, von der fertigen Entwicklung bis zum Betreiben der Applikation im produktiven System, betrachtet. Sofern es sich um bereits im Einsatz befindliche Applikationen handelt, wird ein Vorher/Nacher-Vergleich aufgestellt.

Zusammenfassung der Forschungsfragen:

- Wie viel Implementationsaufwand ist für eine Umstellung anhand verschiedener Applikationen erforderlich?
 - Bleibt der Aufwand der Implementierung annähernd gleich oder verändert sich dieser mit Anzahl der Applikationen?
 - Gibt es nutzbare Gemeinsamkeiten bei der Implementation?
 - Gibt es Charakteristiken, anhand derer sich der Aufwand abschätzen lässt?
- Welche Infrastruktur ist grundsätzlich nötig für den Betrieb mit DevOps?
 - Welche neuen Konzepte werden benötigt und wie können diese aussehen?
 - Wie viel Aufwand ist die Erstellung der Konzepte und der Infrastruktur?
 - Ist dieser Aufwand einmalig oder pro Applikation erforderlich?
- Wie verändert sich die Effizienz nach der Umstellung?

3.1.3 Definition der Metriken

Um die Forschungsfragen zu quantifizieren, wurden folgende Metriken definiert:

Initialer Aufwand: Für die Messung des initialen Implementationsaufwands wird erfasst, wie viele Personentage investiert werden, um die Build und Deployment Pipeline einer Applikation zu automatisieren. Dies wird getrennt für die Applikationen und für die Basisinfrastruktur betrachtet. Bei den Applikationen wird erfasst, wie viele Personentage die Umstellung benötigt und wie viele Teile der Automatisierung von anderen Applikationen wiederverwendet werden können. So wird verglichen, ob sich der initiale Aufwand pro weiterer Applikation verändert oder ob dieser konstant bleibt.

Bei der Basisinfrastruktur wird ebenfalls der Aufwand, der für den Aufbau benötigt wird, ermittelt. Der Aufwand wird für die Entwicklung der entsprechenden Konzepte und die Implementation der Automatisierung getrennt erfasst. Weiters wird ermittelt,

wie viel Aufwand pro Applikation an der Basisinfrastruktur nötig ist, um diese entsprechend zu integrieren. Die Angabe des Aufwands erfolgt in Personenstunden (PS), mit 15 Minuten Intervallen als kleinste Einheit.

Durchlaufzeit: Die folgenden Metriken beschreiben die Durchlaufzeiten verschiedener Arten von Deployments von Applikationen, anhand derer, die erreichte Effizienz quantifizieren wird. Die Definitionen der Metriken orientieren sich an den Tätigkeiten, die das Operations-Team regelmäßig manuell durchführt. Es werden folgende drei Arten von Deployments definiert, deren Dauer separat gemessen wird:

- a.) Deployment einer neuen Version der Applikation auf einer bereits vorhandenen Maschine.
- b.) Deployment einer neuen Version der Applikation auf allen bereits vorhandenen Maschinen.
- c.) Deployment einer zusätzlichen Maschine mit der aktuellen Version der Applikation, inklusive aller notwendiger Third-Party-Software, sowie Integration der Maschine in die Basisinfrastruktur.

Sofern eine Applikation bereits nach klassischen Konzepten betrieben wird, werden diese Metriken vor und nach der Umsetzung der Automatisierung erfasst. Aus diesen Durchlaufzeiten wird errechnet, um welchen Faktor sich die Effizienz steigert. Da die Durchlaufzeiten pro Deployment auf Grund vieler Faktoren variieren können, repräsentiert die angegebene Dauer den Durchschnitt von mindestens fünf Deployments, sofern darin keine erheblichen Ausreißer enthalten sind.

Charakteristiken für den Aufwand: Als potentielle Charakteristik zur Ermittlung des erforderlichen Implementationsaufwands könnte die Komplexität einer Applikation herangezogen werden. Um die Komplexität einer Applikation zu quantifizieren wird erfasst, aus wie vielen unterschiedlichen Komponenten eine Applikation besteht. Beispiele für Komponenten sind der Apache Tomcat oder die Java Runtime Umgebung, die für Java Web Applikationen benötigt werden. Die Anzahl der Komponenten wird dem benötigten Aufwand für die Implementierung gegenübergestellt.

3.2 Beschreibung der Fallstudie

Damit die Fragestellungen beantwortet und die definierten Metriken erfasst werden können, wird eine Fallstudie durchgeführt. Dabei wird untersucht, wie sich die Implementierung von DevOps mit einem Team aus drei Mitarbeitern durchführen lässt.

3.2.1 Rahmenbedingungen

Die Fallstudie wird komplett im Umfeld des Red Bull Media House durchgeführt. Die Abteilung Red Bull Media Base, in der der Autor tätig ist, ist zuständig für die Entwicklung der Multi-Media-Asset-Management Plattform des Red Bull Media House. Im Zuge der Arbeit wird die Umsetzung der Konzepte von DevOps und Continuous Delivery anhand der Applikationen der Red Bull Media Base betrachtet. Diese Projekte werden bereits nach agilen Vorgehensmodellen entwickelt, aber noch nach klassischen Vorgehensmodellen betrieben. Das Deployment und der Betrieb der Applikationen werden daher schrittweise, je Applikation separiert, umgestellt. Die gesamte bisherige Infrastruktur der Red Bull Media Base befindet sich in einem Rechenzentrum in Frankfurt. Zusätzlich gibt es noch kleinere Einheiten an einzelnen wichtigen Standorten wie London, Los Angeles, Sydney und Sao Paulo. Ein weiteres größeres Rechenzentrum ist in den USA geplant, das sich allerdings erst im Aufbau befindet. Diese Standorte müssen bei der Entwicklung von Konzepten berücksichtigt werden, da es global verteilte Applikationen im Portfolio der Red Bull Media Base gibt.

Wie in Abschnitt 3.1 erwähnt, ist DevOps bereits in großen Unternehmen im Einsatz. Kleinere Unternehmen mit weniger Mitarbeitern tragen hier aber ein großes Risiko bei der Umsetzung. Um die Umsetzung mit geringer Personenanzahl zu demonstrieren, wurden alle in dieser Arbeit erwähnten Tätigkeiten (außer Abschnitt 4.1) von einem drei Personen Team durchgeführt. Die Durchführung startete zu Beginn des Jahres 2015 und endet mit ca. Ende Juni 2015. Bis dahin wird der Aufbau der Basisinfrastruktur und die Implementation für die Beispielapplikationen abgeschlossen sein.

Für die Umsetzung der Konzepte von DevOps und Continuous Delivery sind grundlegende Änderungen an der Infrastruktur eines klassischen Betriebs notwendig. Diese Änderungen werden im Abschnitt 4.1 als Voraussetzung erläutert. Diese Arbeiten sind sehr speziell auf die Red Bull Media Base zugeschnitten und sind somit nicht allgemein repräsentativ. Ihre Umsetzung ist deshalb nicht Teil dieser Arbeit und wurde bis Ende 2014 bereits zum Großteil durchgeführt. Sie fließen nicht in die Berechnung des Gesamtaufwands mit ein, werden aber erwähnt um einen Überblick zu geben.

3.2.2 Vorgehensweise

Die Fallstudie ist in zwei Teilbereiche unterteilt und untersucht einerseits den Aufbau der Basisinfrastruktur und andererseits die Implementierung der DevOps Konzepte für Applikationen. Für die Basisinfrastruktur wird untersucht, welche Systeme und Konzepte minimal notwendig sind, um Applikationen im Umfeld der Red Bull Media Base (siehe Abschnitt 3.2.1) nach DevOps betreiben zu können. Die in Abschnitt 4.3

genannten Applikationen beschreiben dabei die minimale Menge an Systemen die nötig sind, um für Applikationen der Red Bull Media Base die Umsetzung von DevOps zu ermöglichen. Mit Hilfe dieser Basisinfrastruktur kann ein verlässlicher Betrieb der Applikationen gewährleistet werden und sie unterstützt die vollständige Einführung der Automatisierung. Für jeden Bestandteil der Basisinfrastruktur wird untersucht, welche Probleme im klassischen Betrieb auftreten, wie diese Probleme mit neuen Konzepten gelöst werden können und wie der tatsächliche Aufbau aussieht. Abschließend wird eruiert, wie viel Aufwand der Aufbau der Basisinfrastruktur insgesamt benötigt.

Der zweite Teilbereich der Fallstudie befasst sich mit der Implementierung der Konzepte von DevOps für beispielhafte Applikationen und ihre Integration in die Basisinfrastruktur. In der Fallstudie werden dabei vier unterschiedliche Applikationen betrachtet. Alle Applikationen werden von der Red Bull Media Base entwickelt. Sie unterscheiden sich aber in Komplexität, Reifegrad und Anzahl der beteiligten Entwickler. Bei der jüngsten Applikation (APEX, Abschnitt 4.4.4), wurde die Entwicklung gerade abgeschlossen. Sie ist noch nicht produktiv in Betrieb. Die älteste Applikation (CPAS, Abschnitt 4.4.3), wird bereits seit knapp sieben Jahren eingesetzt. Die Größe des Entwicklungsteams der Applikationen reicht von einer Person, (Delivery Agent, Abschnitt 4.4.1) bis zu Teams mit ca. 10 Personen. Die Menge der Applikationen weist Verteilungseigenschaften mit einer Bandbreite von weltweit verteilt (CPAS, Abschnitt 4.4.3) bis zu zentralen Clustern (OIDC, Abschnitt 4.4.2) auf. Mit dieser Menge an Applikationen ist ein breites Spektrum abgedeckt und soll einen Überblick geben, wie die Implementierung für Applikationen mit unterschiedlichen Eigenschaften verläuft.

Ein besonderer Schwerpunkt vorliegender Arbeit ist die Ermittlung, ob der initiale Aufwand der Implementierung für diese Applikationen in etwa gleich ist, oder ob dieser mit Anzahl der Applikationen ab bzw. zunimmt. Außerdem sollen Kriterien ermittelt werden anhand derer man den initialen Aufwand abschätzen kann. Dafür werden die Metriken je Applikation erfasst und anschließend gegenübergestellt.

Damit die Entwicklung der Basisinfrastruktur begonnen werden konnte, mussten Vorarbeiten geleistet werden. Diese Vorarbeiten sind im Falle der Red Bull Media Base einerseits von einem größeren Team als nur drei Personen umgesetzt worden, andererseits wurden diese Arbeiten völlig separiert vom laufenden Betrieb durchgeführt. Aus diesem Grund werden diese Vorarbeiten nur kurz im Abschnitt 4.1 erläutert.

3.2.3 Ist-Zustand der Red Bull Media Base

Die Red Bull Media Base entwickelt fast ausschließlich Applikationen für das Red Bull Media House, die fast alle Bereiche von der Kontribution bis zur Distribution von

Multi-Media Dateien abdecken. Die Hauptapplikation ist eine zentrale Multi-Media-Asset-Management Applikation, in der viele Bereiche des Red Bull Media House abgebildet werden. Zusätzlich dazu gibt es ein globales Netzwerk von Standorten (siehe Abschnitt 3.2.1), um den Austausch von Multi-Media Dateien zwischen den Außenstellen zu ermöglichen. Alle Applikationen die aktuell in Betrieb sind, werden von einem Hosting Provider aus Deutschland betreut. Dies bedeutet, dass die Installation und das Überwachen der Applikationen von diesem Provider übernommen wird. Alle Tätigkeiten die auf einzelnen Maschinen durchgeführt werden, müssen mit dem Hosting Provider abgestimmt werden, damit dieser einen einwandfreien Betrieb gewährleisten kann.

Dies hat zur Folge, dass beim Deployment einer neuen Applikation, einer neuen Maschine oder einer neuen Version, neben den jeweiligen Entwicklern auch immer mindestens eine Person des Providers beschäftigt ist. Indem die Entwickler einiger Applikationen im Laufe der letzten Jahre die nötigen Rechte erhalten haben selbstständig Deployments durchzuführen, wurde dieser Prozess bereits etwas vereinfacht. Sind jedoch Tätigkeiten notwendig, die außerhalb dieses vordefinierten Bereiches liegen, ist wiederum eine Ressource des Providers notwendig um diese durchzuführen. Darüber hinaus wird ein Großteil der Tätigkeiten noch manuell durchgeführt und somit fehleranfällig ist und mehr Zeit in Anspruch nimmt. Ein weiteres Problem ist, dass das Know-How über den Betrieb der Applikationen fast ausschließlich beim Hosting Provider liegt und daher eine Abhängigkeit besteht.

Das beschriebene Vorgehen ist allerdings nicht flexibel genug um auf individuelle Anforderungen reagieren zu können. Um diesen Anforderungen entsprechen zu können und Abhängigkeiten aufzulösen, wird im Zuge vorliegender Arbeit DevOps in der Red Bull Media Base eingeführt.

3.2.4 Das Projektteam

Wie in den Forschungsfragen definiert, besteht das Projektteam aus drei Personen, um die Umsetzung der Konzepte von DevOps und Continuous Delivery anhand eines kleinen Teams zu demonstrieren. Die Aufgabe des Projektteams ist es die komplette Umstellung voran zu treiben und zu implementieren. Um die Rahmenbedingungen besser darzustellen, werden im Folgenden die Mitglieder des Projektteams mit ihren bisherigen Erfahrungen beschrieben. Die jeweilige Personenbeschreibung wurde von jedem Mitglied selbst verfasst. Sie entstammen somit nicht dem Autor der restlichen Arbeit.

Michael Haslauer (Autor): Geboren 1991, studiert derzeit an der Fachhochschule Salzburg, Studiengang Informationstechnik und Systemmanagement und untersucht im Zuge seiner Masterarbeit die Konzepte von DevOps. Er arbeitet seit April 2011 für das Red Bull Media House und ist beteiligt an der Entwicklung der hauseigenen Multi-Media-Asset-Management Plattform. Seit Mai 2014 beschäftigt er sich mit DevOps und Continuous Delivery und ist beteiligt an der firmeninternen Initiative, die gesamte Plattform nach den Konzepten von DevOps zu betreiben. Bisher sind keine persönlichen Erfahrungen mit dem Betrieb von Software vorhanden.

Alexander Dobriakov: *Alexander is a seasoned Software Delivery and Architecture Expert, who conceptualized, designed, implemented, and optimised Enterprise level software systems. With over 25 years experience in Information Technology for eCommerce, Analytical CRM, Logistics, Direct Marketing, Health Care, Media and Publishing industries, he has a wide range of hands-on expertise through the whole Software Lifecycle in Germany, Netherlands and Russia. Alexander is Oracle Certified Professional, iSAQB certified Software Architect, and hold a M.S. Degree in Mathematics and Aerospace engineering from South Ural State University.¹*

Michael Kutzner: *In 1994, during his computer science studies at the University of Paderborn, he founded his first company paderLinux, a software development company and early adaptor in the internet business. In his role as CEO he lead the company during 7 years. 2001 Michael sold his company to mediaWays, a joint venture of Bertelsmann and Debis Systemhaus. 2002 mediaWays was acquired by the Spanish telecommunications company Telefónica. He was appointed Director Systems&Applications being responsible of the hosting and service providing business line. Within three years he consolidated and developed Systems&Applications to become a leading and profitable service provider in Germany, providing hosting and streaming services to top news magazines and TV stations. In 2004 he joined arvato mobile, a mobile unit of Bertelsmann where he took over the management of operations in the newly established GNAB business line. In 2006 he was appointed Vice President Operations of arvato mobile global being responsible for the operational departments, serving infrastructure and applications for the mobile and internet music and video services.²*

¹Verfasst von Alexander Dobriakov

²Verfasst von Michael Kutzner

Kapitel 4

Implementierung

In den folgenden Abschnitten wird die konkrete Durchführung der in Kapitel 3 definierten Fallstudie beschrieben. Der erste Teil des Kapitels beschäftigt sich mit den in Abschnitt 3.2.1 erwähnten Voraussetzungen. Anschließend wird die Entwicklung der notwendigen Konzepte sowie der Aufbau der unterstützenden Systeme und Basisinfrastruktur erläutert. Zuletzt folgt die Implementierung der Konzepte von DevOps für ausgewählte Applikationen.

4.1 Voraussetzungen

Um die Konzepte von DevOps und Continuous Delivery umsetzen zu können, sind nicht nur Änderungen am Deployment der Software notwendig. Es müssen viele weitere Bereiche betrachtet werden, in denen Anpassungen notwendig sind. Andernfalls kann nicht das volle Potential von DevOps und Continuous Delivery genutzt werden und es ergeben sich nur begrenzt Vorteile. Im nachfolgenden Abschnitt werden einige dieser Bereiche, wie zum Beispiel der logische Aufbau des Rechenzentrums (Abschnitt 4.1.1) oder das VM Management (Abschnitt 4.1.2), genauer betrachtet. Es wird beschrieben, welche Probleme in den jeweiligen Bereichen vorhanden waren und wie die Lösung für diese aussieht. Die Arbeiten in den folgenden Bereichen wurden bereits vor Beginn der Fallstudie fertiggestellt. Sie sind speziell auf die Infrastruktur der Red Bull Media Base zugeschnitten, weswegen sie nicht allgemein repräsentativ sind. Außerdem wurde ihre Umsetzung zum Großteil vom Hosting Provider der Red Bull Media Base durchgeführt. Aus diesen Gründen fließen sie nicht in die Berechnung des Gesamtaufwandes mit ein. Der Implementationsaufwand der einzelnen Bereiche wird aber zur Übersicht angeführt.

4.1.1 Logischer Aufbau des Rechenzentrums

Der logische Aufbau des Rechenzentrums betrifft die grundlegende Aufteilung der Netzwerkbereiche mittels Subnetze und das Netzwerkmanagement zwischen diesen Bereichen. Der bisherige logische Aufbau des Rechenzentrums, sowie die dazugehörige Infrastruktur, werden im folgenden als *DC1.0* bezeichnet. Das neue Setup trägt den Namen *DC2.0*. Die Abkürzung *DC* steht hierbei für *Datacenter*.

Problemstellung: Wie im Ist-Zustand (siehe 3.2.3) beschrieben, ist für die Verwaltung des DC1.0 ein externer Hosting Provider verantwortlich. Um hohe Sicherheit zu garantieren, werden nur die nötigsten Netzwerkverbindungen zwischen einzelnen Maschinen ermöglicht. Werden zusätzliche Verbindungen benötigt, so muss dies beim Provider angefragt werden, der die Änderung anschließend durchführt. Dieser Vorgang dauert in den meisten Fällen einen Arbeitstag und bietet somit keine ausreichend hohe Flexibilität, um schnell neue Maschinen ins Rechenzentrum integrieren zu können. Das Ziel ist es, diesen Prozess soweit wie möglich zu automatisieren. Die aktuell eingesetzte Infrastruktur bietet allerdings nicht die Möglichkeit, sie in die Automatisierungsprozesse einzubinden. Aus diesem Grund muss ein generisches Netzwerkschema mit vorkonfigurierten Verbindungen definiert werden, das die bekannten Anforderungen erfüllt. Dies bietet einen Kompromiss zwischen möglichst hoher Flexibilität und Sicherheit. Als Ergebnis sollen möglichst wenig Änderungen an den Netzwerkverbindungen nötig sein.

Lösung: Alle aktuell eingesetzten Applikationen wurden, anhand ihrer Anforderung an die Sicherheit, in logische Gruppen unterteilt. Alle Applikationen einer Gruppe benötigen dieselben Netzwerkverbindungen. Somit sind die Netzwerkregeln nur mehr pro Gruppe und nicht mehr pro Applikation nötig. Das reduziert die Menge der Netzwerkregeln drastisch. Für jede Gruppe wurde daher ein eigenes Subnetz, mit fixem IP-Adressbereich, angelegt. Die Zugehörigkeit einer Maschine zu einem Subnetz ergibt sich somit anhand ihrer IP-Adresse, was wiederum implizit die möglichen Netzwerkverbindungen in andere Subnetze ergibt. Bei der Erstellung der Maschine muss nur darauf geachtet werden, welche IP-Adresse vergeben wird, um sie dem korrekten Subnetz zuzuweisen. Eine Maschine kann auch zu mehr als einem Subnetz gehören. In diesem Fall muss sie mehrere Netzwerkschnittstellen aufweisen und jeweils eine IP-Adresse aus dem jeweiligen Subnetz zugewiesen bekommen. Folgende Subnetze ergeben sich aus den Applikationen im DC1.0, die für das DC2.0 implementiert wurden:

- **DMZ:** Demilitarisierte Zone für öffentliche, nicht kritische Applikationen, wie beispielsweise NTP und DNS.

- **ADMIN:** Administratives Subnetz. Enthält alle Applikationen die nötig sind, um das DC2.0 zu verwalten. Dies beinhaltet beispielsweise den Build und Deployment Server (siehe 4.2.2).
- **MONITORING:** Enthält alle Applikationen die nötig sind, für Monitoring und Logging aller Maschinen und Applikationen im DC2.0. Diese Applikationen gehören zur Basisinfrastruktur und sind im Abschnitt 4.3 beschrieben.
- **PUBLIC:** Einziges, öffentliches Subnetz, das aus dem Internet erreichbar ist. Öffentliche Applikationen müssen diesem Subnetz angehören, um aus dem Internet erreichbar zu sein.
- **<ENV>_AUTH:** Speziell geschütztes Subnetz, das Applikationen für die Authentifizierung und Autorisierung von Benutzern bereitstellt. Da diese Applikationen sensible Benutzerdaten enthalten, müssen sie auf Grund des Datenschutzes speziell geschützt sein.
- **<ENV>_BACKNET:** Subnetz für die Kommunikation zwischen Applikationen über einen zentralen Service, wie beispielsweise eine Messaging Queue.
- **<ENV>_SERVICE:** Standard Subnetz für Applikationen

Alle Subnetze die mit <ENV> beginnen, existieren in mehrfacher Ausführung, um die einzelnen Umgebungen zu trennen. Im Umfeld der Red Bull Media Base werden grundsätzlich vier Umgebungen betrieben. Diese sind *Produktion*, *Pre-Produktion*, *Staging* und *Test*. Beim Namen des jeweiligen Subnetzes wird dabei das Prefix <ENV> durch ein Namenskürzel der Umgebung ersetzt. Alle anderen Subnetze werden von den einzelnen Umgebungen geteilt.

Folgende Netzwerkregeln wurden zwischen den oben genannten Netzwerksegmenten definiert. Diese gelten für alle Maschinen, die sich in dem jeweiligen Subnetz befinden. Sie decken die aktuellen Anforderungen aller Applikationen ab. Ausgehende Verbindungen werden grundsätzlich erlaubt (auch ins Internet). Tabelle 4.1 beschreibt daher nur die erlaubten, eingehenden Verbindungen in das jeweilige Subnetz. Die Bezeichnung *RFC1918* umfasst dabei alle internen Subnetze, nicht aber das Public Subnetz. Innerhalb eines Netzwerksegments ist die Kommunikation zwischen den Maschinen uneingeschränkt erlaubt.

Mit diesem Aufbau ist eine ausreichend hohe Sicherheit und Flexibilität gegeben. Es sind nur mehr Änderungen notwendig, wenn komplett neue Applikationen in das Rechenzentrum aufgenommen werden. In diesem Fall kann die Zeit für manuelles konfigurieren der Netzwerkregeln mit einkalkuliert werden.

Netzname	Port	Erreichbar von
DMZ	Alle	RFC1918
ADMIN	Alle	MONITORING
MONITORING	Alle	RFC1918
PUBLIC	80, 443 HTTP(S)	Internet
	Alle	RFC1918
<ENV>_AUTH	636 LDAPS	RFC1918
	Alle	MONITORING
	Alle	ADMIN
<ENV>_SERVICE	Alle	MONITORING
	Alle	ADMIN
<ENV>_BACKNET	3306 MySQL, 1521 Oracle	RFC1918
	Alle	MONITORING
	Alle	ADMIN

Tabelle 4.1: Die Tabelle zeigt die generellen Netzwerkregeln des DC2.0. Sie beschreiben die möglichen Verbindungen zwischen den einzelnen Netzwerksegmenten.

4.1.2 Virtual Maschine Management

Im DC2.0 der Red Bull Media Base ist die gesamte Infrastruktur virtualisiert. Es gibt somit keine Maschine die direkt auf reeller Hardware betrieben wird. Als Hypervisoren werden aktuell VMware und KVM eingesetzt.

Problemstellung: Im DC1.0 wurden die virtuellen Maschinen ausschließlich vom Hosting Provider verwaltet. Daher musste jede neue Maschine explizit angefragt werden. Die neue Maschine wurde anschließend manuell erstellt und diese dem jeweiligen Antragsteller bereitgestellt. Dieser Vorgang benötigte oft einige Arbeitstage und ist daher zu langsam und zeitintensiv, im Hinblick auf die Konzepte von DevOps. Ziel ist es, eine Möglichkeit zu finden, die Erstellung von virtuellen Maschinen in die Automatisierungsprozesse integrieren zu können. Damit werden neue Maschinen automatisiert erstellt und sind anschließend das Ziel für ein Deployment einer Applikation.

Lösung: Für das DC2.0 wird eine private Cloud Infrastruktur aufgebaut, die aus einem Hardware Cluster mit mehreren Hosts besteht. Auf jedem dieser Hosts läuft VMware oder KVM als Hypervisor. Der gesamte Cluster wird von einer Cloud Mana-

gement Plattform verwaltet, um dort virtuelle Maschinen zu betreiben. Für diese Aufgabe wird OpenNebula¹ eingesetzt. Wie in Abbildung 4.1 dargestellt, kann OpenNebula mit unterschiedlichen Hypervisoren arbeiten und abstrahiert ebenfalls das Ressourcen-Management über den gesamten Hardware Cluster. Ein enormer Vorteil von OpenNebula ist, dass die Funktionen über eine API exponiert sind. Das bietet die Möglichkeit, die Verwaltung von virtuellen Maschinen in die Automatisierungsprozesse einzubinden.

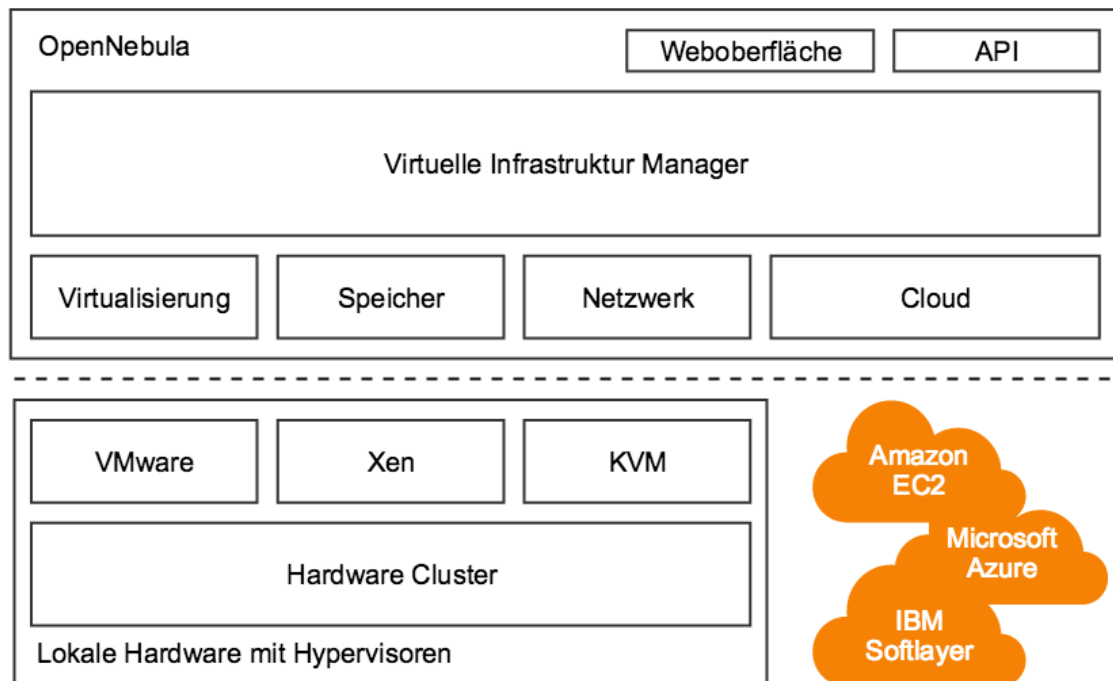


Abbildung 4.1: Grundlegende Architektur von OpenNebula. Die strichlierte Linie zeigt die Trennung zwischen OpenNebula und der verwalteten Hardware. Apatiert aus [13]

Ein weiterer Vorteil von OpenNebula ist, dass neben lokaler Hardware auch gängige Cloud Services (siehe Abbildung 4.1) wie Amazon EC2, Microsoft Azure oder IBM Softlayer als Infrastruktur für virtuelle Maschinen genutzt werden können. Somit können mehrere Infrastrukturen über dasselbe System verwaltet werden. Weiters ist es möglich Rechte für die Verwaltung einzelner virtueller Maschinen an Benutzer zu koppeln. Somit können die einzelnen Umgebungen auch mit unterschiedlichen Rechten getrennt werden, um mögliche Fehler durch Verwechslungen zu verhindern [13].

4.1.3 Agiles Entwicklungsmodell

DevOps und Continuous Delivery versuchen die Konzepte und Vorteile von agilen Entwicklungsmethoden auf den Software-Betrieb anzuwenden. Aus diesem Grund können

¹<http://opennebula.org>, Cloud Management Plattform für hybride Infrastrukturen

die Vorteile von DevOps ebenfalls nur dann entsprechend zur Geltung kommen, wenn auch die Entwicklung bereits nach agilen Methoden vorgeht. Vor allem, wenn zusätzlich zu DevOps noch Continuous Delivery erreicht werden soll, sind agile Entwicklungsmodelle unumgänglich. Ansonsten kann im Betrieb schnell auf Anforderungen reagiert werden, aber die Entwicklung verzögert den Vorgang. Es müssen also die Software-Entwicklung **und** der Betrieb nach agilen Vorgehensmodellen arbeiten. Die Red Bull Media Base entwickelt seit Beginn 2013 jede Applikation nach dem Kanban Prinzip. Die meisten Projekte veröffentlichen neue Versionen in einem regelmäßigen Abstand von zwei Wochen [6].

4.2 Ausgewählte Systeme und Konzepte

Wie im theoretischen Teil erwähnt, behandelt der technische Aspekt von DevOps die Automatisierung von Aufgaben. Häufig auftretende Tätigkeiten wie Software bauen, testen und deployen sollen nicht mehr manuell durchgeführt werden. Für die Automatisierung dieser Tätigkeiten werden entsprechende Systeme benötigt. Eine große Anzahl an Beispielen für Systeme unterschiedlicher Aufgaben sind in [20] gelistet. In den nachfolgenden Abschnitten werden die vom Projektteam gewählten Systeme, sowie die entwickelten Konzepte, beschrieben. Je Abschnitt wird das vorhandene Problem erläutert und wie dieses mit dem gewählten System/Konzept gelöst wird.

4.2.1 Konfigurationsmanagement mit Ansible

Das Konfigurationsmanagement stellt einen zentralen Bestandteil von DevOps dar. Damit wird die Infrastruktur nachvollziehbar und reproduzierbar definiert. Es unterstützt die Automatisierung und eliminiert folglich manuelle Tätigkeiten. Bisher wurde im Umfeld der Red Bull Media Base kein Konfigurationsmanagementsystem genutzt.

Für das Konfigurationsmanagement und die Orchestrierung der Infrastruktur gibt es unzählige Systeme auf dem Markt [20]. Viele davon, wie etwa Puppet² oder Chef³, werden laut eigenen Angaben von vielen namhaften Unternehmen verwendet. Eine Evaluierung dieser Systeme ergab aber einen entscheidenden Nachteil für die Red Bull Media Base. Puppet sowie Chef verfolgen eine Client-Server Architektur der Infrastruktur. Ein zentraler Server verwaltet die Konfiguration der Clients und sendet diese regelmäßig an die Clients. Dafür muss aber auf jeder Maschine der entsprechende Client installiert und konfiguriert sein [15]. Bevor man mit dem Deployment der Software

²<https://puppetlabs.com/about/customers>, u.a. Cisco, Intel, NASA und Spotify

³<https://www.chef.io/customers/>, u.a. Etsy, Rackspace und Facebook

beginnen kann, müssen somit noch zusätzliche Tätigkeiten, nach der Erstellung der Maschine, durchgeführt werden. Daher wurde nach einer clientlosen Alternative gesucht. Ansible stellte sich als eine Alternative heraus.

Kriterien für die Wahl von Ansible: Ansible ist ein Konfigurationsmanagement System auf Basis von Python, das keine Clients auf den Maschinen benötigt. Voraussetzungen für Ansible sind lediglich eine SSH Verbindung zur Maschine und eine Python Umgebung mit minimaler Version 2.6. Da Python grundsätzlich Bestandteil von Linux Distributionen ist, sind diese Voraussetzungen gegeben [1, 7].

Die Vorteile der clientlosen Architektur von Ansible sind laut [1] folgende und haben maßgeblich zu dessen Auswahl beigetragen:

- Zwischen der Maschine die Ansible ausführt und der Zielmaschine werden nur Python Skripte und deren Ergebnisse ausgetauscht. Es erfolgt keine weitere Kommunikation und beschränkt den Netzwerkverkehr auf ein Minimum.
- Ansible bietet hohe Sicherheit, da die ausgereifte Implementierung von OpenSSH genutzt und keine eigene Verschlüsselung und Transportsicherung entwickelt wird.
- Da es keinen zentralen Konfigurationsserver gibt, können Clients auch keine kritischen Daten/Konfigurationen (z.B. Passwort-Dateien) anderer Maschinen abfragen. Ansible transferiert nur die unbedingt benötigten Daten auf eine Maschine. Eine kompromittierte Maschine kann daher keine anderen Daten abfragen.
- Die Verwaltung des Konfigurationsservers entfällt und es muss nicht darauf geachtet werden, dass die Version von Server und Client zusammenpassen. Updates von Ansible müssen lediglich auf der ausführenden Maschine abgewickelt werden.
- Es gibt keine Skalierungsprobleme bei Ansible, da die Clients nicht ständig von einer zentralen Instanz überprüft werden. Die tatsächlichen Aufgaben werden direkt von der Zielmaschine ausgeführt und nicht von einem zentralen Server. Damit verteilt sich die komplette Last über alle Maschinen.
- Da kein Client benötigt wird, verbraucht dieser auch keine Ressourcen auf der Zielmaschine. Es wird nur Last erzeugt, wenn konkrete Aufgaben durchgeführt werden und nicht durch regelmäßige Checks.
- Die Deklarationen werden im YAML Format verfasst und daher für Entwickler leicht lesbar. Da auch immer nur der gewünschte Endzustand beschrieben wird, und nicht die nötigen Schritte um dorthin zu gelangen, kann schnell erfasst werden, welche Konfiguration angewandt wird.

Ein Nachteil von Ansible, gegenüber Systemen mit zentralem Server ist, dass Konfigurationen nicht ständig erzwungen werden. Bei einem zentralen System werden manuell durchgeführte Änderungen auf einer Maschine beim nächsten Check wieder entfernt. Die lokale Konfiguration wird ständig mit der zentralen Konfiguration abgeglichen. Bei Ansible werden manuelle Änderungen nur bei erneuter Provisionierung der Maschine entfernt. Ein weiterer Nachteil ist, dass Ansible aktuell nur sehr wenig Unterstützung für Windows bietet. Da die Red Bull Media Base aber aktuell keine Windows Server einsetzt, ist dieser Nachteil nicht relevant. Bei Ansible überwiegen im konkreten Bedarf die Vorteile gegenüber den Nachteilen, weswegen sich das Projektteam für den Einsatz von Ansible entschieden hat [7].

Grundsätzliche Funktionsweise von Ansible: Ähnlich wie bei Chef/Puppet wird bei Ansible der gewünschte Endzustand einer Maschine beschrieben. Dies geschieht in der Sprache YAML⁴. Die Summe aller Deklarationen wird Playbook genannt. Führt man ein Playbook aus, so werden alle nötigen Schritte unternommen, um den darin definierten Zustand zu erreichen. Auflistung 4.1 zeigt als Beispiel das Playbook der Delivery Agent Applikation aus Abschnitt 4.4.1.

```
1 - hosts: da-application
2   sudo: yes
3   vars:
4     umgebung: "{{lookup('env','ONE_INVENTORY_ENV')}}"
5     sensu_server_rabbitmq_hostname: sensu.rbmbtnx.net
6     logstash_server: logstash.rbmbtnx.net
7   pre_tasks:
8     - name: load variables for the particular environment
9       include_vars: "{{item}}"
10      with_items:
11        - "../aux_vars/all/main.yml"
12        - "../aux_vars/da/{{umgebung}}.yml"
13   roles:
14     - {role: rbmh.common, tags: 'common'}
15     - {role: rbmh.delivery-agent.ha, tags: 'deployment'}
16     - {role: rbmh.logstash-shipper}
17     - {role: rbmh.sensu-client}
18   tasks:
19     - name: register public service url with dns
20       uri: url="https://{{ip_dns_vip}}/api/add_zone.php?name={{
21         public_service_uri}}&ext={{keepalived_shared_ip}}&int={{
22         keepalived_shared_ip}}&typ=nosite&id=8dfk38sdERe8sdA923n"
23       run_once: true
```

Auflistung 4.1: Beispiel: Ansible Playbook des Delivery Agents

⁴<http://www.yaml.org/spec/1.2/spec.html>, YAML Ain't Markup Language

Für eine bessere Gliederung der einzelnen Zustände von Komponenten, können diese in Pakete namens Roles unterteilt werden. Somit kann ein Playbook als eine Menge von Roles angesehen werden, die wiederum aus einer Vielzahl an Tasks bestehen. In diesem Fall werden Roles für die Installation der Delivery Agent Applikation, des Sensus Clients (Abschnitt 4.3.5) und der Log-Lieferung (Abschnitt 4.3.4) durchgeführt. Wie Auflistung 4.1 zu entnehmen, können auch noch zusätzliche Tasks angegeben werden.

Aus den Deklarationen generiert Ansible Python Skripte (sogenannte Module), die auf die Maschinen transferiert und dort ausgeführt werden. Module erkennen automatisch, welche Schritte nötig sind, um den gewünschten Zustand zu erreichen. Ist dieser Zustand bereits vorhanden, so werden keine Schritte vollzogen. Egal wie oft man eine Aufgabe durchführt, der Endzustand ist immer der Gleiche. Dieses Verhalten nennt man *Idempotent* [7].

Die zweite Komponente zu den Playbooks ist das sogenannte Inventory. Playbooks geben an, welcher Zustand erreicht werden soll und das Inventory, auf welchen Maschinen dieser Zustand gewünscht ist. Es beinhaltet die Liste aller Maschinen, aufgeteilt in Gruppen [7]. Im Playbook wird an der Stelle *hosts: da-application* angegeben, dass es nur für die Maschinen der Gruppe *da-application* gültig ist. Ein Inventory kann in zwei Formaten angegeben werden. Entweder im INI-Format oder als JSON. Auflistung 4.2 zeigt beide Möglichkeiten für drei Maschinen des Delivery Agents.

```
1 "da-application": {
2     "hosts": [
3         "172.20.35.84",
4         "172.20.35.85",
5         "172.20.35.86",
6     ]
7 }
8 -----
9 [da-application]
10 172.20.35.84
11 172.20.35.85
12 172.20.35.86
```

Auflistung 4.2: Beispiele: Ansible Inventory des Delivery Agent

Einsatz bei der Red Bull Media Base: Quellcode sollte möglichst wiederverwendbar geschrieben werden, um zukünftige Aufwände zu reduzieren. Daher wird der Aufgabenbereich einer Ansible Role so aufgeteilt, dass sie jeweils für die Installation einer Komponente zuständig ist. So ergeben sich beispielsweise Roles für die Installation von Java, oder des Tomcat Webserver. Dadurch können die Roles bei jedem Service, das eine entsprechende Komponente benötigt, wiederverwendet werden. Dieser Ansatz

ist von entscheidender Bedeutung und reduziert den nötigen Aufwand der Implementierung drastisch. Ein weiterer Vorteil ist, dass die Komponente auf allen Maschinen gleich installiert ist. Unterschiedliche Arten von Installationen sind nicht möglich und daher wird der Verwaltungsaufwand reduziert. Die spezielle Konfiguration der Komponenten erfolgt danach in einer Role für die jeweilige Applikation. Sind die Aufgabenbereiche der einzelnen Roles voneinander abgegrenzt, so besteht ein Playbook grundsätzlich nur mehr aus der Kombination der Roles für die Installation der benötigten Komponenten, plus eine Role für die Applikation selbst, wie in Auflistung 4.1 zu sehen ist.

Außerdem sind alle umgebungsspezifischen Konfigurationen als Variablen ausgegliedert. Somit wird dieselbe Role für die Installation in allen vier Umgebungen verwendet. Die spezifischen Variablen für die jeweilige Umgebung werden zu Beginn des Playbook, wie in Auflistung 4.1 in Zeile 10, geladen. Dies gilt ebenso für die lokale Entwicklungsumgebung. In den folgenden Abschnitten wird bei den Metriken immer auf die Anzahl der Roles verwiesen, die wiederverwendet wurden. Dieser Aufwand entfällt somit für die jeweilige Applikation.

Durch die Möglichkeit, das Inventory in JSON angeben zu können, ist es möglich, Ansible mit OpenNebula zu kombinieren. Über die API von OpenNebula kann die Liste der aktuell verfügbaren Maschinen abgerufen werden. Diese Liste wird in ein JSON Format umgewandelt und anhand von Metadaten an den Maschinen die Gruppen erstellt. Somit ist immer ein aktuelles Inventory für Ansible verfügbar und muss nicht manuelle gepflegt werden.

Erfassen der Metriken: Da Ansible für die Provisionierung zuständig ist, können die Metriken der Durchlaufzeit für Ansible nicht erfasst werden. In Tabelle 4.2 sind daher nur die Aufwände für die Konzeption der Struktur der Roles, das Einlernen des Projektteams und die Installation von Ansible auf einer Maschine aufgeführt.

Metrik		Menge
Initialer Aufwand	Konzeption	24 PS
	Einlernen	100 PS
	Installation	0.5 PS

Tabelle 4.2: Erfasste Metriken für Ansible

Die Konzeption der Struktur der einzelnen Elemente von Ansible und ein Machbarkeitstest beanspruchte etwa drei Arbeitstage. Dies wurde von Alexander Dobriakov durchgeführt, da er bereits Erfahrung mit Ansible aufweisen konnte. Die restlichen Projektmitglieder mussten sich erst mit Ansible vertraut machen. Das Einlernen in

Ansible erstreckte sich auf mehr als eine Arbeitswoche pro Person. Anschließend war genug Wissen vorhanden, um die meisten Aufgaben durchzuführen. Die Installation von Ansible war in wenigen Minuten abgeschlossen, da es via pip⁵ verfügbar ist.

4.2.2 Build und Deployment Pipeline mit Bamboo

In vielen Software Unternehmen wird bereits Continuous Integration (CI) betrieben, um möglichst schnell Änderungen in die Software zu integrieren. Somit bekommen die Entwickler schneller Feedback, ob die Software nach den Änderungen noch gebaut werden kann. Diese Aufgabe wird meist von einem sogenannten Build-Server durchgeführt [23]. Die Red Bull Media Base verwendet als Build und Deployment Server *Bamboo*⁶. In der Vergangenheit wurde Jenkins⁷ verwendet, allerdings bietet dieser nur über Community Plugins die Möglichkeit, Deployments zu verwalten. Die Integration der Plugins funktioniert allerdings nicht immer einwandfrei. Daher wurde entschieden, bei diesem zentralen Bestandteil auf kommerzielle Software zu setzen.

Einsatz bei der Red Bull Media Base: Bamboo stellt den zentralen Bestandteil der gesamten DevOps Infrastruktur dar. Es ist bei allen Projekten für den Build der Software Artefakte und die Verwaltung der Deployments zuständig. Diese Maschine muss somit Zugriff auf alle anderen Maschinen im DC2.0 haben. Daher ist Bamboo eine der wenigen Maschinen die im ADMIN Subnetz des DC2.0 angesiedelt ist.

Für die Verwaltung der Builds und Deployments gibt es in Bamboo zwei Konzepte namens *Build- und Deploymentpläne*. Für jedes Software-Projekt wird ein Buildplan konfiguriert. Dieser führt automatisch die nötigen Schritte aus und erzeugt Artefakte für die neueste Applikationsversion. Anschließend werden Unit-Tests ausgeführt und ausgewertet. Der Build ist nur dann erfolgreich abgeschlossen, wenn diese keinen Fehler melden. Die erzeugten Artefakte werden anschließend in einem Artefakt Repository (siehe 4.2.6) gespeichert. Da der Quellcode der Projekte mit Git verwaltet wird, überwacht Bamboo die Repositories auf neue Commits. Werden Änderungen eingecheckt so startet Bamboo automatisch den Build um Continuous Integration zu betreiben.

Mittels Deploymentplänen werden die Deployments auf die einzelnen Maschinen gesteuert und verwaltet. In Bamboo ist es möglich, pro Projekt mehrere Umgebungen zu verwalten und demnach die im Media Base Umfeld existierenden vier Umgebungen abzubilden. Für die Provisionierung der Software, werden bei jedem Deploymentplan

⁵http://docs.ansible.com/intro_installation.html#latest-releases-via-pip, Installationsanleitung der offiziellen Dokumentation

⁶<https://de.atlassian.com/software/bamboo>, kommerzielles Produkt von Atlassian

⁷<https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>, OpenSource Build Server

entsprechende Ansible Playbooks hinterlegt. Wird nun ein Deployment durchgeführt, so startet Bamboo das hinterlegte Playbook und markiert die Umgebung als OK, sofern dieses keinen Fehler liefert. Es ist außerdem möglich, Deploymentpläne an die dazugehörigen Buildpläne zu koppeln. Sobald ein erfolgreicher Build stattgefunden hat, wird der Deploymentplan automatisch ausgeführt und somit Continuous Delivery erreicht. Dies wird aber aktuell noch bei keinem Projekt verwendet. Die Deploymentpläne werden alle manuell ausgeführt.

Versionsverwaltung der Ansible Playbooks: Die genauen Schritte eines Deployments sind in den Playbooks definiert. Um ein Deployment jederzeit nachvollziehen zu können, muss der aktuelle Stand der Playbooks, zum Zeitpunkt des Deployments, erfasst werden. Ansonsten ist es möglich, unterschiedliche Versionen der Playbooks für dasselbe Deployment in unterschiedlichen Umgebungen zu verwenden. Dies kann zu schwer nachvollziehbaren Fehlern führen. Um die Version der Playbooks festzuhalten, wurde ein eigener Buildplan eingerichtet. Dieser testet die aktuellsten Playbooks auf syntaktische Korrektheit und speichert diese Version als eigenes Artefakt mit laufender Nummer ab. Im Deploymentplan eines Projekts kann gewählt werden, welche Version für das Deployment verwendet wird.


Deployment - Delivery Agent in DC2.0	Test	release-Ansible CI-325	 Logs	10 Apr 2015 05:44 PM
	Stage	release-Ansible CI-264	 Logs	17 Mar 2015 01:07 PM
	Preprod	release-Ansible CI-264	 Logs	17 Mar 2015 12:49 PM
	Prod	release-Ansible CI-264	 Logs	17 Mar 2015 02:03 PM

Abbildung 4.2: Der Deploymentplan des Delivery Agent. Die verwendete Version der Playbooks ist direkt an der Umgebung ersichtlich. Somit wird ein hohes Maß an Nachvollziehbarkeit erreicht.

Abbildung 4.2 zeigt den Deploymentplan des Delivery Agent. Die Version der Playbooks, die verwendet wurde um die Applikation in einer bestimmten Umgebung zu deployen, ist sofort ersichtlich. In diesem Beispiel erkennt man, dass Staging, PreProduktion und Produktion aktuell denselben Stand aufweisen, *CI-264*. In der Testumgebung wurde bereits eine neuere Version deployed, *CI-325*. Zusätzlich wird angezeigt, ob das letzte Deployment korrekt verlaufen ist und wann es durchgeführt wurde.

Erfassen der Metriken: Tabelle 4.3 listet die Metriken für die Aufwände der Installation von Bamboo, sowie die Durchlaufzeit für ein Update auf. Da die Installation von Bamboo und aller benötigten Komponenten mit Ansible durchgeführt wird, ist der initiale Aufwand höher als bei einer manuellen Installation. Dadurch können Updates von Bamboo über Ansible durchgeführt werden, das wiederum weniger Zeit erfordert,

Metrik		Menge
Initialer Aufwand	Installation von Bamboo	40 PS
	Basiskonfiguration	4 PS
	Konfiguration je Projekt	1 PS
Durchlaufzeit	Update von Bamboo	ca. 15 Minuten

Tabelle 4.3: Erfasste Metriken für Bamboo

gegenüber manuell durchgeführten Updates. Die Basiskonfiguration von Bamboo beinhaltet allgemeinen Dinge, die nur einmal für alle Projekte notwendig sind. Je Software-Projekt muss wie erwähnt ein Build- und Deploymentplan erstellt werden. Dies dauert jeweils etwa eine Personenstunde.

4.2.3 Lokale Entwicklungsumgebung mit Vagrant

Beim Betrieb einer Applikation wird meist nur auf die Umgebungen im Rechenzentrum geachtet, nicht aber auf die lokale Entwicklung. So wird lokal unter anderen Voraussetzungen entwickelt, als die Applikation später im Rechenzentrum läuft. Dies macht das lokale Testen ineffizient und führt zu unentdeckten Fehlern. Um dem entgegen zu wirken, wird eine Kombination aus Vagrant⁸ und Ansible eingesetzt. Mit Vagrant lassen sich virtuelle Umgebungen in textueller Form definieren, was diese leicht reproduzierbar und portierbar macht. So ist sichergestellt, dass die Entwickler immer dieselbe virtuelle Infrastruktur für die lokale Entwicklung nutzen. Bei der Erstellung der virtuellen Umgebung können außerdem Provisionierungsprozesse mit Ansible angestoßen werden, um die benötigte Software in der virtuellen Umgebung zu installieren. Da hier dieselben Ansible Playbooks wie in Produktion verwendet werden, ist diese Umgebung sehr nahe an der produktiven Umgebung. Ganze Cluster können ohne erheblichen Aufwand aufgebaut werden, um solche Szenarien in der Entwicklung zu simulieren. Obwohl sich nicht alle Unterschiede beseitigen lassen, steigt dennoch die Effizienz bei der Entwicklung erheblich.

```

1 Vagrant.configure(2) do |config|
2   config.vm.box = "centos-6.6-x86_64"
3   config.vm.box_url = "https://repo.rbmb.com/vagrant/centos66.json"
4
5   config.vm.define :web1 do |web1_config|
6     web1_config.vm.hostname = "web1.da.vagrant"
7     web1_config.vm.provider "virtualbox" do |vb|

```

⁸<http://docs.vagrantup.com/v2/why-vagrant/>, leichtgewichtige Virtualisierungsumgebung

```

8      vb.customize ["modifyvm", :id, "--name", "web1.da.vagrant", "--
      memory", "512", "--cpus", "1", "--ioapic", "on"]
9  end
10 web1_config.vm.network "private_network", ip: "192.168.33.101"
11 web1_config.vm.provision "ansible" do |ansible|
12     ansible.playbook = "test.yml"
13     ansible.inventory_path = "vagrant_servers"
14     ansible.limit = "da-application"
15 end
16 end
17
18 # VM Konfiguration der Loadbalancer Node
19 config.vm.define :lb1 do |lb_config|
20     lb_config.vm.hostname = 'lb1.da.vagrant'
21     lb_config.vm.provider "virtualbox" do |vb|
22         vb.customize ["modifyvm", :id, "--name", "lb1.da.vagrant", "--
        memory", "256", "--cpus", "1", "--ioapic", "on"]
23     end
24     lb_config.vm.network "private_network", ip: "192.168.33.103"
25     lb_config.vm.provision "ansible" do |ansible|
26         ansible.playbook = "lb.yml"
27         ansible.inventory_path = "vagrant_servers"
28         ansible.limit = "lbh-service"
29     end
30 end
31 end

```

Auflistung 4.3: Beispielhaftes Vagrantfile des Delivery Agent

Die virtuelle Umgebung wird in einem sogenannten *Vagrantfile* definiert. Auflistung 4.3 zeigt das Vagrantfile des Delivery Agent. Zuerst werden allgemeine Definitionen der Umgebung angegeben, wie beispielsweise das Betriebssystem für die virtuellen Maschinen (Zeilen 2 und 3). Anschließend wird eine Node für den Delivery Agent, sowie eine Loadbalancer Node definiert. Je Node wird der Hostname, RAM, CPU und die IP-Adresse angegeben. Für jede Node ist angegeben, welches Playbook und Inventory für Ansible verwendet werden soll (Zeilen 11-14 bzw. 25-28). Die Provisionierung wird durchgeführt, sobald die virtuellen Maschinen verfügbar sind.

Metrik		Menge
Initialer Aufwand	Installation von Vagrant	0.5 PS

Tabelle 4.4: Erfasste Metriken für Vagrant

Vagrant bietet jedoch keine Virtualisierung und benötigt eine bereits vorhandene Virtualisierungsumgebung, wie etwa Virtualbox. Dies wird als *Provider* bezeichnet und

ebenfalls im Vagrantfile angegeben (Zeile 7 bzw. 21). Vagrant ist für alle Betriebssysteme verfügbar und lässt sich innerhalb einer halben Stunde installieren (Tabelle 4.4). Eine spezielle Konfiguration ist nicht erforderlich. Die Installation muss allerdings auf jedem Entwicklungsrechner durchgeführt werden.

4.2.4 Lifecycle der Applikationen im DC2.0

Die Konzepte von DevOps müssen nicht nur im Betrieb, sondern bereits in der Entwicklung beachtet werden. Die Applikationen selbst müssen nach gewissen Vorgaben adaptiert werden, um sie entsprechend betreiben zu können. Die nachfolgende Liste erklärt generelle Schritte, die je Applikation umgesetzt werden müssen. Diese Definitionen garantieren, dass eine Applikation mit den Konzepten des DC2.0 konform ist und deshalb auch betrieben werden kann.

1. Der Build der Applikation muss mit Gradle durchgeführt, und über Buildpläne in Bamboo verwaltet werden. Resultierende Artefakte müssen im Artefakt Repository gespeichert werden. So wird die Reproduzierbarkeit der Artefakte sichergestellt. Alle Projekte der Red Bull Media Base werden bereits mit Gradle gebaut. Daher müssen nur mehr die entsprechenden Buildpläne in Bamboo erstellt werden.
2. Die Erstellung der virtuellen Maschine und die Installation aller Applikationen müssen mit Ansible durchgeführt werden. Dies gilt auch für jegliche Konfigurationsänderung. Es dürfen keinerlei manuellen Tätigkeiten mehr notwendig sein, um die Applikation zu installieren und zu konfigurieren.
3. Die Provisionierung der Software in den verschiedenen Umgebungen muss mit denselben Playbooks durchgeführt werden. Unterschiede zwischen den einzelnen Umgebungen müssen als Variablen definiert sein.
4. Eine entsprechende Dokumentation muss für ein Projekt vorhanden sein. Diese Dokumentation muss Informationen über Kontrollprozeduren, Notfallmaßnahmen, Monitoring und Backup enthalten. Ohne Dokumentation kann keine Verfügbarkeit für die Applikation im DC2.0 garantiert werden.
5. Bei neuen Projekten muss geprüft werden, ob die generellen Netzwerkregeln (Abschnitt 4.1.1) ausreichend sind oder ob eine Erweiterung notwendig ist.
6. Um ausreichende Monitoring Checks durchführen zu können, muss die Applikation zwei Diagnose-Endpunkte (/health und /metrics) anbieten. Der Endpunkt /health muss die Information beinhalten, ob die Applikation operativ ist oder ob es Fehler gibt. Der Endpunkt /metrics muss sinnvolle Metriken anführen, wie

beispielsweise die aktuelle Last oder Anzahl der offenen Verbindungen.

7. Als Logging Framework soll logback⁹ verwendet werden, da somit eine einfache Integration von Logstash (Abschnitt 4.3.4) möglich ist.
8. Die virtuellen Maschinen für die Applikation müssen über eine Formation (Abschnitt 4.2.5) erstellt werden. Es darf keine virtuelle Maschine geben, die manuell erstellt wurde. Das garantiert die Reproduzierbarkeit der Maschinen im DC2.0.
9. In Bamboo muss ein Deploymentplan angelegt werden. Bei Applikationen die aus einem Cluster mit mehreren Nodes bestehen, muss das Deployment mit dem Zusatz *serial: 1* ausgeführt werden. Dies garantiert, dass Ansible immer nur eine Node zur gleichen Zeit provisioniert und es somit zu keinem Ausfall kommt.
10. Applikationen die über das Internet erreichbar sein sollen, müssen in den Loadbalancer (Abschnitt 4.3.3) integriert werden. Dafür müssen sich die einzelnen Maschinen über Consul beim Loadbalancer registrieren.
11. Direkt nach der Installation der Applikation müssen sogenannte Smoke-Tests durchgeführt werden, um die grundsätzliche Verfügbarkeit und korrekte Arbeitsweise zu garantieren.

Nach Durchführung dieser Schritte ist die Applikation bereit für den Betrieb im DC2.0. Weiters garantieren sie, dass jederzeit ein lauffähiger Zustand der Applikation, innerhalb einer kurzen Dauer, wiederhergestellt werden kann.

Erfassen der Metriken: Der initiale Aufwand für das Konzept des Lifecycle kann nur schwer erfasst werden. Der Lifecycle hat sich aus den Rahmenbedingungen ergeben, die mit der Basisinfrastruktur einhergehen und wurde nicht explizit als Erstes erstellt. Er wurde nachträglich als Richtlinie für die Applikationen definiert, um den Prozess zu formalisieren. Der Aufwand in der Tabelle 4.5 bezieht sich also nur auf die Formalisierung des Lifecycle unter Absprache der Projektmitglieder.

Metrik		Menge
Initialer Aufwand	Konzeption & Formalisierung	15 PS

Tabelle 4.5: Erfasste Metriken für den Lifecycle

Würde der Lifecycle als Erstes erstellt, so würde der Aufwand entsprechend höher ausfallen. Allerdings muss für diese Vorgehensweise ein hohes Maß an Erfahrung vorhanden sein. Ansonsten würde sich die Definition des Lifecycle im Laufe der Implementierung, aller Voraussicht nach, mehrmals verändern.

⁹<http://logback.qos.ch>, Logging Framework für Java

4.2.5 Erstellung von virtuellen Maschinen

Die Automatisierung der Prozesse umfasst möglichst alle Bereiche des Betriebs. Oftmals wird die Erstellung der virtuellen Maschinen nicht berücksichtigt. Sie werden noch manuell erstellt und anschließend in die Automatisierungsprozesse eingebunden. Diese Vorgehensweise hat zur Folge, dass die Maschinen nicht reproduzierbar sind. Wird eine neue Maschine hinzugefügt, so ist nicht festgehalten wie die genaue Konfiguration lautet. Daher kann es zu unterschiedlichen Konfiguration kommen, was wiederum zu schwer nachvollziehbaren Fehler führen kann.

Einsatz bei der Red Bull Media Base: Um dieses Problem zu bewältigen, setzt die Red Bull Media Base sogenannte Formation Skripte¹⁰ ein. Durch seine API ermöglicht OpenNebula die automatisierte Erstellung von virtuellen Maschinen. Dazu muss der API die Definition der Maschine in einem strukturierten Format übergeben werden. Das ermöglicht, die Maschinen für eine Applikation als Template zu definieren und sie nachvollziehbar festzuhalten. Die Templates werden von Ansible verwendet und die Variable im Template bei der Ausführung ersetzt. Ansible übergibt das ausgefüllte Template der API von OpenNebula und erstellt die Maschine. Durch die Verwendung dieser Templates entsteht jedes mal dieselbe virtuelle Maschine. Unterschiede zwischen den Maschinen werden verhindert und Fehler dieser Art können ausgeschlossen werden. Die Auflistung 4.4 zeigt ein Beispiel für ein Template.

```

1 NAME="{{ vm_name }}"
2 CONTEXT=[
3     SET_HOSTNAME="$NAME.$NETWORK[FUNCTIONAL_NAME, NETWORK="\{{
4         primary_network}}\".{{datacenter}}.rbmbtnx.net",
5     SSH_PUBLIC_KEY="$USER[SSH_PUBLIC_KEY]",
6     NETWORK="YES"
7 ]
8 HYPERVISOR="kvm"
9 DISK=[IMAGE="centos-7.1.0", IMAGE_UNAME="opennebula-test"]
10 DISK=[DRIVER="qcow2", SIZE="100000", TYPE="fs", FORMAT="qcow2"]
11 NIC=[NETWORK_UNAME="oneadmin", NETWORK="{{primary_network}}"]
12 MEMORY="4096"
13 CPU="0.02"
14 VCPU="2"
15 SCHED_REQUIREMENTS="ID=\"12\""
16 # Red Bull spezifische Variablen
17 DATACENTER="{{datacenter}}"
18 UMGEBUNG="{{umgebung}}"
19 LAYER="{{layer}}"

```

¹⁰Der Name ist in Anlehnung an die CloudFormation von Amazon

20 | `STACK="{{service}}"`

Auflistung 4.4: Beispiel eines Formation Template für virtuelle Maschinen

In diesem Template werden Konfigurationen, wie der verwendete Hypervisor (Zeile 7), die Netzwerkschnittstellen (Zeile 10), der verfügbare RAM & CPU (Zeilen 11-13), zusätzliche Festplatten (Zeile 9) und das Basisimage (Zeile 8) für die virtuelle Maschine festgehalten. Außerdem können an einer Maschine beliebige Variablen gespeichert werden (ab Zeile 16), die man wiederum in Ansible Playbooks nutzen kann.

Erfassen der Metriken: Für die Konzeption der Formation Skripte musste die API von OpenNebula und die Struktur der Templates verstanden werden. Da jede Applikation meist unterschiedlich konfigurierte Maschinen erfordert, müssen die Formation Skripte pro Applikation erstellt werden. Meist sind nur leichte Anpassungen nötig, weswegen bereits erstellte Formation Skripte wiederverwendet werden können.

Metrik		Menge
Initialer Aufwand	Konzeption	24 PS
	je Projekt	2 PS
Durchlaufzeit	erstellen einer Maschine	ca. 5-10 Minuten

Tabelle 4.6: Erfasste Metriken für Formation Skripte

Die Durchlaufzeit in Tabelle 4.6 bezeichnet die Zeit die benötigt wird, bis eine Maschine für die Installation der Komponenten einsatzbereit ist, inklusive der Bootzeit des jeweiligen Betriebssystems. Hier wurden erhebliche Unterschiede zwischen den Hypervisoren festgestellt. Während bei KVM eine Maschine bereits nach ca. 5 Minuten bereit war, dauerte derselbe Prozess bei VMware ca. 10 Minuten. Die Bootzeit ist unter anderem davon abhängig, wie viele Komponenten im Basisimage vorhanden sind und bei Systemstart initialisiert werden. Bei der Erfassung der Durchlaufzeiten für zusätzliche Nodes (Abschnitt 3.1.3, Metrik c) wird die Bootzeit nicht mit eingerechnet, da sie für alle Applikationen gleich ist. Die in den nachfolgenden Tabellen angeführten Durchlaufzeiten für diese Metrik beziehen sich auf das Deployment bei bereits vorhandener virtueller Maschine.

4.2.6 Sonstige Systeme

Nachfolgende Systeme werden ebenfalls im Umfeld der Red Bull Media Base verwendet. Da diese Systeme aber grundsätzlich zu einer Software-Entwicklung gehören, sind sie

nur zur Vollständigkeit kurz angeführt. Sie wurden nicht im Zuge vorliegender Arbeit aufgesetzt, sondern waren bereits vorher in Verwendung.

Quellcode Verwaltung mit Git: Der Quellcode aller Projekte wird mittels eigenem Git Server in der Infrastruktur der Red Bull Media Base verwaltet. Dieser Server ist mit dem Bamboo Server verbunden, um Builds für Änderungen am Quellcode starten zu können. Für die strukturierte Verwaltung des Quellcodes der Projekte wird das Git-Flow Modell¹¹ eingesetzt.

Artefakt Repository Nexus: Um Artefakte, die bei Builds erzeugt werden, entsprechend aufzubewahren und wiederverwenden zu können, kommt Nexus¹² von Sonatype zum Einsatz. Es ermöglicht, die Artefakte in einem Maven Repository abzulegen und über eindeutige IDs abzufragen. Selbst entwickelte Applikationen werden nur von hier von Ansible abgerufen und anschließend deployed. Somit steht immer eine definierte Version der Applikation zur Verfügung.

Build-Framework Gradle: Zum Bauen der Applikation kommt bei allen Projekten Gradle zum Einsatz. Gradle ist ein auf Java Applikationen spezialisiertes Build-Framework, deren Skripte in der Sprache Groovy verfasst werden. Durch die Definition der Builds in Gradle sind diese formalisiert und reproduzierbar. Außerdem wird das Abhängigkeitsmanagement der Projekte zu anderen Projekten (vor allem Third Party) per Gradle durchgeführt. Dem Bamboo Server ist es möglich die Gradle Skripte auszuführen und die Builds zu starten. Bamboo benötigt kein spezielles Wissen über den Buildprozess jedes Projekts, sondern muss lediglich Gradle ausführen.

4.3 Basisinfrastruktur

In den folgenden Abschnitten wird die Basisinfrastruktur der Red Bull Media Base genauer erklärt. Sie geben einen Überblick welche Infrastruktur grundsätzlich notwendig ist, um Applikationen mit den Konzepten von DevOps zu betreiben. Zuerst werden die Auswahlkriterien für jede Applikation der Basisinfrastruktur angeführt. Danach folgt die Betrachtung der Funktionsweise des jeweiligen Systems und anschließend wie dieses konkret im Umfeld der Red Bull Media Base eingesetzt wird. Schlussendlich werden die in Kapitel 3 definierten Metriken erfasst.

¹¹<http://nvie.com/posts/a-successful-git-branching-model>, meist eingesetztes Git Modell

¹²<http://www.sonatype.com/nexus>, OpenSource Repository Manager

4.3.1 Erstellen von VM-Images mit Packer

Als Basis aller virtuellen Maschinen im DC2.0 wird die auf RedHat basierende Linux Distribution CentOS¹³ verwendet. Der Grund dafür ist, dass im DC1.0 alle Maschinen mit CentOS betrieben werden und damit bereits ausreichend Erfahrung gesammelt wurde. Nach Möglichkeit wird für alle Maschinen CentOS 7 verwendet. Diese Version bietet eine gute Zusammenstellung aktueller Pakete und eine geringe Fehlerquote. Für Applikationen die ältere Pakete benötigen, wird CentOS 6.6 eingesetzt. Diese Version verwendet allerdings noch *SysVinit* als Init-System, weshalb sie nach Möglichkeit vermieden wird. CentOS 7 verwendet bereits *systemd*. Andere Linux Distribution werden nicht verwendet, da dies wesentlich mehr Aufwand in der Verwaltung erfordert. RedHat nutzt beispielsweise für die Installation von Paketen *yum* und Debian *aptitude*. Die Ansible Playbooks müssen auf diese Unterschiede acht geben, was komplexere Skripte ergeben würde.

Definition des Basis-Image: Damit alle Maschinen eine gemeinsame Basis nutzen, die möglichst an die Bedürfnisse der Red Bull Media Base angepasst ist, wurde ein eigenes Basis-Image mit der kleinsten gemeinsamen Menge an Paketen und Konfigurationen definiert. Diese Ausgangsbasis wird also nicht per Ansible sichergestellt. Nachfolgend sind die Bereiche die im Basis-Image beachtet werden aufgelistet:

- Das *yum fastest mirror plugin* wird deaktiviert, damit yum nicht bei jeder Ausführung zeitaufwändig nach dem schnellsten Mirror sucht. Dadurch werden Installationen schneller durchgeführt.
- Lediglich SSH darf von außerhalb der Maschine erreichbar sein (keine weiteren Netzwerkdienste). Somit ist sichergestellt, dass keine Dienste, die zu Sicherheitslücken führen können, ausgeführt werden.
- Eine Installation von Python 2.6 muss vorhanden sein, da dies die Grundvoraussetzung für Ansible ist. Ansonsten kann kein Konfigurationsmanagement durchgeführt werden.
- Zusätzlich zu Python wird *pip* installiert, da dies die Verwaltung von Python-Paketen vereinfacht. Hierbei wird auch das Paket *httplib2* installiert, da dieses für die Registrierung beim Loadbalancer (Abschnitt 4.3.3) notwendig ist.
- Der lokale Mailrelay Server des DC2.0 wird in die Datei */etc/postfix/main.cf* eingetragen, um den Versand von Emails zu ermöglichen.

¹³<https://www.centos.org/about/>, freier Klon von RedHat

- Damit alle Maschinen dieselbe Zeit verwenden, muss ein NTP Server installiert werden, der die Uhrzeit vom globalen NTP Server des DC2.0 verwendet.
- Für die Provisionierung der Services mit Ansible wird ein Account mit Superuser Rechten benötigt, der bereits ins Image integriert ist. Jede Maschine im DC2.0 hat demnach einen Account namens *admin*, der passwortlose Superuser Rechte besitzt.
- Die root Partition des Image beträgt 10GB. Dies ist groß genug für Applikationen, die nur Konfigurationen und Log-Dateien ablegen. Maschinen die Daten speichern, erhalten bei ihrer Erstellung eine zusätzliche virtuelle Festplatte.
- Neben KVM wird auch VMware als Hypervisor eingesetzt, wobei jeder ein eigenes Basis-Image benötigt. Im Image für VMware werden die VMware Tools installiert, womit zusätzliche Verwaltungsfunktionen direkt verfügbar sind. Im KVM-Image werden keine weiteren Tools installiert.
- CentOS 7 verwendet ein neues Schema für die Benennung der Netzwerkschnittstellen. Damit das bisherige Schema, wie bei CentOS 6.6 verwendet wird, muss bei CentOS 7 ein zusätzliches Kernel Flag gesetzt werden.

Verwaltung der Images mit Packer: Bei zwei Distributionen (CentOS 6.6 / 7) für zwei Hypervisoren (KVM / VMware) müssen vier Images verwaltet werden. Für die Entwicklungsumgebung (Vagrant) mit Hypervisor Virtualbox sind zwei weitere Images notwendig. Um den Verwaltungsaufwand so gering wie möglich zu halten, muss die Erstellung der Images automatisiert erfolgen. Damit wird der Prozess vereinfacht und reproduzierbar. Für diese Aufgabe wird Packer¹⁴, ein OpenSource System, das diese Funktionen bietet, eingesetzt. Für CentOS 6.6 und für 7 werden die Basis-Images in textueller Form definiert, woraus dann Images für die unterschiedlichen Hypervisoren erzeugt werden. Es müssen daher nur zwei Definitionen gepflegt werden anstatt unzähliger Images. Ein weiterer Vorteil von Packer ist, dass Provisionierungssysteme wie Ansible direkt in den Prozess integrieren werden können. So ist es möglich, Software bzw. Konfigurationen im Image per Ansible anzupassen und somit auf die gleiche Weise zu handhaben, wie auf den virtuellen Maschinen. Kein zusätzliches System ist notwendig und die Ansible Skripte können wiederverwendet werden.

Erfassen der Metriken: Die Tabelle 4.7 zeigt die erfassten Metriken für Packer. Die Installation von Packer gestaltet sich sehr einfach, da nur ein ZIP File entpackt werden muss, das alle nötigen Bestandteile enthält. Wird Ansible für die Installation

¹⁴<https://www.packer.io/intro>, OpenSource System für die Erzeugung von Images

von Software im Image verwendet, so muss auf der Maschine auch Ansible installiert sein. Der entsprechende Aufwand ist in Tabelle 4.2 zu finden.

Metrik		Menge
Initialer Aufwand	Installation von Packer	0.5 PS
	Allgemeine Definition des Basis-Image	16 PS
	Packer Definition des Basis-Image	32 PS
Durchlaufzeit	Erstellung eines Images	ca. 10-20 Minuten

Tabelle 4.7: Erfasste Metriken für Packer

Die allgemeine Definition des Basis-Image hat etwa zwei Arbeitstage in Anspruch genommen. Die Definition in textueller Form für Packer und das Testen des Image benötigten zusätzliche vier Arbeitstage. Dies wurde von Alexander Dobriakov durchgeführt, da er bereits Erfahrung mit Packer gesammelt hat. Die Dauer zum Erstellen eines Image für einen Hypervisor, hängt vom gewählten Hypervisor ab. Der Prozess zum Erstellen des Image für CentOS7 auf KVM ist nach ca. 10 Minuten abgeschlossen. Für andere Hypervisoren beträgt die Dauer zwischen 10 und 20 Minuten.

4.3.2 Automatic Service Discovery/Registry mit Consul

Viele Systeme, die in den letzten Jahren entwickelt wurden, sind bereits auf einen verteilten Ansatz ausgelegt. Sie ermöglichen Cluster automatisch zu bilden und ihre Konfigurationen dynamisch an die Anzahl der vorhandenen Nodes anzupassen. Ein Beispiel hierfür ist Elasticsearch, das in der Logging Infrastruktur (Abschnitt 4.3.4) verwendet wird. Solche Applikationen minimieren manuelle Tätigkeiten und verwalten sich größtenteils selbst. Dies trifft aber nicht auf die meisten älteren Applikationen zu. Daher ist ein Konzept nötig, um die Konfiguration solcher Applikationen dynamisch anpassen zu können. Ziel dabei ist eine sogenannte Service Registry aufzubauen, in der alle verfügbaren Applikationen und deren Nodes registriert sind. Dafür wird die in Abschnitt 2.2.4 erwähnte Applikation Consul verwendet. Aus dieser sollen dann dynamisch Konfigurationen für die Applikationen erstellt werden.

Grundsätzliche Funktionsweise von Consul: Applikationen können sich bei Consul registrieren und mitteilen, welche Services sie anbieten. Consul verwaltet diese Information und liefert sie auf Anfrage aus. Alle Nodes, die den Consul Agent installiert haben, können sich zu einem Cluster verbinden und Informationen austauschen. Grundsätzlich gibt es zwei Betriebsmodi des Consul Agent, als *Server* oder als *Client*.

Der Consul Agent im Server Modus ist dafür zuständig, die Service-Informationen der einzelnen Applikationen zu speichern und zu verwalten. Alle Consul Server im Cluster replizieren diese Informationen untereinander. Die einzelnen Nodes ermitteln automatisch einen sogenannten *Leader*, der zuständig ist für die Verwaltung des Clusters. Alle anderen Server sind sogenannte *Follower* und agieren je nach konfigurierter Konsistenz des Consul Clusters. Damit wird Ausfallsicherheit garantiert, solange ein Server vorhanden ist.

Alle Applikationen die Services bereitstellen, verwenden den Consul Agent im Client Modus. Diese Clients müssen sich initial im Consul Cluster registrieren, wofür die Node nur ein beliebiges Mitglied des Clusters kennen muss. So ist an zentraler Position gespeichert, welche Services wo vorhanden sind. Weiters ist es möglich, für jeden Service bzw. für jede Node, Werte als Key/Value Paare zu speichern. Diese werden mit ausgeliefert und können entsprechend in der Applikation verarbeitet werden.

Benötigt eine Applikation Informationen über einen bestimmten Service, so kann diese über eine HTTP REST Schnittstelle beim Consul Cluster abgefragt werden. Dazu muss die Applikation wiederum nur ein Mitglied des Clusters kennen. Ist ein Consul Client installiert, kann dies sogar lokal geschehen. Das ermöglicht eine Kommunikation mit nahezu allen Applikationen. Mit der erhaltenen Information kann anschließend die Konfiguration angepasst werden, um einen Service zu nutzen.

Consul bietet demnach eine einfache und ausfallsichere Möglichkeit, Service Informationen an zentraler Stelle zu speichern und sie leicht anderen Applikationen zugänglich zu machen. Möchte man allerdings die Konfiguration bestehender Third-Party Applikationen, wie beispielsweise HAProxy, damit verwalten, benötigt man zusätzliche Systeme, die diese Tätigkeit übernehmen. Die Entwickler von Consul bieten eine dazugehörige Applikation namens *Consul-Template*¹⁵ dafür an. Damit können Templates für Konfigurationsdateien definiert werden, deren Inhalt je nach Clusterstatus angepasst wird. So lassen sich auch die Konfigurationen von nicht dynamischen Services automatisiert anpassen. Consul-Template generiert bei Events im Cluster, die Konfigurationsdatei mit den geänderten Informationen neu. Ein Event ist beispielsweise Node hinzugefügt oder Node gelöscht.

Aufbau des Consul Clusters: In der offiziellen Dokumentation wird empfohlen, einen Cluster mit 3-5 Consul Servern zu verwenden. Deswegen wurden im DC2.0 drei dedizierte Maschinen mit Consul im Server Modus aufgebaut. Ein solcher Cluster existiert je Umgebung, um diese sauber voneinander zu trennen. Das verhindert, dass durch fehlerhafte Konfiguration, Services unterschiedlicher Umgebungen vermischt

¹⁵<https://github.com/hashicorp/consul-template>, Template Rendering Applikation

werden. Für Applikationen die aus dem Internet erreichbar sein sollen, ist aufgrund des Loadbalancer-Aufbaus (Abschnitt 4.3.3) der Consul Client zwingend notwendig. Grundsätzlich wird jede Applikation im DC2.0 in der Service Registry registriert. Da der Consul Client nur minimale Systemressourcen benötigt, kann er auf jeder Maschine installiert werden.

Ein Sicherheitsaspekt der hier beachtet werden muss ist, dass jedes Mitglied lokal die Service Informationen abfragen kann. Ist eine Maschine kompromittiert, so kann ein Angreifer leicht weitere Ziele ausfindig machen. Auffällige Netzwerkscans, die oft zur Entdeckung des Angreifers führen, entfallen. Um dieses Risiko zu minimieren, wird im DC2.0 ein Consul Cluster je Subnetz aufgebaut. Somit kann ein Angreifer nur die Services eines Subnetzes und einer Umgebung abfragen, aber nicht das gesamte Datacenters.

Erfassen der Metriken: Wie in Tabelle 4.8 ersichtlich, war die Automatisierung des Consul Cluster Deployments innerhalb von zwei Wochen abgeschlossen. Dieser Zeitraum beinhaltet den Aufbau eines Server Clusters, sowie die Installation des Clients. Hierbei konnten keine Ansible Roles wiederverwendet werden, da Consul die erste Applikation war, die aufgebaut wurde.

Metrik		Menge
Initialer Aufwand	Implementation	80 PS
	wiederverwendete Roles	0 Stk
	je Applikation	1 PS
Komplexität	Anzahl der Komponenten	2
	verteiltes System	Ja
Durchlaufzeit	a.) eine Node	ca. 5 Minuten
	b.) alle Nodes	ca. 15 Minuten
	c.) zusätzliche Node	ca. 7 Minuten

Tabelle 4.8: Erfasste Metriken des Consul Clusters

Die Automatisierung war unkompliziert, da nur die zwei Komponenten Consul und Supervisor¹⁶, nötig waren. Supervisor verwaltet den Consul Prozess. Obwohl Consul ein verteiltes System ist, erhöht sich die Komplexität nicht, da es speziell auf einen verteilten Einsatz ausgelegt ist. Der Aufwand je zusätzlicher Applikation beschränkt sich auf die Anpassung der Konfiguration des Clients und dauert im Durchschnitt eine Stunde.

¹⁶<http://supervisord.org/introduction.html>, Applikation zur Prozesskontrolle

Werden alle Consul Server gleichzeitig abgeschaltet, so wird der Cluster inkonsistent und muss neu aufgebaut werden. Um dies zu verhindern, wird bei einem Update immer nur ein Server nach dem anderen aktualisiert. Daher ist ein Deployment auf allen Nodes (Tabelle 4.8, Durchlaufzeit b) ca. drei mal so lang wie auf einer Node (a). Die Dauer steigt linear mit der Anzahl der Consul Server. Dies ist allerdings vernachlässigbar, da ein unterbrechungsfreier Service garantiert wird. Das Hinzufügen einer zusätzlichen Node benötigt etwas mehr Zeit, als ein erneutes Deployment, was auf den Download der fehlenden Komponenten zurückzuführen ist. Bei neuerlichem Deployment sind diese Komponenten bereits vorhanden und diese Schritte entfallen.

4.3.3 Loadbalancing mit HAProxy

Um die anfallende Last auf die Nodes einer Applikation optimal aufteilen zu können, wird ein Loadbalancer benötigt. Da im DC2.0 vor allem verteilte Applikationen zum Einsatz kommen, musste ein entsprechendes Loadbalancer Konstrukt aufgebaut werden. Dieses Konstrukt muss im Besonderen folgende Punkte beachten:

- Die Anzahl an Public IPv4 Adressen, die der Red Bull Media Base zur Verfügung stehen, ist begrenzt. Aktuell sind nur 160 Adressen vorhanden und voraussichtlich werden keine zusätzlichen Adressen möglich sein.
- Es sollten keine Applikationen selbst Public verfügbar gemacht werden. Jede Applikation, die aus dem Internet erreichbar sein sollte, muss dies über den Loadbalancer durchführen.
- Es darf keine Applikation mit HTTP ausgeliefert werden. Nur sichere HTTPS Verbindungen sind erlaubt.

Funktionsweise des Loadbalancers: Grundsätzlich gibt es im DC2.0 zwei Arten von Loadbalancern. Erstens, ein globaler Loadbalancer für das gesamte Datacenter und Zweitens, dedizierte Loadbalancer für spezielle Applikationen. Alle Applikationen sollten den globalen Loadbalancer nutzen, nur in Ausnahmefällen darf eine Applikation einen dedizierten Loadbalancer verwenden. Ein Ausnahmefall ist, wenn sich die Applikation in einem anderen Subnetz als dem SERVICE Subnetz befindet. Der globale Loadbalancer ist primär dafür zuständig, Applikationen aus einem SERVICE Subnetz ins Internet freizuschalten.

Würde jede Applikation einen dedizierten Loadbalancer verwenden, so wären pro Applikation mindestens 12 Public IP-Adressen nötig. Jeweils eine IP-Adresse für die zwei Loadbalancer Nodes (für Ausfallsicherheit), plus eine IP-Adresse für die virtuelle IP. Diese drei IP-Adressen multiplizieren sich mit der Anzahl der Umgebungen (Vier),

da für eine saubere Trennung jede Umgebung seinen eigenen Loadbalancer verwenden sollte. Bei 12 IP-Adressen pro Applikation könnten somit maximal 13 Applikationen ins Internet freigeschaltet werden, unter der Voraussetzung, dass nie mehr als zwei Loadbalancer Nodes im Einsatz sind. Durch den globalen Loadbalancer vermindert sich dieses Problem, da insgesamt nur zwei Loadbalancer Nodes im Einsatz sind. Die IP-Adressen für die Nodes der dedizierten Loadbalancer entfallen. Hochgerechnet auf alle Umgebungen, werden für den Loadbalancer nur mehr acht IP-Adressen benötigt. Je Applikation wird nur mehr eine IP-Adresse bzw. vier IP-Adressen für alle Umgebungen, für die virtuelle IP benötigt. Es bleiben 152 Adressen, womit 38 Applikationen bedient werden können. Um unbegrenzt viele Applikationen bedienen zu können, kann eine 1:n Verbindung, zwischen virtueller IP und den Applikationen, genutzt werden. Die Applikationen teilen sich eine einzige virtuelle IP und der Loadbalancer entscheidet anhand der Subdomain, an welche Applikation die Anfrage weitergeleitet wird. So können beispielsweise `app01.redbullmediahouse.com` und `app02.redbullmediahouse.com` dieselbe öffentliche IP-Adresse verwenden.

Weiters übernimmt der Loadbalancer die SSL-Terminierung, um die Kommunikation ins Internet zu verschlüsseln. Die Kommunikation zwischen dem Loadbalancer und den einzelnen Applikationen geschieht hierbei unverschlüsselt. Dies stellt aber kein Problem dar, da sich die Kommunikation nur innerhalb des DC2.0 bewegt. Dieser Ansatz hat den Vorteil, dass sich nicht mehr jede einzelne Applikation um die Verschlüsselung kümmern muss. Dies geschieht an einer zentralen Stelle, womit die Zertifikate auch nur hier verwaltet werden müssen. Ein Nachteil ist allerdings, dass Verschlüsselung und Entschlüsselung rechenintensive Aufgaben sind. Daher ist mit höheren Anforderungen an eine Loadbalancer Node zu rechnen. Durch die horizontale Skalierbarkeit kann dieses Problem bewältigt werden.

Aufbau des Load Balancers: Jede Node des Loadbalancers befindet sich gleichzeitig in zwei Subnetzen und benötigt deswegen zwei Netzwerkschnittstellen. Eine davon befindet sich im PUBLIC Subnetz, für die Kommunikation mit dem Internet. Die Andere befindet sich im SERVICE Subnetz der jeweiligen Umgebung, für die Kommunikation mit den Applikationen. Abbildung 4.3 zeigt die einzelnen Komponenten einer Node und deren Zusammengehörigkeit. Die Komponenten sind:

- **Keepalived¹⁷:** Es überwacht den HAProxy der aktuellen Master-Node des Loadbalancers. Ist der HAProxy der Master-Node fehlerhaft, weist Keepalived die virtuelle IP einer nicht fehlerhaften Node zu. Diese wird so zum neuen Master und übernimmt die Verteilung der Anfragen.

¹⁷<http://www.keepalived.org>, Management von virtuellen IPs für Loadbalancing

- **HAProxy**¹⁸: Er fungiert als Loadbalancer und verteilt die, über die virtuelle IPs ankommenden Anfragen, auf die Applikationen. HAProxy unterstützt die vorher beschriebenen Konzepte des Loadbalancings, sowie die SSL Terminierung.
- **Consul**: Der Loadbalancer erkennt mit Consul, welche Nodes einer Applikation vorhanden sind und wohin er Anfragen weiterleiten kann. Daher ist auch ein Loadbalancer pro Subnetz notwendig, da Consul auch nur pro Subnetz existiert.
- **Consul-Template**: Sobald es Veränderungen an den Nodes einer Applikation gibt, generiert diese Komponente anhand eines Templates eine neue Konfigurationen für Keepalived und HAProxy, um auf diese Änderungen zu reagieren.

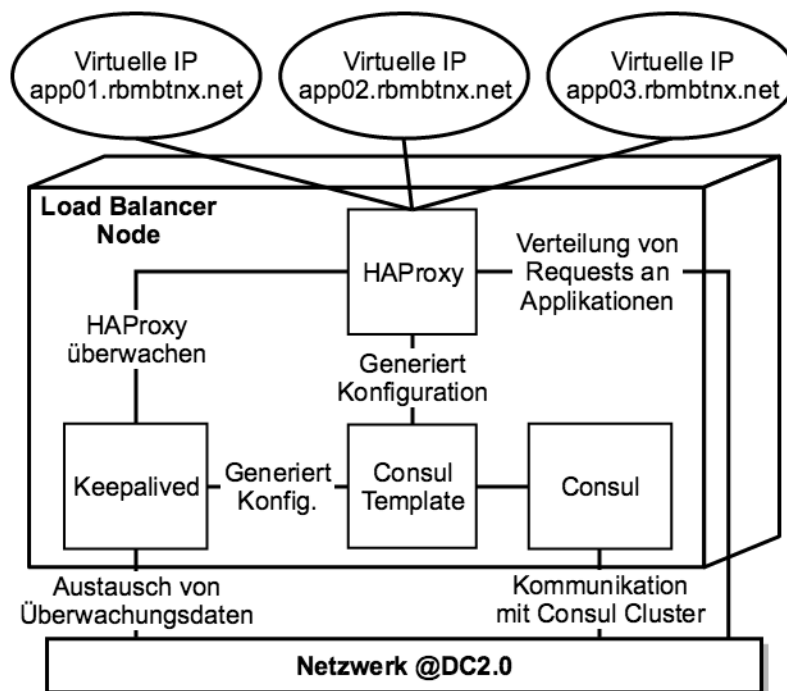


Abbildung 4.3: Grundsätzlicher Aufbau einer Loadbalancer Node. Alle Komponenten kommen je Node vor und kommunizieren über das Netzwerk.

Erfassen der Metriken: Die Tabelle 4.9 listet die erfassten Metriken für den Aufbau des Loadbalancers. Ein Großteil des Aufwands für die Erstellung des Loadbalancers, war für die Entwicklung des Konzeptes notwendig. Um ein Konstrukt zu ermitteln, das den Anforderungen entspricht, musste zuerst ein Proof of Concept erstellt und getestet werden. Die Erstellung des automatisierten Deployments konnte danach mit wenig Aufwand durchgeführt werden. Hierfür können die Roles für Consul und Supervisor wiederverwendet werden. Dennoch ist durch die Anzahl der Komponenten die Komplexität des Loadbalancers höher als bei Consul, weswegen mehr Aufwand für die Implementierung notwendig war. Der zusätzliche Aufwand je Applikation, für die

¹⁸<http://www.haproxy.org/>, hochperformanter, leichtgewichtiger Loadbalancer

Integration in den Loadbalancer ist verschwindend gering, da nur der Consul Client installiert werden muss.

Metrik		Menge
Initialer Aufwand	Konzept & Implementation	230 PS
	wiederverwendete Roles	2 Stk
	je Applikation	0 PS
Komplexität	Anzahl der Komponenten	5
	verteiltes System	Ja
Durchlaufzeit	a.) eine Node	ca. 7 Minuten
	b.) alle Nodes	ca. 16 Minuten
	c.) zusätzliche Node	ca. 10 Minuten

Tabelle 4.9: Erfasste Metriken des Loadbalancers

Der Loadbalancer darf bei einem Update nicht komplett ausgeschaltet werden, ansonsten gibt es Unterbrechungen in der Erreichbarkeit der Applikationen. Daher darf immer nur eine Node nach der anderen aktualisiert werden. Aus diesem Grund ist die Zeit für alle Nodes (b), ähnlich wie bei Consul, die Zeit einer Node (a) mal der Anzahl der Nodes. Allerdings muss der Loadbalancer selten neu provisioniert werden, da die häufige Änderung der Konfiguration von Consul-Template durchgeführt wird. Das Deployment einer zusätzlichen Node braucht etwas länger als ein erneutes Deployment. Der Grund hierfür ist, wie bei Consul, der Download der Komponenten.

4.3.4 Log-Aggregation mit Logstash

Im klassischen Betrieb werden Log-Dateien auf den jeweiligen Maschinen gesammelt. Kommt es zu Problemen, können sich die Entwickler auf einer Maschine einloggen und Log-Dateien abrufen. Das setzt voraus, dass entsprechende Zugangsberechtigungen existieren, die für jede neue Maschine eingerichtet werden müssen. Bei DevOps ändert sich allerdings die Infrastruktur häufig. Die Anzahl der Maschinen ist sehr volatil, um beispielsweise auf verschiedene Lastsituationen reagieren zu können. Es erfordert hierbei einen hohen Aufwand, die Zugangsberechtigungen zu verwalten. Um dem Problem entgegenzuwirken, muss ein zentraler Ansatz verfolgt werden. Die Log-Dateien aller Applikationen und Maschinen werden an einer zentralen Stelle gesammelt und können von hieraus von den Entwicklern eingesehen werden. Die Red Bull Media Base setzt im DC2.0 für diese Aufgabe den sogenannten ELK-Stack ein. Dieser besteht aus

Logstash¹⁹, Elasticsearch²⁰ und Kibana²¹ [16].

Grundsätzliche Funktionsweise von Logstash: Dieses System ermöglicht es, die Log-Nachrichten über sogenannte Log-Shipper, die lokal auf den Maschinen installiert sind, direkt an die zentrale Instanz zu schicken. Somit müssen dem ELK-Stack die Maschinen nicht bekannt sein. Die einzelnen Applikationen schicken automatisch ihre Logs und es ist keine spezielle Konfiguration des ELK-Stacks notwendig. Der ELK-Stack analysiert und indiziert die ankommenden Daten und macht sie für die Benutzer durchsuchbar. Das Schema in Abbildung 4.4 zeigt den grundsätzlichen Aufbau des ELK-Stacks.

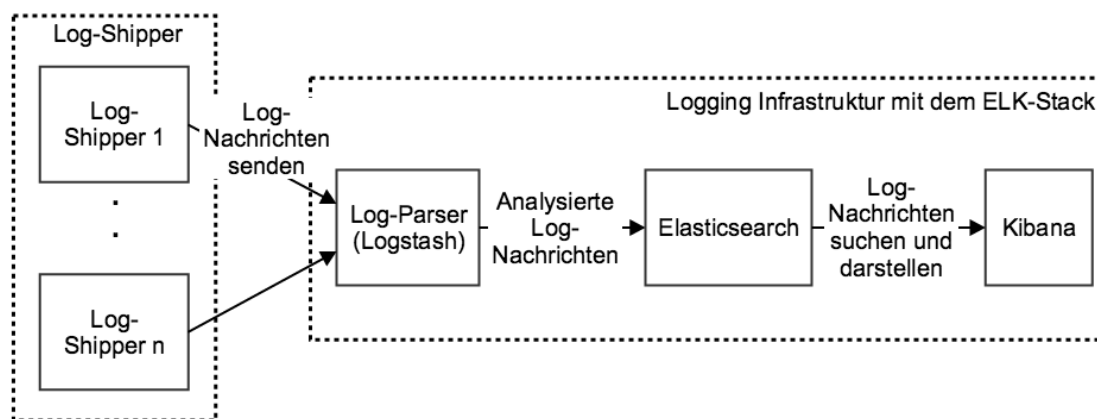


Abbildung 4.4: Grundsätzliches Funktionsschema des ELK-Stacks, adaptiert aus [16]. Die Pfeilrichtungen geben den Datenfluss zwischen den Komponenten an.

Der ELK-Stack besteht grundsätzlich aus drei Komponenten für die Analyse, Indizierung & Speicherung sowie Visualisierung der Log-Nachrichten. Die Log-Lieferung kann über diverse Applikationen (siehe Abschnitt 4.3.4) erfolgen und wird üblicherweise nicht zum ELK-Stack gezählt. Sobald ein Client Log-Nachrichten produziert, werden diese vom Log-Shipper an die zentrale Logging Infrastruktur geschickt. Dort werden die Nachrichten von Logstash analysiert und entsprechend aufbereitet. Zu diesem Zeitpunkt ist es möglich, die Log-Nachrichten durch weitere Daten anzureichern oder unnötige Teile zu filtern. Nach erfolgreicher Analyse, werden sie an Elasticsearch weitergeleitet. Elasticsearch indiziert die Daten anhand ihres Erstellungsdatums und speichert sie. Mit Hilfe von Kibana können die Benutzer die Daten aus Elasticsearch abfragen, filtern und entsprechend visualisieren.

Logstash bietet eine Vielzahl von möglichen Ausgabeformaten²² für Speicher wie Solr,

¹⁹<http://logstash.net/>, Log-Parser auf Basis von Java

²⁰<https://www.elastic.co/products/elasticsearch>, NoSQL Datenspeicher

²¹<https://www.elastic.co/products/kibana>, Javascript Applikation zur Datenvisualisierung

²²<http://logstash.net/docs/1.4.2/>, Spalte *outputs* bei *plugin documentation*

S3 oder Redis an. Es muss nicht Elasticsearch verwendet werden. Allerdings sind die Komponenten im ELK-Stack miteinander bereits erprobt und werden von demselben Unternehmen (Elastic) entwickelt. Die Kompatibilität der einzelnen Komponenten untereinander sollte daher gegeben sein [16].

Aufbau der Logging Infrastruktur: Abbildung 4.5 zeigt den grundsätzlichen Aufbau der verschiedenen Nodes in der Logging Infrastruktur. Auf der Log-Parser Node (Abbildung 4.5a) läuft Logstash als Prozess. Logstash selbst ist eine zustandslose Applikation, in der eine Log-Nachricht immer nur von einem Prozess verarbeitet wird. Es besteht keine Abhängigkeit zu anderen Log-Nachrichten. Daher ist es möglich, mehrere Logstash Instanzen parallel zu betreiben, ohne dass Konflikte entstehen. Die Hauptaufgabe der Elasticsearch Nodes (Abbildung 4.5b) ist die Speicherung und Indizierung der Daten. Elasticsearch ist eine hochperformante NoSQL Datenbank und bietet die Möglichkeit, einen verteilten Cluster zu bilden. Die Daten werden automatisch auf allen operativen Nodes verteilt. Es ist also möglich, mehrere Log-Parser und Elasticsearch Nodes parallel in einem Cluster zu betreiben. Kibana ist eine Javascript Applikation, die auf dem Rechner der Benutzer läuft und muss somit nicht skaliert werden. Da Logstash wie auch Elasticsearch CPU intensive Applikationen sind, werden sie auf unterschiedlichen Nodes getrennt betrieben, um die Last besser zu verteilen.

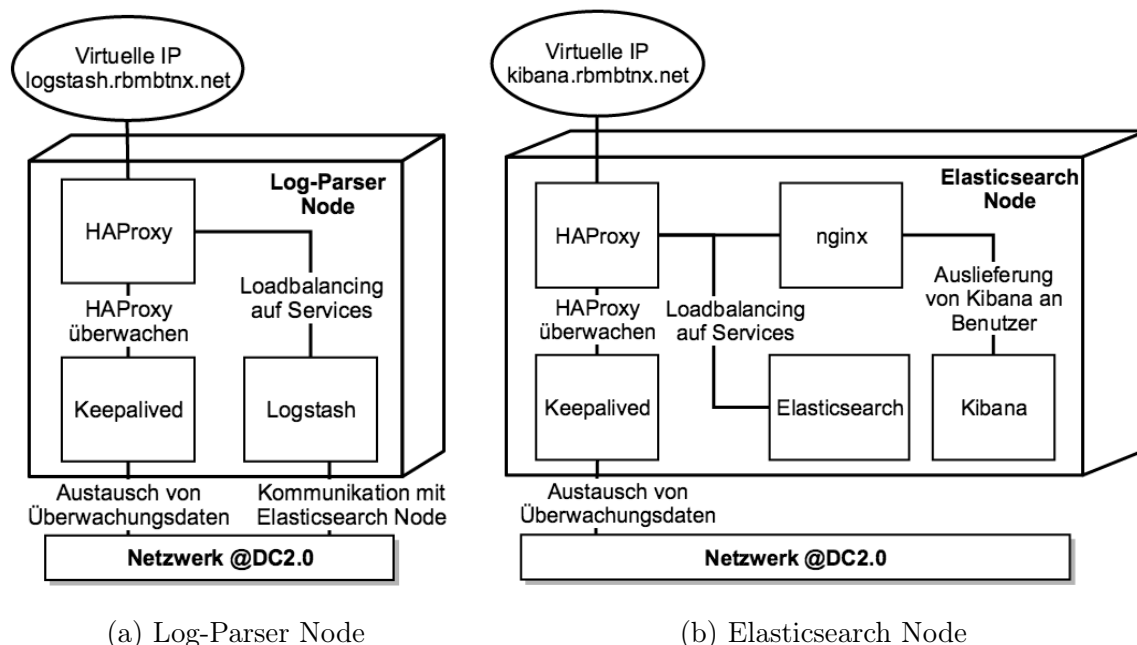


Abbildung 4.5: Grundsätzlicher Aufbau der Nodes unseres ELK-Stacks in der Logging Infrastruktur. Alle Komponenten kommen je Typ von Node vor und sie bilden Cluster, um sich über alle Nodes gleichen Typs hinweg zu synchronisieren.

Folgende Aufgaben erfüllen die weiteren Komponenten. Nicht alle Komponenten sind auf beiden Nodetypen installiert, siehe Abbildung 4.5:

- **Keepalived:** Keepalived übernimmt hierbei dieselbe Aufgabe wie beim Loadbalancer (siehe Abschnitt 4.3.3).
- **HAProxy:** Er fungiert als Loadbalancer und verteilt die ankommenden Anfragen auf alle funktionalen Log-Parser/Elasticsearch Nodes im Cluster.
- **nginx²³:** Dies ist ein leichtgewichtiger Webserver, der für die Auslieferung von Kibana an die Benutzer zuständig ist. Außerdem fungiert er als Reverse-Proxy für die Kommunikation von Kibana mit Elasticsearch. Damit werden Daten immer nur von der ursprünglichen Node von Kibana abgefragt und die Last wird verteilt. Außerdem wird damit die CORS²⁴ Problematik umgangen.

Auf jeder Node des jeweiligen Typs sind alle in Abbildung 4.5 dargestellten Komponenten installiert. Durch diesen Aufbau ist die Möglichkeit zur horizontalen Skalierung, in allen relevanten Teilen der Logging Infrastruktur, gegeben. Daher ist der ELK-Stack zukunftssicher, im Bezug auf steigende Lastanforderungen und Datenmengen. Außerdem weist dieser Aufbau einen hohen Grad an Ausfallsicherheit auf. Solange jeweils eine Log-Parser und Elasticsearch Node korrekt funktionieren, ist der Service verfügbar. Durch die Replikation der Daten auf alle Elasticsearch Nodes ist es unwahrscheinlich, dass Daten verloren gehen. Es ist nicht anzunehmen, dass der gesamte Cluster innerhalb kurzer Zeit fehlerhaft ist und somit eine Datenrettung nicht mehr möglich ist. Aus diesem Grund kann auf ein Backup verzichtet werden.

Log-Shipper und Konfiguration: Für die Log-Lieferung bietet Logstash eine Menge an möglichen Inputs²⁵ an. Im aktuellen Aufbau sind zwei Möglichkeiten der Lieferung umgesetzt, die alle nötigen Anforderungen erfüllen. Diese sind rsyslog²⁶ und logback²⁷. Ersteres ist ein Log-Verarbeitungssystem, das in Linux Distributionen weit verbreitet ist. Mit rsyslog ist es möglich, Log-Dateien auf einer Maschine zu überwachen (ähnlich dem Befehl *tail -f*) und neue Log-Nachrichten an Logstash weiterzuleiten. Der Vorteil hierbei ist, dass an der Applikation keine Änderung notwendig ist, sondern lediglich rsyslog konfiguriert werden muss. Die zweite Möglichkeit, logback, ist ein Logging-Framework für Java Applikationen, das im Umfeld der Red Bull Media Base häufig eingesetzt wird. Logback ermöglicht es, durch Konfiguration sogenannte *Log-Appender* hinzuzufügen. Alle generierten Log-Nachrichten werden an diese weitergeleitet. Auch hier sind keine Änderungen der Applikation notwendig, es muss lediglich ein Log-Appender für Logstash hinzugefügt werden.

²³<http://nginx.org/en/>, wird u.a. von Netflix und Wordpress eingesetzt

²⁴<http://www.w3.org/TR/cors/>, Cross-Origin Resource Sharing

²⁵<http://logstash.net/docs/1.4.2/>, Spalte *inputs* bei *plugin documentation*

²⁶<http://www.rsyslog.com/doc/v8-stable/>, weit verbreitet auf Linux Systemen

²⁷<http://logback.qos.ch/>, Nachfolger von Log4J

Um die Log-Daten entsprechend aufbereiten zu können, werden vier zusätzliche Metadaten definiert, die jede Log-Nachricht enthalten muss. Diese sind:

- **Environment:** Info aus welcher Umgebung die Log-Nachricht kommt (Produktion, Pre-Produktion, Staging, Test).
- **Stack:** Info zu welchem Applikationsstack die Log-Nachricht gehört.
- **Layer:** Info zu welchem Layer eines Applikationsstacks die Log-Nachricht gehört.
- **Datacenter:** Info in welchem Rechenzentrum die Applikation steht. Das ist bereits vorbereitend falls zusätzliche Rechenzentren an anderen Standorten aufgebaut werden.

Rsyslog sowie logback ermöglichen, die zusätzlichen Metadaten an jede Log-Nachricht hinzuzufügen.

Erfassen der Metriken: Die Logging Infrastruktur ist entsprechend der Menge an Komponenten komplex aufzubauen. Allerdings konnten vom Loadbalancer bereits ei-

Metrik		Menge
Initialer Aufwand	Implementation	120 PS
	wiederverwendete Roles	3 Stk
	je Applikation	0 PS
Komplexität	Anzahl der Komponenten (Log-Parser)	5
	Anzahl der Komponenten (Elasticsearch)	7
	verteiltes System (Beide)	Ja
Durchlaufzeit	a.) eine Node (Log-Parser)	ca. 3 Minuten
	a.) eine Node (Elasticsearch)	ca. 2.5 Minuten
	b.) alle Nodes	ca. 19 Minuten
	c.) zusätzliche Node (Log-Parser)	ca. 6 Minuten
	c.) zusätzliche Node (Elasticsearch)	ca. 5 Minuten

Tabelle 4.10: Erfasste Metriken der Logging Infrastruktur

nige Roles für Keepalived, Supervisor und HAProxy wiederverwendet werden, um den Aufwand entsprechend zu reduzieren. Die Tabelle 4.10 zeigt die konkreten Werte. Im initialen Aufwand inbegriffen, ist die Automatisierung der Installation des Log-Shippers. Dadurch fällt je Applikation nur mehr ein geringer Aufwand an. Um einen unterbrechungsfreien Logging Service zu garantieren und keine Log-Nachrichten zu verlieren,

wird bei einem Deployment auf allen Nodes immer eine Node nach der anderen aktualisiert. Aktuell besteht der Logging Cluster aus fünf Nodes. Zwei Log-Parser Nodes und drei Elasticsearch Nodes. Daher ist die Metrik b, wie erwartet ca. fünf mal so lang wie Metrik a. Die Deploymentzeiten einer neuen Node sind etwas länger, was auf den Download der Komponenten zurückzuführen ist. Der längste Download hierbei ist Java, der jedoch bei einem erneuten Deployment entfällt.

4.3.5 Monitoring mit Sensu

Auch beim Monitoring, muss ein anderer Ansatz als üblich verfolgt werden. In klassischen Umgebungen wird oft Nagios²⁸ als Monitoring Infrastruktur eingesetzt. Nagios verfolgt einen zentralen Ansatz, bei dem alle Maschinen explizit bei der zentralen Instanz konfiguriert werden müssen. Daher muss die Konfiguration jedes mal angepasst werden, wenn Maschinen erstellt oder entfernt werden. Um dieses Problem zu lösen, muss eine Monitoring Infrastruktur mit dezentralem Ansatz aufgebaut werden. Neue Maschinen sollen sich selbstständig beim Monitoring System anmelden, ohne dieses jedes mal neu konfigurieren zu müssen. Dadurch können Maschinen dynamisch hinzugefügt und entfernt werden, ohne dass manuelle Schritte notwendig sind. Außerdem muss die Monitoring Infrastruktur horizontal skalierbar sein, um steigende Lastanforderungen bewältigen zu können. Die Red Bull Media Base setzt hierfür Sensu²⁹ ein, das die gewünschten Ansätze verfolgt.

Grundsätzliche Funktionsweise von Sensu: Sensu ist ein leichtgewichtiges Monitoring Framework, dessen grundsätzlicher Aufbau in Abbildung 4.6 dargestellt ist. Der Sensu Server ist die zentrale Instanz, die alle Monitoring Checks auslöst, auswertet und speichert. Auf den Maschinen läuft jeweils der ressourcensparende Sensu Client, der die konkreten Checks durchführt. Die Kommunikation zwischen Server und Clients erfolgt über eine verteilte Messaging-Queue³⁰, über die Nachrichten dezentral ausgetauscht werden. Der Server sendet über die Queue in regelmäßigen Abständen, Aufforderungen zur Ausführung von Monitoring Checks. Die Clients lauschen auf diese Befehle und führen, sofern für sie relevant, entsprechende Checks aus. Die Last verteilt sich somit, da jeder Client die Checks selbst ausführt und diese nicht vom Server durchgeführt werden. Das Ergebnis der Checks wird über die Messaging-Queue zurück an den Server gesendet, wo diese ausgewertet und am Dashboard dargestellt werden.

Wird eine zusätzliche Maschine hinzugefügt, so meldet sich diese automatisch beim

²⁸<http://www.nagios.org/>, weit verbreitetes Monitoring Framework der letzten Jahre

²⁹<http://sensuapp.org/>, dezentrale Monitoring Infrastruktur mit Hilfe von Messaging Queues

³⁰Sensu verwendet hierfür RabbitMQ: <https://www.rabbitmq.com/>

Server an und es ist keine Änderung der Konfiguration notwendig. Ab diesem Zeitpunkt ist der neue Client dem Server bekannt und dieser muss nur auf die Nachrichten in der Queue achten. Der Server muss somit nicht wissen, wo und wie viele Clients vorhanden sind. Lediglich die Messaging-Queue muss für Beide erreichbar sein. Das Entfernen von Maschinen kann über das Dashboard erfolgen und benötigt ebenso keine Konfigurationsänderung.

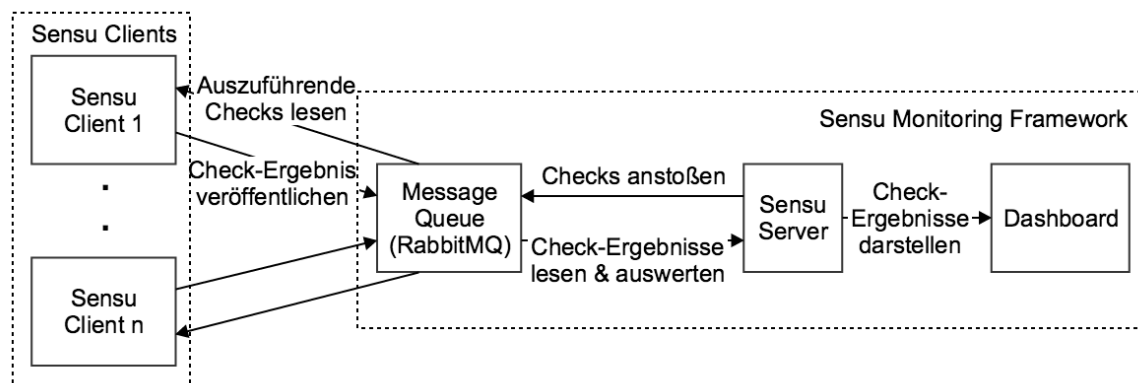


Abbildung 4.6: Grundsätzliches Funktionsschema der Sensu Monitoring Infrastruktur. Die Pfeilrichtungen geben den Datenfluss zwischen den Komponenten an.

Das Ergebnis eines Checks kann grundsätzlich zwei Formen annehmen. Herkömmliche Checks überprüfen die korrekte Funktionsweise einer Applikation und liefern grundsätzlich nur *OK* oder *nicht OK* als Ergebnis zurück. Eine spezielle Form der Checks sammelt Metriken (wie beispielsweise aktuelle CUP Last, verbrauchter Speicher, Netzwerkauslastung) über die jeweilige Maschine und wertet diese Anhand von Grenzwerten aus. Wird ein Grenzwert verletzt, so erfolgt eine entsprechende Benachrichtigung. Checks können grundsätzlich in jeglicher Programmiersprache verfasst werden. Die einzigen Voraussetzungen sind, dass die ausführende Maschine eine entsprechende Laufzeitumgebung aufweist und sich der Check an die minimale Spezifikation³¹ hält. Daher können ebenso Nagios Plugins bei Sensu wiederverwendet werden.

Aufbau der Monitoring Infrastruktur: Abbildung 4.7 zeigt den grundsätzlichen Aufbau einer Node der Monitoring Infrastruktur. Sensu nutzt ausschließlich Komponenten die clusterfähig sind und sich gegenseitig synchronisieren. Daher ist es möglich, mehrere Nodes aus Abbildung 4.7 gleichzeitig nebeneinander zu betreiben und somit ein ausfallsicheres System zu garantieren. Außerdem ist die Möglichkeit zur horizontalen Skalierung bei allen last-intensiven Komponenten gegeben.

Folgende Aufgaben erfüllen die einzelnen Komponenten:

³¹<http://sensuapp.org/docs/0.16/checks>, erster Absatz enthält die minimale Spezifikation

- **Keepalived:** Keepalived übernimmt hierbei dieselbe Aufgabe wie beim Loadbalancer (siehe Abschnitt 4.3.3).
- **HAProxy:** HAProxy fungiert als Loadbalancer und verteilt die ankommenden Anfragen auf alle Nodes im Sensu Cluster. Dies sind Zugriffe der Benutzer auf das Dashboard sowie die Kommunikation der Clients mit der Messaging-Queue.
- **Sensu Server/API:** Er verwaltet die Checks und verarbeitet deren Ergebnisse. Alle Instanzen des Servers ermitteln automatisch einen Master um sicherzustellen, dass Aufforderungen für Checks nicht mehrfach gestellt werden. Zusätzlich dazu bietet der Server eine API, über die beispielsweise das Dashboard Daten bezieht.
- **Sensu Dashboard:** Das Dashboard ist die Benutzeroberfläche und zeigt den aktuellen Status aller Clients und ihre letzten Check-Ergebnisse an.
- **RabbitMQ:** Ist eine Messaging-Queue für die Kommunikation zwischen Server und Clients. Alle RabbitMQ Nodes verbinden sich zu einem Cluster und synchronisieren ihre Daten über das Netzwerk hinweg. Es ist somit unerheblich bei welcher Node Daten abgefragt werden. Es wird immer das gleiche Ergebnis zurückgeliefert. Der Sensu Server kann also seine lokale RabbitMQ Node nutzen.
- **Redis³²:** Ist ein Key-Value Store und wird vom Sensu Server genutzt, um Daten zu speichern. Redis persistiert unter anderem die Ergebnisse der Checks über einen gewissen Zeitraum. Folglich ist in Sensu stets eine Historie über die letzten Checks vorhanden. Redis synchronisiert seine Daten ebenfalls über alle Nodes.
- **Redis-Sentinel³³:** Ist eine Redis Node, die in einem speziellen Modus gestartet wird. Sie ist zuständig für die Überwachung des Redis Clusters und erkennt fehlerhafte Nodes, um diese aus dem Cluster zu entfernen. Alle Redis-Sentinel Nodes synchronisieren sich ebenfalls über das Netzwerk untereinander.

Da alle gezeigten Komponenten auf jeder Node installiert sind, müssen nicht mehrere Typen von Nodes unterschieden werden und vereinfacht somit die Verwaltung des Clusters. Dieser Aufbau garantiert ein ausfallsicheres System, da alle Nodes gleichzeitig fehlerhaft sein müssen, um einen kompletten Systemausfall herbeizuführen. Ein weiterer Vorteil von Sensu ist, dass sich die Monitoring Infrastruktur selbst überwachen kann und kein externes System dafür benötigt wird.

Zusätzlich zu den Sensu Nodes existiert eine weitere Maschine, die gesondert behandelt werden muss. Auf dieser Maschine wird Graphite³⁴ und Grafana³⁵ betrieben. Graphite

³²<http://redis.io/topics/cluster-tutorial>, Redis Cluster inkl. Replikation der Daten

³³<http://redis.io/topics/sentinel>, Redis Node im Sentinel Modus

³⁴<http://graphite.wikidot.com/>, leichtgewichtige Datenbank zur Datenaggregation

³⁵<http://docs.grafana.org/>, Javascript Applikation zur Graphenvisualisierung

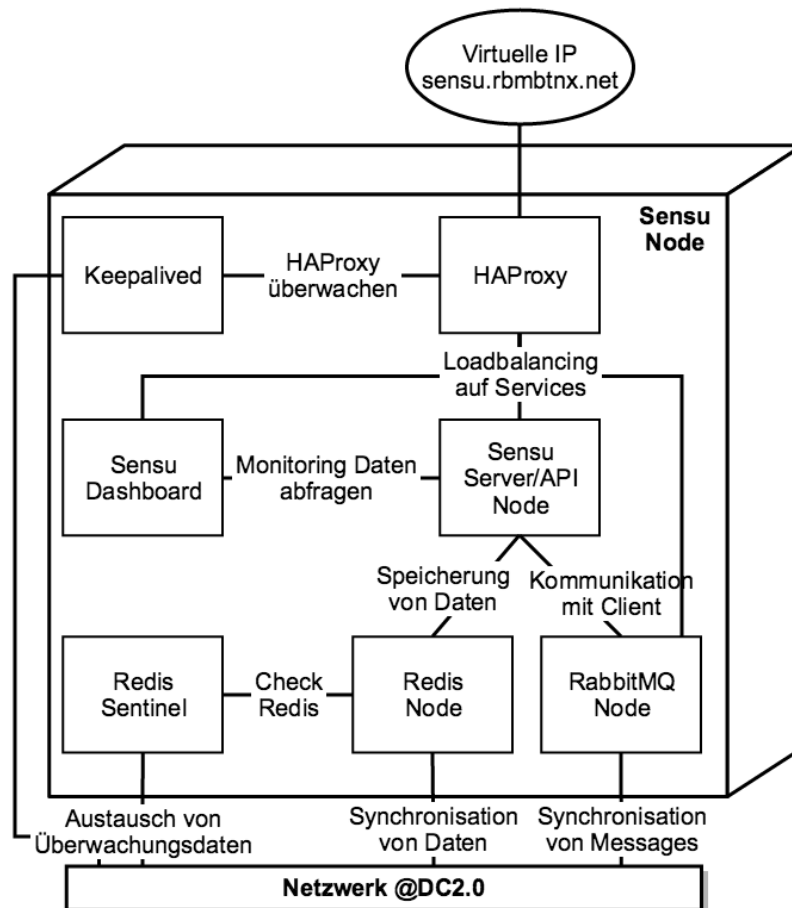


Abbildung 4.7: Grundsätzlicher Aufbau einer Sensu Node in der Monitoring Infrastruktur. Alle Komponenten sind je Node vorhanden und die meisten Komponenten bilden Cluster, um sich gegenseitig zu synchronisieren.

speichert die Daten der Client-Metriken und aggregiert sie über konfigurierbare Zeiträume. Mittels Grafana können anschließend Graphen erstellt werden, die den zeitlichen Verlauf der Metriken darstellen. Somit ist es möglich, detaillierte Reports über beispielsweise die Last eines Systems zu generieren. Anhand dieser Daten können dadurch potentielle Probleme frühzeitig vorausgesagt werden. Da auf dieser Maschine Daten gespeichert werden, muss sie anders behandelt werden als die Sensu Nodes und darf nicht einfach entfernt und neu aufgebaut werden.

Client Konfiguration: Die Konfiguration der Clients³⁶ enthält Basis-Informationen über den Client selbst und die Zugehörigkeit (*subscriptions*) der Maschine zu einer oder mehreren Applikationsgruppen (siehe Listing 4.5). Die Gruppen geben an, welche Checks für die jeweilige Maschine relevant sind und ausgeführt werden müssen.

```
1 {
2   "client": {
```

³⁶<http://sensuapp.org/docs/0.16/clients>, Konfiguration des Sensu Clients

```

3  "name": "prod-da-app02.rbmbtnx.net",
4  "address": "172.20.32.71",
5  "subscriptions": ["common","da-application","webserver"],
6  "rbmb": {
7    "dc": "ffm",
8    "env": "prod",
9    "stack": "delivery-agent",
10   "layer": "application"
11  }
12 }
13 }

```

Auflistung 4.5: Beispielhafte Client Konfiguration des Delivery Agent

Um den Client jederzeit eindeutig im DC2.0 identifizieren und die Daten entsprechend verarbeiten zu können, wurden wie bei Logstash, die vier eigenen Metadaten hinzugefügt. Diese sind in dem Element *rbmb* eingebettet, um keinen Konflikt mit anderen Elementen zu verursachen.

Erfassen der Metriken: Für die Monitoring Infrastruktur müssen neun verschiedene Komponenten automatisiert werden. Allerdings konnten die Roles für HAProxy, Supervisor und Keepalived wiederverwendet werden (Tabelle 4.11). Man erkennt also

Metrik		Menge
Initialer Aufwand	Implementation	150 PS
	wiederverwendete Roles	3 Stk
	je Applikation	0 PS
Komplexität	Anzahl der Komponenten	9
	verteiltes System	Ja
Durchlaufzeit	a.) eine Node	ca. 6 Minuten
	b.) alle Nodes	ca. 25 Minuten
	c.) zusätzliche Node	ca. 8 Minuten

Tabelle 4.11: Erfasste Metriken der Monitoring Infrastruktur

bereits die Tendenz, je mehr Applikationen automatisiert wurden, desto mehr Roles sind wiederverwendbar. Der initiale Aufwand der Implementierung ist dennoch hoch, da die Monitoring Infrastruktur viele Komponenten verwendet, die bisher nicht eingesetzt wurden. Wie bei Logstash ist auch hier die Installation des Clients mit eingerechnet. Der Aufwand pro Applikation ist daher verschwindend gering. Es müssen lediglich Monitoring Checks erstellt werden, sofern die vorhandenen Checks nicht ausreichend sind.

Dieser Aufwand reduziert sich daher bei steigender Anzahl der Applikationen erheblich. Um einen unterbrechungsfreien Monitoring Service zu garantieren, wird bei einem Deployment auf allen Nodes immer nur eine Node zur gleichen Zeit aktualisiert. Aktuell besteht der Sensus Cluster aus vier Nodes: drei Sensus Nodes und einer Node mit Graphite. Daher ist die Metrik b (Deployment auf allen Nodes) auch wie erwartet etwa vier mal so lang wie Metrik a (Deployment auf einer Node). Das Deployment einer zusätzlichen Node ist im Vergleich zu Logstash weniger stark angestiegen. Ein Grund hierfür ist, dass die Komponenten von Sensus geringere Dateigrößen aufweisen und somit der Download schneller abgeschlossen ist. Es ist daher die Tendenz erkennbar, dass je größer die Dateien sind die benötigt werden, desto höher ist der Anstieg der Zeit für ein neues Deployment im Vergleich zu einem Re-Deployment.

4.3.6 Zusammenfassung der Basisinfrastruktur

Zusammenfassend betrachtet fallen für den Aufbau der Basisinfrastruktur Aufwände von ca. 628 Personenstunden an. Sofern von den Systemen und Konzepten aus Abschnitt 4.2 noch nichts bestehendes vorhanden ist, fallen dafür noch einmal zusätzlich 210 Personenstunden an. Insgesamt ergibt sich somit ein Aufwand von 838 Personenstunden, der aufgeteilt auf die drei Personen des Projektteams, eine minimale Durchlaufzeit von knapp sieben Wochen ergibt. Der Aufbau der Basisinfrastruktur kann daher in etwa zwei Monaten abgeschlossen werden und benötigt nur ein Drittel der verfügbaren Zeit. Betrachtet man die Aufwände genauer, ist zu erkennen, dass es sich fast nur um einmalige Aufwände handelt. Die Entwicklung der Konzepte, sowie der Aufbau der Applikationen muss nur einmalig durchgeführt werden. Lediglich bei der Erstellung der virtuellen Maschine und der Integration in den Consul Cluster ist ein Aufwand je Applikation erforderlich. Sofern komplett neue Komponenten betrieben werden, müssen entsprechende Monitoring Checks erstellt werden. Steigt die Anzahl der Komponenten, steigt auch die Anzahl der vorhandenen Checks. Diese können somit wiederverwendet werden, was den wiederkehrenden Aufwand gering hält.

4.4 Implementierung der DevOps Konzepte

Die nachfolgenden Abschnitte befassen sich mit dem zweiten Teil der Fallstudie. Es werden insgesamt vier Applikationen beschrieben, bei denen die Konzepte von DevOps eingeführt wurden. Je Applikation wird zuerst ihr aktueller Aufgabenbereich beschrieben. Anschließend wird auf den Aufbau der Applikation genauer eingegangen, um einen Überblick über dessen Komplexität zu geben. Als Nächstes wird die Implementierung

beschrieben. Hierbei wird im Besonderen auf neue Bestandteile eingegangen und welche Teile von anderen Applikationen wiederverwendet wurden. Abschließend werden die Metriken erfasst und erklärt.

4.4.1 Applikation - Delivery Agent

Ein wiederkehrender Anwendungsfall bei vielen Applikationen, ist die Auslieferung von Multi-Media-Ressourcen an die Benutzer. Vorschaubilder oder Proxies für Videodateien, wie im Outlet³⁷ der Red Bull Media Base, sind Ressourcen die häufig benötigt werden. Damit nicht jede Applikation diese Funktionalität bereitstellen muss, wurde sie in eine eigene Applikation namens Delivery Agent ausgegliedert. Dieser ist dafür zuständig, Ressourcen durch unterschiedliche Abgriffsmethoden (Progressive Download, Streaming, etc.) bereitzustellen. Applikationen die Ressourcen benötigen, können diese über Links einbinden und somit vom Delivery Agent ausliefern lassen. Durch den begrenzten Aufgabenbereich des Delivery Agents, ist dieser Teil der weniger komplexen Applikationen, die im Zuge der Fallstudie betrachtet werden. Allerdings gelten für ihn spezielle Anforderungen an den Betrieb. Da er einen der häufigsten Anwendungsfälle bedient, ist die Last auf diesem System entsprechend hoch. Der Delivery Agent muss mit dieser hohen Last umgehen können. Außerdem stellt er eine grundlegende Funktionalität bereit und muss deswegen entsprechend ausfallsicher sein. Ansonsten wäre die Funktionalität der abhängigen Applikationen beeinträchtigt.

Aufbau der Applikation: Um die, im vorherigen Absatz beschriebenen, Anforderungen erfüllen zu können, muss der Delivery Agent als verteiltes System aufgebaut sein. Der Delivery Agent ist eine zustandslose Applikation und erleichtert somit die Verteilung des Systems, da keine Daten zwischen den unterschiedlichen Nodes synchronisiert werden müssen. Die Abbildung 4.8 zeigt den grundsätzlichen Aufbau einer Delivery Agent Node. Durch die zustandslose Arbeitsweise der Komponenten, können beliebig viele Nodes parallel betrieben werden und es ist keine Clusterbildung notwendig. Durch diese Möglichkeit zur horizontalen Skalierung, lassen sich zukunftsicher, steigende Lastanforderungen bewältigen. Folgende Aufgaben führen die einzelnen Komponenten aus:

- **Delivery Agent:** Ist eine Spring-Boot-Applikation³⁸ die für die Verwaltung und Auslieferung der Ressourcen zuständig ist. Sie fordert die Lage der Ressourcen von einer zentralen Instanz an und stellt diese anschließend bereit.

³⁷www.redbullcontentpool.com

³⁸<http://projects.spring.io/spring-boot/>, Framework für Standalone Applikation mit Java

- **nginx:** Der nginx Webserver übernimmt das Caching von häufig angefragten Ressourcen, um diese schneller ausliefern zu können. Ist eine Ressource nicht im Cache vorhanden, so wird die Anfrage an die Delivery Agent Komponente weitergeleitet.
- **Consul:** Um Ausfallsicherheit und optimale Lastverteilung zu erreichen, ist der Delivery Agent an den globalen Loadbalancer des DC2.0 angeschlossen. Wie in Abschnitt 4.3.3 beschrieben, ist hierfür der Consul Client auf der Node notwendig.
- **Sensu-Client:** Damit die Node entsprechend vom Monitoring System überwacht wird, ist der Sensu-Client erforderlich. Dieser überwacht die Komponenten und führt Checks durch, ob diese noch korrekt funktionieren.
- **Log-Shipper:** Die Log-Dateien der einzelnen Komponenten werden an das zentrale Logging System aus Abschnitt 4.3.4 geschickt. Diese Aufgabe wird vom Log-Shipper (in diesem Fall logback) übernommen.

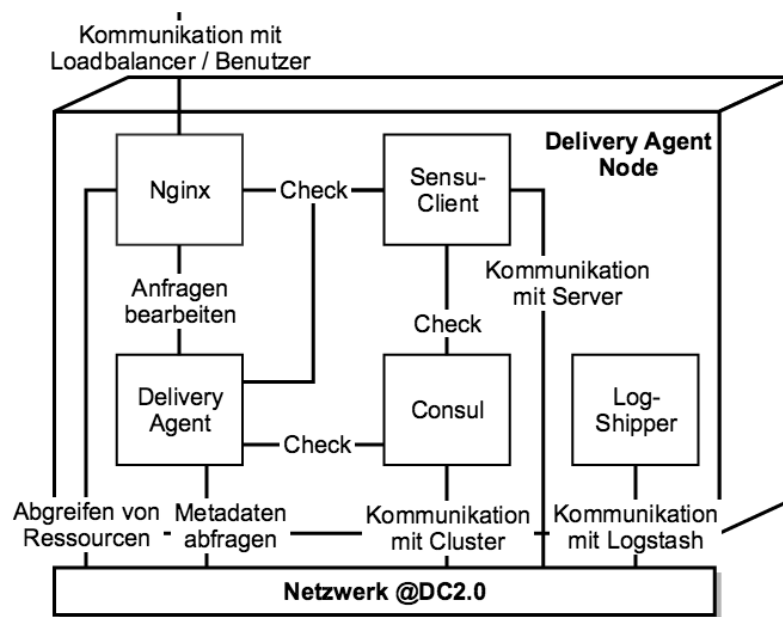


Abbildung 4.8: Grundsätzlicher Aufbau einer Delivery Agent Node. Alle Komponenten kommen pro Node vor. Die Nodes sind unabhängig voneinander und teilen keine Daten.

Zusätzlich zu den oben beschriebenen Komponenten, sind auch noch die Java Runtime Umgebung sowie Supervisor für die Verwaltung der Komponenten installiert.

Automatisierung des Deployments: Da der Delivery Agent den globalen Loadbalancer nutzt, muss kein dedizierter Loadbalancer aufgebaut werden. Durch die Implementierung der Basisinfrastruktur sind bereits Roles für die Komponenten Java, nginx, Supervisor, Sensu-Client, Log-Shipper und Consul vorhanden. Hierbei ergibt

sich bereits ein hohes Maß an Wiederverwendbarkeit. Bei der Automatisierung des Deployments reduziert sich deswegen der Aufwand auf ein Minimum, da nur die Installation der Delivery Agent Komponente und die Konfiguration zweier Komponenten betrachtet werden muss. Die Roles für den nginx und Supervisor sind sehr allgemein gehalten, um sie wiederverwenden zu können. Aus diesem Grund müssen beide für den konkreten Anwendungsfall des Delivery Agent noch speziell konfiguriert werden.

Zusätzlich zur Automatisierung des Deployments, müssen für den Delivery Agent noch Formation Roles (Abschnitt 4.2.5) für die virtuellen Maschinen, sowie entsprechende Build und Deploymentpläne in Bamboo (Abschnitt 4.2.2) angelegt werden. Nachdem diese Tätigkeiten beendet wurden, konnte der Delivery Agent vollständig automatisiert deployed werden. Es sind keine manuellen Schritte mehr notwendig.

Erfassen der Metriken: Der initiale Aufwand der Implementierung für den Delivery Agent fällt sehr gering aus, da nur eine einzige weitere Komponente automatisiert werden musste. Auch die Verteilungsaspekte des Delivery Agent haben den Aufwand nicht erhöht. Der Delivery Agent wurde bereits im DC1.0 betrieben. Aus diesem Grund wird ein Vergleich der Deploymentzeiten vor und nach der Automatisierung durchgeführt. Wie in Tabelle 4.12 ersichtlich, betrug die Zeit vor der Automatisierung, ca. 20

Metrik		Menge
Initialer Aufwand	Implementation	80 PS
	wiederverwendete Roles	6
Komplexität	Anzahl der Komponenten	7
	verteiltes System	Ja
Durchlaufzeit: Vorher	a.) eine Node	ca. 20 Minuten
	b.) alle Nodes	ca. 40 Minuten
	c.) zusätzliche Node	ca. 90 Minuten
Durchlaufzeit: Nacher	a.) eine Node	ca. 7 Minuten
	b.) alle Nodes	ca. 15 Minuten
	c.) zusätzliche Node	ca. 10 Minuten

Tabelle 4.12: Erfasste Metriken des Delivery Agents

Minuten für ein Deployment und ca. 1.5 Stunden für das Deployment einer zusätzlichen Node. Nach der Automatisierung benötigt ein Deployment ca. 7 Minuten und das Deployment einer zusätzlichen Node ca. 10 Minuten. Die Provisionierung einer Node ist daher fast dreimal bzw. neunmal so schnell als zuvor. Außerdem waren vor der Auto-

omatisierung die Mitarbeiter die komplette Zeit beschäftigt und somit nicht verfügbar. Nach der Automatisierung ist auch während eines Deployments keine menschliche Ressource belegt und kann somit anderen Tätigkeiten nachgehen. Der Zeitgewinn bei der Verfügbarkeit der Mitarbeiter ist somit enorm.

Im aktuellen Setup sind zwei Delivery Agent Nodes vorhanden. Im manuellen Fall muss eine Node nach der anderen aktualisiert werden. Die Zeit für die Belegung der Mitarbeiter multipliziert sich also mit der Anzahl der Nodes. Im automatisierten Fall können alle Nodes, bis auf Eine³⁹, zur gleichen Zeit aktualisiert werden. Somit beträgt die insgesamt benötigte Zeit, nur die doppelte Zeit einer Node. Der Zeitgewinn beträgt hierbei bereits 26 Minuten (40 gegenüber 14 Minuten). Der Zeitgewinn bei einem Deployment ist daher umso größer, je mehr Nodes vorhanden sind. Es können zwar mehrere Mitarbeiter gleichzeitig arbeiten, dies belegt aber auch entsprechend viele Ressourcen und hat höhere Kosten zur Folge.

4.4.2 Applikation - OAuth2 Server (OIDC)

Nahezu jedes Unternehmen benötigt Applikationen für die Authentifizierung und Autorisierung von Benutzern. Häufig wird dafür ein LDAP System, wie zum Beispiel OpenLDAP, eingesetzt. Nutzen nur interne Applikationen diese Informationen, sind die Sicherheitsanforderungen an diese Systeme gering. Sobald auch andere, externe Applikationen diese Infrastruktur nutzen möchten, müssen umfassende Sicherheitsmaßnahmen ergriffen werden. Bei LDAP besteht allerdings das Problem, dass die Applikation beim Login die Passwörter der Benutzer im Klartext vernehmen und diese unter bösem Vorwand speichern kann. Um dieses Problem zu umgehen, gibt es sichere Alternativen wie OAuth2⁴⁰. Dieser Standard beschreibt eine Möglichkeit, wie sich Benutzer an einem sicheren, zentralen Server anmelden können und dann mittels Tokens bei der jeweiligen Applikation authentifiziert und autorisiert werden. Die nutzende Applikation erhält somit nicht mehr das Passwort im Klartext.

Die Red Bull Media Base betreibt bisher ein eigenes LDAP, das auch von Applikationen anderer Abteilungen genutzt wird. Aus den oben genannten Gründen hat man sich entschieden, das LDAP durch OpenID Connect (OIDC)⁴¹ abzulösen. OpenID Connect ist ein Standard, der auf OAuth2 basiert und diesen um SingleSignOn Funktionalitäten erweitert. Da dieser Service einen zentralen Bestandteil darstellt und viele Applikationen davon abhängen, muss hier besonders auf Ausfallsicherheit geachtet werden.

³⁹Um einen unterbrechungsfreien Service zu garantieren.

⁴⁰<http://tools.ietf.org/html/rfc6749>, Standard für sichere Authentifizierung mittels Tokens.

⁴¹http://openid.net/specs/openid-connect-core-1_0.html, Spezifikation von OIDC

Aufbau der Applikation: Da der OIDC Server im AUTH Subnetz stationiert ist, muss neben dem eigentlichen Service auch ein eigener Loadbalancer und Consul Cluster aufgebaut werden. Die Installation dieser beiden Applikationen ist bereits automatisiert und somit fällt nur minimaler Mehraufwand an. Damit Ausfallsicherheit des Service erreicht wird, muss der OIDC Server als Cluster mit mehreren Nodes betrieben werden. Da jede Node selbst Daten erzeugt, müssen diese synchronisiert werden. Folgende, in Abbildung 4.9 dargestellten Komponenten kommen je Node vor:

- **Connect2ID**⁴²: Der Connect2ID Server implementiert den OIDC Standard und stellt die grundlegenden Funktionalitäten für die Nutzung von OIDC bereit. Er ist die Hauptkomponente der Node, die von anderen Komponenten genutzt wird.
- **OpenLDAP**⁴³: Ist die Datenbank der Node und enthält Informationen über die Benutzer, die vom Connect2ID Server abgerufen werden, um die Berechtigungen der Benutzer zu ermitteln. Außerdem wird die Gültigkeit von Tokens gespeichert.
- **OIDC Login Service**: Diese Komponente ist eine eigens entwickelte Applikation und bietet die Login Maske für die Benutzer, um sich beim OIDC Server zu authentifizieren. Applikationen leiten die Benutzer hierauf weiter, um die Authentifizierung durchzuführen und anschließend Tokens zu erhalten.
- **Password Grant Handler**: Diese Komponente implementiert den Password Grant Flow des OAuth2 Standards. Dieser Grant ist im OIDC Standard nicht enthalten, da hierbei dasselbe Problem mit den Passwörtern im Klartext besteht. Um bestehende Anbindungen von Applikationen noch zu unterstützen, wird dieser Grant temporär angeboten.
- **Consul**: Für den eigenen Consul Cluster im AUTH Subnetz ist ebenfalls der Client notwendig. Er wird ebenso für die Integration in den Loadbalancer genutzt.
- **Sensu-Client / Log-Shipper**: Selbe Funktion wie beim Delivery Agent aus Abschnitt 4.4.1.

Zusätzlich zu den oben beschriebenen Komponenten sind auch noch die Java Runtime Umgebung, sowie Supervisor für die Verwaltung der Komponenten installiert. Der Connect2ID Server, OIDC Login Service und Password Grant Handler nutzen außerdem einen Tomcat Webserver. Dieser wird von den drei Applikationen geteilt.

Bei der Clusterbildung stellte die Synchronisation der Daten das schwerwiegendste Problem dar. Da OpenLDAP als Datenbank genutzt wird, müssen dessen Daten synchronisiert werden. Allerdings ist OpenLDAP nicht für einen verteilten Betrieb ausgelegt. Es bietet zwar klassische Master-Slave Szenarien an, wobei diese hier nicht praktikabel

⁴²<http://connect2id.com/products/server>, kommerzielles Produkt zur Nutzung von OIDC

⁴³<http://www.openldap.org/project/>, Open Source LDAP Implementierung

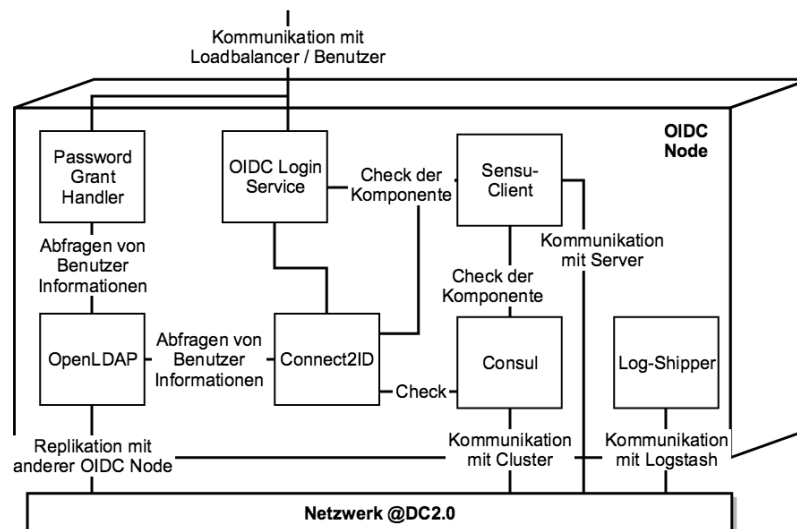


Abbildung 4.9: Grundsätzlicher Aufbau einer OIDC Node. Alle Komponenten kommen pro Node vor. Alle Nodes verbinden sich zu einem Cluster, um Daten im OpenLDAP zu synchronisieren.

sind, da Daten auf jeder Node des OIDC Cluster geschrieben werden. Bei Master-Slave Szenarien sind Schreiboperationen nur auf einer Node erlaubt. OpenLDAP bietet nur eine Konfiguration, bei der auf mehreren Nodes gleichzeitig geschrieben werden kann, den sogenannten Mirror-Mode. Dieser Modus unterstützt aber maximal zwei Nodes, weswegen der OIDC Cluster auf diese Zahl beschränkt ist. Dies erfüllt die grundlegende Anforderung der Ausfallsicherheit, ist aber kein optimales Setup, da nicht weiter skaliert werden kann. Auf Grund fehlender Alternativen wurde dieses Setup aber umgesetzt.

Automatisierung des Deployments: Durch die hohe Anzahl an Komponenten und die Einschränkungen von OpenLDAP, ist der OIDC Server die komplexeste Applikation, die in der Fallstudie betrachtet wird. Auf Grund bisheriger Arbeiten sind bereits Roles für die Komponenten Java, Supervisor, Consul, Sensu-Client und den Log-Shipper vorhanden. Bei der Implementierung müssen noch fünf weitere Komponenten betrachtet werden. Der Tomcat, die beiden selbst entwickelten Applikationen, sowie der Connect2ID Server stellen hierbei kein Problem dar, da sie mit aktuellen Technologien entwickelt wurden. Deren Automatisierung konnte deshalb in angemessener Zeit abgeschlossen werden. Ein Problem stellt die Automatisierung von OpenLDAP dar. OpenLDAP wurde 1998 gestartet und damals wurden noch keine Konzepte für Automatisierung und für einen verteilten Betrieb berücksichtigt. Auch neuere Versionen (2.4) unterstützen diese Konzepte noch nicht vollständig. Vor allem die Konfiguration der OpenLDAP Instanzen ist problematisch. Ein Großteil der Konfiguration ist in der Datenbank gespeichert. Es unterstützt jedoch keine idempotenten Prozesse, weswegen

bereits vorhandene Konfigurationen Fehler erzeugen, sofern sie erneut importiert werden. Dies muss bei der Automatisierung speziell berücksichtigt werden. Manche Teile der Konfiguration müssen außerdem auf beiden Nodes importiert werden, andere Teile der Konfiguration nur auf einer Node, da diese anschließend auf die weitere Node synchronisiert wird. Weiters dürfen nicht beide Nodes gleichzeitig aktualisiert werden, da andernfalls die Replikation fehlerhaft sein kann. Deswegen ist es vergleichsweise Komplex, die Automatisierung von OpenLDAP umzusetzen.

Wie für alle Applikationen, müssen zusätzlich noch Formation Roles und Deploymentpläne angelegt werden. Dies betrifft hier auch den Consul Cluster und den Loadbalancer, die extra für das AUTH Subnetz aufgebaut werden. Nach Beendigung dieser Arbeit konnte der OIDC Cluster vollständig automatisiert deployed werden.

Erfassen der Metriken: OIDC wurde bisher im DC1.0 nicht eingesetzt, weswegen ein Vergleich der Zeiten nicht möglich ist. Die Anzahl der Komponenten und Roles beziehen sich hierbei rein auf den OIDC Server. Der Consul Cluster und der Loadbalancer werden hierbei nicht berücksichtigt. Deren Metriken können aus dem jeweiligen Abschnitt (4.3.2 und 4.3.3) entnommen werden. Durch die hohe Anzahl an Kompo-

Metrik		Menge
Initialer Aufwand	Implementation	300 PS
	wiederverwendete Roles	4
Komplexität	Anzahl der Komponenten	10
	verteiltes System	Ja
Durchlaufzeit	a.) eine Node	ca. 12 Minuten
	b.) alle Nodes	ca. 28 Minuten
	c.) zusätzliche Node	ca. 20 Minuten

Tabelle 4.13: Erfasste Metriken des OIDC Servers

ponenten, für die keine Automatisierung vorhanden war, fällt der Aufwand entsprechend höher aus. Es ist bereits die Tendenz erkennbar, dass je homogener die Applikationslandschaft eines Unternehmens ist, desto geringer ist der Aufwand für weitere Applikationen. Ein weiterer Grund für den hohen Aufwand bestand darin, dass im Projektteam bisher wenig Erfahrung mit OpenLDAP vorhanden war. Im aktuellen Fall ist es das erste Mal, dass die Verteilung der Applikation direkte Auswirkungen auf den Aufwand hat. Der Grund hierfür ist OpenLDAP, da es diese Konzepte kaum unterstützt.

Da für einen unterbrechungsfreien Service ohnehin nur eine Node zur gleichen Zeit aktualisiert werden kann, ist diese Einschränkung von OpenLDAP vernachlässigbar.

Ein Deployment des gesamten Clusters benötigt ca. 28 Minuten. Auffallend ist der große Unterschied zwischen einem Deployment (a) und dem Deployment einer zusätzlichen Node (c). Dies ist einerseits zurückzuführen auf den Import des Schemas in OpenLDAP, der hier durchgeführt wird und andererseits dem Download der vielen Komponenten.

4.4.3 Applikation - Content Processing Agent (CPAS)

Zusätzlich zur Zentrale des Red Bull Media House in Salzburg, gibt es Außenstellen die weltweit verteilt sind. Diese befinden sich in London, Los Angeles, Sydney, Sao Paulo, Frankfurt und Wien. Häufig müssen Multimedia Dateien zwischen diesen Standorten ausgetauscht werden. Das Transcodieren der Dateien ist ebenfalls eine häufig anfallende Aufgabe. Um diese Prozesse zu vereinheitlichen und auch direkt in einer Außenstelle durchführen zu können, wurde der Content Processing Agent (kurz CPAS) entwickelt. Der CPAS ist eine Applikation, die alle nötigen Werkzeuge beinhaltet, um Multimedia Dateien zu verarbeiten. Er steht in Verbindung mit der zentralen Asset-Management-Plattform und erhält von dort Aufgaben, die er durchführt. So können in einer Außenstelle ebenfalls Transcodings bearbeitet werden, und die Dateien müssen nicht zuerst zum zentralen Transcodingcluster transferiert werden. Diese Vorgehensweise spart erheblich Zeit. Außerdem können die CPAS in unterschiedlichen Standorten, Dateien über ein optimiertes Protokoll transferieren. Dieses Protokoll ermöglicht eine bessere Auslastung der verfügbaren Bandbreite.

Der CPAS kann als zustandslose Applikation betrachtet werden, da er keine Daten mit anderen CPAS Instanzen teilt. Er speichert immer nur die Daten, die für die Verarbeitung seiner aktuellen Aufgaben nötig sind. Die Verwaltung der Aufgaben aller CPAS Instanzen übernimmt die zentrale Asset-Management-Plattform. Ein CPAS kennt eine andere Instanz nur dann, wenn er einen Transfer dorthin durchführt. Es ist also möglich mehrere CPAS Instanzen nebeneinander zu betreiben, ohne dass diese miteinander synchronisiert werden müssen.

Aufbau der Applikation: Die zentrale Plattform kennt alle CPAS Instanzen und verteilt die Aufgaben gleichmäßig. Daher muss beim CPAS nicht auf Ausfallsicherheit geachtet werden und es ist kein Loadbalancing notwendig. Auch wird in den Außenstellen meist nur eine CPAS Instanz betrieben, da diese Kapazitäten ausreichen. Einzig in den Rechenzentren in Salzburg und Frankfurt werden mehrere CPAS Instanzen bereitgestellt, um höhere Kapazitäten beim Transcoding zur Verfügung zu haben. Nahezu alle Komponenten am CPAS sind für die Verarbeitung von Multimedia Dateien ausgelegt. Die einzelnen Komponenten, wie in Abbildung 4.10 dargestellt, sind:

- **CPAS Applikation:** Diese Komponente wurde selbst entwickelt und nutzt die anderen Komponenten zur Verarbeitung von Dateien. Sie kommuniziert mit der zentralen Plattform und erhält von dort ihre Aufgaben. Diese werden an die zuständigen Komponenten weiter verteilt. Gleichzeitig regelt diese Komponente den Zugriff auf den Speicher, um Dateien und berechnete Derivate zu lesen bzw. zu schreiben.
- **Transcoder (ffmpeg)⁴⁴:** Der CPAS setzt ffmpeg als Transcoder ein. Über ffmpeg wird die Berechnung von statischen Derivaten durchgeführt, wie beispielsweise Proxies für die Voransicht eines Videos oder die Berechnung eines gewünschten Bestellformats für Kunden.
- **MP4Box⁴⁵:** Wird für die Aufbereitung von Videos genutzt, um diese per Stream (HTTP Adaptive Streaming) zur Verfügung stellen zu können.
- **Media Processing Service (MPS)⁴⁶:** Ist ein eigener Service, der auf dem CPAS läuft, um aus Video-Dateien Subclips zu erzeugen. Dieser Vorgang spart in der Post-Produktion Zeit, da dort oftmals nur kleine Teile eines Videos benötigt werden.
- **UDT Agent:** Ist eine eigens entwickelte Applikation, die für Transfers zwischen den CPAS Instanzen zuständig ist. Sofern an der Quelle und am Ziel ein UDT Agent vorhanden ist, können die Dateien mit dem UDT Protokoll⁴⁷ optimiert transferiert werden.
- **Sensu-Client / Log-Shipper:** Selbe Funktion wie beim Delivery Agent aus Abschnitt 4.4.1.

Zusätzlich zu den oben beschriebenen Komponenten, sind noch die Java Runtime Umgebung in zweifacher Ausführung (Version 7 für CPAS und Version 8 für MPS), sowie Supervisor für die Verwaltung der Komponenten installiert. Der UDT Agent benötigt außerdem einen Tomcat Webserver als Laufzeitumgebung.

Automatisierung des Deployments: Bei der Automatisierung des Deployments muss beim CPAS, vor allem auf die Unterschiede zwischen den einzelnen Standorten geachtet werden. So müssen an jedem Standort andere Speichersysteme eingebunden werden. Dies wurde hauptsächlich über Ansible Variablen abgebildet, die abhängig vom jeweiligen Standort des CPAS geladen werden. So kann derselbe Code für das Mounten

⁴⁴<https://www.ffmpeg.org/about.html>, weit verbreiteter Open Source Transcoder

⁴⁵<http://gpac.wp.mines-telecom.fr/mp4box/>, Open Source Multimedia Paketierungssystem

⁴⁶<http://www.lesspain-software.com/products/>, Transcoding Service auf Java Basis

⁴⁷<http://udt.sourceforge.net/>, Protokoll auf UDP Basis mit eigener Transfersicherheit

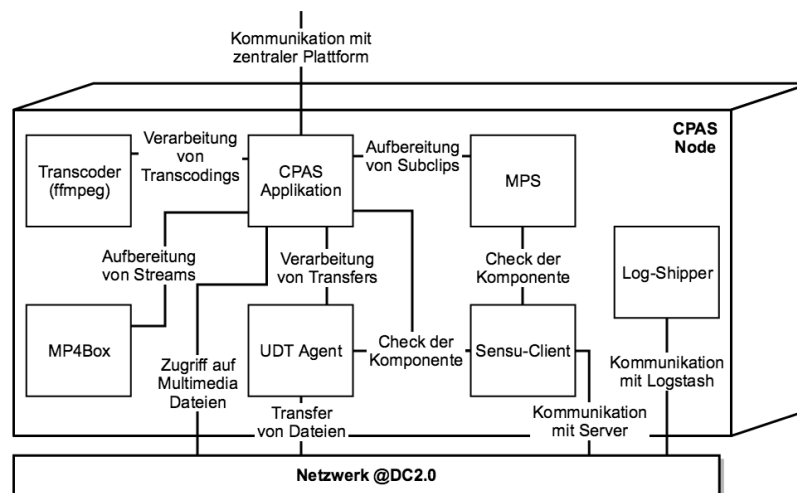


Abbildung 4.10: Grundsätzlicher Aufbau einer CPAS Node. Alle Komponenten kommen pro Node vor und agieren unabhängig von anderen CPAS Instanzen.

der Speichersysteme verwendet werden. Ansonsten gibt es keine Unterschiede zwischen den Standorten, die beim Deployment speziell berücksichtigt werden müssen.

Für die Komponenten Java, Tomcat, Supervisor, Sensu-Client und Log-Shipper konnten Roles wiederverwendet werden. Da die CPAS Applikation und MPS vergleichsweise einfache Services sind, konnte das Deployment ohne erheblichen Aufwand automatisiert werden. Gleiches gilt für den UDT Agent, da dieser nur ein WAR Archiv ist, das in den Tomcat integriert wird. Der größte Teil des Aufwands der Implementierung floss in die Automatisierung von ffmpeg und MP4Box. Der CPAS nutzt, wie alle Maschinen, CentOS 7 als Basis. Für CentOS 7 werden allerdings keine fertigen Pakete für diese beiden Komponenten angeboten. Das Deployment dieser Komponenten beschränkt sich also nicht auf eine einfache Installation, sondern sie müssen für unsere Plattform eigens kompiliert werden. Zuerst wird das System entsprechend vorbereitet, damit alle nötigen Werkzeuge für die Kompilierung vorhanden sind. Anschließend werden die beiden Komponenten erstellt. Um Speicherplatz nicht unnötig zu belegen, kann man die Werkzeuge anschließend wieder entfernen. Die implementierte Automatisierung stellt außerdem sicher, dass dieser Prozess nur durchgeführt wird, wenn diese Komponenten noch nicht vorhanden sind, um die Durchlaufzeit zu verringern.

Wie für alle Applikationen fällt noch die Erstellung der Pläne in Bamboo an. Nach Beenden dieser Tätigkeiten, konnte der CPAS vollständig automatisiert deployed werden. Dies betrifft allerdings nur die Installation des CPAS. Aktuell muss eine neue Instanz anschließend der zentralen Plattform manuell bekannt gegeben werden. Um dieses Problem dauerhaft zu lösen, steht noch einmalige Entwicklungsarbeit an.

Erfassen der Metriken: Der CPAS wird bereits seit sieben Jahren eingesetzt und deshalb ist ein Vergleich von vorher zu nachher möglich. Die Tabelle 4.14 zeigt die unterschiedlichen Deploymentzeiten. Der größte Unterschied ergibt sich bei einem Deployment einer neuen Version, auf allen vorhandenen CPAS. Durch die große Anzahl an Nodes (13 Stück⁴⁸), die weltweit verteilt sind, multipliziert sich die Zeit bei manueller Durchführung erheblich. Bei einem Deployment mussten mehrere Mitarbeiter gleichzeitig arbeiten, um die Aktualisierung in angemessener Zeit abzuschließen. Hier ergibt sich ein Zeitgewinn, nach der Implementierung der Automatisierung, von über 2000% und somit eine erhebliche Steigerung der Effizienz.

Da jede CPAS Instanz unabhängig fungiert und dabei kein Loadbalancing benötigt, wird der CPAS nicht als verteilte Applikation angesehen. Die komplette Logik, die für die Verteilung der Aufgaben auf alle CPAS benötigt wird, ist in der zentralen Plattform gekapselt. Der CPAS selbst hat keine Kenntnis über seine Verteilung.

Metrik		Menge
Initialer Aufwand	Implementation	180 PS
	wiederverwendete Roles	5
Komplexität	Anzahl der Komponenten	10
	verteiltes System	Nein
Durchlaufzeit: Vorher	a.) eine Node	ca. 25 Minuten
	b.) alle Nodes	ca. 320 Minuten
	c.) zusätzliche Node	ca. 180 Minuten
Durchlaufzeit: Nacher	a.) eine Node	ca. 5-15 Minuten
	b.) alle Nodes	ca. 15 Minuten
	c.) zusätzliche Node	ca. 30-60 Minuten

Tabelle 4.14: Erfasste Metriken des CPAS

Die hohen Intervalle bei den Deploymentzeiten, ergeben sich aus den Standorten der CPAS. Ein CPAS in Frankfurt ist schneller aktualisiert, da die Transferzeiten der Komponenten wesentlich kürzer sind, als für einen CPAS in Sydney. Die untere Grenze des Intervalls gibt somit die Zeit für einen CPAS in Europa an. Die obere Grenze gilt für CPAS Instanzen, die weiter entfernt sind.

Der ebenfalls erhebliche Anstieg der Zeit, für ein Deployment einer neuen Instanz, im Vergleich zu einem Re-Deployment, lässt sich mit der Kompilierung von ffmpeg und MP4Box erklären. Bei einem Re-Deployment sind diese Komponenten bereits vor-

⁴⁸London/Los Angeles/Sydney/Sao Paulo/Wien je ein Stück und Frankfurt/Salzburg je vier Stück

handen, weswegen die Kompilierung entfällt. Dies reduziert die Durchlaufzeit erheblich.

4.4.4 Applikation - Outlet Service (APEX Framework)

Ein frequenter Anwendungsfall im Bereich der Red Bull Media Base ist die Distribution von Multimedia Dateien an Kunden. Das Problem hierbei ist, dass die Anforderungen an die Distributionsmöglichkeiten je Stakeholder im Red Bull Media House sehr divergieren. So werden unterschiedliche Funktionen und auch ein entsprechend individuelles Erscheinungsbild benötigt. Es müsste demnach eine individuelle Lösung pro Stakeholder verwaltet werden, was auf Grund der begrenzten Ressourcen nicht möglich ist. Die einzelnen Stakeholder müssen daher selbst für die Entwicklung eines eigenen Outlets sorgen. Dieses muss mit den Schnittstellen der zentralen Asset-Management-Plattform kompatibel sein, um Daten zu erhalten. Damit die Kompatibilität gegeben ist und die individuellen Lösungen auch im DC2.0 betrieben werden können, wurde das APEX Framework entwickelt. Das APEX Framework ist eine Basis, die von den Stakeholdern genutzt werden kann, um ihre individuelle Lösung zu entwickeln. Somit ist sichergestellt, dass grundlegende Schnittstellen unterstützt sind und die Applikationen den Prinzipien des DC2.0 (Abschnitt 4.2.4) entspricht. Zusätzlich wird die individuelle Applikation in Bamboo gebaut und über die Infrastruktur der Red Bull Media Base verwaltet. Der Stakeholder muss sich demnach nur um die Entwicklung kümmern und hat dort nahezu alle Freiheiten. Dies bedeutet aber auch, dass die individuellen Lösungen nicht zwingend skalierbar sind. Auf diesen Umstand muss der Stakeholder bei der Entwicklung achten.

Aufbau der Applikation: Da der Stakeholder viele Freiheiten mit dem APEX Framework hat, können auch zusätzliche Komponenten verwendet werden, die das Framework nicht anbietet. Im weiteren Verlauf wird nur der minimale Aufbau des APEX Frameworks betrachtet. Alle weiteren Komponenten entsprechen individuellen Anforderungen und müssen je Stakeholder betrachtet werden.

Das APEX Framework besteht grundsätzlich aus der Basisapplikation von APEX und zusätzlichen Bibliotheken. Die Basisapplikation basiert auf Vert.x⁴⁹ und wird zwingend benötigt. Sie stellt die Laufzeitumgebung für den individuellen Code der Stakeholder dar. Außerdem garantiert die Basisapplikation die Voraussetzungen für das DC2.0 und bietet unter anderem die benötigten Monitoring Endpunkte. Damit ist ein gewisses Maß an Kontrolle über den individuellen Code gegeben und der Betrieb kann gewährleistet werden. Die zusätzlichen Bibliotheken sind optional und kapseln häufig wiederkehrende

⁴⁹<http://vertx.io/manual.html>, Applikationsplattform für die JVM

Anwendungsfälle. Sie können je nach Bedarf in die Basisapplikation von APEX geladen werden. Aktuell gibt es zwei Bibliotheken:

- **Content Library:** Diese Bibliothek kapselt ein Outlet, wie es von der Red Bull Media Base bereitgestellt wird. Es stellt alle grundlegenden Funktionalitäten, für die Auslieferung von Multimedia Dateien, bereit. Der Stakeholder muss sich in diesem Fall nur mehr um die Benutzeroberfläche kümmern, hat dadurch aber nicht mehr alle Freiheiten bei den Funktionalitäten.
- **Cortex Library:** Cortex ist die zentrale Reporting Infrastruktur der Red Bull Media Base. Diese Bibliothek kapselt alle Funktionalitäten, um diese Reporting Infrastruktur zu nutzen. Der Stakeholder muss sich nicht um die Speicherung und Auswertung von Reporting Daten kümmern.

Ein weiterer häufiger Anwendungsfall ist, das Durchsuchen der vorhandenen Dateien. Dafür wird, zusätzlich zu den Komponenten des APEX Frameworks, noch eine Apache Solr⁵⁰ Instanz vorausgesetzt. Diese ermöglicht es, die Metadaten der vorhandenen Dateien zu speichern und hochperformant zu durchsuchen. Den Benutzern wird somit eine schnelle und flexible Suche ermöglicht. Wie bei den bisher beschriebenen Appli-

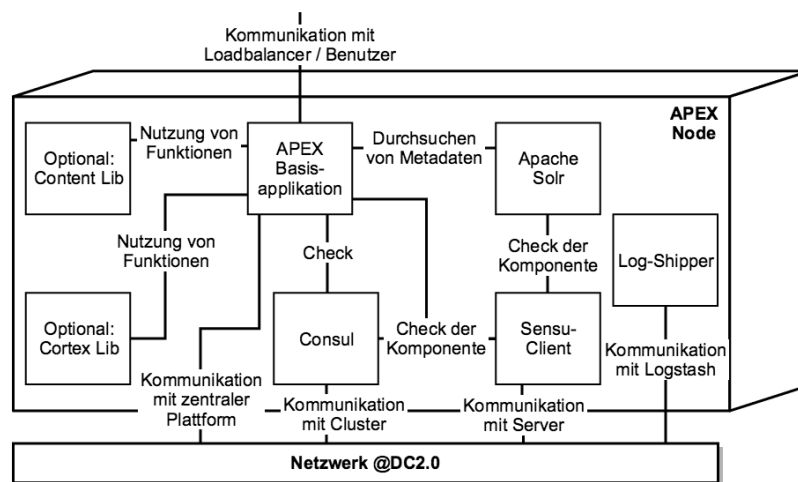


Abbildung 4.11: Minimaler Aufbau einer APEX Node inklusive aller aktuell verfügbaren optionalen Bibliotheken

kationen, sind außerdem noch Consul für die Kommunikation mit dem Loadbalancer und der Sensu-Client/Log-Shipper für die Integration in die Monitoring/Logging Infrastruktur vorhanden. Alle grundlegenden Komponenten des minimalen Aufbaus und ihre Abhängigkeiten untereinander, sind in Abbildung 4.11 zu sehen. Zusätzlich zu den dargestellten Komponenten sind auch noch Java als Laufzeitumgebung und Supervisor für die Verwaltung der Komponenten installiert.

⁵⁰<http://lucene.apache.org/solr/>, hochperformante Suchmaschine von Apache

Automatisierung des Deployments: Bei der Automatisierung des Deployments muss nur APEX selbst, sowie der Apache Solr betrachtet werden. Für alle anderen Komponenten sind bereits Roles vorhanden, die wiederverwendet werden können. Die Automatisierung des Apache Solr stellte sich als unproblematisch heraus, da keine Abhängigkeiten bestehen und auch nicht auf Verteilungseigenschaften Rücksicht genommen werden musste. Eine größere Herausforderung stellt die Automatisierung von APEX selbst dar, da hierbei eine große Anzahl von Freiheitsgraden vorhanden ist. Außerdem kann nicht vorausgesagt werden, welche Komponenten die Stakeholder benötigen. Daher wurde das Deployment von APEX so generisch wie möglich gestaltet, um zukünftige Anforderungen der Stakeholder leicht unterstützen zu können.

Für das grundlegende Deployment von APEX konnte deswegen eine Role erstellt werden, bei dem aktuell bekannte Freiheitsgrade über Variablen gesteuert werden. Allerdings ist je Stakeholder ein eigenes Playbook notwendig, das diese APEX Role nutzt und zusätzlich den individuellen Code deployed. In diesem Playbook kann auch auf zusätzliche Komponenten Rücksicht genommen werden, die der Stakeholder benötigt. In diesem Fall ist es also nicht möglich, für jeden Stakeholder dieselben Ansible Playbooks wiederzuverwenden. Man kann lediglich das Grundgerüst der Komponenten automatisieren, es wird aber für jeden neuen Stakeholder ein erneuter initialer Aufwand notwendig sein.

Erfassen der Metriken: Bei dem APEX Framework handelt es sich um eine Neuentwicklung, weswegen kein Vergleich zum DC1.0 aufgestellt werden kann.

Die Anzahl der Komponenten bezieht sich auf das Grundgerüst von APEX, mit der Basisapplikation und dem Apache Solr. Es wurde keine spezifische Komponente eines Stakeholders mit einbezogen. Aus diesem Grund ist der initiale Aufwand auch geteilt. Die erste Zahl in Tabelle 4.15 bezieht sich auf den Aufwand der Implementierung für das Grundgerüst von APEX. Dieser Aufwand ist nur einmal notwendig und unabhängig von der Anzahl der Stakeholder. Die zweite Zahl bezieht sich auf den einmaligen Aufwand, der für jeden weiteren Stakeholder nötig ist, um APEX vorzubereiten. Zusätzlich kommt noch der Aufwand für die Automatisierung spezieller Komponenten, die der Stakeholder benötigt dazu. Diese können aber im Voraus nicht abgeschätzt werden.

Die Basisapplikation von APEX ist zustandslos und kann verteilt betrieben werden. Da die Möglichkeit der Verteilung grundlegend vom individuellen Code des Stakeholders abhängt, wird APEX in diesem Fall nicht als verteilte Applikation betrachtet. Dies wird voraussichtlich auch den Hauptanwendungsfall von APEX ausmachen.

Da es, je Ausprägung von APEX, nur eine Node geben wird, ist die Zeit für alle

Metrik		Menge
Initialer Aufwand	Implementation	100 PS
	je Stakeholder	8 PS
	wiederverwendete Roles	5
Komplexität	Anzahl der Komponenten	7
	verteiltes System	Nein
Durchlaufzeit	a.) eine Node	ca. 4 Minuten
	b.) alle Nodes	ca. 4 Minuten
	c.) zusätzliche Node	ca. 8 Minuten

Tabelle 4.15: Erfasste Metriken des APEX Frameworks

Nodes gleich der Zeit für eine Node. Ein Deployment auf allen Nodes macht in diesem Fall keinen Sinn und müsste auch mit allen Stakeholdern abgesprochen werden. Dies wäre nur der Fall, wenn sich kritische Änderungen am Grundgerüst ergeben, die sofort auf allen Nodes ausgerollt werden müssen. Dann müsste man die Zeit für eine Node, lediglich mit der Anzahl der Nodes multiplizieren, um die gesamt benötigte Zeit für ein sequentielles Update zu erhalten.

Kapitel 5

Ergebnisse der Fallstudie

Zum Abschluss der Arbeit wird das Ergebnis der Fallstudie betrachtet. Dazu werden die Metriken (siehe Abschnitt 3.1.3) aller Abschnitte zusammengerechnet und analysiert. In Abschnitt 5.2 werden weitere Erkenntnisse aus der Implementationsphase aufgezeigt und diskutiert. Insbesondere wird dabei auf die Forschungsfragen aus Abschnitt 3.1.2 eingegangen und diese beantwortet. Die gewonnenen Erkenntnisse geben einen Einblick in die Umsetzung der Konzepte von DevOps. Zuletzt wird im abschließenden Fazit das Ergebnis kurz zusammengefasst.

5.1 Auswertung der Metriken

Im Folgenden werden die erfassten Werte der Metriken **Initialer Aufwand**, **Durchlaufzeit** und **Charakteristiken für den Aufwand** analysiert und erläutert.

Initialer Aufwand: Zur Auswertung dieser Metrik wird zuerst ein Vergleichswert aufgestellt, der die maximal mögliche Arbeitszeit des Projektteams in Personenstunden (PS) repräsentiert. Betrachtet man den Zeitraum in der diese Arbeit durchgeführt wurde, (01.01.2015 bis 30.06.2015) so ergeben sich in etwa 100 Arbeitstage¹ pro Person. Bei drei Personen im Projektteam und jeweils acht Arbeitsstunden pro Tag, ergibt sich eine gesamte maximale Arbeitszeit von 2400 Personenstunden.

Wie in Abschnitt 4.3.6 kalkuliert, benötigt der Aufbau der Basisinfrastruktur 628 Personenstunden. Die Entwicklung aller Konzepte und der Aufbau der Systeme erfordern weitere 210 Personenstunden. Die Umsetzung der Konzepte von DevOps für die ausgewählten Applikationen benötigt zusammen 668² Personenstunden. Der Aufwand für

¹Abzüglich aller Feiertage, Wochenenden, Krankenstände und Urlaube

²Delivery Agent: 80 PS, OIDC: 300 PS, CPAS: 180 PS, APEX: 108 PS

die Implementierung beträgt also insgesamt 1506 Personenstunden. Verglichen mit den maximal möglichen 2400 Stunden verbleiben 894 freie Personenstunden. Die 1506 Personenstunden inkludieren lediglich den Aufwand für die konkrete Umsetzung, nicht aber weitere Tätigkeiten, wie beispielsweise Besprechungen oder organisatorische Aufgaben. Die Fallstudie hat gezeigt, dass die Umsetzung mit einem drei Personen Team innerhalb des vorgegebenen Zeitraums möglich ist. Die gesamte Implementierung wurde Ende Mai 2015 abgeschlossen, das entspricht einer Dauer von fünf Monaten. Eine Umsetzung der Konzepte von DevOps kann demnach auch mit einer geringen Anzahl von Mitarbeitern durchgeführt werden.

Durchlaufzeit: Für zwei Applikationen (Delivery Agent und CPAS) kann ein Vergleich der Durchlaufzeiten aufgestellt werden, da diese bereits im DC1.0 betrieben wurden. Die Tabelle 5.1 listet die Metriken der Durchlaufzeit für vorher und nachher auf.

Metrik	Delivery Agent			CPAS		
Durchlaufzeit	vorher	nachher	Gewinn	vorher	nachher	Gewinn
a.) eine Node	20 Min.	7 Min.	285%	25 Min.	10 Min.	250%
b.) alle Nodes	40 Min.	15 Min.	266%	320 Min.	15 Min.	2133%
c.) zusätzliche Node	90 Min.	10 Min.	900%	180 Min.	45 Min.	400%

Tabelle 5.1: Vergleich der Durchlaufzeiten des Delivery Agent und CPAS von vorher zu nachher. Die Definitionen für a, b und c finden sich in Abschnitt 3.1.3.

Wie man erkennen kann, ergeben sich erhebliche Steigerungen in der Effizienz. Der ermittelte Zeitgewinn beträgt mindestens 250 Prozent. Das volle Potential zeigt sich bei der CPAS Applikation und der Metrik b (alle Nodes): Durch die hohe Anzahl an Nodes (13 Stück), die gleichzeitig aktualisiert werden können, ist der Zeitgewinn wesentlich größer, als beim Delivery Agent mit nur zwei Nodes. Das bedeutet, umso mehr Nodes einer Applikation existieren, desto größer ist der sich ergebende Zeitgewinn. Beim CPAS entspricht dies einer Steigerung von 2133 Prozent. Soll ein unterbrechungsfreier Service garantiert werden, so dürfen nicht alle Nodes gleichzeitig aktualisiert werden. In diesem Fall entspricht die benötigte Zeit näherungsweise dem doppelten Zeitaufwand der Metrik a (eine Node). Auch hierbei ergibt sich ein Zeitgewinn, verglichen mit der Situation vor der Umsetzung genannter Konzepte.

Untersuchte Durchlaufzeiten zeigen also, dass mit der Umsetzung von DevOps Konzepten eine enorme Zeitersparnis erreicht wird und daher die Effizienz erheblich steigt.

Charakteristiken für den Aufwand: Die letzte Metrik aus Abschnitt 3.1.3 befasst sich mit Charakteristiken für die Abschätzung des nötigen Aufwands. Mögliche Kandidaten dafür sind unter anderem, die Komplexität einer Applikation anhand der Anzahl ihrer Komponenten und die Verteilungseigenschaften. Diese beiden Merkmale wurden für die Applikationen vorliegender Arbeit erfasst. Die Tabelle 5.2 fasst diese Metriken zusammen und stellt sie gegenüber.

Applikation	Komponenten	Verteiltes System	Aufwand
Consul Cluster	2	Ja	80 PS
Loadbalancer	5	Ja	230 PS
Logstash	13	Ja	120 PS
Sensu	9	Ja	150 PS
Delivery Agent	7	Ja	80 PS
OIDC	10	Ja	300 PS
CPAS	10	Nein	180 PS
APEX	7	Nein	100 PS

Tabelle 5.2: Zusammenfassung der Charakteristiken aller Applikationen. Je höher die Zahl in Spalte Komponenten, desto höher ist die Komplexität einer Applikation.

Betrachtet man oben stehende Tabelle 5.2 so fällt auf, dass die Anzahl der Komponenten keinen direkten Einfluss auf den Implementierungsaufwand hat. Die Komponentenanzahl lässt lediglich eine sehr grobe Abschätzung des nötigen Aufwandes zu. Ebenso scheinen die Verteilungseigenschaften einer Applikation keinen erheblichen Einfluss auf den benötigten Aufwand zu nehmen. In vorliegender Arbeit konnte demnach keine Charakteristik anhand der zwei ausgewählten Kandidaten ermittelt werden, die den Aufwand der Implementierung abschätzen lässt.

5.2 Diskussion der Ergebnisse

Die Ergebnisse aus Abschnitt 5.1 werden im Folgenden genauer erläutert. Dazu wird im speziellen auf die Forschungsfragen aus Abschnitt 3.1.2 eingegangen und diese diskutiert.

- **Bleibt der Aufwand der Implementierung annähernd gleich oder verändert sich dieser mit Anzahl der Applikationen? / Gibt es nutzbare Gemeinsamkeiten bei der Implementation?**

Um diese Frage ausreichend beantworten zu können, war die Menge an untersuchten Applikationen zu klein. So kann aus einem Vergleich der Aufwände der vier Applikationen keine detaillierte Schlussfolgerung gezogen werden. Allerdings ist eine Tendenz erkennbar. Je mehr Komponenten bereits automatisiert wurden, desto weniger Aufwand erfordern weitere Applikationen, die dieselben Komponenten nutzen. Der notwendige Aufwand nimmt demnach potentiell mit Anzahl der Applikationen ab. Je homogener die Applikationslandschaft eines Unternehmens ist, desto weniger Aufwand wird demnach benötigt. Da in der Fallstudie aber sehr verschiedene Applikationen mit unterschiedlichen Komponenten betrachtet wurden, tritt dieser Effekt kaum auf. Ein paar Komponenten wie Supervisor, Java, Keepalived, HAProxy oder Tomcat wurden allerdings immer wieder benötigt. In Tabelle 5.3 erkennt man, dass für diese wiederkehrenden Komponenten bereits bestehende Roles wiederverwendet wurden. Der Aufwand der Implementierung für die jeweilige Applikation konnte demnach in diesem Maß verringert werden.

Applikation	wiederverwendete Roles
Consul Cluster	0
Loadbalancer	2
Logstash	3
Sensu	3
Delivery Agent	6
OIDC	4
CPAS	5
APEX	5

Tabelle 5.3: Anzahl der wiederverwendeten Roles pro Applikation. Je höher die Zahl, desto höher ist der Grad an Wiederverwendung und desto stärker sinkt der Implementationsaufwand.

Eine Voraussetzung, um diesen positiven Effekt zu erhalten ist, dass die Ansible Roles für die Komponenten entsprechend generisch ausgeführt sind. Jede Komponente muss in einer eigenen Role gekapselt sein. Diese darf nur eine Basisinstallation der Komponente, ohne spezielle Konfiguration für eine Applikation, vornehmen. Ansonsten ist eine Wiederverwendung der Role nicht möglich und der Implementationsaufwand wird jedes mal notwendig. Erst in der Role für die von den anderen Komponenten abhängige Applikation, dürfen diese entsprechend konfiguriert werden. Der minimale Aufwand für die Automatisierung einer Applikation ist demnach die Erstellung einer Role für diese Applikation. Dies gilt unter der Voraussetzung, dass bereits alle anderen Komponenten automatisiert sind. Die richtige Strukturierung der Roles ist also entscheidend.

Zusammenfassend wurde festgestellt, dass es nutzbare Gemeinsamkeiten gibt, die den Arbeitsaufwand reduzieren. Dies sind vor allem mehrfach genutzte Komponenten. Für jede Applikation ist allerdings ein Mindestmaß an Aufwand notwendig. Dieser lässt sich auch mit hoher Anzahl an Applikationen nicht auf Null reduzieren.

- **Gibt es Charakteristiken anhand derer sich der Implementationsaufwand abschätzen lässt?**

Wie bereits in Abschnitt 5.1 erwähnt, konnten im Laufe vorliegender Arbeit keine Charakteristiken ermittelt werden, die den Aufwand annähernd abschätzen lässt. So benötigte der Loadbalancer mit nur fünf Komponenten wesentlich mehr Zeit als Sensu mit neun Komponenten. Ein Grund hierfür war, dass beim Loadbalancer zuerst ein Proof of Concept aufgebaut werden musste, der bei Sensu nicht nötig war. Auch die Applikationen CPAS und OIDC, mit gleicher Anzahl an Komponenten, weisen einen signifikanten Unterschied im benötigten Aufwand auf. Grund hierfür waren die Probleme bei der Automatisierung von OpenLDAP bei OIDC. Die Anzahl der Komponenten ist demnach kein ausreichend gutes Kriterium, um den benötigten Aufwand abschätzen zu können. Es hängt sehr davon ab, welche Komponenten benötigt werden und ob diese bereits Konzepte für eine Automatisierung unterstützen. Ist dies nicht der Fall, so steigt der Implementationsaufwand erheblich.

Auch die Verteilungseigenschaften einer Applikation (Tabelle 5.2) haben keinen erkennbaren Einfluss auf den benötigten Aufwand. So wurde für den Delivery Agent, der Verteilungseigenschaften aufweist, weniger Zeit benötigt als für APEX ohne Verteilung. Bei OpenLDAP hingegen ergab die notwendige Verteilung einen wesentlich größeren Mehraufwand. Die Fallstudie lässt die Schlussfolgerung zu, dass kein Mehraufwand entsteht, wenn eine Applikation auf einen verteilten Einsatz ausgelegt ist. Werden diese Konzepte allerdings nicht ausreichend unterstützt, so ergeben sich wesentlich mehr Probleme und somit mehr Aufwand bei der Automatisierung.

Der Aufwand der Implementierung für eine Applikation lässt sich demnach im Vorfeld nur schwer abschätzen. Für eine annähernd genaue Schätzung ist ein hohes Maß an Erfahrung notwendig. Anhand der beteiligten Komponenten lässt sich lediglich eine sehr grobe Schätzung erstellen.

- **Welche Infrastruktur ist grundsätzlich nötig für den Betrieb mit Dev-Ops? / Welche neuen Konzepte werden benötigt und wie können diese aussehen?**

Betrachtet man die beiden Abschnitte 4.2 und 4.3 so waren für die Red Bull Media Base einige neue Konzepte und Infrastrukturen notwendig. Diese müssen aber nicht

zwingend bei jedem Unternehmen nötig sein. Grundsätzlich werden Konzepte für die Automatisierung jener Tätigkeiten benötigt, die noch manuell durchgeführt werden. Im klassischen Betrieb betrifft dies oft die Konfiguration von Monitoring und Logging Systemen. Auch die Konfiguration des Loadbalancers ist meist eine manuelle Tätigkeit. Diese drei Bereiche sollten demnach in jedem Fall betrachtet werden. Möchte man die Konzepte von DevOps in vollem Umfang umsetzen, so müssen diese Systeme die Möglichkeit bieten, sich selbst automatisch zu konfigurieren. Dies erfolgt meist durch eine selbständige Anmeldung der jeweiligen neuen Maschine beim entsprechenden System. Die gewählten Systeme Sensu und Logstash verfolgen genau diesen Ansatz und gehören somit zur nächsten Generation von Monitoring/Logging Systemen. Das bisher weit verbreitete System Nagios unterstützt diese Konzepte derzeit nicht vollständig und ist daher nicht für den Betrieb mit DevOps geeignet.

Weitere Aufgaben, wie die Erstellung der virtuellen Maschinen, dürfen in diesem Prozess nicht außer acht gelassen werden. Wichtig ist dabei, dass sich auch die Applikationen an diese neuen Konzepte anpassen müssen, da nur eine Änderung der Basisinfrastruktur nicht ausreichend ist.

Eine Automatisierung der Applikationen kann grundsätzlich auch ohne Basisinfrastruktur durchgeführt werden. Allerdings sind so immer manuelle Tätigkeiten notwendig, um den kompletten Prozess eines Deployments abzuschließen. Man erhält also nicht alle Vorteile von DevOps.

- **Ist der Aufwand der Basisinfrastruktur einmalig oder pro Applikation erforderlich?**

Grundsätzlich ist der Aufwand nur einmalig notwendig. Allerdings ergeben sich wenige Aufgaben, die pro Applikation betrachtet werden müssen. Diese Aufgaben sind in Tabelle 5.4 gelistet. So muss für jede neue Applikation zumindest der Build- und Deploymentplan konfiguriert, sowie eine Formation angelegt werden. Muss eine Applikation aus dem Internet erreichbar sein, so muss noch der Aufwand für Consul betrieben werden. Die Integration in den Loadbalancer ist somit ebenfalls abgeschlossen. Dies ergibt also etwa vier Personenstunden, die je Applikation an der Basisinfrastruktur anfallen. Dies ist verglichen mit den 838 Stunden, die für den Aufbau der Basisinfrastruktur nötig waren, vernachlässigbar.

Für die Integration in Logstash und Sensu werden entsprechende Monitoring Checks und Log-Filter benötigt. Diese sind allerdings wiederverwendbar und können bei einem Großteil der Applikationen angewandt werden. Die Erstellung eines Basissets von Checks und Filter ist bereits in den Aufwand für den Aufbau von Sensu/Logstash mit eingerechnet. Für die meisten Applikationen fällt hiermit kein weiterer Aufwand mehr

System/Aufgabe	Aufwand je Applikation
Konfiguration Build & Deploymentplan	1 PS
Erstellung Formation	2 PS
Health Check für Consul	1 PS
Integration in Loadbalancer	0 PS
Integration in Logstash	0 PS
Integration in Sensu	0 PS

Tabelle 5.4: Arbeitsaufwände je Applikation, die für eine Integration in die Basisinfrastruktur notwendig sind.

an. Es kann jedoch vorkommen, dass spezielle Applikationen weiteren Arbeitsaufwand erfordern. Dieser beträgt erfahrungsgemäß 1-3 Personenstunden.

- **Wie verändert sich die Effizienz nach der Umstellung?**

Wie in Tabelle 5.1 dargestellt, wurde im Laufe der Fallstudie eine Steigerung der Effizienz von mindestens 200 Prozent beobachtet. Dies betrifft allerdings nur die Zeit für ein Deployment einer Applikation. Betrachtet man die Verfügbarkeit der Mitarbeiter, so ergibt sich eine wesentlich höhere Effizienzsteigerung. Bei einem manuellen Deployment sind die Mitarbeiter die gesamte Zeit mit dem Deployment beschäftigt und können somit keiner anderen Tätigkeit nachgehen. Beim automatisierten Deployment laufen alle nötigen Schritte ohne eingreifen von Personal ab. Es muss lediglich die Zeit zum Starten des Deployments betrachten werden, die erfahrungsgemäß bei weniger als einer Minute liegt. Währenddessen können andere Tätigkeiten durchgeführt werden. Betrachtet man also nicht nur die Durchlaufzeit des Deployments sondern auch die Verfügbarkeit von Mitarbeitern, so ergibt sich eine weitaus höhere Steigerung der Effizienz.

- **Weitere Erkenntnisse:** Im Laufe vorliegender Arbeit wurden noch weitere Erkenntnisse gesammelt, die im Folgenden aufgeführt sind:

Es konnte beobachtet werden, dass vor allem ältere Applikationen, wie OpenLDAP mehr Probleme bei der Automatisierung verursachen als neuere Applikationen. Dies liegt daran, dass viele neue Applikationen bereits einige der Konzepte von DevOps unterstützen und somit eine Automatisierung vereinfachen. Ein Beispiel hierfür ist etwa Consul, bei dem, nach erfolgreicher Installation, nur in Ausnahmefällen manuelle Tätigkeiten notwendig sind. Grundsätzlich konnte beobachtet werden, dass neue Applikationen und Technologien weniger Aufwand benötigen und deshalb zu bevorzugen sind.

Eine weitere Beobachtung die gemacht wurde ist, dass DevOps bei sehr kleinen Änderungen und wenigen Nodes langsamer ist, als die bisherige manuelle Tätigkeit. Muss beispielsweise eine Konfiguration auf nur einer einzigen Node angepasst werden, so ist die manuelle Durchführung schneller. Umso mehr Nodes betroffen sind, desto geringer wird allerdings der Zeitvorteil. Im Sinne von DevOps müsste die Änderung zuerst lokal getestet und dann in das Source Code Verwaltungssystem eingchecked werden. Anschließend muss ein Deployment gestartet werden, um diese Änderung auf der Node auszurollen. Der Vorteil hierbei ist aber, dass die Änderung im Source Code Verwaltungssystem dokumentiert und somit nachvollziehbar ist.

5.3 Abschließendes Fazit

Die Einführung von DevOps und Continuous Delivery ist mittlerweile nicht nur großen Unternehmen wie Amazon oder Google vorbehalten. Immer mehr Unternehmen, vor allem auch kleine Unternehmen wie Start-Ups, verfolgen diese Konzepte. Dies zeigt im Besonderen die Untersuchung von [2]. So sagen 68 Prozent der befragten Unternehmen, dass DevOps und Continuous Delivery bereits ein Standard ist bzw. bald werden wird. Es ist die Tendenz erkennbar, dass DevOps und Continuous Delivery für Software-Unternehmen notwendig wird, um wettbewerbsfähig zu bleiben.

Die Umstellung auf die Konzepte von DevOps für die Red Bull Media Base war ein voller Erfolg. Die Umsetzung konnte unter den zeitlichen Vorgaben (Beginn 2015 bis Mitte 2015) und mit den geplanten Ressourcen (drei Personen) durchgeführt werden. Die Vorteile die dadurch erzielt wurden sind enorm und ermöglichen es, personelle Ressourcen auf wichtigere Tätigkeiten zu fokussieren. Dies betrifft insbesondere die Steigerung der Effizienz, die kurzen Durchlaufzeiten und die höhere Flexibilität. Trotzdem darf der Aufwand der Implementierung nicht unterschätzt werden. Die Umstellung betrifft viele Bereiche die mit einbezogen werden müssen. Das führt zu erheblichen Änderungen in den bisherigen Prozessen und der Infrastruktur eines Unternehmens. Darüber hinaus ist die gesamte Unternehmenskultur betroffen. Eine vollständige Umstellung auf DevOps kann deswegen nur erreicht werden, wenn die Konzepte von allen Beteiligten verstanden und umgesetzt werden.

Da eine komplette Umstellung allerdings mit viel Aufwand und auch mit Risiken verbunden ist, kann eine schrittweise Umstellung sinnvoll sein. Es ist für kleinere Unternehmen eventuell leichter, zuerst nur Teile der Build- und Deployment-Pipeline zu automatisieren. Dadurch können vorerst geringe Verbesserungen erzielt werden. In weiterer Folge kann dies ausgeweitet und ein Teil nach dem Anderen umgestellt werden, um vollständig die Konzepte von DevOps einzuführen.

Tabellenverzeichnis

4.1	Generelle Netzwerkregeln des DC2.0	39
4.2	Erfasste Metriken für Ansible	45
4.3	Erfasste Metriken für Bamboo	48
4.4	Erfasste Metriken für Vagrant	49
4.5	Erfasste Metriken für den Lifecycle	51
4.6	Erfasste Metriken für Formation Skripte	53
4.7	Erfasste Metriken für Packer	57
4.8	Erfasste Metriken des Consul Clusters	59
4.9	Erfasste Metriken des Loadbalancers	63
4.10	Erfasste Metriken der Logging Infrastruktur	67
4.11	Erfasste Metriken der Monitoring Infrastruktur	72
4.12	Erfasste Metriken des Delivery Agents	76
4.13	Erfasste Metriken des OIDC Servers	80
4.14	Erfasste Metriken des CPAS	84
4.15	Erfasste Metriken des APEX Frameworks	88
5.1	Vergleich der Durchlaufzeiten von vorher und nachher	90
5.2	Zusammenfassung der Charakteristiken aller Applikationen	91
5.3	Anzahl der wiederverwendeten Roles pro Applikation	92
5.4	Arbeitsaufwand je Applikation für die Basisinfrastruktur	95

Abbildungsverzeichnis

2.1	Budgetverteilung einer Applikation ohne DevOps, aus [19]	14
2.2	Budgetverteilung einer Applikation mit DevOps, aus [19]	15
2.3	Grundlegende zu automatisierende Aufgaben	17
2.4	Testpyramide, adaptiert aus [23]	20
2.5	Schema einer Deployment Pipeline, adaptiert aus [8] und [23]	23
4.1	Architektur von OpenNebula, adaptiert aus [13]	40
4.2	Deploymentplan des Delivery Agent	47
4.3	Aufbau einer Loadbalancer Node	62
4.4	Grundsätzliches Funktionsschema des ELK-Stacks, adaptiert aus [16] .	64
4.5	Aufbau der ELK-Stack Node	65
4.6	Grundsätzliches Funktionsschema der Sensu Monitoring Infrastruktur .	69
4.7	Aufbau einer Sensu Node	71
4.8	Aufbau einer Delivery Agent Node	75
4.9	Aufbau einer OIDC Node	79
4.10	Aufbau einer CPAS Node	83
4.11	Minimaler Aufbau einer APEX Node	86

Auflistungsverzeichnis

4.1	Beispiel: Ansible Playbook des Delivery Agents	43
4.2	Beispiele: Ansible Inventory des Delivery Agent	44
4.3	Beispielhaftes Vagrantfile des Delivery Agent	48
4.4	Beispiel eines Formation Template für virtuelle Maschinen	52
4.5	Beispielhafte Client Konfiguration des Delivery Agent	71

Literaturverzeichnis

- [1] Ansible, Inc.: *The Benefits of Agentless Architecture*. Technischer Bericht, Ansible, Inc., 2014. http://cdn2.hubspot.net/hub/330046/file-479013288-pdf/pdf_content/The_Benefits_of_Agentless_Architecture.pdf.
- [2] DZone, Inc.: *Key Research Findings*. The Guide to Continuous Delivery, 2:4–5, Feb. 2015. https://dzone.com/storage/assets/18140-dzone_2015continuousdelivery_9.pdf.
- [3] Feitelson, Dror G., Eitan Frachtenberg und Kent L. Beck: *Development and Deployment at Facebook*. IEEE Internet Computing, 17(4):8–17, 2013. <http://doi.ieeecomputersociety.org/10.1109/MIC.2013.25>.
- [4] Fowler, Martin: *PhoenixServer*. <http://martinfowler.com/bliki/PhoenixServer.html>, 2012. Abgerufen am 12.06.2015.
- [5] Fowler, Martin: *ContinuousDelivery*. <http://martinfowler.com/bliki/ContinuousDelivery.html>, 2013. Abgerufen am 06.05.2015.
- [6] Gluchowski, Alexander und Stefan Reinheimer: *Agilität in der IT*. Dpunkt.Verlag GmbH, 1. Auflage, 2013, ISBN 9783864900587.
- [7] Hall, Daniel: *Ansible Configuration Management*. Packt Publishing Ltd, Birmingham, 2013, ISBN 9781783280810.
- [8] Humble, Jez und David Farley: *Continuous Delivery - Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, Amsterdam, 1. Auflage, 2010, ISBN 9780321670229.
- [9] Humble, Jez, Joanne Molesky und Barry O'Reilly: *Lean Enterprise - How High Performance Organizations Innovate at Scale*. O'Reilly Media, Inc., Sebastopol, 1. Auflage, 2014, ISBN 9781491946541.
- [10] Kim, Gene: *Uncovering The DevOps Improvement Principles At Google (Randy Shoup Interview)*. <http://itrevolution.com/>

- uncovering-the-devops-improvement-principles, 2014. Abgerufen am 14.04.2015.
- [11] Lawton, George: *How Amazon Made the Leap to a DevOps Culture*. <http://servicevirtualization.com/how-amazon-made-the-leap-to-a-devops-culture/>, 2013. Abgerufen am 14.04.2015.
- [12] Meckfessel, Melody: *How DevOps and the Cloud Changed Google Engineering*. <http://www.infoq.com/presentations/google-devops-cloud>, 2014. Abgerufen am 14.04.2015.
- [13] Molina, Daniel, Carlos Martín Sánchez, Jaime Melis, Javier Fontán, Constantino Vázquez, Ruben S Montero und Ignacio M Llorente: *The OpenNebula Cloud Toolkit*. Open Source Cloud Computing Systems: Practices and Paradigms: Practices and Paradigms, 2012.
- [14] Morris, Kief: *ImmutableServer*. <http://martinfowler.com/bliki/ImmutableServer.html>, 2013. Abgerufen am 12.06.2015.
- [15] Pandey, Sudhir: *Investigating Community, Reliability and Usability of CFEngine, Chef and Puppet*. Diplomarbeit, Oslo and Akershus University College, 2012. <https://www.duo.uio.no/bitstream/handle/10852/9083/pandey.pdf>.
- [16] Papaspyrou, Dr. Alexander: *Eine Bresche schlagen: Loganalyse mit dem freien ELK-Stack*. iX Magazin für professionelle Informationstechnik, 11:124–128, 2014.
- [17] Peschlow, Patrick: *Die DevOps-Bewegung: Was ist das eigentlich und was bedeutet es für uns?* Java Magazin, 1:32–40, 2012.
- [18] Schöder, René: *Kettenreaktion: Von Continuous Integration zu Continuous Delivery mittels DevOps*. Entwickler Magazin Spezial, 4:6–10, 2015.
- [19] Siprell, Stefan: *Auch Umgebungen fallen vom Himmel: Konfigurations- und Betriebskosten in den Griff bekommen*. Java Magazin, 6:85–90, 2014.
- [20] Skelton, Matthew: *The Continuous Delivery Toolchain*. 2014 Guide to Continuous Delivery, 1:14–15, 2014. https://dzone.com/storage/assets/9049-dzr_2014guidetocontinuousdelivery_1.pdf.
- [21] Vaughan-Brown, Justin: *Markteinführung mit DevOps schneller und besser: Wie die Zusammenarbeit von Entwicklung und Betrieb den Geschäftserfolg von Unternehmen beflügeln kann*. Business Technology, 3:30–34, 2014.
- [22] Walls, Mandi: *Building a DevOps Culture*. O'Reilly Media, Inc., Sebastopol, 1. Auflage, 2013, ISBN 9781449368364.

- [23] Wolff, Eberhardt: *Continuous Delivery - Der pragmatische Einstieg*.
Dpunkt.Verlag GmbH, Heidelberg, 1. Auflage, 2014, ISBN 9783864902086.