dotSource®

Deployment und Continuous Integration mit Containerisierung in der Cloud

[redacted]

Projektarbeit Nr.: [redacted]

von: [redacted]

Matrikelnummer: [redacted]

Duale Hochschule: [redacted]

Studienbereich: Technik

Studiengang: Praktische Informatik

Kurs: [redacted]

Ausbildungsstätte: [redacted]

Gutachter der
Dualen Hochschule Gera-Eisenach: [redacted]

Gutachter des Praxispartners
(Betreuer i.S.v. §20(1) DHGEPrüfO): [redacted]

dotSource®

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften – auch in digitaler Form – gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH. Die Arbeit ist nur Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CD	Continuous Delivery bzw. Continuous Deployment
CI	Continuous Integration
IT	Informationstechnik
JAR	Java Archive
JRE	Java Runtime Environment
JSON	JavaScript Option Notation
YAML	YAML Ain't Markup Language

1 Einleitung

In den letzten Jahren konnten mehrere größere Umbrüche in der digitalen Industrie festgestellt werden. Der erste Umbruch brachte eine neue Art des Deployments und die damit verbundene Infrastruktur mit sich, nämlich die Cloud. Cloud Computing ist seit Jahren ein diskutiertes Thema, da es immer wieder als die Zukunft der Informationstechnik (IT) bezeichnet wird. Außerdem sorgt es bei vielen Unternehmen für geringere Kosten, da selbst verwaltete Infrastruktur als auch das zu verwaltende Personal wegfallen. Auf Cloud-Ebene wurde vor allem von der Technologie der Virtualisierung gebraucht gemacht. Damit wurde reale Hardware abstrahiert und den verschiedenen Nutzern angeboten. Es stellte sich jedoch schnell heraus, dass Virtualisierung mit virtuellen Maschinen ineffizient ist. Dies ist der Grund für den zweiten Umbruch, mit dem sogenannte Container in die Cloud und die Server Einzug hielten. Diese ermöglichten einen schnellen und zuverlässigen Betrieb ohne ein seperates Betriebssystem, da sie auf dem Host-System ausgeführt werden und dieses zusammen mit seinen Ressourcen nutzen. Nun stellte sich die Frage, wie diese Container am besten orchestriert werden können. Obwohl es einige Lösungen für dieses Problem gab, setzte sich zuerst Kubernetes als Open-Source-Projekt zur Container-Orchestrierung durch und stellt momentan einen Industriestandard dar.

Mit Kubernetes als De-facto-Standard für verteilte Systeme in der Cloud müssen sich Unternehmen nun die Frage stellen, wie sie am besten diese neue Technologie in ihr Unternehmen, ihre Software, Produkte als auch ihren Deploymentprozessen mit integrieren können. Letztendlich musste die Option eines Zero-Downtime Deployments, bei der der eigentlich angebotene Service, wenn Updates bereit gestellt werden, nicht heruntergefahren werden muss, sowie ein im Allgemeinen stabileres bzw. robusteres System gegeben sein.

Auch in der dotSource GmbH gewinnt die containerisierte Entwicklung und Deployment von Softwarelösungen zunehmend an Bedeutung. Denn auch den Kunden sind die Vorteile der Cloud bekannt, sodass entsprechende Lösungen auch in ihren eigenen Systemen umgesetzt werden sollen. Diese Arbeit beschäftigt sich mit der historischen Entwicklung von DevOps, dessen Anforderungen und damaligen Lösungsansätzen. Mit diesen neuen An-

forderungen wird versucht einen Vergleich mit echten und modernen Tools durchzuführen und Schwächen und Stärken genauer zu untersuchen. Schließlich soll ein Prototyp in einem der evaluierten Tools umgesetzt werden und dessen Schwächen und Stärken während der Umsetzung nochmal genau betrachtet werden.

2 Theoretische Grundlage

2.1 Cloud Computing

Cloud Computing betrifft das Bereitstellen und Verwalten von IT-Ressourcen über ein Netzwerk. Das entsprechende Netzwerk bei Public Clouds ist meist das Internet kann aber auch bei anderen Cloudmodellen, wie Private Clouds, ein selbst verwaltetes Netzwerk sein. Über dieses Netzwerk wird dann meist eine breite Palette von Diensten angeboten. Es handelt sich dabei um virtuelle Maschinen, Application Programming Interface (API)-Gateways, Computercluster, Netzwerkinfrastruktur und andere IT-bezogene Ressourcen. Diese werden in einigen Abonnement-Modellen nach dem Gebrauch abgerechnet.[1]

Abhängig von der Einbindung der eigenen Infrastruktur können vier verschiedene Deployment Modelle unterschieden werden. So wird eine Private Cloud-Infrastruktur nur für eine einzelne Organisation oder Gruppe bereitgestellt. Das Community Cloud-Modell öffnet die Private Cloud weiter und ermöglicht die Nutzung durch mehrerer Organisationen und Gruppen mit den gleichen Interessensbereichen. Die Public Cloud öffnet ihre Dienste der Öffentlichkeit, egal ob Privatperson, Staatliche Behörde oder Unternehmen. Sollten nun zwei verschiedene Cloud Modelle zum Einsatz kommen und durch properitäre Technologien verbunden sein, welche den Datenaustausch ermöglichen, spricht man von einer Hybrid Cloud.[2]

Abseits des gewählten Deployment Modells können die einzelnen Dienste der verschiedenen Clouds in unterschiedliche Servicemodelle unterschieden werden. Je nach gewähltem Modell werden administrative Aufgaben gewonnen oder abgegeben (siehe Abbildung 1). Generell bleibt einem Kunden oder einer Gruppen von Kunden die Wahl, des zu wählenden Servicemodells, selbst überlassen, da diese an die einzelnen Dienste gebunden sind und wiederum frei wählbar sind. Will der Nutzer eine Virtuelle Maschine mit einem selbstgewähltem Betriebssystem aufsetzen, um eigene Dienste oder Webserver zu hosten, muss auf Dienste aus dem Bereich Infrastructure as a Service zurückgegriffen werden. Sollte

[1] [RK16], Vgl.
[2] [MG11], Vgl.

allerdings nur ein bestimmter Service laufen, wo dann auch der Traffic automatisch von einem Load Balancer verteilt wird, wird eine Platform as a Service genutzt. Dort kann sich auf die Entwicklung des Services konzentriert werden und die zugrunde liegenden Schichten müssen nicht konfiguriert werden. Wenn allerdings nur der Zugang zu Anwendungssoftware gewährt werden soll, so spricht man von Software as a Service. Bekannte Beispiele solcher Dienste sind die Microsoft Office-Produkte, welche auch im Browser geöffnet werden können.[3]

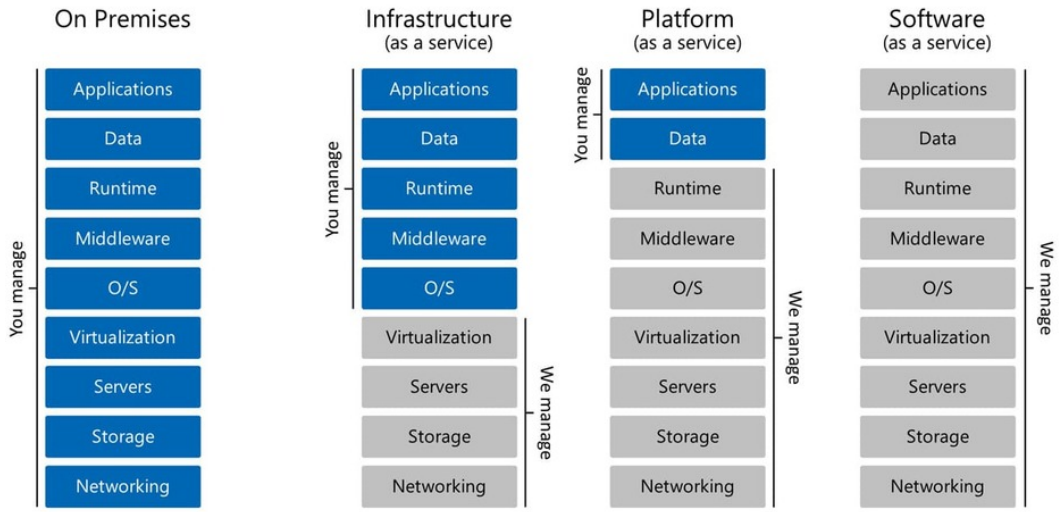

Abbildung 1: Vergleich der Verwaltungsaufgaben[4]

Zusätzlich zu den gewählten Diensten bietet Cloud Computing einige Vorteile für ein Unternehmen oder einer Einzelperson. In der Regel wird nur das gezahlt, was letztendlich gebraucht wird. Es entstehen keine Kosten durch nicht verwendete Hardware. Sollte die momentane Hardware für die vorhandene Last nicht ausreichen, kann ohne Schwierigkeiten das System skaliert und zusätzliche Ressourcen bereitgestellt werden. Direkte Kosten durch Hardwarekäufe und IT-Personal entfallen zwar nicht komplett können aber bei guter Umsetzung reduziert werden. Die Cloud unterstützt auch ein gewisses Maß an Flexibilität und ermöglicht eine schnellere Veröffentlichung und Skalierung der Ressourcen.[5]

[3] [RK16], Vgl.
[4] Tur16
[5] [RK16], Vgl.

2.2 Containerisierung

2.2.1 Allgemeiner Aufbau

Virtualisierung ist das Kernelement der Cloud und eines effektiven Deployment- bzw. Development-Prozesses. Nur selten bieten Cloud Anbieter ihren Kunden Hardware an, welche nur von ihnen und ohne irgendeine Virtualisierung verwendet werden kann. Es ist jedoch immer zu sehen, wie die klassische Virtualisierung, also der Einsatz von einem Hypervisor auf der physischen Hardware zum Extrahieren von Rechnerressourcen, durch die Conatainer-Virtualisierung abgelöst wird[6]. Googles eigene Dienste, wie Gmail, YouTube und die Cloud Shell werden ebenfalls in eigenen Containern ausgeführt[7].

Die Container selbst basieren auf der Idee, die verpackten Anwendungen von der Ausführungsumgebung zu abstrahieren und loszulösen. Dies gewährleistet eine einfache und konsistente Bereitstellung gewährleistet. Es muss dabei nicht das eigentliche Zielsystem berücksichtigt werden, da zum Ausführen eines Containers nur eine geeignete Container Runtime erforderlich ist. Bei der Ausführung eines Containers wird nicht wie bei virtuellen Maschinen ein gesamtes Gastsystem erstellt, welches über das Hostsystem und den Hypervisor auf die entsprechende Hardware zugreift. Die Container mit ihren enthaltenen Anwendungen werden vielmehr auf dem Host-Betriebssystem ausgeführt. Die dazu nötige Verbindung ist die Container Runtime (siehe Abbildung 2).[8]

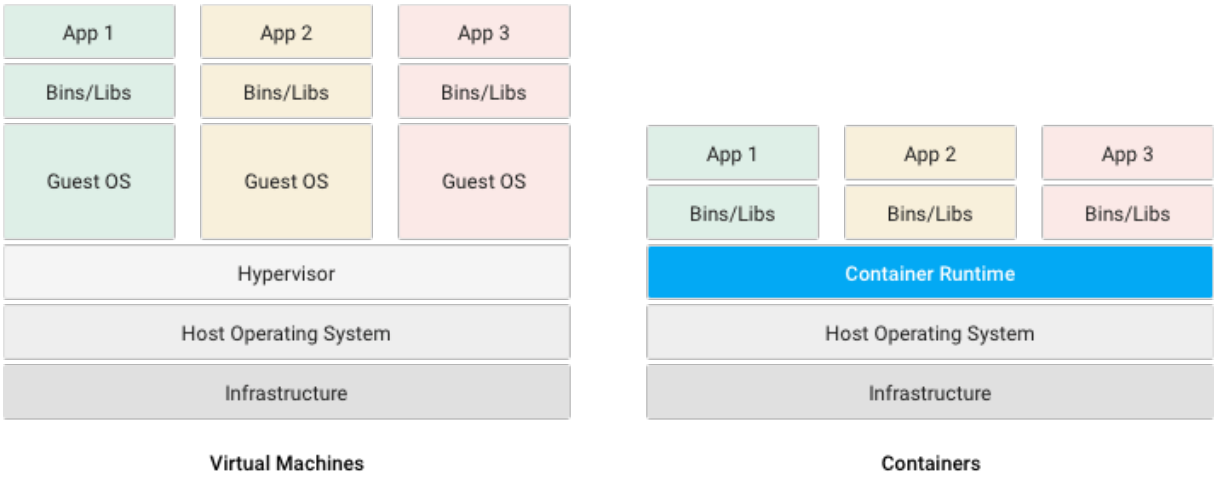

Abbildung 2: Vergleich von virtueller Maschine und Container[9]

[6] [Wol18], Vgl.
[7] [Goo19a], Vgl.
[8] Ebenda, Vgl.
[9] Goo19a

Aus diesem Grund bieten Container einige Vorteile gegenüber Virtuellen Maschinen. Auf diese Weise bieten sie eine konsistente Umgebung, in der unterschiedliche Anwendungen voneinander losgelöst sind. Da diese Umgebung alle für die Ausführung erforderlichen Abhängigkeiten enthält, kann sie überall dort wo eine Container Runtime vorhanden ist ausgeführt werden.

2.2.2 Docker

Die Docker Engine ist die weitverbreitetste Container Runtime, welche umfangreich von der Industrie genutzt wird[10]. Auf ihr laufende Container können mit Hilfe sogenannter Dockerfiles und Docker Compose files konfiguriert werden. Dabei wird in den Dockerfiles ein Baseimage genutzt, auf das entweder weitere Images geladen oder Befehle ausgeführt werden. Diese aus den Dockerfiles gebauten Images können dann als Container gestartet werden[11]. Wenn weitere Anwendungen in anderen Images genutzt werden, wird eine Composefile verwendet. Auf diese Weise wird verwaltet, welche Images mit welcher Version gestartet werden, welche Ports einander zugewiesen sind und weitere Dinge für den effektiven Einsatz, zum Beispiel das Erstellen von Volumes[12].

Mit diesen verschiedenen Konfigurationsmöglichkeiten können die Docker Container auf jedem System ausgeführt werden, solange die Docker Engine als Runtime vorhanden ist. Soll allerdings ein produktiver Einsatz erfolgen, wird ein System zum Orchestrieren von Containern benötigt. Die Lasten müssen auf mehrere Container auf unterschiedlichen Hardwaresystemen verteilt werden, Container müssen automatisch gestartet und heruntergefahren werden und die Fehlererkennung und -behandlung muss durchgeführt werden.

2.2.3 Kubernetes

Kubernetes ist ein Tool zur „Automatisierung der Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen“[13]. Es wurde ursprünglich von Google entwickelt, allerdings mit der Zeit in ein Open-Source-Projekt umgewandelt bzw. „an die Cloud Native Computing Foundation gespendet“[14]. Um seine Aufgaben als Container Verwaltung zu erfüllen, wird die Kubernetes-API genutzt, um den Status des ihr zugrundeliegenden Clu-

[10] [Doc20c], Vgl.
[11] [Doc20a], Vgl.
[12] [Doc20b], Vgl.
[13] [Kub20d]
[14] [Wik20a]

sters zu steuern. Sie bestimmt unter anderem, welche Containerimages verwendet werden, wie viele Repliken von einem Container existieren und teilt Festplatten- und Netzwerkressourcen zu. Der für den Cluster zuständige Entwickler, DevOps-Angestellte oder Systemadmin kann mit dem Konsolenbefehl kubectl über HTTP-Requests der Kubernetes-API mitteilen, wie genau sie sich verhalten muss. Diese Einstellungen werden dann auf den einzelnen Nodes angewandt, welche entweder physische Maschinen oder virtuelle Maschinen sind[15].

Neben den verschiedenen Nodes gibt es auch einen physischen oder virtuellen Rechner, welcher die Aufgabe des Masters übernimmt, und auf dem unter anderem der kube-apiserver ausgeführt wird. Der kube-apiserver ist derjenige Prozess, welcher mit dem Befehl kubectl angesprochen werden kann und die gesamte Kubernetes-API darstellt. Auf den Nodes werden zusätzlich zu den Prozessen kubelet, welcher zur Kommunikation mit dem Master dient, und kube-proxy, welcher Netzwerkdienste von Kubernetes darstellt, auch noch sogenannte Pods ausgeführt[16](siehe Abbildung 3). Pods sind ein abstrakter Wrapper für einen oder mehrere Container, wobei ein Container nicht zwangsläufig ein Docker Container sein muss. In den Pods werden die einzelnen verwendeten und unterstützten Container Runtimes ausgeführt[17].

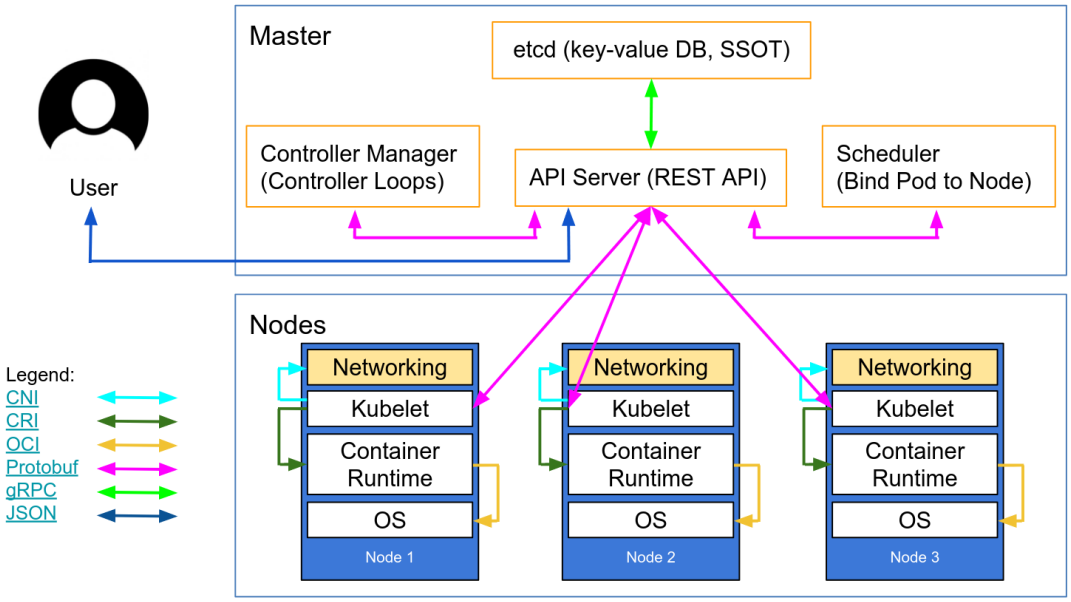

Abbildung 3: Übersicht über Kubernetes Architektur[18]

[15] [Kub20b], Vgl.
[16] Ebenda, Vgl.
[17] [Kub20c], Vgl.
[18] Kub20a

Zusätzlich zu den genannten Elementen bietet Kubernetes auch einen Service, welcher als Load-Balancer fungiert. Ein Service teilt jedem einzelnen Pod, welcher von der Kubernetes-API einer Node zugeteilt wurden ist, eine eigene IP-Adresse und einen einzigen DNS-Namen für eine Gruppe von Pods zu[19].

Dank dieser stark ausgebauten internen Prozesse und des bewusst modular gewählten Aufbaus, welcher multiple Container Runtimes unterstützt, ist Kubernetes in kurzer Zeit zum Industriestandard geworden.

2.3 DevOps

DevOps ist eine Sammlung an verschiedenen „Denkweisen, Praktiken und Tools“[20] und kann somit als eine Art Philosophie betrachtet werden. Das übergeordneten Ziel besteht darin, die angebotenen Anwendungen und Services schneller und robuster bereitzustellen. Dies wird durch eine bewusste Vermischung von Entwicklern und Mitarbeitern, die für DevOps Tätigkeiten zuständig sind, versucht zu erziehlen. Es wird auf personeller Ebene versucht, die Absprache, den Wissensaustausch und die Koordination der unterschiedlichen Abteilungen zu verbessern. Einige Unternehmen gehen sogar soweit, dass jedes Entwicklerteam einen oder mehrere DevOps-Mitarbeiter besitzt, welche zusammen mit der IT für das Deployment von Anwendungen zuständig sind. Letztendlich besteht das ultimativ Ziel darin ein cross-funktionales Team zu erhalten, wo nicht nur Kenntnisse aus Entwicklung, IT und DevOps vorhanden sind, sondern auch fließend von den Mitarbeitern beherrscht werden[21]. Dies bedeutet, dass jeder Mitarbeiter einen Bereich hat in dem er sich spezialisiert und darüber hinaus weitere Interessensbereiche.

DevOps verfolgt jedoch nicht nur zwischenmenschliche Ansätze, sondern bedient sich auch verbreiteten Praktiken zur Automatisierung von manuellen bzw. zeitaufwendigen Prozessen[22]. Dieser Ansatz steht unter anderem in Verbindung mit einer Microservice-Architektur, welche in sich geschlossene und überschaubare Features beinhaltet, die unter Verwendung von Continuous Integration (CI) und Continuous Delivery bzw. Continuous Deployment (CD) Lösungen, schneller dem Endnutzer zur Verfügung gestellt werden kön-

[19] [Kub20e], Vgl.
[20] [Ama19]
[21] [Seb20], Vgl.
[22] [Ama19], Vgl.

nen[23].

Der letzte große Bestandteil von DevOps betrifft die Überwachung und Protokollierung der eingesetzten Prozesse, Anwendungen und Infrastrukturen, die zur Verbesserung der Endbenutzererfahrung verwendet werden. Dies erleichtert es, Auswirkungen von Updates besser zu überwachen bei Problemen schneller zu handeln. Auch kann durch kontinuierliches Monitoring der CI/CD-Pipelines der Reibungsverlust zwischen Entwicklung und Operations minimiert werden, wodurch auch interne Abläufe verbessert werden.

2.4 Reproduzierbare Systeme

In der IT und der Softwareentwicklung ist Konsistenz bezüglich der Entwicklungsumgebung von großer Bedeutung. Ungleiche Systeme können Probleme bei der Weitergabe von Software und dem damit verbundenen Code verursachen. Innerhalb des Testsystem könnte der Code nicht kompilieren oder Tests schlagen aufgrund von abweichenden Versionen und Konfigurationen fehl. Es wäre schwerwiegender, wenn das Bereitstellen der Software auf dem Produktivsystem fehlschlagen würde oder nur Teile nicht funktionieren könnten.

Um diese Probleme zu lösen, wird das Konzept von Infrastructure as Code verwendet. Mit Hilfe von Code können IT-Infrastrukturen automatisch bereitgestellt und verwaltet werden. Es entfällt die manuelle Verwaltung der einzelnen Systeme bzw. Instanzen. Mit Ansible, einem IT-Mangament-Tool, kann nicht nur ein SQL Server installiert, sondern auch hinsichtlich der korrekten Konfiguration getestet werden[24].

Neben Ansible können auch andere Tools wie Terraform eingestzt werden. Dies bringt eine gute Integration von Clouddiensten der verschiedenen Cloudanbieter[25]. In dieser Arbeit wird Terraform verwendet, um einen reproduzierbaren Cluster innerhalb der Kubernetes-Engine der Google Cloud Platform zu erstellen und zu verwalten.

[23] [Eis15], Vgl.
[24] [Rou16], Vgl.
[25] [Has20], Vgl.

2.5 Continuous Integration, Continuous Delivery und Continuous Deployment

2.5.1 moderne CI/CD-Architekturen

Da bei größeren Projekten viele verschiedene Entwickler an unterschiedlichen Features arbeiten, können bei der Code Zusammenführung Problemen auftreten. Ziel von CI ist es den Code möglichst zeitnah zusammenzuführen, zu testen und anderen Entwicklern zur Verfügung zu stellen. Im besten Fall bedeutet ein zeitnahes Zusammenführen, dass Codeabschnitte mehrmals täglich gemergt werden[26].

Continuous Delivery bedeutet aus dem getesteten Code eine produktionsreife Version in einer produktionsnahen Umgebung zu erstellen. Danach greift Continuous Deployment, bei dem die Software automatisch in die Produktionsumgebung überführt werden soll. Auf diese Weise kann jede Änderung, die die Tests bestanden hat, unverzüglich an den Endnutzer ausgeliefert werden[27].

Moderne Lösungsarchitekturen, welche sich der CI und CD Ansätze bedienen, sind mehrstufig konstruiert und benutzen mehrere Tools, um den Software Development Life Cycle zu steuern (siehe Abbildung 4). Um diesen Cycle zu steuern, wird als erstes ein CI/CD-Framework benötigt, welches verschiedene Tools und Services bündelt und steuert. Dieses Framework kann den vorhanden Code aus einem Versionskontrollsystem auslesen und weiterverarbeiten. Der vorhandene Code wird durch ein Build Automation Tool mit benötigten Dependencies versorgt, gesäubert, compilliert und getestet. Bekannte Vertreter solcher Systeme sind Gradle, Grunt, Maven und Ant. Um den nun verpackten Code zu testen, werden eigene Frameworks benötigt, wie JUnit oder Mockito. In diesen wird den Entwicklern oder der Qualistätssicherung ermöglicht für jeden Teil des Codes einfache Tests zu schreiben und zu entwerfen[28].

Diese Pipeline und Tools verändern sich, wenn die Cloud native Softwarewelt angesprochen wird und nicht nur Code, sondern ganze Container zwischen den einzelnen Schritten

[26] [MTA20], Vgl.
[27] Ebenda, Vgl.
[28] [Son19], Vgl.
[29] Qua20

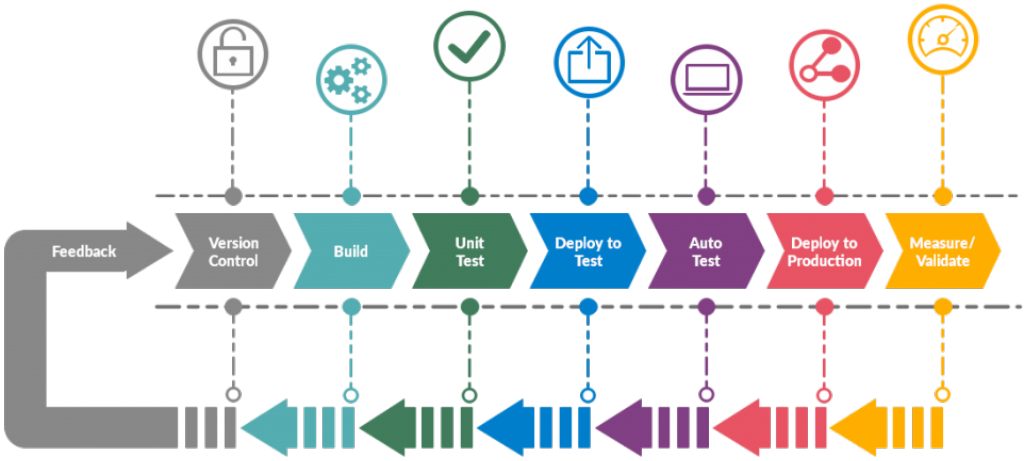

Abbildung 4: Beispielhafte DevOps-Pipeline[29]

verschoben werden.

2.5.2 GitOps

GitOps ist eine mögliche Umsetzung einer CD-Pipeline und basiert auf einer vorhandenen CI-Lösung an. Es enkoppelt das Deployment vom Bauen und Testen der Anwendung, da die eigentliche Abarbeitung durch einen Pull-/Merge-Request gestartet wird. Diese Request richtet sich gegen einen vorher definierten Branch im verwendeten Versionskontrollsystem und startet eine GitOps-Pipeline.

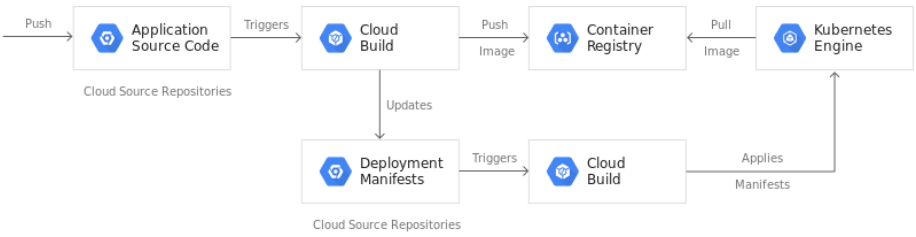

Abbildung 5: Beispielhafte GitOps-Pipeline in der Google Cloud Platform[30]

Eine beispielhafte Pipeline von der Google Cloud Platform (siehe Abbildung 5) zeigt, wie der genaue Ablauf von GitOps in einer containerbasierten Umgebung aussieht. Die Abarbeitung beginnt mit dem Push von neuem oder geänderten Source Code in ein Git Repository. In diesem Beispiel wird GitOps sogar für die CI-Pipeline verwendet. Der Push startet automatisch das Google Cloud Build, das Unit Tests ausführt, aus Containern Images baut und diese Images zur Container Registry hinzugefügt. In diesem Schritt

[30] Goo19b

wird auch das Manifest aktualisiert und entsprechende Änderungen auf einen definierten Branch gepusht. Diese Änderungen werden ebenfalls erneut registriert und das geupdatete Manifest wird auf das Kubernetes Cluster angewendet. Es wird damit der Produktionszweig und somit auch die Produktionsumgebung erneuert.

3 Ausgangssituation im Unternehmen

3.1 Historische Entwicklung

Die historische Entwicklung von DevOps und der allgemeine Prozess der Softwareentwicklung lassen sich Softwareseitig grob in zwei Bereiche unterteilen. Einerseits gibt es Ansätze zum Bündeln von Abhängigkeiten, um einen unabhängigen Einsatz und Lauffähigkeit des Codes auf den unterschiedlichen Rechnern gewährleisten zu können. Auf der anderen Seite gibt es verschiedene Lösungen zum automatischen Testen, Bauen und Bereitstellen der erstellten Software.

Java Archive (JAR)-Dateien sind ein weit verbreiteter Ansatz, um Java-Klassen und Metadaten in einem komprimierten Format zu bündeln. Für die Verwendung der JAR-Dateien wird auch eine Java Runtime Environment (JRE) benötigt. Mit dieser kann der enthaltene Code compiliert und ausgeführt werden. In den einzelnen verschiedenen Rechnerumgebungen ist somit nur ein JRE erforderlich. Alle für den Code erforderlichen Abhängigkeiten, sind in der JAR-Datei gebündelt und müssen nicht installiert oder hinzugefügt werden[31].

Ein anderer Ansatz schlägt die Verwendung des Omnibus Packages vor. Wie eine JAR-Datei bündelt dieses Paket alle wichtigen Informationen die dazu benötigt werden ein Full-Stack Projekt auszuführen[32]. Omnibus wird unter anderem von GitLab verwendet, um alle nötigen Konfigurationen für eine lokale Installation bereitzustellen[33].

Einige Unternehmen, darunter auch die dotSource GmbH, nutzen in einigen Projekten virtuelle Maschinen, um Entwicklungs- und Testumgebungen zu definieren. Beispielsweise erstellt ein Entwickler normalerweise eine rohe virtuelle Maschine und führt dann alle erforderlichen Schritte aus, um neuen Entwicklern einen sofortigen Einstieg in das Projekt zu ermöglichen. Eine entsprechende Software-Suite wie Eclipse ist normalerweise installiert und alle nötigen Einstellungen und Pakete sind konfiguriert. Das Problem bei einem

[31] [Ora20], Vgl.
[32] [Che20], Vgl.
[33] [Git20], Vgl.

solchen Ansatz sind Veränderungen an der Entwicklungsumgebung. Sollte nun ein weiteres Commandoline Programm benötigt werden, muss es jeder Entwickler selber anpassen, insofern es nicht zentral durch einen Systemadmin gepatcht werden kann.

3.2 CI/CD-Software

3.2.1 Übersicht in der dotSource GmbH

Innerhalb der dotSource GmbH wird eine Vielzahl an verschiedener Software zum Testen und Bereitstellen von Programmcode verwendet. Dieses Konzept baut auf der agnostischen Haltung der dotSource GmbH auf. Es werden lieber mehrere unabhängige Tools verwendet, welche ihre Aufgabe sehr gut erfüllen, als ein einzelnes Tool, was jeden Anwendungsfall verarbeiten kann. Das Jenkins-Tool wird normalerweise zum automatischen Testen bzw. zum Überprüfen von Tests genutzt. Zur statischen Code-Analyse wird SonarQube verwendet. Die verschiedenen Teams verwenden GitLab CI, Travis CI oder ähnliche Programme, um CI/CD-Pipelines zu steuern.

3.2.2 Jenkins

Jenkins ist ein Open-Source-Automatisierungsserver mit dem beliebige Aufgaben z.B. nach einem festen Zeitplan ausgeführt werden. Er kann dafür verwendet werden, um Unit Tests auszuführen und somit die Funktionsfähigkeit vom Programmcode zu testen oder er führt Aufgaben zum Compilieren undBbereitstellen des Programmcodes aus[34]. Bei der dotSource GmbH wird Jenkins genau für diesen Zweck eingesetzt. Jenkins dient somit als zentraler Orchestrationsort für Test- und Build-Ausführungen.

3.2.3 Selenium

Da die dotSource GmbH ursprünglich eine E-Commerce Agentur war, existieren viele Projekte rund um das Erstellen von Webshops. Auf Programmcode-Ebene können diese durch Unit Tests auf Funktionalität geprüft werden. Es müsste jedoch auch das Frontend, die Perspektive des Endbenutzers und dessen Funktionalität, getestet werden. Dies erfolgt häufig durch das Durchführen von Seleniumtests. Selenium selbst ist ein Framework zum automatisierten Testen von Weboberflächen und der Funktionen. Es startet einen oder

[34] [Jen20], Vgl.

mehrere Browser und führt in diesen verschiedene Interaktionen mit der grafischen Benutzeroberfläche durch[35]. Es kann also auf einen Link geklickt werden und dann analysiert werden, ob auf die richtige Seite weitergeleitet wurde. Zusammen mit Jenkins können diese Tests problemlos durchgeführt und ihre Ergebnisse dokumentiert werden.

3.2.4 SonarQube

Bei SonarQube handelt es sich um ein Tool zur statischen Code-Analyse. Es wird nicht auf die Funktionalität des Codes geschaut, sondern es finden nur vorher definierte Regeln zum Aussehen des Programmcodes Anwendung. In der kostenlosen und Open-Source-Version bietet SonarQube für mehrere Programmiersprachen vordefinierte Regeln an. Abhängig von den verwendeten Konventionen können auch projektspezifische Regeln hinzugefügt und getestet werden. Ziel ist es schlechte Code-Umsetzung und Bugs zu finden bzw. vorzubeugen. Innerhalb der dotSource GmbH findet SonarQube eine intensive Verwendung in Java-basierten Projekten.

3.2.5 GitLab CI und Travis CI

Bei den beiden benannten Tools handelt es sich um Tools zum Testen und Bereitstellen von Software. Im Wesentlichen unterscheiden sich beide Tools in der von ihnen gewählten Plattform. Travis CI unterstützt Github und hat sich auf Repositorys spezialisiert, welche dort gehostet sind. GitLab CI hingegen ist eine Lösung aus dem Haus der GitLab Inc. und ist für die gleichnamige Gitplatform konzeptioniert wurden.

3.3 Herausforderungen vom cloudnativen DevOps

Innerhalb der dotSource GmbH gibt es immer mehr Projekte, die nicht nur Cloud-Computing-Dienste integrieren, sondern teils komplett auf public Cloud Diensten aufbauen und Infrastruktur nur über die Cloud beziehen. Um das Ziel einer Bereitstellung in der Cloud zu erreichen, müssen einige Hürden überwunden werden. Ein Vorteil der Cloud besteht darin, dass der erstellte Service global zugänglich gemacht werden kann. Ältere und nicht Cloud-native Systeme wie Hybris, eine E-Commerce-Lösung von SAP, weisen jedoch einige Probleme auf, z.B. wenn diese in einen Kubernetes Kluster laufen. In der Hybris wird ein persistenter Speicher, Sessions und States benötigt. Insbesondere Kuber-

[35] [Wik20b], Vgl.

netes, wo es oft zum verwerfen von Pods (Containern) kommt, macht es den Entwickler Teams nicht einfach diese alten Systeme umzuziehen. Am Beispiel von Hybris kann auch gesehen werden, dass ein globales Deployment schwer umzusetzen ist, da keine verteilten Datenbanken zum Einsatz kommen können.

Darüber hinaus sollten diese Projekte nicht nur in der Cloud bereitgestellt werden. Ziel ist es auch, Build- und Testprozesse in eine Cloudumgebung zu verlagern und somit weg von on premise gehosteten Jenkinsservern hin zu im Kubernetes laufenden Tools. Wichtig bleibt allerdings, dass Cloud native nicht durch die Ausführung von Diensten in der Cloud definiert wird. Sie wird vielmehr durch die verwendeten Tools bestimmt[36]. Beispielsweise kann auch ein on premise gehosteter Kuberneteskluster eine cloud native Lösung sein, da die cloud native Technologie Kubernetes verwendet wurde. Modernes und cloud natives DevOps muss somit mit den cloud nativen Technologien und ihren Fähigkeiten umgehen können. So entstehen verschiedene Schwerpunkte:

- Cloud-nativer DevOps muss verschiedene Deploymentstrategien unterstützen. Bei der dotSource GmbH gibt es die Möglichkeit für A/B Testing, wo nur ein Teil der User die neue Version testet, und Blue/Green, wo die neue Version parallel zu der alten hochgefahren wird und die Nutzer nach und nach umgezogen werden. Dabei können die vorhanden Funktionen von Kubernetes bezüglich fortlaufender Updates und native Rollbacks instrumentalisiert werden.

- Cloud-native Tools sollten eine einfache Integration von verschiedenen Cloud Lösungen bieten. Eine direkte Integration in die Cloud mit Kubernetes oder ähnlichen Technologien wäre hier vorzuziehen. Infolgedessen können die angebotenen Dienste und ihre DevOps-Lösungen in der selben Umgebung vorhanden sein.

- Sie sollten nicht nur auf einen bestimmten Fall hin eingeschränkt sein, sondern eine größere Masse an Einsatzmöglichkeiten anbieten bzw. integrieren.

- Die DevOps-Lösungen sollten wichtige Kernfeatures, wie automatische Tests und Skriptausführung, bieten.

- Geeignete Lösungen sollten losgelöst von geografischen Beschränkungen sein und zum Deployen und Testen von Software global einsetzbar sein.

[36] [Wil19], Vgl.

Basierend auf diesen Anforderungen können verschiedene Cloud-native DevOps-Lösungen angezeigt und miteinander verglichen werden. Dabei sollte vor allem ein Augenmerk auf Universalität gelegt werden und nicht auf einzelne extrem spezifische Features.

4 Toolübersicht und -auswahl

In Bezug auf Cloud-native CI/CD-Lösungen werden vier verschiedene Tools betrachtet: Argo CD, Spinnaker, Tekton Pipelines und JenkinsX. Ein gemeinsames Merkmal der Tools ist ihr Open-Source-Charakter, allerdings sind nur die letzten drei Teil der CD Foundation, ein neues Linux Foundation-Projekt. Auch vom Alter unterscheiden sich die einzelnen Tools nur wenig. Spinnaker wurde 2015 von Netflix als quelloffenes Projekt eingeführt und stellt damit den ältesten Vertreter dar. Die restlichen Tools kamen 2018 auf. Trotz ihres kurzen Daseins bietet jede Software gute Lösungsansätze und für ihr spezialisiertes Gebiet teils sehr gute Möglichkeiten an. Es ist dementsprechend wichtig, dass einerseits jedes Tool eher für ein spezielles Gebiet konzeptioniert wurde und deslhab meistens nicht als Komplettlösung betrachtet werden kann. Anderseits bedeutet Cloud-native auch, nicht nur von einem Tool über den gesamten Prozess Gebrauch zu machen. Vielmehr sollte das richtige Tool für den beabsichtigten Gebrauch ausgewählt werden.

Spinnaker ist der Nachfolger von Asgard und bietet hauptsächlich die Möglichkeit Cloudinfrastruktur unabhängig von der genutzten Cloud zu implementieren und bereitzustellen. Dieser agnostische Ansatz wird durch eine Plug-In-Struktur im Programmcode von Spinnaker ermöglicht. Die Funktionen von Spinnaker werden unabhängig von ihrer Implementierung durch die Cloud Provider entwickelt. Nur die Plugins setzen dann die Features von Spinnaker in API-Calls der einzelnen Cloud-Plattformen um[37]. Der Kern von Spinnaker ist allerdings CD und die verbundene Unterstützung der Teams bei der Veröffentlichung oder Verwaltung ihre Software. Dazu bildet Spinnaker die einzelnen Services oder Teile einer Software als Applications ab. Diese können dann Clustern, Servergruppen, Firewalls oder Load Balancern zugeteilt werden. Zum Bereitstellen der geplanten Struktur wird ein Pipeline-Ansatz verwendet, in dem alle wichtigen Bereitstellungsschritte, wie das Bauen von Docker Images oder Aktualisieren von Registereinträgen, definiert und ausgeführt werden können. Am Ende können auch eine Reihe von Deployment Strategien eingesetzt werden. So kann mit Blue/Green oder Canary ein unterbrechungsfreies Bereitstellen von Änderungen garantiert werden[38].

[37] [Kie19], Vgl.
[38] [Spi20b], Vgl.

Zwar sind CI-Integrationen vorhanden, welche es ermöglichen auf Events von Jenkins oder Travis CI zu warten[39]. Aber diese Integrationen und GitOps-Operationen sind jedoch nur rudimentär vorhanden. Darüber hinaus ist Spinnaker eine große und komplexe Software, welche auf die einzelnen Implementierungen durch die Cloud Plugins aufbaut. Argo CD hingegen stellt einen einfachen Ansatz für das Deployment von Software auf Kubernetes dar. Während Spinnaker eine große Menge von Clouds durch ihre Plugins abdeckt, optimiert Argo CD bewusst, um einen leicht verständlichen Dienst für das Bereitstellen von Software auf Kubernetes bereitzustellen. Es verwendet sogenannte Manifest Dateien, um den aktuellen Stand der Kubernetes Kluster aufzuzeigen und eventuelle Änderungen dem Kluster verfügbar zu machen. Dabei überwacht es zwei Manifeste. Das erste Manifest befindet sich im Git-Repository der Entwickler oder am Ende einer CI-Route. Sollte sich dies ändern, wird es mit dem zweiten Manifest im entsprechenden Repository von Argo verglichen. Je nach Konfiguration werden unter anderem Änderungen am Cluster durchgeführt und das Deployment eingeleitet[40].

Durch den minimalen Aufbau von Argo CD fallen einige Features weg. Daher wird keinerlei Unterstützung bezüglich CI angeboten. Wenn solche Funktionen zum Einsatz kommen, muss weiterhin auf Jenkins, Travis oder Gitlab CI gesetzt werden. Dies bedeutet, dass alle Möglichkeiten für die statische Codeanalyse, Testen und Build entfallen[41]. Tekton hingegen bietet diese Funktionen im CI- und CD-Bereich an. Es bietet verschiedene Module, um entsprechende Funktionen zu ermöglichen. Im Kern besteht es aus sogenannten Tasks, die in verschiedene Schritte gegliedert werden können. Innerhalb einer Task können Images gebaut, Tests durchgeführt oder der Code analysiert werden. Sie können von sogenannten „PipelineResources“ Gebrauch machen und diese in ihre Abarbeitung einbeziehen. So sind Git-Repositories oder Docker-Images Ressourcen. Ein Git-Repository könnte somit als Input für einen Task dienen. Die Task klont dieses Repository lokal in den Speicher und kann dann in einem späteren Schritt Bezug auf dieses nehmen und entsprechende Befehle ausführen, um mit Hilfe einer enthaltenen Dockerfile ein Image zu erstellen. In weiteren Tasks können dann Tests und Deploymentprozesse ausgeführt werden. Diese Tasks können dann in einer durch einen „TaskRun“ in einem „PipelineRun“ ausgeführt werden. Ein „PipelineRun“ stellt dabei eine Pipeline dar welche durch verschiedene Trigger ausgeführt

[39] [Spi20a], Vgl.
[40] [Kie19], Vgl.
[41] Ebenda, Vgl.

wird. Daher bietet Tekton in einem eigenen Modul Möglichkeiten für GitOps oder ein Dashboard zur Überwachung der ausgeführten Ereignisse an[42].

Durch den Pipeline Aufbau bietet Tekton ein sehr schlankes in Kubernetes ausgeführtes Framework, welches mit prinzipiell unbegrenzten Möglichkeiten aufwartet. Es muss bedacht werden, dass z.B. für das übersichtliche Bereitstellen von Testergebnissen oder dergleicheren externe Tools von Nöten sind. Jenkins X hingegen verfügt über viele zusätzliche Funktionen, z.B. ein standardmäßig installiertes Dashboard. Es verwendet Tekton und Prow, ein Tool zum Erstellen von Kubernetes-aware Applikationen, im Hintergrund und baut auf deren Features auf[43].

Gemäß der betrachteten Features kann folgende Tabelle (siehe Tabelle 1) aufgestellt werden:

Vergleichspunkt	Spinnaker	Tekton	Jenkins X	Argo CD
Open Source	ja	ja	ja	ja
Framework Größe	groß	klein	groß	klein
GitOps	nein	ja (Zusatzmodul)	ja	ja
CI	grundlegend	ja	ja	nein
CD	ja	ja	ja	ja
Geschwindigkeit	langsam (Vgl. Argo CD)	schnell	schnell	schnell

Tabelle 1: Vergleich der verschiedenen Tools

[42] [Tek20], Vgl.
[43] [Kie19], Vgl.

5 Anforderungen an den Prototypen

Der entwickelte Prototyp soll mehrere Features unterstützen und darstellen. Anhand dieser Features soll dann die Auswertung erfolgen. Folgende Funktionen sollten bereitgestellt werden:

- Es soll die Möglichkeit bestehen den Programmcode zu testen und sich diese Tests übersichtlich anzeigen zu lassen.
- Getesteter Programmcode sollte gebaut und deployt werden können.
- Es sollte möglich sein die oberen Punkte vollständig automatisch auszuführen, z.B. druch Webhooks und GitOps Operationen.
- Der Prototyp sollte innerhalb der Cloud laufen und bereitgestellt werden können. Dabei sollten die nötigen Schritte einfach in ein Skript verpackt werden können, um ein multiples Deployment der CI/CD-Lösung sicherzustellen.
- Für die Umsetzung sollte der Ansatz von Infrastructure as Code verfolgt werden, damit Änderungen einfach und nachvollziehbar vorgenommen werden können.

6 Umsetzung eines Tekton-Prototyps

Bevor der eigentliche Tekton Prototyp erstellt werden kann muss die nötige Infrastruktur eingerichtet und zur Verfügung gestellt werden. Da es sich bei Tekton um eine cloud native Lösung handelt wird von vornherein ein cloudbasierter Ansatz verfolgt, indem auf einer Kubernetes Instanz der Google Kubernetes Engine gearbeitet wird. Die Entwicklung könnte allerdings auch lokal mit Hilfe von Minikube, einem Programm zum Hosten von einem Kubernetes Cluster auf einem einzelnen Computer, durchgeführt werden.

Neben der Google Kubernetes Engine könnte auch der Azure Kubernetes Service oder Amazons Elastic Kubernetes Service verwendet werden. Innerhalb der verwendeten Cloud Lösung muss ein neues Projekt angelegt und eingerichtet werden. Diesem wird der Name gegeben, welcher später noch zur weiteren Einrichtung wichtig sein wird. Für das Projekt werden drei verschiedene Service Accounts benötigt. Service Accounts sind spezielle Nutzer innerhalb der Google Cloud Platform, denen Rechte in Form von Rollen gewährt werden und die von den verschiedenen Anwendungen verwendet werden können. Der erste Account wird von Terraform, einem Softwaretool zur Umsetzung von Infrastructure as Code, benötigt. Dieser erfordert die Rolle des `Project Editor`, um Services erstellen und löschen zu können. Die anderen beiden dienen dem Kubernetes Cluster bzw. dessen Services. So wird ein Account benötigt mit `Storage Object Viewer`, `Monitoring Admin` und `Logging Admin` Rollen. Der Kubernetes Cluster verwendet diesen Account, um den einzelnen Nodes, realen oder virtuellen Maschinen, Zugriff auf Google Cloud Ressourcen zu gewähren. Schließlich benötigt Kaniko, ein Tool zum Bauen und Deployen von Docker-Images in einem Container, einen Service Account mit der `Storage Admin` Rolle, damit erstellte Images hochgeladen werden können.

Um den ersten Serviceaccount für Terraform nutzbar zu machen muss ein Schlüssel im JavaScript Option Notation (JSON) Format generiert und im Terraform Code als `credentials` referenziert werden. Terraform kommt mit einer eigenen, an JSON angelegten Syntax. Mit dieser wird die zu erstellende Infrastruktur genauer definiert (siehe A1). Dort muss ein Provider definiert werden, welcher eine Sammlung von Befehlen und

Ressourcen bietet. Hier wird der Google Provider genutzt, um Google Cloud Ressourcen anzulegen. Nach der Definition des Provider können verschiedene Ressourcen angelegt werden, so müssen für eine Kubernetes Instanz zwei unterschiedliche Ressourcen beschrieben werden, ein `google_container_cluster` mit dem Namen `primary` und ein `google_container_node_pool`. Beim Definieren eines Clusters ist es wichtig, den Standard Node Pool zu löschen und das Feld `min_master_version` mit einer bestimmten Version zu populieren. Die Master Version muss zum Nutzen von Tekton, mindestens 1.15.0 entsprechen. Auch ist die Angabe eines Service Accounts von Bedeutung, da dieser die Rechte der einzelnen Nodes des später erstellten Node Pools definiert. Dort wird der Account mit den Monitoring, Logging und Storage Rechten verwendet. Innerhalb der Node Pool Konfiguration ist die Referenz auf den Cluster, wie viele Nodes es geben soll und welche Ausstattung die einzelnen Nodes haben sollen, von Bedeutung. Dabei wurde für den einfachen Prototypen nur eine Node mit einer kostengünstigen CPU-Ausstattung mit einer virtuellen CPU und 3,75 Gigabyte an RAM ausgewählt.

Nun kann auf einer beliebigen Kommandozeile, vorrausgesetzt das Terraform Command Line Interface wurde installiert, der Befehl `terraform apply` ausgeführt werden. Dies veranlasst Terraform dazu nach einer `main.tf` Datei zu suchen und entsprechenden Code darin zu interpretieren. Anschließend wird ein Plan, wie in Anhang A2 zu sehen ist, generiert. Der Plan könnte mit `terraform plan -out=./plan` auch abgespeichert werden, um zu garantieren, dass er immer gleich ausgeführt wird. Im Plan selbst ist zu erkennen, wie Terraform die zwei zu erstellenden Ressourcen in Verbindung setzt, und als erstes den Cluster und dann den Node Pool erstellt. Dieses Verhalten ist durch die Referenzierung des Clusters in der Node Pool Definition ermöglicht wurden. Außerdem können hier die gesetzten Felder nochmals überprüft werden. Konfigurationen, die nicht manuell vorgenommen wurden, werden als `(known after apply)` gekennzeichnet und werden bei Ausführung mit Standardwerten gefüllt. Nach Bestätigung des Plans wird Terraform alle nötigen API-Aufrufe tätigen, um die nötigen Ressourcen zu erstellen.

Als nächstes muss der Kubernetes Kluster eingerichtet werden. Mit dem Befehl `kubectl create namespace go-web-app` wird ein Namespace angelegt, der alle Ressourcen bezüglich einer der beispielhaften Webapp beinhaltet. Namespaces sind virtuelle Cluster die alle auf dem selben physischen Cluster von Kubernetes laufen. Sie werden hauptsächlich verwendet, um eine logische Trennung zwischen verschiedenen Bereichen und Services

aufrechtzuerhalten. Der Namespace mit dem Namen `go-web-app` ist getrennt von Kubernetes eigenen Services und auch den Tekton Services. Der Programmcode bzw. die Images, aus denen die Container gebaut werden, müssen nicht direkt auf den Kubernetes Cluster hochgeladen werden. Kubernetes bietet die Möglichkeit ein Container Registry zu bestimmen aus denen die einzelnen verwendeten Images downgeloadet werden und daraus dann Container gebaut werden, welche in den Pods auf den Nodes laufen. Eine Container Registry ist für die Versionskontrolle von Images verantwortlich. Dort hochgeladene Images können von verschiedenen Apps in verschiedenen Versionen genutzt werden. Um unterschiedliche Versionen und Typen bereitstellen zu können werden Tags verwendet. Bei diesem Projekt wird Googles Container Registry verwendet.

In Anhang A3 ist die einfache Webapplikation zu sehen, welche im Kluster laufen soll. Innerhalb dieser wird ein Webserver gestartet, welcher Anfragen für den Port 8080 verarbeitet, sofern keine Umgebungsvariable mit dem Namen `Port` vorhanden ist. Dabei werden nur Anfragen an das Stammverzeichnis berücksichtigt. Parameter werden ebenfalls ignoriert.

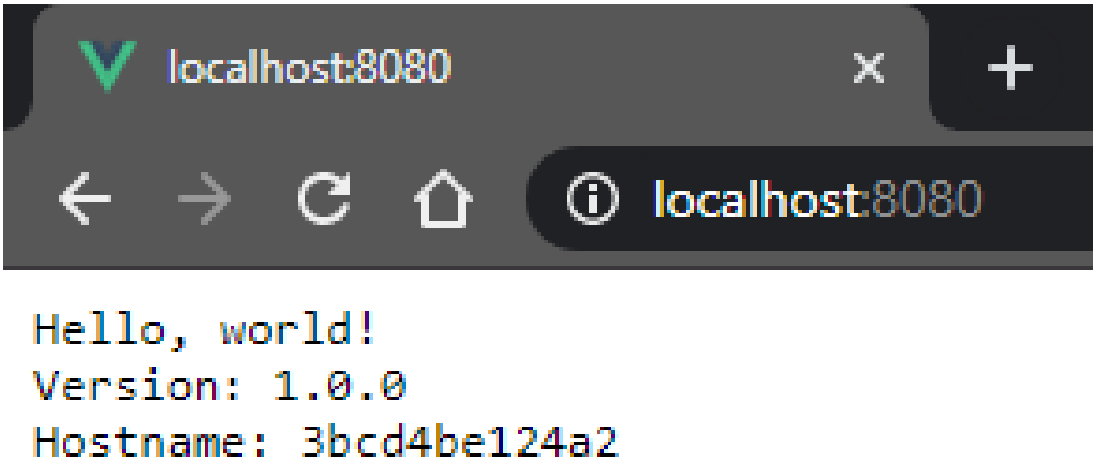

Abbildung 6: Rückgabe der Webapp

Auf Anfragen hin wird der Text „Hello World!“, wie in Abbildung 6 zu sehen ist, ausgegeben. Dieses Programm muss nun in ein Image umgewandelt werden. Hierzu wird der Befehl `docker build -t go-app .` benutzt. Das Flag `-t` setzt hier den Namen des Images und der Befehl build erzeugt aus einer gegebenen Dockerfile ein Image (siehe Anhang A5). In der Dockerfile wird die Binary aus dem Programmcode erzeugt, eine Umgebungsvariable namens Port mit dem Wert 8080 festgelegt und die erzeugte Programmbi-

nary ausgeführt, sobald ein Container aus dem gebauten Image erstellt wird. Das erstellte Image wird nun mit `docker tag go-app eu.gcr.io/tekton-prototyp/go-app:latest` getaggt. Hier muss zuerst der Hostname dann das Google Cloud Projekt und zuletzt der Name des Images angegeben werden. Da es sich um Googles Container Registry handelt muss ein `gcr.io` Host gewählt werden. Der Tag `latest` würde automatisch erzeugt werden, insofern kein anderer definiert wurde. Mit `docker push eu.gcr.io/tekton-prototyp/go-app:latest` wird das getaggte Image in die Container Registry hochgeladen (siehe Anhang 7). Das Hochladen eines Image vor dem Konfigurieren von Kubernetes ist wichtig, da sonst versucht wird ein Image aus einem nicht existenten Registry zu laden und somit ein Fehler auftritt.

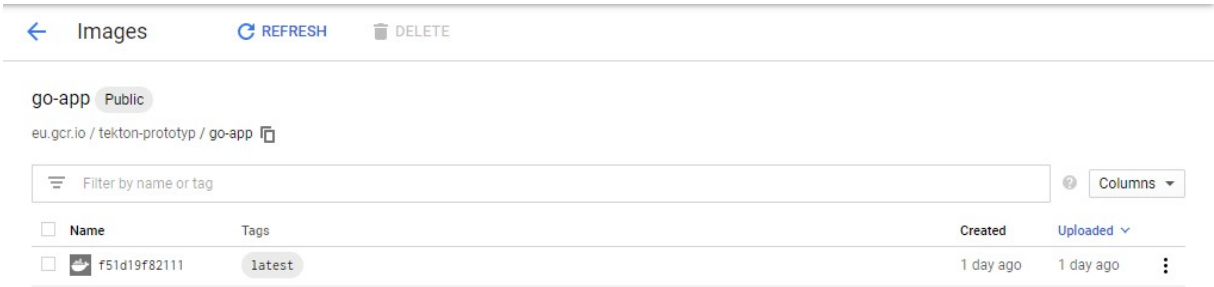

Abbildung 7: Docker-Image im Google Container Registry

Nun kann mit `kubectl apply go-app.yaml` die Konfiguration der Webapp (siehe Anhang A6) in den Kubernetes Cluster hochgeladen werden. Kubernetes verwendet dabei die vereinfachte Markupsprache YAML Ain't Markup Language (YAML). Dort muss ein Deployment erstellt werden. Ein Deployment verwaltet einen oder mehrere Pods und bestimmt wie oft ein gewisser Pod Typ vorkommen kann. Darüber hinaus wird darin mit dem Feld `image` eine Adresse zu einem Image, in diesem Falle das Image im Container Registry, und welcher Port nach außerhalb des Pods geöffnet ist definiert. Auch wird ein Name und der Namespace des Deployments sowie Selectoren für den Service erstellt. Der erstellte Service dient zur Kommunikation der einzelnen Pods, welche mit dem Selector übereinstimmen, und leitet Anfragen vom Port 80 des Services auf Port 8080 des Pods und somit des Containers weiter. Schließlich referenziert ein Ingress den Namen des Service und leitet alle Anfragen auf das Wurzelverzeichnis an den Service auf Port 80 weiter. Ein Ingress ist für Kubernetes eine Schnittstelle zum Internet und Eingangspunkt für Anfragen aller Art. Dabei werden Anfragen je nach Pfad und Protokoll an unterschiedliche Services weitergeleitet. In diesem Beispiel wird ein Nginx Ingress verwendet, der zuvor mit den folgenden Befehlen installiert wurde.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
    ingress-nginx/master/deploy/static/mandatory.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
ingress-nginx/master/deploy/static/provider/cloud-generic.yaml
```

Mit `kubectl get ingress -n go-web-app` kann der Ingress und seine automatisch von der Google Cloud zugewiesene IP-Adresse ausgegeben werden. Zu beachten bleibt, dass die Goolge Cloud für jeden erstellten Ingress automatisch einen externen Load Balancer bereitstellt, sodass eine Architektur wie in Abbildung 8 entsteht.

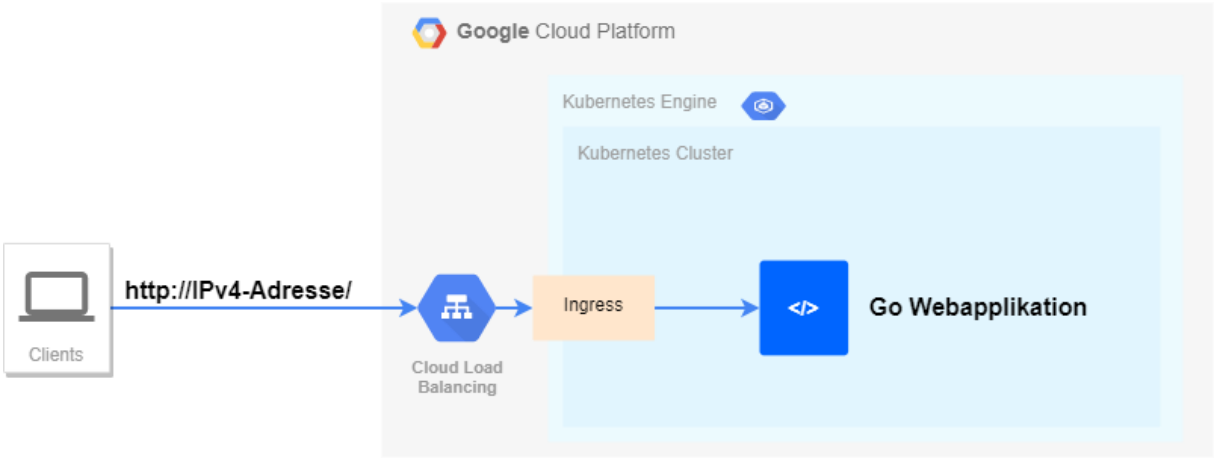

Abbildung 8: Überblick über externe Load Balancer und Ingress von Google Coud

Dadurch ist die Webapp vom Internet aus zugänglich. Anschließend muss CI/CD-Tool Tekton auf dem Cluster installiert und bereitgestellt werden.

Tekton bietet durch drei unterschiedliche Repositories verschiedene Funktionen. Bei diesem Prototyp wurde der Kern, Tekton-Pipelines, eine GitOps-Erweiterung, Tekton-Trigger und das Tekton-Dashboard implementiert. Alle drei Funktionen können durch das Hinzufügen der Konfigurations-YAML-Dateien installiert werden (siehe Anhang A7). Diese Befehle erstellen automatisch einen neuen Namespace mit dem Namen `tekton-pipelines` und mehrere Pods, in denen Tekton läuft. Bevor die prototypspezifischen YAML-Dateien hinzugefügt werden können, muss ein Secret und ein neuer Namespace für die DevOps-Funktionalität erstellt werden. Das Secret beinhaltet Konfigurationen für einen Service Account mit Schreibrechten für den Google Cloud Storage.

```
kubectl create namespace tekton-ops
```

```
kubectl create secret generic kaniko-secret -namespace=tekton-ops
-from-file=/<path-to>/kaniko.json
```

Die angegebene Datei `kaniko.json` entspricht hierbei dem Pfad zum generierten Schlüssel für den Service Account mit den entsprechenden Rechten. Jetzt können die einzelnen YAML-Dateien mit dem Befehl `kubectl apply -f PATHTOFILE`, wo `PATHOFFILE` dem Pfad zur ensprechenden Konfigurationsdatei entspricht, hinzugefügt werden. Tekton bietet verschiedene Ressourcen zum Konstruieren von DevOps-Abläufen. Der kleinste Baustein wird dort durch einen `Task` bestimmt, welcher wiederum aus einem oder mehreren Schritten besteht, welche allerdings nicht einzeln definiert werden können. `Tasks` können Parameter entgegen nehmen und Input- bzw. Output-Ressourcen, sogenannte `PipelineResources`, bestimmen.

Jeder Schritt innerhalb eines Tasks nutzt ein eigens downgeloadetes Image um einen Container zu bauen, in dem die definierten Anweisungen und Aufgaben abgearbeitet werden. Innerhalb der Schritte können Input- und Output-Ressourcen referenziert werden. Entsprechende `PipelineResources` können unter anderem Gitrepositorys und Docker Images enthalten. Jede erstellte `Task` kann in einem `Taskrun` referenziert und somit ausgeführt werden. Allerdings werden `Tasks` meistens in `Pipelines` aufgerufen. Eine `Pipeline` ist eine Ansammlung von verschiedenen Tasks denen benötigte Parameter übergeben bzw. Outputs an andere `Tasks` weitergereicht werden. Genau wie ein `Task` durch einen `Taskrun` kann eine `Pipeline` durch einen `Pipelinerun` ausgeführt werden. Beide Runs haben eigen, dass dort benötigte Ressourcen das erste Mal referenziert werden und dann nur noch per Parameter weitergereicht werden.

Der Tekton Prototyp dieser Projektarbeit besitzt zwei `Pipelines`, von denen jede jeweils einen `Task`, mit unterschiedlich vielen Schritten, aufruft. Die erste Pipeline, welche im Anhang A9 dargestellt ist, beinhaltet einen `Task`, eine `Pipeline` und einen `PipelineRun`. Innerhalb des `PipelineRuns` wird auf eine Git-Ressource verwiesen. Dies ist neben einem dazugehörigen `Secret` in Anhang A8 zu sehen. Dieses `Secret` hält Informationen zur Authentifizierung gegen das entsprechende Git-Repository. Hierbei wurde die Basic Authentification Methode ausgewählt und ein Nutzername mit einem Passwort wurden angegeben.

Das `Secret` wird dann in einer `PipelineResource` mit einem Git Repository in Verbindung gebracht. Dabei muss die URL des Repositorys angegeben werden und unter dem Punkt `revision` muss der gewünschte Branch bestimmt werden. Da die Pipeline auf einem Entwicklerbranch namens `dev` automatisch Unittests ausführen soll, muss dieser hier angegeben werden. Bevor die Ressource allerdings innerhalb der `Pipeline` genutzt werden kann, muss ein `ServiceAccount` erstellt werden (siehe Anhang A8), welcher ein entsprechendes `Secret` bereitstellt. Ebenso muss dieser `ServiceAccount` mit Rechten versehen werden, weleche mittels einer `ClusterRole` definiert werden. So kann der `ServiceAccount myapp` nicht nur Gebrauch vom `Secret` machen, sondern auch `Deployments, Pods, Services und Ingresses` erstellen und verändern. Durch zwei `RoleBindings` wird der entsprechende `ServiceAccount` mit der Rolle `myapp-cluster-role` für die Namespaces `tekton-ops` und `go-web-app` in Verbindung gebracht. Es wurden ihm somit die definierten Rechte in beiden Namespaces gewährt.

Im `PipelineRun` wird dann unter `serviceAccountName` der Name des `ServiceAccounts` zur Verfügung gestellt und eine Ressource angelegt, welche die `PipelineResource` für das Git referenziert. Nun verweist die `Pipeline`, welche beim `PipelineRun` angegeben wurde, diese Git-Ressource und reicht sie an den `Task test-go-app` weiter. Wenn Git-Ressourcen als Inputs definiert wurden, wird unter Verwendung des durch den `Service` bereitgestellten `Secrets` der ausgewählte Branch des Git-Repositorys geklont. Die Dateien sind dann unter dem Wurzelverzeichnis liegenden Ordner `workspace` und dem Namen der Input-Variable, nicht dem eigentlichen Repository Namen, zu erreichen. Der Schritt innerhalb des `Tasks` verwendet das Docker Image `tetafro/golang-gcc:1.11-alpine`, da dieses im Gegensatz zum offiziellen Golang Image eine GNU Compiler Collection mitliefert, welche für die Ausführung der Tests benötigt wird. Mit dem Befehl `go test -v` werden die vorhandenen Tests ausgeführt und ihre Ergebnisse in die Datei `test-results` geschrieben. Die Result-Datei könnte von weiteren Schritten verwendet werden. In diesem Falle wird sie mit `cat` im Log ausgegeben.

Mit der zweiten `Pipeline` soll ein Docker-Image gebaut und in das konfigurierte Container Registry hochgeladen werden. Die `Pipeline build-deploy-go` ähnelt der ersten im Aufbau, allerdings wird in diesem Fall eine neue `PipelineResource` (siehe `go-git-repo` in Anhang A8) genutzt, welche diesmal den `master` klont. Hierbei handelt es sich um eine Dupplizierung im Code. Die `URL` und die `revision` können für spätere Projekte parame-

trisiert werden, um nur eine Ressource zu haben, welche dann bei der Einbindung näher definiert wird. Innerhalb des `Tasks` werden dieses Mal zwei Schritte ausgeführt.

Im ersten Schritt wird das Image `gcr.io/kaniko-project/executor:v0.18.0` benötigt. Kaniko ist ein Open-Source-Tool von Google, welches unter anderem das Bauen und Hochladen von Docker Images innerhalb eines Containers ermöglicht. Dazu muss lediglich der Pfad zur Dockerfile, das Ziel Container Registry und Kontext, in diesem Falle der Pfad zum Ordner der Dockerfile, angegeben werden. Damit Kaniko allerdings die Möglichkeit hat das erstellte Image hochzuladen, muss ein `Volume` eingehangen werden. Dies entspricht dem per Kommandozeile erstellten Secret mit der Schreibberechtigung für den Cloud Storage. Ein `Volume` ist ein persistenter Ordner, der nicht verworfen wird, wenn der Container bzw. der Pod gelöscht wird. Es bietet somit auch eine Möglichkeit um Daten zwischen Pods bereitzustellen. In diesem Beispiel wird das `Volume` hinter dem benötigten `Secret` unter `\secret` eingebunden. Dieser Ordner ist nun innerhalb des Tasks für alle Schritte verwendbar. Allerdings muss im Schritt `build-and-push-go` noch eine Umgebungsvariable mit dem Namen `GOOGLE_APPLICATION_CREDENTIALS` definiert werden, welche den geladenen JSON referenziert. Kaniko nutzt diese Variable zur Authentifizierung gegen das Container Registry. Im Schritt danach wird nun ein Image genutzt, welches das Kubernetes Befehlszeilenkommando bereitstellt und die Rechte des `ServiceAccounts myapp` nutzt, um den `kubectl apply`-Befehl für die `go-app.yaml`(Anhang A6) auszuführen, damit das neueste Image aus dem Registry geladen werden kann. Durch die Definition von `PipelineRuns` werden beide `Pipelines` einmal ausgeführt, sobald ihre Konfigurationsdateien angewandt wurden.

Nun muss nur noch die Trigger Funktionalität hinzugefügt werden. Um Trigger nutzen zu können, muss eine Adminrolle für die Trigger erstellt werden. Diese Rolle (siehe Anhang A11) besitzt die Rechte verschiedene Ressourcen, unter anderem `PipelineRuns` zum Ausführen von `Pipelines`, zu erstellen und zu beobachten. Durch einen `ServiceAccount`, welcher durch das `RoleBinding tekton-triggers-admin-binding` an die Rolle gebunden wird, wird die Rolle im Cluster zur Verfügung gestellt. Dabei wird Kubernetes `Role-based access control`-System genutzt. Sobald die Rolle erstellt und an den Cluster propagiert wurde, kann die eigentliche Konfiguration der Trigger erfolgen.

In Anhang A12 wird ein `EventListener` definiert, welcher auf Webhook-Events hört und

diese mit Hilfe von `Interceptoren` auf verschiedene `TriggerTemplates` und `TriggerBindings` verteilt. In diesem Fall wird ein `cel interceptor` verwendet, welcher es ermöglicht mit cel-Termen Informationen aus dem Head und Body der Anfrage zu vergleichen. Nur wenn `body.ref` dem Wert `refs/heads/dev` entspricht wird das `TriggerTemplate go-test-app-triggertemplate` ausgeführt und die Go App getestet. Sollte der `body.ref`-Wert auf den `master` zeigen, wird das `TriggerTemplate go-deploy-app-triggertemplate` ausgeführt. `TriggerTemplates` fungieren als eine Art `PipelineRun`, indem sie solche erstellen. Sie rufen in ihrer Definition entsprechende `Pipelines` auf bzw. referenzieren `PipelineResources`. `TriggerBindings` wurden in diesem Beispiel nicht genutzt, allerdings fungieren sie als Ort zum Binden von Anfrage-Werten, welche später dann an die `Pipelines` weitergegeben werden können.

Der `EventListener go-app-listener` bekommt intern von Kubernetes den Namen `el-go-app-listener` und muss nun im `Ingress ingress-tekton-ops` mit dem Port 8080 referenziert werden (siehe Anhang A13). Nach dem Ausführen der entsprechenden YAML wird erneut eine externe IP-Adresse zugeteilt, welche es ermöglicht den `EventListener` von außerhalb auszuführen.

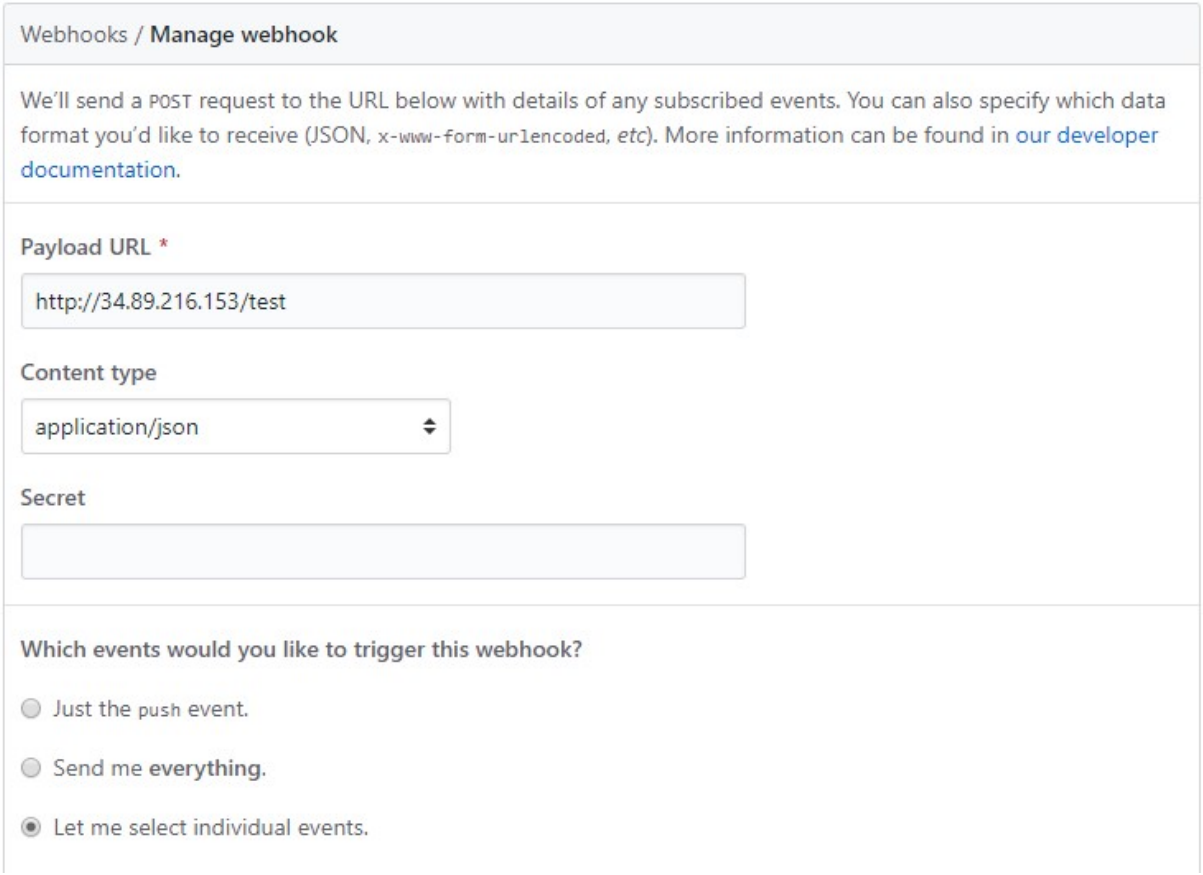

Abbildung 9: Beispielhafter GitHub-Webhook

In diesem Prototypen wurde ein GitHub-Repository und GitHub-Webhooks verwendet. Wie in Abbildung 9 zu sehen ist, muss als `Payload URL` die externe Adresse des `Ingress` und der genaue Pfad, in diesem Fall `/test`, angegeben werden. Darüber hinaus müssen noch die Fälle ausgewählt werden, wann so eine Anfrage gesendet wird. In diesem Fall, sobald Änderungen hochgeladen oder Branches zusammengeführt werden. Dann wird, wie im Anhang A14 zu sehen ist, ein Request mit dem Feld `ref` im Body geschickt. Dies wird dann im `EventListener` ausgewertet und entsprechende `PipelineRuns` gestartet.

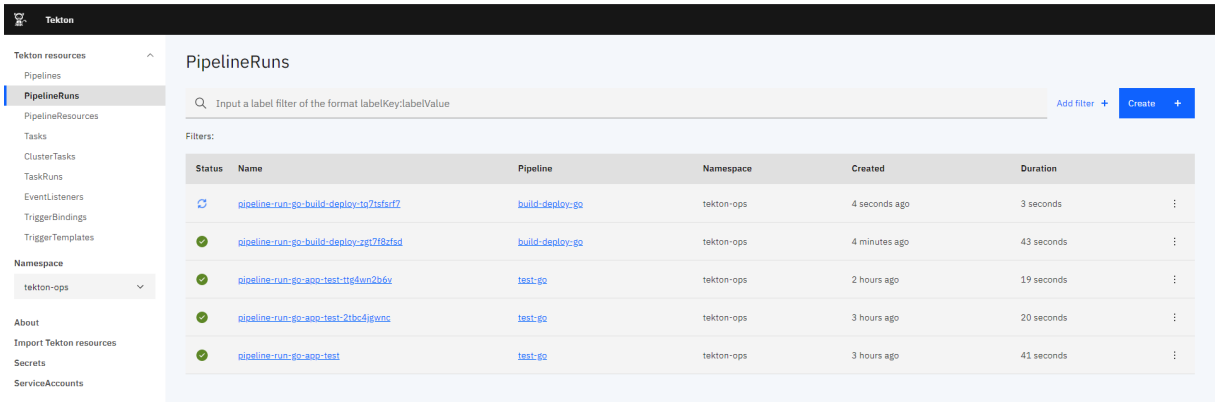

Abbildung 10: Tekton-Dashboard-Pipeline Durchläufe

Nun kann mit `kubectl proxy` eine Verbindung zum lokalen Rechner aufgebaut werden und unter der Adresse `http://localhost:8001/api/v1/namespaces/tekton-pipelines/services/tekton-dashboard:http/proxy/` das Tekton-Dashboard aufgerufen werden. Wenn nun neue Änderungen an das GitHub-Repository propagiert werden, können hier die Ergebnisse der `PipelineRuns` beobachtet werden (siehe Abbildung 10). Das Dashboard bietet Entwicklern die Möglichkeit alle konfigurierten Ressourcen geordnet anzuzeigen. Dabei kann nach Namespaces gefiltert werden, um bei größeren Projekten nur eine begrenzte Ressourcenanzahl zu sehen. Es ist auch leicht zu erkennen wie die einzelnen `PipelineRuns` erfolgreich mit einem push auf den `master` oder `dev` Branch gestartet und beendet werden.

7 Zusammenfassung

Innerhalb der Projektarbeit wurde sich, auf Basis einer theoretischen Betrachtung, mit mehreren Cloud-nativen CI/CD-Tools beschäftigt. Diese wurden kurz in ihrem Funktionsumfang beschrieben und verglichen. Darüber hinaus wurde die Situation innerhalb der dotSource GmbH betrachtet und die momentane Situation aufgezeigt. Durch die Toolbetrachtung und den Stand in der dotSource GmbH wurde Tekton mit seinem modularen Ansatz zur Bildung von CI/CD-Pipelines als zu testendes Tool ausgewählt.

Der Tekton Prototyp wurde innerhalb der gegebenen Zeit nach den vorher definierten Anforderungen konstruiert. Eine beispielhafte Bereitstellung dieses Prototypen erfolgte dann innerhalb der Google Cloud Platform.

8 Fazit

Während der Umsetzung halfen vor allem die Dokumentation von den einzelnen Tekton Repositorys und einige Beiträge von der IBM-Cloud, da diese eine gute Integration von Tekton bietet. Problematisch waren einige Gegebenheiten der Google Cloud. So werden gelöschte Service Accounts nicht wirklich entfernt, sondern über eine nicht genauer bestimmte Zeit lang gecacht. Sollte also ein Service Account mit demselben Namen, aber anderen Rechten angelegt werden, wird auf die Rechtedefinition des gelöschten gleichnamigen Accounts zurückgegriffen. Dieses nicht dokumentierte Verhalten kann zu Problemen führen, wenn die Rechte eines Accounts ausgeweitet werden sollen.

Tekton selber ermöglichte, durch seine überschaubare Anzahl an Ressourcen, einen einfachen Einstieg in die Konzeption von Pipelines. Darüber hinaus können die gegeben Ressourcen möglichst modular konzipiert werden, um eine Wiederverwendung zwischen Projekten zu ermöglichen. Durch die Verwendung von Docker Images für die einzelnen Abarbeitungsschritte ermöglicht es Tekton dem DevOps-Engineer jede nötige Anweisung auszuführen, da ein passendes Docker Image jederzeit erstellt und zur Verfügung gestellt werden kann. Tekton selbst bietet jedoch keine Option für verschiedene Deployment-Strategien. Da es aber auf Docker Images für die einzelnen Abarbeitungsschritte setzt, können dort nötige Einstellungen für z.B. A/B-Testing vorgenommen werden.

Gerade durch seinen low-level Aufbau überzeugt Tekton. Solange die nötige Version vom Kubernetes Cluster Master gegeben ist, kann es auf jedem Cluster ohne weitere große Anpassungen installiert werden. Es müssen lediglich die in der Dokumentation definierten Schritte ausgeführt werden.

Damit erfüllt Tekton den Anspruch der Reproduzierbarkeit und stellt ein gutes cloud natives CI/CD-Tools für die dotSource GmbH und Kubernetes Projekte dar.

9 Ausblick

Für den späteren Produktiveinsatz von Tekton, sollte ein Repository als eine Art Sammlung von möglichst modularen Ressourcen angelegt werden. Diese sollten so konzeptioniert werden, dass sie über mehrere Projekte hinweg Verwendung finden können und somit den Aufbauprozess beschleunigen. Dabei kann auf das Tekton Repository `Plumbing` zurückgegriffen werden, welches schon eine Sammlung von verschiedenen parametrisierten `Tasks` enthält.

Literaturverzeichnis

[Ama19] Amazon: „Was ist DevOps?“, 2019.
https://aws.amazon.com/de/devops/what-is-devops/
Abruf: 2020.01.22

[Che20] Chef: „Omnibus“, 2020.
https://github.com/chef/omnibus
Abruf: 2020.02.06

[Doc20a] Docker: „Dockerfile reference“, 2020.
https://docs.docker.com/engine/reference/builder/
Abruf: 2020.01.20

[Doc20b] Docker: „Overview of Docker Compose“, 2020.
https://docs.docker.com/compose/
Abruf: 2020.01.20

[Doc20c] Docker: „The Industry-Leading Container Runtime“, 2020.
https://www.docker.com/products/container-runtime
Abruf: 2020.01.20

[Eis15] Eisele, M.: „Applikationen schneller bereitstellen mit DevOps und Microservices“, 2015.
https://www.tecchannel.de/a/applikationen-schneller-bereitstellen-mit-devops-und-microservices,3282046
Abruf: 2020.01.22

[Git20] GitLab: „Omnibus GitLab Docs“, 2020.
https://docs.gitlab.com/omnibus/
Abruf: 2020.02.06

[Goo19a] Google: „CONTAINER BEI GOOGLE“, 2019.
https://cloud.google.com/containers/?hl=de
Abruf: 2020.01.22

[Goo19b] Google: „Continuous Delivery gemäß GitOps mit Cloud Build“, 2019.
https://cloud.google.com/kubernetes-engine/docs/tutorials/gitops-cloud-build?hl=de
Abruf: 2020.01.22

[Has20] HashiCorp: „Terraform“, 2020.
https://www.terraform.io/
Abruf: 2020.02.04

[Jen20] Jenkins: „Jenkins User Documentation“, 2020.
https://jenkins.io/doc/
Abruf: 2020.02.06

[Kie19] Kienzler, S.: „Spinnaker vs. Argo CD vs. Tekton vs. Jenkins X: Cloud-Native CI/CD“, 2019.
https://www.inovex.de/blog/spinnaker-vs-argo-cd-vs-tekton-vs-jenkins-x/
Abruf: 2020.02.13

[Kub20a] Kubernetes: „11 Ways (Not) to Get Hacked“, 2020.
https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/
Abruf: 2020.01.20

[Kub20b] Kubernetes: „Konzepte“, 2020.
https://kubernetes.io/de/docs/concepts/
Abruf: 2020.01.20

[Kub20c] Kubernetes: „Pod Overview“, 2020.
https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
Abruf: 2020.01.20

[Kub20d] Kubernetes: „Produktionsreife Container-Orchestrierung“, 2020.
https://kubernetes.io/de/
Abruf: 2020.01.20

[Kub20e] Kubernetes: „Service“, 2020.
https://kubernetes.io/docs/concepts/services-networking/service/
Abruf: 2020.01.20

[MG11] Mell, P. et al. „The Nist Definition of Cloud Computing, Recommendations of the National Institute of Standards and Technology“, National Institute of Standard und Technology, 2011

[MTA20] MT-AG: „CI, CD und DevOps“, 2020.
https://www.mt-ag.com/application-development/ci-cd-devops/
Abruf: 2020.01.20

[Ora20] Oracle: „JAR File Overview“, 2020.
https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html
Abruf: 2020.02.06

[Qua20] Quartech: „What is DevOps?“, 2020.
https://www.quartech.com/technologies/devops/
Abruf: 2020.01.22

[RK16] Radtke, M. et al. „Was ist Cloud Computing?“, 2016.
https://www.cloudcomputing-insider.de/was-ist-cloud-computing-a-563624
Abruf: 2020.01.20

[Rou16] Rouse, M.: „Infrastructure as Code (IAC)“, 2016.
https://www.computerweekly.com/de/definition/Infrastructure-as-Code-IAC
Abruf: 2020.02.04

[Seb20] Sebastian: „Das cross-funktionale Team“, 2020.
https://scrumkurs24.de/cross-funktionale-interdisziplinaere-teams/
Abruf: 2020.03.04

[Son19] Son, B.: „A beginner's guide to building DevOps pipelines with open source tools“, 2019.
https://opensource.com/article/19/4/devops-pipeline
Abruf: 2020.01.22

[Spi20a] Spinnaker: „Cloud Native Continuous Delivery“, 2020.
https://www.spinnaker.io/
Abruf: 2020.02.13

[Spi20b] Spinnaker: „Concepts“, 2020.
https://www.spinnaker.io/concepts/
Abruf: 2020.02.13

[Tek20] Tekton: „Tekton“, 2020.
https://github.com/tektoncd
Abruf: 2020.02.13

[Tur16] Turnkey, T.: „Hosting Microsoft Dynamics in the Cloud – evaluating IaaS, PaaS and SaaS“, 2016.
https://www.turnkeytec.com/hosting-applications-in-the-cloud-iaas-paas-and-saas-explained/
Abruf: 2020.01.20

[Wik20a] Wikipedia: „Kubernetes“, 2020.
https://de.wikipedia.org/wiki/Kubernetes
Abruf: 2020.01.20

[Wik20b] Wikipedia: „Selenium“, 2020.
https://de.wikipedia.org/wiki/Selenium
Abruf: 2020.02.06

[Wil19] Williams, A.: „Guide to cloud native DevOps“, The New Stack, 2019

[Wol18] Wolff, E.: „Kommentar: Docker – das Ende der Virtualisierung“, 2018.
https://www.heise.de/developer/meldung/Kommentar-Docker-das-Ende-der-Virtualisierung-3949022.html
Abruf: 2020.01.22

Anhänge

A1 Terraform Main Datei

```
provider "google" {
  version = "3.5.0"

  credentials = file("gcp_key.json")

  project = "tekton-prototyp"
  region  = "europe-west3"
  zone    = "europe-west3-a"
}

resource "google_container_cluster" "primary" {
  name     = "tekton-tf-cluster"
  location = "europe-west3-a"

  remove_default_node_pool = true
  initial_node_count       = 1

  min_master_version = "1.15.9-gke.12"

  node_config {
    service_account = "kubernetes-storage@tekton-prototyp.iam.
   gserviceaccount.com"
  }
}

resource "google_container_node_pool" "primary_preemptible_nodes" {
  name       = "tekton-tf-pool"
  location   = "europe-west3-a"
  cluster    = google_container_cluster.primary.name
  node_count = 1

  node_config {
    preemptible  = true
    machine_type = "n1-standard-1"

    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
}
```

Anhang A1: Terraform-Datei

A2 Terraform Plan

```
# google_container_cluster.primary will be created
+ resource "google_container_cluster" "primary" {
    + additional_zones            = (known after apply)
    + cluster_ipv4_cidr           = (known after apply)
    + default_max_pods_per_node   = (known after apply)
    + enable_binary_authorization = (known after apply)
    + enable_intranode_visibility = (known after apply)
    + enable_kubernetes_alpha     = false
    + enable_legacy_abac          = false
    + enable_tpu                  = (known after apply)
    + endpoint                    = (known after apply)
    + id                          = (known after apply)
    + initial_node_count          = 1
    + instance_group_urls         = (known after apply)
    + location                    = "europe-west3-a"
    + logging_service             = "logging.googleapis.com/kubernetes
  "
    + master_version              = (known after apply)
    + min_master_version          = "1.15.9-gke.12"
    + monitoring_service          = "monitoring.googleapis.com/
  kubernetes"
    + name                        = "tekton-tf-cluster"
    + network                     = "default"
    + node_locations              = (known after apply)
    + node_version                = (known after apply)
    + operation                   = (known after apply)
    + project                     = (known after apply)
    + region                      = (known after apply)
    + remove_default_node_pool    = true
    + services_ipv4_cidr          = (known after apply)
    + subnetwork                  = (known after apply)
    + zone                        = (known after apply)

    + addons_config {
        + horizontal_pod_autoscaling {
            + disabled = (known after apply)
          }

        + http_load_balancing {
            + disabled = (known after apply)
          }

        + kubernetes_dashboard {
            + disabled = (known after apply)
          }

        + network_policy_config {
            + disabled = (known after apply)
          }
      }

    + authenticator_groups_config {
        + security_group = (known after apply)
      }

    + cluster_autoscaling {
        + enabled = (known after apply)

```

```
57          + auto_provisioning_defaults {
58              + oauth_scopes    = (known after apply)
59              + service_account = (known after apply)
60            }
61
62          + resource_limits {
63              + maximum       = (known after apply)
64              + minimum       = (known after apply)
65              + resource_type = (known after apply)
66            }
67        }
68
69      + master_auth {
70          + client_certificate     = (known after apply)
71          + client_key             = (sensitive value)
72          + cluster_ca_certificate = (known after apply)
73          + password               = (sensitive value)
74          + username               = (known after apply)
75
76          + client_certificate_config {
77              + issue_client_certificate = (known after apply)
78            }
79        }
80
81      + network_policy {
82          + enabled  = (known after apply)
83          + provider = (known after apply)
84        }
85
86      + node_config {
87          + disk_size_gb      = (known after apply)
88          + disk_type         = (known after apply)
89          + guest_accelerator = (known after apply)
90          + image_type        = (known after apply)
91          + labels            = (known after apply)
92          + local_ssd_count   = (known after apply)
93          + machine_type      = (known after apply)
94          + metadata          = (known after apply)
95          + oauth_scopes      = (known after apply)
96          + preemptible       = false
97          + service_account   = "kubernetes-storage@tekton-prototyp.iam.
    gserviceaccount.com"
98          + taint             = (known after apply)
99
100         + sandbox_config {
101             + sandbox_type = (known after apply)
102           }
103
104         + shielded_instance_config {
105             + enable_integrity_monitoring = (known after apply)
106             + enable_secure_boot          = (known after apply)
107           }
108
109         + workload_metadata_config {
110             + node_metadata = (known after apply)
111           }
112       }
113
114     + node_pool {
115         + initial_node_count  = (known after apply)
```

```
116           + instance_group_urls = (known after apply)
117           + max_pods_per_node   = (known after apply)
118           + name                = (known after apply)
119           + name_prefix         = (known after apply)
120           + node_count          = (known after apply)
121           + version             = (known after apply)
122
123           + autoscaling {
124               + max_node_count = (known after apply)
125               + min_node_count = (known after apply)
126             }
127
128           + management {
129               + auto_repair  = (known after apply)
130               + auto_upgrade = (known after apply)
131             }
132
133           + node_config {
134               + disk_size_gb      = (known after apply)
135               + disk_type         = (known after apply)
136               + guest_accelerator = (known after apply)
137               + image_type        = (known after apply)
138               + labels            = (known after apply)
139               + local_ssd_count   = (known after apply)
140               + machine_type      = (known after apply)
141               + metadata          = (known after apply)
142               + min_cpu_platform  = (known after apply)
143               + oauth_scopes      = (known after apply)
144               + preemptible       = (known after apply)
145               + service_account   = (known after apply)
146               + tags              = (known after apply)
147               + taint             = (known after apply)
148
149               + sandbox_config {
150                   + sandbox_type = (known after apply)
151                 }
152
153               + shielded_instance_config {
154                   + enable_integrity_monitoring = (known after apply)
155                   + enable_secure_boot          = (known after apply)
156                 }
157
158               + workload_metadata_config {
159                   + node_metadata = (known after apply)
160                 }
161             }
162         }
163
164       + pod_security_policy_config {
165           + enabled = (known after apply)
166         }
167     }
168
169   # google_container_node_pool.primary_preemptible_nodes will be created
170   + resource "google_container_node_pool" "primary_preemptible_nodes" {
171       + cluster             = "tekton-tf-cluster"
172       + id                  = (known after apply)
173       + initial_node_count  = (known after apply)
174       + instance_group_urls = (known after apply)
175       + location            = "europe-west3-a"
```

```
      + max_pods_per_node    = (known after apply)
      + name                 = "tekton-tf-pool"
      + name_prefix          = (known after apply)
      + node_count           = 1
      + project              = (known after apply)
      + region               = (known after apply)
      + version              = (known after apply)
      + zone                 = (known after apply)

      + management {
          + auto_repair  = (known after apply)
          + auto_upgrade = (known after apply)
        }

      + node_config {
          + disk_size_gb      = (known after apply)
          + disk_type         = (known after apply)
          + guest_accelerator = (known after apply)
          + image_type        = (known after apply)
          + labels            = (known after apply)
          + local_ssd_count   = (known after apply)
          + machine_type      = "n1-standard-1"
          + metadata          = {
              + "disable-legacy-endpoints" = "true"
            }
          + oauth_scopes      = (known after apply)
          + preemptible       = true
          + service_account   = (known after apply)
          + taint             = (known after apply)

          + sandbox_config {
              + sandbox_type = (known after apply)
            }

          + shielded_instance_config {
              + enable_integrity_monitoring = (known after apply)
              + enable_secure_boot          = (known after apply)
            }

          + workload_metadata_config {
              + node_metadata = (known after apply)
            }
        }
    }
```

Anhang A2: Terraform-Ausführungsplan zur Klustererstellung auf der Google Cloud

A3 Go Main Datei

```go
package main

import (
  "fmt"
  "log"
  "net/http"
  "os"
)

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", hello)

  port := os.Getenv("PORT")
  if port == "" {
    port = "8080"
  }

  log.Printf("Server listening on port %s", port)
  log.Fatal(http.ListenAndServe(":"+port, mux))
}

func hello(w http.ResponseWriter, r *http.Request) {
  log.Printf("Serving request: %s", r.URL.Path)
  host, _ := os.Hostname()
  fmt.Fprintf(w, "Hello, world!\n")
  fmt.Fprintf(w, "Version: 1.0.0\n")
  fmt.Fprintf(w, "Hostname: %s\n", host)
}
```

Anhang A3: Hauptdatei der Go-Applikation

A4 Go Test Datei

```
1  package main
2
3  import (
4    "net/http"
5    "net/http/httptest"
6    "strings"
7    "testing"
8  )
9
10 func TestGetRoot(t *testing.T) {
11   req, err := http.NewRequest("GET", "/", nil)
12
13   if err != nil {
14     t.Fatal(err)
15   }
16
17   rr := httptest.NewRecorder()
18   handler := http.HandlerFunc(hello)
19   handler.ServeHTTP(rr, req)
20
21   if !strings.Contains(rr.Body.String(), "Hello, world!") {
22     t.Fatal("Handler returned unexpected body!")
23   }
24 }
```

Anhang A4: Datei, die alle Tests für die Go-App beinhaltet

A5 Dockerfile

```
FROM golang:1.8-alpine
ADD . /go/src/ds-src
RUN go install ds-src

FROM alpine:latest
COPY --from=0 /go/bin/ds-src .
ENV PORT 8080
CMD ["./ds-src"]
```

Anhang A5: Dockerfile zum Erstellen eines Images aus der Go-App

A6 Konfigurationsdatei Go App

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-web
  namespace: go-web-app
spec:
  selector:
    matchLabels:
      app: go
      tier: web
  template:
    metadata:
      labels:
        app: go
        tier: web
    spec:
      containers:
        - name: go-app
          image: eu.gcr.io/tekton-prototyp/go-app:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: go-web-service
  namespace: go-web-app
spec:
  selector:
    app: go
    tier: web
  ports:
    - port: 80
      targetPort: 8080
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-go-web
  namespace: go-web-app
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
    - http:
        paths:
          - path: /
            backend:
              serviceName: go-web-service
              servicePort: 80
```

Anhang A6: Kubernetes-Konfigurationsdatei für die Go-App

A7 Tekton-Installations Skript

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml &&
kubectl apply --filename https://github.com/tektoncd/dashboard/releases/download/v0.5.1/tekton-dashboard-release.yaml &&
kubectl apply --filename https://storage.googleapis.com/tekton-releases/triggers/latest/release.yaml
```

Anhang A7: Befehle zum Installieren von Tekton

A8 Tekton Pipeline Ressources und Service Account

```
apiVersion: v1
kind: Secret
metadata:
  namespace: tekton-ops
  name: go-git-basic-user
  annotations:
    tekton.dev/git-0: https://github.com
type: kubernetes.io/basic-auth
stringData:
  username: YOURGITHUBUSERNAME
  password: YOURGITHUBPASSWORD
---
# Resources
# Git Connection Master
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  namespace: tekton-ops
  name: go-git-repo
spec:
  type: git
  params:
    - name: url
      value: YOURGITREPOURL
    - name: revision
      value: master
---
# Git Connection DEV
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  namespace: tekton-ops
  name: go-git-repo-dev
spec:
  type: git
  params:
    - name: url
      value: YOURGITREPOURL
    - name: revision
      value: dev
---
# Service Accounts implement Resources
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: tekton-ops
  name: myapp
secrets:
  - name: go-git-basic-user
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: myapp-cluster-role
rules:
  - apiGroups: ["extensions", "apps", ""]
    resources:
      - services
```

```
      - deployments
      - pods
      - ingresses
    verbs:
      - get
      - create
      - update
      - patch
      - list
      - delete
      - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tekton-ops-role-binding
  namespace: tekton-ops
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: myapp-cluster-role
subjects:
  - kind: ServiceAccount
    name: myapp
    namespace: tekton-ops
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: go-web-app-role-binding
  namespace: go-web-app
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: myapp-cluster-role
subjects:
  - kind: ServiceAccount
    name: myapp
    namespace: tekton-ops
```

Anhang A8: Konfiguration für GitHub Secrets und Service Account

A9 Tekton Test-Pipeline

```yaml
apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  namespace: tekton-ops
  name: test-go-app
spec:
  results:
    - name: test-results
      description: Failure and success results of the included tests.
  inputs:
    resources:
      - name: git-repo
        type: git
  steps:
    - name: test-go
      image: tetafro/golang-gcc:1.11-alpine #golang:1.14.0-alpine3.11
      script: |
        #!/bin/ash
        cd /workspace/git-repo/go-app
        go test -v > /tekton/results/test-results
        cat /tekton/results/test-results
---
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  namespace: tekton-ops
  name: test-go
spec:
  resources:
    - name: go-git-repo-dev
      type: git
  tasks:
    - name: test-app
      taskRef:
        name: test-go-app
      resources:
        inputs:
          - name: git-repo
            resource: go-git-repo-dev
---
apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  namespace: tekton-ops
  name: pipeline-run-go-app-test
spec:
  serviceAccountName: myapp
  pipelineRef:
    name: test-go
  resources:
    - name: go-git-repo-dev
      resourceRef:
        name: go-git-repo-dev
```

Anhang A9: Pipeline zum Ausführen von Unit-Tests

A10 Tekton Deploy Pipeline

```
# Tasks
# Task Write to File
apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  namespace: tekton-ops
  name: build-deploy-go-app
spec:
  inputs:
    resources:
      - name: git-repo
        type: git
  steps:
    - name: build-and-push-go
      image: gcr.io/kaniko-project/executor:v0.18.0
      command:
        - /kaniko/executor
      args:
        - --dockerfile=/workspace/git-repo/go-app/Dockerfile
        - --destination=eu.gcr.io/tekton-prototyp/go-app:latest
        - --context=/workspace/git-repo/go-app/.
      volumeMounts:
        - name: kaniko-secret
          mountPath: /secret
      env:
        - name: GOOGLE_APPLICATION_CREDENTIALS
          value: /secret/kaniko.json
    - name: deploy-go
      image: lachlanevenson/k8s-kubectl
      command: ["kubectl"]
      args:
        - "apply"
        - "-f"
        - "/workspace/git-repo/kube/go-web-app/go-app.yaml"
  volumes:
    - name: kaniko-secret
      secret:
        secretName: kaniko-secret
---
# Pipeline
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  namespace: tekton-ops
  name: build-deploy-go
spec:
  resources:
    - name: go-git-repo
      type: git
  tasks:
    - name: test-app
      taskRef:
        name: build-deploy-go-app
      resources:
        inputs:
          - name: git-repo
            resource: go-git-repo
---
```

```
# Pipelinerun
apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  namespace: tekton-ops
  name: pipeline-run-go-build-deploy
spec:
  serviceAccountName: myapp
  pipelineRef:
    name: build-deploy-go
  resources:
    - name: go-git-repo
      resourceRef:
        name: go-git-repo
```

Anhang A10: Pipeline zum Bauen und Deployen der App

A11 Tekton Trigger Admin

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tekton-triggers-admin
  namespace: tekton-ops
rules:
  - apiGroups:
      - tekton.dev
    resources:
      - eventlisteners
      - triggerbindings
      - triggertemplates
      - pipelineresources
    verbs:
      - get
  - apiGroups:
      - tekton.dev
    resources:
      - pipelineruns
      - pipelineresources
    verbs:
      - create
  - apiGroups:
      - ""
    resources:
      - configmaps
    verbs:
      - get
      - list
      - watch
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tekton-triggers-admin
  namespace: tekton-ops
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tekton-triggers-admin-binding
  namespace: tekton-ops
subjects:
  - kind: ServiceAccount
    name: tekton-triggers-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: tekton-triggers-admin
```

Anhang A11: Datei zum erstellen einer Adminrolle für den Trigger

A12 Tekton Trigger

```yaml
apiVersion: tekton.dev/v1alpha1
kind: TriggerTemplate
metadata:
  name: go-test-app-triggertemplate
  namespace: tekton-ops
spec:
  resourcetemplates:
    - apiVersion: tekton.dev/v1alpha1
      kind: PipelineRun
      metadata:
        generateName: pipeline-run-go-app-test-$(uid)
        namespace: tekton-ops
      spec:
        serviceAccountName: myapp
        pipelineRef:
          name: test-go
        resources:
          - name: go-git-repo
            resourceRef:
              name: go-git-repo-dev
---
apiVersion: tekton.dev/v1alpha1
kind: TriggerTemplate
metadata:
  name: go-deploy-app-triggertemplate
  namespace: tekton-ops
spec:
  resourcetemplates:
    - apiVersion: tekton.dev/v1alpha1
      kind: PipelineRun
      metadata:
        generateName: pipeline-run-go-build-deploy-$(uid)
        namespace: tekton-ops
      spec:
        serviceAccountName: myapp
        pipelineRef:
          name: build-deploy-go
        resources:
          - name: go-git-repo
            resourceRef:
              name: go-git-repo
---
apiVersion: tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: go-app-listener
  namespace: tekton-ops
spec:
  serviceAccountName: tekton-triggers-admin
  triggers:
    - name: test-go-app-trigger
      interceptors:
        - cel:
            filter: "body.ref == 'refs/heads/dev'"
      template:
        name: go-test-app-triggertemplate
    - name: deploy-go-app-trigger
      interceptors:
```

```
59          - cel:
60              filter: "body.ref == 'refs/heads/master'"
61        template:
62          name: go-deploy-app-triggertemplate
```

Anhang A12: Definition des Dateitrigger und des Event Listeners

A13 Tekton Trigger Ingress

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-tekton-ops
  namespace: tekton-ops
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
    - http:
        paths:
          - path: /test
            backend:
              serviceName: el-go-app-listener
              servicePort: 8080
```

Anhang A13: Datei, die einen Ingress für die Trigger erstellt

A14 GitHub Webhook Payload

Recent Deliveries

242e7d06-62b8-11ea-930f-827f70449847 2020-03-10 11:16:00 ...

Request Response 201 Redeliver Completed in 0.3 seconds.

Headers

```
Request URL: http://34.89.216.153/test
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/728df4e
X-GitHub-Delivery: 242e7d06-62b8-11ea-930f-827f70449847
X-GitHub-Event: push
```

Payload


```
{
  "ref": "refs/heads/master",
  "before": "1c922c63a203327ec8f24c2c56e9c8e996c227bd",
  "after": "2d00567e78442eb4f9d598543cbc722fe45fbd58",
  "repository": {
    "id": 245228068,
    "node_id": "MDEwOlJlcG9zaXRvcnkyNDUyMjgwNjg=",
    "name": "tekton-ds",
    "full_name": "MaxSkoff/tekton-ds",
    "private": true,
    "owner": {
      "name": "MaxSkoff",
      "email": "33379098+MaxSkoff@users.noreply.github.com",
      "login": "MaxSkoff",
      "id": 33379098,
      "node_id": "MDQ6VXNlcjMzMzc5MDk4",
      "avatar_url": "https://avatars1.githubusercontent.com/u/33379098?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/MaxSkoff",
      "html_url": "https://github.com/MaxSkoff",
      "followers_url": "https://api.github.com/users/MaxSkoff/followers",
```


Abbildung A14: Head und Teile des Body einer GitHub Webhook-Anfrage

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit mit dem Thema:

   „Deployment und Continuous Integration mit Containerisierung in der Cloud“

   ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Projektarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Osterfeld, den 18. März 2020	[redacted]
.....	.....
Ort, Datum	Unterschrift