Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

STEFAN WAGNER
BACHELOR THESIS

KONZEPTION UND IMPLEMENTIERUNG EINER CONTINUOUS DEPLOYMENT PIPELINE FÜR CMSUITE

Eingereicht am 14. Januar 2019

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.; Maximilian Capraro M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 14. Januar 2019

License

Erlangen, 14. Januar 2019

Abstract

A wide spread concept of minimizing expenditure is automation, which is often applied in software development. Especially working with agile priciples based on rapid feedback, short productioncyles and frequent changes requires an automated development. Through an central instance the developer is able to perform standardized tests and deploy applications without environment dependencies. New features can be used instantly and reverted to previous versions without a workintense manual configuration.

At interaction with version management a continuous integration and deployment pipeline provides a tool to control the development process from codechanges to finished products. A deployment to kubernetes includes scalability and extended functionality to the deployment process. The imlementation of this pipeline and configuration of kubernetes will be part of this thesis.

Zusammenfassung

Ein weit verbreitetes Konzept zur Aufwandsminimierung ist Automatisierung, diese wird auch in der Softwareentwicklung häuftig angewendet. Besonders in der Softwareentwicklung nach agilen Prinzipien, welche auf schnellem Feedback, kurzen Produktionszyklen und häufigen Änderungen beruht ist ein automatisierter Entwicklungsprozess elementar. Durch eine zentrale Instanz können einheitlich Tests ausgeführt und fertige Anwendungen zum Review bereitgestellt werden ohne Umgebungsabhängigkeiten berücksichtigen zu müssen. Das automatische Ausliefern von geprüfter Software auf ein Produktivsystem bietet den Vorteil Neuentwicklungen sofort nutzen zu können bei gleichzeitiger Möglichkeit bei Problemen auf eine frühere Version zurückzugreifen ohne zeit- und arbeitsintesive Konfigurationen.

Im Zusammenspiel mit einer Versionsverwaltung bietet eine Continuous Integration und Deployment Pipeline dem Entwickler ein Werkzeug zur Kontrolle des Entwicklungsprozesses von der ersten Codeänderung bis zum fertigen Produkt. Dabei hat er die Möglichkeit die Auswirkungen von Änderungen in einer standartisierten und abgeschlossenen Umgebung zu evaluieren. Ein Deployment zu Kubernetes ist aufgrund der Skalierbarkeit und des erweiterten Funktionsumfangs der nächste logische Schritt zum lauffähigen System. Die Umsetzung der Pipeline sowie Installation von Kubernetes zur Deploymentinstanz und entsprechende Konfiguration ist Teil dieser Arbeit.

Inhaltsverzeichnis

1 Anforderungen

1.1 Notwendigkeit der Arbeit

Die Collaboration Management Suite (CMSuite) ist eine Software die von der Professur für OpenSource Software zur Messung und Bewertung von InnerSource Projekten von Firmen entwickelt wird. Dabei arbeiten mehrere Mitarbeiter ständig an verschiedenen Teilen der Softwareimplementierung mit. Die Zahl der Mitarbeiter macht es nötig möglichst einfache und automatisierte Abläufe in der Softwareentwicklung und Auslieferung von lauffähigen Versionen zu etablieren. Gängige Praxis zum gemeinschaftlichen Arbeiten an Software sind Versionsverwaltungstools wie Git, AWSCodeCommit oder SVN. Diese dienen zum Protokollieren und Archivieren von Änderungen sowie der Koordinierung von Zugriffen. Damit wird das gleichzeitige Arbeiten vieler Entwickler innerhalb eines Projekts auf unterschiedlichen Versionszweigen ermöglicht. Diese Versionszweige (Branches) können nun lokal weiterentwickelt, getestet und in das Gesamtprojekt aufgenommen werden. Das lokale Testen birgt allerdings das Risiko von Umgebungsabhängigkeiten und komplizierter, möglicherweise fehlerhafter Testausführung. Außerdem ist oft die Entwicklungsumgebung nicht geeignet, um die fertige Software auszuführen und das Ergebnis zu evaluieren. Daher wird ein für alle Entwickler gleiches System benötigt, welches Zugriff auf die Änderungen und die Möglichkeit einer unabhängigen Beurteilung des Fortschritts bietet. Um das zu erreichen wird sich diese Arbeit mit der Continuos Integration und Deployment für CMSuite beschäftigen und dabei eine Pipeline erstellen, welche als letzten Schritt das Ausliefern von CMSuite auf ein Kubernetes Cluster hat.

1.2 Anforderungen

Die Anforderungen an die Arbeit ergeben sich aus den Anwendungsfällen der Pipeline: Zum einen für Entwickler, die automatisiert testen und ihre Änderungen am fertigen Produkt sehen möchten, zum anderen für mögliche Kunden, welche von Anfang an die laufende Software ausprobieren und nutzen wollen.

Dementsprechend wurden folgende Anforderungen definiert:

- 1. Funktionalität von Jenkins auf Gitlab übertragen

  Unter Funktionalität versteht sich die Struktur der Jenkins Pipeline mit den drei logischen Stufen Commit, Acceptance und Production. Außerdem sollen die Jobs der Stages Commit und Acceptance erhalten bleiben und weiterhin parallel laufen. Eine Anzeige über den Status der Pipeline, durchgelaufene Jobs, abgebrochene Jobs und laufende Jobs ist nötig. Wie in Jenkins muss vor den Stages ein Environment gestartet werden um die Jobs erst starten zu können.

- 2. Tests automatisiert ausführen

  Automatisiert bedeutet hier, dass bei jeder Änderung des Repositorys auf Gitlab ohne zutun des Entwicklers eine neue Pipeline mit den Stufen aus Anforderung 1. und 5. angestoßen wird. Die Tests sollen dabei dem neuesten Repositorystand entnommen werden. Die Tests müssen für jeden Fork und alle Branches der Forks laufen.

- 3. Auslieferung auf Kubernetes anstoßen

  Das Deployment soll wie auch die Tests bei jeder Repository Änderung neu erstellt werden. Das beinhaltet das Löschen des alten Deployments, Pullen der aktuellsten Images für den Branch und Erstellen entsprechender Konfigurationsdateien.

- 4. Ausführen von CMSuite auf Kubernetes

  Das Deployment muss auf einem Kubernetes Cluster oder einer gleichwertigen Instanz dessen laufen. Es muss eine entsprechende Netzwerkkonfiguration geben, sodass der Entwickler Zugriff auf den Webclient von CMSuite hat. Vorraussetzung für diesen Zugriff muss sein, dass sich Kubernetes und der Entwickler im gleichen Netz befinden. Das Deployment muss unabhängig von möglichen anderen Deployments laufen und muss frei von Seiteneffekten sein sowie auch selbst keine verursachen können.

- 5. Geschwindigkeit genauso gut wie Jenkins

  Die ersten zwei Stufen Commit und Acceptance mit den zugehörigen Jobs müssen, sofern die Jobs dieser Stufen in der Gitlabpipeline nicht wesentlich erweitert werden, mindestens genauso schnell durchlaufen wie in der Jenkinspipeline. Dabei sind gleiche Hardwarevorraussetzungen beim Server auf dem die Jobs ausgeführt werden herzustellen oder die Messung entsprechend zu skalieren. Neue oder deutlich erweiterte Stufen werden in die Messung nicht einbezogen.

- 6. Im Fehlerfall Logs ausgeben

Die Logs beziehen sich zum einen auf die Testlogs der ersten zwei Stufen. Dabei müssen die Logs im gleichen Umfang wie auf Jenkins bereitgestellt werden. Zum Zweiten müssen im Fehlerfall auftretende Logs des Deployments ausgegeben werden. Diese Logs sollen zum einen die Logs des Deployments umfassen, zum anderen die Logs aller fehlerhaften Container. Im Erfolgsfall wird davon ausgegangen, dass Logs für den Entwickler nicht nötig sind.

- 7. Tests und Deployment wiederholbar

  Die Wiederholbarkeit bezieht sich auf einzelne Jobs, egal ob durchgelaufene oder abgebrochene, sowie auf die gesamte Pipeline. Eine von Seiteneffekten freie Umgebung ist bei der Wiederholung herzustellen. Das Deployment muss durch die Wiederholung komplett gelöscht und wieder gestartet werden.

- 8. Einfache Benutzbarkeit für Entwickler

  Die einfache Benutzbarkeit wird definiert durch einen möglichst kleinen Arbeits und Zeitaufwand für verschiedene wichtige Handlungen des Entwicklers die im üblichen Entwicklungsprozess ausgeführt werden müssen. Außerdem sollen diese Handlungen möglichst intuitiv durchgeführt werden können, um mögliche Einarbeitungszeiten zu minimieren.

1.3 Evaluationsmetrik der Anforderungen

Die Evaluationsmetriken für die Anforderungen gestalten sich teilweise kompliziert da die Funktionalität sehr auf dem Benutzerempfinden u.a. der Entwickler beruht. Um dies zu vermeiden wurde versucht die Anforderungen möglichst quantitativ messbare Matriken auszudrücken.

- 1. Funktionalität von Jenkins auf Gitlab übertragen

  Anzahl der Stages und Jobs auf Jenkins im Verhältnis zur Anzahl der nachimplementierten Stages und Jobs auf Gitlab, optimal ist ein Implementierungsgrad von 100%.

- 2. Tests automatisiert ausführen

  Die Verhältnis der tatsächlich ausgeführten zu allen möglichen Tests die bei jeder Änderung des Onlinerepositories, also push, merge und commit ausgeführt werden können. Gezählt werden die Tests für das CMSuite Repository[1] und alle Forks.

- 3. Auslieferung auf Kubernetes anstoßen

[1] https://mojo-forge.cs.fau.de/cmsuite/cmsuite

Bei jeder Repository Änderung wird das alte Deployment gelöscht, neue Images gepulled und das neue Deployment durch entsprechende Konfigurationsdateien angestoßen.

- 4. Ausführen von CMSuite auf Kubernetes

  Zahl der tatsächlich ausgeführten Branches von CMSuite und den Branches aller Forks im Verhältnis zur Gesamtanzahl selbiger.

- 5. Geschwindigkeit genauso gut wie Jenkins

  Prozentualer Vergleich der Laufzeit von Jenkins und Gitlab unter vergleichbaren Hardwarevorraussetzungen sowie gleichen oder nicht deutlich erweiterten Jobs, erreicht werden sollen 100%. Vom Vergleich auszuschließen ist der Deploy.

- 6. Im Fehlerfall Logs ausgeben

  Zahl der erstellten Logs im Fehlerfall bei den möglichen Logs für Jobs, das gesamte Deployment und jeden deployten Container. Logs werden nur als erstellt gewertet wenn diese für den Entwickler unkompliziert einsehbar sind.

- 7. Tests und Deployment wiederholbar

  Zahl der wiederholbaren Aktionen: alle Jobs, alle Stages, gesamte Pipeline, Deploy.

Da einfache Benutzbarkeit eine subjektive und nicht objektiv bewertbare Einschätzung ist musste diese Anforderung in repräsentative und quantitative Eigenschaften umgewandelt werden. Dafür dient die folgende Auflistung von zu erfüllenden Zielen. Die Benutzbarkeit wird hier in Klickanzahl von einem Startpunkt zu einem Zielpunkt definiert da dies gut der dafür aufzuwendenden Zeit entspricht. Die Auflistung stellt Optimalziele dar, jeweils ein Klick mehr ist das zu erfüllende Sollziel und je drei Klicks mehr stellt ein Minimalziel als Grenze der sinnvollen Benutzbarkeit dar.

- Von Commit in zwei Klicks zur Pipeline
- Von Commit in zwei Klicks zur Demoinstanz
- Von Merge Request in zwei Klicks zur Demoinstanz
- Von Commit in zwei Klicks zum Log eines Jobs
- Von Commit in zwei Klicks zum Log des Deployments
- Von Productionbranch in zwei Klicks zum Production Deployment
- Vom Projekt in zwei Klicks zum Production Deployment

- Von Branch zu aktuellem Pipelinestatus in zwei Klicks

1.4 Technische Anforderungen

Im folgenden wird eine elementare Aufstellung von technischen Notwendigkeiten gegeben welche nötig sind um die oben genannten Anforderungen hinsichtlich der Funktionalität umzusetzen.

- Zum Ersten wird eine Gitlabinstanz benötigt auf der die neue Pipeline läuft. Hier liegt es nahe das schon als Repository zur Versionsverwaltung verwendete Gitlab zu verwenden, ein zusätztliches Gitlab nur für die Pipeline würde keinen Sinn ergeben.
- Als Gegenstück zur Onlineinstanz von Gitlab wird eine Ausführungsumgebung für einen Gitlabrunner benötigt. Wo dieser läuft ist nicht von Belang, die Hardware sollte aber aus Performancegründen mindenstens der des Jenkinsserver entsprechen.
- Um das Kubernetes Cluster zu hosten wird ein Server benötigt. Die Hardware sollte dabei imstande sein mindestens 25 CMSuite Instanzen auszuführen, dieser Wert ergibt sich aus 5 Mitarbeitern multipliziert mit je 5 Branches die dort auszuführen sind. Da die tatsächlichen Werte abweichen können muss hier ein entsprechend leistungsfähiges System verwendet werden.

2 Konzept und Design

2.1 Vergleich von Gitlab und Jenkins

Als erste Konzeptentscheidung galt es bei der Implementierung der Countinuos Deployment Pipeline zwischen Gitlab und Jenkins zu wählen. Sollte die Wahl auf Gitlab fallen müsste hier die bestehende Pipeline von Jenkins auf Gitlab migriert werden.

Jenkins wurde bereits vorher für die ersten Schritte der Pipeline eingesetzt und war dort schon erprobt. Jenkins wird oft im Bereich gewerblicher Softwareentwicklung eingesetzt und bietet weitreichende Funktionalitäten v.a. über Plugins an. Als einer der größten Entwickler von Continuous Integration(CI) und Continuous Deployment(CD) Softwareintegrationen bietet Jenkins den Vorteil einer hohen Verbreitung im induestriellen Entwicklerumfeld und die damit verbundene oft geringe Einarbeitungszeit. Gitlab erfreut sich seit der Gründung 2011 zunehmender Beliebtheit im OpenSource Bereich und vereint ein Versionsverwaltungstool mit einer CI/CD Pipeline.

Gitlab wird unter Studenten häufig verwendet, da eigenes Hosting ohne Lizenzgebühren möglich ist sowie die Bedienung der Versionsverwaltung Teil üblicher Lehrpläne ist. Daher wurde es über die letzten Jahre zum "Quasistandart" als Versionsverwaltungstool. Zum Beispiel wurde es bereits vor dieser Arbeit zur Versionsverwaltung des CMSuite Codes eingesetzt.

Um fundiert entscheiden zu können, welches der beiden Systeme geeigneter zur Umsetztung des Contiunuous Deployments ist wurde mit Tabelle 2.1 eine Liste von Anforderungen aufgestellt, welche mindestens erfüllt werden müssen, um alle benötigten Funktionen umsetzen zu können. Vor allem geht es hier um Konfigurierungsmöglichkeiten der Pipeline.

Insgesamt kann festgestellt werden, dass Gitlab und Jenkins alle benötigten Features mitbringen, oft sind die Notationen unter Gitlab intuitiver und lesbarer. Letztendlich entschied ich mich zur Weiterentwicklung der Pipeline für Gitlab, da dies eine Vereinfachung der Arbeitsprozesse bot und sich mehr Konfigurationsmöglichkeiten boten.

Feature	Jenkins	Gitlab
Umgebungvariablen spezifizieren	✓	✓
Stages definieren	✓	✓
Parallel Stages ausführen	✓ explizite Anweisung	✓ implizit
Artefakte an Nutzer ausliefern	✓	✓
Jobs bedingt ausführen	✓	✓
Manuell Jobs triggern	✓ Plugin	✓ Anweisung
Kubernetes integrieren	✓ ein Cluster	✓ mehrere Cluster
Environment vorbereiten	✓ durch zusätzliche Stage	✓ innerhalb jedes Jobs
JUnit report	✓ aber nicht genutzt	✓
Partial build Variable definieren	✓	✗ möglich per Script und API Zugriff
Auf Ausführungsumgebung Software installieren	✓	✓
Auf Branch und Forknamen zugreifen	✓	✓

Tabelle 2.1: Featurevergleich zwischen Gitlab und Jenkins

2.2 Continuos Integration von CMSuite

2.2.1 Gitlab Runner

Der Gitlab Runner ist das Gegenstück zu der auf dem Webinterface von Gitlab dargestellten Pipeline. In der Pipeline wird definiert, welche Aktionen ausgeführt werden sollen, der Runner sorgt dafür dass diese Befehle tatsächlich ausgeführt werden. Die Pipeline gliedert sich in viele Jobs, welche durch ein Environment und Befehle, die dort auszuführen sind, charakterisiert sind. Das Environment kann dabei die Definition von Variablen, das Starten von Anwendungen oder das Vorbereiten von Programmausfürungen sein. Der Runner ist dabei nicht mehr als eine definierbare Shell auf der Befehle ausgeführt werden. Der Ausführungsort des Runners ist dabei unabhängig vom Gitlabserver, benötigt wird nur eine Internetverbindung und die Registrierung bei der entsprechenden Gitlab Instanz. Dadurch ist es möglich auf verschiedenen Server Runner zu registrieren und eine automatische Lastverteilung zu schaffen. Der Gitlabserver weiß durch die Registrierung des Runners, an welche Ausführungsumgebung er anfallende Jobs zuteilen kann und übermittelt sie per TCP.

Bei der Pipeline von CMSuite war es nötig Jobs mit und ohne Environment zu definieren da viele Jobs z.B. eine laufende Docker Umgebung benötigen. Um keinen unnötigen Overhead zu erzeugen mussten also auf einem Runner Pakete für das Environment installiert werden, auf dem anderen nur die Paktete für die Grundfunktionalität. Diese beiden Runner wurden durch die beiden Tags cmsuite_with-env und cmsuite_no-env unterschieden. Die Jobs werden durch diese Tags differenziert und nur einem Runner mit zugehörigem Tag zugeteilt.

Zur Ausführung der Runner werden verschiedene Umgebungen unterstützt: Shell, Virtualbox, SSH, Docker, Parallels(Gitlab-Dokumentation, 2018a). Shell ist dabei die rudimentärste Umgebung, die Befehle des Jobs werden einfach in der gleichen Shell ausgeführt in welcher der Runner läuft. Sicherheitstechnisch und aus Sicht des Ziels von unabhängigen Jobs ist dies auf keinen Fall zu empfehlen. Virtualbox bot sich für unsere Zwecke sehr gut an da es folgende Eigenschaften bot:

- Unabhängige Umgebungen
- Vorinstallation von Paketen auf Images
- Neue Umgebung bei jedem Job
- Skalierbarkeit

Der Nachteil von jeweils neuen Umgebungen ist, dass mit jedem Job eine neue virtuelle Maschine gestartet werden muss, dies bedeutet einen konstanten Geschwindigkeitsverlust bei der Ausführung der Pipeline. Die gewonnene Unabhängigkeit und Skalierbarkeit ist allerdings auf Kosten des Overheads hinnehmbar und durch eine erhöhte Zahl an parallelen Runnern sogar ausgleichbar.

2.2.2 Pipeline

Einige zum Verständnis dieser Arbeit wichtige Definitionen bietet Tabelle 2.2, da als Ausführungsumgebung Gitlab verwendet wird, sind die Definitionen an die offiziellen Beschreibungen angelehnt(Gitlab-Dokumentation, 2018b).

Da die Gliederung der Pipeline in voneinander abhängigen Stages erfolgt müssen verschiedene Stufen des Test-, Build- und Deploymentprozesses definiert werden. Die Stages sind so definiert, dass sie jeweils Vorraussetzung für die nächste Stufe sind. Sobald eine niedrigere Stufe als nicht bestanden gilt werden die darauf folgenden Stufen nicht mehr ausgeführt. Diese Stufen sollen grundlegende Tests von Modulen, Tests von Gesamtfunktionalitäten, das Erstellen der Images für jede Anwendung und die Auslieferung der fertigen Anwendung beinhalten. Daher wurden die Stages wie folgt definiert:

- Commit
- Acceptance

Befehl	Auf einer Kommandozeile ausführbare Funktion mit Eingabewerten und möglicher Ausgabe auf der Kommandozeile, er führt eventuell weitere Befehle aus. Als erfolgreich ausgeführt gilt er dann, wenn der Rückgabewert 0 ist.
Job	Menge von Befehlen, welche in definierter Reihenfolge in einer bestimmten Umgebung ablaufen. Die Umgebung kann durch die ausgeführten Befehle verändert werden. Damit beeinflussen sich Befehle innerhalb eines Jobs gegenseitung, es entsteht eine lineare Abhängigkeit vom ersten bis zum letzten Befehl. Der Job gilt genau dann als bestanden, wenn alle seine Befehle erfolgreich ausgeführt wurden. Eine Ausnahme bildet hier nur die Option allow-failure, ein so gekennzeichneter Job gilt auch bei fehlgeschlagenen Befehlen als bestanden.
Stage	Zusammenfassung von mehreren Jobs die einer logischen Einheit angehören. Die Jobs dürfen untereinander keine Abhängigkeiten aufweisen und werden in zufälliger Reihenfolge ausgeführt. Es ist möglich, dass mehrere Jobs einer Stage parallel laufen. Die Stage gilt als bestanden, wenn alle zugehörigen Jobs als bestanden gelten.
Pipeline	Eine Pipeline definiert eine Abfolge von Stages. Die Stages dürfen untereinander Abhängigkeiten aufweisen da diese linear abgearbeitet werden. Durch diese Abhängigkeit werden, wenn eine Stage abgebrochen wird, alle darauf folgenden Stages nicht ausgeführt. Die Pipeline gilt als bestanden, wenn alle Stages bestanden wurden.

Tabelle 2.2: Definitionen im Zusammenhang einer CI/CD Pipeline

- Build
- Deploy

Teil der Deploymentstage ist das Ausführen der fertigen Anwendung auf einem Kubernetescluster in einem abgeschlossenen Environment. Dieses Deployment dient in erster Linie als Kontrolle für den Entwickler, um die Auswirkungen von Codeänderungen im laufenden System zu analysieren. Der Deploymentbranch ist ein ausgezeichneter Branch im CMSuite Repository und beinhaltet nur fertige und getestete Anwendungen und liefert in der letzten Stage ein vorzeigbares Deployment welches so auch in Produktivumgebungen eingesetzt werden kann. Da der Entwickler nicht immer ein laufendes Deployment benötigt wird der letzte Schritt der Pipeline bei normalen Branches erst nach einem mauellen Trigger durch den Entwickler ausgeführt.

Die Deployments werden in zwei verschiedene Environments eingeordnet, Testing und Production. Für die Deployments eines Branches bietet Gitlab die Möglichkeit bei Problemen ein Rollback auszuführen, also eine frühere Version statt der aktuellen auszuführen. Besonders beim Production Branch kann diese Funktion bei unerwarteten Fehlern eine nützliche Option zur Herstellung eines stabilen, verwendbaren Zustands sein.

2.2.3 Caching

Viele der Jobs haben bestimmte Abhängigkeiten wie NPM und NodeJS. Weil die Jobs in einer virtuellen Maschine (VM) ausgeführt werden müssen bei jedem Durchlauf eines Jobs diese Pakete neu geladen werden. Da der nächste Job wieder aus einem Grundzustand startet werden die geladenen Paktete gelöscht und müssen erneut geladen werden.

Es ergibt also Sinn sich über ein job- und pipelineübergreifendes Caching Gedanken zu machen, die Lösung bietet die Gitlab Pipelineoption cache-folder, dort werden Ordner angegeben die Inhalte enthalten, welche gecached werden sollen. Dabei erstellt der Runner nach jedem Jobdurchlauf eine ZIP Datei in der die Inhalte der angegebenen Ordner komprimiert gespeichert werden. Beim nächsten Job wird die ZIP Datei wieder entpackt und benötigte Dateien sind nun schon verfügbar. Alle noch aktuellen Pakete müssen also nicht erneut geladen werden, dies bietet ähnlich einem Webcache einen erheblichen Geschwindigkeitsvorteil abhängig von Datenmenge und Downloadrate. Dies funktioniert nativ bei shell, ssh und anderen Runnern, bei Virtualbox allerdings nicht(Gitlab-Dokumentation, 2018c). Das liegt daran, dass nach jedem Job die erstellte VM wieder gelöscht und für neue Jobs eine Maschine aufbauend auf dem base-image erstellt wird, dadurch geht die ZIP Datei verloren welche die gecachten Ordner enthält.

Um dieses Problem zu umgehen wurde das Konzept der Shared-Folders benutzt.

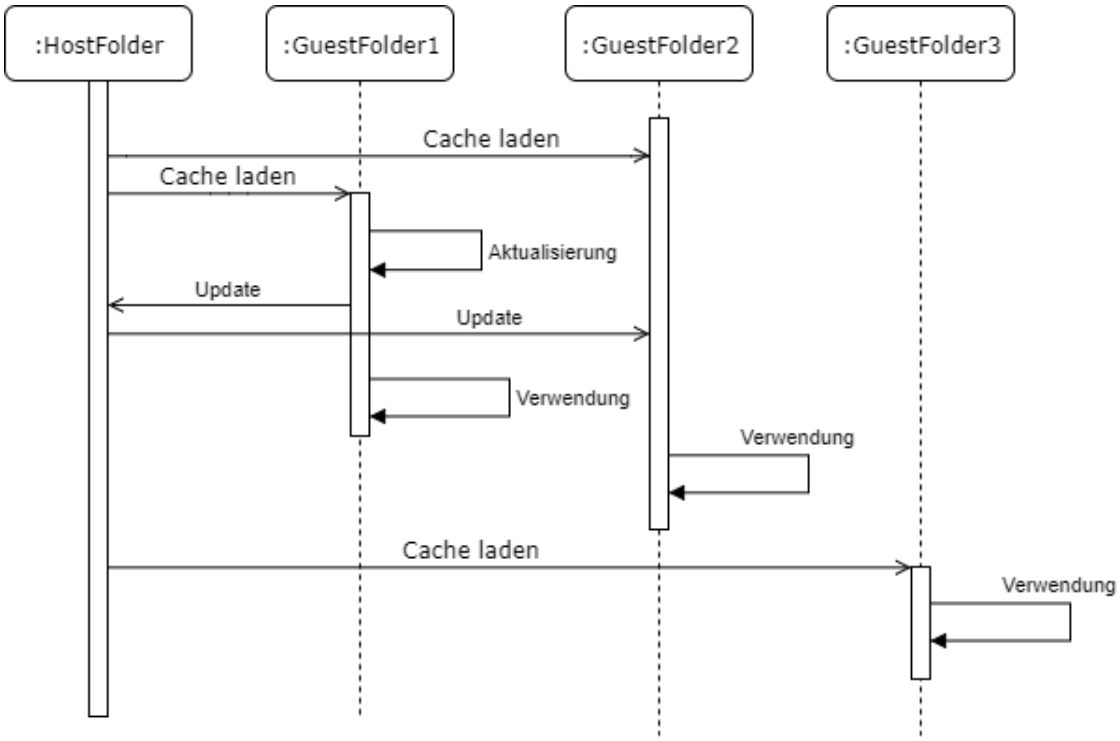

Abbildung 2.1: Zusammenspiel der SharedFolder

Shared-Folders sind ein Teil von GuestAdditions, einer Virtualbox Erweiterung zur Kommunikation zwischen Host- und Gastsystem. Shared-Folders integrieren einen Ordner des Hostsystems in das Gastsystem, hier die VM des Virtualbox runners. Dabei werden die Inhalte bidirektional zwischen den beiden Systemen übertragen.

Abbildung 2.1 zeigt das Zusammenspiel von Shared-Folders bei einem Host Ordner und mehreren Guest Ordnern. Dabei aktualisiert zunächst eine VM den Inhalt des Ordners, die aktualisierte Version wird zunächst beim Host und dann bei den anderen VMs aktualisiert. VM Nummer 3 startet danach erst und muss nichts aktualisieren da die bereits aktuellen Daten verwendet werden. Nach dem Herunterfahren der VMs bleiben die Daten nur auf dem Host bestehen.

Wenn man als Speicherort der ZIP Datei nun diesen SharedFolder angibt, bleibt dieser auch nach dem Löschen der alten VM auf dem Host bestehen und kann von einer neu gestarteten VM wieder genutzt werden. Da nur ein Ordner auf dem Hostsystem benutzt wird erfolgt eine Synchronisierung zwischen den VMs und es ist automatisch sichergestellt, dass immer die aktuellste Version der gecacheten Pakete vorliegt.

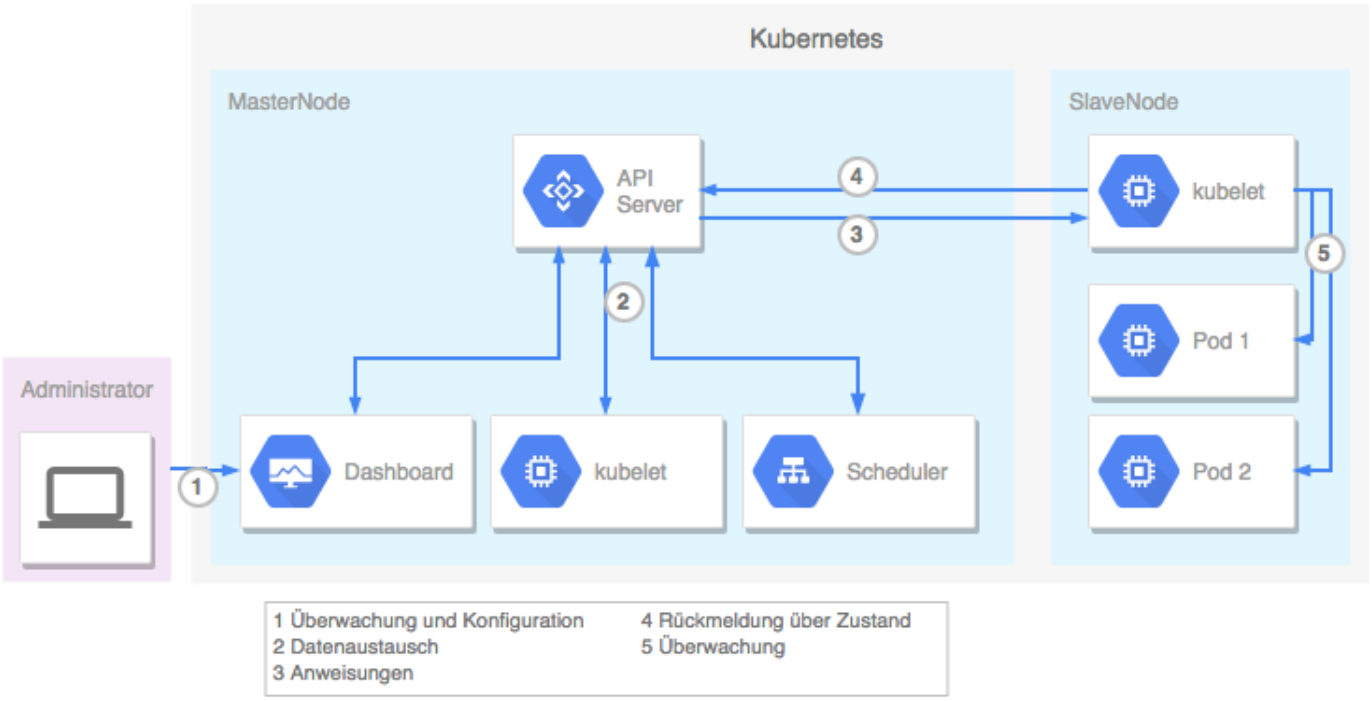

Abbildung 2.2: Kubernetes Architektur

2.3 Kubernetes Setup

Kubernetes ist ein OpenSource System zur Bereitstellung, Skalierung und Verwaltung von containerbasierten Anwendungen. Es wird seit 2014 von Google kontinuierlich weiterentwickelt und ist einer der Marktführer zur Orchestrierung von Containerapplikationen (Linthicum, 2015). Kubernetes setzt dabei auch eine verteilte Cluster Architektur, die von einem zentralen Master verwaltet wird. Dieser verwaltet ein bis beliebig viele Nodes auf denen die Container laufen. Kubernetes bietet für den Benutzer folgende Vorteile im Gegensatz zu rudimentären Container Ausführungsumgebungen:

- Übersichtliche Administration
- Lastverteilung
- Skalierbarkeit
- Integriertes Logging und Monitoring
- Ausfallsicherheit

Abbildung 2.2 zeigt eine Übersicht über die wichtigsten Bestandteile eines Kubernetesclusters und deren Interaktion.

Die oben genannten Features werden großteils durch den Master beeinflusst. Er

bietet Zugriff auf das Dashboard, konfiguriert den API Server und den Scheduler. Um erhöhte Ausfallsicherheit erreichen zu können ist es möglich mehrere Masternodes zu definieren die Aufgaben eines ausgefallenen Masters übernehmen können (Fricke, 2018).

Wichtiger Teil des Master ist der API-Server, dieser bietet eine Schnittstelle zwischen Nutzer, Masternode und den Slavenodes um Informationen per REST auszutauschen.

Der Scheduler überwacht den Auslastungsgrad und die Verfügbarkeit der Nodes mittels API und weist neuen Pods entsprechend Ressourcen zu. Meldet ein Node einen Fehler oder antwortet er nicht, weist der Scheduler einem anderen Node die dort laufenden Pods zu.

Auf jedem Kubernetes Node läuft eine kubelet Instanz. Kubelet liest primär Pod Spezifikationen vom API-Server und versucht diese der Spezifikation nach auszuführen und verfügbar zu halten. Weiterhin sorgt es für das Umsetzen von Anweisungen durch den Masternode und liefert diesem gleichzeitig die nötigen Informationen über den Zustand des Nodes.

2.3.1 Begriffsdefinitionen im Zusammengang mit Kubernetes

In Tabelle 2.3 werden einige Grundlegende Begriffe im Sprachgebrauch von Kubernetes Deployments definiert. Diese sind eine gekürzte und vereinfachte Fassung der offiziellen Dokumentation.

2.3.2 Architekturbeschreibung

Abbildung 2.3 zeigt exemplarisch das Zusammenspiel der wichtigsten Komponenten eines ausgeführten Deployments. Zu beachten ist, dass die nur zwei beispielhafte Deployments dargestellt werden in denen auch nur einige der tatsächlichen Container laufen.

Die als Containerimage gespeicherten Anwendungen von CMSuite werden auf Portus, einer Dockerregistry, bereitgestellt. Diese Images werden bei jedem neuen Deploy heruntergeladen um zu gewährleisten, dass die aktuellste Version deployed wird.

Um eine unabhängige Umgebung für jeden deployten Branch zu gewährleisten müssen alle Branches in unterschiedlichen Pods laufen. Pro Pod sollen alle Container des Branches ausgeführt werden um die interne Kommunikation zu gewährleisten. Bei einem Pod pro Container hätte man unnötigen Overhead durch die zusätzlichen Pods und müsste Services definieren welche die Kommunikation zwischen den Pods ermöglichen.

Pod	Die kleinste logische Einheit eines ausgeführten und verwalteten Containers ist dabei ein Pod. Dieser hat folgende Eigenschaften(Kubernetes-Dokumentation, 2018c): • Führt Imageinstanz aus • Kann mehrere Container enthalten • Hat Internes Netzwerk • Hat innerhalb des Clusters eine eindeutige Pod-IP • Besitzt einen shared context: namespace, cgroups, volume
Deployment	Ein Deployment ist die nächst größere Einheit und fasst beliebig viele Pods zu einer logischen Einheit zusammen. Das Deployment übernimmt die Verwaltung der Pods und startet oder beendet diese. Für ein Deployment kann zB die Replikenanzahl festgelegt werden. Über ein Deployment können auch alle zugehörigen Pods pausiert, gestartet oder gelöscht werden (Kubernetes-Dokumentation, 2018a).
Service	Kubernetes definiert verschiedene Typen von Services die innerhalb des Clusters die Kommunikation zwischen Pods übernehmen. Unter anderem kann eine Lastverteilung zwischen Pods, das Öffnen von Ports für den Nutzer oder eine Weiterleitung von Ports zu bestimmten Pods realisiert werden. Services werden im Rahmen dieser Bachelorarbeit nur zur Kommunikation von einem definierten Eingangspunkt zum Pod verwendet (Kubernetes-Dokumentation, 2018d).
Ingress	Ein Ingress verwaltet den externen Zugriff von Nutzern auf das Cluster. Es ist dabei möglich einen Datenfluss für HTTP und HTTPS zu einem bestimmten Service zu definieren. Die Auswahl der Services kann u.a. durch ein URL Routing erfolgen(Kubernetes-Dokumentation, 2018b).

Tabelle 2.3: Bezeichnungen von Kubernetes im Zusammenhang mit Deployments

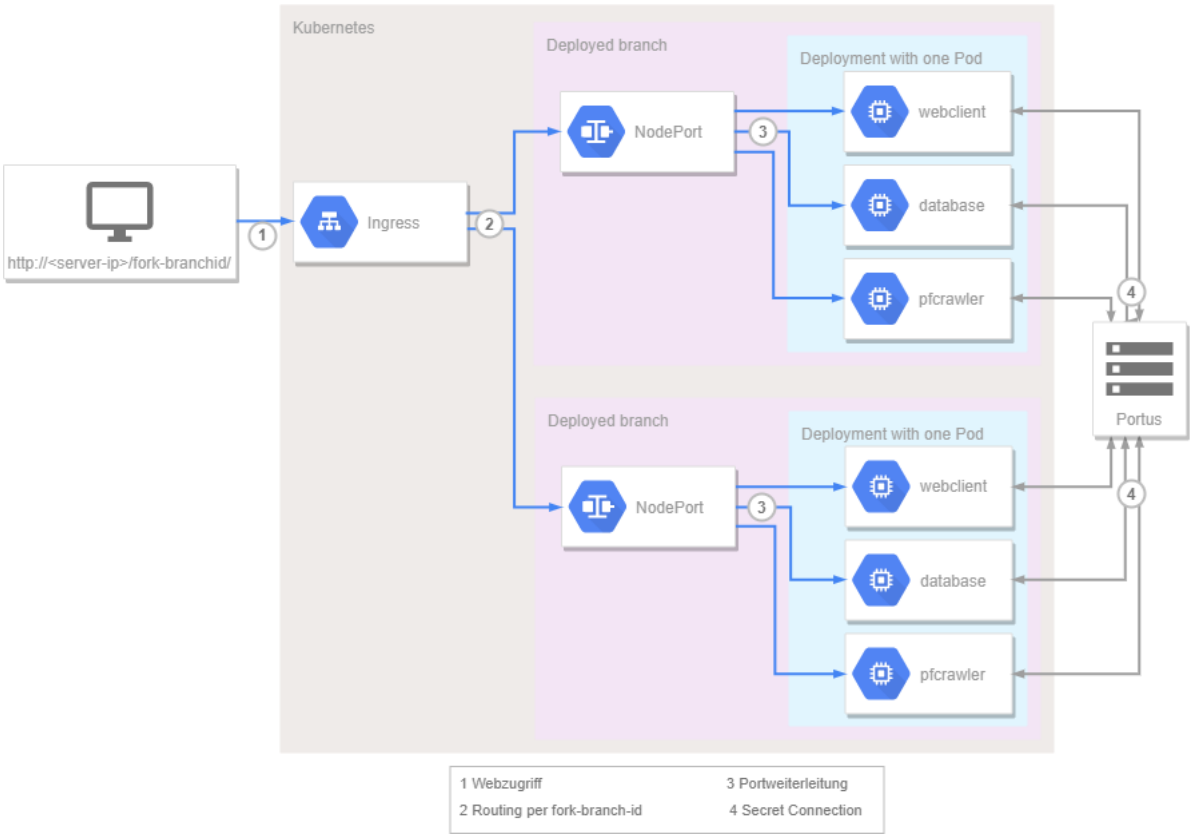

Abbildung 2.3: Deployment Architektur

Bei jedem Pod sind bestimmte für CMSuite benötigte Ports offen um einen Zugriff auf die verschiedenen Services zu ermöglichen. Um den Pod verwalten zu können ist dieser in ein Deployment eingebettet das durch ein eindeutiges Label ansprechbar ist. Das Deployment definiert auch die Schnittstelle zum Docker Repository wo sich die auszuführenden Images befinden. Um diese Schnittstelle, welche Username und Passwort enthält, nicht öffentlich sichtbar machen zu müssen wird hier ein Secret verwendet, dieses wird als Objekt bei Kubernetes definiert und beinhaltet verschlüsselt URL der Registry, Username und Passwort.

2.3.3 Minikube

Minikube ist eine Single-Node Kubernetesinstanz, die sich für Testzwecke und zum Entwickeln auf Notebooks sehr gut eignet. Dabei werden alle zentralen Komponenten eines echten Kubernetesclusters geboten. Wichtig für den Einsatz als Deploymentsystem sind hierbei:

- NodePorts
- ConfigMaps und Secrets
- Mehrere Pods
- Dashboard
- Ingress

Minikube läuft dabei auf einem oder mehreren Kernen in einer virtuellen Maschine, als Virtualisierungssoftware wird Virtualbox verwendet da dieses bereits Anwendung als Executor für Pipelinejobs dient. Dadurch ist es leichter die Übersicht über verschiedene virtuelle Maschinen zu behalten und durch eine zentrale Instanz zu verwalten. Minikube lässt sich mit unterschiedlich vielen Prozessoren und Arbeitsspeicher starten, dadurch ist es möglich es an verschiedene Lastsituationen anzupassen. Minikube lässt sich über die Befehle minikube start und minikube stop steuern, Kubernetesfunktionen werden wie bei einem echten Cluster über das Commandline Tool kubectl gesteuert. Es ermöglicht auch den Zugriff von Entwicklern auf das Kluster per Netzwerkverbindung, dies wird für die Fernkonfiguration von Testdeployments und einfache Administration benötigt.

Statt minikube hätte hier auch ein tatsächliches Kubernetes Cluster verwendet werden können, dies ist aber aufgrund der aktuellen Auslastung und hinsichtlich der zu beschaffenden Infrastruktur nicht sinnvoll. In einem späteren Stadium bei größeren Lasten kann es durchaus Sinn ergeben minikube durch ein Cluster zu ersetzen, alle Befehle für kubectl werden dann weiterhin kompatibel sein.

2.4 Continuous Deployment von CMSuite

2.4.1 Gründe für Continuous Deployment

Continuous Deployment bezeichnet das Zusammenspiel von Softwareentwicklung, Prozessen und Infrastruktur um in möglichst kurzer Zeit dem Endnutzer eine Software in Produktivumgebung vorstellen zu können. Im Teil der Softwareentwicklung wird dazu auf Continuous Integration, dem kontinuierlichen Zusammenfügen und Testen von Teilsystemen zum Gesamtsystem noch im Entwicklungsprozess, aufgebaut. Die Basis bildet also ein getestetes Softwareprodukt. Dieses wird durch automatisierte, am besten Toolgestützte Prozesse zur fertigen Software zusammengetragen und auf einer geeigneten Infrastruktur ausgeführt. Geeignete Infrastruktur bezeichnet hier eine dem Produktivsystem ähnliche Infrastruktur oder das echte Produktivsystem.

Nun stellt sich die Frage wann es nötig und sinnvoll ist ein Continuous Deployment aufzubauen, die Antwort liegt in der Methode der Softwareentwicklung. Open Source Projekte werden in den meisten Fällen nach den Prinzipien der agilen Softwareentwicklung realisiert. Dazu gehören das Entwickeln von Features in einem ein bis zweiwöchigen Zyklus, die Priosisierung von funktionierender Software über Dokumentation und das möglichst häufige Ausliefern von funktionsfähiger Software für Kunden und Projektbetreuer. Diese Prinzipien werden auch bei der Entwicklung von CMSuite angewendet.

Um funktionierende Software in einem kurzlebigen Turnus ausliefern zu können ist es nötig den Deploymentprozess zu automatisieren und für jedes Teammitglied ausführbar zu gestalten. Damit haben diese die Möglichkeit schon vor dem Release ihre Änderungen in der Deploymentumgebung zu testen und zu validieren. Dies kommt dem Prizip zugute, dass Fortschritt in der agilen Softwareentwicklung hauptsächlich durch den Funktionsgrad der laufenden Software bemessen wird. Die Continuous Deployment Pipeline soll dabei helfen immer zum frühest möglichen Zeitpunkt funktionierende Software liefern zu können. Dies kann entweder in Form eines Containerimages sein oder in Form der tatsächlich laufenden Anwendung auf einem Server. Abbildung 2.4 zeigt die Integration einer CI/CD Pipeline in die Abläufe der agilen Softwareentwicklung. Wichtig ist dabei, dass die Pipeline als Kontrollinstanz zum Testen und für Reviews dient. Damit bekommt der Entwickler direktes Feedback und kann dementsprechend seine Software verbessern.

2.4.2 Environmentkonzept

Die Deployments laufen auf Kubernetes alle unabhängig voneinander. Jedes komplette Deployment soll durch eine eigene mit Label versehene Deployment, Netzwerk und Servicekonfiguration definiert werden. Diese Label sollen ein gemein-

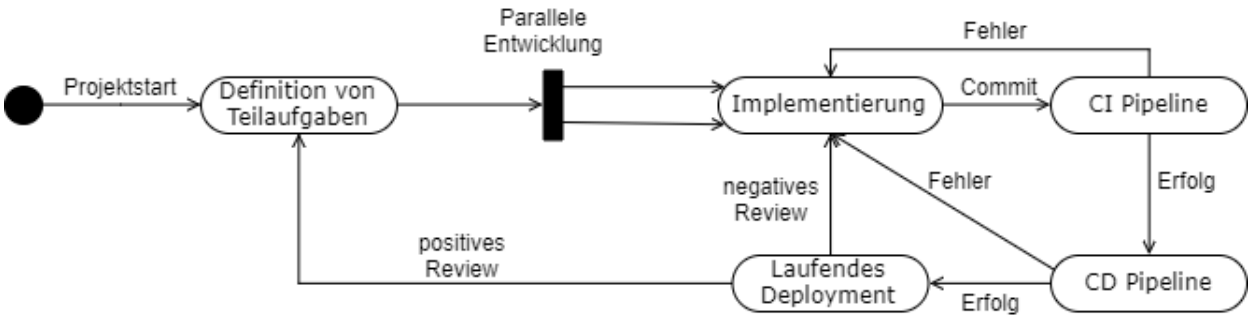

Abbildung 2.4: Integration einer CI/CD Pipeline in den agilen Entwicklungsprozess

sames Ansprechen der Komponenten und, wenn ein neueres verfügbar ist, das einfache Löschen von alten Deployments möglich machen. Die Deployments gliedern sich in Test und Production Environment.

Das Testenvironment wird immer deployed, wenn ein Entwickler eine laufende Instanz von CMSuite eines bestimmten Branches beurteilen will. Dieses läuft möglicherweise mit Fehlern, daher weren bei Fehlschlägen dem Entwickler Logs zur Verfügung gestellt die der Fehlersuche dienen. Das Deployment wird gelöscht sobald eine aktuellere Version des Branches deployed wird. Beim Fehlschlagen des Deployments hat der Entwickler die Möglichkeit auf den Stand eines früheren Deployments per Rollback zurückzukehren.

Das Productionenvironment beinhaltet das Deployment aus dem Productionbranch. Dieser ist ein Teil des CMSuite Repositorys und beinhaltet nur vollends entwickelte und lauffähige Software. Das Deployment dient als Demonstrationsobjekt und könnte gleichermaßen als Produktivsystem eingesetzt werden. Ein Rollback auf vorherige Deployments bei unerwarteten Problemen oder Kundenwünschen ist auch hier möglich.

2.4.3 Internes und externes Routingkonzept

Ziel der Netzwerkkonfiguration soll die korrekte Anzeige der CMSuite Anwendung für den Entwickler oder Kunden sein. Dafür müssen die Container untereinander Daten austauschen können, zB Daten aus der Datenbank lesen, und der Webclient muss dem Nutzer angezeigt werden. Um innerhalb eines Kubernetes Clusters Pods und Container zu verbinden sind zwei Komponenten nötig: das Öffnen eines Ports am Container und das Herstellen der Verbindung zwischen den Containern. Zum Öffnen des Ports innerhalb eines Pods wird bei der Podkonfiguration ein sog. ContainerPort angegeben, dieser ist dann innerhalb des Pods für andere Container nutzbar. Damit können im Cluster die verschiedenen Anwendung wie cmsuite-db, cmsuite-transformationmanger, cmsuite-webclient miteinander

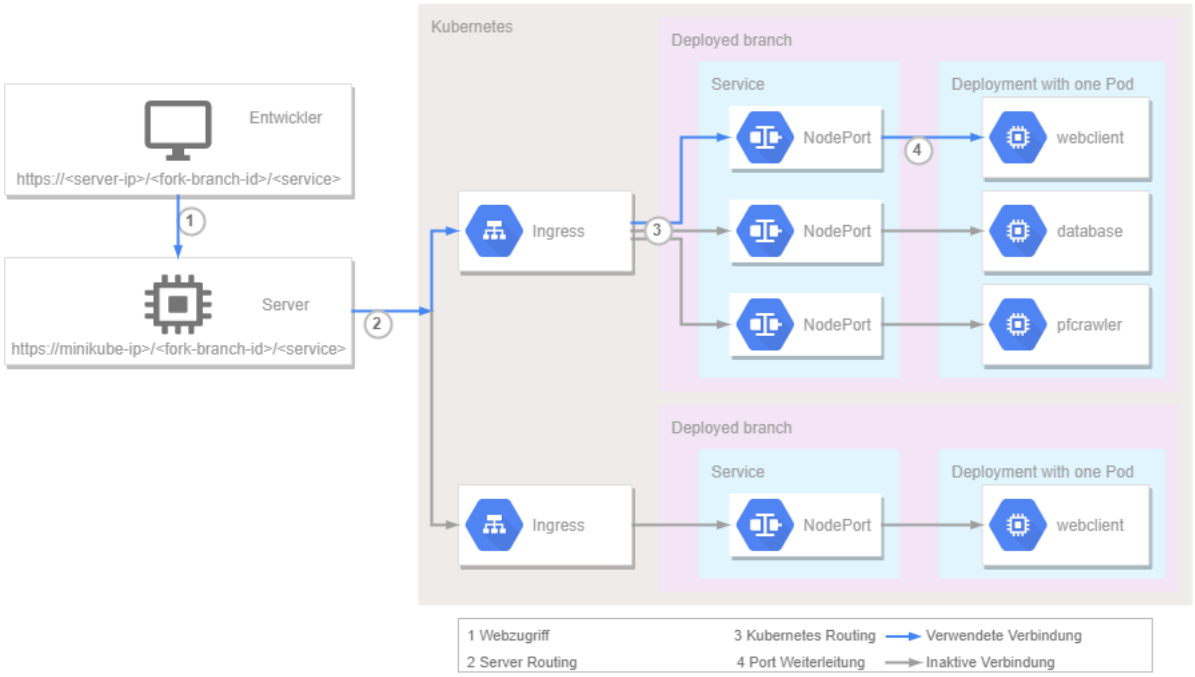

Abbildung 2.5: Netzwerkarchitektur von Server und Kubernetes

Kommunizieren.

Da der Nutzer von außerhalb des Clusters auf die Weboberfläche von CMSuite zugreifen will ist es nötig die entsprechenden Ports auch nach außen zu öffnen. Dies wird durch den NodePort Service umgesetzt. Der Service existiert für jeden Container in jedem Deployment ein mal und definiert eine Weiterleitung eines Ports auf einen per Label definierten Port eines Pods. Auch der Service wird per Label kenntlich gemacht.

Der Ingress verwendet dieses Label zur Weiterleitung von Anfragen auf diesen Service. Weiterhin ordnet er jedem Deployment eine URL nach folgendem Muster zu

https://⟨minikube-ip⟩/⟨fork-branch-id⟩/

Wird diese aufgerufen so leitet er die Anfrage auf den Service mit passendem Port weiter.

Damit der Nutzer nicht die nur intern verwendbare Minikube IP benutzen muss wird eine IPTables Regel definiert welche jeden Traffic von Port 443, dem von SSL verwendeten Port, vom Server auf Minikube weiterleitet. Dadurch ist die Verwendbarkeit des Deployments auch für Nutzer außerhalb des Clusters gewährleistet.

Es ist oft nötig bestimmte Komponenten des Clusters zu inspizieren, dabei ist nicht praktikabel sich immer durch eine SSH-Verbindung Zugang zum Cluster

zu verschaffen. Um diesem Umstand zu begegnen, bietet kubectl die Möglichkeit mehrere, auch externe, Cluster zu definieren und geeignet anzusprechen. Diese Verbindung wird durch den API-Server realisiert und benötigt dementsprechend die URL dessen sowie eine geeignete Verifikation des Nutzers. Beides wird in einer Konfigurationsdatei von kubectl festgelegt.

3 Implementierung

3.1 Continuous Integration von CMSuite

3.1.1 Gitlab Runner

Der Gitlab Runner ist eine Erweiterung zum Gitlab Server und Webinterface der Jobs aus der CI/CD Pipeline ausführt, die aktuellste Version kann als Binary[1] heruntergeladen und installiert werden. Zum Registrieren des Runners und Verbinden mit dem Zielrepository müssen einige Daten eingegeben werden:

- Coordinator URL
- CI Token
- Beschreibung
- Auszuführende Tags
- Executor Typ

Der Executor Typ bestimmt in welcher Umgebung die Befehle der Jobdefinition ausgeführt werden. Im Fall von CMSuite wird Virtualbox verwendet, hier muss der Name eines base-images für eine VM eingegeben werden. Diese virtuelle Maschine baut bei CMSuite auf einem Image namens cmsuite-runner-base-vm auf welches Java JDK 8, Maven, NodeJS und Docker installiert hat. Weiterhin sind die im Kapitel Caching beschriebenen Konzepte implementiert.

Um die Struktur möglichst klar zu halten, gibt es zwei Runner: einen bei dem das Image weitere Environmentpakete enthält und einen der nur die grundlegenden Abhängigkeiten installiert hat. Das Image ohne Environment ist dabei ein direktes Abbild des base-image. Das Environment-Image enthält zusätzlich noch docker-compose und kubectl. Kubectl ist zur Ausführung der Deploystage installiert und auf eine Verbindung zu Minikube konfiguriert. Um die beiden Runner

[1] https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

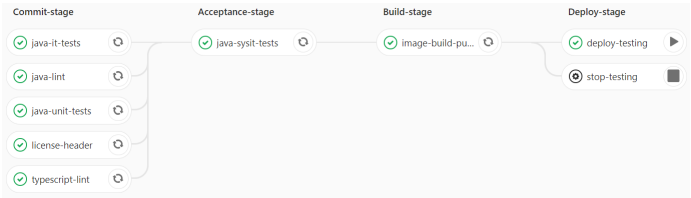

Abbildung 3.1: Exemplarische Gitlab CI/CD Pipeline

unterscheiden zu können werden die beiden Tags cmsuite_with-env und cmsuite_no-env verwendet. Die so getaggten Runner führen nur zum Tag passende Jobs aus.

Grundlage der beiden Images ist das Image ubuntu-18.04.1-live-server-amd64.iso[2], dieses wird noch bis 2023 unterstützt benötigt nur 512 MB RAM bzw 1,5 GB Festplattenspeicher. Die Images laufen nicht mit diesen Minimalvorraussetzungen, wichtig war nur, dass mehrere Runner gleichzeitig Jobs ausführen können.

Beim Ausführen des Jobs startet der Runner zunächst Virtualbox mit dem passenden Image und wartet, bis das System hochgefahren ist. Sobald dies der Fall ist, werden mögliche Anweisungen aus der before_script Sektion der Pipeline ausgeführt, dann die eigentlichen Befehle und abschließend after_script. Alle Ausgaben der Ausführung werden geloggt und können auf dem Gitlab Webinterface betrachtet werden. Sobald der Job durchgelaufen ist, wird die VM heruntergefahren und gelöscht, der nächste Job beginnt wieder mit dem unveränderten base-image.

3.1.2 Pipeline

Die 4 Stages der CI/CD Pipeline Commit, Acceptance, Build und Deploy bilden die logische Gruppierung der Jobs. Alle Jobs einer Stufe werden parallel ausgeführt und sind daher auch funktional unabhängig, der Grad der Parallelität wird nur durch die Anzahl der verfügbaren Runner beschränkt. Abbildung 3.1 zeigt exemplarisch den Aufbau einer Pipeline im Webinterface von Gitlab. Bei dem gezeigten Durchlauf waren alle Jobs erfolgreich.

Die ersten beiden Stufen wurden inhaltlich zu großen Teilen von der vorher bereits bestehenden Jenkinspipeline übernommen. Besonders sind hier nur die Möglichkeit der Sektion before_script ein Environment Setup zu definieren und bei after_script wiederum ein Enviromnet shutdown auszuführen. Dies wird u.a. durch Docker-Compose zum Testen verwendet.

Die Buildstage benutzt ebenso die von Docker-Compose bereitgestellte Umgebung, um die Anwendungen zu kompilieren und mit dem definierten Namen Poject_Path-BranchId zur Docker Registry hochzuladen.

[2]http://releases.ubuntu.com/18.04.1/ubuntu-18.04.1-desktop-amd64.iso

Die Deploymentstage besteht je nach Branch aus unterschiedlichen Jobs. Wird der Productionbranch verwendet so wird ein Job verwendet, welcher auch ein Deploy ins Production Environment ausführt. Bei allen anderen Branches führt diese Stage einen Job zum Deploy ins Testing Environment aus. Als zweiter Job existiert jeweils noch eine Funktion zum Stoppen des Deployments, wird ein Branch öfter deployed wird dieser Job vor dem nächsten Deploy automatisch ausgeführt.

Beide Jobs führen dazu das Script deploy.sh aus, dieses passt zunächst alle zur Konfiguration von Kubernetes nötigen Templatedateien mit dem Parameter fork-branch-id an und erstellt daraus anschließend ein Kubernetesdeployment. Die fork-branch-id ist dabei eine eindeutige Bezeichnung zusammengesetzt aus der Forkbezeichnung und den aktuellen Branchnamen. Die Deploymenterstellung wird ausgeführt durch den Befehl

```
kubectl create -f dateiname
```

kubectl ist dabei so konfiguriert, dass es auf das jeweils aktuelle Deploymentcluster zeigt.

An dieser Stelle könnten auch Runner definiert werden bei denen z.B. für Production Deployments kubectl auf ein anderes Cluster als Minikube verweist. Dies bietet die Möglichkeit für bestimmte Environments ein besonders ausfallsicheres oder schnell erreichbares Cluster zu verwenden.

Das Script overwatch.py wird anschließend dazu verwendet, um die fehlerfreie Erstellung des Deployments zu überprüfen. Auch hier wird wieder kubectl in Verbindung mit dem API-Server dazu verwendet um den Status des Deployments zu überprüfen. Das Script wartet dabei so lange, bis entweder das Deployent als Running angezeigt wird oder ein Fehler festgestellt wird. Sollte ein Fehler festgestellt werden so schreibt das Script dem Entwickler die Logs des Deployments und der einzelnen Container auf die Konsole. Diese Logs können durch den Entwickler im Webinterface betrachtet und ausgewertet werden.

Zeigt das Script deploy.sh oder overwatch.py einen Fehler an so wird der Deploymentjob und die Stage als nicht bestanden markiert und der Entwickler informiert.

3.1.3 Caching

Um ein gemeinsames Caching zwischen den Virtuellen Maschinen welche die Jobs ausführen zu ermöglichen kann wie im Kapitel 2.2.3 beschrieben nicht auf das übliche Caching von gitlab-runnern zurückgegriffen werden. Damit der Cache erhalten bleibt wird ein shared-folder zwischen Hostmaschine und Virtueller Maschine erstellt. Dafür wird die Erweiterung Guestadditions installiert, welches von Virtualbox entwickelt wird. Offizielle Quelle für Guestaddtions ist https://download.virtualbox.org/virtualbox/⟨version⟩/VBoxGuestAdditions_⟨version⟩.iso

mit mindestens Version 4.3. Die Installation kann entweder per .iso image oder durch die Paketverwaltung durchgeführt werden. Guestadditions übernimmt den Datenaustausch jeglicher Art zwischen Host und Virtueller Maschine, im Fall von CMSuite reicht ein gemeinsam benutzter Ordner.

Dafür muss zunächst ein Ordner an einem sinnvollen Ort auf dem Host erstellt werden, hier wurde z.B. /home/cmsuite-build/.gitlab-runner/cache-host gewählt. Dieser sollte durch die Option automount auf der virtuellen Maschine unter /media/sf_cache-host verfügbar sein. Da diese Funktion auf dem Produktivsystem allerdings nicht funktioniert hat, musste hier auf ein manuelles Einbinden zurückgegriffen werden.

```
mount -t cache-host /media/sf_cache-host
```

Als Mountpoint kann durchaus etwas anderes gewählt werden als /media/sf_cache-host, da dies allerdings der von automount gewählte Mountpoint ist wurde er aus Konsistenzgründen beibehalten.

Ein entsprechendes Script führt den mount Befehl aus. Durch einen Upstart-Service wird dieses Script mit jedem Reboot ausgeführt, sodass sichergestellt ist dass bei jedem neuen Job der Ordner verfügbar ist.

Innerhalb der config.toml des Gitlab Runners muss nun noch die Konfiguration für den cache-folder erstellt werden, dafür wird der Runner unter /home/cmsuite-build/.gitlab-runner/config.toml um die Angabe des Ordners mit den gecacheten Dateien erweitert

```
[runners.virtualbox]
    cache_dir = "/media/sf_cache-host/"
```

Aufgrund der Umstellung im CMSuite Projekt von manuell erstellten zu automatisiert erstellten Images konnten die Änderungen zur Aktivierung des Cachings nur in einer Testumgebung implementiert werden. Dort konnten alle beschriebenen Aktionen ausgeführt werden und führten zur erfolgreichen Verwendung eines Ordners auf dem Hostsystem als Speicherort für gecachte Dateien.

3.2 Minikube Setup

Da Minikube nur ein simuliertes Cluster ist, wird zum Ausführen von Minikube eine Virtualisierungsumgebung benötigt. Dafür wird Virtualbox verwendet, dieses ist im Packagerepository von Ubuntu vorhanden und kann per apt-get installiert werden. Minikube wird in den allermeisten Fällen nur lokal verwendet und konfiguriert, daher genügt es hier kubectl zu installieren. Soll auch ein externer Zugriff gewährt werden muss auch an diesem Ort kubectl installiert werden. Bei der Konfiguration einer externen kubectl Instanz kann auf die offizielle

Beschreibung von Kubernetes[3] zurückgegriffen werden.

Die Installation von Minikube folgt ebenfalls in weiten Teilen der offiziellen Beschreibung[4]. Um die korrekte Installation von Minikube zu überprüfen können die Befehle

```
minikube start
minikube status
```

ausgeführt werden, diese starten das Cluster und zeigen eine knappe Übersicht über den Status verschiedener Funktionen an. Als nächstes muss das zum Pullen der CMSuite Images benötigte Secret erstellt werden. Dies passiert durch den Befehl:

kubectl create secret docker-registry docker-registry-secret

–docker-server=https://mojo-docker.cs.fau.de

–docker-username=cmsuite-cd

–docker-password=[password]

–docker-email=cmsuite@group.riehle.org

Damit Minikube bei jedem Reboot läuft, wird das ubuntuinterne Upstart verwendet, wird als Hostsystem etwas anderes als Ubuntu verwendet muss die Startroutine entsprechend angepasst werden, meist wird hierfür ein Cronjob genügen der auf den Reboot wartet. Unter Ubuntu wird ein Service mit folgender Konfiguration erstellt:

```
[Unit]
Description=minikube start
Wants=network-online.target
After=network-online.target
[Service]
ExecStart=/home/cmsuite-build/minikube-start.sh
[Install]
WantedBy=multi-user.target
```

Dabei wartet der Service mit dem Start bis das Netzwerkinterface verfügbar und online ist damit eine Verbindung zum Docker repository hergestellt werden kann. ‘WantedBy=multi-user.target’ führt dazu, dass immer ein Start des Clusters erfolgt und nicht nur bei einem bestimmten Benutzer. Die Datei /home/cmsuite-build/minikube-start.sh ist das eigentlich auszuführende Programm und hat folgenden Inhalt:

[3] https://kubernetes.io/docs/tasks/tools/install-kubectl/

[4] https://github.com/kubernetes/minikube/releases

#!/bin/sh

runuser -l cmsuite-build -c "minikube start"

Würde einfach nur 'minikube start' ausgeführt werden dann würde Minikube mit root Rechten starten da während des reboots alle Prozesse per Root ausgeführt werden. Um nun Minikube nicht unnötig viele Rechte erteilen zu müssen wird runuser -l cmsuite-build verwendet, dadurch wird der Befehl in einer Shellumgebung von cmsuite-build ausgeführt.

Um den Service dem System bekannt zu machen und zu aktivieren, müssen die folgenden Befehle ausgeführt werden:

```
systemctl daemon-reload
systemctl enable minikube-start.service
```

Nun ist sichergestellt dass nach jedem Neustart des Servers auch Minikube neu gestartet wird und für den Nutzer verfügbar ist.

3.3 Continous Deployment von CMSuite

3.3.1 Umsetzung der Deployments

Konfigurationsdateien für alle Kubernetesobjecte können entweder als .yaml oder als .json erstellt werden. In der Arbeit wird .yaml verwendet da dies etwas übersichtlicher ist, sematisch sind die beiden Dateiformate jedoch äquivalent.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: cmsuite-{{FORK_BRANCH_ID}}-deployment
  labels:
    app: cmsuite-{{FORK_BRANCH_ID}}
spec:
    metadata:
      labels:
        app: cmsuite-{{FORK_BRANCH_ID}}
    spec:
      imagePullSecrets:
      - name: docker-registry-secret
      containers:
      - name: cmsuite-db
        image: mojo-docker.cs.fau.de/cmsuite/demo-db:latest
        imagePullPolicy: Always
        ports:
          - containerPort: 54321
```

```
- name: cmsuite-httpd-webclient
  image: mojo-docker.cs.fau.de/cmsuite/httpd-webclient:
                    {{FORK_BRANCH_ID}}
  imagePullPolicy: Always
  ports:
  - containerPort: 80
```

Listing 3.1: Deployment Definition

Um das Deployment mit einem Pod und allen Containern von CMSuite zu erstellen kann man entweder ein Deployment und einen Pod definieren und bereitstellen oder man verwendet eine implizite Definition, bei der man nur das Deployment mit Containern definiert und Kubernetes hierfür automatisch einen Pod zuordnet. Hier wurde die zweite Notation verwendet um die Dateien möglichst kompakt und damit verständlich zu halten.

Ein exemplaritsch Auszug aus der deployment.yaml Datei findet sich unter 3.3.1 . Hier wird zunächst die APIVersion und Art der Anwendung als Deployment festgelegt. Die Metadaten wie ‘name‘ dienen dem Nutzer um sein Deployment eindeutig finden zu können, ‘Label‘ wird von Kubernetes benötigt, über diese eindeutige Kennung findet später die Zuordnung von den Services zum Deployment und Ingress statt.

Die vorhin beschriebene implizite Notation findet sich in der nochmaligen Definition des Labels für den erstellten Pod, da auch dieser identifiziert werden muss. Nun wird noch das Secret bekannt gemacht, welches benutzt werden soll um die Images zu pullen. Danach werden für die Container Namen vergeben und ein Image Pfad angegeben, dieser verweist auf die Dockerregistry von CMSuite und wird per Secret aufgelöst. Die ‘imagePullPolicy: Always‘ bestimmt, dass keine lokalen Images z.B. von älteren Deployments verwendet werden sollen, sondern bei jedem Aufruf neue Images geholt werden sollen. Zum Schluss wird noch der ‘ContainerPort‘ angegeben, dieser ist später für die Verbindung zwischen den Containern untereinander und dem Nutzer wichtig.

Die Containerdefinition wiederholt sich gleichbleibend für die alle Container. Damit die Container, die auf der Datenbank arbeiten, wissen mit welchen Zugangsdaten die Datenbank erreichbar ist wird ein Environment durch eine Configmap definiert, diese ist für alle Container gleich und wird unter cmsuite-[fork-branch-id]-configmap 3.3.1 erstellt.

```
kind: ConfigMap
metadata:
  name: cmsuite-{{FORK_BRANCH_ID}}-configmap
  namespace: default
  labels:
    app: cmsuite-{{FORK_BRANCH_ID}}
```

```
data:
  CMSUITE_COMMONS_PERSISTENCE_DBCOORDINATEMANAGER_HOST: localhost
  CMSUITE_COMMONS_PERSISTENCE_DBCOORDINATEMANAGER_PORT: "5432"
  CMSUITE_COMMONS_PERSISTENCE_DBCOORDINATEMANAGER_DBNAME: postgres
  CMSUITE_COMMONS_PERSISTENCE_DBCOORDINATEMANAGER_USERNAME: postgres
  CMSUITE_COMMONS_PERSISTENCE_DBCOORDINATEMANAGER_PASSWORD: postgres
```

Listing 3.2: Deployment Definition

Die Configmap beinhaltet den einen Deploymentspzifischen Namen, das übergreifende Label für den Branch und die Informationen für die Datenbankverbindung. Die Variablennamen ergeben sich aus dem Packagepfad der zu überschreibenden Standartwerte von CMSuite.

3.3.2 Umsetzung von Netzwerks und Routing

Die Netzwerkkonfiguration wurde in zwei Teilen implementiert, als interne Konfiguration in service.yaml und als externe Kofiguration in ingress.yaml. Des Weiteren gibt es eine Netzwerkweiterleitung vom Server zum Minikube Cluster.

Service

Zur Identifizierung wird dem Service eines Deployments zunächst ein Name und ein App-Label mit der fork-branch-id zugewiesen. Als Type wird ein NodePort definiert, dieser entspricht der Weiterleitung eines Ports an einen Pod. Die Ports umfassen hierbei die Ports 80, 5432, 8080, 8081, 8082, 8083, 8084 welche von den Containern benötigt werden. Pro Definition einer Weiterleitung wird ein Eingangsport, im Dokument nur als port bezeichnet, und ein Ausgangsport, als targetPort bezeichnet, benötigt. Um dem Cluster bekannt zu machen, für welchen Pod diese Weiterleitungen gelten sollen gibt es den ‘selector’, dieser bezieht sich auf das ‘label: app’ des Pods. Einen gekürzten Ausschnitt dieser Definition bietet Codeabschnitt 3.3.2. Das dort generisch eingefügte FORK-BRANCH-ID wird bei jedem Deployment durch den entsprechenden Wert ersetzt.

```
kind: Service
metadata:
  name: cmsuite-{{FORK_BRANCH_ID}}-service
  labels:
    app: cmsuite-{{FORK_BRANCH_ID}}
spec:
  type: NodePort
  ports:
    - name: {{FORK_BRANCH_ID}}-httpd-webclient
      port: 80
      targetPort: 80
```

```
  selector:
    app: cmsuite-{{FORK_BRANCH_ID}}
```

Listing 3.3: Service Definition

Ingress

Der Ingress ist dafür zuständig eine vom Nutzer aufgerufene URL auf einen Port des oben definierten Service zu verweisen. Die URL ist dabei wie folgt aufgebaut:

http://⟨hostip⟩/⟨FORK-BRANCH-ID⟩/⟨servicename⟩

Diese URL wird als Grundlage der Weiterleitungsregel als ´path´ definiert. Jede URL, die auf diesen Pfad zutrifft wird dann auf eine Backend weitergeleitet. Das Backend ist im Fall dieser Arbeit einfach eine Spezifikation des gerade eben definierten Service. Dieser wird über den vergebenen Namen identifiziert. Pro Servicename gibt es eine Regel, die diesen Aufruf auf den Service mit dem entsprechenden Serviceport verweist. Eine beispielhafte verkürzte Implementierung findet sich unter Codeabschnitt 3.3.2.

```
kind: Ingress
metadata:
  name: cmsuite-{{FORK_BRANCH_ID}}-ingress
  labels:
    app: cmsuite-{{FORK_BRANCH_ID}}
spec:
  rules:
  - http:
      paths:
      - path: /{{FORK_BRANCH_ID}}/webclient/
        backend:
          serviceName: cmsuite-{{FORK_BRANCH_ID}}-service
          servicePort: 80
```

Listing 3.4: Ingress Definition

Sollte keine der angegebenen Regeln zutreffend sein, so wird auf Port 80 des default-http-backend weitergeleitet, dieses ist für die WebAPI und weitere Funktionen von Kubernetes zuständig. Wird auch hier keine valide URL identifiziert, wird dem Nutzer eine Fehlermeldung angezeigt.

Sollte zusätzlich zu dem einfachen Anzeigen des Deployments auch noch eine Verbindung zur Kontrolle, Überwachung und Steuerung des Klusters eingeführt werden, so muss eine Verbindung per kubectl hergestellt werden. Kubectl verwendet zur Konfiguration eines Clusters einen sogenannten Context, dieser beschreibt, welches Cluster im Moment angesprochen wird, unter welcher API es verfügbar ist und durch welche Daten eine Verbindung hergestellt werden kann. In Fall

dieser Arbeit muss ein externes Cluster konfiguriert werden, welches über die IP des Servers auf dem Minikube läuft angesprochen wird. Um eine verschlüsselte Verbindung herzustellen und den Endnutzer zu Authentifizieren wird ein key mit Zertifikat benötigt. Beide werden beim Start von Minikube für die lokale kubectl Version automatisch angelegt und können auch von den Remotenutzern verwendet werden. Eine Definition des Kontexts mit Cluster sieht wie folgt aus:

```
clusters:
  - cluster:
    insecure-skip-tls-verify: true
    server: https://10.131.64.198:8443
    name: minikube-cluster
contexts:
  - context:
    cluster: minikube-cluster
    user: minikube
    name: minikube-remote
current-context: minikube-remote
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /path/to/client.crt
    client-key: /path/to/client.key
```

Listing 3.5: kubectl Konfiguration

Da im Fall von Minikube die Zertifikate selbst signiert sind, kann keine tatsächliche Verifikation gegen ein root-Zertifikat durchgeführt werden, daher der Zusatz ‘insecure-skip-tls-verify:true’ welcher die Verifizierung des Zertifikats verhindert.

Nach dem Wechsel zu diesem Context hat kubectl Zugriff auf alle Komponenten des Cluster, insbesondere Deployments, Pods und Services. Dies kann helfen lokal das Cluster zu inspizieren und mögliche Probleme ohne eine SSH-Verbindung auf den Server festzustellen.

Wird dazu noch der Befehl:

kubectl proxy http://localhost

ausgeführt wird der lokale Traffic auf das Cluster weitergeleitet, es ist also einfach möglich auf Features wie das Dashboard unter http://localhost:8001/api/v1/services/kubernetes-dashboard:80/ zuzugreifen oder andere Funktionen der gesamten API zu benutzen.

Der Entwickler besitzt damit auch von einem anderen Rechner aus die volle Kontrolle über das Cluster und kann Tests und Änderungen vornehmen.

4 Evaluation

Die folgende Evaluation prüft die Erfüllung der Anforderungen von 1.2 in Bezug auf die unter 1.3 aufgestellten Metriken. Jede Anforderung wird dabei in die Kategorien erfüllt, teilweise erfüllt und nicht erfüllt eingeordnet. Zur Kategorie teilweise erfüllt wird, sofern möglich, ein Grad der Fertigstellung angegeben.

- 1. Funktionalität von Jenkins auf Gitlab übertragen teilweise erfüllt

  Es gibt unter Jenkins insgesamt 6 Jobs die in den Stufen Commit und Acceptance gegliedert sind welche zu übertragen waren. Der Job ´Wait for Environment´, welcher bei Jenkins noch vor der Commit Stage ausgeführt wurde, wurde bei Gitlab in jeden einzelnen Job integriert. Die restlichen Jobs wurden auf Gitlab übertragen und dort entsprechend neu implementiert. Das Feature incremental-build welches zur Beschleunigung von aufeinanderfolgenden Builds dient konnte aus Zeitgründen nicht mehr implementiert werden. Der Funktionalität ist dies nicht abträglich, es erhöht nur den Zeitaufwand der Stages. Abbildung 4.1 zeigt die Ursprünglichen Pipelinestufen, Abbildung 4.2 die neu implementierte Variante auf Gitlab.

- 2. Tests automatisiert ausfüren erfüllt

  Als Pipelinetrigger wurden alles definiert was das Repository ändert, also merge, commit, push. Diese decken die geforderten Ausführungskriterien ab, die Trigger gelten für das CMSuite Repository sowie für alle Forks.

- 3. Auslieferunge auf Kubernetes anstoßen erfüllt

  Die Auslieferung ist Teil der Pipeline mit dem Stufen Build und Deploy, dabei werden zunächst Containerimages von CMSuite erstellt und anschließend verschiedene Konfigurationsdateien für Kubernetes erstellt. Dazu zählen Deployment, Service und Ingresskonfigurationen welche dann das gesamte Deployment bilden. Durch die Strategie PullAlways ist wie gefordert sichergestellt dass nur die neuesten Images ausgeführt werden. Das Löschen von alten Deployments wird immer am Anfang der Deploystage ausgeführt.

  Die ursprüngliche Metrik sah vor bei jedem Branch aller Forks ein laufendes

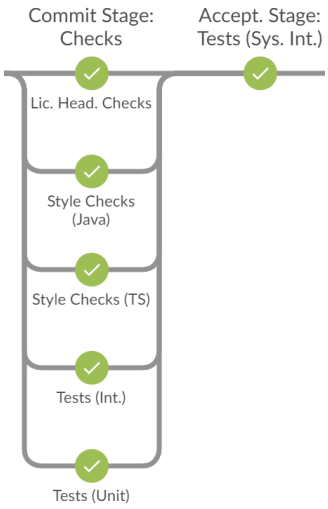

Abbildung 4.1: Ursprüngliche Stufen auf Jenkins

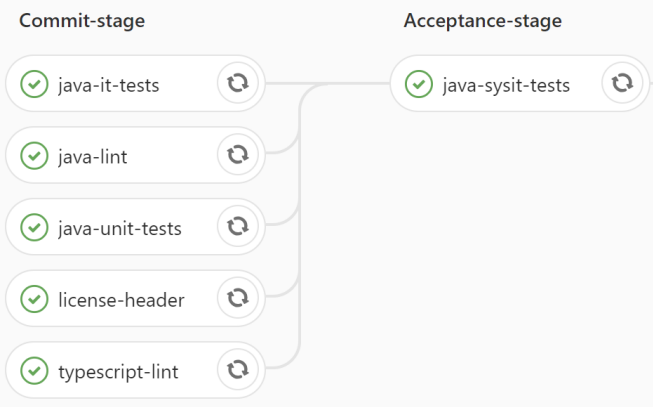

Abbildung 4.2: Übertragene Stages auf Gitlab

Deployment als Ziel zu haben, dies war aufgrund der zu geringen Serverleistung nicht möglich. Bei mehr als 4 parallelen Deployments konnte festgestellt werden dass Minikube nicht mehr reagiert, daher wurde entschieden die Deploy Stage nur nach einem Klick des Entwicklers auszurühren um zu hohe Lasten zu vermeiden. Entsprechend wird nun nicht mehr zugrunde gelegt dass alle Deployments laufen müssen, sondern nur die tatsächlich getriggerten. Bei diesen wird die Auslieferung auf Kubernetes immer angestoßen.

- Ausführen von CMSuite auf Kubernetes erfüllt

  Die Ausführung von CMSuite wird sichergestellt durch das script overwatch.py welches auf den Status Running der Container wartet und bei Problemen den Entwickler informiert. Das zugrundeliegende Script wird bei jedem Trigger der Deploy ausgeführt.

- Geschwindigkeit genauso gut wie Jenkins erfüllt

  Die Geschwindigkeit hängt von 3 Faktoren ab: Dem Server, dem Parallelisierungsgrad und der Struktur der Jobs. Jenkins führt startet vor der gesamten Pipeline ein Environment welches für den Durchlauf aller Stages bestehen bleibt und am Ende wieder heruntergefahren wird. Da Gitlab die Jobs in unabhängigen Environments und damit auch unabhängig voneinander ausführt muss am Anfang von Jedem Job das Environment gestartet und am Ende des Jobs heruntergefahren werden. Dies bedeutet einen erheblichen Overhead, die folgende Tabelle bietet eine Übersicht über einige Zeitmessungen in Minuten:

Der Parallelisierungsgrad ist bei Jenkins immer so hoch wie es parallele Jobs gibt, bei Gitlab nur die Anzahl der Runner. Bei Gitlab werden im Moment nur 3 Runner verwendet, deren Anzahl ist durch die Hardware des Servers beschränkt, konzeptionell könnte ein beliebiger Parallelisierungsgrad erreicht werden. Daher wird für die Messung angenommen dass nur der längste Job pro Stage die Gesamtzeit der Stage definiert. Um diesen Faktor bereinigt ergibt sich die letzte Spalte der Tabelle. Hier sieht man dass eine etwas bessere Zeit erreicht wird als unter Jenkins. Da es aufgrund einer internen Umstellung nicht möglich war das Caching im Produktivsystem einzusetzen, könnte hierdurch ein weiterer Geschwindigkeitsvorteil errungen werden.

Aufgrund der Tatsache dass die bereinigten Werte unter Gitlab eine bessere Performance zeigen als unter Jenkins wird diese Anforderung als erfüllt gewertet.

- 6. Im Fehlerfall Logs ausgeben erfüllt

  Beim Ausführen der Pipeline können in den ersten zwei Stufen Commit und Acceptance Tests fehlschlagen. Falls dies passiert wird die Pipeline ebenso als fehlgeschlagen markiert und die Logs dazu ausgegeben. Wenn

Ausführungsort	**Zeit gesamt**	**Zeit ohne Environment**	**Zeit bei max. Paralellität**
Gitlab #5042	36,50	23,06	14,15
Gitlab #5047	44,10	25,55	15,15
Gitlab #5079	38,57	22,31	14,59
Gitlab #5081	39,42	24,48	15,39
Gitlab Durchschnitt	39.54	24,05	15,02
Jenkins #751	19,25	17,59	17,59
Jenkins #757	22,30	21,38	21,38
Jenkins #769	23,03	21,56	21,56
Jenkins #770	19,34	18,20	18,20
Jenkins Durchschnitt	21,08	19,58	19,58

Tabelle 4.1: Geschwindigkeitsvergleich zwischen Gitlab und Jenkins

in der Buildstage oder der Deploystage während der Skriptausführung ein Problem auftritt können die Logs auch im Job betrachtet werden. Da dies keine Logs umfassen würde welche während des Startens des Deployments ausgegeben werden existiert das script overwatch.pyB. Dieses wartet auf die erfolgreiche Ausführung des Deployments und gibt bei Problemen dem Entwickler Logs zum gesamten Deployment sowie die Logs der einzelnen Container aus. Damit werden an jeder möglicherweise fehlerhaften Stelle dem Entwickler Logs zur Verfügung gestellt.

- Tests und Deployment wiederholbar erfüllt

  Gitlab bietet die Möglichkeit jeden Job bei dem die Abhängigkeiten aus den vorherigen Stufen erfüllt sind beliebig oft zu wiederholen. Da jeder einzelne Job wiederholbar ist ist dies auch mit der Gesamtheit der Pipeline der Fall. Beim der Deploystage ist zu beachten dass dem Deployjob vorangestellt zunächst ein möglicherweise altes Deployment gelöscht wird. Dies ist wichtig um bei einer Wiederholung immer nur ein laufendes Deployment mit gleichem Label vorzufinden und so eine korrekte Zuordnung sicher zu stellen. Um eine Wiederholung mit den gleichen Images sicherzustellen und möglichen Mehraufwand zu vermeiden wurde die Stage Build und Deploy auf zwei separate Stages aufgeteilt, dies stellt sicher dass auch bei einer Wiederholung des Deploys nur ein mal die Images erstellt werden müssen.

- 8. Einfache Benutzbarkeit für Entwickler

  Die Benutzbarkeit wurde als Zahl der Klicks von bestimmten Seiten zu

einer anderen Übersicht definiert. Die folgende Aufstellung zeigt die dazu definierten Aktionen mit der jeweils benötigten Klickanzahl:

- Von Commit zur Pipeline 2
- Von Commit zur Demoinstanz 2
- Von Merge Request zur Demoinstanz 2
- Von Commit zum Log eines Jobs 2
- Von Commit zum Log des Deployments 2
- Von Productionbranch zum Production Deployment 2
- Vom Projekt zum Production Deployment 2
- Von Branch zu aktuellem Pipelinestatu 1

Es ist deutlich zu erkennen dass alle Aktionen die in der Evaluationsmetrik festgelegten maximalen 2 Klicks erfüllen oder unterschreiten, damit ist eine einfache Benutzbarkeit für den Entwickler gegeben.

Insgesamt kann festgestellt werden dass die Aufgabe eine Deploymentpipeline zu implementieren gelungen ist. Der Entwickler bekommt eine laufende Demoinstanz mit der er arbeiten kann, der Productionbranch wird als Produktivsystem ausgeliefert und eine Verbindung zu den jeweiligen Deployments ist über einen einfachen Netzzugang gewährleistet. Als grundsätzliches Problem ist hierbei noch die Stabilität von Minikube zu nennen, diese ist teilweise unter hohen Lasten nicht mehr erreichbar und kann damit die vorgegebenen Aufgaben nicht erfüllen. Ein entsprechend leistungsstarker Server wäre die Grundlage für eine dauerhaft stabile und ausgeweitete Nutzung der Deployments. Die Implementierungsarbeit hierfür ist schon geleistet und muss nur noch am passenden System umgesetzt werden, damit ist jetzt der gesamte Integration und Deploymentprozess von CMSuite automatisiert und kann von Entwicklern wie auch Kunden verwendet werden.

Anhang A deploy.sh

```
# Make sure script fails if one command fails
set -e

# Make sure the script runs in the right folder
working_dir=$(pwd -P)
parent_path=$( cd "$(dirname "${BASH_SOURCE[0]}")" ; pwd -P )
cd "$parent_path"

# Save branchvariable to temporary variables with yaml-
    definitions
forkbranchid=$1
ingress=`cat "ingress_template.yaml" | sed "s/{{FORK_BRANCH_ID
    }}/$forkbranchid/g"`
service=`cat "service_template.yaml" | sed "s/{{FORK_BRANCH_ID
    }}/$forkbranchid/g"`
deployment=`cat "deployment_template.yaml" | sed "s/{{
    FORK_BRANCH_ID}}/$forkbranchid/g"`
configmap=`cat "configmap.yaml" | sed "s/{{FORK_BRANCH_ID}}/
    $forkbranchid/g"`

# Create new files for kubernetes config
echo "$ingress" > temp-kubernetes-files/ingress.yaml
echo "$service" > temp-kubernetes-files/service.yaml
echo "$deployment" > temp-kubernetes-files/deployment.yaml
echo "$configmap" > temp-kubernetes-files/configmap.yaml

# Remove running deployments of same branch
./clean.sh $forkbranchid

# Create internal deployment
kubectl create -f temp-kubernetes-files/ingress.yaml
kubectl create -f temp-kubernetes-files/service.yaml
kubectl create -f temp-kubernetes-files/deployment.yaml
kubectl create -f temp-kubernetes-files/configmap.yaml

# Remove previously created files
rm temp-kubernetes-files/ingress.yaml
rm temp-kubernetes-files/service.yaml
rm temp-kubernetes-files/deployment.yaml
rm temp-kubernetes-files/configmap.yaml

# Restore working directory
```

```
pwd=$working_dir
```

Anhang B overwatch.py

```
import subprocess
import argparse
import time
import sys

MAX_ATTEMPTS = 10
WAIT_AFTER_ATTEMPT_IN_SECOND = 15

def main():
    parser = argparse.ArgumentParser(description='Check if
        deployment is running')
    parser.add_argument('fork_branch_id', help='fork_branch_id
        from gitlab-ci pipeline')

    fork_branch_id = parser.parse_args().fork_branch_id



    # try MAX_ATTEMPTS times to get status Running, afterwords
        print log and return a failure
    for i in range(MAX_ATTEMPTS):
        pod = get_pod_to_id(fork_branch_id)

        # read the pods information and store them per line
        podinfo_lines = read_pod(pod).split("\n")

        statusPos = position_of_STATUS(podinfo_lines[0])
        status = get_status_on_position(statusPos,
            podinfo_lines[1])
        if status == "Running":
            print("Pod is running correctly")
            sys.exit(0)
        # wait 15 seconds, maby its not up yet
        time.sleep(WAIT_AFTER_ATTEMPT_IN_SECOND)

    printLogs(pod)
    sys.exit(1)

def read_pod(pod):
    # call "kubectl get pod <podid>" in subprocess
```

```
    podinfo = subprocess.run(['kubectl', 'get', 'pod', pod],
        stdout=subprocess.PIPE).stdout.decode('utf-8')
    return podinfo

def position_of_STATUS(podinfo):
    # position of the first occurence of STATUS
    return podinfo.find("STATUS")

def get_status_on_position(position, line):
    # return first String on the requested position
    return line[position:].split("␣")[0]

def get_pod_to_id(fork_branch_id):
    # get all pods with matching label from kubectl, command
        runs in subprocess
    allPods = subprocess.run(['kubectl', 'get', '--no-headers=
        true', 'pods', '-l', 'app=cmsuite-'+fork_branch_id, '-o
        ', 'custom-columns=:metadata.name', '--sort-by=.
        metadata.creationTimestamp'],stdout=subprocess.PIPE).
        stdout.decode('utf-8')
    # decode the output of the subprocess and get its last
        line member, this is the newly deployed
    # sometimes there might be the one more terminating pod
        which should be ignored
    allPodsOrdered = allPods.split('\n')[::-1]
    # only the newest created pod is relevant
    pod = allPodsOrdered[1]
    return pod

def printLogs(pod):
    containers = ['cmsuite-db',
                  'cmsuite-httpd-webclient',
                  'cmsuite-service-datamanager',
                  'cmsuite-service-pfcrawler',
                  'cmsuite-service-accountancy',
                  'cmsuite-service-transformationmanager',
                  'cmsuite-service-analysisresultprovider']
    for container in containers:
        print("==========␣Logs␣for:␣"+container+"==========")
        log = subprocess.run(['kubectl', 'logs', pod, '-c',
            container], stdout=subprocess.PIPE)
        print(log.stdout.decode('utf-8'))

if __name__ == '__main__':
    main()
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Fricke, D. T. (2018). Kubernetes Architektur. https://www.informatik-aktuell.de/entwicklung/methoden/kubernetes-architektur-und-einsatz-einfuehrung-mit-beispielen.html. [Online; accessed 15-December-2018].

Gitlab-Dokumentation. (2018a). Gitlab Executors. https://docs.gitlab.com/runner/executors/README.html. [Online; accessed 08-November-2018].

Gitlab-Dokumentation. (2018b). Gitlab Pipeline Definitionen. https://docs.gitlab.com/ee/ci/pipelines.html. [Online; accessed 26-December-2018].

Gitlab-Dokumentation. (2018c). Virtualbox Executor. https://docs.gitlab.com/runner/executors/virtualbox.html. [Online; accessed 26-December-2018].

Kubernetes-Dokumentation. (2018a). Deployment Definition. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; accessed 17-November-2018].

Kubernetes-Dokumentation. (2018b). Ingress Definition. https://kubernetes.io/docs/concepts/services-networking/ingress/. [Online; accessed 17-November-2018].

Kubernetes-Dokumentation. (2018c). Pod Definition. https://kubernetes.io/docs/concepts/workloads/pods/pod/. [Online; accessed 17-November-2018].

Kubernetes-Dokumentation. (2018d). Service Definition. https://kubernetes.io/docs/concepts/services-networking/service/. [Online; accessed 17-November-2018].

Linthicum, D. (2015). Search Data Center. https://www.searchdatacenter.de/tipp/Swarm-und-Kubernetes-Zwei-Tools-zur-Orchestrierung-von-Docker-Containern. [Online; accessed 07-December-2018].