Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Phil Adam

Die Bedeutung von Optimierungswerkzeugen in der Softwareentwicklung mit Kubernetes und deren Auswirkung auf den Entwicklungsprozess

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Phil Adam

Die Bedeutung von Optimierungswerkzeugen in der Softwareentwicklung mit Kubernetes und deren Auswirkung auf den Entwicklungsprozess

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Olaf Zukunft

Abgegeben am 16.10.2020

Phil Adam

Thema der Arbeit

Die Bedeutung von Optimierungswerkzeugen in der Softwareentwicklung mit Kubernetes und deren Auswirkung auf den Entwicklungsprozess

Stichworte

Skaffold, Tilt, Garden, Kubernetes, Docker, Cluster, Google Cloud, Microservices, Continuous Integration, Continuous Deployment, CI/CD-Pipeline

Kurzzusammenfassung

Ziel dieser Arbeit ist es einen Überblick über die Werkzeuge Tilt, Skaffold und Garden zu ermöglichen und ihre Bedeutung für die Optimierung des Softwareentwicklungsprozesses mit Kubernetes herauszuarbeiten. Hierfür werden die Werkzeuge an einem Fallbeispiel verprobt und anschließend anhand definierter Kriterien miteinander verglichen. Diese Arbeit soll zukünftig als Guide für die Integration der Werkzeuge für neue Projekt genutzt werden können.

Phil Adam

Title of the paper

The importance of optimisation tools in software development with Kubernetes and their impact on the development process.

Keywords

Skaffold, Tilt, Garden, Kubernetes, Docker, Cluster, Google Cloud, Microservices, Continuous Integration, Continuous Deployment, CI/CD-Pipeline

Abstract

The goal of this thesis is to give an overview of the tools Tilt, Skaffold and Garden and to work out their importance for the optimization of the software development process with Kubernetes. For this purpose the tools are tested on a case study and compared afterwards on the basis of defined criteria. In the future, this work should be used as a guide for the integration of the tools for new projects.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Motivation 1

1.2 Zielsetzung 2

1.3 Aufbau der Arbeit 2

2 Grundlagen 3

2.1 Softwarearchitekturen 3

2.1.1 Monolith 3

2.1.2 Microservices 3

2.2 Docker 5

2.2.1 Docker Container 5

2.2.2 Docker Image und Dockerfile 6

2.2.3 Docker Compose 6

2.2.4 Register 7

2.3 Kubernetes 7

2.3.1 Cluster 8

2.3.2 Kubernetes-Deployment 10

2.3.3 Kubernetes-Service 11

2.3.4 Minikube 11

2.4 Agile Softwareentwicklung 12

2.4.1 Continuous Integration 12

2.4.2 Continuous Deployment 13

2.4.3 GitLab 14

3 Analyse 16

3.1 Softwareentwicklung im Cluster 16

3.1.1 Cluster in der Entwicklung 16

3.1.2 Der Bereitstellungsprozess und seine Hindernisse 17

3.1.3 CI/CD Pipeline in der Entwicklung 19

3.2 Werkzeuge zur Entlastung der Entwickler 20

3.2.1 Bewertungskriterien der Optimierungswerkzeuge 22

3.2.2 Kriterien 22

3.3 Fallbeispiel 23

3.3.1 Architektur 23

3.3.2 Technologien 25

4 Optimierungswerkzeuge 26

4.1 Tilt 26

4.1.1 Bereitstellungsprozess 27

4.1.2 CLI 28

4.1.3 Graphische Benutzerschnittstelle 28

4.1.4 Snapshots und Tilt-Cloud 29

4.1.5 Konfiguration 30

4.1.6 Lizenz 34

4.2 Skaffold 34

4.2.1 Bereitstellungsprozess 35

4.2.2 CLI 38

4.2.3 Profile 39

4.2.4 Konfiguration 40

4.2.5 Integration CI/CD Pipeline 42

4.2.6 Lizenz ... 43

4.3 Garden ... 43

4.3.1 Bereitstellungsprozess ... 44

4.3.2 Stack Graph ... 46

4.3.3 CLI ... 49

4.3.4 Konfiguration ... 50

4.3.5 Integration CI/CD-Pipeline ... 52

4.3.6 Lizenz ... 53

5 Durchführung Fallbeispiel ... 54

5.1 Rahmenbedingungen und Implementierung ... 54

5.1.1 Google Cloud ... 54

5.1.2 Implementierung ... 55

5.2 Optimierungswerkzeuge ... 56

5.2.1 Tilt ... 56

5.2.2 Skaffold ... 59

5.2.3 Garden ... 63

6 Evaluation ... 68

6.1 Konfiguration ... 68

6.2 Performanz ... 70

6.3 Transparenz ... 70

6.4 Lizenz und Kosten ... 71

6.5 Dokumentation ... 71

6.6 Langlebigkeit ... 72

6.7 Auswertung ... 72

7 Fazit und Ausblick 75

7.1 Fazit 75

7.2 Ausblick 76

Abbildungsverzeichnis

Abbildung 1: Vergleich Microservices und Monolith (Malav, 2017).................................... 4
Abbildung 2: Schematischer Aufbau Kubernetes Cluster - In Anlehnung an (Kubernetes, 2019) und Wikipedia.. 8
Abbildung 3: CI/CD Pipeline (GitLab, [o. Jahr]).. 13
Abbildung 4 Entwicklungszyklus im Container nativen Umfeld (Eigene Darstellung) 18
Abbildung 5: Fallbeispiel -Architektur 24
Abbildung 6: Tilt - Bereitstellungsprozess 27
Abbildung 7: Tilt - Graphische Benutzerschnittstelle (Browser)... 29
Abbildung 8: Skaffold - Bereitstellungsprozess.. 36
Abbildung 9: Garden - Bereitstellungsprozess 45
Abbildung 10: Garden - Stack Graph (Garden, 2020h) 47
Abbildung 11: Tilt - Graphische Benutzerschnittstelle (Terminal)....................................... 58
Abbildung 12: Skaffold - Benutzerschnittstelle (Terminal).. 61
Abbildung 13: Garden - Fallbeispiel: Stack Graph.. 66

Quellcodeverzeichnis

Quellcode 1: Dockerfile 6
Quellcode 2: docker-compose 7
Quellcode 3: Kubernetes Deployment 11
Quellcode 4: Beispiel - GitLab-CI 14
Quellcode 5: Tilt - Basis-Tiltfile 31
Quellcode 6: Tilt - Überarbeitetes Dockerfile (Tilt, 2020) 32
Quellcode 7: Tilt - Best Practise Tiltfile (Tilt, 2020) 33
Quellcode 8: Skaffold - Container Structure Test 37
Quellcode 9: Skaffold - Skaffold.yaml mit jib 41
Quellcode 10: Garden – project.garden.yaml (Garden, 2020i) 50
Quellcode 11: Garden – garden.yaml für ein Modul (Garden, 2020i) 52
Quellcode 12: Tilt - Fallbeispiel: Ausschnitt - Tiltfile 57
Quellcode 13: Skaffold - Ausschnitt: skaffold.yaml Fallbeispiel 60
Quellcode 14: Skaffold - Integration in .gitlab-ci.yaml 63
Quellcode 15: Garden - Fallbeispiel: Ausschnitt project.garden.yaml 64
Quellcode 16: Garden - Fallbeispiel: Integrationstest 64
Quellcode 17: Garden - Fallbeispiel: .gitlab-ci.yaml 67

Glossar

API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create – Read – Update – Delete
GA	GA (General Availability) steht für eine allgemeine Verfügbarkeit. Der Begriff verdeutlicht, dass die Version für den Praxiseinsatz freigegeben ist.
IDE	Integrated Development Environment
Kernel	Der Kernel eines Betriebssystems, der für die grundlegenden Funktionen wie Speicher-, Prozess-, Task- und Diskmanagement zuständig ist und ständig im Hauptspeicher verbleibt.
REST	Representational-State-Transfer
SaaS	Software as a Service (SaaS) bezeichnet ein Distributionsmodell für Anwendungen über den Webbrowser. SaaS wird als Teilbereich des Cloud Computings verstanden, da angeforderte Applikationen nie direkt auf dem Gerät des Nutzers vorhanden sind (Floyd, et al., 2017).
VM	Virtual Machine

1 Einleitung

1.1 Motivation

Cloud Programmierung hat nun schon seit einigen Jahren einen hohen Stellenwert in der Softwareentwicklung. Vor allem die Container-Images und das flexible Orchestrieren bringen viele Vorteile gegenüber dem ursprünglichen VM-Image. Im agilen Umfeld mit Microservices profitiert man besonders von dem einfachen und effizienten Erstellen der Container-Images. Das Arbeiten mit Technologien, wie Kubernetes und Docker, ist schon lange nicht mehr nur Aufgabe von Cloud-Architekten und Administratoren, sondern findet zunehmend Anklang in der Softwareentwicklung. Dabei müssen sich Entwickler allerdings gleich mehreren Herausforderungen stellen. Zum einen wird der Technologie-Stack um Werkzeuge, wie Docker und Kubernetes, erweitert und zum anderen führt das kontinuierliche Bereitstellen der Anwendung während der Entwicklung zu einem Mehraufwand. Um eine Änderung an einer Anwendung im Cluster vorzunehmen, muss das Image jedes Mal neu gebaut und anschließend im Cluster bereitgestellt werden. Arbeitet man mit einem remote Cluster, wie bspw. der Google Kubernetes Engine, muss das Image zusätzlich noch in ein Register geladen werden. Diese Schritte müssen von den Entwicklern täglich mehrmals durchgeführt werden. Durch die hohe Anzahl an Wiederholungen am Tag, ist dies nicht nur sehr fehleranfällig, sondern auch ein großer Zeitaufwand und vor allem eine Unterbrechung des Entwicklungsprozesses. Seit einigen Monaten haben sich neue Werkzeuge etabliert, die genau hier Abhilfe schaffen sollen. Werkzeuge wie Skaffold, Tilt und Garden automatisieren das Build-, Deploy- und Push-Verfahren bei einer Quellcode-Änderung und versprechen ein sekundenschnelles Aktualisieren der Kubernetes Pods. Diese Werkzeuge lassen sich zum Teil auch in eine CI/CD Pipeline integrieren, welche mittels Caching optimiert werden kann.

1.2 Zielsetzung

Das Ziel dieser Bachelorarbeit ist es einen Überblick über die Werkzeuge Tilt, Skaffold und Garden zu ermöglichen und ihre Bedeutung für die Optimierung des Softwareentwicklungsprozesses mit Kubernetes herauszuarbeiten. Darüber hinaus sollen die Werkzeuge anhand eines Fallbeispiels verprobt und miteinander verglichen werden, sodass zum einen eine valide Beurteilung erfolgt und zum andern die optimalen Use-Cases der aufgezeigt werden. Anhand eines Fallbeispiels werden die *best-practise* Konfigurationen vorgestellt. Diese Arbeit soll zukünftig als Guide für die Integration der Werkzeuge für neue Projekt genutzt werden können.

1.3 Aufbau der Arbeit

Im Kapitel 2 wird auf die technischen Grundlagen eingegangen. Neben Microservices gehören im Rahmen dieser Bachelorarbeit Technologien wie Docker, Kubernetes und CI/CD Pipelines dazu.

Kapitel 3 beinhaltet die Analyse. Hier werden verschiedene Arten von Clustern mit ihren Eigenschaften diskutiert und die Hürden von Kubernetes in der Softwareentwicklung aufgezeigt. Außerdem wird das Fallbeispiel vorgestellt und die Bewertungskriterien für die Evaluation festgelegt.

In Kapitel 4 werden die drei Werkzeuge Tilt, Skaffold und Garden mit ihren spezifischen Eigenschaften, Funktionen und Konfigurationsmöglichkeiten vorgestellt.

Kapitel 5 beschreibt die Durchführung des Fallbeispiels unter Verwendung des jeweiligen Werkzeuges. Außerdem werden die Werkzeuge, soweit möglich, in eine CI/CD-Pipeline integriert.

In Kapitel 6 findet die Evaluation zu dem Fallbeispiel statt. Hier werden die Werkzeuge miteinander verglichen und ihr jeweils optimaler Use-Case vorgestellt.

Im letzten Kapitel 7 werden die Ergebnisse der Arbeit abschließend nochmal zusammengefasst. Des Weiteren wird ein Ausblick gegeben, welche Erweiterungen oder Änderungen in zukünftigen Projekten umgesetzt werden könnten.

2 Grundlagen

2.1 Softwarearchitekturen

2.1.1 Monolith

When all functionality in a system had to be deployed together, we consider it a monolith. (Newman, 2019 S. 12)

Die monolithische Struktur ist einer der bekanntesten und ältesten Softwarearchitekturen. Ein Monolith bildet eine untrennbare homogene Einheit, welche keine explizite Gliederung in Teilsysteme vorsieht. Dies hat zur Folge, dass Monolithen nur als vollständiges geschlossenes System bereitgestellt werden können (vgl. Fink, 2012). Dies wiederum führt vor allem im Cloud Umfeld zu Nachteilen. So ist eine Skalierung des Systems nur als Ganzes möglich und bei Änderungen muss das gesamte System neu gebaut und anschließend bereitgestellt werden.

2.1.2 Microservices

Wie in Abb. 1 dargestellt, besteht eine Microservice-Architektur im Vergleich zu einer monolithischen Architektur aus mehreren modularen Systemen (Microservices). Jeder Microservice übernimmt dabei eine wohldefinierte Aufgabe und arbeitet autark. Durch die klare Teilung der Zuständigkeiten entsteht eine lose Kopplung, was es ermöglicht, dass Microservices einzeln bereitgestellt werden können. Somit kann ein einzelner Service unabhängig von den anderen Services bereitgestellt oder ausgetauscht werden. Außerdem lässt sich das System bei Last besser skalieren. So lassen sich besonders beanspruchte Microservices durch Replikationen entlasten. Untereinander kommunizieren die Services über klar definierte Schnittstellen, welche als Vertrag zwischen den Microservices gesehen werden können. Meist findet die Kommunikation über REST oder Message Queues statt (vgl.

Newman, 2019 S. 1-6). Im Vergleich zu einem Monolithen entsteht bei einer Microservices-Architektur aber auch ein deutlich höherer Verwaltungsaufwand und eine höhere Komplexität.

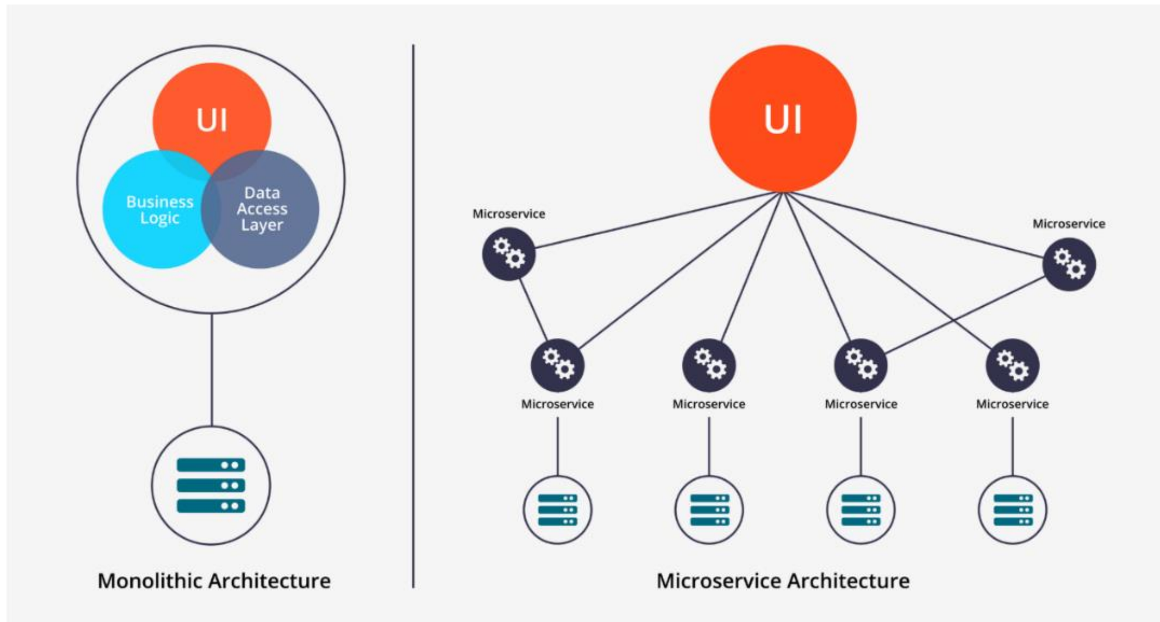

Abbildung 1: Vergleich Microservices und Monolith (Malav, 2017)

Microservices und der Cloud-native-Ansatz

Durch die lose Kopplung der Microservice-Architektur lässt sich jeder Microservice auf unterschiedlichen Servern und Standorten individuell bereitstellen. Die somit entstehende Flexibilität und Agilität der Software bietet vor allem im Bereich des Cloud Computing große Vorteile. Deshalb sind Microservices Teil des Cloud-nativen-Ansatzes. Ziel dieses Ansatzes ist es, die Besonderheiten der Cloud -Computing-Architektur zum Vorteil der Anwendung zu nutzen. Microservices im Zusammenspiel mit der Container-Technologie sind somit nicht mehr aus dem Cloud Computing wegzudenken (vgl. Luber, 2017).

2.2 Docker

Build, Ship and Run, Any App, Anywhere - Docker Mantra (Mouat, 2016 S. vii)

Docker ist eine 2013 veröffentlichte Open Source Virtualisierungssoftware, die Anwendungen in Containern isoliert und die sich im Laufe der Jahre zu der führenden Container-Software entwickelt hat (vgl. Mouat, 2016 S. 9 f.).

2.2.1 Docker Container

Wie auch virtuelle Maschinen können Docker-Container-Anwendungen isoliert vom Host-Betriebssystem gestartet werden. Ein Container ist eine Verkapselung einer Anwendung inklusive ihrer Konfiguration und Abhängigkeiten. Mit Docker-Containern können Anwendungen und ihre Umgebungen betriebsbereit und portabel zur Verfügung gestellt werden. Sie unterscheiden sich somit von dem herkömmlichen Ansatz einer virtuellen Maschine und bieten einige Vorteile ihr gegenüber. Jede virtuelle Maschine nutzt ein eigenes Gastbetriebssystem, welches über einen Hypervisor mit dem darunterliegenden Host-Betriebssystem kommuniziert. Container benötigen hingegen kein eigenes Betriebssystem. Alle Container auf einem Host teilen sich denselben Kernel des Hosts und dessen Container-Engine. Die Container-Engine ist für das Starten und Stoppen von Containern verantwortlich, vergleichbar wie der Hypervisor bei einer virtuellen Maschine. Bibliotheken können unter den Containern geteilt werden, was im Gegensatz dazu bei in sich geschlossenen virtuellen Maschinen nicht möglich ist. Container sind dadurch deutlich leichtgewichtiger als virtuelle Maschinen und lassen sich auch deutlich besser skalieren. Außerdem haben die Container den großen Vorteil der Portabilität. Da jeder Container die gesamte Laufzeitumgebung beinhaltet, verhält sich die Anwendung auf jedem Host unabhängig vom Host-Betriebssystem gleich (Mouat, 2016 S. 3-6).

2.2.2 Docker Image und Dockerfile

Docker Images werden zur Instanziierung von Containern benötigt. Anhand eines Images können beliebig viele Container instanziiert werden. Images bestehen aus stapelbasierten Dateisystemen. Dabei wird ausgehend von einem Basis-Image das Image schichtweise aufgebaut. Jede Schicht stellt dabei ein mögliches eigenes Image dar. Ein Image lässt sich anhand eines Dockerfiles definieren, auch *Manifest* genannt. Das Dockerfile ist eine Textdatei im YAML Format, in dem Anweisungen zur Instanziierung eines Containers stehen. Der *Docker Deamon*, welcher für die Ausführung von Anweisungen zuständig ist, verarbeitet diese und erstellt anschließend einen Container (vgl. Drilling, 2018).

```
FROM openjdk:11
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

Quellcode 1: Dockerfile

Der Quellcode 1 zeigt ein Beispiel für ein Dockerfile, welches ein Image für eine Java Anwendung definiert. Jedes valide Dockerfile beginnt mit dem Keyword *FROM*, welches das Base Image beschreibt. Anschließend wird in diesem Beispiel die bereits kompilierte Jar aus dem *target* Ordern in das Image kopiert. Da es sich um eine Web-Server handelt, wird der Port 8080 durch die *EXPOSE* Anweisung freigegeben. Die letzte Zeile *ENTRYPOINT* dient zur Ausführung der Jar Datei (vgl. Docker, [o. Jahr]a).

2.2.3 Docker Compose

Häufig besteht ein System nicht nur aus einer Anwendung. Gerade in Hinsicht auf Microservices werden mehrere einzelne Services für das Gesamtsystem benötigt. Das Starten jedes Service per Hand ist aufwendig und fehleranfällig. Docker bietet für diese Szenarien das Werkzeug *docker-compose* an, welches sich ebenfalls in einer YAML-Datei definieren lässt. Es bietet die Möglichkeit alle nötigen Services gemeinsam zu konfigurieren und anschließend zu starten. Anschließend wird mittels *docker-compose up* der Service gestartet (vgl. Docker,

[o. Jahr]b). Der Quellcode 2 zeigt ein exemplarisches Beispiel für die Verwendung von docker-compose. Nach der Angabe der Version können unter *services* beliebig viele Anwendungen angegeben werden. Diese können mit Images, Ports und ggf. weiteren Angaben konfiguriert werden. Über die Angabe von *depends_on* können Abhängigkeiten definiert werden. In diesem Beispiel führt dies dazu, dass die Datenbank Redis vor der Web-Anwendung gestartet wird.

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - redis
  redis:
    image: redis
```

Quellcode 2: docker-compose

2.2.4 Register

Zugang zu den Images erhält man über ein sogenanntes Register. Das offizielle Docker Register *Docker-Hub* bietet einige geprüfte Images an. Immer wenn ein referenziertes Image nicht lokal vorliegt, bezieht der Docker Deamon das Image aus dem Online Register (vgl. Drilling, 2018). Über den CLI-Befehl *docker login* lässt sich Docker mit einen belieben Register konfigurieren.

2.3 Kubernetes

Kubernetes ist ein Open Source System für die Container-Orchestrierung. Unter der Container-Orchestrierung versteht man das automatisierte Bereitstellen, Skalieren und Verwalten von containerisierten Anwendungen. Die Container-Einheiten werden dabei zu Clustern zusammengefasst und verwaltet. Dies ist vor allem für Microservice Anwendungen von großem Vorteil, da diese sich meist über mehrere Container erstrecken. Kubernetes bietet

eine einfache Administration und Überwachung der Zustände von Cluster an. Über das CLI von Kubernetes "*kubectl*" können Entwickler und Administratoren mit Kubernetes interagieren.

2.3.1 Cluster

Ein Cluster ist der Zusammenschluss mehrere Server, welche im Kubernetes Zusammenhang auch Nodes genannt werden. Ein Node kann sich dabei auf einer physischen oder virtuellen Maschine befinden. Die Cluster basieren auf dem Master-Slave Prinzip. Jedes Cluster besitzt dabei mindestens einen Master-Node, der für Skalierungs-, Replikations- und Verwaltungsaufgaben zuständig ist. Die anderen Nodes werden Worker-Nodes genannt und sind für die Verwaltung der Container Zuständig (vgl. Kubernetes, 2019). In Abb. 2 wird der schematische Aufbau eines Clusters illustriert.

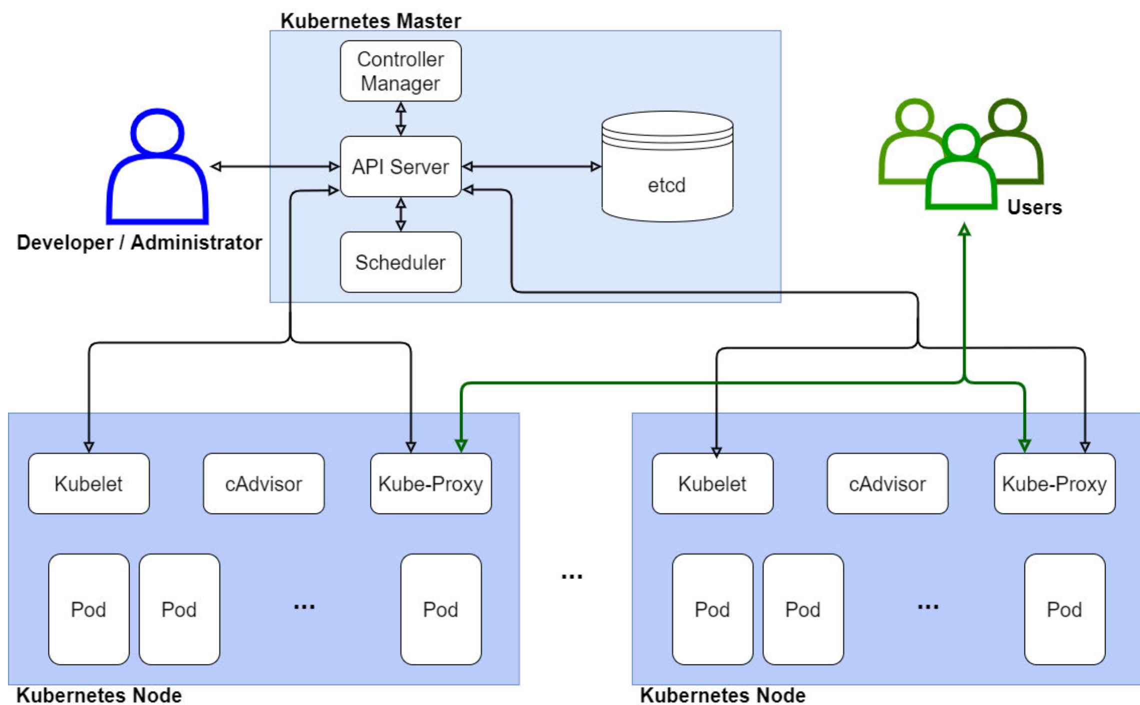

Abbildung 2: Schematischer Aufbau Kubernetes Cluster - In Anlehnung an (Kubernetes, 2019) und Wikipedia

Master-Node

API-Server

Der API-Server ist ein zentraler Prozess des Masters. Über eine REST-Schnittstelle kommuniziert er mit allen Diensten und Komponenten innerhalb und außerhalb des Clusters (vgl. Luber, 2019).

Scheduler

Der Scheduler legt auf Grund von Ressourcen fest, auf welchem Node welcher Pod läuft (vgl. Luber, 2019).

Controller Manager

Der Controller Manager ist für die Überwachung der Nodes zuständig. Falls ein Node fehlerhaft sein sollte, schichtet er die Aufgaben des fehlerhaften Nodes auf einen anderen Node um (vgl. Kubernetes, 2019).

etcd

etcd ist eine Key-Value Datenbank, die vom API-Server genutzt wird. Die Datenbank umfasst alle nötigen Konfigurationsdateien für das Cluster (vgl. Kubernetes, 2019).

Worker-Node

Pod

Ein Pod ist die kleinstmögliche Einheit in der Kubernetes Architektur. Er beinhaltet einen oder mehrere Container und läuft auf einem Node als Prozess (vgl. Luber, 2019).

Kubelet

Das Kubelet kommuniziert mit dem Master und führt seine Anweisungen zum Starten und Stoppen von Containern aus. Der Zustand der gestarteten Container wird anschließend vom Kubelet überwacht (vgl. Kubernetes, 2019).

Kube-Proxy

Der Kube-Proxy ist für die Kommunikation innerhalb und außerhalb der Nodes zuständig. Außerdem dient er als Loadbalancer für die ankommenden Anfragen (vgl. Kubernetes, 2019).

cAdvisor

Der cAdvisor bietet gesammelte Informationen über die genutzten Ressourcen eines Containers an (vgl. Luber, 2019).

2.3.2 Kubernetes-Deployment

In einem Deployment wird der angestrebte Zustand einer Anwendung beschrieben. Es umfasst beliebig viele Pods, eines oder mehrerer Nodes und schließt sie zu einer logischen Einheit zusammen. Das Deployment übernimmt die Verwaltung der Pods und kann diese Starten und auch Beenden. In einer YAML-Konfigurationsdatei wird anhand von Key-Value Paaren der angestrebte Zustand beschrieben. Sobald die Konfigurationsdatei übergeben wurde, wird diese vom Controller Manager umgesetzt (vgl. Kubernetes, 2020a). Im Quellcode 3 sieht man ein Beispiel für eine Deployment-Datei. Jedes Deployment muss zu Beginn eine API Version angeben und sich als Deployment mit *kind: Deployment* beschreiben. In den Metadaten wird der Inhalt des Deployments beschrieben. Unter den *specs* wird die eigentliche Konfiguration vorgenommen. In diesem Beispiel wurden unter anderem Replicas angegeben.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysite
  labels:
    app: mysite
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysite
  template:
    metadata:
      labels:
        app: mysite
    spec:
      containers:
        - name: mysite
          image: philadam/hello:v1
          ports:
            - containerPort: 80
```

Quellcode 3: Kubernetes Deployment

2.3.3 Kubernetes-Service

In Kubernetes sind Services für die Kommunikation zwischen den Pods zuständig. Da Pods dynamisch gestartet und terminiert werden, ist es nicht möglich über die dynamisch vergebene IP-Adresse eine konstante Kommunikation aufzubauen. Services steuern die Kommunikation nicht über konkrete IP-Adressen, sondern über Labels. Der Datenverkehr wird somit an den Pod mit dem entsprechenden Label weitergeleitet. Ähnlich wie ein Deployment werden Services ebenfalls anhand einer YAML-Datei konfiguriert (vgl. Kubernetes, 2020b).

2.3.4 Minikube

Minkube ist ein Single-Node Kubernetes-Cluster, welches für die lokale Entwicklung auf einen Rechner entwickelt wurde. Das Single-Node Cluster läuft in einer virtuellen Maschine, meist VirtualBox, und unterstützt dabei folgende zentralen Funktionen von Kubernetes:

- DNS
- NodePorts

- ConfigMaps and Secrets
- Dashboards
- Docker als Container Laufzeitumgebung
- Unterstützung von CNI (Container Network Interface)
- Ingress

Minikube lässt sich über einen einfachen Start- und Stopp-Befehl starten und beenden. Wie bei einem richtigen Kubernetes-Cluster, wird Minikube über das Command Line Interface *kubectl* gesteuert. Minikube lässt sich mit unterschiedlich vielen CPUs und Arbeitsspeicher konfigurieren und somit an unterschiedliche Lastsituationen anpassen. Bei fordernden Anwendungen kann es jedoch schnell zu einer Limitierung durch die verfügbaren lokalen Ressourcen kommen. Außerdem unterscheidet sich die Minkube Umgebung meist von der produktiven Umgebung. Minikube eignet sich demnach nicht für komplexe Anwendungen, die mehr als einen Node benötigen. (vgl. Wagner, 2019). Ein großer Vorteil von lokalen Clustern ist der Kostenfaktor. Im Vergleich zu remote Clustern fallen keine Kosten durch Nutzung eines Cloud-Services an. Durch den geringen administrativen Aufwand und den Kostenvorteil ist Minikube nichtsdestotrotz für den täglichen Gebrauch und für Testzwecke ein hilfreiches Werkzeug für die Entwicklung mit Kubernetes (vgl. Gentle, et al., 2019 S. 54).

2.4 Agile Softwareentwicklung

2.4.1 Continuous Integration

Continuous Integration, auch CI abgekürzt, ist eine Entwicklungsmethodik, bei der die Mitglieder eines Entwicklungsteams ihren Code fortlaufend in ein gemeinsames Verzeichnis integrieren. Dies geschieht pro Entwickler mehrfach am Tag. Bei jeder Integration wird das System automatisch neu gebaut und anschließend durch Tests validiert. Kommt es zu Problemen während der Integration, lässt sich einfach zu einer fehlerfreien Version zurückspringen (vgl. Flower, 2006). Das vergleichsweise schnelle Feedback verhindert zeitaufwendige Rückkopplungsprozesse und Bug-Beseitigungen, die entstehen können, wenn

auf einer fehlerhaften Version aufgebaut wird. Die Arbeitseffizienz und die Softwarequalität wird somit gesteigert (vgl. HJL, 2018).

2.4.2 Continuous Deployment

Wie in Abb. 3 illustriert ist Continuous Deployment, auch CD abgekürzt, eine Erweiterung der Continuous Integration und geht über die Ansätze des automatisierten Bau- und Testverfahrens hinaus. Beim Continuous Deployment wird die Anwendung, nach der erfolgreichen Integration, dem Kunden auf der Produktionsebene automatisiert bereitgestellt. Dies hat den Vorteil, dass Auslieferungszeiten und somit die „time to market" minimal gehalten werden. Somit ist es möglich schnell auf die Wünsche oder Bug-Reports des Kunden einzugehen. Eine CD Pipeline bringt aber auch das Risiko fehlerhaften Code bereitzustellen mit sich. Tests in einer CD Pipeline müssen demnach ausgereift sein und eine fehlerfreie Anwendung garantieren können (vgl. Pittet, [o. Jahr]).

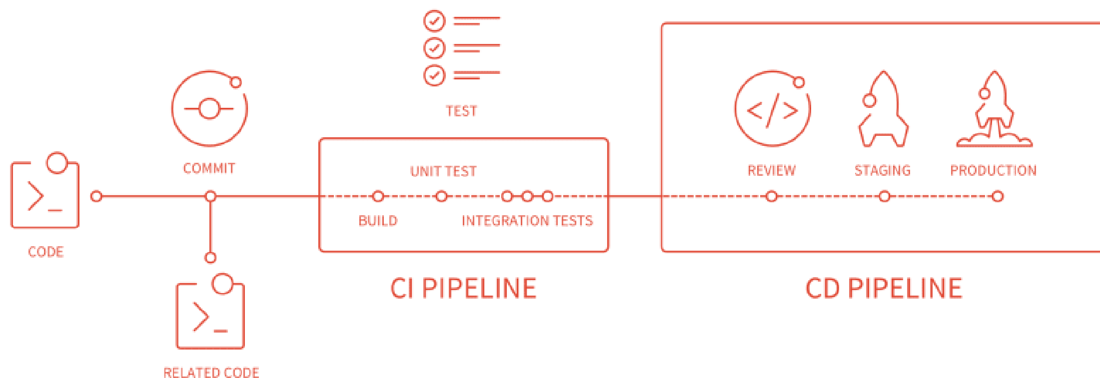

Abbildung 3: CI/CD Pipeline (GitLab, [o. Jahr])

2.4.3 GitLab

GitLab ist ein Version Control System (VCS), das zur Versionsverwaltung von Software Projekten mittels Git genutzt wird. Außerdem bietet GitLab einige zusätzliche Funktionen für eine agile Softwareentwicklung, wie z.B. das Erstellen und Verwalten von User Stories. GitLab ermöglicht außerdem die Konfiguration einer CI/CD Pipeline. Im Rahmen dieser Bachelorarbeit wird GitLab als Versionierungswerkzeug genutzt, da es von der HAW Hamburg für die Studierenden bereitgestellt wird.

CI/CD Pipeline

In GitLab lassen sich über die *.gitlab-ci.yaml* Datei verschiedene Jobs für die CI Pipeline konfigurieren. Die dort definierten Jobs werden dann nach jedem Commit durch den sogenannten GitLab-Runner ausgeführt. Eine Pipeline besteht dabei aus mehreren Stages, welche sukzessive ausgeführt werden. Beinhaltet eine Stage allerdings mehrere Jobs, werden diese parallel ausgeführt. Meist beinhaltet eine CI Pipeline den Build- und Testprozess. Bei einer CD Pipeline wird die CI-Pipeline um die Deployment Stage erweitert. Kommt es zu einem Fehler einer Stage, wird der Prozess abgebrochen und keine der nachfolgenden Stages werden mehr ausgeführt. Im Quellcode 4: Beispiel - GitLab-CI sieht man ein Beispiel einer *.gitlab-ci.yaml* Datei für ein Maven Projekt. Die beiden Jobs “build“ und “test“ gehören der gleichnamigen Stage an und werden sukzessive ausgeführt.

```yaml
image: 'maven:latest'

stages:
  - build
  - test
build:
  stage: build
  script:
    - 'mvn compile'
test:
  stage: test
  script:
    - 'mvn test'
```

Quellcode 4: Beispiel - GitLab-CI

GitLab- Registry

Das GitLab-Register bietet die Möglichkeit Docker Images in einem GitLab Projekt zu verwalten. GitLab kann somit zeitgleich als Version Control System und als Container Register genutzt werden. Über die GitLab-CI oder über die Kommandozeile können, nach dem man sich mittels *docker login* verifiziert hat, Images gepullt oder gepusht werden (vgl. Pundsack, 2016).

3 Analyse

3.1 Softwareentwicklung im Cluster

3.1.1 Cluster in der Entwicklung

In der Entwicklung mit Kubernetes werden mehrere Bereitstellungsumgebungen definiert. Meist werden dabei die drei Umgebungen: Dev (Development), Test und Prod (Production) bereitgestellt. In dieser Arbeit wird der Fokus auf der Dev Umgebung liegen.

Für die Entwicklung mit einem Kubernetes-Cluster gibt es grundsätzlich zwei Möglichkeiten. Entweder in einem lokalen Cluster oder in einem remote Cluster in der Cloud. Die Vor- und Nachteile eines lokalen Clusters wurden bereits in Kapitel 2.3.4 aufgeführt und diskutiert. Ein remote Cluster wird meist mit Hilfe eines großen Cloud-Anbieters, wie bspw. der Google-Cloud, realisiert. Dabei wird zwischen zwei möglichen Ansätzen differenziert. Entweder erhält jeder Entwickler ein eigenes Cluster oder das Entwicklerteam teilt sich ein Cluster. Bei der Bereitstellung eines Clusters für jeden Entwickler kommt es unweigerlich zu erhöhten administrativen Aufwänden und Kosten. Der zweite Ansatz beschreibt das sogenannte shared-Cluster. Das Cluster wird bei diesem Ansatz durch Namespaces separiert, sodass die Ressourcen des Clusters auf unterschiedliche Nutzer aufgeteilt werden können. Die Namespaces sind dabei logisch voneinander getrennt. Der administrative Aufwand und die Kosten sind in diesem Fall deutlich geringer. Dennoch profitieren beide Ansätzen davon, dass die Entwicklungsumgebung sich kaum von der späteren produktiven Umgebung unterscheidet. Außerdem wird die Entwicklung nicht, wie bei einem lokalen Cluster, durch die Ressourcen des lokalen Rechners limitiert (Gentle, et al., 2019 S. 54 f.). Beide Möglichkeiten sind in der Entwicklung denkbar. Eine Entscheidung muss je nach Use-Case und verfügbaren Ressourcen getroffen werden.

3.1.2 Der Bereitstellungsprozess und seine Hindernisse

Der Einsatz der Container-Orchestrierungstechnologie Kubernetes nimmt in den Unternehmen kontinuierlich zu. Der Einsatz von Kubernetes hat den Vorteil, dass bereits in der Entwicklung die Systeme an die spätere produktive Umgebung in einem Cluster angepasst werden können. Außerdem entlastet Kubernetes durch Administration und Orchestrierung die Projekte bereits in der Entwicklungsphase. Obwohl die Nutzung von Kubernetes aus der Sicht des Managements oft gewünscht wird, kann es für Entwicklungsteams eine große Herausforderung darstellen. Die erste Hürde liegt in der Komplexität von Kubernetes. Es ist ein sehr mächtiges Werkzeug, welches einiges an Erfahrung benötigt, bis es gemeistert werden kann. Nicht jeder Entwickler bringt diese Erfahrungen mit. Eine weitere Hürde, die sich in der Entwicklung mit Kubernetes offenbart, ist der kontinuierliche Bereitstellungsprozess. Aufgrund der wiederholten Bereitstellung schon während der Entwicklungsphase, wird dieser zu einer zeitintensiven und fehleranfälligen Belastung. Deshalb steht er im Fokus dieser Bachelorarbeit. In der traditionellen Softwareentwicklung besteht der Softwareentwicklungsprozess aus den vier Schritten:

1. Code schreiben
2. Build Code
3. Code bereitstellen
4. Testen und ggf. Fehler identifizieren

Dabei wird das System am Ende der Entwicklungsphase meist über eine virtuelle Maschine auf einem Server bereitgestellt. Im Cloud nativen Umfeld im Zusammenspiel mit einem remote Cluster wird der Bereitstellungsprozess erweitert (bei lokalen Clustern entfällt ggf. Schritt 4):

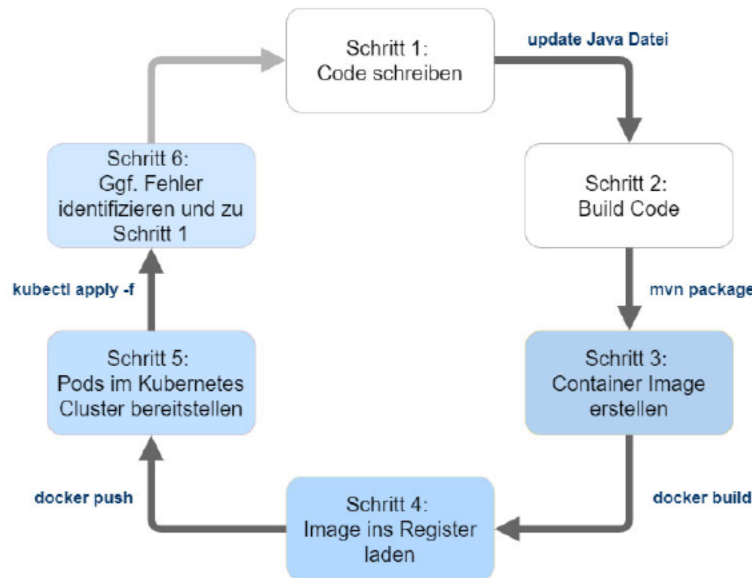

Abbildung 4 Entwicklungszyklus im Container nativen Umfeld (Eigene Darstellung)

In Abb. 4 wird der Entwicklungszyklus anhand eines Maven Projekts im Cloud nativen Umfeld illustriert. Die ersten zwei Schritt unterscheiden sich dabei nicht von der traditionellen Softwareentwicklung. Erst wenn es um die Bereitstellung geht, zeigen sich die Unterschiede. Nach dem der Build-Prozess abgeschlossen ist, wird die Anwendung nicht direkt auf einem Server mit Hilfe einer virtuellen Maschine bereitgestellt, sondern es wird zunächst ein neues Container-Image mittels des CLI-Befehles *docker build* erstellt. Anschließend wird das neue Image über den Befehl *docker push* in ein Register geladen. Über den *kubectl* Befehl *kubectl apply -f <deployment.yaml>* lassen sich die Pods im Cluster aktualisieren oder neu erstellen. Die Änderungen werden nun im Cluster sichtbar (vgl. Boxell, 2019). Werden diese Schritte jedes Mal vom Entwickler händisch ausgeführt, kann ggf. durch die hohe Anzahl an Wiederholungen am Tag ein nicht zu unterschätzender Aufwand entstehen. Da es sich im Cloud nativen Umfeld meist um Microservice-Architekturen handelt und jeder Microservice separat bereitgestellt werden muss, führt dies zu einer erhöhten Belastung. Microservices arbeiten zwar autark, dennoch wird für die Prüfung der Funktionalität des Gesamtsystems jeder Microservice benötigt. Aufgrund dieser Interdependenz ist eine effiziente Software-

Entwicklung nur möglich, wenn alle Dienste in derselben Umgebung bereitgestellt werden. Folgende Probleme ergeben sich dabei für den Entwickler:

Zeitverlust
Jedes manuelle Bereitstellen kostet Zeit und ist vor allem eine Unterbrechung des Workflows. Vor allem bei größeren Systemen kann die Komplexität dazu führen, dass der Bereitstellungsprozess zum Zeitfresser wird. Entwickler setzen sich somit ggf. mehr mit dem Bereitstellungsprozess auseinander als mit der eigentlichen Softwareentwicklung.

Fehler
Jeder Entwickler stellt täglich mehrere Services in einen Entwicklungscluster zu Verfügung. Durch die hohe Redundanz dieser Tätigkeit, kann es schnell zu Fehlern kommen.

Verlängerte Feedbackschleife
Durch den Bereitstellungsprozess und dem Ausführen im Cluster erhält jeder Entwickler erst verzögert Feedback zu den vorgenommenen Änderungen. Um jedoch in der Entwicklung möglichst effizient zu arbeiten, ist es wichtig genau diese so kurz wie möglich zu halten.

Debugging
Durch die Modularisierung des Systems in eigene Microservices und der Ausführung in einem Cluster wird das Debugging erschwert. Die Log-Dateien müssen manuell von jedem Service geladen und anschließend untersucht werden.

3.1.3 CI/CD Pipeline in der Entwicklung

Eine CI/CD-Pipeline bietet durch ihre automatisierte Bereitstellung vermeidlich Abhilfe im dem sich wiederholenden Bereitstellungsprozess während der Entwicklung. Dies ist jedoch nicht der Fall. Bei einer Pipeline handelt es sich um eine *post-commit* Variante der Bereitstellung. Die Änderungen im Quellcode werden demnach erst nach dem Einchecken in

einem VC System im Cluster wirksam. Es wäre somit nötig, jede Änderung in das VC System einzuchecken. Eine Pipeline würde zwar die Fehleranfälligkeit dabei minimieren jedoch keines der anderen in Kapitel 3.1.2 aufgeführten Probleme lösen. Es würde sogar zu einem größeren Zeitverlust kommen, da Pipelines meist durch einen dahinterliegenden Server geplant und ausgeführt werden. Dies kann, je nach Auslastung und Pipeline, mehrere Minuten dauern. Eine Pipeline ist demnach kein performantes und vor allem kein ratsames Werkzeug, um Quellcode Änderungen automatisiert im Entwicklungscluster bereitstellen zu lassen.

3.2 Werkzeuge zur Entlastung der Entwickler

Es gibt zahlreiche Werkzeuge, die die Entwicklung mit Kubernetes erleichtern sollen. Einige widmen sich, darunter mit unterschiedlichen Ansätzen, dem Problem des Bereitstellungsprozesses. Ein Ansatz ist die Verbannung des Bereitstellungsprozesses aus der Entwicklungsphase. Ein Beispiel hierfür wäre das Werkzeug Telepresence. Mit Hilfe von Telepresence lässt sich ein Service lokal ausführen, während er sich mit dem remote Cluster verbindet. So wird das Ausführen im Cluster simuliert und der Service kann lokal entwickelt und debuggt werden (vgl. Telepresence, [o. Jahr]). Dies hat jedoch den Nachteil, dass es zu Verzögerungen in der Kommunikation zwischen den Services kommen kann. Außerdem lassen sich persistente Daten nur erschwert integrieren (vgl. Gentle, et al., 2019 S. 56). Aufgrund dessen haben sich seit einiger Zeit neue Werkzeuge in der Entwicklung mit Kubernetes etabliert. Der Schwerpunkt der Werkzeuge liegt auf der Automatisierung des Bereitstellungsprozesses in der Entwicklung. Durch die Automatisierung entfällt die direkte Kommunikation mittels *kubectl* zwischen dem Entwickler und dem Cluster. Diese wird von den Werkzeugen übernommen. Dabei werden die Werkzeuge ausschließlich auf den lokalen Entwicklungsrechnern installiert und ausgeführt (Ausnahme Garden). Das Cluster ist bei Verwendung der Werkzeuge nicht direkt betroffen und benötigt keine zusätzliche Installation und Konfiguration. Die Werkzeuge beobachten dabei den Quellcode und stoßen den Bereitstellungsprozess nach jeder Änderung im Code an. Damit wird das Bauen, Containerisieren, ggf. Pushen in ein Register und Bereitstellen komplett automatisiert. Außerdem versprechen sie eine deutlich kürzere Feedbackschleife und somit eine deutlich

effizientere Entwicklung. Im Gegensatz zu der CI/CD Pipeline bieten sie eine *pre-commit* Variante der Bereitstellung an. Der Code wird somit vor dem Einchecken in einem VC System bereitgestellt. Die Werkzeuge liegen im Entwicklungszyklus vor einer CI/CD Pipeline und sind somit keine Alternative für eine Pipeline. Sie sind vielmehr als eine Erweiterung zu einer Pipeline zu sehen. Einige dieser Werkzeuge lassen sich aber in eine CI/CD Pipeline integrieren und können diese ggf. optimieren (siehe 4.2.5 und 4.3.5). Durch In-Cluster Entwicklung überzeugen die Werkzeuge vor allem im Punkt Performanz (vgl. Gentle, et al., 2019 S. 55 f.). Die Images werden dabei nicht nach jeder Änderung neu gebaut und bereitgestellt, sondern die veränderten Dateien werden in den laufenden Pod geladen. Wenn nötig, wird die Anwendung anschließend im Pod neu kompiliert. Dies führt zu einer deutlichen Verkürzung der benötigten Bereitstellungszeit. Des Weiteren sammeln die Werkzeuge Fehlermeldungen der einzelnen Pods und zeigen sie dem Entwickler gebündelt an, was die Fehlersuche deutlich angenehmer und effizienter gestaltet. Die Werkzeuge lassen sich über eine eigene Konfigurationsdatei an die entsprechenden Bedürfnisse anpassen. Die bisher definierten Docker und Kubernetes Dateien werden zum Teil einfach in die Konfigurationsdatei eingebunden und vom jeweiligen Werkzeug genutzt, sodass die bisherigen Konfigurationen einfach mit übernommen werden können. Entwicklern können somit von ihrem eigenen Rechner eine eigene Pipeline starten.

Im Rahmen dieser Arbeit werden die drei Werkzeuge Tilt, Skaffold und Garden untersucht und miteinander verglichen. Es gibt noch weitere Werkzeuge, die ebenfalls solche Funktionalitäten bieten, aber nicht Teil dieser Arbeit sind. Die Auswahl fiel auf diese drei, da sie unteranderem die populärsten sind und sich unterschiedliche Schwerpunkte gesetzt haben. So fokussiert sich Tilt, im Gegensatz zu den anderen beiden, vor allem auf die grafische Benutzerschnittstelle, die ein effizientes Arbeiten weiter unterstützen soll. Skaffold hingegen wurde von Google entwickelt und zählt somit zur populärsten Lösung. Außerdem lässt sich Skaffold besonders gut mit anderen Werkzeugen von Google kombinieren. Das Werkzeug Garden nutzt einen sogenannten Stack Graphen, der es ermöglicht besonders gut Abhängigkeiten zwischen den Artefakten abzubilden und das Werkzeug einzigartig macht.

3.2.1 Bewertungskriterien der Optimierungswerkzeuge

Alle drei Werkzeuge haben sich als zentrales Ziel die Entlastung und die Steigerung der Effizienz der Entwickler gesetzt. Um die Werkzeuge bewerten und vergleichen zu können, werden im Folgenden Bewertungskriterien für die Werkzeuge definiert. Die Kriterien werden an Hand eines Fallbeispiels (siehe 3.3) untersucht.

3.2.2 Kriterien

Konfigurationen

Die Einstiegsschwierigkeit ist besonders für neue Nutzer ein wichtiges Kriterium. Eine zu komplexe Einrichtung schreckt neue Nutzer schnell ab. Hier wird die Komplexität des Einrichtungsprozesses der Werkzeuge untersucht und verglichen.

Performanz

Da all die Werkzeuge ein sekundenschnelles Aktualisieren der Kubernetes Pods versprechen, ist es besonders interessant, ob ein Werkzeug durch seine Performanz heraussticht. Hier wird die Zeit verglichen, die ein Werkzeug benötigt, um die Pods zu aktualisieren.

Transparenz

Den Überblick über die ablaufenden Prozesse zu haben, ist besonders in einem Fehlerfall wichtig. Eine transparente Ausführung von allen Befehlen im Hintergrund wäre somit wünschenswert.

Lizenz und Kosten

Die Lizenzen und der damit verbundene Kostenfaktor spielen bei der Wahl eines Werkzeugs eine nicht unwesentliche Rolle. Da alle drei zu untersuchenden Werkzeuge zumindest eine Open-Source Variante ihres Werkzeuges anbieten, spielt der Kostenfaktor nur bei den Enterprise Versionen eine Rolle. Die mit einer Enterprise Version zusätzlich zu erhaltenen Funktionen gehen mit in die Bewertung ein.

Dokumentation

Eine gute und aktuelle Dokumentation ist für das effiziente Nutzen eines Werkzeuges essenziell. Aufgrund dessen ist die Güte der Dokumentation ein weiteres Bewertungskriterium.

Langlebigkeit

Die Langlebigkeit der Werkzeuge ist besonders für Unternehmen interessant. Ein Werkzeug sollte möglichst eine hohe Langlebigkeit haben und regelmäßig durch neue Versionen oder Updates aktualisiert werden.

3.3 Fallbeispiel

Anhand des folgenden Fallbeispiels werden die Werkzeuge während der Implementierung verprobt und untersucht. Bei dem Fallbeispiel handelt es sich um einen Blog-Webservice, welcher mittels Docker containerisiert und sowohl in einem lokalen als auch in einem remote Kubernetes Cluster bereitgestellt wird. Über ein Web-UI kann ein Nutzer mit dem System interagieren. Folgende Funktionalitäten werden dabei dem Nutzer vom Webservice bereitgestellt:

Ein Nutzer kann:

- sich über seine Login Daten beim Webservice anmelden
- seine persönlichen Daten bearbeiten
- die neusten Blogposts einsehen
- eigene Posts verfassen und sie ggf. wieder löschen
- Kommentare zu einem Post schreiben und sie ggf. wieder löschen
- Kommentare durch Likes bewerten

3.3.1 Architektur

Wie in Abb. 5 illustriert, besteht das System aus einer Microservice-Architektur mit vier Microservices. Das Frontend dient als Benutzerschnittstelle und interagiert ausschließlich mit

dem API-Gateway-Microservice. Durch den Gateway-Microservice werden alle verfügbaren Schnittstellen des Systems veröffentlicht. Durch die *authenticate* Schnittstelle können sich Nutzer beim System authentifizieren. Die Anfrage wird anschließend von der *API* Komponente an die *Authentication* Komponente weitergeleitet, welche anschließend Nutzerdaten vom User-Microservice anfragt und die Identität ggf. bestätigt. Über die *request Data* Schnittstelle können anschließend konkrete Daten über das Gateway von der Benutzerschnittstelle angefragt werden. Die Anfragen werden daraufhin anhand definierter Schnittstellen vom Gateway an den zuständigen Microservice weitergeleitet. Sobald das Gateway eine Antwort vom zuständigen Microservice erhalten hat, wird diese an die Benutzerschnittstelle zurückgegeben. Jeder der drei Microservices *Comment*, *Post* und *User* besitzt dabei seine eigene Datenbank, um seine Daten zu persistieren und um möglichst autark von den anderen Services arbeiten zu können.

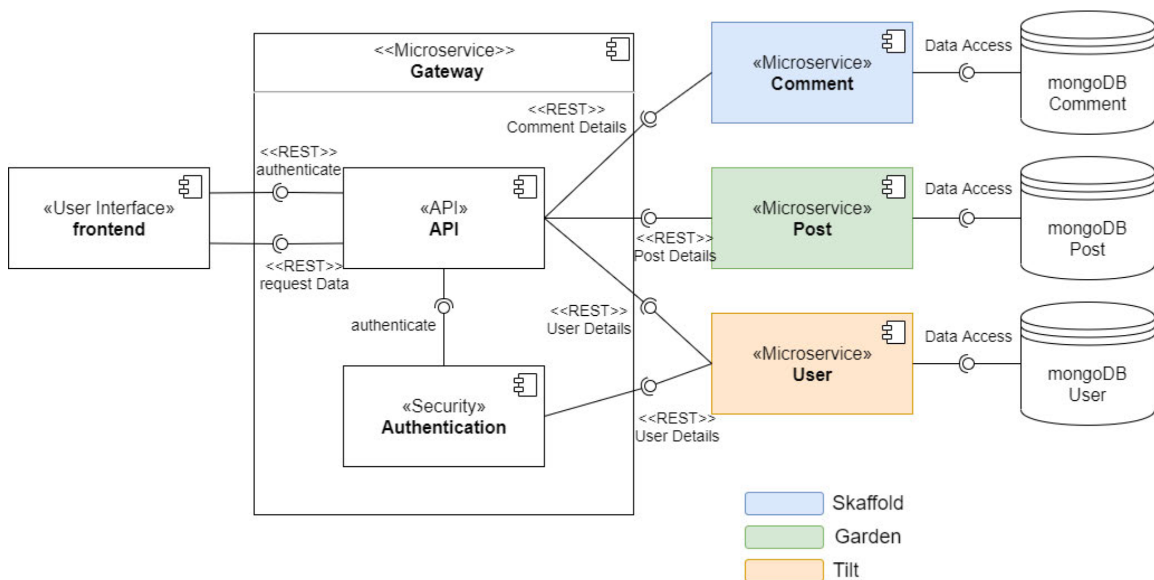

Abbildung 5: Fallbeispiel -Architektur

Die drei farblich markierten Microservices werden sukzessive mit dem jeweiligen Werkzeug implementiert. Um die volle Funktion der Werkzeuge von Beginn an nutzen zu können, werden zu Beginn der Entwicklung für jeden der drei Microservices Stubs, die statische Mock

Daten zurückgeben, erstellt. Die anderen Services wurden bereits vor dem Verproben der Werkzeuge implementiert. Die Kommunikation der Services basiert ausschließlich auf REST-Schnittstellen.

3.3.2 Technologien

Für das Erstellen der Benutzerschnittstelle wird das Webframework Angular (Version 9.1.1) genutzt. Die Microservices werden mit Hilfe des Java Frameworks Spring Boot (Version 2.3.3) in Kombination mit Maven umgesetzt. Als lokale Kubernetes Instanz wird Minikube (siehe 2.3.4) genutzt. Die Google Cloud wird als remote Host für ein remote Cluster verwendet.

4 Optimierungswerkzeuge

4.1 Tilt

Kubernetes for Prod, Tilt for Dev (Tilt, [o. Jahr]c)

Tilt ist ein Kommandozeilen-Werkzeug des Start-Ups Windmill Engineering aus New York City – USA und wurde erstmals im August 2018 veröffentlicht. Tilt wurde mit dem Ziel entwickelt Softwareentwickler im Bereich der lokalen kontinuierlichen Entwicklung und Bereitstellung von Mikroservices in Kubernetes Clustern zu unterstützen. Ähnlich wie die beiden anderen Werkzeuge Skaffold und Garden zielt Tilt auf die Verkürzung der Feedbackschleife des Softwareentwicklungsprozesses ab. Änderungen am lokalen Quellcode sollen nahe Echtzeit und automatisiert im Entwicklungscluster wirksam werden. Ein weiteres Anliegen von Tilt ist es, den Entwicklern das parallele Arbeiten mit Kubernetes in mehreren offenen Terminal-Fenstern zu ersparen. Tilt bietet hierfür eine eigene graphische Web- und Terminal-Benutzerschnittstelle an, die es ermöglicht auf einen Blick alle aktuellen Informationen über alle Microservices einer Anwendung abzulesen (siehe Kapitel 4.1.3). Die Entwickler sollen damit auch das Verhalten der einzelnen Microservices besser nachvollziehen können. Tilt fokussiert sich dabei eindeutig auf die Verwendung von Kubernetes-Clustern. Trotzdem bietet es auch die Möglichkeit, die Container über *docker-compose* oder über lokale Shell-Kommandos zu starten. Tilt lässt sich mit lokalen oder remote Kubernetes-Clustern konfigurieren, empfiehlt aber für Entwicklungszwecke ein lokales Cluster (vgl. Tilt, [o. Jahr]h). Durch das Verwenden eines lokalen Clusters können die Images direkt im Cluster gebaut werden, was das Einchecken des Docker Images in einem Register obsolet macht. Der Bereitstellungsprozess kann somit weiter beschleunigt werden (.vgl Tilt, [o. Jahr]a). Im Rahmen dieser Bachelorarbeit wurde die Version 0.13.2 von Tilt verwendet.

4.1.1 Bereitstellungsprozess

Der Bereitstellungsprozess von Tilt umfasst die in Abbildung 6 illustrierten Schritte.

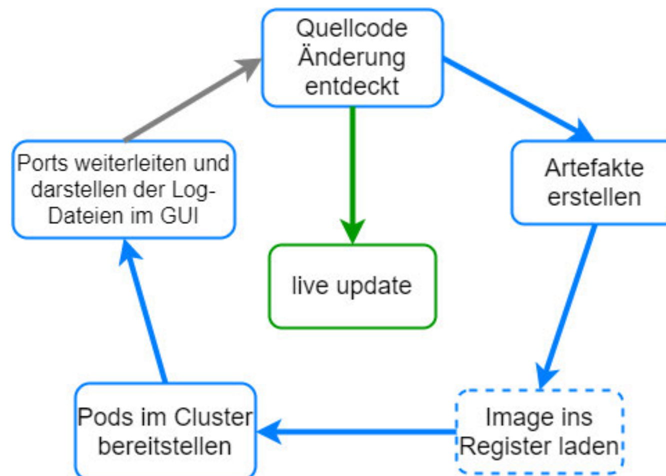

Abbildung 6: Tilt - Bereitstellungsprozess

Bei dem initialen Bereitstellen eines Services wird zunächst der gesamte Prozess automatisiert durchlaufen (hier blau). Über Docker oder einem anderen Build-Werkzeug wird ein Image lokal erstellt, welches anschließend mittels *kubectl* im Cluster bereitgestellt wird. Handelt es sich um ein remote Cluster, wird das Image vor der Bereitstellung im Cluster in ein Register geladen, sodass das remote Cluster anschließend darauf zugreifen kann. Kommt es nach der initialen Bereitstellung zu Änderungen im Quellcode, wird der gesamte Prozess nicht erneut durchlaufen, sondern es wird ein sogenanntes *live update* durchgeführt. Über die *live update* Funktion bietet Tilt die Möglichkeit einen Pod im Kubernetes Cluster zur Laufzeit anzupassen. Somit wird der bearbeitete Quellcode direkt in den laufenden Pod migriert und die Anwendung wird mit dem aktualisierten Quellcode im Pod neu gestartet. Das erneute Bauen und Bereitstellen von Images entfällt hiermit. Die benötigte Zeit zum Aktualisieren eines Pods wird somit deutlich verkürzt. (vgl. Tilt, [o. Jahr]d)

4.1.2 CLI

Die beiden Kommandozeilenbefehle *tilt up* und *tilt down* bilden die Kernfunktionalität des Werkzeuges ab. Sobald *tilt up* in einem Verzeichnis aufgerufen wird, wird die hinterlegte Konfigurationsdatei (Tiltfile) in diesem Verzeichnis mit seinen konfigurierten Schritten ausgeführt und die Anwendung wird initial im Cluster bereitgestellt. Außerdem öffnet sich die graphische Benutzerschnittstelle im Terminal und im Browser. Im Hintergrund wird zusätzlich ein sogenannter File-Watcher gestartet, der das Verzeichnis des Quellcodes überwacht. Bei jeder Änderung einer Datei im überwachten Verzeichnis wird ein *live update* ausgeführt und die Anwendung wird im Cluster aktualisiert. Über den Befehl *tilt down* lassen sich alle erstellten Ressourcen von Tilt im Cluster wieder entfernen und das Cluster befindet sich wieder im ursprünglichen Zustand. Zusätzlich bietet Tilt noch einige weitere Befehle, wie bspw. *tilt ci,* der den Bereitstellungsprozess als Batch startet, oder *tilt trigger*, welcher gezielt ein Update einer bereitgestellten Ressource triggert, an. (vgl. Tilt, [o. Jahr]i).

4.1.3 Graphische Benutzerschnittstelle

Alleinstellungsmerkmal von Tilt ist die graphische Benutzerschnittstelle. Sie ermöglicht es den Entwicklern, den aktuellen Staus der Pods mit einem Blick zu überprüfen. Wie in Abbildung 7 dargestellt, werden die definierten Ressourcen im Erfolgsfall durch eine grüne und in einem nicht erfolgreichen Fall durch eine rote Markierung dargestellt. Fehler bei der Bereitstellung können also auf einem Blick erkannt werden. Zudem enthält die Oberfläche auch Informationen zu den Namen der Pods, der Bereitstellungshistorie und den Pod-Protokollen (sowohl Build- als auch laufende Container-Protokolle). Dies ist besonders hilfreich, wenn mehrere Microservices parallel entwickelt werden. Klickt man ein Artefakt in der rechten Menüleiste an, so werden nur Log-Daten zu diesem Artefakt aufgeführt. So lassen sich die Daten gezielt filtern und eine potenzielle Fehlersuche wird noch präziser. Die Benutzeroberfläche bietet keine Informationen, die nicht auch über *kubectl* Befehle verfügbar wären. Allerdings werden die Informationen und vor allem Fehler-Informationen, die in der manuellen Bearbeitung evtl. untergehen könnten, dem Entwickler übersichtlich angezeigt.

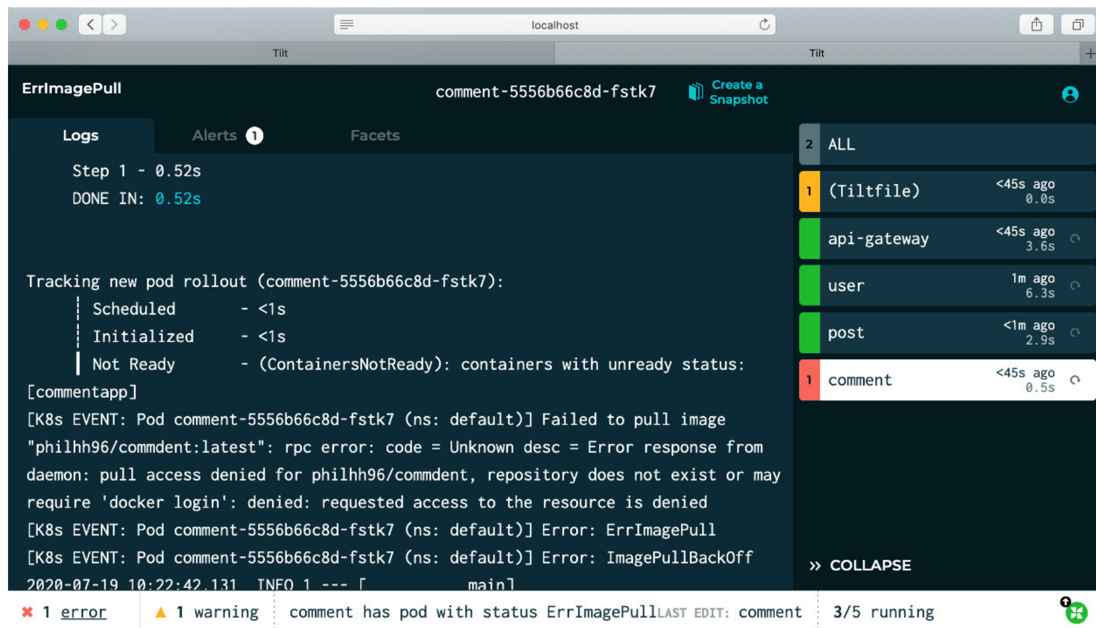

Abbildung 7: Tilt - Graphische Benutzerschnittstelle (Browser)

Die graphische Benutzerschnittelle im Terminal beinhaltet die gleichen Informationen und ähnelt optisch der Darstellung im Browser. Für ein Beispiel dieser siehe Kapitel 5.2.1.

4.1.4 Snapshots und Tilt-Cloud

Mit Hilfe eines Snapshots lassen sich Momentaufnahmen des aktuellen Tilt-Zustandes auf einem lokalen Entwicklungsrechner festhalten. Dabei wird die graphische Benutzerschnittstelle mit all ihren Informationen eingefroren, in der Tilt-Cloud abgespeichert und anschließend zugänglich gemacht. Über die Tilt-Cloud lassen sich Snapshots leicht mit anderen Entwicklern teilen. Auf diese Weise können Fehler in der Konfiguration oder sonstige Entwicklungsprobleme von mehreren Entwicklern asynchron untersucht und behoben werden (vgl. Tilt, [o. Jahr]b). Die Tilt-Cloud ist ein SaaS-Backend, das als Austauschplattform für Teammitglieder genutzt werden kann. Das Ziel der Tilt-Cloud ist das Teilen von Erfahrungen, Konfigurationen und Problemen innerhalb eines Entwicklerteams (.vgl Bentley, 2020). Die Tilt-Cloud ist über ein GitHub-Account frei zugänglich.

4.1.5 Konfiguration

Für das Nutzen von Tilt wird die Installation von *kubectl* und *Docker-Desktop* vorausgesetzt. Für die *best practise* Konfiguration werden außerdem die Kommandozeilenwerkzeuge *rsync* und *unzip* benötigt. Tilt lässt sich anschließend über das Terminal als Binärdatei installieren. Die Konfiguration von Tilt für ein Softwareprojekt geschieht anschließend über die Konfigurationsdatei (Tiltfile). Dabei können beliebig viele Microservices in einem Tiltfile definiert werden. Im Vergleich zu den anderen Werkzeugen, wird keine YAML-Datei zur Konfiguration verwendet, sondern eine Starlark-Datei. Starlark ist ein Dialekt der Programmiersprache Python und ist für das Definieren von Konfigurationsdateien ausgelegt. Bei einer Starlark-Datei handelt es sich nicht nur ausschließlich um eine Konfigurationsdatei, sondern um ein eigenes Programm (.vgl Tilt, [o. Jahr]e). Dies hat den Vorteil das Programmiersprachen typische Anweisungen, wie Schleifen und If-Else-Anweisungen, genutzt werden können. Konfigurationen müssen also nicht statisch definiert werden, sondern können zur Laufzeit aufgeschlüsselt werden. Tilt nutzt als Default *kubectl* mit seinem konfigurierten Cluster. Außerdem unterstützt Tilt auch die Verwendung von Helm und Kustomize, zwei Hilfswerkzeuge für die Bereitstellung komplexer Kubernetes Anwendungen. Möchte man das Cluster oder das verwendete Hilfswerkzeug für ein Projekt spezifisch anpassen, lässt sich dies im Tilfile konfigurieren. Im Quellcode 5 wird ein exemplarisches Tiltfile für eine Java Maven Implementierung gezeigt. Diese Basis-Konfiguration besteht aus vier Funktionsaufrufen und verwendet nicht die Funktion des *live updates.* Aus Verständnisgründen wird diese der „*best practice*“ Konfiguration vorgeschoben.

Basis Konfiguration

Bei der Basiskonfiguration wird bei jeder Änderung im Quellcodeverzeichnis eine neue Jar-Datei erstellt, die anschließend über ein neues Docker-Image im Cluster bereitgestellt wird.

```
local_resource('example-java-compiler', 'mvn package',
   deps=['src'])
docker_build('example-java-image', '.')
k8s_yaml('kubernetes.yaml')
k8s_resource('example-java', port_forwards=8000,
   resource_deps=['example-java-compiler'])
```

Quellcode 5: Tilt - Basis-Tiltfile

Der erste Funktionsaufruf der Datei *local_resource(<wählbarer Name>, <Befehl>, <[zu überwachende Pfade] >)* dient zum automatisierten Bauen einer Jar-Datei. Sobald eine Datei in einem der übergebenen Pfade bearbeitet wurde (hier der ganze *src* Ordner), wird der davor definierte Befehl (in diesem Fall der Maven-Build-Befehl) ausgeführt. Über den Aufruf *docker_build(<Image-Name>, <Pfad zum Dockerfile>)* wird anschließend ein neues Image erstellt. Das Dockerfile (siehe Kapitel 2.2.2) kopiert nun die erstellte Jar-Datei in das Image. Das zuvor definierte Kubernetes-Manifest, bestehend aus einem Deployment und einem Service, wird anschließend über den Befehl *k8s_yaml(<Manifest>)"* ausgeführt. Der letzte Befehl *k8s_resource(<Name des Kubernetes-Deployments>, <Portweiterleitung>, [<Abhängigkeiten>])* richtet zum einen eine Portweiterleitung ein und zum anderen definiert er gleichzeitig die Abhängigkeit zum Build-Prozess. Das Image wird demnach erst bereitgestellt, nachdem der Build-Prozess mindestens einmal erfolgreich abgeschlossen wurde. Der Entwickler kann anschließend die Schnittstellen der Anwendung über *localhost:8000* ansprechen.

Best Practice Konfiguration

Die im Folgenden vorgestellte *best paractise* Konfiguration für ein Java Maven Projekt nutzt die *live update* Funktion und orientiert sich an der empfohlenen Konfiguration der Tilt Dokumentation (vgl. Tilt, [o. Jahr]f). Sobald eine Änderung im Quellcode vorgenommen wird, wird ausschließlich die bearbeitete Quellcode-Datei in den laufenden Pod im Cluster migriert. Durch einen Neustart der Anwendung im Pod wird die Änderung im Cluster wirksam.

Um die *live update* Funktion nutzen zu können, muss im Gegensatz zur Basis-Konfiguration, zunächst das Dockerfile angepasst werden. Wie in Quellcode 6 dargestellt, wird nicht die Jar-Datei selbst in das Image geladen, sondern die entpackten Dateien. Dadurch können die einzelnen Dateien nach Anzahl der Zugriffe sortiert werden. Dies führt zu einer Optimierung des Caches und somit auch zu einer Beschleunigung des Bereitstellungsprozesses. Außerdem lassen sich so, für den nächsten Schritt des *live updates,* einzelne Dateien in das Image laden. Als *Entrypoint* wird nun die Mainklasse direkt angegeben, da keine ausführbare Jar-Datei mehr verwendet wird. (.vgl Tilt, [o. Jahr]f)

```
FROM openjdk:11

WORKDIR /app
ADD BOOT-INF/lib /app/lib
ADD META-INF/lib /app/META-INF
ADD BOOT-INF/classes /app

ENTRYPOINT java -cp .:./lib/* dev.tilt.example.ExampleApplication
```

Quellcode 6: Tilt - Überarbeitetes Dockerfile (Tilt, 2020)

Als nächster Schritt muss das Tiltfile angepasst werden. In Quellcode 7 ist das überarbeitete Tiltfile dargestellt. Da das Dockerfile nun eine entpackte Jar-Datei erwartet, wird das Bauen der Jar, welches in der Funktion *„local_resource(<wählbarer Name>, <Befehl>, <[Pfad] >)"* definiert ist, um einige Kommandozeilenbefehle erweitert. So wird die erstellte Jar-Datei über den *unzip* Befehl entpackt und anschließend mittels *rsync* in den Zielordner synchronisiert, welcher vom *live update* überwacht wird. Dabei ist Synchronisieren der entpackten Dateien in einen neuen Ordner nötig, da beim Entpacken der Jar alle Dateien neu erstellten werden und einen neuen Zeitstempel erhalten. Tilt könnte dabei nicht zwischen veränderten und nicht veränderten Dateien differenzieren, sondern würde beim *live update* zu Lasten der Performanz jedes Mal alle Dateien in den Pod migrieren. Durch den Befehl *rsync --checksum* kann dies verhindert werden. Es werden dann nur die Dateien in den von Tilt überwachten Ordner synchronisiert, die entweder neu sind oder sich in der Größe gegenüber der Vorversion verändert haben. Tilt registriert somit nur die wirklich bearbeiteten oder neuen

Dateien und migriert diese in den laufenden Pod. Da nun nicht mehr bei jedem Update ein neues Image erstellt werden muss, wird die ursprüngliche Funktion *docker_build* durch die Funktion *docker_build_with_restart(<Image Name>, <Pfad zur entpackten Jar>, <Neustart Befehl>, <Pfad zum Dockerfile>, [<zu überwachende Pfade >])* ersetzt. Diese Funktion gehört nicht zu den Standartfunktionen und muss deshalb zu Beginn des Tiltfiles geladen werden. Ein *live update* wird immer dann durchgeführt, wenn sich mindestens eine Datei unter den drei übergebenen *sync* Pfaden ändert. Der *Entrypoint* gibt dabei den auszuführenden Befehl bei einem Neustart an.

```
load('ext://restart_process', 'docker_build_with_restart')
local_resource(                               Jar-Datei erstellen und entpacken
   'example-java-compile',
   'mvn package && ' +
   'unzip -o target/example-0.0.1-SNAPSHOT.jar -d
   target/jar-staging && ' +
   'rsync --delete --inplace --checksum -r target/jar-staging/
   target/jar',
   deps=['src'],
   resource_deps = ['deploy'])

docker_build_with_restart(                    live update mit Neustart der Anwendung
  'example-java-image',
  './target/jar',
  entrypoint=['java', '-noverify', '-cp', '.:./lib/*',
  'dev.tilt.example.ExampleApplication'],
  dockerfile='./Dockerfile',
  live_update=[
    sync('./target/jar/BOOT-INF/lib', '/app/lib'),
    sync('./ target /jar/META-INF', '/app/META-INF'),
    sync('./ target /jar/BOOT-INF/classes', '/app'),],)

k8s_yaml('kubernetes.yaml')
k8s_resource('example-java', port_forwards=8000,
   resource_deps=['example-java-compiler'])
```

Quellcode 7: Tilt - Best Practise Tiltfile (Tilt, 2020)

Die beiden letzten Aufrufe unterscheiden sich nicht von der Basis-Konfiguration. Hier wird das Kubernetes Manifest übergeben, ausgeführt und eine Portweiterleitung auf den localhost:8000 vorgenommen. Außerdem besteht auch hier wieder die Abhängigkeit zum Build-Prozess, sodass die Bereitstellung erst angestoßen wird, wenn der Build-Prozess mindestens einmal erfolgreich abgeschlossen wurde (vgl.Tilt, [o. Jahr]f).

Tilt bietet in seiner Dokumentation auch noch weitere *best practice* Konfigurationen für andere Programmiersprachen an. Des Weiteren bietet Tilt auch die Verwendung von dem Build-Werkzeug jib von Google an, welches die Verwendung eines Dockerfiles obsolet machen würde. Dies würde eine Konfiguration verkürzen, dennoch empfiehlt Tilt die oben vorgestellte Konfiguration.

4.1.6 Lizenz

Tilt wird als Open Source Projekt angeboten und ist frei zugänglich und erweiterbar. Es wird zusätzlich eine Enterprise Version angeboten. Die Enterprise Version umfasst vor allem den Support der Tilt Entwickler, die Bereitstellung von *on promise* Versionen und die Priorisierung von Feature-Anfragen. Zudem dürfen beliebig viele Snapshots in der Tilt-Cloud erstellt werden, welches im Gegensatz dazu in der freien Version limitiert ist. (.vgl Tilt, [o. Jahr]g)

4.2 Skaffold

> *Skaffold has grown to be much more than just a build and deployment tool—instead, it's become a tool to increase developer velocity and productivity. (Kubala, et al., 2019)*

Skaffold ist ein von Google entwickeltes Open-Source Kommandozeilen Werkzeug, welches erstmals im März 2018 vorgestellt wurde. Seit Ende 2019 befindet sich Skaffold in dem GA (General Availability) Status und ist somit offiziell für den produktiven Einsatz freigegeben. Aufgrund des Herstellers Google und die damit verbundene Reichweite, ist Skaffold das

populärste der drei vorgestellten Werkzeuge. Wie auch die anderen Werkzeuge möchte Skaffold die Arbeit mit Kubernetes erleichtern und übernimmt das Bauen, das Pushen und die Bereitstellung einer Anwendung in Kubernetes. Allerdings geht Skaffold noch über diese Funktionen hinaus und bietet neben der möglichen Integration in eine CI/CD Pipeline noch einige weiter Funktionen an (vgl. Kubala, et al., 2019). So lassen sich beispielsweise mit Hilfe von Skaffold die Anwendungen im Pod live debuggen, was das Entwicklern im Cluster weiter vereinfacht. Außerdem bietet Skaffold aufgrund seiner Plugin-Architektur eine große Auswahl an Konfigurationsmöglichkeiten. So lassen sich, neben zahlreichen Werkzeugen z.B. Jib, Bazel, Helm etc., auch die auszuführenden Bereitstellungsschritte modular zusammensetzen. Jeder Schritt einer Skaffold-Pipeline lässt sich optional mit dem präferierten Werkzeug ausführen. (vgl. Skaffold, 2020e) Dadurch, dass sich mehrere Bereitstellungsprofile mit unterschiedlichen Technologie-Stacks und Bereitstellungsumgebungen konfigurieren lassen, ist Skaffold noch vielseitiger einsetzbar. Im Rahmen dieser Bachelorarbeit wurde die Version 1.13.1 von Skaffold verwendet.

4.2.1 Bereitstellungsprozess

Der in Abbildung 8 dargestellte Bereitstellungsprozess von Skaffold wird im Vergleich zu dem von Tilt (Kapitel 4.1.1 - Abbildung 6) um die hervorgehobenen Schritte erweitert bzw. ersetzt. Skaffold bietet aufgrund seiner Plugin-Architektur zusätzlich die Möglichkeit den Bereitstellungsprozess, durch das Überspringen einzelner Schritte und das Verwenden unterschiedlichster Werkzeuge, genau auf die jeweiligen Bedürfnisse anzupassen. So lässt sich bereits beim Build-Prozess zwischen Dockerfile, Jib, Bazel, Custom Scripts, und Cloud nativen Buildpacks wählen. Auch bei der Bereitstellung im Kubernetes Cluster bietet Skaffold eine Auswahl von verschiedenen Werkzeugen, wie Helm, kubectl und kustomize, an (vgl. Skaffold, 2020e).

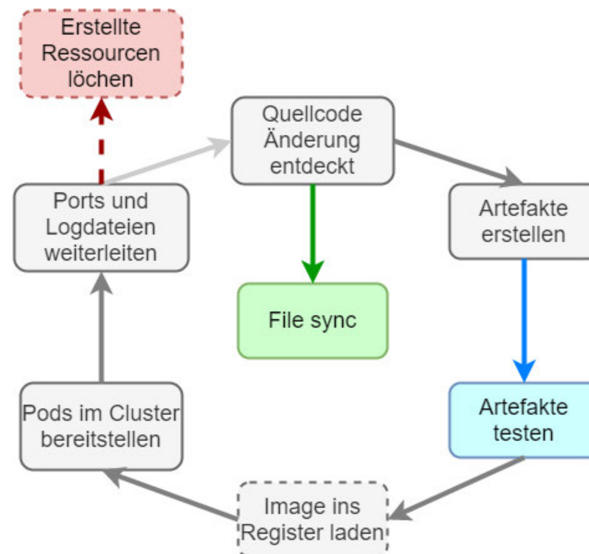

Abbildung 8: Skaffold - Bereitstellungsprozess

Im Folgendem werden die drei hervorgehobenen Schritte erläutert.

File Sync

Wie Tilt mit dem *live update,* bietet auch Skaffold die Möglichkeit Dateien zur Laufzeit in den Pod zuladen und dort durch einen Neustart die Änderungen wirksam zu machen. Skaffold erstellt dabei eine Tar-Datei mit den modifizierten Dateien, welche durch einen File-Watcher identifiziert werden, und lädt diese in den laufenden Pod. Anschließend werden die Dateien im Pod entpackt und migriert. Dabei können die zu synchronisierenden Verzeichnisse entweder manuell oder durch Skaffold automatisch in der Konfigurationsdatei gesetzt werden. Bei der automatischen Synchronisation müssen keine konkreten Verzeichnisse angegeben werden, weder lokal noch im Pod. Dies erleichtert nochmals die Konfiguration. Der automatisierte Ansatz wird aber nur unter Verwendung von Jib oder Cloud nativen Buildpacks angeboten (vgl. Skaffold, 2020c).

Artefakte testen

Über Strukturtests können die erstellten Images von Skaffold validiert werden, bevor sie in einem Cluster bereitgestellt werden. Darunter fallen auch die Validierungen von Inhalten und

Strukturen der Docker-Container. Skaffold bietet für diesen Zweck eine optionale integrierte Testphase zwischen der Bau- und der Bereitstellungsphase in ihrer Pipeline an. Strukturtests werden je Image in der Skaffold-Konfiguration angeben. Jedes Mal, wenn ein Image neu gebaut wird, führt Skaffold die zugehörigen Strukturtests aus. Wenn die Tests fehlschlagen, fährt Skaffold nicht mit der Bereitstellung fort. Aufgrund der dadurch bedingten verschlechterten Performanz werden die Tests meist unter einem gesonderten Profil definiert, sodass die Tests nur auf Wunsch ausgeführt werden können. Für die Durchführung der Tests muss das von Google entwickelte Testframework *Container-Structure-Test* auf dem Entwicklerrechner installiert sein. Außerdem müssen für die zu testenden Images YAML-Testdateien angelegt werden (vgl. Skaffold, 2020f). In Quellcode 8 ist eine solche YAML-Testdatei für ein Java Image dargestellt. Die Test-Datei beinhaltet drei unterschiedliche Tests und Testarten. Über den ersten Test *commandTest* wird zunächst die verwendete Java Version im Container geprüft. Der *fileExistenceTest* prüft anschließend die Existenz der auszuführenden Jar-Datei und seine Berechtigungen. Als letztes wird ein *metadataTest* ausgeführt, welcher die Angaben in dem verwendeten Dockerfile überprüft.

```yaml
schemaVersion: '2.0.0'
commandTests:
  - name: 'java'
    command: 'java'
    args: ['-version']
    expectedError: ['openjdk version \"11\..*']
    exitCode: 0

fileExistenceTests:
  - name: 'application'
    path: '/app/app.jar'
    permissions: '-rw-r--r--'
    shouldExist: true

metadataTest:
  workdir: "/app"
  exposedPorts: ["8080"]
  entrypoint: ["java", "-jar", "/app/app.jar"]
  cmd: []
```

Quellcode 8: Skaffold - Container Structure Test

Löschen der Ressourcen

Im Gegensatz zu Tilt löscht Skaffold per Default alle erstellten Ressourcen nach Abbruch (Strg + C) des Kommandozeilenbefehls. Dies kann auf Wunsch durch die Angabe des Parameters --*no-prune* unterbunden werden.

4.2.2 CLI

Der Kommandozeilen Befehl *skaffold dev* ist der zentrale Befehl von Skaffold und äquivalent zu Tilts *tilt up*. Über diesen Befehl wird die hinterlegte Konfigurationsdatei (skaffold.yaml) im aktuellen Verzeichnis ausgeführt. Auch Skaffold startet dabei, ebenso wie Tilt, einen File-Watcher, der das definierte Quellcode-Verzeichnisse auf Änderungen überwacht. Bei jeder Änderung des Quellcodes wird der Pod im Kubernetes Cluster entweder durch das Bauen eines neuen Images oder das Synchronisieren der veränderten Dateien aktualisiert (vgl. Skaffold, 2020a). Skaffold bietet keine eigene graphische Benutzerschnittstelle an, sondern leitet lediglich die Log-Daten des Bereitstellungsprozesses und der Anwendung an das Terminal weiter. Durch den Paramater *–port-forward* werden alle definierten Ports des Kubernetes-Manifests auf den localhost weitergeleitet, sodass der Entwickler einfacher mit der Anwendung interagieren kann.

Der Befehl *skaffold run* bietet dieselben Funktionalitäten wie *skaffold dev,* nur wird hier kein File-Watcher gestartet, sodass eine Änderung am Quellcode keine neue Bereitstellung anstößt. Die Anwendung wird also nur einmalig initial bereitgestellt. Dieser Befehl eignet sich besonders für die Integration in eine CD Pipeline. Durch die Möglichkeit der Konfiguration mehrerer unterschiedlicher Profile, lassen sich in der gleichen Konfiguration mehrere Bereitstellungsumgebungen mit unterschiedlichen Technologie-Stacks definieren (vgl.Skaffold, 2020b). Über die Befehle *skaffold build* und *skaffold deploy* lässt sich der *skaffold run* Befehl weiter unterteilen (siehe 4.2.5).

Der *skaffold debug* Befehl verhält sich ähnlich zu dem *skaffold dev* Befehl. Jedoch bietet dieser Befehl zusätzlich die Möglichkeit, die laufende Anwendung im Cluster zu debuggen. Skaffold

untersucht dabei die bereitgestellten Artefakte und stellt sprachenspezifische Debugging-Container im laufenden Pod zur Verfügung. Kubernetes-Manifeste, die auf diese Artefakte verweisen, werden dabei on-the-fly transformiert, um die Debugging-Funktionalität zur Laufzeit zu ermöglichen. Dabei werden unter anderen Umgebungsvariablen und Entrypoints hinzugefügt oder geändert. Der Debugg Modus der IDE kann somit wie bei einer lokalen Entwicklung genutzt werden. *Skaffold debug* unterstützt die Sprachen Go, NodeJS, Java (alle JVM Sprachen) und Python (vgl. Skaffold, 2020g).

Skaffold bietet neben den wichtigsten hier aufgeführten Befehlen noch weitere Kommandozeilenbefehle an, welche in dem zu untersuchenden Rahmen vernachlässigt werden können. Zu diesen Befehlen gehören bspw. *skaffold init,* welcher eine initiale Skaffold-Konfigurationsdatei erstellt, oder *skaffold delete*, der alle von Skaffold erstellten Ressourcen aus einem Cluster entfernt.

4.2.3 Profile

Skaffold bietet die Möglichkeit für den Bereitstellungsprozess unterschiedliche Profile zu definieren. Dies hat den Vorteil, dass zwischen unterschiedlichen Technologien und Bereitstellungsumgebungen schnell und einfach gewechselt werden kann. Profile lassen sich, ebenso wie die Default-Konfiguration, in der Konfigurationsdatei hinterlegen. So lässt sich Skaffold bspw. mit der Default Konfiguration für die lokale Entwicklung konfigurieren. Artefakte werden so mit dem lokalen Docker-Daemon erstellt und mit kubectl im lokalen Cluster Minikube bereitgestellt. Sobald die Anwendung später in das produktive remote Cluster bereitgestellt werden soll, kann dies einfach durch die Verwendung eines anderen Profils geschehen. Mit der Verwendung des Profils, wird die Anwendung nun bspw. mit dem Google Cloud Build gebaut und mit Helm im remote Cluster bereitgestellt. Profile lassen sich durch den Zusatz des Parametes *-p <profile>* beim Aufruf eines Befehls nutzen. Sobald ein Profil aktiviert ist, überschreibt es die als Default definierten Bereitstellungsschritte (siehe für ein Beispiel Kapitel 5.2.2). Dabei können einzelne oder auch alle Schritte der Bereitstellung überschrieben werden. Alternativ können durch sogenannte Patches auch feiner abgestufte

Änderungen am Bereitstellungsprozess vorgenommen werden. Durch einen Patch kann somit bspw. nur das genutzte Dockerfile im Build-Prozess ausgetauscht werden (vgl. Skaffold, 2020d). Für die Integration in die CI/CD Pipeline ist dies besonders hilfreich. So lässt sich in der Konfigurationsdatei einfach ein Pipeline Profil einrichten, das die gewünschten Werkzeuge nutzt, um die Anwendung im produktiven Cluster bereitzustellen.

4.2.4 Konfiguration

Skaffold lässt sich, wie auch Tilt, über das Terminal als Binärdatei installieren. Die Konfiguration für eine Anwendung erfolgt anschließend über eine YAML-Konfigurationsdatei (skaffold.yaml). Dabei können auch hier mehrere Microservices in einer Datei konfiguriert werden, sodass bei Starten von Skaffold alle definierten Services bereitgestellt werden.

Im Quellcode 9 wird eine exemplarische Skaffold Konfigurationsdatei für eine Java Maven Implementierung gezeigt. Diese nutzt das Build-Werkzeug jib. Dies hat zum einen den Vorteil, dass kein Dockerfile angelegt werden muss und zum anderen muss für die Dateiensynchronisation keine explizite Verzeichnis-Angabe gemacht werden. Jib lässt sich über die Maven pom.xml Datei in das Projekt integrieren. Bei der Integration in Maven wird ein Base-Image angegeben, welches anschließend von jib als Grundlage für das zu erstellende Image genutzt wird. Als *best practise* wird dort ein *distroless* Java-Docker-Image empfohlen. Die *distroless* Images wurden von Google entwickelt und besitzen im Vergleich zu den normalen Images nur die auszuführende Anwendung und ihre Bibliotheken. Ein *distroless* Image bietet demnach keine/n Package-Manager, Shell oder sonstige bekannte Funktion einer Linux Distribution an. Dies hat den Vorteil, dass ausschließlich benötigte Dateien in das Image geladen werden. Die Container werden somit noch leichtgewichtiger und lassen sich noch schneller erstellen (vgl. Google, 2020). Der Nachteil ist, dass keine Shell-Befehle im Container mehr ausgeführt werden können.

```yaml
apiVersion: skaffold/v2beta6
kind: Config
build:
  artifacts:
    - image: example-file-sync
      jib:
        type: maven
        args:
          - --no-transfer-progress
          - -Psync
      sync:
        auto: {}
deploy:
  kubectl:
    manifests:
      - k8s.yaml
```

Quellcode 9: Skaffold - Skaffold.yaml mit jib

Zu Beginn der Datei wird als Erstes die verwendete Version und die Art der skaffold.yaml definiert. Als Nächstes wird bereits der Build-Prozess beschrieben. Hier wird zunächst über *-image* der zu erstellende Image-Name vergeben, sodass das Kubernetes-Manifest dieses referenzieren kann. Danach folgt die jib Konfiguration. Da jib sich für Maven oder Gradle konfigurieren lässt, muss hier das verwendete Werkzeug entsprechend angegeben werden. Außerdem werden die beiden Maven Argumente *--no-transfer-progress* und *-Psync* übergeben. Durch die Verwendung von *--no-transfer-progress* werden die Ausgaben des Fortschritts beim Herunterladen der in der pom.xml angegeben Bibliotheken unterdrückt. Bei größeren pom.xml Dateien kann es dazu führen, dass das Terminal mit Angaben der bereits heruntergeladenen Bytes überflutet wird und ein Ablesen der wichtigen Information dadurch erschwert wird. Das zweite Argument *-Psync* wird für die Verwendung der Dateiensynchronisation benötigt. Durch diese Angabe wird das, in der pom.xml definierte *sync* Profile aktiviert, was wiederum den Entwicklungsmodus von Spring aktiviert. Der Entwicklungsmodus führt dazu, dass die laufende Anwendung bei einer Änderung des Quellcodes automatisch neu gestartet wird. Durch die Verwendung von jib als Bau-Werkzeug wird die Dateiensynchronisation selbständig konfiguriert. Deshalb wird hier nur das Keyword *auto* mit einem leeren Block übergeben. Als letzter Schritt wird für die Bereitstellung ein zuvor

definiertes Kubernetes-Manifest übergeben. Dieses wird in diesem Falle durch *kubectl* umgesetzt.

4.2.5 Integration CI/CD Pipeline

Skaffold lässt sich durch seine modularen Befehle optimal in eine CI/CD Pipeline integrieren. Dies hat den Vorteil, dass alle Funktionen von Skaffold und die dazu angelegten Konfiguration, in der Pipeline genutzt werden können. Über die Konfiguration eines Profils für die CI/CD Pipeline lässt sich mühelos eine produktive Bereitstellungsumgebung mit unterschiedlichen Bereitstellungsschritten definieren. Es wird also von der Entwicklung bis hin zum produktiven Einsatz, das gleiche Werkzeug verwendet. Für eine einfache Bereitstellung in einem Cluster wird der *skaffold run* Befehl empfohlen. Dieser bietet alle nötigen Schritte für die Bereitstellung eines Images im Cluster an. Dabei werden unteranderen auch sogenannte *Healthchecks* ausgeführt, die überprüfen, ob eine bereitgestellte Ressource erfolgreich gestartet wurde. Optional kann ein Timeout gesetzt werden, sodass es nach Ablauf einer definierten Zeit zu einem automatischen Abbruch der Bereitstellung kommt. Dies kann vor allem in einer CD Pipeline nützlich sein, da vor jedem Schritt sichergestellt wird, dass die Ressourcen lauffähig sind (vgl. Skaffold, 2020b). Nach Bedarf kann der Prozess auch noch weiter in die folgenden Befehle unterteilt werden:

skaffold build

Erstellt die Artefakte und lädt diese in das hinterlegte Register. *Skaffold build* kann eine sehr einfache Möglichkeit sein, Build-Prozesse für eine CI-Pipeline einzurichten.

skaffold deploy

Über *skaffold deploy* lassen sich anhand eines Kubernetes-Mainfests bereits gebaute Images in einem Kubernetes Cluster bereitstellen.

skaffold render

Über *skaffold render* lassen sich Momentaufnahmen von einer Anwendung erstellen, ähnlich wie ein Snapshot bei Tilt. Im Gegensatz zu Tilt werden dabei die Images erstellt und mit einem Tag markiert. Die neu erstellten Image-Tags werden anschließend in das Kubernetes Manifest einfügt und als neues Manifest zurückgegeben. So lassen sich einfach Snapshot-Versionen der Entwicklung generieren und Bereitstellen.

4.2.6 Lizenz

Skaffold ist Open-Source und somit frei zugänglich. Eine zusätzliche Enterprise Version gibt es nicht.

4.3 Garden

Garden ist ein Werkzeug des gleichnamigen deutschen Start-Ups und wurde erstmals 2018 vorgestellt. Auch Garden fokussiert sich mit ihrem Werkzeug auf die Optimierung der Entwicklung mit Kubernetes und übernimmt, wie auch die anderen Werkzeuge, das Bauen, Pushen und Bereitstellen von Images. Garden verflogt dabei als einziges der vorgestellten Werkzeuge den projektbasierten Ansatz. Wohingegen sich die anderen Werkzeuge für einzelne Module des Systems konfigurieren lassen, setzt Garden auf eine konsequente Konfiguration über das ganze Projekt. Garden hat sich von Beginn an auf die Testphase in dem Prozess spezialisiert und fokussiert, sodass sich die erstellten Artefakte über Unit- und Integrationstests validieren lassen. Hierbei speichert Garden bei einem In-Cluster-Build alle erstellten Images und ausgeführten Tests in einen eigenen, im Cluster liegenden Cache, welcher dem ganzen Cluster zugänglich gemacht wird. Bei einem In-Cluster-Build werden alle Images über einen von Garden installierten Docker Daemon oder wahlweise über das Bau-Werkzeug *Kaniko* im Cluster gebaut. Dieser Vorteil kommt besonders bei einem shared-Cluster (siehe Kapitel 3.1) zur Geltung, weshalb Garden sich auch auf diese Art von Cluster fokussiert. Garden lässt sich aber für jede Art, auch lokale Cluster, konfigurieren. Bei einem shared-Cluster, mit mehreren Entwicklern (pro Entwickler ein Namespace), können somit erstellte Ressourcen unter den Entwicklern geteilt werden. Ein bereits erstelltes Image,

welches von einem Entwickler in seinen Namespace gebaut wurde, kann somit von einem anderen Entwickler einfach übernommen werden, ohne dieses erneut in seinen eigenen Namespace zu bauen. Für die Entwickler im Cluster kommt es somit zu einer deutlichen Zeitersparnis. Durch eine mögliche Integration von Garden in eine CI-Pipeline, lässt sich der Cache Vorteil auch in der Pipeline nutzen. (vgl. Edvald, 2019a). Garden setzte auf den eigen entwickelten Stack Graphen, welcher die erstellten und bereitgestellten Artefakte mit ihren Abhängigkeiten graphisch darstellt. Dieser wird bei der Bereitstellung automatisch von Garden erstellt. Durch seine Plugin-Architektur lässt sich Garden an die entsprechenden Bedürfnisse eines Projekts einfach anpassen. Im Rahmen dieser Bachelorarbeit wurde die Version 0.12.2 von Garden verwendet.

4.3.1 Bereitstellungsprozess

Garden ist das einzige der vorgestellten Werkzeuge, welches im konfigurierten Cluster eigene Services bereitstellt. Garden wird deshalb nicht nur lokal auf dem Entwicklerrechner ausgeführt, sondern auch im Cluster. Eine zusätzliche Installation oder Konfiguration im Cluster wird aber nicht benötigt. Sobald Garden lokal auf dem Entwicklerrechner ausgeführt wird und es sich mit einem Cluster verbindet, legt es sich eigene Namespaces an, in denen mehrere Services bereitgestellt werden. Unteranderem werden somit die Funktionen des Image Cachings und das Bauen von Images im Cluster ermöglicht. Sobald ein Image von einem Entwickler über einen solchen Service im Cluster gebaut wurde, wird es von den Garden Services in dem lokalen Cache gespeichert. So können alle Entwickler in diesem Cluster auf dasselbe Image zugreifen, ohne es erneut bauen zu müssen. Das Gleiche gilt für die Tests. Tests werden nur erneut ausgeführt, wenn das zu testende Artefakt modifiziert wurde und somit auch neu gebaut werden muss. Durch den bereitgestellten Build-Service und die Kommunikation zwischen Garden mit Kubernetes ist eine lokale Docker- und Kubernetes-Installation nicht nötig (vgl. Garden, 2020c).

Der Bereitstellungsprozess von Garden umfasst die in Abbildung 9 angegebenen Schritte. Garden bietet durch seine PlugIn-Architektur die Möglichkeit die Schritte *Tasks ausführen*,

Artefakte testen und *Pods im Cluster bereitstellen*, zu überspringen. Das Laden von Images in ein Register wird dabei automatisch bei der Verwendung eines lokalen Clusters übersprungen. Zudem bietet Garden die Möglichkeit unterschiedliche Bereitstellungswerkzeuge (bspw. kubectl und Helm) zu verwenden. Die hier hervorgehobenen Schritte unterscheiden sich von den anderen beiden Werkzeugen und werden im Folgenden erläutert.

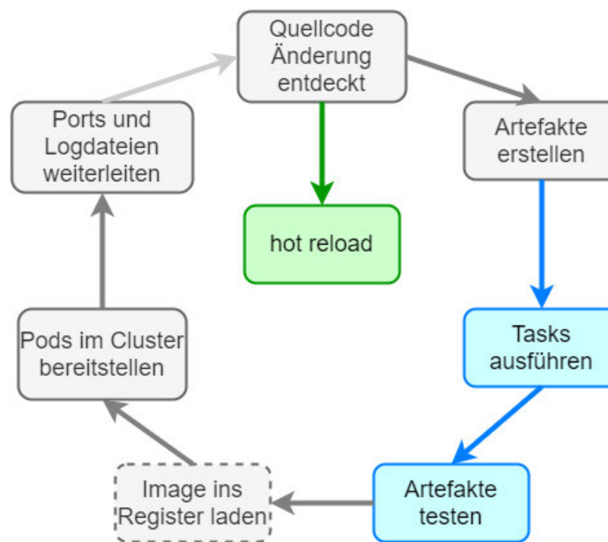

Abbildung 9: Garden - Bereitstellungsprozess

Hot Reload

Auch Garden bietet die Möglichkeit über einen sogenannten *hot reload* Dateien in einen laufenden Pod zu laden und dort zu integrieren. Wie auch bei den anderen Werkzeugen, werden hierzu die Dateien verpackt und an den Pod gesendet, wo diese anschließend wieder entpackt und migriert werden (Garden, 2020a).

Tasks ausführen

Über Tasks lassen sich optional beliebige Dienste definieren. Für einen Task werden Befehle definiert, die anschließend bei der Bereitstellung automatisiert ausgeführt werden. Ebenfalls können bspw. Skripte ausgeführt oder Datenbanken migriert werden. Artefakte können

beliebig viele Tasks besitzen, welche untereinander ebenfalls Abhängigkeiten besitzen können (vgl. Garden, 2020b).

Artefakte testen
In der Testphase lassen sich optional beliebig viele Tests zu einem Artefakt definieren und ausführen. Dabei lassen sich unter anderen im Quellcode definierte Unit-Tests im Container starten. Anhand des Stack Graphens, welcher Abhängigkeiten zwischen den Artefakten darstellt, lassen sich einfach Integrationstest definieren (vgl. Garden, 2020f). Bei einem fehlerhaften Test kommt es zum Abbruch der Bereitstellung des jeweiligen Artefakts.

4.3.2 Stack Graph

Der Bereitstellungsprozess von Garden dreht sich um den Stack Graphen. Mit dem Stack Graphen wird das gesamte System modular und strukturiert als gerichteter Abhängigkeitsgraph beschrieben. Dabei werden die Artefakte, hier *Module,* einzeln mit ihren Abhängigkeiten untereinander beschrieben und in weitere Sub-Prozesse unterteilt. Garden verwendet den Stack Graphen, um bei Änderungen zu erkennen, welche Module neu gebaut, getestet und bereitgestellt werden müssen. So weiß Garden bei einer Änderung genau, welche Sub-Module betroffen sind und welche nicht. Der Graph wird anhand der Konfigurationsdateien erstellt. Dabei wird durch die Erstellung der *project.garden.yaml* Konfigurationsdatei ein Garden-Projekt erstellt und das aktuelle Verzeichnis als Root-Verzeichnis definiert. Beim Ausführen von Garden werden alle Sub-Verzeichnisse auf weitere *garden.yaml* Konfigurationsdateien, welche Module des Graphens beschreiben, durchsucht. Ähnlich wie bei Docker, werden die Konfiguration der einzelnen Module im Quellcode-Verzeichnis definiert. Durch die Angabe weiterer Module in den Konfigurationsdateien werden Abhängigkeiten definiert, welche dann im Graphen angezeigt werden. Die Module sind dabei austauschbar, sodass der Graph individuell zusammengesetzt werden kann. (vgl. Edvald, 2019b)

In Abbildung 10 sieht einen exemplarischen Stack Graphen, der das gesamte System mit seinen Build-, Test- und Bereitstellungsprozessen beschreibt. Der Stack Graph besteht in diesem Beispiel aus den zwei Modulen Frontend und Backend, welche wiederum aus den Schritten Build, Deploy und Test bestehen.

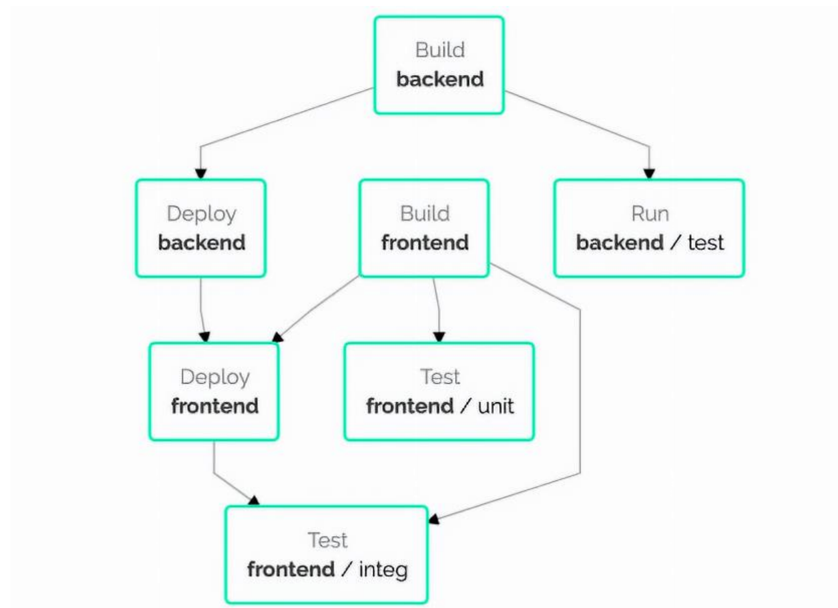

Abbildung 10: Garden - Stack Graph (Garden, 2020h)

Die möglichen Elemente eines Stack Graphens werden im Folgenden erläutert.

Project:

Das Projekt gibt die Rahmenbedingungen des Graphens vor. Es wird initial erstellt und beinhaltet mindestens ein Modul. Über das Projekt werden unter anderem die Provider konfiguriert (vgl. Garden, 2020d).

Provider

Provider werden auf Projektebene konfiguriert. Sie stellen Bereitstellungsumgebungen (Cluster) bereit und kontrollieren, was innerhalb der Knoten des Graphen geschieht, z.B. wie

Module gebaut und bereitgestellt werden. Sie spezifizieren auch die Modultypen (vgl. Garden, 2020d).

Module

Ein Modul ist ein Artefakt, welches gebaut wird. Jedes Modul wird durch einen Typ spezifiziert (bspw. Container oder Helm), welches indiziert wie das Modul gebaut und bereitgestellt wird. Ein Container Modul wird bspw. über ein Dockerfile erstellt. Ein Modul kann dabei keine oder mehrere Services, Tasks und Tests beinhalten. Außerdem kann es Abhängigkeiten zu anderen Modulen haben (vgl. Garden, 2020d).

Service

Ein Service wird im Cluster bereitgestellt. Hinter einem Service versteckt sich ein Kubernetes-Service. Die Angaben eines Garden Service werden intern zu Anweisungen für einen Kubernetes-Service übersetzt. Services können voneinander abhängig sein (bspw. das Frontend von dem Backend). Außerdem können sie Abhängigkeiten zu einem oder mehreren Tasks haben (vgl. Garden, 2020d). Ein zusätzliches Kubernetes Manifest wird nicht zwingend benötigt. Nach Bedarf können aber auch zuvor erstellte Kubernetes Manifeste von Garden für die Bereitstellung genutzt werden.

Task

Bei einem Task handelt es sich um kleine Dienste, die gestartet werden und auf deren Beendigung gewartet wird. Dabei können bspw. Datenbankenintegrationen oder sonstige Skripte ausgeführt werden. Tasks können Abhängigkeiten zu anderen Tasks und Services haben (vgl. Garden, 2020d).

Test

Tests werden zur Validierung der Module ausgeführt. Dabei lassen sich im Quellcode definierte Unit-Tests im Container ausführen. Aufgrund der definierten Abhängigkeiten lassen sich Integrationstests ausführen. Tests können eine Abhängigkeit zu Services oder Tasks

haben (vgl. Garden, 2020d). Ein fehlgeschlagener Test führt zum Abbruch der Bereitstellung des Moduls.

4.3.3 CLI

Auch Garden bietet mit dem Kommandozeilenbefehl *garden dev* einen Entwicklungsmodus an. Dieser Befehl ist das äquivalente Gegenstück zu Tilts *tilt up* und Skaffolds *skaffold dev*. Dabei überwacht Garden alle Quellcode-Verzeichnisse mit einem File-Watcher und stößt die Bereitstellung bei einer Änderung erneut an. Garden bietet die Möglichkeit ein sogenannten *hot reload* (siehe Abbildung 9) durchzuführen, wobei die modifizierten Dateien in den laufenden Pod geladen werden und somit ein neues Bauen und Bereitstellen der Anwendung obsolet wird. Sobald der Befehl ausgeführt wurde, öffnet sich das Dashboard im Browser, welches sowohl den Stack Graphen als auch die Log-Daten der gestarteten Microservices anzeigt (vgl. Garden, 2020e). Optional können die Log-Daten der Microservices durch den Befehl *garden logs -f* im Terminal kontinuierlich ausgegeben werden. Zu dem Build-, Test-, oder Deployment-Prozess gibt es nur im Fehlerfall Ausgaben. Der Bereitstellungsprozess lässt sich, wie bei den anderen Werkzeugen, weiter in einzelne Sub-Befehle unterteilen. Diese nutzen keinen File-Watcher und starten somit bei einer Änderung des Quellcodes keinen neuen Bereitstellungsprozess. Über den Befehl *garden build* lassen sich alle Module des Systems bauen. Durch die Angabe von speziellen Modulnamen können auch nur einzelne Modul gebaut werden. Der Befehl *garden deploy* kann anschließend für die Bereitstellung der Services genutzt werden. Wird kein spezifischer Servicename angegeben, werden alle Services bereitgestellt. Des Weiteren lassen sich über *garden test <Testname | Modulname>* und *garden run task <Modulname | Taskname>* auch alle oder spezifische Taks oder Tests ausführen. Durch das Setzen des Arguments *–watch* kann bei dem Aufruf des *build*, *deploy* und *test* Befehls optional das Dashboard im Browser geöffnet werden (vgl. Garden, 2020e). Diese Befehle eigenen sich besonders für die Integration in eine CI/CD-Pipeline, da so der Bereitstellungsprozess in die typischen Stages einer Pipeline gegliedert werden kann.

4.3.4 Konfiguration

Auch Garden lässt sich, wie die anderen beiden Werkzeuge Tilt und Skaffold, als Binärdatei auf dem Entwicklerrechner installieren. Wie auch Skaffold lässt sich Garden anschließend über YAML-Dateien konfigurieren. Aufgrund des projektbasierten Ansatzes wird zunächst im übergeordneten Verzeichnis, der zu erstellenden Module, eine *project.garden.yaml* Datei angelegt. Die *project.garden.yaml* enthält Information über das bereitzustellende Projekt, darunter bspw. das zu verwendende Cluster. Ein Modul wird anschließend im jeweiligen Quellcodeverzeichnis über eine eigene *garden.yaml* Datei konfiguriert. Im Folgenden wird zur Erläuterung eine beispielhafte Konfiguration für ein Java Maven Projekt vorgestellt. Diese Konfiguration wurde aus Gardens Dokumentation übernommen und entspricht somit der *best-practise* Konfiguration für ein Java Maven Projekt.

Im Quellcode 10 wird die *project.garden.yaml* Datei gezeigt, welche ein Garden Projekt initialisiert. Dabei wird in der Konfigurationsdatei zunächst angegeben, dass es sich um eine Projekt Konfiguration handelt. Anschließend wird ein freiwählbarer Name vergeben, über den sich das Projekt anschließend referenzieren lässt. Über die *environments* lassen sich Bereitstellungsumgebungen definieren. In diesem Fall wird nur ein lokales Cluster konfiguriert (siehe Kapitel 5.2.3 für eine Konfiguration eins remote Clusters). Durch das Erstellen der *project.garden.yaml* wird das aktuelle Verzeichnis als project-root-directory für dieses Projekt konfiguriert, sodass Garden beim Start alle dazugehörigen Unterverzeichnisse auf Garden Module durchsucht.

```yaml
kind: Project
name: spring-boot-hot-reload
environments:
  - name: local
providers:
  - name: local-kubernetes
```

Quellcode 10: Garden – project.garden.yaml (Garden, 2020i)

In Quellcode 11 ist die Konfiguration für ein Modul zu sehen. Auch hier wird ein freiwählbarer Name angegeben, über welchen das Modul im Projekt referenziert wird. Durch die Angabe *type: Container* wird Garden mitgeteilt, dass es sich hier um einen ausführbaren Container handelt. Garden nutz dabei per Default das im selben Verzeichnis hinterlegte Dockerfile, um ein Image für den Container zu erstellen. Als nächstes wird die *hot reload* Konfiguration definiert. Dabei werden die beiden zu synchronisierenden Verzeichnissen, lokal und im Pod, angegeben. In diesem Fall ist die Source der *target* Ordner, in dem sich die verpackten Quellcode Dateien in Form einer Jar befinden. Garden bietet für eine Java Implementierung, im Gegensatz zu den anderen beiden Werkzeugen, momentan keine native Lösung, um einzelne Quellcode-Dateien einer verpackten Jar nachträglich im Pod zu aktualisieren. Es muss bei einem *hot reload* immer lokal eine neue Jar-Datei manuell erstellt werden. Der von Garden automatisierte Bereitstellungsprozess setzt erst nach dem Erstellen einer Jar-Datei ein.

Eine Integration des Bauwerkzeugs jib steht noch aus, sodass zukünftig, wie bei den anderen Werkzeugen, dieser Prozess automatisiert wird. Bei der Verwendung einer anderen Sprache, wie bspw. Python, würde es diesen Nachteil nicht geben, da es dort keine verpackten Quellcodedateien gibt. Garden würde in diesem Fall die modifizierte Datei direkt in den Pod laden.

Als nächstes wird ein Service für die Bereitstellung im Cluster definiert. Auch hier können, wie in einem Kubernetes-Manifest, Ports und sonstige Einstellungen konfiguriert werden. Garden nimmt nach der Erstellung des Pods einen Healtchcheck vor, in dem eine HTTP-Anfrage an den Service gesendet wird. Bei einer Antwort wird der Service als gesund deklariert. Über die Konfiguration eines *Ingresses* wird die HTTP-Schnittstelle veröffentlicht.

```yaml
kind: Module
description: Spring Boot devtools-sample module
type: container
name: devtools
hotReload:
  sync:
    - target: /app/target
      source: target
services:
  - name: devtools
    ports:
      - name: http
        containerPort: 8080
        servicePort: 80
    healthCheck:
      httpGet:
        path: /actuator/health
        port: http
    ingresses:
      - path: /
        port: http
```

Quellcode 11: Garden – garden.yaml für ein Modul (Garden, 2020i)

4.3.5 Integration CI/CD-Pipeline

Auch Garden lässt sich, wie Skaffold, in eine CI-Pipeline integrieren. Eine Integration lohnt sich besonders bei der Verwendung eines shared-remote Clusters. Die CI-Pipeline wird als ein zusätzlicher Entwickler angesehen und profitiert somit von den von Garden angelegten Caches. Alle bereits gebauten Images und ausgeführten Tests werden nicht erneut von der Pipeline gebaut und ausgeführt, sondern werden aus dem Cache geladen. Durch diesen Mechanismus kann die Pipeline deutlich performanter gemacht werden. Des Weiteren wird dadurch die Konfigurationsdatei einer Pipeline deutlich schmaler und einfacher zu erstellen, da nun Garden alle auszuführenden Schritte übernimmt (vgl. Edvald, 2019a). Für die Integration von Garden in eine CI-Pipeline muss lediglich das *Garden CLI* in der CI-Pipeline installiert werden oder das offizielle Garden Image verwendet werden. Bei der Nutzung des Images würde die manuelle Installation in der Pipeline entfallen. Für eine exemplarische Integration von Garden in einer CI-Pipeline siehe Kapitel 5.2.3.

Eine Integration in einer CD-Pipeline wäre ebenfalls möglich, wird jedoch von Garden nicht empfohlen, da in einem produktiven Cluster aus Sicherheitsgründen von In-Cluster-Builds abgesehen werden sollte. Die Cache-Funktionalität wird von Garden aber nur bei einem In-Cluster-Build gewehrleistet (vgl. Garden, 2020g).

4.3.6 Lizenz

Garden bietet eine öffentlich zugängliche Core Version an, welche alle wesentlichen Funktionen des Werkzeuges beinhaltet. Zusätzlich gibt es eine kostenpflichtige Enterprise-Version, die neben den Support der Entwickler einige zusätzliche Funktionen beinhaltet. Unteranderen werden bei der kostenpflichtigen Version die Funktionalitäten Secret Management, Centralized Environment Management und eine direkte Integration eines VCS (GitHub, etc.) angeboten (vgl. Garden, [o. Jahr]b).

5 Durchführung Fallbeispiel

In diesem Kapitel wird die Durchführung des in Kapitel 3.3 vorgestellten Fallbeispiels dargestellt.

5.1 Rahmenbedingungen und Implementierung

Da die im Folgenden vorgestellten Rahmenbedingungen und Implementierungsschritte alle Optimierungswerkzeuge betreffen, werden diese vorgezogen.

5.1.1 Google Cloud

Für die Verprobung des Fallbeispiels wird ein remote Cluster in der Google Cloud, auch GKE (**G**oogle **K**ubernetes **E**ngine) genannt, verwendet. Dieses wurde bereits zuvor über die Web-API der Google Cloud erstellt.

Um das Cluster nutzen zu können, muss dieses in die lokale Installation von dem CLI von Kubernetes *kubectl* integriert werden. Dabei wird das remote Cluster als zusätzlicher Bereitstellungskontext definiert, sodass *kubectl* anschließend zwischen dem lokalen (Minikube) und dem remote (gke_ba-blog_europe-west3-a_ba-blog-cluster) Cluster wechseln kann. Für die Konfiguration der GKE muss Googles SDK (Software-Developer-Kit) lokal installiert sein. Mit der Installation wird unteranderem das CLI *gcloud* installiert, welches für die Kommunikation mit der Google Cloud benötigt wird. Folgende Kommandozeilenbefehle müssen anschließend für die Integration der GKE ausgeführt werden:

- gcloud auth login
- gcloud config set project <*project-id*>
- gcloud config set compute/zone <*compute-zone*>
- gcloud container clusters get-credentials <*cluster-name*>

Dabei wird nach der Authentifizierung das zuvor erstellte Google-Cloud-Projekt über die Angabe der *project-id* referenziert. Anschließend wird die Computing-Zone des Clusters gesetzt. Die Computing-Zone ist der Standort, in der sich das Cluster und deren Ressourcen befinden. Anhand dieser Angaben kann das Cluster im letzten Befehl über den Clusternamen eindeutig referenziert werden. Dieser erstellt den benötigten Bereitstellungskontext in der Konfigurationsdatei von kubectl. Der Kontextname, welcher von den Optimierungswerkzeugen benötigt wird, kann anschließend durch den Befehl *kubectl config get-contexts* abgerufen werden.

Da Skaffold remote-Builds über die Cloud-Build-API der Google Cloud durchführt, muss hierfür ein Dienstkonto und ein dazugehöriger JSON-Zugangstoken in der Google Cloud erstellt werden. Dieses Dienstkonto besitzt die benötigten Berechtigungen, um von außen auf die Cloud-Ressourcen zugreifen zu können. Damit Skaffold nun dieses Dienstkonto verwenden kann, muss der erstellte JSON-Zugangstoken über den Befehl:

- export GOOGLE_APPLICATION_CREDENTIALS= <path/to/json>

als Umgebungsvariable hinterlegt werden. Des Weiteren wird der erstellte Zugangstoken für die Authentifizierung in den zu erstellenden CI/CD-Pipelines verwendet.

5.1.2 Implementierung

Aufgrund desselben Aufbaus werden alle drei zu implementierenden Microservices nach dem gleichen Schema implementiert. Das Vorgehen wird im Folgenden erläutert.

Für die Bereitstellung der drei Datenbanken werden im verwendeten Kubernetes Manifest drei Standalone Instanzen der MongoDB definiert. So werden diese automatisch bei der Ausführung des Manifests bereitgestellt. Über die dort definierten Ports lassen sich die Datenbanken anschließend von den Microservices ansprechen.

In den Microservices werden anschließend nur noch die Schnittstellen zu der jeweiligen Datenbank implementiert. Dabei wird zunächst ein sogenanntes Model erstellt, welches das

zu verwaltende MongoDB-Dokument definiert. Ein Dokument beschreibt somit einen Eintrag in der jeweiligen Datenbank. Anschließend wird über ein Mongo-Repository die Schnittstelle der MongoDB angesprochen. So ist es dem Service möglich CRUD-Operationen auf den Daten auszuführen. Über den Spring Rest-Controller werden anschließend HTTP-Schnittstellen zu den Operationen angeboten, die vom API-Gateway konsumiert werden.

5.2 Optimierungswerkzeuge

5.2.1 Tilt

Wie in Kapitel 3.3 beschrieben, wird Tilt für die Implementierung des User-Microservices genutzt. Dabei wird das Werkzeug Tilt für das gesamte Projekt konfiguriert und anschließend bei der Implementierung verprobt. Der Quellcode und alle anderen verwendeten Konfigurationsdateien können auf der beigelegten CD und dem Verzeichnis *Fallbeispiel* gefunden werden.

Konfiguration

Für die Konfiguration von Tilt wurde die, in Quellcode 7 vorgestellte *best practise* Konfiguration als Vorlage genutzt. Die Dockerfiles der vier Microservices (API-Gateway, User, Comment und Post) wurden nach dem in Quellcode 6 dargestellten Schema angepasst. Da es sich beim Frontend um eine Angular Anwendung handelt, wurde hierfür eine Änderung des Konfigurationsschema, wie in Quellcode 12 zu sehen, vorgenommen. Die Angular Anwendung läuft in dem bereitgestellten Pod im Entwicklungsmodus von Angular (*ng serve*), sodass bei einer Änderung des Quellcodes die Anwendung automatisch im Pod neu kompiliert wird. Für die Dateiensynchronisation über *live updates* werden die beiden Quellcodeverzeichnisse lokal (*src*) und im Pod (*/app*) direkt synchronisiert. Um bei einer Änderung der benötigten Bibliotheken nun nicht ein neues Bauen des Images zu forcieren, wird bei einer Änderung der *package.json* Datei, welche alle benötigten Bibliotheken definiert, der Befehl *npm install* ausgeführt. Durch diesen Befehl werden alle fehlenden Bibliotheken, die in der *package.json* Datei aufgelistet sind, nachgeladen.

```
[…]
# frontend (angular)
docker_build(' eu.gcr.io/ba-blog/frontend', 'frontend/',
    live_update=[
      sync('frontend/src', '/app'),
      run('npm install', trigger=['frontend/package.json'])
])
[…]
k8s_yaml('k8s.yaml')
k8s_resource('mongo-user')
k8s_resource('mongo-comment)
k8s_resource('mongo-post)
k8s_resource('api-gateway', port_forwards=8080, resource_deps=['api-
gateway-compile'])
k8s_resource('user', resource_deps=['user-compile','mongo-user'])
k8s_resource('post', resource_deps=['post-compile','mongo-post])
k8s_resource('comment', resource_deps=['comment-compile','mongo-comment]
k8s_resource('frontend', port_forwards=4200)
```

Quellcode 12: Tilt - Fallbeispiel: Ausschnitt - Tiltfile

Die im Kubernetes-Manifest definierten Datenbank-Ressourcen werden im Tiltfile über ihre Namen (mongo-user, mongo-post und mongo-comment) referenziert, um Abhängigkeiten abzubilden. Für die Bereitstellung der Microservices User, Post und Comment müssen zuerst die referenzierten Datenbanken in *resource_deps* bereitgestellt worden sein.

Da Tilt sich für eine Verwendung eines lokalen Clusters ausspricht, wurde der User-Microservice ausschließlich im lokalen Minikube Cluster implementiert.

Für die Vergleichbarkeit von Tilt mit den anderen Werkzeugen wurde es im Nachgang auch für ein remote Cluster in der Google Cloud konfiguriert. Hierfür wurde lediglich der Funktionsaufruf *allow_k8s_contexts(<contextname>)* dem Tiltfile hinzugefügt. Der *contextname* stellt dabei den Namen eines Eintrags in der Konfigurationsdatei von dem CLI *kubectl* da. In dieser Datei werden die Zugangsdaten der konfigurierten Cluster gespeichert (siehe Kapitel 5.1.1 für die Integration des remote Clusters in *kubectl*).

Durchführung

Die Implementierung des User-Microservices in dem lokalen Cluster ließ sich nach der initialen Bereitstellung, die etwas Zeit benötigt, kaum von einer reinen lokalen Entwicklung unterscheiden. Durch die Dateiensynchronisation mittels der *live updates* ist die Verzögerung so gering, dass diese vernachlässigt werden kann. Außerdem erleichtert die graphische Benutzerschnittstelle, durch ihren guten Überblick über alle laufenden Prozesse, die Entwicklung im Cluster. Fehler lassen sich, anhand der farbigen Markierung und dem zusammentragen der Log-Daten, direkt identifizieren. In Abbildung 11ist die graphische Terminal-Benutzerschnittstelle zu dem Fallbeispiel zusehen. Wie auch bei der Benutzerschnittstelle im Browser, ist es möglich sich durch die Artefakte zu navigieren und nur Log-Daten zu einem bestimmten Artefakt zu erhalten. Unter dem Punkt *uncategorized* werden die *persistentvolumeclaims* der drei Datenbanken zusammengefasst. Diese wurden im Kubernetes-Manifest definiert und dienen zur persistenten Speicherung von Daten. Im Gegensatz zu Pods, die ihre Daten bei einem Neustart oder bei dem Löschen des Pods verlieren, werden die Daten der Datenbanken durch *persistentvolumeclaims* Pod unabhängig im Cluster gespeichert.

```
tilt — tilt up — 104×28
RESOURCE NAME                                          CONTAINER •  UPDATE STATUS      •   AS OF
▶ ● (Tiltfile) ..........................................  N/A                •   1m ago
  ● api-gateway ..................................  Running • OK       (8.5s) • <30s ago
  ○ frontend ....................................  Pending • In prog.  (34s) •      -
  ● user ........................................  Running • OK       (6.1s) • <30s ago
  ● post ........................................  Running • OK       (7.4s) • <30s ago
  ● comment .....................................  Running • OK       (5.5s) • <30s ago
  ● mongo-user ..................................  Running • OK       (0.6s) • <45s ago
  ● mongo-post ..................................  Running • OK       (1.0s) • <45s ago
  ● mongo-comment ...............................  Running • OK       (1.2s) • <45s ago
  ● api-gateway-compile ...................................  OK        (12s) •   1m ago
  ● user-compile ..........................................  OK        (12s) •  <1m ago
  ● comment-compile .......................................  OK       (9.2s) • <45s ago
  ● post-compile ..........................................  OK       (8.8s) • <45s ago
  ● uncategorized .........................................  OK       (0.5s) • <45s ago

 1: ALL LOGS | 2: build log | 3: runtime log |                                      X: expand
         : No active profile set, falling back to default profiles: default
         post | 2020-09-02 12:13:07.237  INFO 11 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data MongoDB repositories in DEFAULT
mode.
         post | 2020-09-02 12:13:07.947  INFO 11 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 681ms. Found 1
MongoDB repository interfaces.
 ✓ OK                                          To explore, open web view (enter) • terminal is limited
                                               Browse (↓ ↑), Expand (→) ¦ (enter) log ¦ (ctrl-C) quit
```

Abbildung 11: Tilt - Graphische Benutzerschnittstelle (Terminal)

CI/CD-Pipeline

Da Tilt momentan keine native Unterstützung für die Integration in eine CI/CD-Pipeline anbietet, entfällt die Integration.

5.2.2 Skaffold

Wie in Kapitel 3.3 beschrieben, wird Skaffold für die Implementierung des Comment-Microservices genutzt. Dabei wird das Werkzeug Skaffold für das gesamte Projekt konfiguriert und anschließend bei der Implementierung des Comment-Microservices verprobt. Der Quellcode und alle anderen verwendeten Konfigurationsdateien können auf der beigelegten CD und dem Verzeichnis *Fallbeispiel* gefunden werden.

Konfiguration

Für die Konfiguration der Backend-Microservices wurde die vorgestellte jib Konfiguration aus Quellcode 9 als Vorlage genommen und entsprechend angepasst.

Da es sich beim Frontend um eine Angular Anwendung handelt, kann hierfür nicht das Java Build-Werkzeug jib verwendet werden. Stattdessen wird für das Frontend auf das Dockerfile der Angular Anwendung zurückgegriffen. Da Skaffold per Default auf das im Verzeichnis liegende Dockerfile zurückgreift, müssen, wie in Quellcode 13 zusehen, keine weiteren Angaben getätigt werden. Die zu synchronisierenden Verzeichnisse für das *file sync* werden dementsprechend manuell gesetzt, sodass die modifizierten Quellcode-Dateien im *src* Verzeichnis direkt in den Pod geladen und dort von Angular selbst kompiliert werden.

```
[…]
- image: eu.gcr.io/ba-blog/frontend
  context: frontend/
  sync:
    manual:
      - src: 'src/**'
        dest: '/app'
deploy:
  kubeContext: minikube
  kubectl:
    manifests:
      - k8s.yaml
profiles:
  - name: remote
    build:
      googleCloudBuild:
        projectId: ba-blog
      tagPolicy:
        sha256: {}
    deploy:
      kubeContext: gke_ba-blog_europe-west3-a_ba-blog-cluster
```

Quellcode 13: Skaffold - Ausschnitt: skaffold.yaml Fallbeispiel

Des Weiteren wurde das lokale Cluster Minikube als Default Kontext konfiguriert, sodass beim Start von Skaffold, ohne Angabe eines Profils, die Anwendung in dem lokalen Minikube Cluster bereitgestellt wird. Für die Bereitstellung der Anwendung in dem remote Cluster wurde das Profil *remote* konfiguriert. Über dieses Profil werden die Images durch die Cloud-Build-API in der Google Cloud gebaut und anschließend in dem remote Cluster bereitgestellt. Hierfür wird sowohl die *project-id* des Google Cloud Projekts als auch die Tag-Police bestimmt. Die Tag-Police *sha256* taggt dabei jeden Imagenamen mit dem Versionstag *latest* und einem Hashwert, sodass die von Skaffold in das Google Cloud Register geladenen Images stets als neuste Version referenziert werden. Durch die Angabe der *project-id* kann Skaffold die Cloud-Build-API des konkreten Google-Cloud-Projekts für ein remote Build ansprechen.

Durchführung

Für die Implementierung des Comment-Microservices wurden die in Kapitel 5.1.2 aufgeführten Schritte durchgeführt.

Die Implementierung des Comment-Microservices im lokalen Cluster hat sich im Wesentlichen nur durch die graphischen Benutzeroberfläche zu der von Tilt unterschieden. Skaffold verliert durch die kontinuierliche Ausgabe der Log-Daten aller Artefakte, wie in Abbildung 12 zu sehen, deutlich an Übersichtlichkeit. Die Ausgaben lassen sich auch nicht, wie bei Tilt, anhand der Artefakte filtern, sodass man sich im Fehlerfall durch alle Log-Daten wühlen muss.

```
tilt — -zsh — 114×32
[apiapp]  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
[apiapp] ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
[apiapp]  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
[apiapp]   '  |____| .__|_| |_|_| |_\__, | / / / /
[apiapp]  =========|_|==============|___/=/_/_/_/
[apiapp]  :: Spring Boot ::        (v2.3.0.RELEASE)
[apiapp]
[commentapp] 2020-09-04 07:46:32.967  INFO 1 --- [  restartedMain] phil.adam.comment.CommentApplication     : Star
ting CommentApplication on comment-77d56c954c-zf9rs with PID 1 (/app/classes started by root in /)
[commentapp] 2020-09-04 07:46:33.036  INFO 1 --- [  restartedMain] phil.adam.comment.CommentApplication     : No a
ctive profile set, falling back to default profiles: default
[postapp]
[postapp]   .   ____          _            __ _ _
[postapp]  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
[postapp] ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
[postapp]  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
[postapp]   '  |____| .__|_| |_|_| |_\__, | / / / /
[postapp]  =========|_|==============|___/=/_/_/_/
[postapp]  :: Spring Boot ::        (v2.3.1.RELEASE)
[postapp]
[userapp] 2020-09-04 07:46:33.452  INFO 1 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtool
s property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
[userapp] 2020-09-04 07:46:33.460  INFO 1 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For add
itional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
[commentapp] 2020-09-04 07:46:33.791  INFO 1 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devt
ools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
[commentapp] 2020-09-04 07:46:33.799  INFO 1 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For
additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
[apiapp] 2020-09-04 07:46:34.037  INFO 1 --- [  restartedMain] phi.adam.api.ApiGateWayApplication       : Starting
 ApiGateWayApplication on api-gateway-64ddd74c84-c5xz2 with PID 1 (/app/classes started by root in /)
[apiapp] 2020-09-04 07:46:34.054  INFO 1 --- [  restartedMain] phi.adam.api.ApiGateWayApplication       : No activ
e profile set, falling back to default profiles: default
```

Abbildung 12: Skaffold - Benutzerschnittstelle (Terminal)

Die Performanz unterschied sich dabei sowohl beim initialen Bereitstellen als auch bei der Dateiensynchronisation nicht zu der von Tilt. Sobald die Anwendung initial bereitgestellt wurde, wurden spätere Änderung am Quellcode durch die Dateiensynchronisation *file sync*

innerhalb weniger Sekunden im laufenden Pod wirksam. Bei der Verwendung eines remote Clusters müssen die Images zusätzlich in ein Register geladen werden und das Bauen und die Kommunikation findet nicht mehr nur lokal statt, dies verzögert den initialen Bereitstellungsprozess um einige Sekunden bzw. Minuten. Sobald die Anwendung jedoch bereitgestellt wurde, gibt es bei der Dateiensynchronisation keinen spürbaren Unterschied zu einem lokalen Cluster.

CI/CD-Pipeline

Skaffold wurde im Rahmen dieser Arbeit, für das Fallbeispiel in eine GitLab CD-Pipeline integriert. Wie in Quellcode 14 zu sehen ist, wurde hierfür eine *.gitlab-ci.yaml* Datei angelegt, welche bei einem *commit* automatisch vom GitLab-Runner ausgeführt wird. Da der *skaffold run* Befehl alle Schritte des Bereitstellungsprozesses beinhaltet, besteht die Pipeline nur aus der *development* Stage. Als Basis Image der Pipeline wird das offizielle Skaffold-Image genutzt, welches sowohl Skaffold selbst als auch Googles SDK beinhaltet. Um Skaffold in Kombination mit dem remote Cluster in der Google Cloud nutzen zu können, muss dieses zunächst in dem verwendeten Skaffold Container konfiguriert werden. Zur Authentifizierung wurde der erstellte JSON-Zugangstoken (siehe Kapitel 5.1.1) als Variable (*$GCP_SERVICE_KEY*) in GitLab hinterlegt. Des Weiteren wird der Token ebenfalls als Umgebungsvariable im Container gespeichert. Auf die Umgebungsvariable greift Skaffold im Build-Prozess zurück, um sich bei der remote Cloud-Build-API zu authentifizieren und anschließend einen remote Build durchzuführen. Danach werden die beiden Konfigurationsvariablen *project* und *compute/zone* gesetzt, sodass anschließend über den Befehl *get-credentials* der Bereitstellungskontext der Google Cloud für Skaffold bereitgestellt wird.

```
services:
  - docker:dind

stages:
  - development

development:
  image:
    name: gcr.io/k8s-skaffold/skaffold:latest
  stage: development
  script:
    - echo "$GCP_SERVICE_KEY" > gcloud-serviceKey.json
    - export GOOGLE_APPLICATION_CREDENTIALS="gcloud-serviceKey.json"
    - gcloud auth activate-service-account --key-file gcloud-servicKey.json
    - gcloud config set project ba-blog
    - gcloud config set compute/zone europe-west3-a
    - gcloud container clusters get-credentials ba-blog-cluster
    - skaffold run -p remote
```

Quellcode 14: Skaffold - Integration in .gitlab-ci.yaml

5.2.3 Garden

Wie in Kapitel 3.3 beschrieben, wird Garden für die Implementierung des Post-Microservices genutzt. Dabei wird das Werkzeug Garden für das gesamte Projekt konfiguriert und anschließend bei der Implementierung des Post-Microservices verprobt. Der Quellcode und alle anderen verwendeten Konfigurationsdateien können auf der beigelegten CD und dem Verzeichnis *Fallbeispiel* gefunden werden.

Konfiguration

Für das Fallbeispiel wurde zunächst über die *project.garden.yaml* Datei ein Garden Projekt erstellt. Jedoch wurde hier im Gegensatz, zu der in Quellcode 10 vorgestellten Version, die Konfiguration um die Bereitstellungsumgebung *remote* erweitert. Hinter dieser Bereitstellungsumgebung verbirgt sich das remote Cluster in der Google Cloud. Wie bei Skaffold, wird in der Konfigurationsdatei von Garden anschließend nur noch der Kontextname des remote Clusters angegeben. Über die Angabe *buildMode: cluster-docker* werden die

erstellten Images im remote Cluster über die im Cluster bereitgestellten Build-Services gebaut. Um die Services von Garden nutzen zu können, müssen diese zuvor initial über den Befehl:

- garden plugins kubernetes cluster-init –env=remote

im Cluster bereitgestellt werden.

```
[...]
providers:
  - name: kubernetes
    environments: [remote]
    context: gke_ba-blog_europe-west3-a_ba-blog-cluster
    defaultHostname: ba-blog-blog.com
    buildMode: cluster-docker
    setupIngressController: nginx
[...]
```

Quellcode 15: Garden - Fallbeispiel: Ausschnitt project.garden.yaml

Für die Konfiguration der Backend-Microservices und des Frontends wurde die in Quellcode 11 vorgestellte *beste-practise* Konfiguration als Vorlage genommen und entsprechend angepasst. Das Frontend-Modul, wie in Quellcode 16 zu sehen, wurde um einen Integrationstest erweitert. Der Test wird durch Angulars CLI Befehl *ng test* ausgeführt. Anhand einer HTTP-Anfrage an das Gateway wird die Verfügbarkeit des Backends geprüft. Falls das Gateway nicht antwortet, schlägt der Test fehl. Dabei wird durch die Angabe des Parameters --*watch=false* das Öffnen der Test-Log-Dateien im Browser unterdrückt. Dies ist notwendig, da in Containern keine Browser geöffnet werden können und es sonst zu Fehlermeldungen kommen würde.

```
[...]
tests:
  - name: integ
    command: [ng, test]
    args: [--watch=false]
    dependencies:
      - frontend-service
```

Quellcode 16: Garden - Fallbeispiel: Integrationstest

Durchführung

Für die Implementierung des Post-Microservices wurden die in Kapitel 5.1.2 aufgeführten Schritte durchgeführt.

Da Garden bei seinem *hot reload* keine native Lösung für das automatisierte Bereitstellen von Java Code anbietet, muss vor jedem Update der Pods als erstes eine neue Jar-Datei lokal erstellt werden. Dies ist zwar kein großer Aufwand, jedoch verliert Garden somit etwas an Leichtigkeit. Nach der initialen Bereitstellung erfolgt über einen *hot reload* ein sekundenschnelles Aktualisieren der laufenden Pods. Der Stack Graph von Garden bietet, wie in Abbildung 13 zu sehen, einen modularen und strukturierten Überblick über das bereitgestellte System. Durch den modularen Aufbau lassen sich Module, Tests und Tasks leicht hinzufügen und ggf. wieder entfernen. Über den Befehl *garden dev* werden Informationen über den aktuellen Bereitstellungsschritt (Build, Test oder Deploy) ausgegeben, allerdings ist es dem Nutzer nicht ersichtlich, welcher konkrete Befehl aktuell ausgeführt wird. Kommt es beispielsweise bei dem Bau eines Images zu einem Timeout, wird dem Entwickler nicht dargelegt welche Schritte schon ausgeführt wurden und welche nicht. Eine gezielte Fehlersuche wird somit erschwert. Ein Vorteil ist jedoch das Image-Caching, welches im Fallbeispiel aufgrund fehlender weiterer Entwickler nur von der CI-Pipeline genutzt werden konnte.

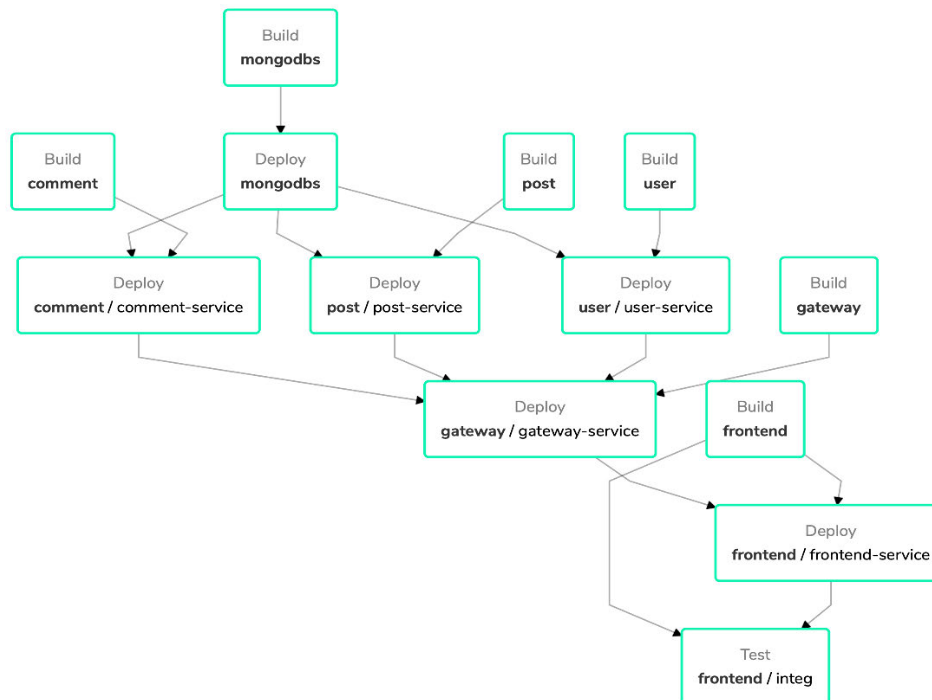

Abbildung 13: Garden - Fallbeispiel: Stack Graph

CI-Pipeline

Für die Integration von Garden in eine CI-Pipeline wurde, wie bei Skaffold, eine GitLab Pipeline über eine *.gitlab-ci.yaml* Datei erstellt. Dabei wird das offizielle Garden Image als Base-Image verwendet. Auch Gardens Image beinhaltet bereits Googles SDK, sodass das remote Cluster, wie bereits in Kapitel 5.1.1 beschrieben, integriert werden kann. Anschließend werden in den Stages die jeweiligen Garden Befehle mit dem *–env=remote* Parameter ausgeführt, sodass die CI-Pipeline das remote Entwicklungscluster verwendet. Hierdurch kann die Pipeline dieselben Caches, wie die Entwickler nutzten. Damit wird der Durchlauf der Pipeline deutlich beschleunigt.

```yaml
services:
  - docker:dind

image:
  name: gardendev/garden-gcloud:latest

stages:
  - test
  - deploy

before_script:
  - echo "$GCP_SERVICE_KEY" > gcloud-service-key.json
  - export GOOGLE_APPLICATION_CREDENTIALS="gcloud-service-key.json"
  - gcloud auth activate-service-account --key-file gcloud-service-key.json
  - gcloud config set project ba-blog
  - gcloud config set compute/zone europe-west3-a
  - gcloud container clusters get-credentials ba-blog-cluster

test:
  stage: test
  script:
    - garden test --env=remote

build:
  stage: deploy
  script:
    - garden deploy --env=remote
```

Quellcode 17: Garden - Fallbeispiel: .gitlab-ci.yaml

6 Evaluation

Im Folgenden werden die drei Werkzeuge anhand der in Kapitel 3.2.2 definierten Kriterien bewertet und miteinander verglichen. Im Kapitel 6.7 wird anschließend eine abschließende Auswertung durchgeführt und die herausgearbeiteten Anwendungsfälle für das jeweilige Werkzeug vorgestellt.

6.1 Konfiguration

Alle drei vorgestellten Werkzeuge lassen sich, nach einer lokalen Installation, auf dem Entwicklungsrechner über Konfigurationsdateien für ein Projekt konfigurieren. Jedes Werkzeug geht dabei mit unterschiedlichen Vorgehensweisen vor. Da im Rahmen dieser Bachelorarbeit Java als zentrale Programmiersprache verwendet wurde, bezieht sich die folgende Bewertung ausschließlich auf die Konfiguration für ein Java-Projekt. Bei der Verwendung andere Programmiersprachen könnte die Bewertung anders ausfallen.

Tilt setzt bei seiner Konfiguration auf eine Starlark-Datei, welche als eigenes Programm fungiert und somit den Vorteil einer dynamischen Konfiguration ermöglicht. Jedoch müssen bei der empfohlenen Konfiguration Änderungen an dem Dockerfile vorgenommen werden, sodass hier zwar ein geringer, aber dennoch ein zusätzlicher Aufwand entsteht. Durch die Angabe der auszuführenden Kommandozeilenbefehle in der Konfigurationsdatei gewinnt Tilt zwar an Transparenz, verliert im gleichen Atemzug aber auch etwas an Übersichtlichkeit.

Skaffold hingegen verwendet YAML-Dateien für ihre Konfiguration. Da Kubernetes-Manifeste ebenfalls mit YAML-Dateien konfiguriert werden, findet sich ein Entwickler aus dem Kubernetes-Bereich, mit dieser Art der Konfiguration schnell zurecht. Im Vergleich zu der *best practise* Konfiguration von Tilt, fällt die empfohlene Konfiguration von Skaffold

deutlich schlanker und übersichtlicher aus. Skaffold bietet dabei deutlich mehr Konfigurationsmöglichkeiten als seine Mitstreiter. Es lässt sich hier zwischen einer großen Auswahl and Build-, Deploy- und sonstigen Werkzeugen wählen.

Auch Garden verwendet YAML-Dateien für ihre Konfigurationen. Da Garden als einziges Werkzeug einen Ansatz auf Projekt-Ebene verfolgt, fällt hier die Konfiguration etwas umfangreicher aus. Für Garden muss neben der Projekt-Konfigurationsdatei, für jeden Microservice eine eigene Modul-Konfigurationsdatei angelegt werden. Dabei lassen sich in der Modul-Konfiguration auch Services definieren, die von Garden bei der Ausführung zu Kubernetes-Manifesten übersetzt werden. Dies ist eine sehr angenehme und übersichtliche Lösung, da auf das manuelle Erstellen von Kubernetes-Manifesten verzichtet werden kann. Alle drei Werkzeuge lassen sich aber auch mit zuvor definierten Manifesten, Build-Skripten oder Dockerfiles konfigurieren.

Alle drei vorgestellten Optimierungswerkzeuge sind erst seit ca. 2 Jahren verfügbar. Tilt und Garden unterstützen zurzeit nur die populärsten Build- und Deploy Werkzeuge. Garden bietet bspw. keine eigene Unterstützung für das Build-Werkzeug jib an. Skaffold hingegen profitiert von seinem Entwickler Google, der eine Vielzahl an unterschiedlichen Werkzeugen für die Integration anbietet. Aufgrund der Möglichkeit, dass Kommandozeilenbefehle bzw. Custom-Skripts ausgeführt werden können, ist es allen drei Optimierungswerkzeugen möglich auch nicht nativ angebotene Werkzeuge zu nutzen.

Jedes der drei Werkzeuge lässt sich anhand der jeweiligen Dokumentation einfach in ein Projekt integrieren. Dabei wirkt die Konfiguration von Skaffold am intuitivsten. Tilt und Garden liegen, durch ihre etwas komplexeren Konfiguration leicht hinter Skaffold.

Es ergibt sich folgendes Ranking:

1. Skaffold
2. Garden und Tilt

6.2 Performanz

Jedes der drei vorgestellten Optimierungswerkzeuge bietet nach einem initialen Bereitstellen eine eigene Dateiensynchronisation an. Alle drei Werkzeuge erstellen eine temporäre verpackte Datei mit den modifizierten Quellcode-Dateien und laden diese in den laufenden Pod, wo diese dann entpackt und migriert werden. Aufgrund desselben Vorgehens der drei Werkzeuge gibt es weder bei der initialen Bereitstellung noch bei der Dateiensynchronisation signifikante Performanz Unterschiede. Sie beschleunigen mit ihren automatisierten Bereitstellungsverfahren den Entwicklungsprozess deutlich. Garden bietet durch seinen Caching Mechanismus als einziges Werkzeug die Möglichkeit, erstellte Ressourcen im Cluster untereinander zu teilen, sodass der Bereitstellungsprozess in einem shared-Cluster weiter beschleunigt werden kann.

Es ergibt sich folgendes Ranking:

1. Garden
2. Tilt und Skaffold

6.3 Transparenz

Bei der Transparenz der Werkzeuge gibt es signifikante Unterschiede. Tilt bietet durch seine graphische Benutzerschnittstelle, die alle ausgeführten Befehle dokumentiert, die größte Transparenz. Aufgrund der aufgelisteten Kommandozeilenbefehle im Tiltfile kann der Entwickler den Bereitstellungsprozess weiter nach seinen Ansprüchen konfigurieren. Skaffold bietet hingegen zwar Informationen zu den im Hintergrund ablaufenden Prozessen, jedoch lassen sich diese, aufgrund des kontinuierlichen Outputs der Log-Daten aller Artefakte, nicht gut ablesen. Garden bietet die schwächste Transparenz, da nur im Fehlerfall Teile, der im Hintergrund ausgeführte Bereitstellungsschritte, angezeigt werden. Im Erfolgsfall werden während der Bereitstellung nur die Bereitstellungsschritte (Build, Test und Deploy), in der sich das jeweilige Modul befindet, ausgegeben. Um auch hier einen Überblick zu behalten und vor allem um evtl. logische Fehler in der Konfiguration zu finden, wäre eine detaillierte Ausgabe über die Bereitstellungsschritte in jedem Fall wünschenswert. Sobald die Module

bereitgestellt wurden, lassen sich die Log-Daten der Anwendung, wie bei Skaffold, als kontinuierliche Ausgabe in der Browserschnittstelle oder auf dem Terminal ausgeben.

Es ergibt sich folgendes Ranking:

1. Tilt
2. Skaffold
3. Garden

6.4 Lizenz und Kosten

Jedes der drei Optimierungswerkzeuge bietet zumindest eine frei zugängliche Open-Source Variante an. Garden und Tilt bieten dabei zusätzlich eine kostenpflichtige Enterprise Version an. Tilt bietet bei ihrer kostenpflichtigen Version hauptsächlich die Unterstützung ihrer Entwickler in Kombination mit präferierten Feature-Requests an. Dabei sind alle Funktion sowohl in der Open-Source als auch in der Enterprise Version enthalten. Garden hingegen bietet bei ihrer Enterprise Version zusätzliche Features an, welche in der frei zugänglichen Version nicht mit inbegriffen sind. So werden unteranderem Features wie Secret Management, Centralized Environment Management und eine direkte Integration von einem VCS (GitHub, etc.) nur in der Enterprise Version angeboten. Skaffold hat im Gegensatz zu den anderen beiden Werkzeugen den Vorteil, dass es von Google entwickelt wurde und Google alle von ihnen entwickelten Werkzeuge im vollen Umfang frei zur Verfügung stellt.

Es ergibt sich folgendes Ranking:

1. Skaffold
2. Tilt
3. Garden

6.5 Dokumentation

Alle drei Unternehmen bieten auf ihrer Webseite eine detaillierte Dokumentation ihres Werkzeuges an. Anhand der vielen Konfigurationsbeispiele lassen sich die Werkzeuge leicht

im einen eigenen Projekt Integrieren. Die Dokumentation von Tilt bietet jedoch als einzige keine zeitliche Referenz zu ihren Einträgen an, sodass als Nutzer nicht erkennbar ist, ob ein Eintrag bereits veraltet ist.

Es ergibt sich folgendes Ranking:

1. Skaffold und Garden
2. Tilt

6.6 Langlebigkeit

Da alle drei der vorgestellten Werkzeuge erst vor ca. 2 Jahren veröffentlicht wurden und seitdem stetig weiterentwickelt wurden, lässt sich aktuell keine valide Beurteilung zur Langlebigkeit vornehmen. Da jedoch Garden und Tilt von kleineren Start-Ups ins Leben gerufen wurden, besteht hier eine höhere Wahrscheinlichkeit, dass diese Projekte in Zukunft nicht weiter betreut werden. Auch hier profitiert Skaffold eindeutig davon, dass es von dem Tech-Giganten Google entwickelt wurde. Die Entwicklung von Skaffold wird mit hoher Wahrscheinlichkeit auch noch in Zukunft weiter vorangetrieben werden. Da alle Werkzeuge jedoch Open-Source sind, besteht die Möglichkeit die Werkzeuge zukünftig auch selbst auf die eigenen Bedürfnisse anzupassen.

Es ergibt sich folgendes Ranking:

1. Skaffold
2. Tilt und Garden

6.7 Auswertung

Aus den Bewertungskriterien ergibt sich folgendes finales Ranking

Werkzeug	Anzahl 1. Plätze	Anzahl 2. Plätze	Anzahl 3. Plätze	Ø - Platzierung
Skaffold	4	2	0	1,3
Tilt	1	5	0	1,8
Garden	2	2	2	2,0

Skaffold liegt mit der besten durchschnittlichen Platzierung vor Tilt und Garden an der Spitze. Dabei profitiert Skaffold vor allem von seinem Entwickler Google. Neben dem Google typischen lizenzfreien Werkzeug bietet Skaffold eine große Auswahl an Google eigenen Bauwerkzeugen an, sodass bspw. Bazel und jib nativ unterstütz werden. Die anderen Werkzeuge wurden von kleineren Start-Ups erstellt, welche einerseits nicht die Entwicklerteam-Größe und andererseits nicht die finanziellen Rücklagen wie Google besitzen. Skaffold ist definitiv der Allrounder unter den Werkzeugen und lässt sich aufgrund seiner hohen Flexibilität in jedes Softwareprojekt und Cluster optimal integrieren. Unter bestimmten Umständen glänzen jedoch die anderen beiden Werkzeuge. Wie so oft kommt es bei der Wahl des richtigen Technologie-Stacks immer auf den Use-Case und die Rahmenbedingen an. Dabei spielt bereits die Wahl des verwendeten Entwicklungsclusters und die Größe des Entwicklerteams eine große Rolle. Gerade für kleine Projekte wirkt Skaffold, mit seiner großen Auswahl an Konfigurationsmöglichkeiten und seiner Plugin-Architektur, erschlagend. Tilt hingegen eignet sich besonders gut als Werkzeug für kleinere Projekte, die vor allem auf lokale Kubernetes Cluster in der Entwicklung setzen. Im Vergleich zu Skaffold ist Tilt ein deutlich leichtgewichtigeres Werkzeug. Tilt bietet dabei zwar nicht die Möglichkeit einer Integration in eine CI/CD-Pipeline, ermöglicht jedoch durch seine graphische Benutzerschnittstelle den Entwicklern den besten Überblick über die im Hintergrund ablaufenden Prozesse. Besonders im Fehlerfall profitieren die Entwickler von isoliert angezeigten Log Daten. Für große Projekte, die mehr Flexibilität und einen größeren Pool an mögliche Konfiguration benötigen, bietet sich Tilt allerdings nicht an. Auch wenn Garden anhand der Bewertungskriterien am schlechtesten abgeschnitten hat, bietet Garden mit seiner ganz eigenen Herangehensweise ein durchaus potentes Werkzeug. Dabei bietet auch Garden durch seine Plugin-Architektur eine große Auswahl an Konfigurationsmöglichkeiten. Durch

das Image-Caching und das damit verbundene Teilen von bereits erstellten Ressourcen, bietet sich Garden besonders für shared-Clusters mit mehreren Entwicklern an. Garden eignet sich außerdem besonders für eine Integration in eine CI-Pipeline, da auch die Pipeline von dem Cache profitieren kann und somit die Performanz optimiert wird. Über den einzigartigen Stack Graphen lassen sich im Team besonders gut Abhängigkeiten im System aufzeigen, was bei komplexeren Projekten sehr hilfreich ist.

Alle drei der vorgestellten Werkzeuge bieten für die kontinuierliche Beistellung, bei der Integration von Kubernetes in den Softwareentwicklungsprozess, Abhilfe an. Die Einbindung von Optimierungswerkzeugen lohnt sich besonders bei Microservice-Architekturen, da hier die einzelnen Services separat bereitgestellt werden müssen. Der Aufwand einer zusätzlichen Konfiguration eines Optimierungswerkzeuges lohnt sich aufgrund der dadurch erzielten sekundenschnellen automatisierten Bereitstellung und der damit verbundenen Verkürzung der Feedbackschleife. Die direkte Interaktion mit Docker und Kubernetes über das Terminal wird aus dem Entwicklungsprozess verbannt. Einige der Werkzeuge gehen sogar soweit, dass eine lokale Installation von Docker und Kubernetes nicht mehr nötig ist. Der Entwickler wird somit weitestgehend von dem zusätzlichen Aufwand der Entwicklung in einem Cluster entlastet.

7 Fazit und Ausblick

7.1 Fazit

Die Arbeit ist mit dem Ziel gestartet, einen Überblick über die neuartigen Optimierungswerkzeuge in der Entwicklung mit Kubernetes zu ermöglichen. Es sollte eine Möglichkeit aufgezeigt werden, den Umgang mit Kubernetes in der Softwareentwicklung zu erleichtern.

Neben dem Aufzeigen der unterschiedlichen Möglichkeiten und Eigenschaften der herfür genutzten Werkzeuge, wurden diese anhand eines Fallbeispiels, unter der Verwendung eines lokalen und remote Clusters erfolgreich verprobt. Dabei stellte sich heraus, dass die Werkzeuge sich eigene Schwerpunkte gesetzt haben, um das Problem der Integration von Kubernetes in der Softwareentwicklung zu lösen. Alle drei Werkzeuge bieten durch ihre Automatisierung des Bereitstellungsprozesses, eine klare Entlastung für den Entwickler und erhalten dabei den Workflow der Entwicklung. Die Entwicklung im Cluster unterschied sich, durch das sekundenschnelle updaten der laufenden Kubernetes Pods, kaum von einer lokalen Entwicklung. Das Werkzeug Garden konnte durch seinen Caching-Mechanismus sogar eine CI-Pipeline optimieren.

Durch die abschließende Evaluation wurden die drei Werkzeuge anhand definierter Kriterien verglichen und in Relation zueinander gesetzt. Die herausgearbeiteten Use-Cases und die aufgezeigte *best-practise* Konfiguration ermöglichen einen Überblick über die Werkzeuge und dient als Guide für zukünftige Integrationen.

7.2 Ausblick

Um das Projekt weiterzuführen wäre eine Untersuchung weiterer Werkzeuge mit anderen Schwerpunkten interessant. So gibt es bspw. das Werkzeug DevSpace, welches gleiche Funktionalitäten wie die vorgestellten Werkzeuge bietet, aber nicht Teil dieser Arbeit war. Ein weiterer Aspekt wäre die Verprobung der Werkzeuge in einem größeren Projekt mit einem Entwicklerteam. Da die Werkzeuge, im Rahmen dieser Arbeit, nur von einem Entwickler an einem konkreten Fallbeispiel verprobt wurden, war es nicht möglich alle Funktionen der Werkzeuge zu untersuchen. Besonders die Integration der Werkzeuge in einem größeren komplexeren Projekt bietet sich für eine weitere Untersuchung an, da es dabei ggf. zu Limitierungen der einzelnen Werkzeuge kommen kann.

Ein weiterer interessanter Aspekt wäre die Erweiterbarkeit der Werkzeuge zu untersuchen. Da alle drei Werkzeuge eine Open-Source Variante ihres Werkzeuges anbieten, könnte man diese durch eigene Plugins erweitern. Hier könnte bspw. ein Build-Werkzeug, welches bisher nicht nativ unterstütz wird, durch das Schreiben eines eigenen Plugins hinzugefügt werden.

Literaturverzeichnis

Bentley, Dan. 2020. Tilit Blog. [Online] 2020. [Zitat vom: 19. Juli 2020.] https://blog.tilt.dev/2020/04/21/tilt-cloud.html.

Boxell, Mickey. 2019. ITNext. [Online] 2019. [Zitat vom: 24. Mai 2020.] https://itnext.io/local-container-native-development-tools-ef4b1beb472c.

Docker. [o. Jahr]b. Docker Docs. [Online] [o. Jahr]b. [Zitat vom: 25. Mai 2020.] https://docs.docker.com/compose/.

—. [o. Jahr]a. Docker Docs. [Online] [o. Jahr]a. [Zitat vom: 2020. Mai 22.] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

Drilling, Thomas. 2018. Dev Insider. [Online] 2018. [Zitat vom: 11. Mai 2020.] https://www.dev-insider.de/images-und-container-unter-docker-a-735468/.

Edvald, Jon. 2019a. medium. [Online] 2019a. [Zitat vom: 21. August 2020.] https://medium.com/garden-io/you-dont-need-kubernetes-on-your-laptop-37653cbb28c9.

—. 2019b. medium. [Online] 2019b. [Zitat vom: 22. August 2020.] https://medium.com/garden-io/why-distributed-systems-are-hard-to-develop-and-how-to-fix-it-e5dfe9bac421.

Fink, Andreas. 2012. Enzyklopädie der Wirtschaftsinformatik. [Online] 2012. [Zitat vom: 4. Mai 2020.] https://enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Monolithisches-IT-System.

Flower, Martin. 2006. martinFlower. [Online] 2006. [Zitat vom: 7. Mai 2020.] https://www.martinfowler.com/articles/continuousIntegration.html.

Floyd, Blue und Karlstetter, Florian. 2017. Cloudcomputing Insider. [Online] 2017. [Zitat vom: 5. Oktober 2020.] https://www.cloudcomputing-insider.de/was-ist-software-as-a-service-a-622859/.

Garden. [o. Jahr]a. Garden. [Online] [o. Jahr]a. [Zitat vom: 21. August 2020.] https://garden.io/.

—. [o. Jahr]b. Garden. [Online] [o. Jahr]b. [Zitat vom: 27. August 2020.] https://garden.io/product.

—. 2020b. Garden Docs. [Online] 2020b. [Zitat vom: 22. August 2020.] https://docs.garden.io/using-garden/tasks.

—. 2020c. Garden Docs. [Online] 2020c. [Zitat vom: 22. August 2020.] https://docs.garden.io/basics/how-garden-works.

—. 2020d. Garden Docs. [Online] 2020d. [Zitat vom: 22. August 2020.] https://docs.garden.io/basics/stack-graph.

—. 2020e. Garden Docs. [Online] 2020e. [Zitat vom: 23. August 2020.] https://docs.garden.io/using-garden/using-the-cli.

—. 2020a. Garden Docs. [Online] 2020a. [Zitat vom: 22. August 2020.] https://docs.garden.io/guides/hot-reload.

—. 2020f. Garden Docs. [Online] 2020f. [Zitat vom: 23. August 2020.] https://docs.garden.io/using-garden/tests.

—. 2020g. Gareden Docs. [Online] 2020g. [Zitat vom: 24. August 2020.] https://docs.garden.io/guides/in-cluster-building#security-considerations.

—. 2020h. GitHub. [Online] 2020h. [Zitat vom: 27. August 2020.] https://github.com/garden-io/garden.

—. 2020i. GitHub. [Online] 2020i. [Zitat vom: 27. August 2020.] https://github.com/garden-io/garden/tree/master/examples/spring-boot-hot-reload.

Gentle, Lukas und Hausenblas, Michael. 2019. Einsam, gemeinsam oder im Team. *iX Developer.* 2019, Winter 19/20.

GitLab. [o. Jahr]. GitLab. [Online] [o. Jahr]. [Zitat vom: 8. Mai 2020.] https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

Google. 2020. GitHub. [Online] 2020. [Zitat vom: 7. August 2020.] https://github.com/GoogleContainerTools/distroless.

HJL. 2018. Dev Insider. [Online] 2018. [Zitat vom: 8. Mai 2020.] https://www.dev-insider.de/was-ist-continuous-integration-a-690914/.

Kubala, Nick und Wolf, Russel. 2019. Google Cloud Blog. [Online] 2019. [Zitat vom: 26. Juli 2020.] https://cloud.google.com/blog/products/application-development/kubernetes-development-simplified-skaffold-is-now-ga.

Kubernetes. 2019. Kubernetes. [Online] 2019. [Zitat vom: 18. Mai 2020.] https://kubernetes.io/docs/concepts/overview/components/.

—. 2020b. Kubernetes. [Online] 2020b. [Zitat vom: 22. Mai 2020.] https://kubernetes.io/docs/concepts/services-networking/service/.

—. 2020a. Kubernetes. [Online] 2020a. [Zitat vom: 18. Mai 2020.] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

Luber, Stefan. 2017. CloudComputing Insider. [Online] 2017. [Zitat vom: 7. Mai 2020.] https://www.cloudcomputing-insider.de/was-ist-cloud-native-a-669681/.

—. 2019. CloudComputing Insider. [Online] 2019. [Zitat vom: 2020. Mai 18.] https://www.cloudcomputing-insider.de/was-ist-kubernetes-k8s-a-832381/.

Malav, Bhagwati. 2017. medium. [Online] 2017. [Zitat vom: 4. Mai 2020.] https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4.

Mouat, Adrian. 2016. *Software entwickeln und deployen mit Containern.* 1. Auflage. Heidelberg : dpunkt, 2016. ISBN 978-3-86490-384-7.

Newman, Sam. 2019. *Monolith to Microservices.* 1. Auflage. s.l. : O'Reilly, 2019.

Pittet, Sten. [o. Jahr] . Atlassian. [Online] [o. Jahr] . [Zitat vom: 8. Mai 2020.] https://www.atlassian.com/continuous-delivery/continuous-deployment.

Pundsack, Mark. 2016. GitLab. [Online] 2016. [Zitat vom: 22. Mai 2020.] https://about.gitlab.com/blog/2016/05/23/gitlab-container-registry/.

Skaffold. 2020d. Skaffold Docs. [Online] 2020d. [Zitat vom: 31. Juli 2020.] https://skaffold.dev/docs/environment/profiles/.

—. 2020a. Skaffold Docs. [Online] 2020a. [Zitat vom: 31. Juli 2020.] https://skaffold.dev/docs/workflows/dev/.

—. 2020b. Skaffold Docs. [Online] 2020b. [Zitat vom: 31. Juli 2020.] https://skaffold.dev/docs/workflows/ci-cd/.

—. 2020c. Skaffold Docs. [Online] 2020c. [Zitat vom: 31. Juli 2020.] https://skaffold.dev/docs/pipeline-stages/filesync/.

—. 2020e. Skaffold Docs. [Online] 2020e. [Zitat vom: 6. August 2020.] https://skaffold.dev/docs/.

—. 2020f. Skaffold Docs. [Online] 2020f. [Zitat vom: 6. August 2020.] https://skaffold.dev/docs/pipeline-stages/testers/.

—. 2020g. Skaffold Docs. [Online] 2020g. [Zitat vom: 8. August 2020.] https://skaffold.dev/docs/workflows/debug/.

Telepresence. [o. Jahr]. telepresence. [Online] [o. Jahr]. [Zitat vom: 30. Mai 2020.] https://www.telepresence.io/discussion/overview.

Tilt. 2020. GitHub. [Online] 2020. [Zitat vom: 5. Oktober 2020.] https://github.com/tilt-dev/tilt-example-java/tree/master/4-recommended.

—. [o. Jahr]b. Tilt Docs. [Online] [o. Jahr]b. [Zitat vom: 19. Juli 2020.] https://docs.tilt.dev/snapshots.html.

—. [o. Jahr]a. Tilt Docs. [Online] [o. Jahr]a. [Zitat vom: 17. Juli 2020.] https://docs.tilt.dev/product_faq.html.

—. [o. Jahr]c. Tilt Docs. [Online] [o. Jahr]c. [Zitat vom: 19. Juli 2020.] https://docs.tilt.dev/.

—. [o. Jahr]d. Tilt Docs. [Online] [o. Jahr]d. [Zitat vom: 20. Juli 2020.] https://docs.tilt.dev/live_update_reference.html#synclocal_path-str-remote_path-str.

—. [o. Jahr]e. Tilt Docs. [Online] [o. Jahr]e. [Zitat vom: 20. Juli 2020.] https://docs.tilt.dev/api.html.

—. [o. Jahr]f. Tilt Docs. [Online] [o. Jahr]f. [Zitat vom: 20. Juli 2020.] https://docs.tilt.dev/example_java.html.

—. [o. Jahr]g. Tilt Docs. [Online] [o. Jahr]g. [Zitat vom: 24. Juli 2020.] https://tilt.dev/enterprise.

—. [o. Jahr]h. Tilt Docs. [Online] [o. Jahr]h. [Zitat vom: 24. Juli 2020.] https://docs.tilt.dev/choosing_clusters.html.

—. [o. Jahr]i. Tilt Docs. [Online] [o. Jahr]i. [Zitat vom: 25. Juli 2020.] https://docs.tilt.dev/cli/tilt.html.

Wagner, Stefan. 2019. Friedrich-Alexander Universität Erlangen-Nürnberg. [Online] 2019. [Zitat vom: 2020. Mai 22.] https://osr.cs.fau.de/wp-content/uploads/2019/03/wagner-2019-thesis.pdf.

Wolff, Eberhard. 2018. heise. [Online] 2018. [Zitat vom: 4. Mai 2020.] https://www.heise.de/developer/artikel/Microservices-Oder-lieber-Monolithen-3944829.html.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 16.10.2020