

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medieninformatik

Bachelorarbeit

von

Albert Hahn

**Konzeption und Implementierung einer Microservice
Architektur in einem hybriden kubernetes Cluster für
industrielle KI-Anwendungsfälle**

Conceptual Design and Implementation of a Microservice
Architecture in a Hybrid Kubernetes Cluster for Industrial
AI Use Cases

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medieninformatik

Bachelorarbeit

von

Albert Hahn

**Konzeption und Implementierung einer Microservice
Architektur in einem hybriden kubernetes Cluster für
industrielle KI-Anwendungsfälle**

Conceptual Design and Implementation of a Microservice
Architecture in a Hybrid Kubernetes Cluster for Industrial
AI Use Cases

Bearbeitungszeitraum: von 4. Oktober 2021
bis 3. März 2022

1. Prüfer: Prof. Dr.-Ing. Christoph Neumann

2. Prüfer: Prof. Dr. Dieter Meiller

Bestätigung gemäß § 12 APO

Name und Vorname
der Studentin/des Studenten: **Hahn, Albert**

Studiengang: **Medieninformatik**

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel:

**Konzeption und Implementierung einer Microservice Architektur in einem
hybriden kubernetes Cluster für industrielle KI-Anwendungsfälle**

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine
anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und
sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 28. Februar 2022

Unterschrift:

Bachelorarbeit Zusammenfassung

Studentin/Student (Name, Vorname):	Hahn, Albert
Studiengang:	Medieninformatik
Aufgabensteller, Professor:	Prof. Dr.-Ing. Christoph Neumann
Durchgeführt in (Firma/Behörde/Hochschule):	Krones AG, Neutraubling
Betreuer in Firma/Behörde:	Ottmar Amann
Ausgabedatum: 4. Oktober 2021	Abgabedatum: 3. März 2022

Titel:

**Konzeption und Implementierung einer Microservice Architektur in einem
hybriden kubernetes Cluster für industrielle KI-Anwendungsfälle**

Zusammenfassung:

Das Ziel dieser Bachelorarbeit ist es, eine flexible und nahtlose Lösung für ein Hybrides Cluster aus on-premise Edge Devices und Cloud Ressourcen bereitzustellen. Produktionslinienanwendungen/Microservices sollen zukünftig beliebig skalierbar und agil sein, dabei sollen für die Anwendungen generell keine Differenzierung zwischen offline und online Ressource getroffen werden. Im Zuge dessen wird die Umsetzbarkeit und Relevanz von cloudbasierten Microservices im Bereich der künstlichen Intelligenz auf einer zukünftigen Produktionsanlage untersucht.

Schlüsselwörter:

Inhaltsverzeichnis

1	Einleitung	2
1.1	Motivation	3
1.2	Zielsetzung	3
2	Grundlagen	4
2.1	Docker	4
2.1.1	Architektur	4
2.1.2	Images und Container	5
2.1.3	Containervirtualisierung	6
2.2	Kubernetes	7
2.2.1	Cluster	8
2.2.2	Pods	9
2.2.3	Deployment	9
2.2.4	Service	10
2.2.5	Ingress	11
2.2.6	Lightweight Kubernetes	12
2.2.7	Rancher	13
2.2.8	Hybrid Cloud	15
2.3	Microservice	15
2.3.1	Begriffserklärung	16
2.3.2	Charakteristiken	17
3	Analyse	20
3.1	Proof of Concept	20
3.1.1	Edge-Computing	20
3.1.2	Kubernetes	21
3.2	Resultate	23
4	Lösungsansatz	24
4.1	Fachkonzept	24
4.2	Grobkonzeption	24
4.2.1	Grundlegende Idee	25
4.2.2	Infrastruktur	26
4.2.3	Anwendung	26
4.2.4	Entwicklungsprozess	27

5	Lösungskonzept	29
5.1	Design Entscheidungen	29
5.1.1	Backend	29
5.1.2	Frontend	30
5.1.3	Kommunikation	30
5.1.4	Datenbank	31
5.1.5	Versionsverwaltungssystem	31
5.2	Entwicklung	31
5.2.1	Microservice-Entwicklung	31
5.2.2	Helm-Chart-Entwicklung	32
5.3	Architektur	33
5.3.1	Microservices	33
5.3.2	Helm-Installation	33
6	Umsetzung des Lösungskonzepts	36
6.1	Konfiguration und Einrichtung	36
6.1.1	SSL-Verschlüsselung	37
6.1.2	Node-Affinity	38
6.1.3	Taints and Tolerations	38
6.2	KubeVision	39
6.2.1	Frontend-Service	39
6.2.2	Authentication-Service	40
6.2.3	Facerecognition-Service	40
6.3	Gesichtserkennung	41
6.3.1	Alignment	41
6.3.2	Training	41
6.3.3	Model	41
6.4	Dockerisierung	41
6.4.1	Dockerfile	41
6.4.2	Docker-Compose	42
6.5	Helm-Chart	43
6.5.1	Service	43
6.5.2	Ingress	44
6.5.3	Deployment	45
6.5.4	Persistent-Volumes	46
7	Fazit und Ausblick	48
7.1	Fazit	48
7.2	Einschränkungen	48
7.3	Ausblick	49
8	Ergebnisse	50
8.1	Microservice	50
8.1.1	Frontend-Serivce	50
8.1.2	Backend-Serivce	50
8.1.3	Authentifizierungs-Serivce	50

8.1.4	Loadbalancer	50
8.1.5	Kubernetes Cluster	50
	Abkürzungsverzeichnis	51
	Literaturverzeichnis	52
	Abbildungsverzeichnis	56
	Quellcodeverzeichnis	58

Kapitel 1

Einleitung

Die Krones AG bietet Anlagen sowohl für die Getränkeindustrie als auch Nahrungsmittelhersteller, von der Prozesstechnik bis hin zur IT-Lösung. Die Komplettlinie beinhaltet auch das Bereitstellen von Software auf den einzelnen Produktionsanlagen. Hierfür werden eine Vielzahl von Produktionslinienanwendungen auf den Anlagen installiert, gewartet und verwaltet. Dementsprechend hoch ist der Aufwand, der Fehleranfälligkeiten sowie fehlende Frameworks, Bibliotheken und anderer Abhängigkeiten mit sich bringt. Eigene Server müssen für die Kommunikation der Anlagen verbaut und gewartet werden, was zusätzlich Ressourcen beansprucht und automatisch die Kosten für die Inbetriebnahme einer solchen Linie erhöhen. Die Weiterentwicklung der zukünftigen Bereitstellung von Produktionsanlagensoftware erfolgt mithilfe eines Proof of Concept (PoC), welcher die Möglichkeiten einer wartungsfreien Infrastruktur durch ein continuous Delivery System evaluiert. Dies verläuft in Zusammenarbeit mit dem Kooperationspartner und Softwareunternehmen SUSE GmbH, welches das wartungsfreie Betriebssystem SUSE Linux Enterprise Micro und die multi-cluster Orchestrierungsplattform Rancher anbietet.

Als Grundlage hierfür dient das Open-Source-System Kubernetes, welches zur Automatisierung, Skalierung und Verwaltung von containerisierten Anwendungen verwendet wird. Künftige Produktionsanlagen sollen mittels zusätzlicher Edge Devices als Knotenpunkte in einem Kubernetes-Cluster fungieren, Ressourcen teilen, untereinander kommunizieren und Softwarepakete unkompliziert bereitstellen. Die Integration der kompakten Linux-Rechner ermöglichen den variablen Einsatz von Hardwareressourcen des Kunden, der je nach Leistungsanspruch Knotenpunkte erweitern kann. Dabei soll es für die einzelnen Anwendungen möglich sein, sowohl auf cloudbasierten als auch auf on-premise Hardware zur Verfügung gestellt zu werden. Ein hybrides Kubernetes-Cluster ermöglicht es somit, lokale Rechenleistung oder öffentliche Cloudressourcen in der selben Softwareumgebung zu nutzen.

1.1 Motivation

Die Vorteile von Kubernetes und dem stetigen Paradigmenwechsel der Softwarelandschaft im Cloudbereich, welcher den Wechsel von monolithischen Architekturen zu flexibleren Microservice-Architekturen bevorzugt, sind das Hauptmotiv der Auswertung neuer, agiler Distributionsmöglichkeiten. Die Containerisierung von Anwendungen ermöglicht erst die Aufteilung großer Projekte in kleine unabhängige Services, die mittels Orchestrierungsplattformen adäquat konzentriert werden können. Namhafte Unternehmen wie Netflix, Amazon und Uber entwickeln und verwenden bereits robuste und komplexe Microservices die containerisiert auf Kubernetes-Plattformen verwaltet werden [1].

Durch die Flexibilität einer solchen Infrastruktur ist es möglich Anwendungsfälle im Bereich der künstlichen Intelligenz für die Industrie zu testen. Die Anlage Linatronic AI der Krones AG nutzt bereits Deep-Learning-Technologie, um in der Linie mittels Vollinspektion Schäden, Dichtflächen oder Seitenwanddicken zu erkennen und Prozesse zu optimieren [2]. Allgemein sind Anwendungen mit künstlicher Intelligenz durch ihre Komplexität und Vielzahl an Abhängigkeiten schwierig zu entwickeln und bereitzustellen. Eine passende Plattform für Anwendungsfälle mit Bezug zur künstlichen Intelligenz muss eine Vielzahl an Services anbieten. Zu diesen gehören die Verwaltung von Ressourcen wie Speicher, Rechenleistung und Verbindungsgeschwindigkeit, für die Datenübertragung bei der Ausführung einzelner Phasen von der Informationsverarbeitung bis hin zur Evaluierung und Entwicklung von Modellen im Bereich der künstlichen Intelligenz. [3].

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Microservice-Architektur in einem hybriden Kubernetes-Cluster. Das Endresultat soll eine Anwendung werden, die mittels einer Weboberfläche, welche über eine Domain erreichbar ist, ein Login-Verfahren mittels einem Backend-Service ermöglichen der ein Authentifizierungsverfahren per Gesichtserkennung verwendet. Diese Daten sollen schließlich verarbeitet und persistent gespeichert werden, um bei erneutem Aufruf der Website bestehen zu bleiben. Die Konzeption der Anwendung findet containerisiert auf mehreren Software und Hardware-schichten statt. Das ganze System wird auf einem Kubernetes-Cluster bereitgestellt und verwaltet. Das Bereitstellen eines Services kann bei Vorkonfiguration auf on-premise oder cloudbasierten Ressourcen stattfinden. Ein Ingress-Controller dient dabei als Loadbalancer und verteilt die Last beim Aufrufen der Website und der Kommunikation zwischen den Backend-Services.

Kapitel 2

Grundlagen

Dieses Kapitel erläutert die grundlegenden Begriffe und Konzepte, die zum Verständnis dieser Bachelorarbeit notwendig sind. Dabei wird der Technologie-Stack aufsteigend beschrieben. Als Fundament dient die Container Technologie Docker. Orchestriert wird diese durch die Containerplattform Kubernetes. Abschließend folgt ein Abschnitt zu Microservices.

2.1 Docker

In diesem Abschnitt wird die Technologie Docker näher erläutert und nicht das Unternehmen Docker, Inc. ,welches für die maßgebliche Entwicklung dessen verantwortlich ist [4, S.11]. Es folgt eine aufsteigende Erklärung der Architektur hin zum Aufbau eines Containers.

2.1.1 Architektur

Die Docker-Technologie ist in der Programmiersprache GO geschrieben und nutzt Funktionalitäten des Linux-Kernels, wie cgroups und namespaces. Namespaces ermöglichen die Isolation von Prozessen in sogenannte Container, welche unabhängig voneinander arbeiten [5]. Diese beinhalten alle nötigen Abhängigkeiten zur Ausführung der vordefinierten Anwendungen. Container gewinnen dadurch an Portabilität, sodass sie auf allen Infrastrukturen mit Docker-Laufzeit bereitgestellt werden können. Die Laufzeit setzt sich aus „runc“ einer low-level-Laufzeit und „containerd“ einer higher-level-Laufzeit zusammen (vgl. Abbildung 2.1). Runc dient als Schnittstelle zum Betriebssystem und startet und stoppt Container. Containerd verwaltet die Lebenszyklen eines Containers, das Ziehen von Images, das Erstellen von Netzwerken und die Verwaltung von runc. Die allgemeine Aufgabe des Docker-Daemons ist es, eine vereinfachte Schnittstelle für die Abstraktion der darunterliegenden Schicht zu gewährleisten, wie zum Beispiel dem Verwalten von Images, Volumes und Netzwerken [4, S.12]. Auf die Orchestrierung mit Swarm wird nicht weiter eingegangen, da sie zum Verständnis nicht nötig ist.



Abbildung 2.1: Docker Architektur in Anlehnung an [4, S.11]

2.1.2 Images und Container

Ein Docker-Image ist ein Objekt, das alle Abhängigkeiten, wie Quellcode, Bibliotheken und Betriebssystemfunktionen für eine Anwendung beinhaltet.

Registries

Das beziehen von Images erfolgt über sogenannte „Image Registries“. Bei Docker ist dies standardmäßig <https://hub.docker.com> und das eigene lokale Registry. Es ist auch möglich, eigene zu hosten oder diejenigen von Drittanbietern zu nutzen.

Schichten

Docker Images bestehen aus mehreren Schichten, jede davon abhängig von der Schicht unter ihr und erkennbar durch IDs in Form von SHA256-Hashes (vgl. Abbildung 2.2). Docker kann dadurch beim Bauen oder Updaten von neuen Images vorhandene Schichten erneut verwenden. Die feste Reihenfolge ermöglicht eine ressourceneffiziente Verwaltung von Builds, indem man oft wechselnde Schichten oben platziert. Die Leistung beim Erstellen und Zusammenführen von Schichten hängt vom Dateisystem des Hostsystems ab. Eine Schicht kann aus mehreren Dateien bestehen und einzelne Dateien aus der unterliegenden Schicht mit einer neuen ersetzen.

Das Starten eines Containers fügt auf die bereits bestehenden Schichten einen „Thin R/W layer“ - „Container layer“ hinzu. Dieser gewährt Schreib- und Lese-rechte während der Laufzeit des Prozesses. Jeder dieser Container hat somit einen

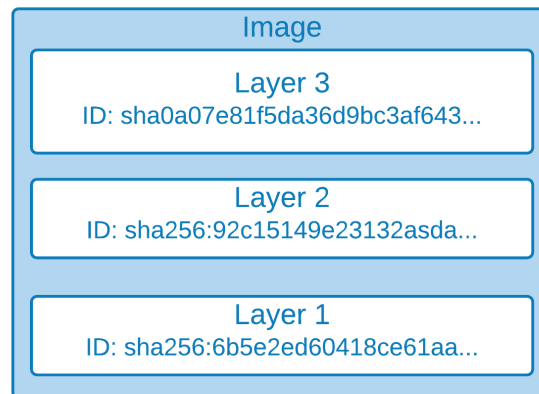


Abbildung 2.2: Image Layers in Anlehnung an [4, S.61]

individuellen Zustand, der unähnlich vom abstammendem Image ist. Bei Löschung des Containers verschwindet auch die dazugewonnene Schicht. Das Entfernen eines Images ist durch die Konzeption des Schichtensystem erst möglich, wenn alle darauf basierenden Container gelöscht sind [6].

Dockerfile

Zur Erstellung eines Docker-Images wird ein Dockerfile benötigt. Dies beinhaltet alle Anweisungen zum Aufbau der einzelnen Schichten. Diese Aufrufe erstellen die Schichten eines Images [7].

- **FROM** Erstellen einer Schicht auf Basis eines base-images.
- **COPY** Hinzufügen von Dateien aus dem aktuellen Arbeitsverzeichnis.
- **RUN** Bauen der Anwendung mit make.

Diese hingegen fügen nur Metadaten hinzu [7].

- **EXPOSE** informiert Docker, an welchem Port der Container innerhalb seines Netzwerks lauscht.
- **ENTRYPOINT** ermöglicht es, einen Container als ausführbare Datei zu starten.
- **CMD** Befehl beim Ausführen des Containers.

2.1.3 Containervirtualisierung

Aus dem Wissen des letzten Abschnitts lässt sich schlussfolgern, dass ein Container eine laufende Instanz eines Images ist. Vergleichbar ist dieses Konzept mit dem einer VM. Denn Images ermöglichen ähnlich wie VM-Templates die Erstellung von mehreren Instanzen durch eine Vorkonfiguration. Mit dem Unterschied, dass die Einrichtung von VMs arbeitsintensiver ist und weitaus mehr Ressourcen beansprucht, da sie ein ganzes Betriebssystem ausführt [8]. Containertechnologien bauen hingegen nur auf



Abbildung 2.3: Virtualisierungsmöglichkeiten angelehnt an [9].

bestimmte Funktionalitäten des Kernels auf und sparen damit an Rechenleistung (vgl. Abbildung 2.3).

Durch die Vorteile eines geteilten Kernels und dessen Betriebssystemabhängigkeiten, erzielen Virtualisierungen basierend auf Containern eine höhere Anzahl an virtuellen Instanzen. Images beanspruchen weniger Speicherplatz als hypervisor-basierende Ansätze [8].

Die Einsparung von Ressourcen und dem einfachen Bereitstellen auf Hostsystemen prädestinieren containerisierte Anwendungen für die Verwendung von Microservices auf Containerplattformen, wie Kubernetes.

2.2 Kubernetes

„Der Name Kubernetes stammt aus dem Griechischen, bedeutet Steuermann oder Pilot, [...] K8s ist eine Abkürzung, die durch Ersetzen der 8 Buchstaben "ubernete" mit "8" abgeleitet wird“ [10].

Dieser Abschnitt befasst sich zunächst mit den einzelnen Komponenten der Kubernetes-Architektur. Hinleitend werden spezielle Themen wie k3s, Hybrid Cloud und Rancher näher erläutert. Kubernetes ermöglicht die Orchestrierung von containerisierten Arbeitslasten und Diensten. Seit 2014 hat Google das Open-Source-Projekt zur Verfügung gestellt und baut auf 15 Jahre Erfahrungen mit Produktions-Workloads auf [10].

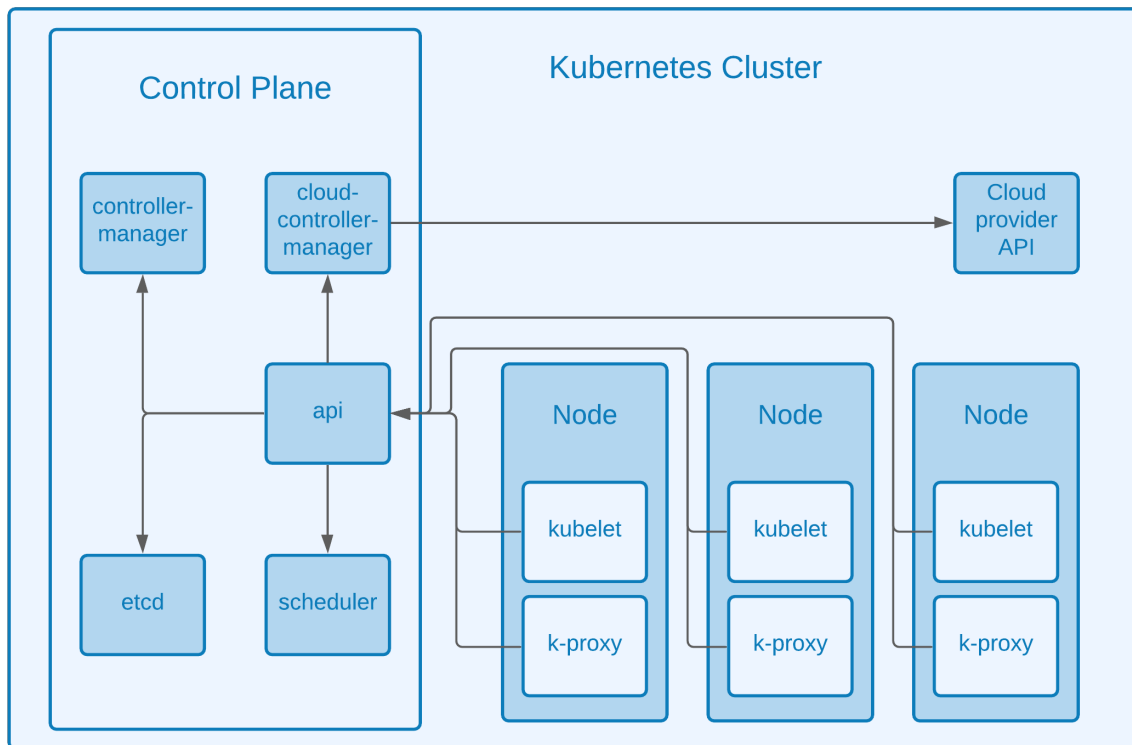


Abbildung 2.4: Komponenten eines Kubernetes Cluster in Anlehnung an [11].

2.2.1 Cluster

Die Zusammensetzung der beschriebenen Kubernetes-Komponenten ergeben ein Kubernetes-Cluster (vgl. Abbildung 2.4).

Control Plane

Control Planes¹ sind für die Steuerungsebene des Clusters zuständig. Dabei entscheidet und reagiert dieser auf globaler Ebene auf eintreffende Clustereignisse. Die Kubernetes-Dokumentation beschreibt diese Komponenten wie folgt [11]:

- **API-Server:** Der API-Server ist REST-konform und bietet eine Schnittstelle zu Diensten inner- und außerhalb der Control-Plane.
- **etcd:** etcd ist der primäre Datenspeicher von Kubernetes und sichert alle Zustände eines Clusters.
- **Scheduler:** Der Scheduler ist zuständig für die Verteilung und Ausführung von Pods auf Nodes.
- **Controller Manager:** Der Controller Manager reagiert auf Ausfälle von Nodes, stellt die korrekte Anzahl von Replikationen eines Pods sicher und verbindet Services miteinander.

¹Seit Kubernetes v1.20, ist Control Plane die korrekte Bezeichnung für die Master Node [12]

Node

Eine Node² ist eine Hardware-Einheit, die je nach Kubernetes-Einrichtung eine VM, eine physische Maschine oder eine Instanz in einer privaten oder öffentlichen Cloud darstellen kann. Diese umfasst folgende Komponenten [13]:

Container Laufzeit

Die Laufzeit wurde bereits in Abschnitt 2.1 ausführlich besprochen. Desweiteren ist es erwähnenswert, dass Containerd als Container-Laufzeit von Kubernetes nach Version 1.20 auslaufen wird [14]. Dies beeinträchtigt die spätere Implementierung dieser Arbeit jedoch nicht, da k3s Containerd als Standard Laufzeit weiterhin unterstützt.

Kubelet

Kubelet fungiert als „node agent“ und registriert die Node mit dem API-Server eines Clusters und stellt dabei sicher, dass Container innerhalb eines Pods funktionieren.

Kube-Proxy

Ein Kube-Proxy ist ein Netzwerk Proxy und verwaltet die Netzwerkzugriffe auf Nodes. Kube-Proxys erlauben die Kommunikation zwischen Pods inner- und außerhalb des Clusters.

2.2.2 Pods

Ein Pod stellt die kleinste Einheit eines Kubernetes-Clusters dar und ist eine Gruppe aus mindestens einem Container. Pods erlauben Containern die gemeinsame Nutzung von Speicher- und Netzwerkressourcen.

2.2.3 Deployment

Ein Deployment ist ein Ressourcenobjekt, das mit einem Deployment-Controller den gewünschten Zustand einer Anwendung aufrechterhält. Diese Spezifikationen sind in Form von YAML-Dateien definiert (vgl. Quellcode 2.1). Im Folgenden ist eine kurze Aufschlüsselung der einzelnen Instruktionen [15].

- **APIVersion:** definiert die einzelnen Workload-API-Untergruppen und die Version.
- **kind:** bestimmt das zu erstellende Kubernetes-Objekt.
- **metadata:** definiert einzigartige Bestimmungsmerkmale.
- **spec:** gewünschter Ausgangszustand des Objekts.

²Um den Sprachfluss zu wahren wird der englische Begriff Node, als Kubernetes-Ressourcenobjekt nicht übersetzt. Die Übersetzung Knoten findet lediglich als Hardwareinstanz statt.


```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: nginx
12     spec:
13       containers:
14         - name: nginx
15           image: nginx:1.14.2
16           ports:
17             - containerPort: 80
18
```

Quellcode 2.1: deployment.yaml [16]

Deployments und Pods

Das Einbinden von Pods in Deployments ermöglicht Kubernetes das Beziehen von Metadaten für die Verwaltung von Skalierung, Rollouts, Rollbacks und Selbstheilungsprozessen [17, S.75]. Der höhere Grad an Abstraktion dient auch der Aufteilung von Microservice-Stacks, zum Beispiel dem Aufteilen von Frontend- und Backend-Pods in eigene Deployment-Zyklen.

2.2.4 Service

Ein Service ist für die Zuweisung von Netzwerkdiensten zu einer logischen Gruppe an Pods zuständig. Services dienen als Abstraktion von Pods und ermöglichen die Replizierung und Entfernung von Pods ohne Beeinträchtigung der laufenden Anwendung [18].

Pods beanspruchen Netzwerkressourcen, wie IP-Adresse und DNS-Name innerhalb ihres Clusters. Der Ausfall oder die Zerstörung eines Pods führt zu Beeinträchtigung der Kommunikation zwischen Anwendungen. Services können dies präventiv verhindern, indem sie mit selector und labeler eine Kommunikation zwischen zwei Kubernetes Objekten etablieren. Das Beispiel zeigt eine solche Konfiguration (vgl. Quellcode 2.2). Die einzelnen Spezifikationen werden folgendermaßen definiert [18]:

- **selector:** definiert die Abbildung auf ein Label.
- **app:** führt den Service für Pods mit dem vorgegebenen Label aus.
- **ports:** Netzkonfiguration zwischen Service und Pod.

- **targetPort:** Port auf dem die Anwendung im Pod lauscht.
- **port:** Port auf dem der Service lauscht.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: nginx-service
5  spec:
6    selector:
7      app: nginx
8    ports:
9      - protocol: TCP
10      port: 80
11      targetPort: 9376
12
```

Quellcode 2.2: service.yaml [18]

Bei der Erstellung eines Services entsteht ein REST Objekt. Der zugehörige Service-Controller lauscht auf die Endpunkte des selektierten Pods und konfiguriert den Service dementsprechend.

2.2.5 Ingress

Ein Ingress ist ein Kubernetes-Ressourcenobjekt, das die Bereitstellung von internen Services auf öffentliche Endpunkte ermöglicht. Diese Routen werden mittels HTTP oder HTTPS freigegeben und können in Form einer URL verwendet werden [19]. Die Anforderung für die Implementierung eines Ingress ist der Ingress-Controller, eine Vielzahl an Optionen dafür wird in der Dokumentation aufgelistet [20]. Für die Realisierung des Prototyps kommt ein NGINX-Ingress-Controller in Einsatz, weshalb dieser näher erläutert wird.

NGINX-Ingress-Controller

Der Ingress-Controller ist für die Umsetzung einer vorgegebenen Objektspezifikation zuständig [19]. Die übliche Verwendung eines Controllers beinhaltet die Lastenverteilung durch Weiterleiten des Datenverkehrs an Services. Diese Kommunikation findet, wie auch bei dem NGINX-Ingress-Controller [21], in der Anwendungsschicht des OSI-Schichtenmodells statt und ermöglicht dadurch die Lastenverteilung von öffentlichen Endpunkten zu internen Pods in einem Cluster [22]. Wie für alle anderen Kubernetes-Objekte auch werden vordefinierte Aufgaben des Ingress-Controllers durch YAML-Dateien abgebildet (vgl. Beispiel 2.3). Im Folgenden finden sich wichtige Optionen, die genauer erklärt werden [19]:

- **ingressClassName:** definiert den Ingress-Controller.
- **rules:** die Zusammensetzung der einzelnen HTTP-Regeln.

- **host:** definiert das Ziel des eintreffenden Datenverkehrs.
- **paths:** gibt die Endpunkte des verbundenen Services an.
- **backend:** leitet die Anfragen an den Service mit der richtigen Port Zuweisung weiter.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: minimal-ingress
5    annotations:
6      nginx.ingress.kubernetes.io/rewrite-target: /
7  spec:
8    ingressClassName: nginx
9    rules:
10     - http:
11       paths:
12         - path: /testpath
13           pathType: Prefix
14           backend:
15             service:
16               name: nginx-service
17             port:
18               number: 80
19
```

Quellcode 2.3: ingress.yaml [19]

2.2.6 Lightweight Kubernetes

Lightweight Kubernetes auch K3s genannt ist eine Open-Source-Kubernetes-Distribution des Unternehmens Rancher. Der größte Unterschied der Distribution ist die Speichernutzung auf Hostsystemen mit einer einzelnen Binärdatei von nur 40MB. Durch die Verschlinkung der Distribution ist der ideale Anwendungszweck IoT-Geräte mit wenig Rechenleistung. Denn die minimalen Systemanforderungen für Hostsysteme liegen bei 512MB Hauptspeicher und einer Pi4B BCM2711, 1.50 GHz CPU³ [24]. Der hauptsächliche Verwendungszweck von k3s liegt in IoT-Geräte, da sekundäre Kubernetes-Inhalte entfernt wurden. [25]. Trotz dieser Reduzierung bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden, soweit möglich, parallel auf dem neusten Stand gehalten [26].

Besonderheiten

Die Abbildung 2.5 zeigt die Architektur von k3s auf. Das Kubernetes-Äquivalent zur Control Plane und Node sind Server und Agent. Eine Besonderheit hiervon ist,

³Einplatinencomputer Raspberry Pi 4B, basierend auf ARM [23]

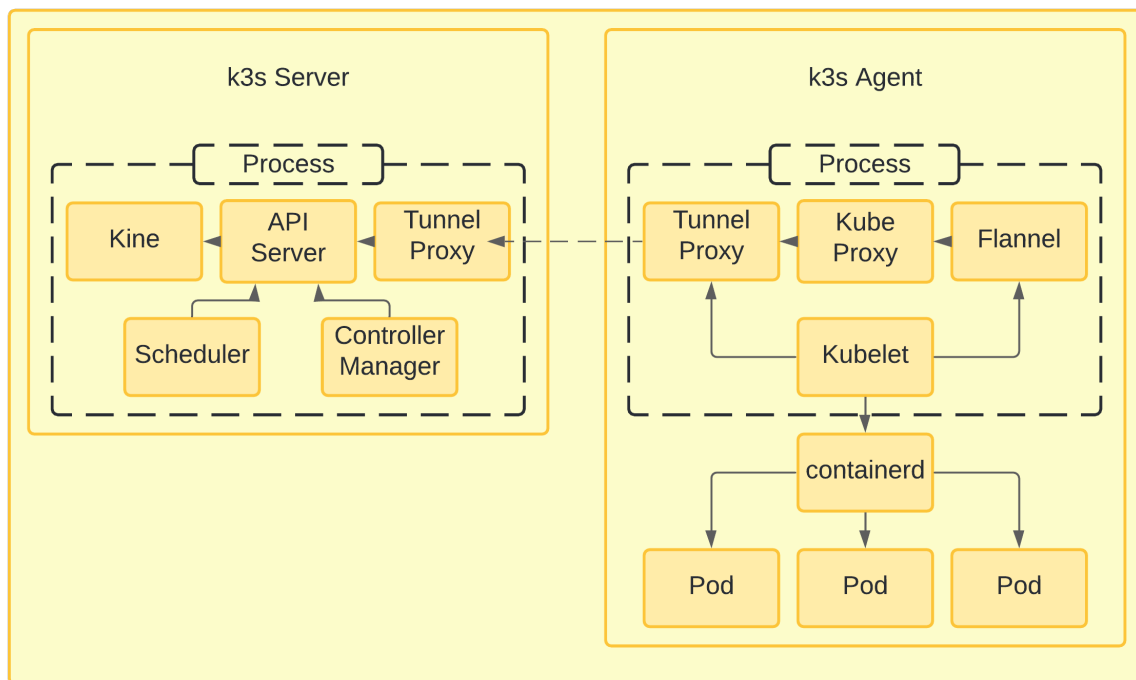


Abbildung 2.5: K3s Architektur in Anlehnung an [25].

dass Server parallel einen Agent-Prozess auf dem selben Knoten starten und somit Arbeitslasten mithilfe von Kubelet ausführen [27]. Weiterhin wird, im Gegensatz zu Kubernetes, containerd weiterhin unterstützt und kommt mit Kubelet vorinstalliert [25]. Zwei weitere Unterschiede werden näher erläutert:

Kine das Akronym steht für „Kine is not etcd“ und ist eine Abstraktionsschicht für die etcd API und übersetzt die Aufrufe von Kubernetes in sqlite, Postgress, Mysql und dqlite [26]. Dadurch kann der Backend Speicher des Clusters durch die oben genannten Datenbanksysteme ersetzt werden.

Flannel ist ein überlagerndes Netzwerkmodell in k3s und ermöglicht IPv4 Netzwerke innerhalb eines Clusters mit mehreren Knoten. Dazu wird eine einzelne Binärdatei gestartet, welche Agents auf Hostssystemen startet. Dieser alloziert Subnetze in einem vorkonfigurierten Adressraum. Das Modell ist dabei für die Übertragungsart des Datenverkehrs zwischen unterschiedlichen Knotenpunkten zuständig. Die Speicherung der Netzwerkkonfiguration erfolgt über etcd oder der Kubernetes-API [28].

2.2.7 Rancher

In diesem Unterabschnitt wird die Open-Source-Lösung Rancher von dem gleichnamigen Unternehmen zur Orchestrierung von Kubernetes Clustern näher behandelt. Sie ermöglicht das Verwalten von Kubernetes-Clustern auf der eigenen Infrastruktur, sowohl vor Ort als auch in der Cloud. Die Bereitstellung von Clustern mittels Rancher ist Cloud-Anbieter unabhängig, weshalb Cluster in der Praxis mit derselben Rancher

Instanz auf AWS, Azure oder anderen Cloud-Anbietern betreut werden können [29].

Die Rancher-Benutzeroberfläche vereinfacht das Steuern von Arbeitslasten, auf einer zentralen administrativen Instanz, welche gleichzeitig Authentifizierung und Rechteverteilung von Benutzern anbietet. Das grundsätzliche Verwalten von Arbeitslasten verlangt kein tiefgründiges Wissen bezüglich Kubernetes-Konzepte. Die mitgelieferten Tools ermöglichen die Auslieferung und Verbindung von Kubernetes-Objekten und abstrahieren die Komplexität, die für die Betreuung eines solchen Systems notwendig sind [29, 30].

Für komplexere Konfigurationen kann über die Oberfläche ein Terminal mit Kubectl aufgerufen werden. Wie auch in Kubernetes ist der Zugang auf ein Kubernetes-Cluster von einer lokalen Entwicklungsumgebung mit einer kubeconfig-Datei möglich, diese beinhaltet die Adresse zum Rancher-Server, Nutzerrechte und Zertifizierungszeichen [31].

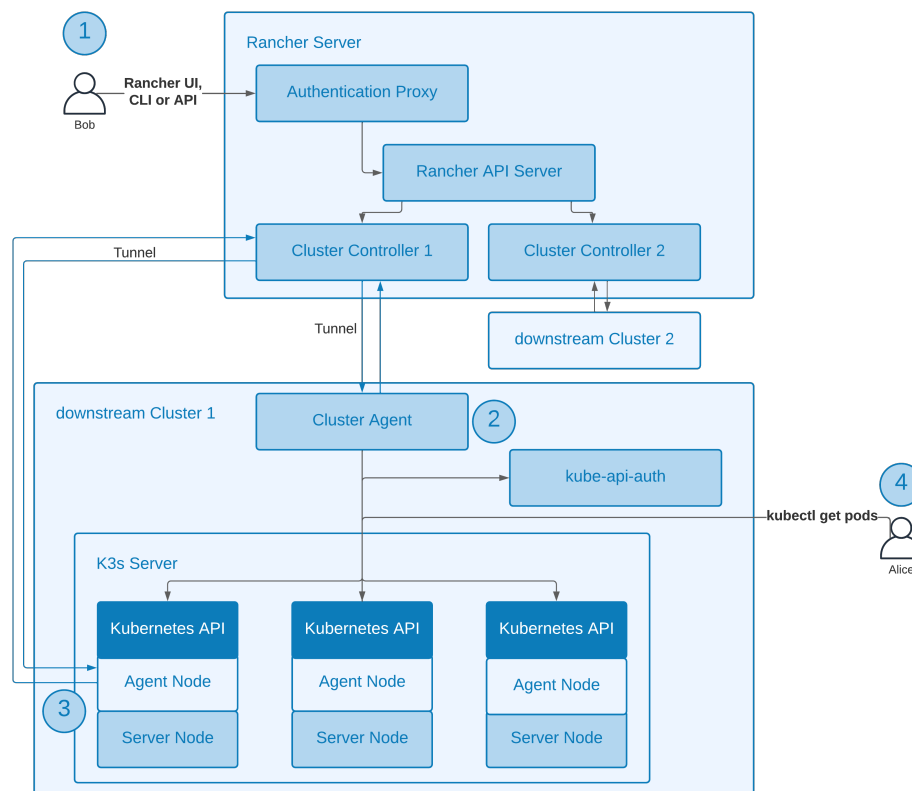


Abbildung 2.6: Rancher Server Kommunikation mit einem downstream k3s Cluster, überarbeitete Abbildung von [32]. (Im Sinne der späteren Architektur nachgebildet)

Die Abbildung 2.6 zeigt den Vorgang von zwei Benutzern, die auf ein von Rancher verwaltetes downstream-k3s-Cluster⁴ zugreifen. Die nachfolgende Beschreibung aus der Dokumentation gibt die einzelnen Schritte mit der in der Abbildung nummerierten Posten wieder [32].

⁴Die offizielle Bezeichnung für ein Kubernetes-Cluster unter Rancher ist **downstream Cluster** [33]

1. Zuerst authentifiziert sich Bob mit seinen Benutzerdaten bei dem Authentifizierungs-Proxy an seinem Rancher-Server. Dieser Proxy leitet den Aufruf über eine Kommandozeile oder der Rancher-Benutzeroberfläche zu der ausgewählten downstream-Cluster-Instanz weiter und führt diese aus. Dafür wird vor dem Weiterleiten des Aufrufs der angemessene Kubernetes-Impersonation-Header gesetzt, welcher sich als Service-Account der Rancher-Instanz ausgibt.
2. Die Übertragung des Aufrufs erfolgt über einen Cluster-Controller auf dem Rancher-Server und dem parallel laufenden Cluster-Agent des downstream-Clusters. Der Controller ist für die Überwachung, Veränderung und Konfiguration von Zuständen auf dem laufenden Cluster zuständig.
3. Wenn der Cluster-Agent nicht erreichbar ist, werden die Aufrufe an den Node-Agent⁵ überreicht, welcher standardmäßig auf jedem downstream-Cluster läuft.
4. Zuletzt hat auch die Benutzerin Alice die Möglichkeit, sich über einen autorisierten Cluster-Endpunkt zu verbinden. Denn jeder downstream-Cluster verfügt über eine Kubeconfig, welche den Zugang ohne Authentifizierungs-Proxy erlaubt. Durch den Microservice kube-api-auth wird eine Kommunikation über einen Web-Hook realisiert, der die Verbindung zwischen Alice und dem downstream-Cluster aufbaut. Dies ermöglicht die Verwendung von Befehlszeilentools, wie Kubectl und Helm.

2.2.8 Hybrid Cloud

Eine Hybrid-Cloud ist eine Kombination aus öffentlichen und privaten Cloud-Diensten, die auf einer einzigen Infrastruktur laufen. Dies ermöglicht die flexible Orchestrierung von Anwendungen auf Hostsystemen vor Ort oder in der Cloud [35].

Der Schwerpunkt solcher Hybrid-Clouds liegt dabei bei der Portierbarkeit der Arbeitslasten auf allen Cloud-Umgebungen. Dafür ist die Aufbereitung oder Entwicklung alter oder neuer Anwendungen für cloudnative Technologien nötig; mehr dazu im Abschnitt 2.3 zu Microservices. Private-Clouds können auch von Drittanbietern, durch externe Rechnzentren, als Enterprise Modell angeboten werden. Dabei ist die Nutzung eines einzigen Betriebssystems ratsam, um Abhängigkeiten bei der Automatisierung von cloud-nativen Anwendungen zu verhindern. Die Verwaltung erfolgt, dabei mit einer Container-Orchestrierungsplattform, wie Kubernetes und ermöglicht die nahtlose Implementierung von Cloud-Umgebungen [35].

2.3 Microservice

Im Folgenden wird der Microservice Architektur-Stil und dessen Eigenschaften näher erläutert. Als Hauptquelle dient der häufig zitierte Artikel [36] von Fowler und Lewis.

⁵Ein Rancher-DaemonSet zur Interaktion mit Nodes. Nicht zu verwechseln mit dem Node-Agent von k3s [34].

2.3.1 Begriffserklärung

Fowler und Lewis beschreiben den Microservice-Architektur-Stil als Entwicklung einer einzigen Anwendung, die aus einer Reihe unabhängiger Dienste besteht. Die Kommunikation der einzelnen Dienste untereinander wird häufig durch API-Aufrufe über HTTP realisiert. Diese Dienste sind vollautomatisch auszuliefern und orientieren sich bei der Entwicklung um Business-Capabilities⁶. Zusammenhängende Dienste werden minimal zentral gehalten und können in unterschiedlichen Programmiersprachen oder Technologien realisiert werden [36].

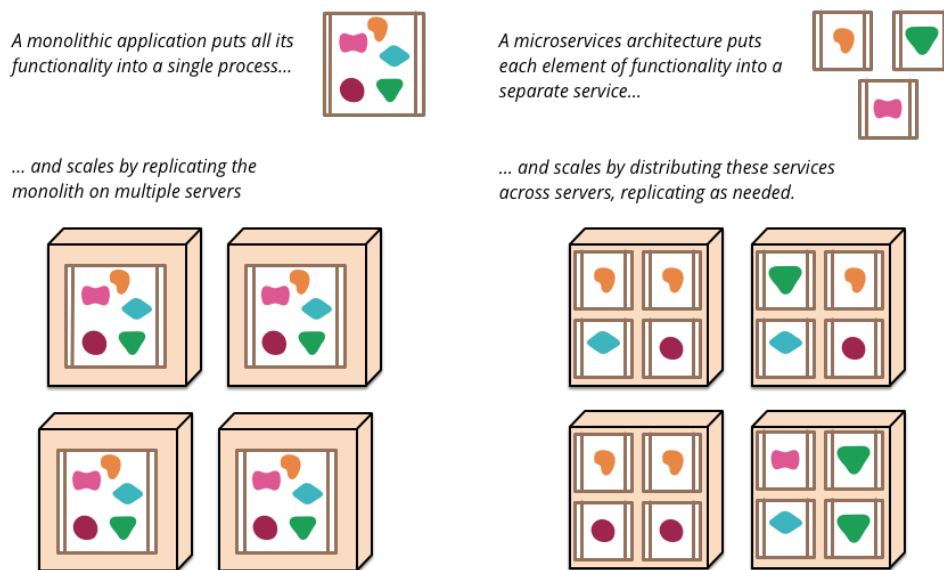


Abbildung 2.7: Gegenüberstellung von Monolithen und Microservices [36]

Sinnvoll ist es hierbei, den Vergleich zu monolithischer Softwareentwicklung zu ziehen. Ein Monolith folgt hierbei der Grundprämisse mittels der verwendeten Programmiersprache die Anwendung in einzelne Klassen, Funktionen und Namensräume aufzuteilen. Dieser Ansatz ist gängig und erfolgsversprechend. Jedoch argumentieren Fowler und Lewis, dass mit dem Zuwachs an Cloud-Technologien dieser Ansatz immer frustrierender für Entwickler ist, denn bereits kleine Änderungen an einem Modul benötigen einen neuen Software-Build und Auslieferungsprozess. Weiterhin merken Fowler und Lewis an, dass die Skalierbarkeit einer solchen Architektur mehr Ressourcen erfordert, da nicht einzelne Teile der Anwendung repliziert werden, sondern der vollständige Monolith (vgl. Abbildung 2.7). Die Verwendung von einzelnen Diensten würde dieser Problematik entgegenreten und es Entwicklerteams ermöglichen einzelne Softwarekomponenten zu verwalten und gegebenenfalls in einer anderen Programmiersprache zu verwirklichen [36].

⁶Business-capability bezeichnet ein Konzept, das aus Sicht der Geschäftsarchitektur modelliert wird [37]

2.3.2 Charakteristiken

Eine Microservice-Architektur prägt sich durch bestimmte Charakteristika aus. Die Architektur muss nicht zwingend alle in diesem Abschnitt beschriebenen Eigenschaften erfüllen. Jedoch sollte ein Großteil der Konzepte in einer Microservice-Architektur auffindbar sein [36]. Die folgenden Unterabschnitte erläutern diese Charakteristika etwas näher.

Komponententrennung durch Dienste

Fowler definiert Komponenten einer Software wie folgt:

„Eine Komponente ist eine Softwareeinheit, die unabhängig austauschbar und erweiterbar ist.“ [38]

Dienste einer Microservice-Architektur stellen Softwarekomponenten dar, die mittels Web-Service-Anfragen oder Remote-Procedure-Calls⁷ interagieren. Bibliotheken hingegen beschreiben einen Verbund aus mehreren Komponenten, die lokale Funktionsaufrufe nutzen. Der resultierende Vorteil ist, dass Dienste unabhängig voneinander verändert und ausgeliefert werden können. Denn bei Prozessen mit mehreren eingebundenen Bibliotheken, muss die gesamte Anwendung neu ausgeliefert werden [36].

Dadurch wird der Fokus auf die Entwicklung von unabhängigen Diensten umso wichtiger, da die Veränderung an kooperierenden Schnittstellen zum Ausfall anderer Dienste führt. Um dem entgegenzuwirken, müssen Schnittstellen gut koordiniert werden und eine starke Kohäsion gewährleisten. Service Contracts⁸ der jeweiligen Dienste müssen sinnvoll gestaltet werden. Weiterhin müssen Schnittstellen grobkörniger entworfen werden, um den höheren Ressourcenverbrauch der lokalen Variante auszugleichen [36, 41].

Ein Dienst kann jedoch aus mehreren Prozessen bestehen. Ein Beispiel wäre ein Anwendungsprozess mit einer Datenbank, die nur von dieser Anwendung genutzt wird [36, 41].

Strukturierung nach Business-Capabilities

Bei der Entwicklung von großen Anwendungen werden Teams oft nach technologischen Schichten getrennt. Es werden Teams aus Benutzeroberflächen-, Anwendungs- und Datenbankentwicklern gebildet. Die Entwicklung einer Microservice-Architektur bedarf jedoch eine Organisation um die Business-Capabilities. Entwickler arbeiten funktionsübergreifend in allen Bereichen der Softwareentwicklung und bringen vielfältige Kompetenzen mit. Der Grund dafür ist, dass bei Konstellationen mit einseitiger Softwarekompetenz, kleinste Änderungen zu teamübergreifenden Projekten und den

⁷Remote-Procedure-Calls bezeichnet die Ausführung eines lokalen Aufrufs auf einem anderen Dienst [39].

⁸Service Contracts bezeichnen die Vereinbarung zwischen zwei Diensten, darin werden die Übertragungsformate von Daten festgelegt [40].

damit verbundenen Kosten führt. Effiziente Entwickler werden sich immer für den Weg des geringsten Widerstands entscheiden und ihre Logik dort implementieren, zu der das Team Zugang hat [36].

Produkte nicht Projekte

Microservice-Entwicklungen tendieren dazu, den kompletten Lebenszyklus einer Software zu begleiten. Der inspirierende Leitspruch bei Amazon dazu ist

„you build it, you run it “ [42].

Dem Gedanken nach übernimmt das Entwicklungsteam die vollständige Produktion der Software und übergibt diese nicht an ein Wartungsteam. Dadurch stehen die Entwickler im direkten Kontakt mit dem Endnutzer und erfahren wie sich die Software im Betrieb verhält, da diese auch Zuständigkeiten des Supports übernehmen [36].

Intelligente Endpunkte statt komplexer Infrastruktur

Die Kommunikation von Diensten über Endpunkte soll so weit wie möglich entkoppelt und kohäsiv sein. Anwendungen im Microservice-Stil enthalten ihre eigene Logik und agieren als Filter für das Empfangen, Verarbeiten und Beantworten einer Anfrage. Die Umsetzung erfolgt dabei mit RESTful-Protokollen für die Kommunikation über HTTP oder der leichtgewichtigen Kommunikation mit Messaging⁹. Ein weiterer Ansatz ist der Nachrichtenaustausch über leichtgewichtige Bussysteme. Die gewählte Infrastruktur muss hier nicht mehr als einen rudimentären Informationsaustausch gewährleisten. Die Dienste sind so konzipiert, den größten Mehrwert über Endpunkte zu erreichen und Redundanz beim Nachrichtenaustausch zu vermeiden.

Dezentrale-Governance

Die Dezentralisierung einer Anwendung in Softwarekomponenten ermöglicht den Einsatz von unterschiedlichen Technologien. Da die einzelnen Anwendungskomponenten über Endpunkte kommunizieren, ist die Wahl der Programmiersprache weniger relevant als bei einer monolithischen Architektur. Entwicklerteams gewinnen so an Handlungsspielraum und können bessere Werkzeuge für spezifische Probleme verwenden [36].

Dezentrales Datenmanagement

Die Dezentralisierung von Daten geschieht auf höchster Ebene und abstrahiert diese für kontextbasierende Modelle. Die Integration solcher Modelle wird durch die unterschiedliche Auffassung verschiedener System erschwert. Dabei besteht die Gefahr das Abteilungen innerhalb eines Unternehmens Attribute unterschiedlich interpretiert und zu Inkonsistenz in Datensätze führt. Eine Anwendung mit getrennten Softwarekomponenten erhöht diese Komplexität weiter [36]. Weshalb es sinnvoll ist, einen „Bounded

⁹Kommunikation über Binäre Protokolle wie Protocol-Buffers [43].

Context“ zu definieren, welcher zur Darstellung von Wechselwirkungen eines Modells innerhalb größerer Teams dient [44].

„Design for failure“

Softwarekomponenten müssen den Ausfall von anderen Diensten tolerieren. Eventbasierte Kommunikation führt oft zu Fehlverhalten und kann durch Überwachungstools präventiv verhindert werden [36].

Kapitel 3

Analyse

Das vorherige Kapitel widmete sich dem Aufbau von Containern und der Verwaltung dessen. Darauf aufbauend auch die mögliche Realisierung von Anwendungen im Micorservice-Architektur-Stil. Dieser Teil widmet sich den Innovationsforschungen der Krones AG in Form eines Proof of Concepts. Abschließend folgen die Resultate und möglichen neuen Anwendungsgebiete im Bereich Cloud-Technologie.

3.1 Proof of Concept

Die Krones AG entwickelt neue Konzepte, um Produktionsanlagen standortübergreifend zu modernisieren. In einem davon wurde ein Proof of Concept (PoC) mit dem Software-Unternehmen SUSE durchgeführt, um die Umsetzung neuer Cloud-Technologien zu evaluieren. Die folgenden Punkte behandeln die Kernthemen des PoC wie dem Edge-Computing und Kubernetes.

3.1.1 Edge-Computing

Edge-Computing bezeichnet die dezentrale Verarbeitung von Daten in direkter Nähe der Datenquelle. Das verringert den Bedarf an lokale Rechenzentren und senkt die Latenzzeiten bei der Übertragung von Daten. Betrachtungsgegenstand des PoC war die Virtualisierung der bereits vorhandenen Industrierechner der Firma B&R, um sie als Virtual-Edge-Devices zu verwenden. Auf den Virtual-Edge-Devices werden dann Operationen wie Erfassen, Aggregieren und Aufbereiten von Daten direkt an der Anlage ausgeführt. Die derzeitigen Anwendungen der Krones AG sind für das Betriebssystem Windows konzipiert und entwickelt worden. Für das Edge-Szenario soll aber ein auf Linux basierendes Betriebssystem verwendet werden, weshalb die Integration über Virtualisierungsmöglichkeiten realisiert wird.

Virtualisierung

Das Unternehmen B&R steht in Kooperation mit der Firma Real-Time-Systems (RTS), welches Technologien für die Virtualisierung von Echtzeit Betriebssystemen anbietet

[45]. Dafür wird ein Hypervisor genutzt, um gleichzeitig unterschiedliche Echtzeit Betriebssysteme in Form von VMs auszuführen. Dies ermöglicht auch die Zuteilung von Hardwareressourcen auf den laufenden VMs. Ein Vorteil ist das keine zusätzliche Hardware benötigt wird. Die Zuweisung für Hardwareschnittstellen wie Ethernet USB-Ports ist durch diesen Ansatz auch möglich. Virtuelle Netzwerke erlauben die Zuweisung von IPv4 und Mac-Adressen einzelner Prozessorkerne, welche eine direkte Kommunikation über Internetprotokolle wie TCP/IP oder COBRA ermöglichen. Weiter kann jedes virtualisierte Betriebssystem Daten über eine gemeinsame Speicherpartitionen verwaltet werden [46].

Connected Human Interface (HMI)

Die Produktionsanlagen nutzen Windows 10 Embedded als Betriebssystem. Darauf läuft das HMI mit einer Touch-Oberfläche für Benutzereingaben. Dies ist für die zentrale Überwachung und Steuerung von Anlagenprozessen zuständig. Und erlaubt die Einteilung von Produktionsrelevanten Aufgaben wie Wartungsarbeiten, Materialversorgung und Qualitätskontrollen [47].

SUSE Linux Enterprise Micro

Das für Edge-Szenarien entwickelte Open-Source-Betriebssystem SUSE Linux Enterprise Micro ist das zweite virtualisierte Betriebssystem, dass auf dem Hypervisor laufen soll. Es arbeitet mit transactional-updates, welche Updates erst aktivieren, wenn das Betriebssystem neu gestartet wurde. Erfolgt das Update nicht wird ein Rollback zum vorherigen Versionszustand durchgeführt und ermöglicht wartungsfreie Zustände der Geräte. Basierend auf der Idee von containerisierten Arbeitlasten und Microservices wurde das Betriebssystem entwickelt.

3.1.2 Kubernetes

Auf der Grundlage des Edge-Computing war ein weiterer Schwerpunkt des PoC die Orchestrierung einer solchen Infrastruktur. Dabei sollen die einzelnen Virtual-Edge-Devices in Zukunft als Knotenpunkte für ein Kubernetes-Cluster dienen. Dafür wird die für Edge-Szenarien entworfene Kubernetes-Distribution k3s installiert.

Auf dieser sollen containerisierte Anwendungen ausgeliefert und bereitgestellt werden. Anwendungen wie Microservices können innerhalb des Cluster über Endpunkte oder Kubernetes-Objekte kommunizieren. Dadurch müssen Hostsysteme und die darauf laufende Software nicht mit deren Netzwerkadresse angesprochen werden, und zusätzlich entfällt die Vorkonfiguration von Anwendungen durch die Architektur von Containern.

Rancher

Zur Orchestrierung der Kubernetes-Cluster Instanzen wurde die Orchestrierungsplattform Rancher näher untersucht. Diese ermöglicht die zentrale Verwaltung mehrerer

Kubernetes-Cluster in Produktionsumgebungen. Weiterhin wurde die Bedienbarkeit der Benutzeroberfläche untersucht und es wurden Tests durchgeführt, bei welchen containerisierte Arbeitslasten verwaltet und Kubernetes-Objekte erstellt und miteinander verbunden. Ein weiterer Vorteil ist die Auslieferung von Kubernetes-Anwendungen durch den Apps & Marketplace von Rancher.

Helm

Helm ist ein Kubernetes-Package-Manager und ermöglicht die Erstellung, Installation und das Updaten von Kubernetes Anwendungen. Wichtige Konzepte von Helm sind Charts, diese sind eine Ansammlung von Informationen zur Erstellung von Anwendungen als Kubernetes-Instanz. Configs beinhalten die Informationen der Konfiguration von Charts und erstellen oder verpacken diese. Release bezeichnet eine laufende Instanz eines Chart in Kombination mit einer spezifischen Config [48].

Helm ist eine ausführbare Datei, die aus einem Kommandozeilentool besteht, dem Helm-Client. Dieser erlaubt die lokale Entwicklung von Charts und dem verwalteten von Repositories und unterschiedlicher Versionen. Weiterhin dient der Client als Schnittstelle zur Helm-Library, welche Operationen mit dem Kubernetes-API-Server ermöglichen. Dadurch können Charts und Konfigurationen als ein Release gebildet und Charts in einem Kubernetes-Cluster installiert, deinstalliert und aktualisiert werden. Die Konfigurationsdateien von Helm werden in der Regel, wie auch bei Kubernetes-Ressourcenobjekten üblich, in YAML geschrieben [48].

Der Anwendungsfall in Bezug auf Kronos AG ist die kundenindividuelle Konfiguration von Kubernetes Anwendungen die in einem Kunden Repository gespeichert werden. Die Kundenkonfigurationen stehen dabei als Helm-Charts verfügbar und ermöglichen eine vereinfachte Auslieferung und Bereitstellung durch Helm-Repositories. Der Rancher Apps & Marketplace ist hierbei eine Möglichkeit Helm-Charts, über eine Benutzeroberfläche zu konfigurieren und installieren.

3.2 Resultate

Die Bestrebungen des PoC zeigten neue Anwendungsgebiete für die weitere Untersuchung von Kubernetes relevanten Themen. Während der Umsetzungsphase zeigten folgende Themen Potenzial.

Hybrid Cloud

Die zukünftige Kubernetes-Infrastruktur kann die Integration von On-Premise und Cloud-Ressourcen in einer Softwareumgebung ermöglichen. Kunden können sensible Daten in Ihrer eigenen privaten Cloud oder einem lokalen Rechenzentrum speichern und gleichzeitig die Vorteile von den erhöhten Rechenressourcen einer verwalteten Public-Cloud nutzen. Kubernetes verfügt über Funktionen, die eine Aufteilung der Arbeitslasten in spezifische Cloud-Umgebungen ermöglicht.

Die Modularität des Kubernetes-Cluster ermöglicht die Steigerung der Gesamtleistung durch Cloud-Ressourcen oder der Integration von neuer Hardware vor Ort. Dies erfordert auch die gegebene Modularität der Softwarekomponenten auf dem Kubernetes-Cluster.

Microservices

Wie in Abschnitt 2.3 behandelt werden die Kernfunktionalitäten der zu entwickelnden Anwendung in einzelne Dienste aufgeteilt. Anwendungen können in Zukunft auf Virtual-Edge-Devices oder in der Cloud bereitgestellt werden. Die modulare Entwicklung von Anwendungen, ermöglicht die geringe Rechenleistung von Virtual-Edge-Devices aufzuteilen. Der zukünftige Kunde kann bei diesem Ansatz nur die Dienste wählen die er für seine Anlage benötigt.

Ein weiterer Vorteil ist das eine modulare Architektur auf Containern die Auslieferung und Bereitstellung erleichtert. Der Package-Manager Helm kann dabei die Vorkonfiguration der einzelnen Dienste gewährleisten.

Künstliche Intelligenz

Das Kubernetes-Cluster bietet eine Infrastruktur für Anwendungen mit Bezug zu Themen aus der künstlichen Intelligenz. Datensätze von Produktionsanlagen können in Echtzeit- oder aggregierte Form verschickt oder gesammelt werden. Diese können dann zur Auswertung von Deep-Learning-Modellen verwendet werden. Die Auswertung der Modelle kann dann von Cloud-Ressourcen oder On-Premise Hardware mit Grafikkarten übernommen werden.

Kapitel 4

Lösungsansatz

Das folgende Kapitel beschreibt den nötigen Lösungsansatz der späteren Konzeptentwicklung.

4.1 Fachkonzept

Anforderungen

Die zu entwickelnde Anwendung soll ein Anwendungsszenario für die neu erschlossenen Anwendungsgebiete des PoCs umsetzen. Die Anwendung muss dabei eine Vielzahl an Anforderungen erfüllen. Die Software muss einfach Ausliefern und bereitzustellen sein. Dieser Prozess muss vor Inbetriebnahme automatisch verlaufen und eine individuelle Konfiguration anbieten. Weiterhin muss bestimmbar sein auf welchen Ressourcentyp die Anwendung bereitgestellt wird. Die Anwendung muss skalierbar sein und mit der Infrastruktur wachsen können. Sie muss auch die Verwaltung von Datensätzen ermöglichen. Auch die Kommunikation zwischen anderen Anwendung muss möglich sein. Die Software muss auch Testprozesse unterlaufen, um die Funktionalität sicherzustellen.

Blaupause

Ein weiterer Schwerpunkt ist die Blaupausenartige Umsetzung der Software Architektur für die moderne Infrastruktur. Die Entwicklung der Anwendung muss so gestaltet werden, dass zukünftige Projekte auf diese aufbauen können. Ansätze bei der Entwicklung müssen austauschbar sein und in Teilschritte zerlegt werden.

4.2 Grobkonzeption

In diesem Abschnitt wird die grundlegende Idee anhand des Fachkonzepts formuliert. Danach folgen die groben Teilschritte zur Realisierung der Testanwendung.

4.2.1 Grundlegende Idee

Die prototypische Anwendung wird containerisiert und über eine Container-Registry verfügbar sein. Weiterhin muss die Anwendung unter der Berücksichtigung der Aspekte einer Microservice-Architektur, wie in Abschnitt 2.3 beschrieben, konzipiert und entwickelt werden. Die einzelnen Dienste der Anwendung müssen auf einem Kubernetes-fähigen Cluster ausgeliefert und in Betrieb genommen werden. Bevor die Anwendung verwendet wird, muss ein fester Testprozess die Funktionalität gewährleisten.

Für die mögliche Auslieferung bei einem Kunden der Krones AG soll die Nutzung bereits vorhandener Hardware mit Grafikkarten möglich sein. Es ist vorgesehen, dass die Anwendung in einem hybriden Cloud-Szenario die vordefinierte Hardware nutzen kann. Folglich soll die Verwendung der Hardware zu einer verbesserten Leistungsauswertung von Modellen im Bereich der künstlichen Intelligenz führen. Arbeitslasten, wie dem Auswerten von Deep-Learning-Modellen wie im Falle der Linatronic AI [2], sollen beispielhaft dargestellt werden. Dafür muss die Kommunikation von Diensten in Echtzeit stattfinden, um Informationen am Zielort schnell zu verarbeiten und eine nahtlose Verarbeitung großer Daten zu ermöglichen.

Anwendungsszenario

Der Schwerpunkt der zu entwickelnden Anwendung soll ein Dashboard mit Authentifizierungsmechanismus sein. Dieser soll Benutzern ermöglichen sich mit ihrem Passwort in ihr Profil einzuloggen. Dabei besteht auch die Möglichkeit eine Zwei-Faktor-Authentifizierung zu aktivieren und sich per Gesichtserkennung einzuloggen. Die Daten sollen persistent gespeichert werden und können bei erneutem Aufruf der Website wieder verwendet werden.

Blaupause

Die Anwendung soll nachvollziehbar entwickelt werden und als Fundament für spätere Entwicklungen dienen. Softwarekomponenten der Anwendung müssen austauschbar sein und mit unterschiedlichen Technologien ersetzt werden können. Schnittstellen müssen für gängige Kommunikationsprotokolle entwickelt werden.

4.2.2 Infrastruktur

Auf der Grundlage von Abschnitt 4.2.1 werden die Grobentwürfe erstellt. Die Konzeptentwürfe gliedern sich in zwei Teile, der Infrastruktur und der Anwendung.

Die Abbildung 4.1 stellt das Zielsystem dar.

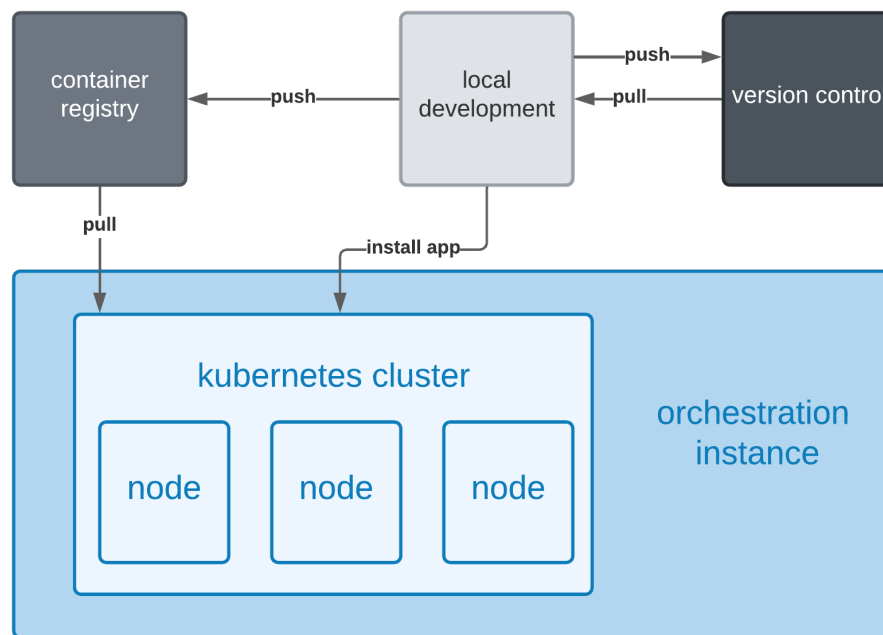


Abbildung 4.1: Grobentwurf der Infrastruktur

Lokale Entwicklungsumgebung: Die Entwicklung der Anwendung verläuft lokal und wird durch ein Versionsverwaltungssystem verwaltet. Ein Befehl an das Kubernetes-Cluster initialisiert die Auslieferung und Bereitstellung der einzelnen Dienste.

Image-Registry: Für die Auslieferung und Bereitstellung von Containern wird ein Image-Registry verwendet. Dienste erhalten separate Images und können unabhängig abgerufen werden.

Kubernetes-Cluster: Das Kubernetes-Cluster wird von einer Orchestrierungsinstanz verwaltet. Das Abrufen der Dienste erfolgt über ein Image-Registry.

4.2.3 Anwendung

Die Abbildung 4.2 stellt das Anwendungsszenario aus dem Unterabschnitt 4.2.1 dar. Die Anwendung aus dem Szenario wird in drei Dienste aufgeteilt.

Frontend-Service: Das Dashboard wird über den Frontend-Service bereitgestellt. Darüber kann ein Benutzer die Funktionalitäten anderer Dienste nutzen.

Authentication-Service: Die Anmeldung und Registrierung in ein Nutzerkonto erfolgt

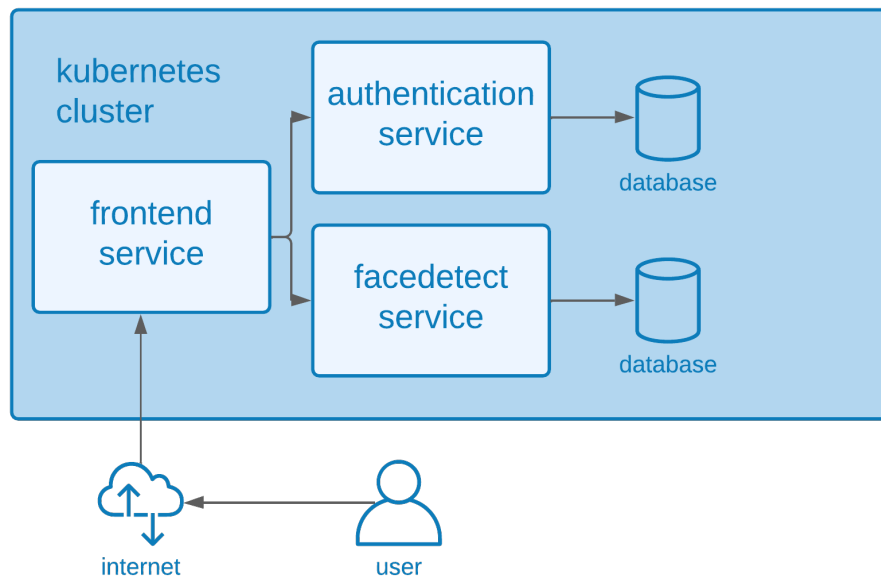


Abbildung 4.2: Grobentwurf der Anwendung

über den Authentication-Service. Dieser ermöglicht die persistente Speicherung der Nutzerdaten.

Facedetection-Service: Der Facedetection-Service bietet eine Anmeldung mithilfe von Gesichtserkennung an. Die relevanten Daten zur Gesichtserkennung werden in einer Datenbank persistent gespeichert.

4.2.4 Entwicklungsprozess

Die Entwicklung der Anwendung wird in vier auf sich aufbauende Schichten eingeteilt (vgl. Abbildung 4.3).

Anwendungsentwicklung: Ein zentrales Repository beinhaltet Verzeichnisse für die einzelnen Dienste. Diese werden lokal entwickelt, getestet und ausgeführt.

Containervirtualisierung: Das entwickelte Programm wird dann containerisiert und weiterhin lokal ausgeführt. Es wird getestet, ob die Containerisierung erfolgreich war und eine Kommunikation untereinander möglich ist. Schließlich wird das Image auf ein öffentliches Registry hochgeladen. Jeder Dienst hat dabei einen eigenen Speicherort in Form eines Images.

Kubernetes-Deployment: Das zentrale Repository beinhaltet ein weiteres Verzeichnis für die Kubernetes-Ressourcenobjekte in Form von YAML-Dateien. Bei Zugang der Entwicklungsumgebung zu einem Kubernetes-Cluster können diese Dateien ausgeführt werden.

Kubernetes-Cluster: Das Testcluster wird aufgesetzt, installiert und in eine Orchestrierungsinstanz integriert. Skripte für die Vorkonfiguration und Installation des Kubernetes-Clusters, erhalten ein eigenes Verzeichnis im zentralen Repository. Die

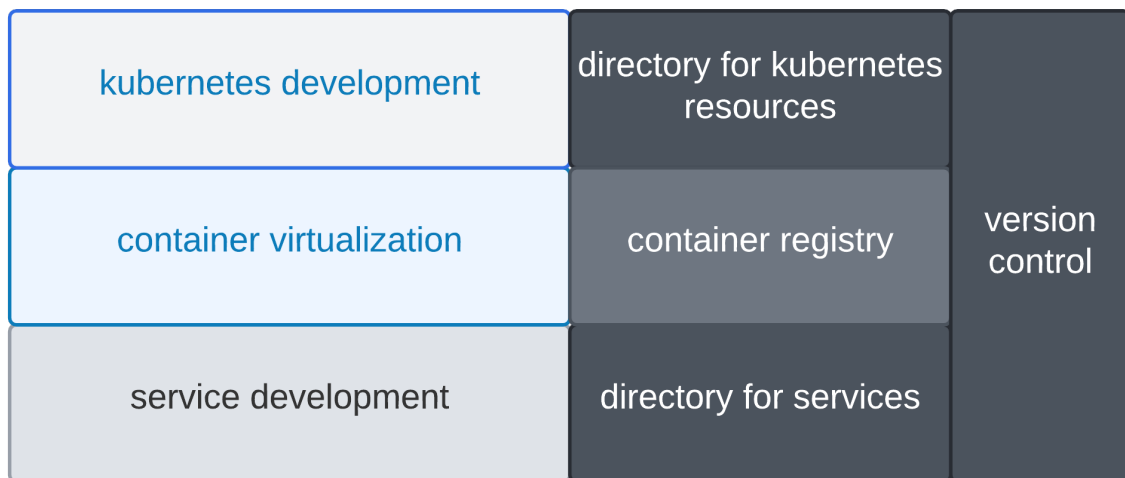


Abbildung 4.3: Vorgehen des Entwicklungsprozesses in Schichten

Auslieferung der Dienste erfolgt über das Image-Registry und werden von dem Kubernetes-Cluster heruntergeladen.

Kapitel 5

Lösungskonzept

Im Fokus des vierten Kapitels steht die Konzeption und Architektur der Microservice-Anwendung.

5.1 Design Entscheidungen

Der folgende Abschnitt behandelt die gewählten Technologien für das Lösungskonzept der Microservice-Architektur.

5.1.1 Backend

Flask

Für die Entwicklung der Webanwendung in Python wird das Microframework Flask verwendet. Dieses beinhaltet nur die wesentlichsten Funktionalitäten für die Webentwicklung. Dafür bietet das Framework eine hohe Flexibilität, da die nötigen Bibliotheken vom Entwickler gewählt werden können [49] und vereinfacht die Erstellung von APIs [50, S.11] durch Blueprints.

Blueprints sind ein Konzept von Flask, welche die Aufteilung von Komponenten einer Webanwendung ermöglichen. Diese Komponenten können in Form von Routen unterschiedliche Endpunkte mit einer view ausgeben. Bei Aufrufen eines Endpunkts wird die dazugehörige view ausgegeben [49]. Dadurch kann man die Funktionalität der Webanwendung nach Endpunkten strukturieren und ist ideal für die Ausführung einer Microservice-Architektur.

Weiterhin ermöglichen die Flask-Abhängigkeiten wie Template-Engine Jinja und dem WSGI Werkzeugkasten, das Rendern von HTML-Templates mit Daten aus der Flask-Anwendung und dem Bereitstellen einer standardisierten Schnittstelle Web Server Gateway Interface (WSGI), welche die Verwendung der meisten Webserver ermöglicht [49].

Gunicorn

Gunicorn ist ein WSGI HTTP server für Unix, welcher mit den meisten Webframeworks kompatibel ist. Das Gunicorn-Modell teilt einen Master-Prozess in mehrere Worker-Prozesse auf. Der Master-Prozess ist lediglich eine Schleife für die bestehenden Worker-Prozesse und ist bei Ausfall für den Neustart zuständig. Die Worker-Prozesse sind für die Verarbeitung von eingehenden Anfragen zuständig. Diese teilen sich in folgende Worker-Class ein. Sync-Workers bearbeiten Anfragen jeweils Einzeln und unterstützen keine persistente Verbindung. Async-Workers sind basierend auf Greenlets und unterstützen mithilfe von Gevent asynchrone Koroutinen [51].

OpenCV

OpenCV ist eine Open-Source-Computer-Vision¹ Bibliothek, die zur Verarbeitung von Bildern verwendet wird [52]. Da ein Video nur eine Serie von Bildern ist, können die Techniken der Bildverarbeitung auch hier genutzt werden [53]. Die Bibliothek beinhaltet eine Vielzahl an Algorithmen mit Bezug zu Computer-Vision oder Machine-Learning. Diese unterstützen auch die Verwendung von Graphics Processing Units (GPUs) die auf den Programmierschnittstellen Compute Unified Device Architecture (CUDA) oder Open Computing Language (OpenCL) basieren [54]. Die Anwendung wird mit der Python-Version der Bibliothek entwickelt, um die Implementierung in die Python-Webanwendung zu vereinfachen.

5.1.2 Frontend

Hypertext Markup Language (HTML) und JavaScript

Die Umsetzung der Benutzeroberfläche erfolgt mit der Auszeichnungssprache HTML5 in Kombination mit der Skriptsprache JavaScript, um die Interaktion des Anwenders zu realisieren. Für die erleichterte Gestaltung der Website wird das Frontend-CSS-Framework Bootstrap Version 5.0 genutzt.

5.1.3 Kommunikation

SocketIO

Die bidirektionale und ereignisbasierte Echtzeitkommunikation zwischen den Diensten wird mithilfe der SocketIO Bibliothek realisiert. Die Bibliothek unterstützt mehrere Programmiersprachen für Server und Client Implementierungen, welche von der Community gewartet werden. Eine Kommunikation zwischen Server und Client erfolgt über WebSockets, wenn dies nicht möglich ist, wird auf die Ressourcen intensivere [55] Alternative HTTP-long-polling zurückgegriffen [56]. Für die Kommunikation der Webanwendung wird die Server Implementierung von Python-Socketio genutzt [57]. Die Implementierung der Anwenderlogik erfolgt über die JavaScript SocketIO Bibliothek, diese sind Plattformunabhängig.

¹Computer-Vision bezeichnet die Transformation von visuellen Daten in eine abgewandelte Form, die zur Beantwortung einer Fragestellung dienen kann.

Representational State Transfer (REST)

REST ist eine Ressourcen basierende Architektur für verteilte Systeme. Diese Ressourcen werden über eine einheitliche Schnittstelle basierend auf HTTP Methoden zugänglich. Dabei ist jede Ressource über eine URL erreichbar. REST erlaubt dabei Ressourcen in verschiedene Datentypen zu repräsentieren, wie text, xml, json, etc. Die Kommunikation über die HTTP-API funktioniert wie die über create, read, update und delete (CRUD) parallel mit den HTTP-Methoden GET, POST, PUT und DELETE [58].

5.1.4 Datenbank

MongoDB

MongoDB ist eine Dokumentenorientierte Datenbank bei den Daten nicht in einer Tabelle, sondern in Dokumenten gespeichert wird und zählt damit zu den NoSQL-Datenbanken. Dies ermöglicht die Darstellung von komplexen hierarchischen Beziehungen mit einem einzigen Eintrag. Dokumente sind nach einer Key-Value-Struktur aufgebaut und besitzen kein vorgeschriebenes Schema zur Erstellung von Einträgen.

5.1.5 Versionsverwaltungssystem

GitHub

Das verwendete Versionsverwaltungssystem für die Entwicklung der Microservices ist GitHub. Dieses basiert auf git und fokussiert sich auf Open-Source-Software und bietet gleichzeitig Enterprise Support für Unternehmen [59]. Die Kronos AG hat die Möglichkeit dies für zukünftige Entwicklungsprozesse von Microservices zu verwenden.

DockerHub

Wie in Abschnitt 2.1 beschrieben ist das Standard Registry für Docker-Images Docker-Hub. Deshalb werden für die Bereitstellung der Docker-Images die kostenfreien und öffentlichen Repositories verwendet. Die Entwicklung der Dienste erfolgt durch getrennte Repositories.

5.2 Entwicklung

In dem folgenden Abschnitt werden die Entwicklungsschritte der Microservice-Anwendungen und der Ressourcenobjekte für das Bereitstellen mit Helm näher erläutert.

5.2.1 Microservice-Entwicklung

Die Abbildung 5.1 zeigt den Arbeitsablauf der Service-Entwicklung auf. Zuerst wird die Funktionalität des Dienstes realisiert und dann mithilfe eines Dockerfiles ein Docker-Image erstellt. Dafür wird ein Base-Image entweder aus einem Docker-Registry,

wie DockerHub oder aus dem lokalen registry benötigt. Um den Container zu starten, kann ein Docker-Befehl auf dem Hostsystem ausgeführt werden. Die Nutzung von Tools wie Docker-Compose erlauben das Starten mehrerer Container mithilfe von Konfigurationsdateien in Form von YAML-Dateien. Nachdem ausgeführt der Container können diese getestet werden. Abschließend kann der Entwicklungsablauf fortgeführt werden oder ein Release für das Versionsverwaltungssystem und dem Container-Repository erstellt werden.

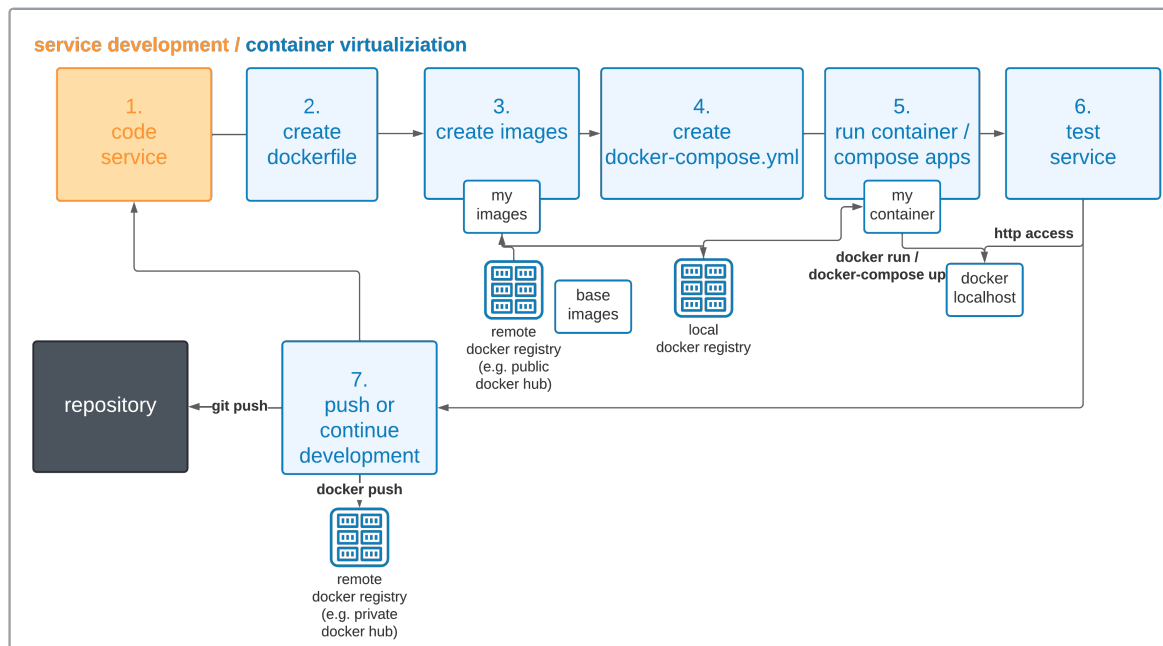


Abbildung 5.1: Microservice-Entwicklung in Anlehnung an [60]

5.2.2 Helm-Chart-Entwicklung

Die Abbildung 5.2 beschreibt den Vorgang bei der Entwicklung von Helm-Charts für Kubernetes. Als Erstes wird ein Helm-Chart entwickelt. Der Kommandozeilenbefehl `helm lint` überprüft den vorgegebenen Pfad zum Chart und führt eine Serie von Tests zur Validierung durch. Danach kann dieser auf einem Kubernetes-Cluster installiert werden, wenn die Kubernetes-Ressourcenobjekte einen Docker-Container benötigen, wird das spezifizierte Image aus dem öffentlichen DockerHub Registry heruntergeladen. Der Service kann jetzt mit dem Kommandozeilentool Kubectl getestet werden. Zuletzt wird die Entwicklung am Helm-Chart fortgeführt oder ins Versionsverwaltungssystem hochgeladen.

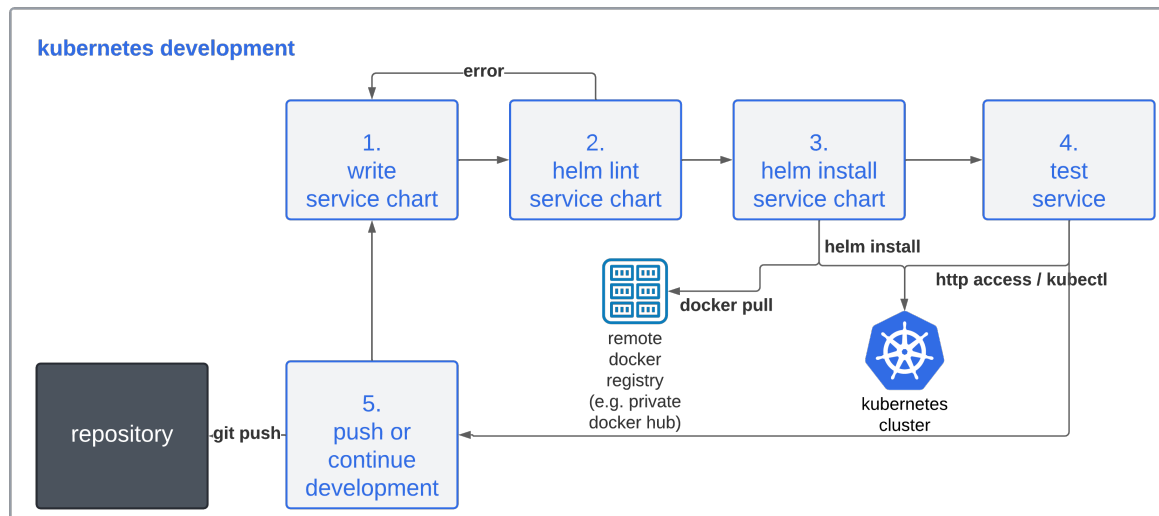


Abbildung 5.2: Kubernetes-Entwicklung in Anlehnung an [60]

5.3 Architektur

Dieser Abschnitt befasst sich mit der Architektur der zu entwickelnden Anwendung. Die Aufgaben der einzelnen Dienste der Microservice-Architektur wird konzipiert und dargestellt. Danach folgt der Vorgang der Installation der losen Dienste mit Helm auf dem Kubernetes-Cluster.

5.3.1 Microservices

Die Abbildung 5.3 zeigt die einzelnen Softwarekomponenten der Webanwendung, welche als Docker-Container laufen. Das Frontend dient als visuelles-Gateway für die anderen Dienste. Dieses bietet vier Endpunkte die für Nutzer über einen Webbrowser erreichbar sind. Der Home-Endpunkt ermöglicht den Login oder Logout eines Nutzers über die REST-API des Authentication-Dienstes. Register erlaubt die Registrierung eines Nutzers in der Datenbank. Train und Facelogin senden Bilder an den Facerecognition-Dienst, dies geschieht mit dem Kommunikationsprotokoll SocketIO. Damit wird das Modell zur Gesichtserkennung trainiert und ermöglicht die spätere Zwei-Faktor-Authentifizierung mittels Login per Gesichtserkennung.

5.3.2 Helm-Installation

Die einzelnen Dienste der Webanwendung werden mithilfe von einem Helm-Chart gleichzeitig auf ein Kubernetes-Cluster installiert. Dabei hat jeder Dienst ein eigenes Verzeichnis mit den notwendigen Kubernetes-Ressourcenobjekten. Der Zugang erfolgt über eine Kubeconfig die den Zugang zum Kubernetes-Cluster ermöglicht. Bei erfolgreichem Zugang kann mit einem Befehl im Verzeichnis die Microservices installiert werden. Das Kubernetes-Cluster bezieht dann die benötigten Docker-Images aus den angegebenen Docker-Repositories. Die erfolgreiche Bereitstellung der Container auf

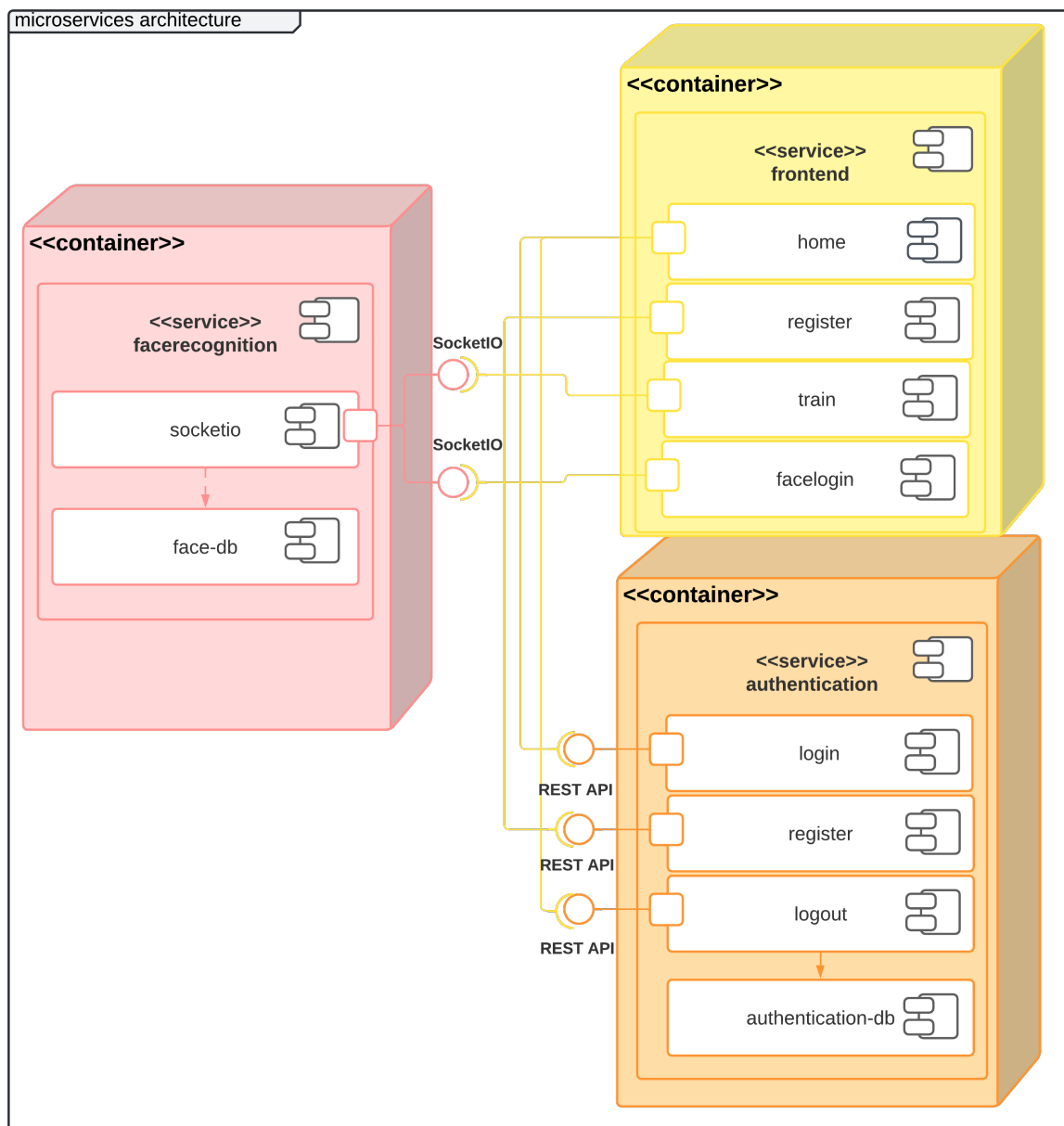


Abbildung 5.3: Lokale Microservice Entwicklung

dem Cluster ist dann unabhängig von der Helm-Installation, wenn die Images nicht von den angegebenen Repositories bezogen werden können. Die Bereitstellung und Auslieferung der Kubernetes-Ressourcenobjekte ist trotzdem Erfolgreich und gibt auf dem Kubernetes-Cluster lediglich Fehlermeldungen an, bei der versuchten Ausführung der Container in einem Pod an.

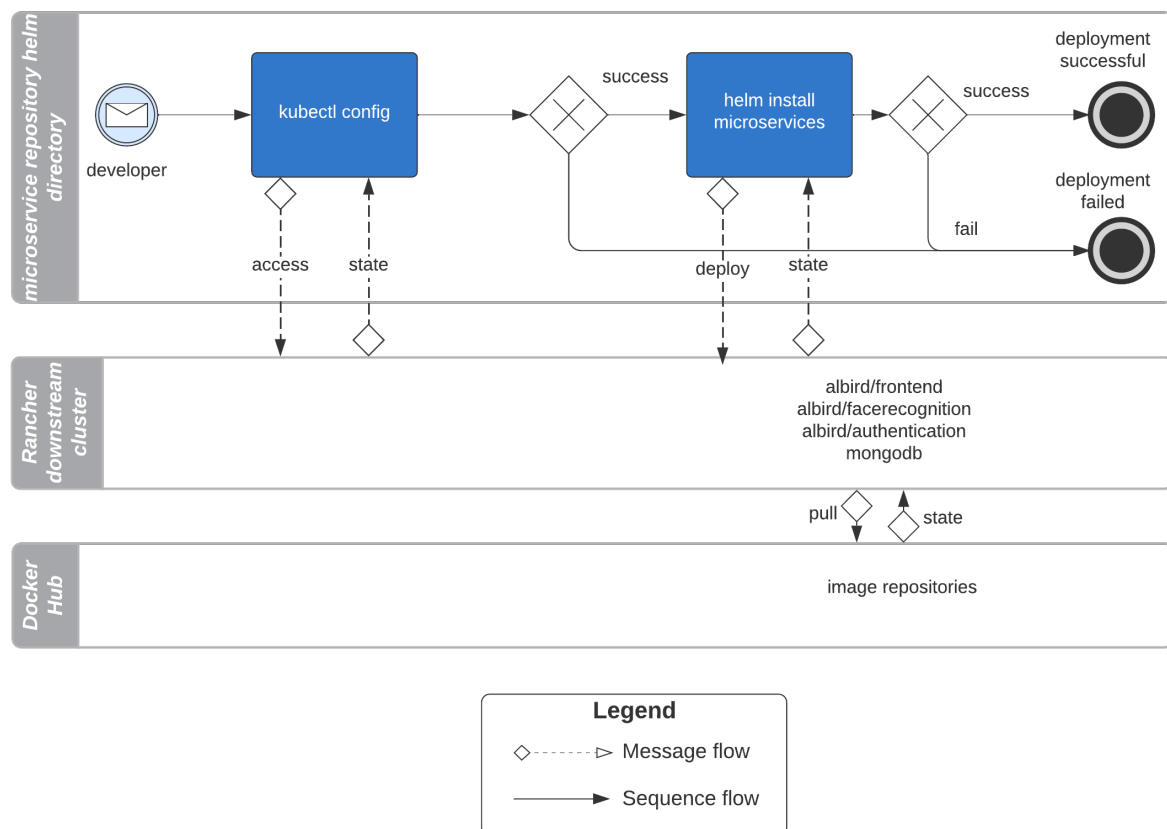


Abbildung 5.4: BPMN Modell - Helm Installation der Microservices

Kapitel 6

Umsetzung des Lösungskonzepts

Das folgende Kapitel beschreibt die Vorgehensweisen der Umsetzung des Konzepts. Diese sind in mehrere Teile gegliedert. Erstens mit der Konfiguration und Einrichtung der Knotenpunkte für das Kubernetes-Cluster. Zweitens mit der ausführenden Entwicklung Anwendung im Microservice-Architektur-Stil KubeVision. Drittens mit der Beschreibung für die implementierte Gesichtserkennung des Facerecognition-Services. Viertens der Dockerisierung der einzelnen Dienste. Fünftens die Ausführung der Helm-Charts für die Auslieferung und Bereitstellung von Kubernetes-Ressourcenobjekte.

6.1 Konfiguration und Einrichtung

In diesem Abschnitt geht es um die Einrichtung der Kubernetes Infrastruktur. Zuerst die Einrichtung der einzelnen virtuellen privaten Server in Vultr, die als Knotenpunkte in unserem Kubernetes-Cluster funktionieren. Zunächst wird eine Domain für den späteren Einsatz der Microservices konfiguriert. Abschließend erfolgt die Bereitstellung von Zertifikaten für die Domain.

Virtueller privater Server

Durch die Einschränkungen, beschrieben in Abschnitt 7.2, werden für die Installation der Kubernetes Plattform virtuelle private Server (VPS) verwendet. Ein VPS ist eine virtuelle Maschine, die von Drittanbietern wie Internet-Hosting-Diensten, als Dienst verkauft wird. Dies ermöglicht das Mieten von Hardware. Die Server dienen als Knotenpunkte für die spätere Kubernetes Installation. Es werden insgesamt drei VPS-Instanzen gemietet auf denen das Betriebssystem SLE-Micro Enterprise 5.1 bereitgestellt und auf den Serverinstanzen installiert.

Domain

Der Zugang zur Webanwendung wird mithilfe einer öffentlichen Domain ermöglicht. Der DNS-Eintrag einer Domain ist für die Adressierung zuständig. Durch die Verände-

ung des A-Records leiten alle Anfragen der Domain auf eine IPv4-Adresse um [61]. Die IPv4 Adresse ist in diesem Fall der Cluster-Master des Kubernetes-Clusters.

6.1.1 SSL-Verschlüsselung

Der Frontend-Service benötigt für die Gesichtserkennung die Webcam eines Benutzers, jedoch ist dies nur in einem sicheren Kontext möglich. Die Kommunikation zwischen einem Client und Ingress muss TLS-Verschlüsselt sein, um JavaScript Methoden wie `MediaDevices.getUserMedia()` auszuführen. Dafür benötigt der Ingress-Controller ein Zertifikat und einen privaten Schlüssel. Dieser kann automatisch mit einem Kubernetes-Issuer erstellt werden und von einem Ingress referenziert werden [62].

Issuer: Das add-on Cert-Manager ist bereits vorinstalliert auf dem Kubernetes-Cluster und ermöglicht die Verwaltung von Zertifikaten. Dieser enthält die Kubernetes-Resource Issuer, welche zur Generierung von privaten Schlüsseln dient. Cert-Manager erlaubt die vereinfachte Bereitstellung von Automatic Certificate Management Environment (ACME) SSL-Zertifikaten für Ingress-Objekte in Kubernetes. Die ACME-Zertifikate sind frei verfügbar und werden von den meisten Webbrowsern als Glaubwürdig eingestuft. Die Verifizierung des Zertifikats erfolgt über eine ACME-Challenge, welche mit einer HTTP-Anfrage validiert werden kann. Dafür wird ein berechneter Schlüssel auf dem Endpunkt der vorgegebenen Domain platziert und von einem öffentlichen ACME-Server abgerufen und bestätigt [63]. Die Grundvoraussetzung dafür war die Änderung des A-Records auf die IPv4-Adresse des Cluster-Masters in Abschnitt 6.1.

```
1  apiVersion: cert-manager.io/v1
2  kind: Issuer
3  metadata:
4    name: letsencrypt-prod
5  spec:
6    acme:
7      server: https://acme-v02.api.letsencrypt.org/directory
8      privateKeySecretRef:
9        name: letsencrypt-key
10     solvers:
11       - http01:
12         ingress:
13           class: nginx
```

Quellcode 6.1: issuer.yaml [63]

Die Ausführung des Issuers erzeugt einen privaten Schlüssel mit der Bezeichnung `letsencrypt-key` und dem Kubernetes-Issuer namens `letsencrypt-prod`.

Cert: Der nächste Schritt ist die Erzeugung eines Zertifikats mit dem vorher erstellten Issuer.

```
1  apiVersion: cert-manager.io/v1
```

```
2  kind: Certificate
3  metadata:
4    name: cert-prod
5  spec:
6    secretName: deploy-secret
7    issuerRef:
8      name: letsencrypt-prod
9    dnsNames:
10     - "example-domain.com"
```

Quellcode 6.2: cert.yaml [63]

Die Ausführung des Kubernetes-Cert erstellt ein signiertes Zertifikat. Dafür ist die Domain mit dem Eintrag des Cluster-Masters notwendig und die Bezeichnung des Issuers. Das erzeugte Secret mit der Bezeichnung deploy-secret kann von einem Ingress zur Verschlüsselung der Kommunikation verwendet werden.

6.1.2 Node-Affinity

Für den Einsatz unterschiedlicher Hardwareressourcen in einem hybriden Kubernetes-Cluster muss eine Kennzeichnung der Nodes erfolgen. Node-Affinity ermöglicht die Benutzung von Labels zur Zuweisung von spezifischen Werten. Bei einer Auslieferung von Kubernetes-Anwendungen lassen sich diese dann auf bestimmte Nodes mit dem vorkonfigurierten Label bereitstellen [64]. In einem hybriden Kubernetes-Cluster kann somit die Unterteilung von Labels in Cloud und On-Premise Hardware erfolgen (vgl. Quellcode 6.3).

```
1  kubectl label nodes microservice0 hardware=cloud
2  kubectl label nodes microservice1 hardware=cloud
3  kubectl label nodes microservice2 hardware=premise
```

Quellcode 6.3: Node-Labels

Die richtige Zuweisung von Pods auf gekennzeichneten Nodes erfolgt mit einem NodeSelector. Diesem können Schlüsselwerte wie die Kennzeichnung der Nodes übergeben werden, um Pods die Bereitstellung zu ermöglichen.

6.1.3 Taints and Tolerations

Taints und Tolerations stellt sicher, dass Pods nicht auf einen ungeeigneten Knoten eingeplant werden. Ein Taint dient zur Markierung von Nodes, demnach akzeptieren diese nur Pods mit der richtigen Tolerations. Für die erforderliche Nutzung von GPU-Nodes einer Anwendung können auch diese gekennzeichnet werden [65].

```
1  kubectl taint nodes microservice2 hardware=gpu:NoSchedule
```

Quellcode 6.4: Node-Taints

Damit werden nur Pods auf der Node *microservice2* eingeplant die als Tolerations den Schlüssel *Hardware*, Wert *gpu* und dem Effekt *NoSchedule* (vgl. Quellcode 6.4). Bereits auf der Node laufende Pods sind davon nicht betroffen, dies erfordert den Taint *NoExecute* [65].

Die Ausführung von Node-Affinity und Taints und Tolerations ermöglicht nun die absolute Ausführung von Pods auf spezifischer Hardware. Durch die Markierung mit Taint werden keine Pods ohne die Schlüsselwerte einer GPU eingeplant. Und die Bereitstellung von Pods lässt sich zielgerichtet auf die Nodes mit den spezifizierten Labels bestimmen.

6.2 KubeVision

Dieser Abschnitt behandelt die einzelnen Softwarekomponenten der Microservice-Anwendung KubeVision. Die Webanwendung ist in drei verschiedene Dienste unterteilt. Erstens der Benutzeroberfläche für die Interaktion mit dem Benutzer. Zweitens dem Authentifizierungsdienst, der für die Registrierung und Anmeldung zuständig ist. Drittens der Gesichtserkennungsdienst, welcher eine Zwei-Faktor-Authentifizierung per Gesichtserkennung ermöglicht.

6.2.1 Frontend-Service

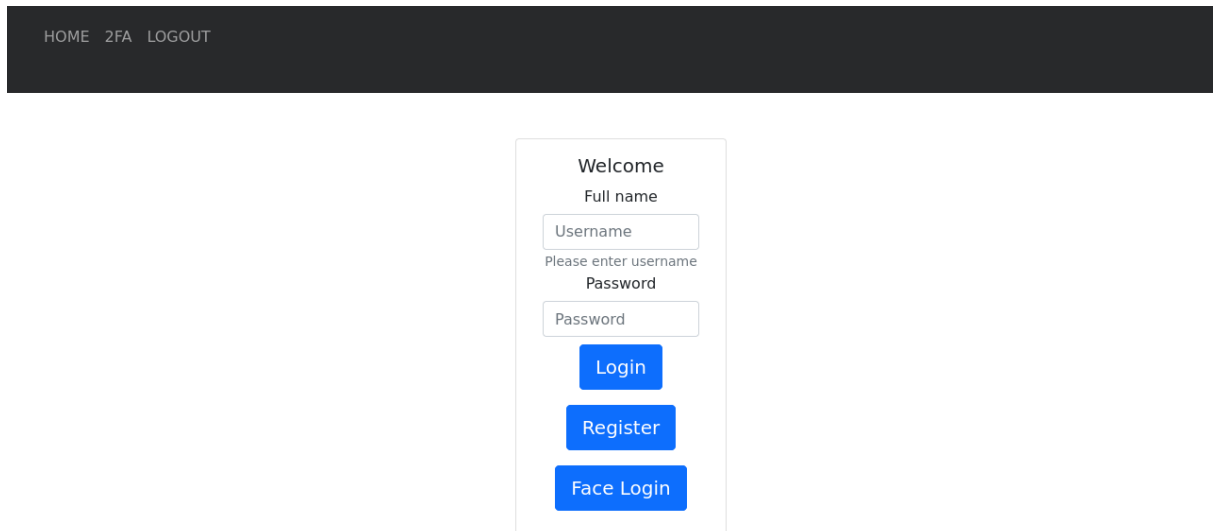
Frontend-Service ist die Benutzeroberfläche zur Interaktion mit dem Benutzer. Der Dienst ist in mehrere Blueprints mit eigenen Endpunkten aufgebaut. Jeder dieser Endpunkte gibt einen URL-Pfad für die Interaktion mit dem Frontend-Service oder einem anderen Dienst an. Bei Aufruf eines Endpunkts wird eine view aufgerufen und mithilfe der Template Engine Jinja2 eine spezifische HTML-Datei aus dem templates-Verzeichnis ausgegeben. Diese spezifische Datei ist ein HTML Code-Block und wird in die Main-View gesetzt.

Es gibt zwei Blueprints einer im Verzeichnis *home*, welcher die Funktionalitäten und Endpunkte für das einloggen, registrieren und anzeigen des Profils ausgibt. Für die Authentifizierungsmöglichkeiten wird auf die Authentication-Service-Endpunkte umgeleitet. Das zweite Blueprint gibt views zum Interagieren mit dem Facerecognition-Service an. Dieser beinhaltet Logik in Form von JavaScript mit der eingebundenen Bibliothek *SocketIO* und ermöglicht das Senden von Bildern mithilfe einer Webcam.

Für die Kommunikation mit dem Facerecognition-Service wird die Kamera des Benutzers benötigt. Das Modul und die enthaltene Klasse *Camera.js* ist für die Verwendung der Webcam zuständig. Die Funktion *navigator.mediaDevices.enumerateDevices()* listet alle angeschlossenen Peripheriegeräte mit Kamerafunktion auf. Diese Geräte werden dann in einer Schleife in eine Dropdown-Liste platziert. Der Nutzer kann danach eine spezifische Kamera auswählen.

Mit der Klasse *socketio.js* lässt sich die bidirektionale Kommunikation mit dem Facerecognition-Service aufbauen. Es gibt drei unterschiedliche Events für die Kommunikation mit dem Dienst. *Stream* sendet eine bestimmte Anzahl an Bildern an den

Dienst und löst im Anschluss das Event traindata aus. Dieses Event wird über den Endpunkt train ermöglicht. Der Zweite Endpunkt facelogin ermöglicht die Kommunikation über das Event Predict. Dieser sendet eine bestimmte Anzahl an Bildern an den Dienst und ermöglicht den Login des Nutzers.



The image shows a dark grey header bar with the links 'HOME', '2FA', and 'LOGOUT' in white. Below the header is a white login form. The form contains the following elements: a 'Welcome' heading, a 'Full name' label, a 'Username' input field with a placeholder 'Please enter username', a 'Password' label, a 'Password' input field, and three blue buttons labeled 'Login', 'Register', and 'Face Login' stacked vertically.

Abbildung 6.1: KubeVision Frontend Dashboard

6.2.2 Authentication-Service

Der Authentication-Service ist für die Authentifizierung des Benutzers über das Frontend zuständig. Dieser Dienst wird mit einer Datenbank bereitgestellt in der Benutzerinformationen gespeichert werden. Über ein Blueprint wird eine API mit dem Endpunkt auth bereitgestellt. Damit kann sich ein Benutzer einen Benutzeraccount erstellen oder sich einloggen. Der Dienst erstellt bei Registrierung einen Eintrag in die MongoDB-Datenbank oder lest diese aus. Bei erfolgreichem Login wird ein Cookie mit dem Benutzernamen gesetzt und weitergeleitet in das Home-Hub. Hier kann der Benutzer die Zwei-Faktor-Authentifizierung per Gesichtserkennung aktivieren und mit dem Facerecognition-Service kommunizieren. Falls das Passwort falsch ist oder der Name bereits in der Datenbank eingetragen ist, wird der Benutzer mit einer Fehlermeldung benachrichtigt. Bei der Abmeldung des Benutzers wird der Benutzer-Cookie gelöscht und wieder auf die Home-Page weitergeleitet.

6.2.3 Facerecognition-Service

Der Facerecognition-Service ermöglicht die Anmeldung eines Benutzers per Gesichtserkennung. Grundvoraussetzung ist die Registrierung des Nutzers beim Authentication-Service. Der Endpunkt wird Komponentenbasiert über eine Blueprint integriert. Über den Endpunkt socketio ist die eventbasierte Kommunikation zwischen Frontend-Service Benutzer und Facerecognition-Service möglich. Das Event stream, nimmt

Bilder in Form von .webp an und speichert diese in einem Verzeichnis. Bei der Kommunikation wird nach jeder Anfrage ein Status zurückgeschickt. Das Event traindata, erstellt ein Klassenobjekt und führt die Funktion train() aus. Diese geht durch das Bilderverzeichnis und erstellt ein Gesichtsdatenmodell zur späteren Validierung. Das Event predict, nimmt wie das Event stream, Bilder an, aber vergleicht diese mit dem vorher trainierten Modell für die Gesichtserkennung. Bei erfolgreicher Übereinstimmung wird die Datenbank von Facerecognition-Service nach dem vorhandenen Benutzer überprüft. Der eigentliche Entwurf sollte den Facerecognition-Service mit einer eigenen Datenbank ausliefern. Aus Zeitgründen in der Entwicklung wurde dieser Teil verworfen und es wird die Datenbank eines anderen Dienstes genutzt.

6.3 Gesichtserkennung

6.3.1 Alignment

6.3.2 Training

6.3.3 Model

6.4 Dockerisierung

Der nächste Schritt ist die Dockerisierung der losen Dienste. Die Dienste liegen in einem eigenen Verzeichnis im Softwareprojekt KubeVision.

6.4.1 Dockerfile

Jeder Dienst verfügt über ein eigenes Dockerfile mit Anweisungen zum Erstellen eines Docker-Images. Das Dockerfile befindet sich im selben Verzeichnis wie die Code-Dateien der Dienste.

```
1 FROM python:3.7.2-stretch
2
3 WORKDIR /app
4 ADD . /app
5
6 RUN apt-get update
7 RUN apt-get install ffmpeg libsm6 libxext6 -y
8 RUN pip install --upgrade pip setuptools wheel
9 RUN pip install -r requirements.txt
10
11 ENV PYTHONUNBUFFERED 1
12 EXPOSE 5000
13
14 CMD ["gunicorn" , "-k"
    , "geventwebsocket.gunicorn.workers.GeventWebSocketWorker", "-w", "3"]
```



```
, "--bind" , ":5000" , "run:app"]
```

15

Quellcode 6.5: Dockerfile

Um Redundanz zu vermeiden wird im Folgenden das Dockerfile zum Facerecognition-Service näher erläutert (vgl. Quellcode 6.5). Dieser ist ähnlich aufgebaut wie die Dockerfiles der anderen Dienste. Die Basis des Docker-Images ist ein Python-Stretch-Image, welches auf dem leichtgewichtigen Betriebssystem Debian-Stretch aufbaut. Zunächst werden die nötigen Abhängigkeiten zur Ausführung von OpenCV installiert. Danach wird mit pip die notwendigen Pythonbibliotheken installiert. In der requirements.txt stehen alle Bibliothekennamen mit der erforderlichen Version. Die Environmental-Variable ermöglicht die Ausgabe des Python-Buffers im Terminal. Die CMD Anweisung des Containers startet immer mit dem Befehl einen Gunicorn-Webserver auszuführen. Die zusätzlichen Flags geben die Art und Anzahl der Worker-Prozesse an. Letztendlich wird die Webanwendung mit der WSGI-Schnittstelle an den gewählten Port 5000 ausgeführt.

6.4.2 Docker-Compose

Der Build-Vorgang und anschließende Ausführungsprozess mehrerer Dockerfiles kann mit dem Tool Docker-Compose vereinfacht werden. Ähnlich wie bei Kubernetes-Ressourcenobjekte werden die Konfigurationen und Installationsansweisungen in einer YAML-Datei gespeichert. Die Datei zur Ausführung von Docker-Compose liegt im Root-Verzeichnis des Projekts.

```
1 authentication:
2   build: ./authentication
3   image: albird/authentication:latest
4   environment:
5     homeEndpoint: http://localhost:8000/
6     trainEndpoint: http://localhost:8000/train
7     mongoEndpoint: mongodb://admin:password@localhost:27017/
8   ports:
9     - "5001:5001"
10  command: gunicorn -w 1 -b 0.0.0.0:5001 run:app
11
12 mongo:
13   image: mongo:4.1.7
14   environment:
15     MONGO_INITDB_ROOT_USERNAME: admin
16     MONGO_INITDB_ROOT_PASSWORD: password
17   ports:
18     - "27017:27017"
19   volumes:
20     - ./mongo-volume:/data/db
```

Quellcode 6.6: Ausschnitt aus dem docker-compose.yaml

Der Befehl `build` gibt die Docker-Anweisung zum Bauen eines Images anhand einer vorhandenen Dockerfile-Datei. Die Webanwendung wurde mit dem Einsatz von Environmental-Variablen entwickelt. Diese können für eine flexible Bereitstellung der Dienste angewendet werden, um Endpunkte in Form von String-Variablen in der Webanwendung zu ändern. `ports` gibt die Ports an, auf die der erstellte Container im Netzwerk lauscht. Die lokale Anwendung kann so im eigenen Hostnetzwerk erreicht werden. Im Kubernetes-Cluster kann der Pod durch einen Service selektiert und mit einem Ingress verbunden werden. Bei der MongoDB-Datenbank werden die Environmental-Variablen zur Übergabe von Passwort und Adminnamen genutzt. Der `Volume` Befehl erstellt ein persistentes Verzeichnis für die Speicherung der Datenbank.

6.5 Helm-Chart

Dieser Abschnitt beschreibt die Entwicklung der Kubernetes-Ressourcenobjekte für die Bereitstellung mit dem Package-Manager Helm. Um Redundanz zu vermeiden wird die Umsetzung der Kubernetes-Ressourcenobjekte beispielhaft am Dienst `Facerecognition` gezeigt. Helm-Charts verfügen über eine YAML-Datei namens `Values`, welche globale Parameter für das Helm-Chart definiert. Dadurch können die Kubernetes-Ressourcenobjekte von einer Datei aus vorkonfiguriert werden.

6.5.1 Service

Dienste werden mit einem eigenen Kubernetes-Service bereitgestellt, um die Kommunikation zwischen Diensten und Kubernetes-Cluster zu realisieren. Bis auf die individuellen Parameter des Helm-Charts, sind die Service-Konfiguration für alle Dienste identisch aufgebaut.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: {{ .Values.face.service.name }}
5    namespace: {{ .Values.namespace }}
6  spec:
7    type: {{ .Values.face.service.type }}
8    ports:
9      - port: {{ .Values.face.image.port }}
10        targetPort: {{ .Values.face.image.port }}
11        protocol: TCP
12        name: http
13  selector:
14    server: {{ .Values.face.name }}
```

Quellcode 6.7: `face-service.yaml`

Der Name des Kubernetes-Service ist für die spätere Angabe im Ingress notwendig. Type definiert den Service-Typen zur Kommunikation innerhalb des Clusters wird deshalb ClusterIP gewählt. Die Ports geben an, welcher Port im lokalen Netzwerk des Pods lauscht. TargetPort gibt dann den Port an der über den Service erreichbar ist. Schließlich wird der zugehörige Pod des Services mit dem Selector ausgewählt.

6.5.2 Ingress

Für die Implementierung der Webanwendung wird ein Nginx-Ingress verwendet. Dieser stellt den Endpunkt eines Services in Form einer URL dar. Als Nächstes wird wie in Abschnitt 2.2.5 beschrieben ein Nginx-Ingress vorkonfiguriert (vgl. Quellcode 6.8).

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: kubevision-ingress
5    annotations:
6      nginx.ingress.kubernetes.io/ssl-redirect: "true"
7      nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
8  spec:
9    tls:
10     - hosts:
11       - {{ .Values.envEndpoint.host }}
12       secretName: deploy-secret
13    rules:
14     - host: {{ .Values.envEndpoint.host }}
15       http:
16         paths:
17           - backend:
18             service:
19               name: {{ .Values.auth.service.name }}
20             port:
21               number: {{ .Values.auth.image.port }}
22           path: /auth
23           pathType: Prefix
24         - backend:
25             service:
26               name: {{ .Values.face.service.name }}
27             port:
28               number: {{ .Values.face.image.port }}
29           path: /socket.io
30           pathType: Prefix
31         - backend:
32             service:
33               name: {{ .Values.frontend.service.name }}
34             port:
35               number: 80
```

```

36     path: /
37     pathType: Prefix
38     ingressClassName: nginx

```

Quellcode 6.8: kubevision-ingress.yaml

Der Bereich annotations passt das Verhalten des Ingress an. Die Optionen zum redirect mit SSL erzwingt die Weiterleitung von HTTP zu einer HTTPS Verbindung mit dem Client. Spec bestimmt die TLS-Verbindung und die Regeln für die Endpunkte der Services über den Ingress. Als Hostname wird die Domain mit dem A-Record Eintrag auf dem Cluster-Master verwendet. In der TLS-Einstellung wird noch das TLS-Zertifikat als Secret referenziert. Jedem Service wird ein Endpunkt zugewiesen. Der Authentication-Service ist über den Prefix auth erreichbar. Facerecognition-Service erhält den Endpunkt socket.io zur Kommunikation mithilfe der gleichnamigen Bibliothek. Der Frontend-Service ist über den Hostnamen erreichbar. Für die Pfade wird kein Prefix wie bei dem Authentication-Service benötigt.

6.5.3 Deployment

Die grundlegende Bereitstellung der Dienste erfolgt mit einem Deployment.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ .Values.frontend.name }}
5    namespace: {{ .Values.namespace }}
6  spec:
7    replicas: {{ .Values.frontend.replicas }}
8    selector:
9      matchLabels:
10       server: {{ .Values.frontend.name }}
11  template:
12    metadata:
13      labels:
14       server: {{ .Values.frontend.name }}
15    spec:
16      containers:
17       - name: {{ .Values.frontend.name }}
18         image: {{ .Values.frontend.image.name }}:{{ .Values.frontend.image.tag }}
19         imagePullPolicy: Always
20         ports:
21          - containerPort: {{ .Values.frontend.image.port }}
22         env:
23          - name: loginEndpoint
24            value: https://{{ .Values.envEndpoint.host }}/auth/login
25          - name: registerEndpoint
26            value: https://{{ .Values.envEndpoint.host }}/auth/register
27          - name: websocketServer

```

```
28     value: https://{ .Values.envEndpoint.host }
29   - name: homeEndpoint
30     value: https://{ .Values.envEndpoint.host }/
31   nodeSelector:
32     hardware: { .Values.frontend.nodeSelector.hardware }
```

Quellcode 6.9: deployment.yaml

Die Spezifikation des Deployments gibt die Anzahl der Replikationen der Pods an. Der Selector ist für die Selektion von Pods durch das Deployment zuständig. Das Template bezeichnet eine Menge von Pods mit einem Label, diese können dann von anderen Kubernetes-Objekten, wie Deployments und Services selektiert werden. Im *spec* werden die Docker-Images angegeben zur Ausführung auf einem oder mehreren Pods. Die *imagePullPolicy* bestimmt die Regeln für das Herunterladen von Images. Mit *Always* wird das gewählte Image immer heruntergeladen auch, wenn es sich bereits auf dem Hostsystem befindet. Ports gibt die lauschenden Ports des Containers im Pod an. Die Environmental-Variables sind für die Definition der Endpunkte des Dienstes. *NodeSelector* gibt die in Abschnitt 6.1.2 gekennzeichneten Nodes an, um zu bestimmen auf, welcher Hardware die Pods ausgeführt werden.

6.5.4 Persistent-Volumes

Die persistente Speicherung von Benutzerinformationen oder Bildern erfolgt durch Persistent-Volumes. Diese werden für die Dienste Authentication-Service und Facerecognition-Service benötigt. Ein PersistentVolume (PV) ist ein Speicher der in einem Kubernetes-Cluster von Administratoren oder dynamisch über Speicherklassen bereitgestellt wird. Ein PersistentVolumeClaim (PVC) ist eine Anfrage zur Nutzung von PV-Ressourcen in einem Kubernetes-Cluster. Im Folgenden werden die Persistent-Volumes der MongoDB Kubernetes-Ressourcenobjekte erläutert.

PersistentVolumeClaim

Ähnlich wie ein Pod, Systemressourcen wie CPU und Hauptspeicher nutzt. Benötigt ein PVC Systemressourcen in Form von Festplattenspeicher. Bei fehlenden PV wird durch dynamische Provisionierung für die Anfrage ein PV erstellt.

Die YAML-Datei für die MongoDB liegt in einem Unterverzeichnis mit den dazugehörigen Ressourcenobjekten. Unter *annotations* wird eine IF-Bedingung gestellt, die das Löschen nach der Nutzung des PVC erlaubt, wenn der Wert wahr ist wird es nicht gelöscht. Die *annotations: "helm.sh/resource-policy": keep* verhindert die Löschung einer Kubernetes-Ressource, wenn ein Helm-Chart deinstalliert wird [66]. Der *spec* beschreibt die Speicherklassen des PVC, dieser ist in dem Test-Cluster standardmäßig *local-path*. Die *accessModes* definieren die Zugriffsmodi des PV wie beispielsweise die Lese- und Schreibrechte mehrerer Clients. Für die MongoDB wird der Modus *ReadWriteOnce* verwendet. Dieser erlaubt Lese- und Schreibrechte für Pods die sich auf derselben Node befinden. *Resources* definiert mit *storage* die Menge an benötigten Speicherplatzes (vgl. Quellcode 6.10).

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: {{ .Values.mongodbvolume.persistence.claimName }}
5   annotations:
6     {{- if .Values.skipuninstall }}
7     "helm.sh/resource-policy": keep
8     {{- end }}
9 spec:
10  storageClassName: {{ .Values.mongodbvolume.persistence.storageClassName }}
11  accessModes:
12    {{- toYaml .Values.mongodbvolume.persistence.accessModes | nindent 4 }}
13  resources:
14    requests:
15      storage: {{ .Values.mongodbvolume.persistence.storage }}
```

Quellcode 6.10: pvc-claim.yaml

PersistentVolume

Der PVC kann dann von dem MongoDB-Deployment referenziert werden und als Speicher in den Pod eingebunden werden. Das Standardverzeichnis für die Aufbewahrung von Daten in MongoDB ist `/data/db` [67] (vgl. Quellcode 6.11).

```
1 volumeMounts:
2   - name: "mongo-data-dir"
3     mountPath: "/data/db"
4 volumes:
5   - name: "mongo-data-dir"
6     persistentVolumeClaim:
7       claimName: "{{ .Values.mongodbvolume.persistence.claimName }}"
```

Quellcode 6.11: Ausschnitt aus dem mongodb-deployment.yaml

Kapitel 7

Fazit und Ausblick

7.1 Fazit

Das Ziel war die Konzeption und Implementierung einer Microservice-Architektur auf einem hybrides-Kubernetes-Cluster. Diese soll als Blaupause für die Vorgehensweise der Implementierung dienen. Anforderungen wie automatischer Tests der Dienste konnte wegen Zeitmangel nicht mehr realisiert werden. Die Gesichtserkennung benutzt auch keine Deep-Learning-Algorithmen, sondern Vektorbasierte Algorithmen. Trotzdem sind die Grundvoraussetzungen für die Auslieferung und Bereitstellung einer solchen Anwendung basierend auf Deep-Learning-Methoden gegeben. Die Installation und Organisation von containerisierten Arbeitslasten durch die Rancher-Plattform erleichterte die Überwachung der Anwendungen. Durch die Benutzeroberfläche wurde auch die Komplexität zur Verwaltung des Kubernetes-Clusters reduziert.

7.2 Einschränkungen

Hardware wie Industrierechner die später in Produktionsanlagen eingesetzt werden standen nicht zur Verfügung. Die Installation des Kubernetes-Cluster mit k3s wurde auf virtuelle private Server realisiert (siehe Abschnitt 6.1). Die Implementierung der Microservices konnte deshalb nicht in einem Produktionsumfeld eingesetzt werden. Der Anwendungsfall der Webanwendung ist deshalb nur bedingt der Realität entsprechend, da Computer-Vision im Bereich der Anlagentechnik genutzt wird und nicht zur Authorisation von Personal. Jedoch sind viele der Schritte, ähnlich ausführbar wie auf einem Kubernetes-Cluster mit on-Premise Geräten anstatt von cloudbasierter Hardware. Der blaupausenartige Aufbau der Entwicklungsschritte gilt auch für den Aufbau von Microservices im Produktionsumfeld.

7.3 Ausblick

Die weitere Umsetzung von Anwendungen im Microservice-Architektur-Stil wird weiter untersucht. Allem voran der Flexibilität des Kubernetes-Clusters.

Kapitel 8

Ergebnisse

8.1 Microservice

8.1.1 Frontend-Serivce

8.1.2 Backend-Serivce

8.1.3 Authentifizierungs-Serivce

8.1.4 Loadbalancer

8.1.5 Kubernetes Cluster

Abkürzungsverzeichnis

PoC	Proof of Concept
VM	Virtuelle Maschine
SHA	Secure Hash Algorithm
API	Application Programming Interface
REST	Representational State Transfer
OSI	Open Systems Interconnection
YAML	Yet Another Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
AWS	Amazon Web Services
HMI	Human Interface
SSH	Secure Shell
WSGI	Web Server Gateway Interface
HTML	Hypertext Markup Language
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
CSS	Cascading Style Sheets
ACME	Automatic Certificate Management Environment
PVC	PersistentVolumeClaim
PV	PersistentVolume

Literaturverzeichnis

- [1] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.
- [2] "Krones linatronic 735," Feb. 2022. [Online]. Available: <https://www.krones.com/de/produkte/maschinen/leerflaschen-inspektionsmaschine-linatronic-735.php>
- [3] Y. Zhou, Y. Yu, and B. Ding, "Towards mlops: A case study of ml pipeline platform," in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 2020, pp. 494–500.
- [4] N. Poulton, *Docker deep dive : zero to Docker in a single book*, 2020th ed. [Germany]: Nigel Poulton, 2020.
- [5] "Docker overview," Jan. 2022. [Online]. Available: <https://docs.docker.com/get-started/overview/>
- [6] "About storage drivers," Jan. 2022. [Online]. Available: <https://docs.docker.com/storage/storagedriver/>
- [7] "Best practices for writing dockerfiles," Jan. 2022. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- [8] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.
- [9] "Are Containers Replacing Virtual Machines?" Aug. 2018. [Online]. Available: <https://www.docker.com/blog/containers-replacing-virtual-machines/>
- [10] "Was ist kubernetes?" section: docs. [Online]. Available: <https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/>
- [11] "Kubernetes components," section: docs. [Online]. Available: <https://kubernetes.io/de/docs/concepts/overview/components/>
- [12] "Kubernetes (k8s)," Feb. 2022, original-date: 2014-06-06T22:56:04Z. [Online]. Available: <https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.20.md/#urgent-upgrade-notes>

- [13] "Nodes," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/nodes/>
- [14] "Don't panic: Kubernetes and docker," Dec. 2020, section: blog. [Online]. Available: <https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/>
- [15] "Understanding kubernetes objects," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>
- [16] "Deployments," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- [17] N. Poulton, *The Kubernetes Book*, 2021st ed. [Germany]: Nigel Poulton, 2021.
- [18] "Service," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/service/>
- [19] "Ingress," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/ingress/>
- [20] "Ingress controllers," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>
- [21] "Layer 4 and layer 7 load balancing." [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/k8s-in-rancher/load-balancers-and-ingress/load-balancers/>
- [22] "What is kubernetes ingress?" [Online]. Available: <https://www.ibm.com/cloud/blog/kubernetes-ingress>
- [23] "Raspberry pi documentation - processors." [Online]. Available: <https://www.raspberrypi.com/documentation/computers/processors.html>
- [24] "K3s resource profiling." [Online]. Available: <https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/resource-profiling/>
- [25] "K3s: Lightweight kubernetes." [Online]. Available: <https://k3s.io/>
- [26] "K3s - lightweight kubernetes," Feb. 2022, original-date: 2018-05-31T01:37:46Z. [Online]. Available: <https://github.com/k3s-io/k3s>
- [27] "Possible to run k3s on one node (server and agent together)? · Issue #1279 · k3s-io/k3s." [Online]. Available: <https://github.com/k3s-io/k3s/issues/1279>
- [28] "flannel," Feb. 2022, original-date: 2014-07-10T17:45:29Z. [Online]. Available: <https://github.com/flannel-io/flannel>
- [29] "Overview." [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/overview/>
- [30] S. Buchanan, J. Rangama, and N. Bellavance, "Deploying and using rancher with azure kubernetes service," in *Introducing Azure Kubernetes Service : A Practical Guide to Container Orchestration*, S. Buchanan, J. Rangama, and

- N. Bellavance, Eds. Berkeley, CA: Apress, 2020, pp. 79–99. [Online]. Available: https://doi.org/10.1007/978-1-4842-5519-3_6
- [31] “Access a Cluster with Kubectl and kubeconfig.” [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/cluster-admin/cluster-access/kubectl/>
- [32] “Overview.” [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/overview/>
- [33] “Architecture Recommendations.” [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/overview/architecture-recommendations/>
- [34] “Rancher Agents.” [Online]. Available: <https://rancher.com/docs/rancher/v2.5/en/cluster-provisioning/rke-clusters/rancher-agents/>
- [35] “hybrid-cloud,” May 2021. [Online]. Available: <https://www.ibm.com/de-de/cloud/learn/hybrid-cloud>
- [36] “Microservices.” [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [37] “Microservices Pattern: Decompose by business capability.” [Online]. Available: <http://microservices.io/patterns/decomposition/decompose-by-business-capability.html>
- [38] “Softwarecomponent.” [Online]. Available: <https://martinfowler.com/bliki/SoftwareComponent.html>
- [39] S. Newman, “Implementing microservice communication.” Sebastopol, CA: O’Reilly Media, Sep. 2021.
- [40] M. Richards, *Microservices vs. Service-Oriented Architecture*. O’Reilly UK.
- [41] S. Newman, *Building microservices*, 2nd ed. Sebastopol, CA: O’Reilly Media, Sep. 2021.
- [42] “A Conversation with Werner Vogels - ACM Queue.” [Online]. Available: <https://queue.acm.org/detail.cfm?id=1142065>
- [43] “Protocol Buffers | Google Developers.” [Online]. Available: <https://developers.google.com/protocol-buffers>
- [44] “Boundedcontext.” [Online]. Available: <https://martinfowler.com/bliki/BoundedContext.html>
- [45] “Zwei Betriebssysteme auf einem Gerät | B&R Industrial Automation.” [Online]. Available: <https://www.br-automation.com/>
- [46] “RTS Hypervisor - Hardware partitioning - Real-Time Systems.” [Online]. Available: <https://www.real-time-systems.com/de/use-cases/rts-hypervisor-hardware-partitioning.html>

- [47] "Connected HMI: die neue Generation der Maschinenvisualisierung - Kro-nes." [Online]. Available: <https://www.krones.com/de/produkte/innovationen/maschinenvisualisierung-connected-hmi.php>
- [48] "Helm Architecture." [Online]. Available: <https://helm.sh/docs/topics/architecture/>
- [49] "Welcome to Flask — Flask Documentation (2.0.x)." [Online]. Available: <https://flask.palletsprojects.com/en/2.0.x/>
- [50] G. C. Hillar, *Hands-On RESTful Python Web Services*, 2nd ed. Birmingham, England: Packt Publishing, Dec. 2018.
- [51] "Design — Gunicorn 20.1.0 documentation." [Online]. Available: <https://docs.gunicorn.org/en/latest/design.html>
- [52] "OpenCV: Introduction." [Online]. Available: <https://docs.opencv.org/3.4/d1/dfb/intro.html>
- [53] S. Ansari, "Core Concepts of Image and Video Processing," in *Building Computer Vision Applications Using Artificial Neural Networks: With Step-by-Step Examples in OpenCV and TensorFlow with Python*, S. Ansari, Ed. Berkeley, CA: Apress, 2020, pp. 9–26. [Online]. Available: https://doi.org/10.1007/978-1-4842-5887-3_2
- [54] "OpenCV: Introduction to OpenCV-Python Tutorials." [Online]. Available: https://docs.opencv.org/4.x/d0/de3/tutorial_py_intro.html
- [55] E. F. de Souza Soares, R. Melo Thiago, L. G. Azevedo, M. de Bayser, V. Torres da Silva, and R. F. de G. Cerqueira, "Evaluation of server push technologies for scalable client-server communication," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 1–10.
- [56] "Introduction | Socket.IO." [Online]. Available: <https://socket.io/docs/v4/>
- [57] "python-socketio — python-socketio documentation." [Online]. Available: <https://python-socketio.readthedocs.io/en/latest/>
- [58] S. Patni, "Fundamentals of RESTful APIs," in *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS*, S. Patni, Ed. Berkeley, CA: Apress, 2017, pp. 1–9. [Online]. Available: https://doi.org/10.1007/978-1-4842-2665-0_1
- [59] "Pricing · Plans for every developer." [Online]. Available: <https://github.com/pricing>
- [60] nishanil, "Entwicklungsworkflow für Docker-Apps." [Online]. Available: <https://docs.microsoft.com/de-de/dotnet/architecture/microservices/docker-application-development-process/docker-app-development-workflow>
- [61] J. Belamaric and C. Liu, *Learning coredns*. Farnham, England: O'Reilly UK, Sep. 2019.
- [62] "Issuer," section: docs. [Online]. Available: <https://cert-manager.io/docs/concepts/issuer/>

- [63] "ACME." [Online]. Available: <https://cert-manager.io/docs/configuration/acme/>
- [64] "Assign Pods to Nodes using Node Affinity," section: docs. [Online]. Available: <https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/>
- [65] "Taints and Tolerations," section: docs. [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/>
- [66] "Chart Development Tips and Tricks." [Online]. Available: https://helm.sh/docs/howto/charts_tips_and_tricks/
- [67] "Manage mongod Processes — MongoDB Manual." [Online]. Available: <https://docs.mongodb.com/manual/tutorial/manage-mongodb-processes/>

Abbildungsverzeichnis

2.1	Docker Architektur in Anlehnung an [4, S.11]	5
2.2	Image Layers in Anlehnung an [4, S.61]	6
2.3	Virtualisierungsmöglichkeiten angelehnt an [9].	7
2.4	Komponenten eines Kubernetes Cluster in Anlehnung an [11].	8
2.5	K3s Architektur in Anlehnung an [25].	13
2.6	Rancher Server Kommunikation mit einem downstream k3s Cluster, überarbeitete Abbildung von [32]. (Im Sinne der späteren Architektur nachgebildet)	14
2.7	Gegenüberstellung von Monolithen und Microservices [36]	16
4.1	Grobentwurf der Infrastruktur	26
4.2	Grobentwurf der Anwendung	27
4.3	Vorgehen des Entwicklungsprozesses in Schichten	28
5.1	Microservice-Entwicklung in Anlehnung an [60]	32
5.2	Kubernetes-Entwicklung in Anlehnung an [60]	33
5.3	Lokale Microservice Entwicklung	34
5.4	BPNM Modell - Helm Installation der Microservices	35
6.1	KubeVision Frontend Dashboard	40

Quellcodeverzeichnis

2.1	deployment.yaml [16]	10
2.2	service.yaml [18]	11
2.3	ingress.yaml [19]	12
6.1	issuer.yaml [63]	37
6.2	cert.yaml [63]	37
6.3	Node-Labels	38
6.4	Node-Taints	38
6.5	Dockerfile	41
6.6	Ausschnitt aus dem docker-compose.yaml	42
6.7	face-service.yaml	43
6.8	kubevision-ingress.yaml	44
6.9	deployment.yaml	45
6.10	pvc-claim.yaml	47
6.11	Ausschnitt aus dem mongodb-deployment.yaml	47