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

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

Ostbayerische Technische Hochschule
Amberg-Weiden

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 benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 2. März 2022

Unterschrift:

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

Ostbayerische Technische Hochschule
Amberg-Weiden

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:	Herr 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 Arbeit ist die Konzeption und Implementierung einer prototypischen Anwendung im Microservice-Architektur-Stil. Die Software soll dabei einen Anwendungsfall im Bereich der künstlichen Intelligenz abdecken. Die Auslieferung und Bereitstellung der Anwendung soll auf einer hybriden-Umgebung in Kubernetes stattfinden.

Das erste Kapitel widmet sich der Zielsetzung für die Entwicklung der Software. Im zweiten Kapitel werden die Grundlagen zum Verständnis der folgenden Kapitel erklärt. Danach werden im dritten Kapitel die derzeitigen Bestrebungen der Krones AG im Bereich der Modernisierung von Infrastrukturen durch Cloud-Technologien analysiert. Darauf aufbauend folgt das vierte Kapitel, welches einen Lösungsansatz in Form von Konzepten und Entwürfen formuliert. Im fünften Kapitel wird dann das Lösungskonzept ausgearbeitet und die Vorgehensweisen hinsichtlich Design, Entwicklung und der Architektur definiert. Das sechste Kapitel ist die technische Umsetzung des Konzepts und beschreibt den Aufbau der Implementierung. Abschließend werden die Vorgehensweisen und Ergebnisse in Kapitel sieben zusammgefasst und ein Ausblick für die Zukunft artikuliert.

Schlüsselwörter: Docker, Kubernetes, Rancher, OpenCV, Helm, Flask, Microservice

Abstract:

The goal of this thesis is the design and implementation of a prototypical application using the microservice architecture style. This prototype has to cover a use case in the area of artificial intelligence. The appliaction will be deployed and delivered on a hybrid Kubernetes cluster.

The first chapter is dedicated to the objective for the development of the software. The second chapter explains the basics to understand the following chapters. Chapter three analyzes Krones AG's current efforts in the area of infrastructure modernization through cloud technologies. This is followed by the fourth chapter, which formulates a solution approach in the form of concepts and designs. Furthermore, the fifth chapter then elaborates the solution concept and defines the approach in terms of design, development and architecture. Based on this, the sixth chapter describes the technical realization of the concept and the structure of the implementation. Finally, chapter seven summarizes the procedures and results and articulates an outlook for the future.

Keywords: Docker, Kubernetes, Rancher, OpenCV, Helm, Flask, Microservice

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Krones AG bietet Anlagen sowohl für die Getränkeindustrie als auch für Nahrungsmittelhersteller an, von der Prozesstechnik bis hin zur IT-Lösung. Die Komplettlinie beeinhaltet auch das Bereitstellen von Software in 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 Softwarteunternehmen 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 Virtual-Edge-Devices als Knotenpunkte in einem Kubernetes-Cluster fungieren, sich Ressourcen teilen, untereinander kommunizieren und Softwarepakete unkompliziert bereitstellen. Die Integration von 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 erleichtert die Aufteilung großer Projekte in kleine unabängige Services, die mittels Orchestrierungsplattformen adäquat kombiniert 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 Flexibiltät einer solchen Infrastuktur ist es möglich Anwendungsfälle im Bereich der künstlichen Intelligenz für die Industrie zu konzipieren. Die Anlage *Linatronic AI* der Krones AG nutzt bereits Computer-Vision-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 der Informationsverarbeitung, und die 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, mit einer Weboberfläche, welche über eine Domain erreichbar ist. Ein Anmeldeverfahren mit 2-Faktor-Authentifizierung soll über einen Backend-Service mit Gesichtserkennung die Autorisierung eines Nutzers ermöglichen. 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 Hardwareschichten statt. Das gesamte 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 Entwicklung dessen maßgeblich 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 beeinhalten 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-Umgebung 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 eingeganen, da diese zum Verständnis nicht nötig ist.

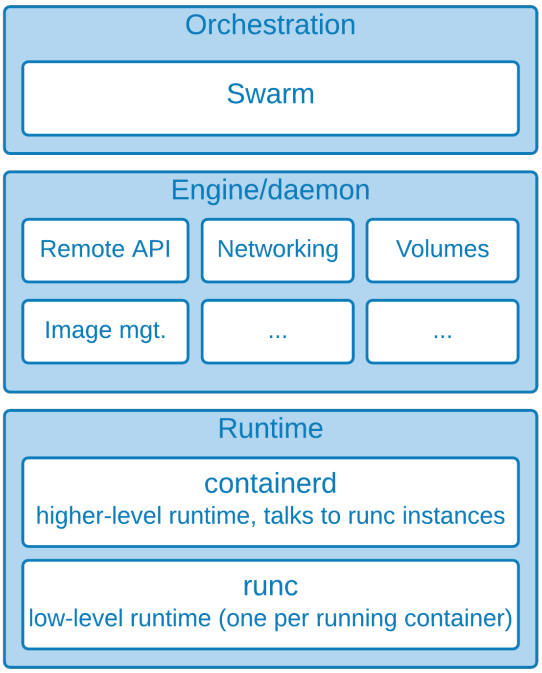

Abbildung 2.1: Docker Architektur in Anlehnung an [4, S.11]

2.1.2 Images und Container

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

Registrys

Das beziehen von Images erfolgt über sogenannte Image Registrys. Bei Docker ist dies standardmäßig `https://hub.docker.com` und das eigene lokale Registry. Es ist auch möglich, eigene Registrys 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 bereits vorhandene Schichten erneut verwenden. Die feste Reihenfolge ermöglicht eine ressourceneffiziente Verwaltung von Builds, indem man oft wechselnde Schichten oben platziert und somit weniger volatile Schichten häufiger wiederverwendet. Die Leistung beim Erstellen und Zusammenführen von Schichtem hängt vom Dateisystem des Hostsystems ab. Eine Schicht kann aus mehreren Dateien bestehen und einzelne Dateien aus der unterliegenden Schicht durch neue ersetzen.

Das Starten eines Containers fügt auf die bereits bestehenden Schichten einen „Thin R/W layer“ – „Container layer“ genannt – hinzu. Dieser gewährt Schreib- und Lese-

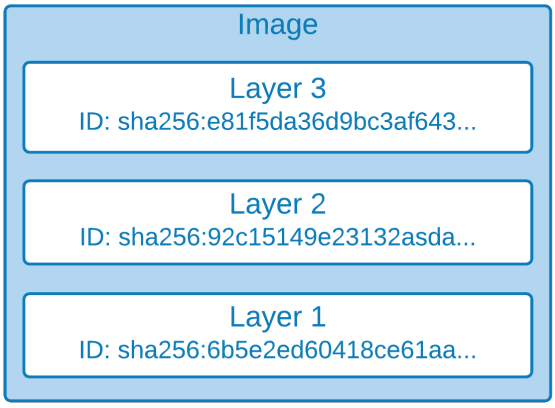

Abbildung 2.2: Image Layers in Anlehnung an [4, S.61]

rechte während der Laufzeit des Prozesses. Jeder dieser Container hat somit einen individuellen Zustand, der unähnlich vom abstammenden 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 beeinhaltet 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. Ein Unterschied ist, dass die Einrichtung von VMs arbeitsintensiver ist und weitaus mehr Ressourcen beansprucht, da sie ein ganzes Betriebssystem ausführt [8]. Container-Technologien bauen hingegen nur auf

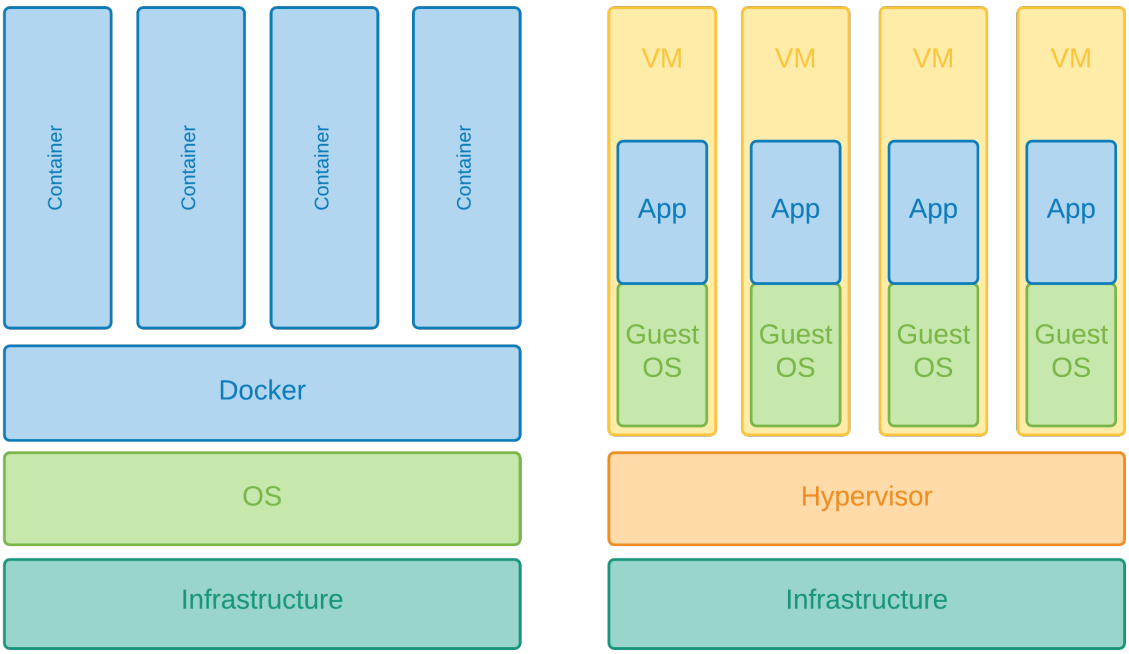

Abbildung 2.3: Virtualisierungsmöglichkeiten angelehnt an [9].

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

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

Die Einsparung von Ressourcen und das einfache 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 stellt Google das Open-Source-Projekt zur Verfügung, das auf 15 Jahre Erfahrungen mit Produktions-Workloads aufbaut [10].

2.2.1 Cluster

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

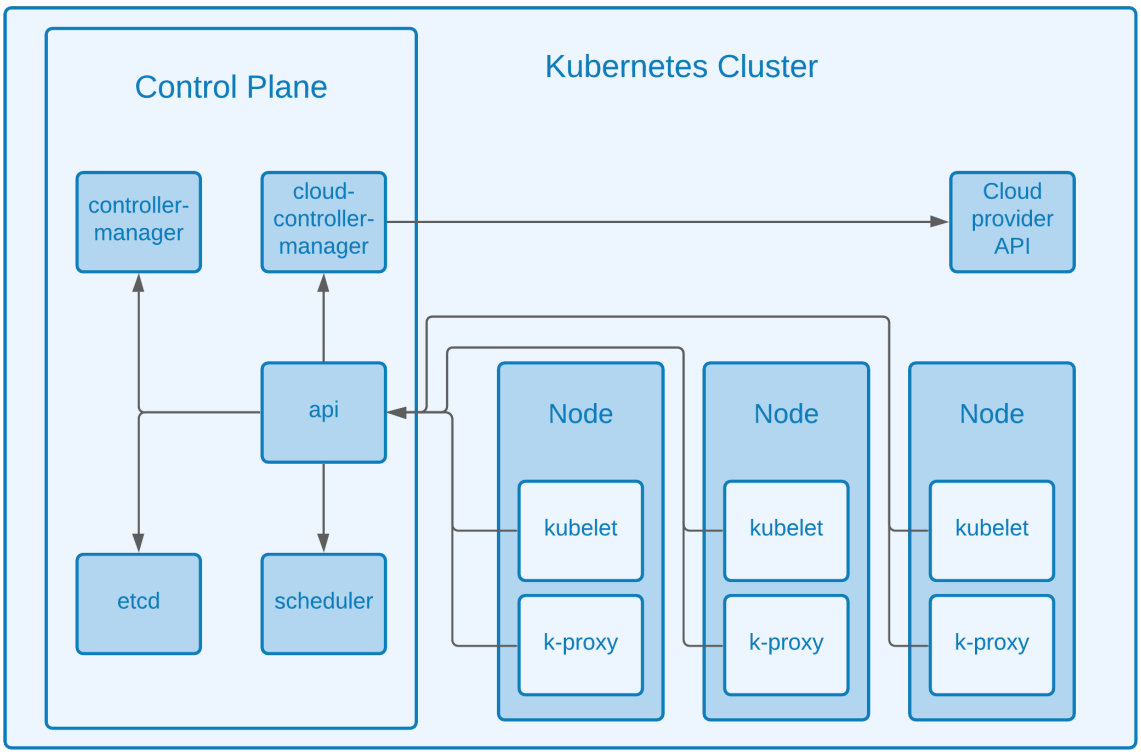

Abbildung 2.4: Komponenten eines Kubernetes Cluster in Anlehnung an [11].

Control Plane

Control Planes[1] 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.

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

Node

Eine Kubernetes-Node[2] 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.

Kubelet

Kubelet fungiert als „node agent“ und registriert die Nodes 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 in Kubernetes, 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 [14].

- **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.

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

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Quellcode 2.1: deployment.yaml [15]

Deployments und Pods

Das Einbinden von Pods in Deployments ermöglicht Kubernetes das Beziehen von Metadaten für die Verwaltung von Skalierung, Rollouts, Rollbacks und Selbstheilungsprozessen [16, 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 [17].

Pods beanspruchen Netzwerkressourcen, wie IP-Adressen und DNS-Namen 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 [17]:

- **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.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Quellcode 2.2: service.yaml [17]

Bei der Erstellung eines Services entsteht ein Endpunkt für REST-Aufrufe. Der zugehörige Service-Controller lauscht auf den Endpunkten des selektierten Pods und konfiguriert den Service dementsprechend. Die Verantwortung des Service-Controllers ist die Erstellung, Aktualisierung und Löschung von Services [17].

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 [18]. Die Anforderung für die Implementierung eines Ingress ist der Ingress-Controller, welcher nicht automatisch mit einem Cluster gestartet wird. In der Dokumentation werden deshalb eine Vielzahl an Third-Party Implementierungen aufgelistet [19]. 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 [18]. Die übliche Verwendung eines Controllers beeinhaltet die Lastenverteilung durch Weiterleiten des Datenverkehrs an Services. Diese Kommunikation findet, wie auch bei dem NGINX-Ingress-Controller [20], in der Anwendungsschicht des OSI-Schichtenmodells statt und ermöglicht dadurch die Lastenverteilung von öffentlichen Endpunkten zu internen Pods in einem Cluster [21]. 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 [18]:

- **ingressClassName**: definiert den Ingress-Controller.
- **rules**: die Zusammsetzung 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.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: "nginx-example.com"
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
```

Quellcode 2.3: ingress.yaml [18]

2.2.6 Lightweight Kubernetes

Ligthweight 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 Verschlankung 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-CPU mit 1,50 GHz[3] [23]. Der hauptsächliche Verwendungszweck von k3s sind IoT-Geräte, da sekundäre Kubernetes-Inhalte entfernt wurden. [24]. Trotz dieser Reduzierung bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden, soweit möglich, parallel auf dem neusten Stand gehalten [25].

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, dass Server parallel einen Agent-Prozess auf demselben Knoten starten und somit Arbeitslasten mithilfe von Kubelet ausführen [26]. Weiterhin wird, im Gegensatz zu

[3] Einplatinencomputer Raspberry Pi 4B, basierend auf ARM [22].

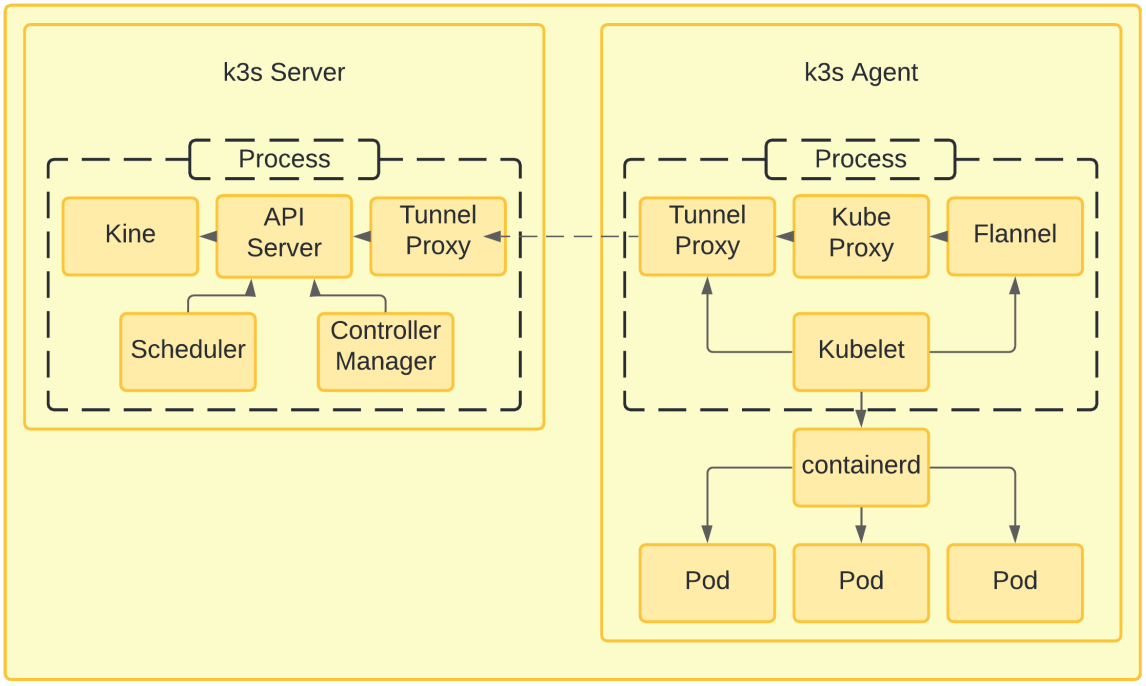

Abbildung 2.5: K3s Architektur in Anlehnung an [24].

Kubernetes, containerd weiterhin unterstüzt und kommt mit Kubelet vorinstalliert [24]. 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 [25]. 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 wiederum Agents auf Hostssystemen startet. Flannel 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 [27].

2.2.7 Rancher

In diesem Unterabschnitt wird die Open-Source-Lösung Rancher von dem gleichnamigen Unternehmen zur Orchestierung 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 unabhängig von Cloud-Anbietern, weshalb Cluster in der Praxis mit dersel-

ben Rancher-Instanz auf AWS, Azure oder anderen Cloud-Anbietern betreut werden können [28].

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 [28, 29].

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 beeinhaltet die Adresse zum Rancher-Server, Nutzerrechte und Zertifizierungszeichen [30].

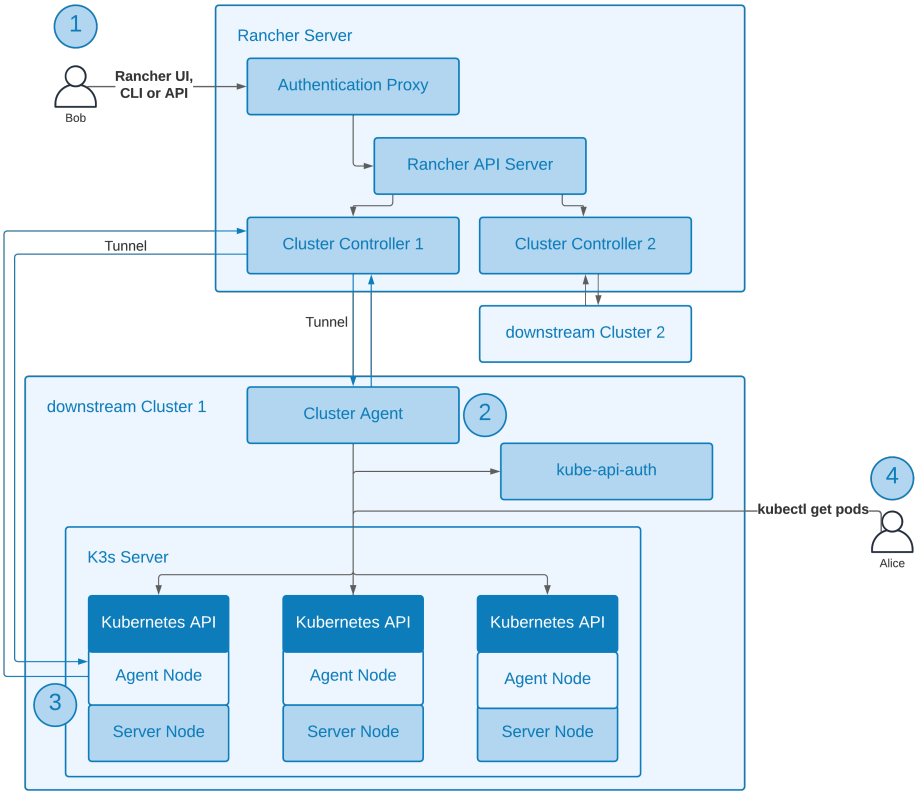

Abbildung 2.6: Rancher-Server-Kommunikation mit einem downstream-k3s-Cluster, überarbeitete Abbildung von [31]. (Im Sinne der späteren Architektur nachgebildet)

Die Abbildung 2.6 zeigt den Vorgang von zwei Benutzern, die auf ein von Rancher verwaltetes downstream-k3s-Cluster[4] zugreifen. Die nachfolgende Beschreibung aus

[4]Die offiziele Bezeichnung für ein Kubernetes-Cluster unter Rancher ist **downstream Cluster** [32]

der Dokumentation gibt die einzelnen Schritte mit der in der Abbildung nummerierten Posten wieder [31].

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-Agent[5] 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 gemeinsamen Infrastruktur laufen. Dies ermöglicht die flexible Orchestrierung von Anwendungen auf Hostssystemen vor Ort oder in der Cloud [34].

Der Schwerpunkt solcher Hybrid-Clouds liegt dabei bei der Portierbarkeit der Arbeitslasten auf alle Cloud-Umgebungen. Dafür ist die Aufbereitung oder Entwicklung alter oder neuer Anwendungen für cloud-native Technologien nötig; mehr dazu im Abschnitt 2.3 zu Microservices. Private-Clouds können auch von Drittanbietern, durch externe Rechenzentren, 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-Orchestierungsplattform, wie Kubernetes, und ermöglicht die nahtlose Implementierung von Cloud-Umgebungen [34].

[5] Ein Rancher-DaemonSet zur Interaktion mit Nodes. Nicht zu verwechseln mit dem Node-Agent von k3s [33].

2.3 Microservice

Im Folgenden wird der Microservice-Architektur-Stil und dessen Eigenschaften näher erläutert. Als Hauptquelle dient der häufig zitierte Artikel [35] von Fowler und Lewis.

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 an Business-Capabilities[6]. Zusammenhängende Dienste werden dezentral gehalten und können in unterschiedlichen Programmiersprachen oder Technologien realisiert werden [35].

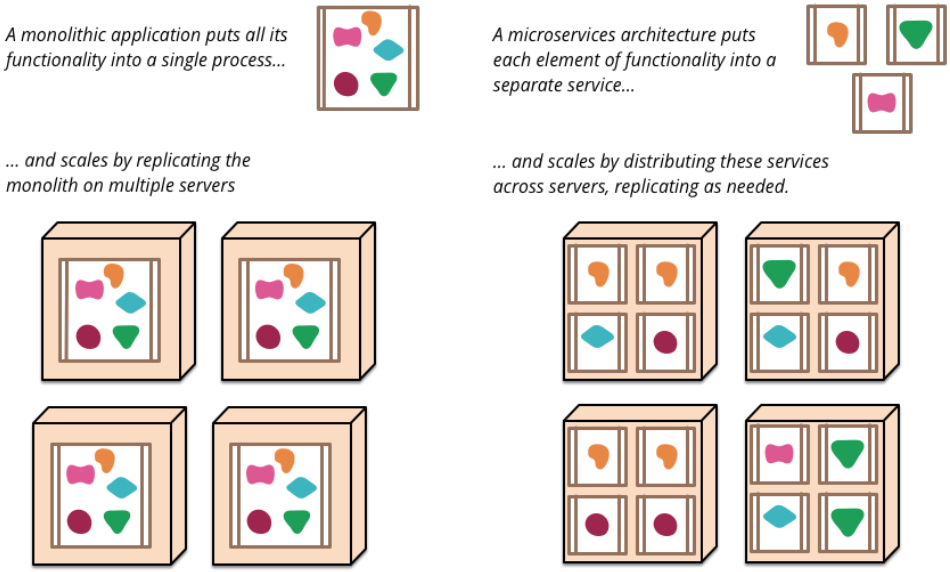

Abbildung 2.7: Gegenüberstellung von Monolithen und Microservices [35]

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 entgegentreten und es Entwicklerteams ermöglichen

[6]Business-capability bezeichnet ein Konzept, das aus Sicht der Geschäftsarchitektur modelliert wird [36].

einzelne Softwarekomponenten zu verwalten und gegebenenfalls in einer anderen Programmiersprache zu verwirklichen [35].

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 [35]. 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.“ [37]

Dienste einer Microservice-Architektur stellen Softwarekomponenten dar, die mittels Web-Service-Anfragen oder Remote Procedure Calls (RPCs)[7] 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 [35].

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 Contracts[8] der jeweiligen Dienste müssen sinnvoll gestaltet werden. Weiterhin müssen Schnittstellen grobkörniger entworfen werden, um den höheren Ressourcenverbrauch gegenüber der lokalen Variante auszugleichen [35, 40].

Ein Dienst kann jedoch aus mehreren Prozessen bestehen. Ein Beispiel wäre ein Anwendungsprozess mit einer Datenbank, die nur von dieser Anwendung genutzt wird [35, 40].

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 Organisierung um die Business-Capabilities. Entwickler arbeiten

[7] RPC bezeichnet die Ausführung eines lokalen Aufrufs auf einem anderen Dienst [38].

[8] Service Contracts bezeichnen die Vereinbarung zwischen zwei Diensten. Darin werden die Übertragungsformate von Daten festgelegt [39].

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 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 [35].

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 “ [41].

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 sie auch Zuständigkeiten des Supports übernehmen [35].

Intelligente Endpunkte statt komplexer Infrastruktur

Die Kommunikation von Diensten über Endpunkte soll soweit 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 Messaging[9]. 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 [35].

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, dass Abteilungen innerhalb eines Unternehmens Attribute unterschiedlich interpretiert und

[9] Kommunikation über binäre Protokolle wie Protocol-Buffers [42].

dies zu Inkonsistenz in Datensätzen führt. Eine Anwendung mit getrennten Softwarekomponenten erhöht diese Komplexität weiter [35]. Deshalb ist es sinnvoll, einen „Bounded Context“ zu definieren, welcher innerhalb größerer Teams zur Darstellung von Wechselwirkungen eines Modells dient [43].

Design for failure

Softwarekomponenten müssen den Ausfall von anderen Diensten tolerieren. Eventbasierte Kommunikation führt oft zu Fehlverhalten und kann durch Überwachungstools präventiv verhindert werden [35].

Kapitel 3

Analyse

Das vorherige Kapitel widmete sich dem Aufbau von Containern und deren Verwaltung. Darauf aufbauend wurde auch die mögliche Realisierung von Anwendungen im Micorservice-Architektur-Stil besprochen. Dieses Kapitel 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 sowie 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. Betrachtungsgenstand des PoC war die Virtualisierung der bereits vorhandenen Industrierechner der Firma B&R, um diese 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 Virualisierungsmöglichkeiten realisiert wird.

Virtualisierung

Das Unternehmen B&R steht in Kooperation mit der Firma Real Time Systems, welches Technologien für die Virtualisierung von Echtzeit Betriebssystemen anbietet [44]. Dafür

wird ein Hypervisor genutzt, um gleichzeitig unterschiedliche Echtzeitbetriebssysteme in Form von VMs auszuführen. Dies ermöglicht auch die Zuteilung von Hardwareressourcen auf die laufenden VMs. Ein Vorteil ist, dass keine zusätzliche Hardware benötigt wird. Die Zuweisung für Hardwareschnittstellen, wie Ethernet und USB-Ports, ist durch diesen Ansatz auch möglich. Virtuelle Netzwerke erlauben die Zuweisung von IPv4- und Mac-Adressen zu einzelnen Prozessorkernen, welche eine direkte Kommunikation über Internetprotokolle, wie TCP/IP oder COBRA, ermöglichen. Weiter kann jedes virtualisierte Betriebssystem Daten über eine gemeinsame Speicherpartition verwalten [45].

Connected Human Machine Interface (HMI)

Als Betriebssystem nutzt die Produktionsanlage *Windows 10 Embedded*, welches mit einem HMI über eine Touch-Oberfläche Bedienbar ist. Dieses ist für die zentrale Überwachung, Steuerung und Parametrisierung von Anlagenprozessen zuständig. Und erlaubt die Einteilung von produktionsrelevanten Aufgaben, wie Wartungsarbeiten, Materialversorgung und Qualitätskontrollen [46].

SUSE Linux Enterprise Micro

Basierend auf der Idee von containerisierten Arbeistlasten und Microservices wurde das Betriebssystem *SUSE Linux Enterprise Micro* entwickelt. Das für Edge-Szenarien entwickelte Open-Source-Betriebssystem ist das zweite virtualisierte Betriebssystem, das 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. Dies ermöglicht wartungsfreie Zustände der Geräte.

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 Mircoservices, können innerhalb des Cluster über Endpunkte oder Kubernetes-Objekte kommunizieren. Dadurch müssen Hostssysteme und die darauf laufende Software nicht mit deren Netzwerkadresse angesprochen werden, und zustätzlich 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 Produktionsumgegebungen. Weiterhin wurde die Bedienbarkeit der Benutzeroberfläche untersucht und es wurden Tests durchgeführt, bei welchen containerisierte Arbeitslasten verwaltet und Kubernetes-Objekte erstellt und miteinander verbunden wurden. 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, die Installation und das Updaten von Kubernetes-Anwendungen. Wichtige Konzepte von Helm sind Charts, die eine Ansammlung von Informationen zur Erstellung von Anwendungen als Kubernetes-Instanz sind. Configs beinhalten die Informationen der Konfiguration von Charts und erstellen oder verpacken diese. Ein Release bezeichnet eine laufende Instanz eines Charts in Kombination mit einer spezifischen Config [47].

Helm ist eine ausführbare Datei, die aus einem Kommandozeilentool besteht, dem Helm-Client. Dieser erlaubt die lokale Entwicklung von Charts und dem Verwalten von Repositoriy und unterschiedlichen 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 [47].

Der Anwendungsfall in Bezug auf die Krones 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 relevanten Themen im Zusammenhang mit Kubernetes. Während der Umsetzungsphase zeigten folgende Themen Potenzial.

Hybrid-Cloud

Die zukünftige Kubernetes-Infrastruktur kann die Integration von On-Premise und Cloud-Ressourcen in einer Softwareumgegbung 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 es, die geringe Rechenleistung von Virtual-Edge-Devices zu kombinieren. Der zukünftige Kunde kann bei diesem Ansatz eine Auswahl an Diensten treffen, die er für seine Anlage benötigt.

Ein weiterer Vorteil ist, dass eine modulare Architektur auf Containern die Auslieferung und Bereistellung 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. Denn die Verwaltung von Prozessen im Bereich Machine Learning umfasst Lebenszyklen mit mehreren Phasen. Diese beinhalten beispielsweise die Datenverarbeitung, Merkmalsextraktion, Modelltraining und Anwendungsbereitstellung. Die Ausführung der Phasen erzeugt Artefakte in Form von Datensätzen, Modellen und Anwendungskonfigurationen. Plattformen mit dieser Arbeitslast benötigen eine erhebliche Menge an Systemressourcen und ein stabiles Netzwerk [3]. Die Verarbeitung der Arbeitslast kann von Cloud-Ressourcen oder On-Premise Hardware mit Graphics Processing Units (GPUs) übernommen werden.

Kapitel 4

Lösungsansatz

Auf der Grundlage von Kapitel 3 wird ein Lösungsansatz für die Konzeption eines Softwareprojekts erabeitet. Zuerst wird das Fachkonzept mit den nötigen Anforderungen formuliert. Darauf aufbauend erfolgt die Grobkonzeption der grundlegenden Idee zum Entwicklungsprozess.

4.1 Fachkonzept

Anforderungen

Die zu entwickelnde Anwendung soll ein Anwendungszenario für die neu erschlossenen Anwendungsgebiete des PoCs umsetzen. Dabei muss die Anwendung eine Vielzahl an Anforderungen erfüllen. Die Software muss einfach auslieferbar, unabhängig und isoliert testbar sein. Dieser Prozess muss vor Inbetriebnahme automatisch ablaufen und eine individuelle Konfiguration bieten. 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 und Speicherung von Daten ermöglichen. Die Kommunikation muss zwischen anderen Anwendungen auf der Infrastruktur stattfinden können. 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 dieser aufbauen können. Ansätze bei der Entwicklung müssen austauschbar sein und in Teilschritte zerlegt werden. Die Schnittstellen der Anwendung müssen eine plattformübergreifende Kommunikation ermöglichen.

4.2 Grobkonzeption

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

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 Kubernetesfähigen 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 Computer-Vision-Modellen, etwa im Fall 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 Verarbeitung großer Daten zu ermöglichen. Für die prototypische Entwicklung der Anwendung sind keine strengen Kriterien für die Echtzeitkommunikation angedacht. Das Überschreiten von Zeitanforderungen stellt kein Versagen der Kommunikation dar.

Anwendungsszenario

Der Schwerpunkt der zu entwickelnden Anwendung soll ein Dashboard mit Authentifizierungsmechanismus sein. Dieses 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 sollen bei erneutem Aufruf der Website wieder verwendet werden.

Blaupause

Anhand der Grundlagen aus dem Abschnitt 2.3 können die wichtigsten Eigenschaften der blaupausenartigen Umsetzung einer Microservice-Architektur definiert werden. Die Anwendung soll nachvollziehbar entwickelt werden und als Fundament für spätere Entwicklungen dienen. Es ist wichtig, dass die Softwarekomponenten funktionsübergreifend entwickelt werden. Die Anwendung soll mit dem Entwicklungsteam wachsen und in agiler Vorgehensweise mittels Nutzerinformationen verbessert werden. Softwarekomponenten der Anwendung müssen austauschbar sein und durch unterschiedliche Technologien ersetzt werden können. Anwendungsschnittstellen müssen für gängige Kommunikationsprotokolle entwickelt werden. Diese müssen über

leichtgewichtige Kommunikationsmechanismen verfügen, wie die Unterstützung von RESTful-Protokollen über HTTP. Der Ausfall von Diensten muss bei Abhängigkeit anderer Softwarekomponenten tolerierbar sein.

4.3 Grobentwürfe

Auf der Grundlage von Abschnitt 4.2 werden die Grobentwürfe erstellt. Zunächst wird die Vorgehensweise bei der Entwicklung auf einer Kubernetes-Infrastruktur entworfen. Anschließend folgt die Anwendungsentwicklung und der allgemeine Entwicklungsprozess.

4.3.1 Infrastruktur

Die Abbildung 4.1 stellt das Zielsystem dar.

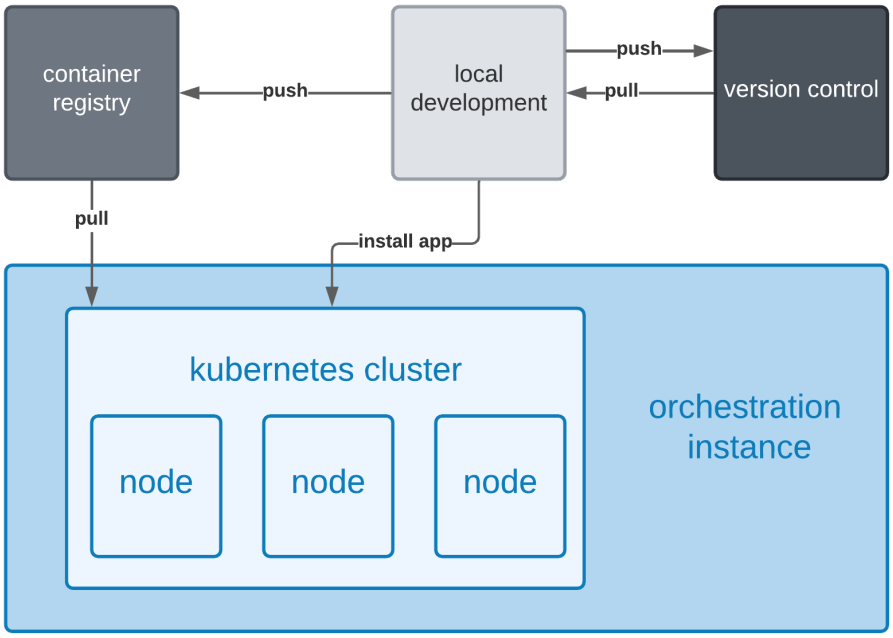

Abbildung 4.1: Grobentwurf der Infrastruktur

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

Container-Registry: Für die Auslieferung und Bereitstellung von Images wird ein Container-Registry verwendet. Dienste erhalten seperate Images mit einem Repository und können unabhängig abgerufen werden.

Kubernetes-Cluster: Das Kubernetes-Cluster wird von einer Orchestierungsinstanz verwaltet. Das Abrufen der Dienste erfolgt über ein öffentliches Container-Registry.

4.3.2 Anwendungsszenario

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

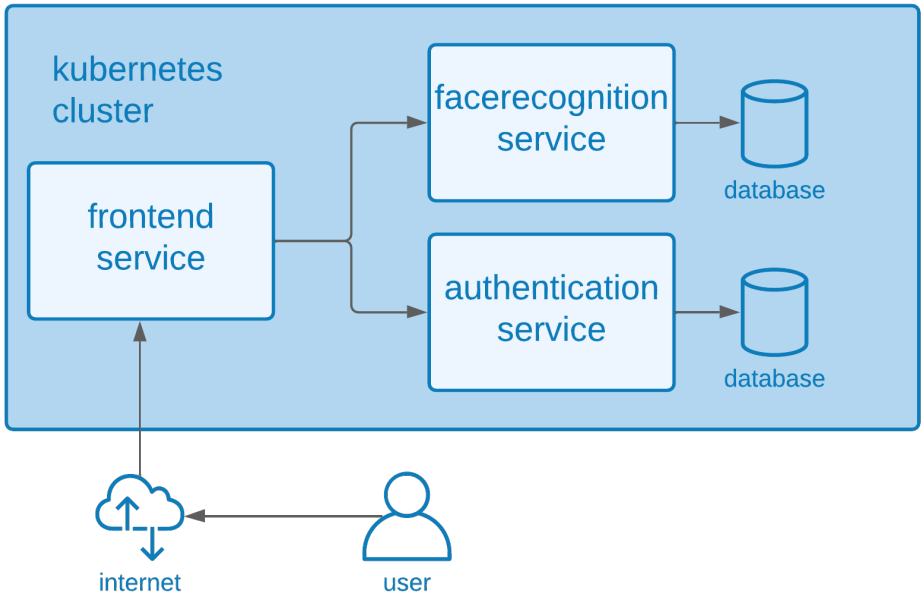

Abbildung 4.2: Grobentwurf der Awendung

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

Authentication-Service: Die Anmeldung und Registrierung eines Nutzerkontos erfolgt über den Authentication-Service. Dieser ermöglicht zudem die persistente Speicherung der Nutzerdaten in einer Datenbank.

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

4.3.3 Anwendungsentwicklung

Die Entwicklung der Anwendung wird in drei aufeinander Schichten eingeteilt (vgl. Abbildung 4.3).

Anwendungsentwicklung: Ein zentrales Repository beeinhaltet Dateiordner 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 Kommmunikation untereinander möglich ist. Schließlich wird das Image auf ein öffentliches Registry hochgeladen. Jeder Dienst hat dabei einen eigenen Speicherort in Form eines Images.

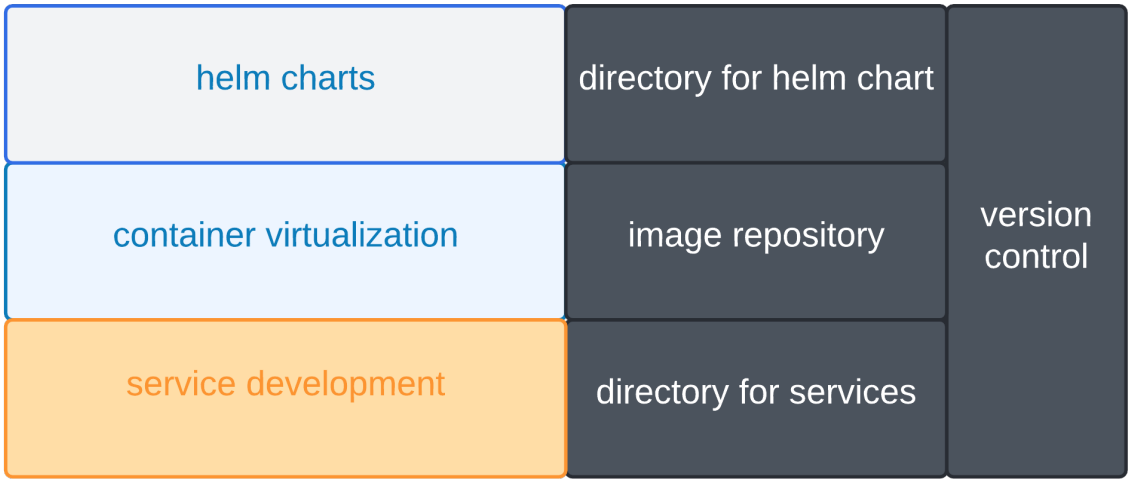

Abbildung 4.3: Vorgehen des Entwicklungsprozesses in Schichten

Helm-Charts: Die Auslieferung der Dienste erfolgt über das Container-Registry. Images werden von dem Kubernetes-Cluster heruntergeladen. Das zentrale Repository beinhaltet ein weiteres Verzeichnis für die Kubernetes-Ressourcenobjekte in Form von Helm-Charts. Falls die Entwicklungsumgebung Zugang zu einem Kubernetes-Cluster hat, können die Helm-Charts über den Helm-Package-Manager installiert und ausgeführt werden.

Kapitel 5

Lösungskonzept

Im Fokus des fünften Kapitels steht die Konzeption einer Anwendung im Microservice-Architektur-Stil.

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 wesentlichen Funktionalitäten für die Webentwicklung. Dafür bietet das Framework eine hohe Flexibilität, da die nötigen Bibliotheken vom Entwickler gewählt werden können [48] und es vereinfacht die Erstellung von APIs durch Blueprints [49, S.11].

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 [48]. Dadurch werden die Funktionalitäten der Webanwendung in Endpunkten strukturiert. Dies ist ideal für die Ausführung von losen Diensten, die über Endpunkte kommunizieren.

Weiterhin ermöglichen Flask-Abhängigkeiten, wie die Template-Engine Jinja das Rendern von HTML-Templates mit Daten aus der Flask-Anwendung. Und das Bereitstellen einer standartisierten Schnittstelle Web Server Gateway Interface (WSGI) über Werkzeugkasten. Diese ermöglicht die Verwendung der meisten Webserver [48].

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 einem Ausfall für den Neustart zuständig. Die Worker-Prozesse sind für die Verarbeitung von eingehenden Anfragen zuständig. Diese teilen sich in folgende Worker-Klassen auf. Sync-Workers bearbeiten Anfragen jeweils einzeln und unterstützten keine persistente Verbindung. Async-Workers basieren auf Greenlets und unterstützten mithilfe von Gevent asynchrone Koroutinen [50].

OpenCV

OpenCV ist eine Open-Source-Computer-Vision-Bibliothek[1], die zur Vearbeitung von Bildern verwendet wird [51]. Da ein Video nur eine Serie von Bildern ist, können die Techniken der Bildverarbeitung auch hier genutzt werden [52]. Die Bibliothek beinhaltet eine Vielzahl an Algorithmen mit Bezug zu Computer Vision oder Machine Learning. Diese untersützten auch die Verwendung von GPUs die auf den Programmierschnittstellen Compute Unified Device Architecture (CUDA) oder Open Computing Language (OpenCL) basieren [53]. 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 Benutzerobfläche erfolgt mit der Auszeichnungssprache HTML5 in Kombination mit der Skriptsprache JavaScript, um Interaktion mit dem Anwender zu ermöglichen. Für die erleichterte Gestaltung der Website wird das Frontend-CSS-Framework Bootstrap in der Version 5.0 genutzt.

5.1.3 Kommunikation

Socket.IO

Die bidirektionale und ereignisbasierte Echtzeitkommunikation zwischen den Diensten wird mithilfe der Bibliothek Socket.IO 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 ressourcenintensivere [54] Alternative HTTP-long-polling zurückgegriffen [55]. Für die Kommunikation der Webanwendung wird die Server Implementierung von Python-Socketio genutzt [56]. Die Implementierung der Anwendungslogik erfolgt über die Plattformunabhängige JavaScript-Bibliothek Socket.IO.

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

Representational State Transfer (REST)

REST ist eine auf Ressourcen basierende Architektur für verteilte Systeme. Diese Ressourcen werden über eine einheitliche Schnittstelle basierend auf HTTP Methoden zugänglich gemacht. Dabei ist jede Ressource über eine URL erreichbar. REST erlaubt dabei Ressourcen in verschiedene Datentypen zu repräsentieren, wie Text, XML, JSON etc. Die CRUD-Funktionen (create, read, update und delete) werden über die HTTP-Methoden GET, POST, PUT und DELETE realisiert [57].

5.1.4 Datenbank

MongoDB

MongoDB ist eine dokumentenorientierte Datenbank, bei der Daten nicht in einer Tabelle, sondern in Dokumenten gespeichert werden. Sie zählt damit zu den NoSQL-Datenbanken. Die Dokumentenorientierung 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 [58].

5.1.5 Versionsverwaltungssystem

GitHub

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

DockerHub

Wie in Abschnitt 2.1 beschrieben ist die Standard-Registry für Docker-Images DockerHub. Deshalb werden für die Bereistellung der Docker-Images die kostenfreien und öffentlichen Repositories verwendet. Die Entwicklung der Dienste erfolgt in getrennten 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 Hostssystem ausgeführt werden. Die Nutzung von Tools wie Docker-Compose erlauben das Starten mehrerer Container mithilfe von Konfigurationsdateien in Form von YAML-Dateien. Nach dem Ausführen der Container können diese getestet werden. Abschließend kann der Entwicklungsablauf fortgeführt werden oder ein Release für das Versionsverwaltungssystem und das Container-Repository erstellt werden.

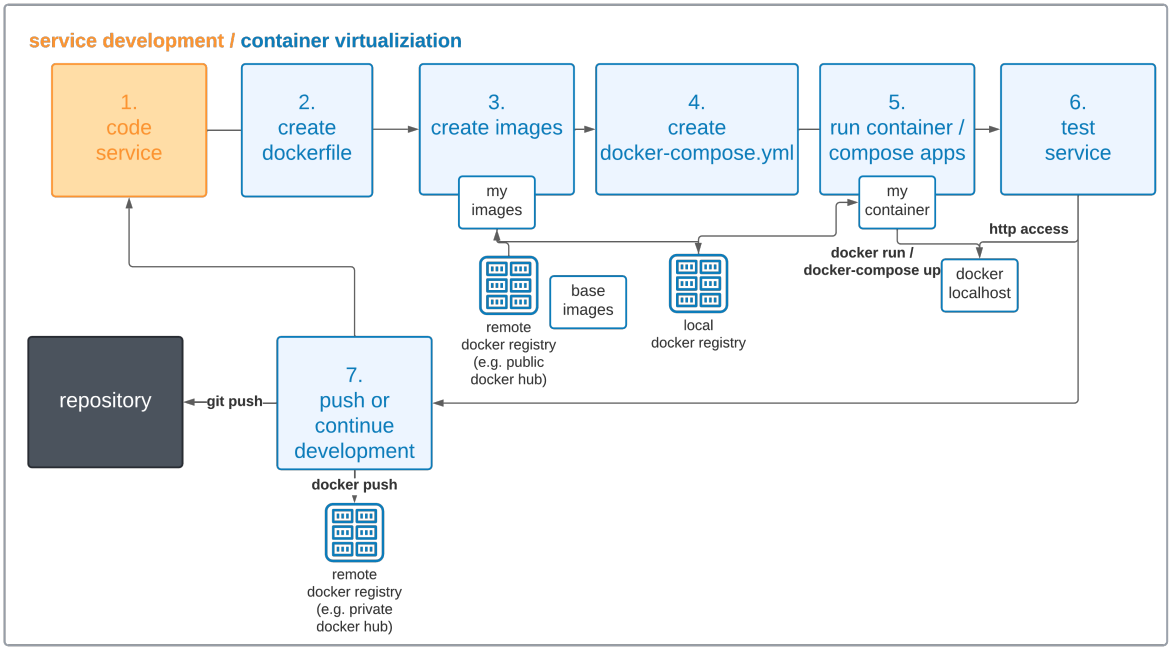

Abbildung 5.1: Microservice-Entwicklung in Anlehnung an [60]

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 das Egebnis Artefakt ins Versionsverwaltungssystem hochgeladen.

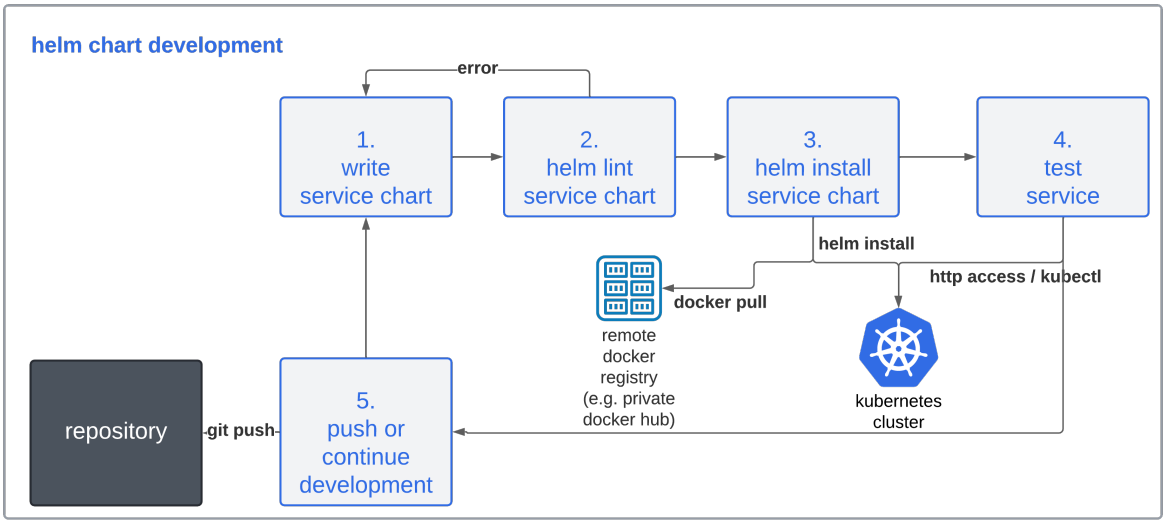

Abbildung 5.2: Kubernetes-Entwicklung in Anlehnung an [60]

5.3 Architektur

Dieser Abschnitt befasst sich mit der Architektur der zu entwickelnden Anwendung. Die Aufgaben der einzelnen Dienste der Microservice-Architektur wurden 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 Bereistellung der Container auf

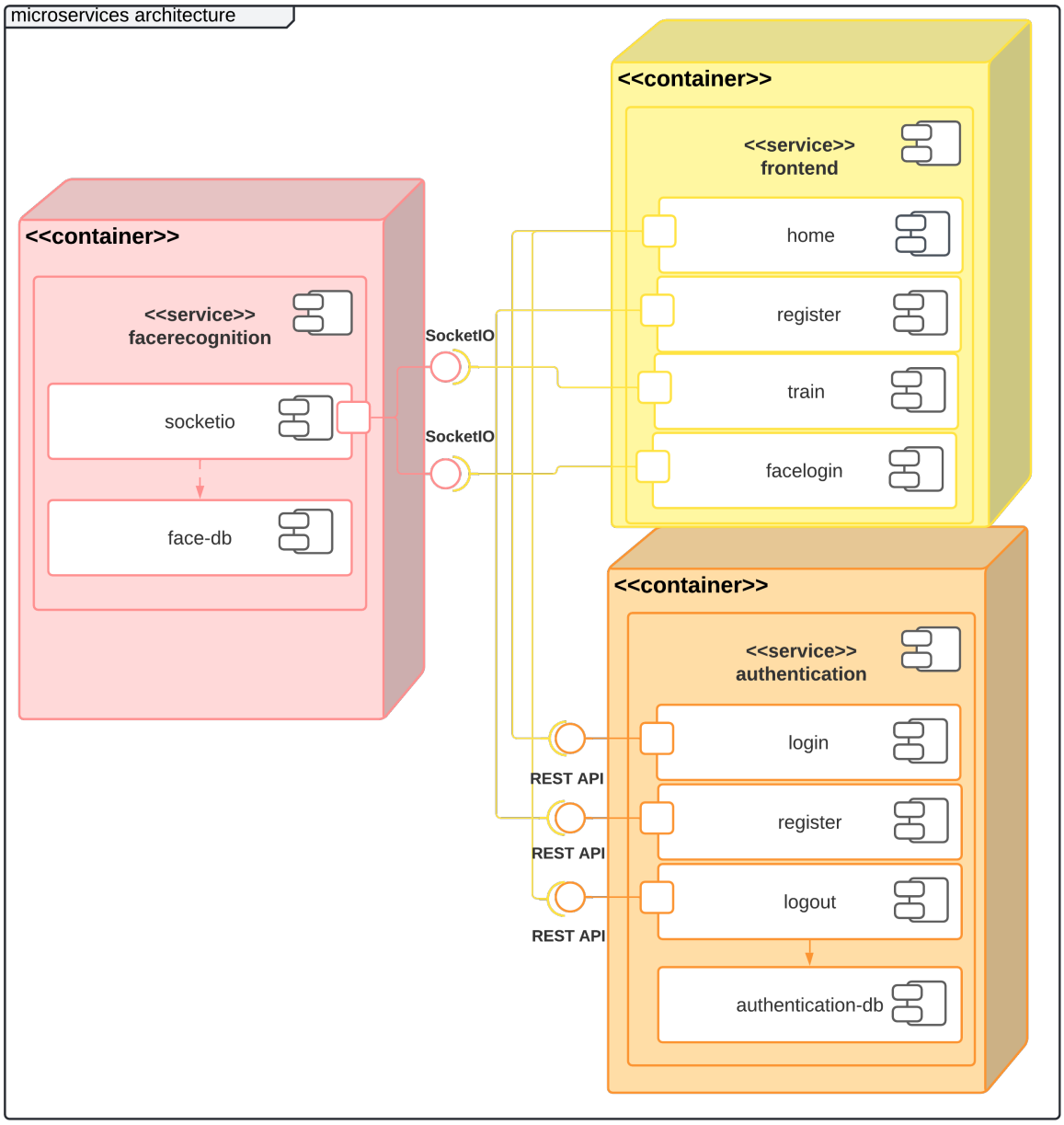

Abbildung 5.3: Lokale Microservice Entwicklung

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

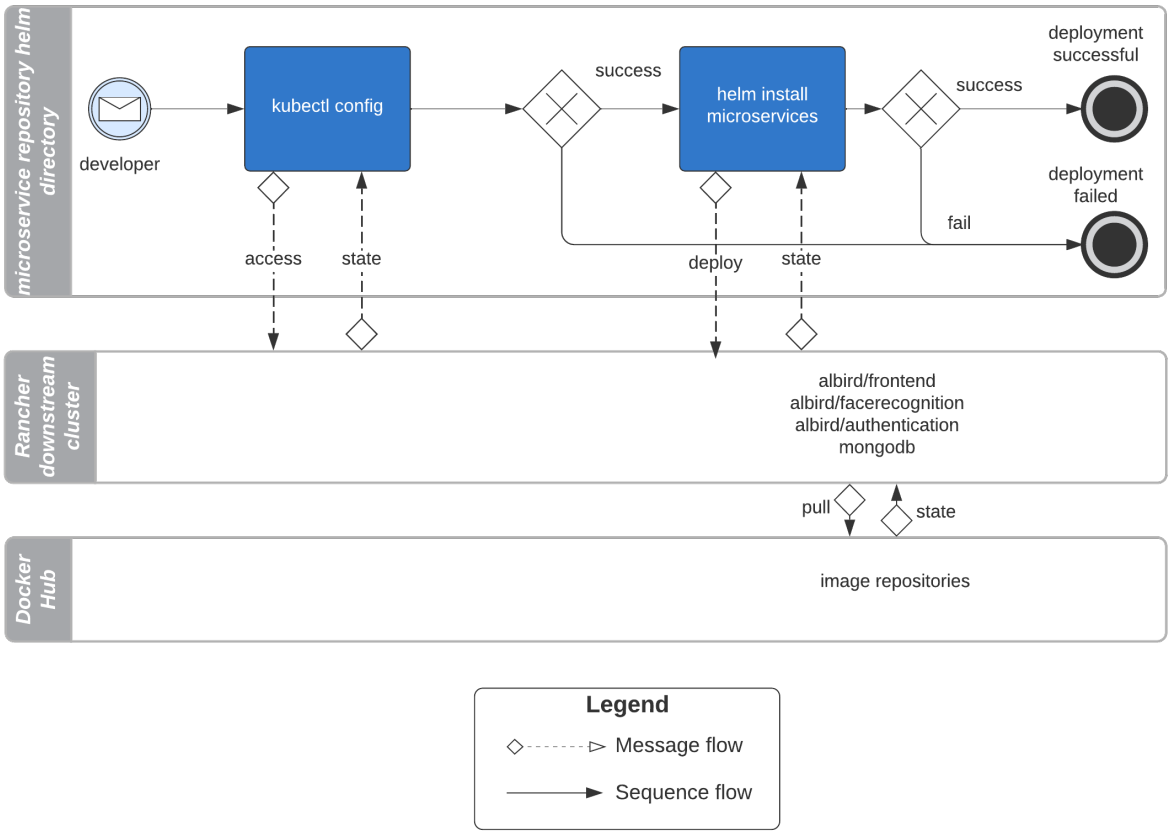

Abbildung 5.4: BPNM Modell - Helm–Installation der Microservices

Kapitel 6

Umsetzung des Lösungskonzepts

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

6.1 Konfiguration und Einrichtung

In diesem Abschnitt geht es um die Einrichtung und Vorkonfiguration des Kubernetes-Clusters mit der Distribution k3s. Zuerst folgt die Einrichtung der einzelnen virtuellen privaten Server, die als Knotenpunkte in unserem Kubernetes-Cluster fungieren. Danach die Konfiguration einer Domain für den späteren Einsatz der Microservices. Abschließend erfolgt die Bereitstellung von Zertifikaten für die verschlüsselte Kommunikation über die Domain.

Virtueller privater Server

Durch die Einschränkungen, beschrieben in Abschnitt 7.2, wird die Installation von Kubernetes auf Virtual Private Server (VPS)-Instanzen stattfinden. Ein VPS wird von Drittanbietern als Internet-Hosting-Dienst angeboten und ermöglicht die Vermietung von Rechenleistung. Die Server dienen als Knotenpunkte im Kubernetes-Cluster. Es werden ingesamt 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 Ver-

änderung des A-Records werden alle Anfragen der Domain auf eine IPv4-Adresse umgeleitet [61]. Die IPv4-Adresse ist in diesem Fall die Server-Node 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 automatischen mit einem Kubernetes-Issuer erstellt werden und von einem Ingress referenziert werden [62].

Issuer

Das Add-On Cert-Manager ist bereits auf dem Kubernetes-Cluster vorinstalliert und ermöglicht die Verwaltung von Zertifikaten. Cert-Manager enthält die Kubernetes-Resource Issuer, welche zur Generieriung von privaten Schlüsseln dient. Cert-Manager erlaubt die vereinfachte Bereistellung von Secure Sockets Layer (SSL)-Zertifikaten durch das Automatic Certificate Management Environment (ACME) 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 der Server-Node in Abschnitt 6.1. Die Ausführung des Issuers erzeugt einen privaten Schlüssel mit der Bezeichnung letsencrypt-key und dem Kubernetes-Issuer namens letsencrypt-prod. Für die Generieriung des Schlüssels wird die offene Zertifizierungsstelle Let's Encrypt verwendet [64] (vgl. Quellcode 6.1).

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-key
    solvers:
    - http01:
        ingress:
          class: nginx
```

Quellcode 6.1: issuer.yaml [63]

Cert

Der nächste Schritt ist die Erzeugung eines Zertifikats mit dem vorher erstellten Issuer. Die Ausführung der Cert-YAML-Datei erstellt ein signiertes Zertifikat. Dafür ist die Domain mit dem Eintrag der Server-Node und der Bezeichnung des Issuers notwendig. Das erzeugte Secret mit der Bezeichnung deploy-secret kann von einem Ingress zur Verschlüsselung der Kommunikation verwendet werden (vgl. Quellcode 6.2).

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: cert-prod
spec:
  secretName: deploy-secret
  issuerRef:
    name: letsencrypt-prod
  dnsNames:
  - "example-domain.com"
```

Quellcode 6.2: cert.yaml [63]

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 [65]. In einem hybriden Kubernetes-Cluster kann somit die Unterteilung von Labels in Cloud- und On-Premise-Hardware erfolgen (vgl. Quellcode 6.3).

```
kubectl label nodes microservice0 hardware=cloud
kubectl label nodes microservice1 hardware=cloud
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 und Tolerations

Taints und Tolerations stellt sicher, dass Pods nicht auf einen ungeeigneten Knoten eingeplant oder ausgeführt werden. Ein Taint dient zur Markierung von Nodes. Demnach akzeptieren diese nur Pods mit der richtigen Toleration. Für die erforderliche Nutzung von GPU-Nodes einer Anwendung können auch diese gekennzeichnet werden [66].

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

Quellcode 6.4: Node-Taints

Damit werden nur Pods auf der Node *microservice2* eingeplant, die als Toleration den Schlüssel *hardware*, Wert *gpu* und dem Effekt *NoSchedule* (vgl. Quellcode 6.4) besitzt. Bereits auf der Node laufende Pods sind davon nicht betroffen, dies erfordert den Taint *NoExecute* [66].

Die Ausführung von Node-Affinity und Taints und Tolerations ermöglicht nun die eindeutige 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 Label bestimmen.

6.2 Gesichtserkennung

In diesem Abschnitt werden die zwei Algorithmen der OpenCV-Bibliothek erläutert, welche zur Gesichtserkennung im Facerecognition-Service in Einsatz kommen.

6.2.1 Viola-Jones

Die Viola-Jones Methode zur Erkennung von Objekten verwendet einen Klassifizierer. Ein Klassifizierer verwendet Trainingsdaten, wie Datenpunkte, um den Zusammenhang von Eingabewerten im Kontext des Klassifizierers zu verstehen. Für die Erkennung von Bildern mit Gesichtsmerkmalen wird dieser, mit positiven Bildern (mit Gesicht) und negativen Bildern (ohne Gesicht) trainiert. Dabei werden die benötigten Gesichtsmerkmale aus den Bildern extrahiert. Dafür wird ein Haar-Feature verwendet, das ähnlich wie ein Image-Kernel eine kleine Matrix darstellt und über die einzelnen Pixelwerte eines Bildes fährt und diese mit dem Kernel multipliziert. Dadurch werden Merkmale, wie Augen, Mund und Nase, klassifiziert. Für die schnelle Berechnung wird ein Cascade-Classifier verwendet. Bei der Klassifikation werden Bilder in Regionen aufgeteilt, die dann bei der Erkennung von Merkmalen weiter aufgeteilt werden. Wenn nicht, wird die Region übersprungen und die nächste berechnet, um Redundanz zu vermeiden. OpenCV bietet hierfür bereits vortrainierte Modelle zur Erkennung von Gesichtsmerkmalen, die im Facerecognition-Service verwendet werden [67].

6.2.2 Local Binary Patterns Histogram (LBPH)

Für die Gesichtserkennung wird der Algorithmus LBPH verwendet. Dieses fasst die lokalen Regionen in einem Bild zusammen, indem jeder Pixel mit Nachbarpixeln verglichen wird. Dabei wird ein Local Binary Patterns (LBP)-Operator verwendet, das eine Region von einer 3x3-Matrix darstellt. Das Pixel im Zentrum gilt als Schwellwert zur Berechnung der benachbarten Pixel. Wenn der Wert eines Nachbarpixels gleich oder größer als der Wert des Pixels im Zentrum ist, wird die Pixelposition mit dem

Wert 1 markiert. Andernfalls wird der Pixel mit einer 0 markiert [68] (vgl. Abbildung 6.1).

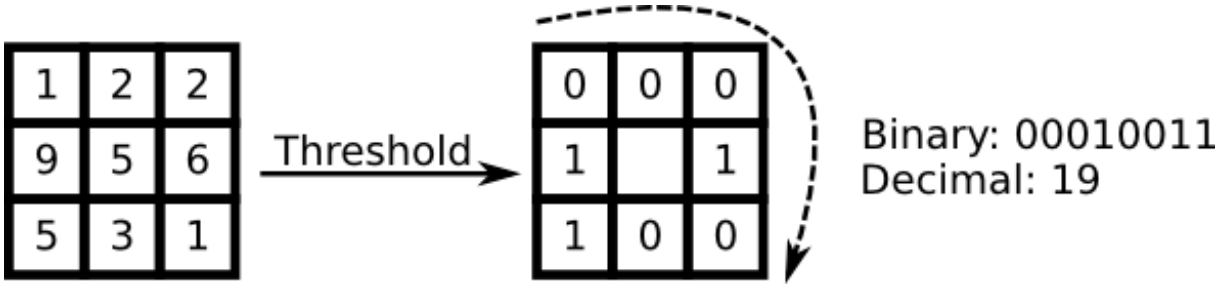

Abbildung 6.1: Local Binary Patterns - Operator [68]

Für die Klassifizierung der räumlichen Information eines Bildes werden die lokalen Regionen als ein Histogram extrahiert, welches die Intensität der einzelnen Pixel ausgibt. Dies führt dazu, dass Merkmale auch bei schlecht belichteten Bildern erkannt werden und weniger Fehlern bei der Gesichtserkennung mit dem LBPH stattfinden [69].

6.3 KubeVision

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

6.3.1 Frontend-Service

Frontend-Service ist die Benutzeroberfläche zur Interaktion mit dem Benutzer. Der Dienst ist aus mehreren 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. Der erste befindet sich im Verzeichnis home und stellt die Funktionalitäten und Authentication-Service-Endpunkte für das Anmelden, Registrieren und Anzeigen des Profils bereit (vgl. Abbildung 6.2). Der zweite Blueprint besteht aus mit Views zur Interaktion mit dem Facerecognition-Service, welche erst nach einer erfolgreichen Anmeldung aufrufbar sind. Der Benutzer wird mit seinem Namen begrüßt und erhält Anweisungen für die Erstellung von Bildern für den Facerecognition-Service. Für die Nutzung des Facerecognition-Services wird die Kamera des Benutzers benötigt. Die eingebundene JavaScript-Bibliothek SocketIO ermöglicht

das Senden von Bildern, die von einer Webcam aufgenommen wurden. Das Modul und die enthaltenen Funktionen in der Datei *Camera.js* ist für die Verwendung der Webcam zuständig. Die Funktion *navigator.mediaDevices.enumerateDevices()* listet alle angeschlossenen Peripheregeräte mit Kamerafunktion auf. Diese Geräte werden dann in eine Dropdown-Liste platziert. Der Nutzer kann danach eine spezifische Kamera auswählen (vgl. Abbildung 6.3).

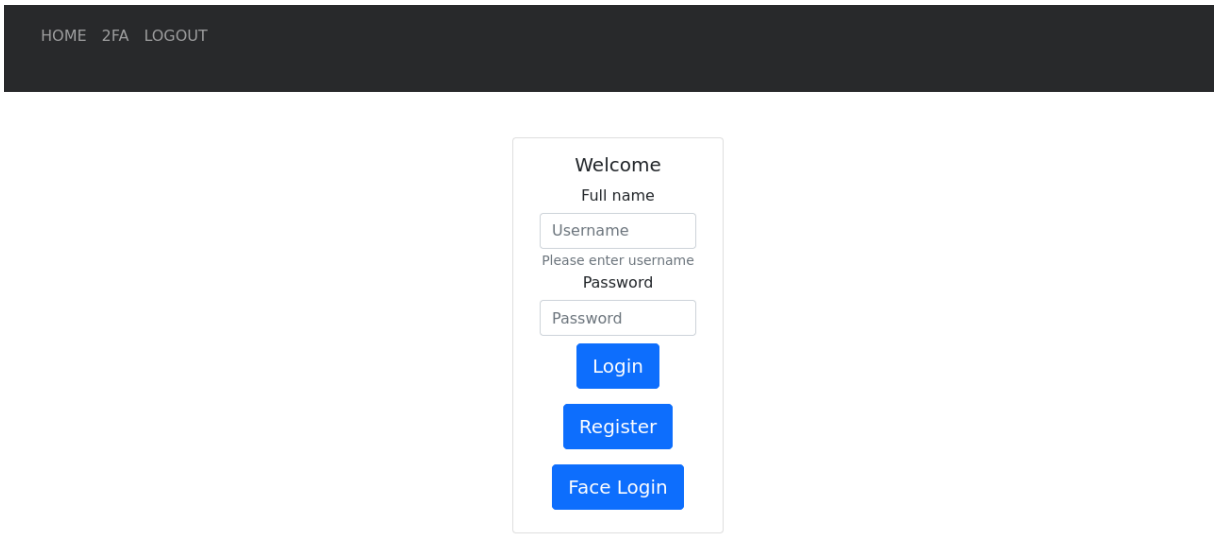

Abbildung 6.2: Frontend-Service - Home

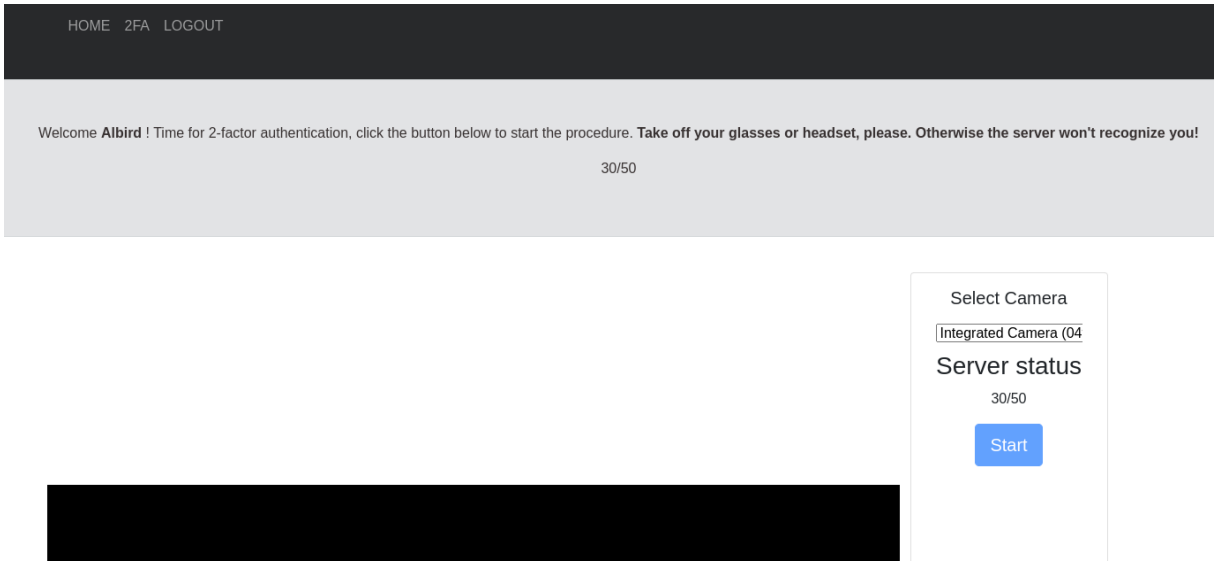

Abbildung 6.3: Frontend-Service - Train

Mit dem JavaScript-Modul *socketio.js* lässt sich die bidirektionale Kommunikation mit dem Facerecognition-Service aufbauen. Es gibt drei unterschiedliche Events für die Kommunikation mit dem Dienst. Das Event *stream* sendet 50 Bildern an den Dienst und löst im Anschluss ein Server-Event aus, das die Bilder des Benutzers auswertet. Diese Interaktion ist über den Endpunkt */train* möglich. 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.

6.3.2 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. Ein Blueprint stellt die API über dem Endpunkt */auth* bereit. Die Routen des Endpunktes */auth* erhalten bis auf */auth/logout* ein POST-Objekt über den Frontend-Service. In diesem steht der Name und das Passwort des Benutzers. Damit kann sich ein Benutzer einen Account erstellen oder sich anmelden. Der Dienst erstellt bei Registrierung einen Eintrag in die MongoDB-Datenbank oder liest diese aus. Die Speicherung der Passwörter erfolgt in Form eines Hashes. Bei erfolgreichem Login wird ein Cookie mit dem Benutzernamen gesetzt und der Benutzer wird in das Homemenü weitergeleitet (vgl. Abbildung 6.4).

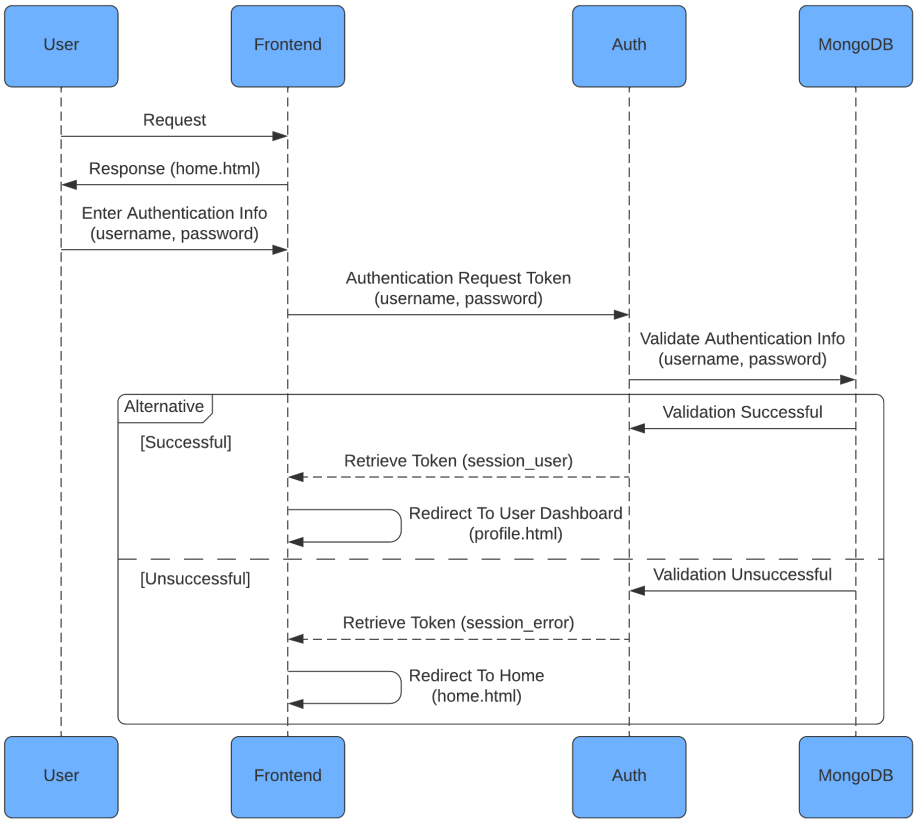

Abbildung 6.4: Ablauf von der Authentifizierung eines Benutzers

Hier kann der angemeldete 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 das Benutzer-Cookie gelöscht und der Benutzer wieder auf das Homemenü weitergeleitet.

6.3.3 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 einen Blueprint realisiert. Über den Endpunkt *socketio* ist die eventbasierte Kommunikation zwischen Frontend-Service Benutzer und Facerecognition-Service möglich. Das Event *stream*, nimmt Bilder im webp-Format 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 durchläuft 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 (vgl. Quellcode 6.5).

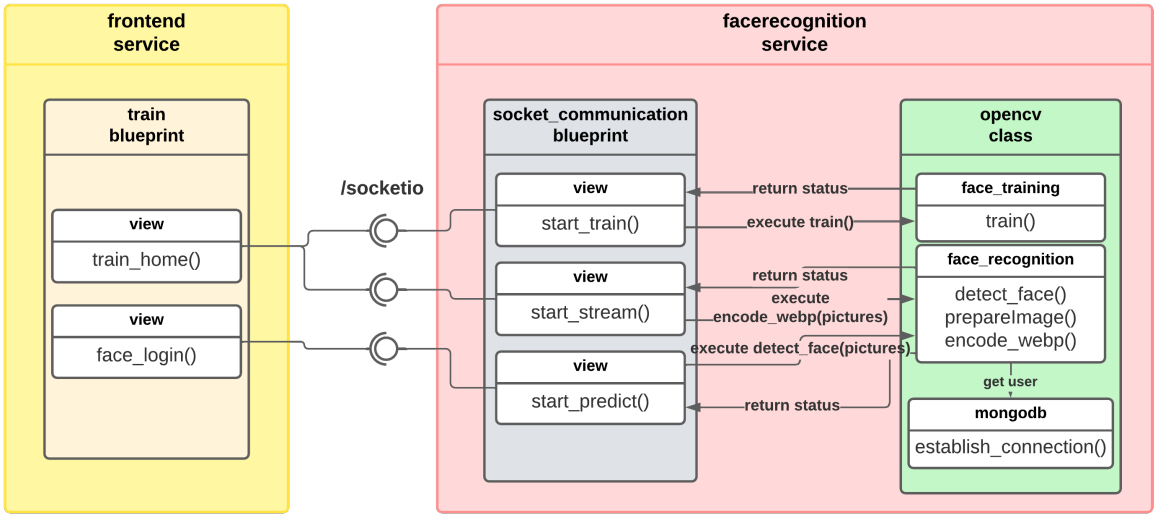

Abbildung 6.5: Komponentendiagramm der Facerecognition-Service Kommunikation

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 des Authentication-Service genutzt.

6.4 Dockerisierung

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

```
FROM python:3.7.2-stretch

WORKDIR /app
ADD . /app

RUN apt-get update
RUN apt-get install ffmpeg libsm6 libxext6  -y
RUN pip install --upgrade pip setuptools wheel
RUN pip install -r requirements.txt

ENV PYTHONUNBUFFERED 1
EXPOSE 5000

CMD ["gunicorn" , "-k"
   ,"geventwebsocket.gunicorn.workers.GeventWebSocketWorker", "-w", "3"
   , "--bind" , ":5000" , "run:app"]
```

Quellcode 6.5: Dockerfile

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.

Um Redundanz zu vermeiden, wird im Folgenden das Dockerfile zum Facerecognition-Service näher erläutert (vgl. Quellcode 6.5). Dieses 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 Datei requirements.txt stehen alle Bibliothekennamen mit der erforderlichen Version. Die umgebungsvariablen ermöglicht die Ausgabe des Python-Buffers im Terminal. Die *CMD*-Anweisung weist Docker an, den Containers immer mit dem Befehl, einen Gunicorn-Webserver auszuführen, zu starten. Die zusätzlichen Flags geben die Art und Anzahl der Worker-Prozesse an. Letztendlich wird die Webanwendung mit der WSGI-Schnittstelle an dem gewählten Port 5000 ausgeführt.

6.4.2 Docker-Compose

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

```yaml
authentication:
  build: ./authentication
  image: albird/authentication:latest
  environment:
    homeEndpoint: http://localhost:8000/
    trainEndpoint: http://localhost:8000/train
    mongoEndpoint: mongodb://admin:password@localhost:27017/
  ports:
    - "5001:5001"
  command: gunicorn -w 1 -b 0.0.0.0:5001 run:app

mongo:
  image: mongo:4.1.7
  environment:
    MONGO_INITDB_ROOT_USERNAME: admin
    MONGO_INITDB_ROOT_PASSWORD: password
  ports:
    - "27017:27017"
  volumes:
    - ./mongo-volume:/data/db
```

Quellcode 6.6: Ausschnit aus dem docker-compose.yaml

Der Befehl build gibt die Docker-Anweisung zum Bauen eines Images anhand einer vorhanden Dockerfile-Datei. Die Webanwendung wurde mit dem Einsatz von umgebungsvariablen entwickelt. Diese können für eine flexible Bereistellung der Dienste verwendet werden, um Endpunkte über String-Variablen in der Webanwendung zu ändern. Ports gibt die Ports an, auf denen 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 umgebungsvariablen zur Übergabe von Passwort und Benutzernamen 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. 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.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.face.service.name }}
  namespace: {{ .Values.namespace }}
spec:
  type: {{  .Values.face.service.type }}
  ports:
    - port: {{  .Values.face.image.port }}
      targetPort: {{  .Values.face.image.port }}
      protocol: TCP
      name: http
  selector:
    server: {{  .Values.face.name }}
```

Quellcode 6.7: face-service.yaml

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.

Der Name des Kubernetes-Service ist für die spätere Angabe im Ingress notwendig. *Type* definiert den Service-Typ 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, über den der 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).

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 der Server-Node verwendet. In der TLS-Einstellung wird noch das TLS-Zertifikat als Secret referenziert. Jedem Service wird ein Endpunkt zugewiesen. Der Authentication-Service ist über das Präfix *auth* erreichbar. Facerecognition-Service erhält den Endpunkt *socket.io* zur Kommunkation mithilfe der gleichnamigen Bibliothek. Der Frontend-Service ist über den Hostnamen erreichbar. Für die Pfade wird kein Präfix wie bei dem Authentication-Service benötigt.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: kubevision-ingress
 annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - {{ .Values.envEndpoint.host }}
    secretName: deploy-secret
  rules:
  - host: {{ .Values.envEndpoint.host }}
    http:
      paths:
      - backend:
          service:
            name: {{ .Values.auth.service.name }}
            port:
              number: {{ .Values.auth.image.port }}
        path: /auth
        pathType: Prefix
      - backend:
          service:
            name: {{ .Values.face.service.name }}
            port:
              number: {{ .Values.face.image.port }}
        path: /socket.io
        pathType: Prefix
      - backend:
          service:
            name: {{ .Values.frontend.service.name }}
            port:
              number: 80
        path: /
        pathType: Prefix
  ingressClassName: nginx
```

Quellcode 6.8: kubevision-ingress.yaml

6.5.3 Deployment

Die grundlegende Bereitstellung der Dienste erfolgt mit einem Deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.frontend.name }}
  namespace: {{ .Values.namespace }}
spec:
  replicas: {{ .Values.frontend.replicas }}
  selector:
    matchLabels:
      server: {{ .Values.frontend.name }}
  template:
    metadata:
      labels:
        server: {{ .Values.frontend.name }}
    spec:
      containers:
        - name: {{ .Values.frontend.name }}
          image: {{ .Values.frontend.image.name }}:{{ .Values.frontend.image.tag }}
          imagePullPolicy: Always
          ports:
            - containerPort: {{ .Values.frontend.image.port }}
          env:
          - name: loginEndpoint
            value: https://{{ .Values.envEndpoint.host }}/auth/login
          - name: registerEndpoint
            value: https://{{ .Values.envEndpoint.host }}/auth/register
          - name: websocketServer
            value: https://{{ .Values.envEndpoint.host }}
          - name: homeEndpoint
            value: https://{{ .Values.envEndpoint.host }}/
      nodeSelector:
        hardware: {{ .Values.frontend.nodeSelector.hardware }}
```

Quellcode 6.9: frontend-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 zur Ausführung auf einem oder mehreren Pods angegeben. 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 Hostssystem befindet. *Ports* gibt die lauschenden Ports des Containers im Pod an. Die umgebungsvariablen definieren die 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 PersistentVolumes

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

Ein PVC benötigt 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 [70]. 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: mongodb-pvc.yaml

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* [71] (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

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Das Ziel dieser Arbeit war die Konzeption und Implementierung einer Microservice-Architektur auf einem hybriden Kubernetes-Cluster für Anwendungen im Bereich der künstlichen Intelligenz. Die Vorgehensweisen bei der Implementierung und Entwicklung der Architektur soll als Blaupause für weitere Konzepte der Krones AG dienen. Die Entwicklung der Microservices basierte auf Containervirtualisierung und der Containerplattform-Kubernetes. Der PoC beschreibt die Modernisierung der Infrastruktur durch Virtual-Edge-Devices. Auf den Industrierechnern in Produktionsanlagen soll das Betriebssystem *Windows 10* und *SUSE Linux Enterprise Micro* auf einem Hypervisor gleichzeitg ausgeführt werden. Dabei stellt das Linux-System ein Virtual-Edge-Device dar, dass als Knotenpunkt im Kubernetes-Cluster fungiert. Im PoC wurde festgestellt, dass On-Premise und Cloud-Technologien in Verbindung mit GPUs für Anwendungsfälle im Bereich der künstlichen Intelligenz eine Verwendung finden.

Für diese Umsetzung erfolgt die Verwaltung und Überwachung des Kubernetes-Clusters mit der Orchestierungsplattform Rancher. Außerdem wird zur Installation der Microservices auf dem Kubernetes-Cluster der Package-Manager Helm verwendet. Das Anwendungsszenario war eine Webanwendung aus losen Diensten die miteinander kommunizieren und unabhängig eingesetzt werden können. Der hauptsächliche Anwendungsfall wurde durch ein Authentifierzungsverfahren mit Gesichtserkennung realisiert. Die Softwarekomponenten der Webanwendung können erfolgreich miteinander kommunizieren. Dazu gehört die Entwicklung der Anwendung, als Docker-Container und das Schreiben der Helm-Charts, sowie dessen Vorkonfiguration. Anforderungen wie automatische Tests der Dienste konnte wegen Zeitmangel und Fokus auf die Implmentierung der Anwendung nicht mehr realisiert werden. Die Anwendung für die Nutzung einer GPU zu entwicklen konnte wegen der komplexen Architektur eines solchen Docker-Containers nicht realisiert werden. Die Installation und Organisation von containerisierten Arbeitslasten durch die Rancher-Plattform erleichterte die Über-

wachung der Anwendungen. Durch die Benutzeroberfläche wurde die Komplextität 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 virtuellen privaten Servern 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 nur im Bereich der Anlagentechnik genutzt wird und nicht zur Authentifizierung von Personal. Jedoch sind viele der Schritte ähnlich ausführbar wie auf einem Kubernetes-Cluster mit On-Premise Geräten anstatt cloudbasierter Hardware. Der blaupausenartige Aufbau der Entwicklungsschritte gilt auch für den Aufbau von Microservices im Produktionsumfeld.

7.3 Ausblick

Mit der Implementierung aus Kapitel 6 wäre es lohnenswert Microservices auf Industrierechnern in einem echten Kundenumfeld zu testen. Dafür muss auch geprüft werden, ob das Kubernetes-Cluster sich mit On-Premise-Hardware gleich verhält wie mit Servern aus der Cloud. Auch die Verwaltung eines hybriden-Kubernetes-Clusters, welches zwischen dem Standort von Hardware unterscheidet, benötigt eine nähere Untersuchung. Die Umsetzung aus Abschnitt 6.5 kann dabei für die Installation der Dienste mit Helm verwendet werden. Mit den erzeugten Daten der Produktionsanlage kann dann eine prototypische Anwendung zur Auswertung von realen Anwendungsfällen mit Bezug zur künstlichen Intelligenz realisiert werden.

Abkürzungsverzeichnis

PoC Proof of Concept

VM Virtuelle Maschine

SHA Secure Hash Algorithm

API Application Programming Interface

RPC Remote Procedure Call

VPS Virtual Private Server

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 Machine 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

SSL Secure Sockets Layer

TLS Transport Layer Security

ACME Automatic Certificate Management Environment

PVC PersistentVolumeClaim

PV PersistentVolume

LBP Local Binary Patterns

LBPH Local Binary Patterns Histogram

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?" Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/

[11] "Kubernetes components," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/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," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/

[14] "Understanding kubernetes objects," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[15] "Deployments," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[16] N. Poulton, *The Kubernetes Book*, 2021st ed. [Germany]: Nigel Poulton, 2021.

[17] "Service," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[18] "Ingress," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/

[19] "Ingress controllers," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

[20] "Layer 4 and Layer 7 Load Balancing," Feb. 2022. [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/k8s-in-rancher/load-balancers-and-ingress/load-balancers/

[21] "What is kubernetes ingress?" Feb. 2022. [Online]. Available: https://www.ibm.com/cloud/blog/kubernetes-ingress

[22] "Raspberry pi documentation - processors," Feb. 2022. [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html

[23] "K3s resource profiling," Feb. 2022. [Online]. Available: https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/resource-profiling/

[24] "K3s: Lightweight kubernetes," Jul. 2021. [Online]. Available: https://k3s.io/

[25] "K3s - lightweight kubernetes," Feb. 2022, original-date: 2018-05-31T01:37:46Z. [Online]. Available: https://github.com/k3s-io/k3s

[26] "Possible to run k3s on one node (server and agent together)? · Issue #1279 · k3s-io/k3s," Jan. 2020. [Online]. Available: https://github.com/k3s-io/k3s/issues/1279

[27] "flannel," Feb. 2022, original-date: 2014-07-10T17:45:29Z. [Online]. Available: https://github.com/flannel-io/flannel

[28] "Overview," Feb. 2022. [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/

[29] 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

[30] "Access a Cluster with Kubectl and kubeconfig," Feb. 2022. [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/cluster-admin/cluster-access/kubectl/

[31] "Overview," Feb. 2022. [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/

[32] "Architecture Recommendations." [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/architecture-recommendations/

[33] "Rancher Agents," Feb. 2022. [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/cluster-provisioning/rke-clusters/rancher-agents/

[34] "hybrid-cloud," May 2021. [Online]. Available: https://www.ibm.com/de-de/cloud/learn/hybrid-cloud

[35] "Microservices," Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[36] "Microservices Pattern: Decompose by business capability," Feb. 2022. [Online]. Available: http://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[37] "Softwarecomponent," Sep. 2015. [Online]. Available: https://martinfowler.com/bliki/SoftwareComponent.html

[38] S. Newman, *Implementing Microservice Communication*, 2nd ed. Sebastopol, CA: O'Reilly Media, Sep. 2021.

[39] M. Richards, *Microservices vs. Service-Oriented Architecture*. O'Reilly UK, Apr. 2016.

[40] S. Newman, *Building microservices*, 2nd ed. Sebastopol, CA: O'Reilly Media, Sep. 2021.

[41] "A Conversation with Werner Vogels - ACM Queue," Jun. 2006. [Online]. Available: https://queue.acm.org/detail.cfm?id=1142065

[42] "Protocol Buffers | Google Developers," Feb. 2022. [Online]. Available: https://developers.google.com/protocol-buffers

[43] "Boundedcontext," Jan. 2015. [Online]. Available: https://martinfowler.com/bliki/BoundedContext.html

[44] "Zwei Betriebssysteme auf einem Gerät | B&R Industrial Automation." [Online]. Available: https://www.br-automation.com/

[45] "RTS Hypervisor - Hardware partitioning - Real-Time Systems," Feb. 2022. [Online]. Available: https://www.real-time-systems.com/de/use-cases/rts-hypervisor-hardware-partitioning.html

[46] "Connected HMI: die neue Generation der Maschinenvisualisierung - Krones," Feb. 2022. [Online]. Available: https://www.krones.com/de/produkte/innovationen/maschinenvisualisierung-connected-hmi.php

[47] "Helm Architecture," Feb. 2022. [Online]. Available: https://helm.sh/docs/topics/architecture/

[48] "Welcome to Flask — Flask Documentation (2.0.x)," Feb. 2022. [Online]. Available: https://flask.palletsprojects.com/en/2.0.x/

[49] G. C. Hillar, *Hands-On RESTful Python Web Services*, 2nd ed. Birmingham, England: Packt Publishing, Dec. 2018.

[50] "Design — Gunicorn 20.1.0 documentation," Feb. 2022. [Online]. Available: https://docs.gunicorn.org/en/latest/design.html

[51] "OpenCV: Introduction," Feb. 2022. [Online]. Available: https://docs.opencv.org/3.4/d1/dfb/intro.html

[52] 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

[53] "OpenCV: Introduction to OpenCV-Python Tutorials," Feb. 2022. [Online]. Available: https://docs.opencv.org/4.x/d0/de3/tutorial_py_intro.html

[54] 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.

[55] "Introduction | Socket.IO," Feb. 2022. [Online]. Available: https://socket.io/docs/v4/

[56] "python-socketio — python-socketio documentation," Feb. 2022. [Online]. Available: https://python-socketio.readthedocs.io/en/latest/

[57] 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

[58] E. B. . K. C. Shannon Bradshaw, *MongoDB: The Definitive Guide, 3rd Edition*. O'Reilly Media, 2019. [Online]. Available: https://learning.oreilly.com/library/view/mongodb-the-definitive/9781491954454/

[59] "Pricing · Plans for every developer," Feb. 2022. [Online]. Available: https://github.com/pricing

[60] nishanil, "Entwicklungsworkflow für Docker-Apps," Jan. 2022. [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," Feb. 2022, section: docs. [Online]. Available: https://cert-manager.io/docs/concepts/issuer/

[63] "ACME," Feb. 2022. [Online]. Available: https://cert-manager.io/docs/configuration/acme/

[64] "Über Let's Encrypt - Let's Encrypt - Freie SSL/TLS Zertifikate," Feb. 2022. [Online]. Available: https://letsencrypt.org/de/about/

[65] "Assign Pods to Nodes using Node Affinity," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/

[66] "Taints and Tolerations," Feb. 2022, section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

[67] "OpenCV: Cascade Classifier," Mar. 2022. [Online]. Available: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[68] "OpenCV: Face Recognition with OpenCV," Mar. 2022. [Online]. Available: https://docs.opencv.org/3.4/da/d60/tutorial_face_main.html#tutorial_face_lbph_algo

[69] X. Zhao and C. Wei, "A real-time face recognition system based on the improved lbph algorithm," in *2017 IEEE 2nd International Conference on Signal and Image Processing (ICSIP)*, 2017, pp. 72–76.

[70] "Chart Development Tips and Tricks," Feb. 2022. [Online]. Available: https://helm.sh/docs/howto/charts_tips_and_tricks/

[71] "Manage mongod Processes — MongoDB Manual," Feb. 2022. [Online]. Available: https://docs.mongodb.com/manual/tutorial/manage-mongodb-processes/

Abbildungsverzeichnis

Quellcodeverzeichnis