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: 17. Februar 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: Ottmar Amann
Ausgabedatum: 4. Oktober 2021 Abgabedatum: 3. März 2022

Titel:

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

---

Zusammenfassung:

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

Schlüsselwörter:

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Krones AG bietet Anlagen sowohl für die Getränkeindustrie als auch Nahrungsmittelhersteller, von der Prozesstechnik bis hin zur IT-Lösung. Die Komplettlinie beeinhaltet auch das Bereitstellen von Software auf den einzelnen Produktionsanlagen. Hierfür werden eine Vielzahl von Produktionslinienanwendungen auf den Anlagen installiert, gewartet und verwaltet. Dementsprechend hoch ist der Aufwand, der Fehleranfälligkeiten sowie fehlende Frameworks, Bibliotheken und anderer Abhängigkeiten mit sich bringt. Eigene Server müssen für die Kommunikation der Anlagen verbaut und gewartet werden, was zusätzlich Ressourcen beansprucht und automatisch die Kosten für die Inbetriebnahme einer solchen Linie erhöhen. Die Weiterentwicklung der zukünftigen Bereitstellung von Produktionsanlagensoftware erfolgt mithilfe eines Proof of Concept (PoC), welcher die Möglichkeiten einer wartungsfreien Infrastruktur durch ein continuous Delivery System evaluiert. Dies verläuft in Zusammenarbeit mit dem Kooperationspartner und 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 Edge Devices als Knotenpunkte in einem Kubernetes-Cluster fungieren, Ressourcen teilen, untereinander kommunizieren und Softwarepakete unkompliziert bereitstellen. Die Integration der kompakten Linux-Rechner ermöglichen den variablen Einsatz von Hardwareressourcen des Kunden, der je nach Leistungsanspruch Knotenpunkte erweitern kann. Dabei soll es für die einzelnen Anwendungen möglich sein, sowohl auf cloudbasierten als auch auf on-premise Hardware zur Verfügung gestellt zu werden. Ein hybrides Kubernetes-Cluster ermöglicht es somit, lokale Rechenleistung oder öffentliche Cloudressourcen in der selben Softwareumgebung zu nutzen.

1.1 Motivation

Die Vorteile von Kubernetes und dem stetigen Paradigmenwechsel der Softwarelandschaft im Cloudbereich, welcher den Wechsel von monolithischen Architekturen zu flexibleren Microservice-Architekturen bevorzugt, sind das Hauptmotiv der Auswertung neuer, agiler Distributionsmöglichkeiten. Die Containerisierung von Anwendungen ermöglicht erst die Aufteilung großer Projekte in kleine unabängige Services, die mittels Orchestrierungsplattformen adäquat konzentiert 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 testen. Die Anlage Linatronic AI der Krones AG nutzt bereits Deep-Learning-Technologie, um in der Linie mittels Vollinspektion Schäden, Dichtflächen oder Seitenwanddicken zu erkennen und Prozesse zu optimieren [2]. Allgemein sind Anwendungen mit künstlicher Intelligenz durch ihre Komplexität und Vielzahl an Abhängigkeiten schwierig zu entwickeln und bereitzustellen. Eine passende Plattform für Anwendungsfälle mit Bezug zur künstlichen Intelligenz muss eine Vielzahl an Services anbieten. Zu diesen gehören die Verwaltung von Ressourcen wie Speicher, Rechenleistung und Verbindungsgeschwindigkeit, für die Datenübertragung bei der Ausführung einzelner Phasen von der Informationsverarbeitung bis hin zur Evaluierung und Entwicklung von Modellen im Bereich der künstlichen Intelligenz. [3].

1.2 Zielsetzung

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

Kapitel 2

Grundlagen

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

2.1 Docker

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

2.1.1 Architektur

Die Docker-Technologie ist in der Programmiersprache GO geschrieben und nutzt Funktionalitäten des Linux-Kernels, wie cgroups und namespaces. Namespaces ermöglichen die Isolation von Prozessen in sogenannte Container, welche unabhängig voneinander arbeiten [5]. Diese 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 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 sie zum Verständnis nicht nötig ist.

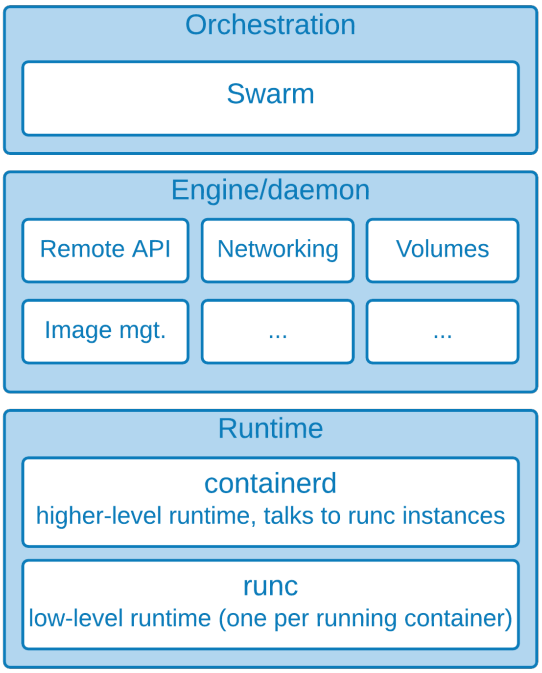

Abbildung 2.1: Docker Architektur in Anlehnung an [4, S.11]

2.1.2 Images und Container

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

Registrys

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

Schichten

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

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

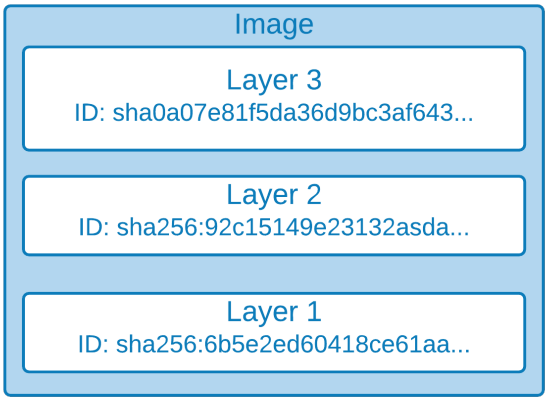

Abbildung 2.2: Image Layers in Anlehnung an [4, S.61]

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

Dockerfile

Zur Erstellung eines Docker-Images wird ein Dockerfile benötigt. Dies 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. Mit dem Unterschied, dass die Einrichtung von VMs arbeitsintensiver ist und weitaus mehr Ressourcen beansprucht, da sie ein ganzes Betriebssystem ausführt [8]. Containertechnologien bauen hingegen nur auf

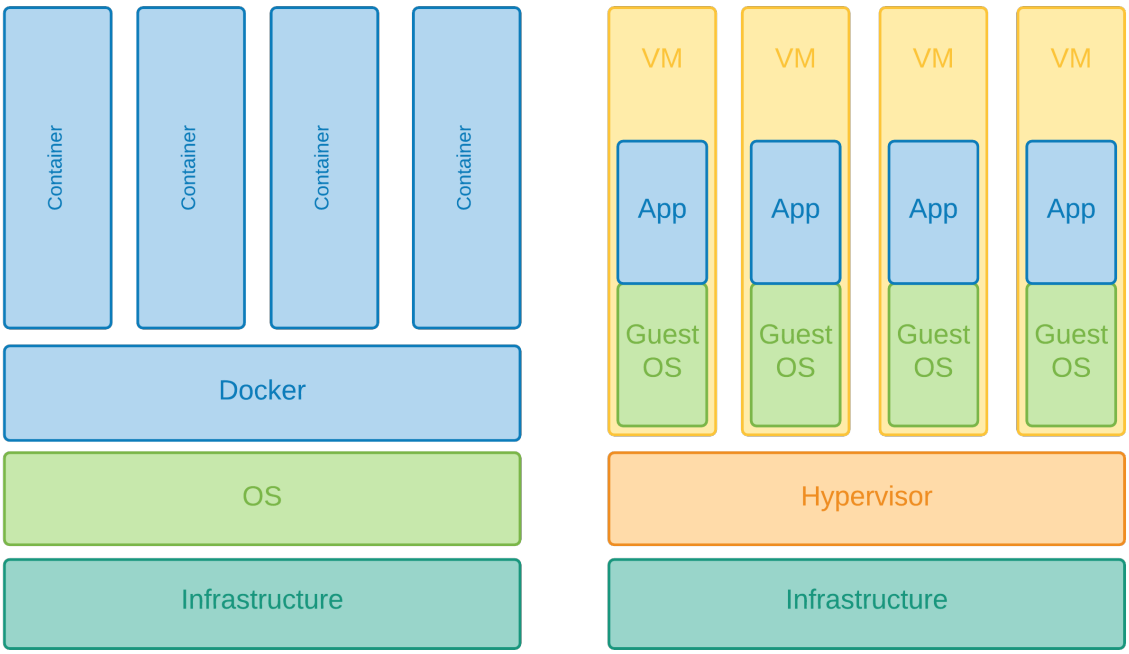

Abbildung 2.3: Virtualisierungsmöglichkeiten angelehnt an [9].

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

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

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

2.2 Kubernetes

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

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

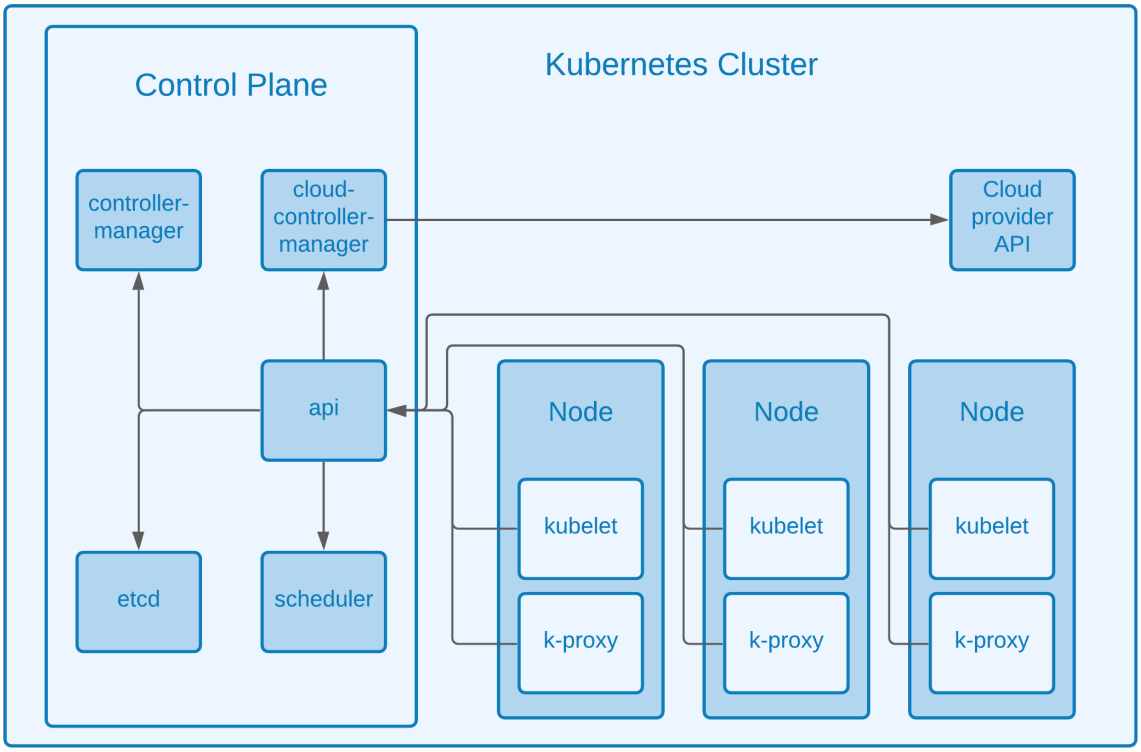

Abbildung 2.4: Komponenten eines Kubernetes Cluster in Anlehnung an [11].

2.2.1 Cluster

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

Control Plane

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

Kubelet

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

Kube-Proxy

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

2.2.2 Pods

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

2.2.3 Deployment

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

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

[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 [16]

Deployments und Pods

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

2.2.4 Service

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

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

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

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

```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 [18]

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

2.2.5 Ingress

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

NGINX-Ingress-Controller

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

- **ingressClassName**: definiert den Ingress-Controller.
- **rules**: die 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:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
```

Quellcode 2.3: ingress.yaml [19]

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, 1.50 GHz CPU[3] [24]. Der hauptsächliche Verwendungszweck von k3s liegt in IoT-Geräte, da sekundäre Kubernetes-Inhalte entfernt wurden. [25]. Trotz dieser Reduzierung bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden, soweit möglich, parallel auf dem neusten Stand gehalten [26].

Besonderheiten

Die Abbildung 2.5 zeigt die Architektur von k3s auf. Das Kubernetes-Äquivalent zur Control Plane und Node sind Server und Agent. Eine Besonderheit hiervon ist, dass Server parallel einen Agent-Prozess auf dem selben Knoten starten und somit Arbeitslasten mithilfe von Kubelet ausführen [27]. Weiterhin wird, im Gegensatz zu

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

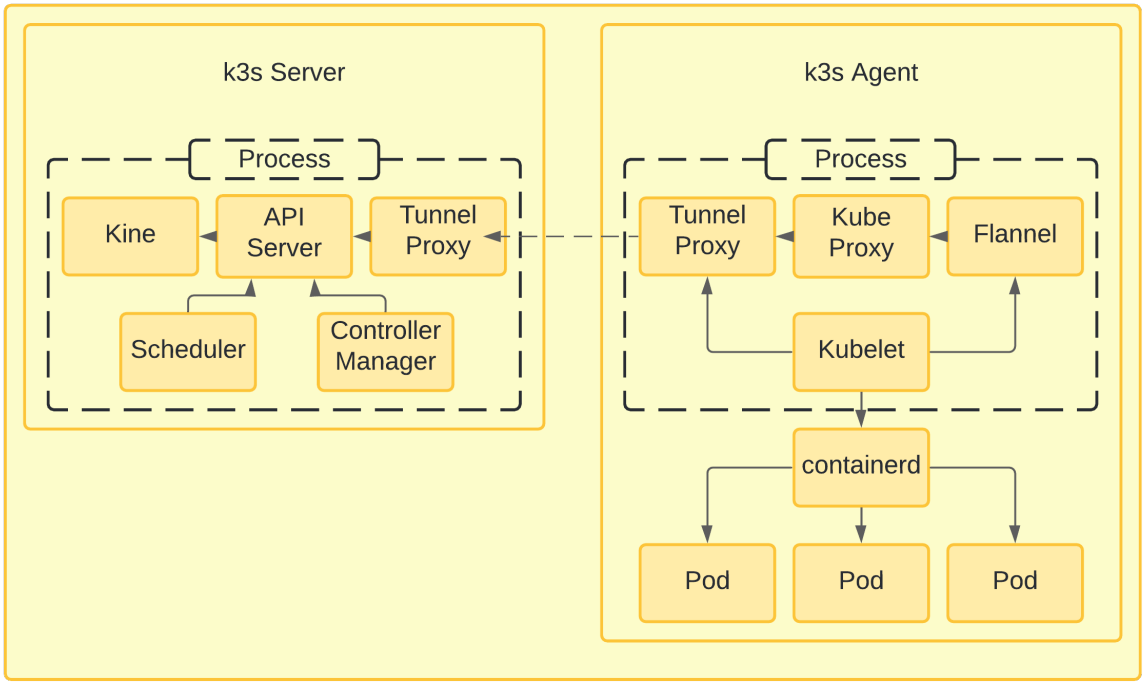

Abbildung 2.5: K3s Architektur in Anlehnung an [25].

Kubernetes, containerd weiterhin unterstüzt und kommt mit Kubelet vorinstalliert [25]. Zwei weitere Unterschiede werden näher erläutert:

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

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

2.2.7 Rancher

In diesem Unterabschnitt wird die Open-Source-Lösung Rancher von dem gleichnamigen Unternehmen zur 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 Cloud-Anbieter unabhängig, weshalb Cluster in der Praxis mit derselben Rancher Instanz auf AWS, Azure oder anderen Cloud-Anbietern betreut werden können [29].

Die Rancher-Benutzeroberfläche vereinfacht das Steuern von Arbeitslasten, auf einer

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

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

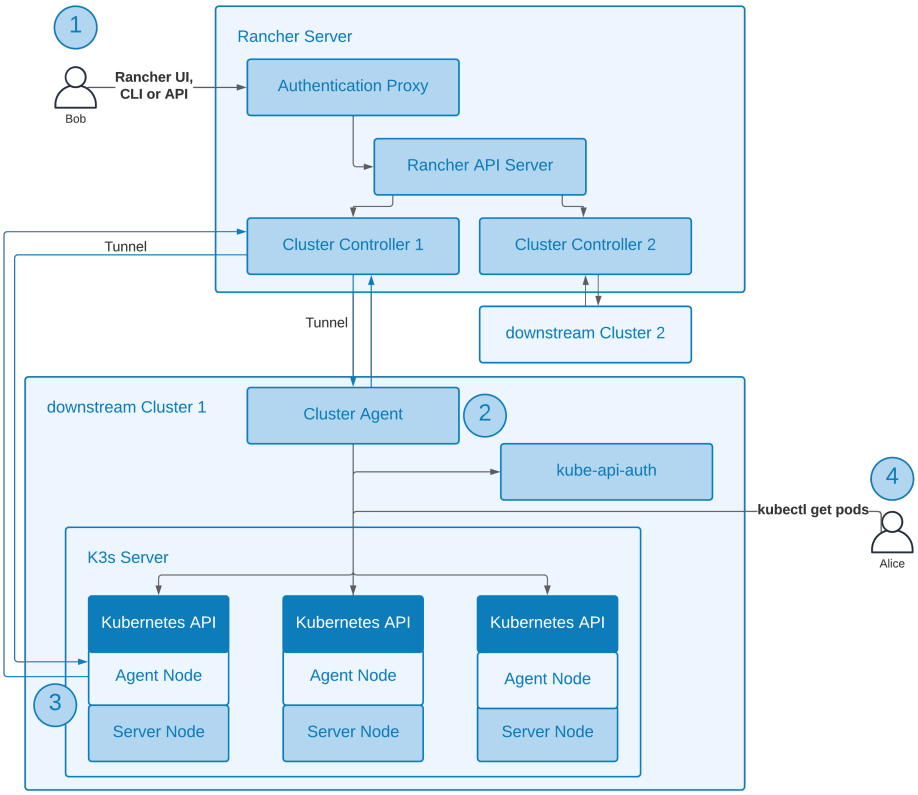

Abbildung 2.6: Rancher Server Kommunikation mit einem downstream k3s Cluster, überarbeitete Abbildung von [32]. (Im Sinne der späteren Architektur nachgebildet)

Die Abbildung 2.6 zeigt den Vorgang von zwei Benutzern, die auf ein von Rancher verwaltetes downstream-k3s-Cluster[4] zugreifen. Die nachfolgende Beschreibung aus der Dokumentation gibt die einzelnen Schritte mit der in der Abbildung nummerierten Posten wieder [32].

1. Zuerst authentifiziert sich Bob mit seinen Benutzerdaten bei dem Authentifizierungs-Proxy an seinem Rancher-Server. Dieser Proxy leitet den Aufruf über

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

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 einzigen Infrastruktur laufen. Dies ermöglicht die flexible Orchestrierung von Anwendungen auf Hostssystemen vor Ort oder in der Cloud [35].

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

2.3 Microservice

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

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

2.3.1 Begriffserklärung

Fowler und Lewis beschreiben den Microservice-Architektur-Stil als Entwicklung einer einzigen Anwendung, die aus einer Reihe unabhängiger Dienste besteht. Die Kommunikation der einzelnen Dienste untereinander wird häufig durch API-Aufrufe über HTTP realisiert. Diese Dienste sind vollautomatisch auszuliefern und orientieren sich bei der Entwicklung um Business-Capabilities[6]. Zusammenhängende Dienste werden minimal zentral gehalten und können in unterschiedlichen Programmiersprachen oder Technologien realisiert werden [36].

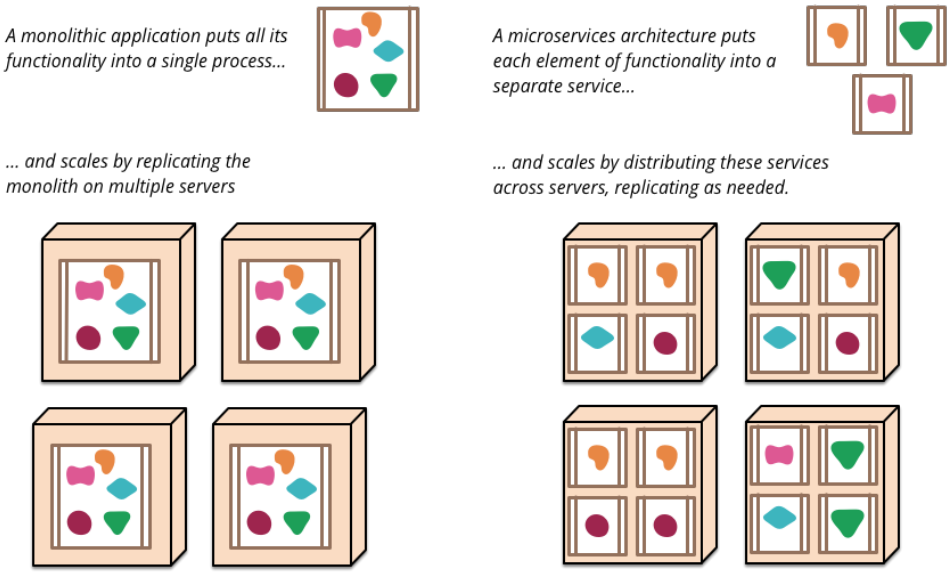

Abbildung 2.7: Gegenüberstellung von Monolithen und Microservices [36]

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

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

2.3.2 Charakteristiken

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

Komponententrennung durch Dienste

Fowler definiert Komponenten einer Software wie folgt:

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

Dienste einer Microservice-Architektur stellen Softwarekomponenten dar, die mittels Web-Service-Anfragen oder Remote-Procedure-Calls[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 [36].

Dadurch wird der Fokus auf die Entwicklung von unabhängigen Diensten umso wichtiger, da die Veränderung an kooperierenden Schnittstellen zum Ausfall anderer Dienste führt. Um dem entgegenzuwirken, müssen Schnittstellen gut koordiniert werden und eine starke Kohäsion gewährleisten. Service Contracts[8] der jeweiligen Dienste müssen sinnvoll gestaltet werden. Weiterhin müssen Schnittstellen grobkörniger entworfen werden, um den höheren Ressourcenverbrauch der lokalen Variante auszugleichen [36, 41].

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

Strukturierung nach Business-Capabilities

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

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

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

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

Produkte nicht Projekte

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

> „you build it, you run it “ [42].

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

Intelligente Endpunkte statt komplexer Infrastruktur

Die Kommunikation von Diensten über Endpunkte soll so weit wie möglich entkoppelt und kohäsiv sein. Anwendungen im Microservice-Stil enthalten ihre eigene Logik und agieren als Filter für das Empfangen, Verarbeiten und Beantworten einer Anfrage. Die Umsetzung erfolgt dabei mit RESTful-Protokollen für die Kommunikation über HTTP oder der leichtgewichtigen Kommunikation mit 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 [36].

Dezentrales Datenmanagement

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

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

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

„Design for failure “

Softwarekomponenten müssen den Ausfall von anderen Diensten tolerieren. Event-basierte Kommunikation führt oft zu Fehlverhalten und kann durch Überwachungs-tools präventiv verhindert werden [36].

Kapitel 3

Analyse

Das vorherige Kapitel beschrieb die nötigen Technologien für die Realisierung und Ausführung einer Anwendung im Microservice-Architektur-Stil. Dieses Kapitel beschäftigt sich nun mit der Analyse für die spätere Konzeptphase.

3.1 Modernisierung der Infrastruktur

Dieser Abschnitt behandelt die aktuellen Bestrebungen der Krones AG hinsichtlich dem Aufbau einer modernen Infrastruktur in Produktionsanlagen. Zunächst wird der Proof of Concept, in dem die Umsetzbarkeit eines Kubernetes fähigen Konzept geprüft wurde behandelt. Anschließend werden Anwendungsmöglichkeiten im Bereich der Echtzeitkommunikation oder künstlichen Intelligenz untersucht.

3.1.1 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 Umsetzbarkeit von Kubernetes-Technologien zu evaluieren. Diese beinhaltete die Aufrüstung einer Produktionsanlage mit Virtual-Edge-Devices, die über einen Hypervisor Typ 1 zwei Betriebssysteme gleichzeitig ausführen. Windows 10 Embedded, als Human Interface (HMI) und Linux SUSE Linux Enterprise Micro. Die Integration der Geräte ermöglicht die maschinennahe Nutzung von Echtzeitinformationen während des Betriebs und dienen als Fundament für den Aufbau einer wartungsfreien Infrastruktur.

Kubernetes

Die einzelnen Virtual-Edge-Devices sollen in Zukunft als Knotenpunkte für ein Kubernetes Cluster dienen. Dafür wird die für Edge-Szenarien entworfene Kubernetes-Distribution k3s installiert. Die Kubernetes-Cluster werden mit dem Orchestrierungstool Rancher verwaltet. Diese abstrahiert die Komplexität für den aktiven Betrieb von mehreren Clustern.

3.1.2 Aufgabenstellung

Im Zuge dessen wird die Umsetzbarkeit und Relevanz von Microservices auf der zukünftigen Produktionsanlage untersucht. Diese muss folgende Aufgabengebiete untersuchen.

Bereitstellung

Die Auslieferung und Entwicklung bestimmter Software soll zukünftig über Kundenrepositories erfolgen. Diese ermöglichen einen zentralen Ort für die kundenindividuelle Konfiguration der Anlagensoftware.

Hybrid Cloud

Die Anwendung funktioniert in einem hybriden Cloud-Szenario mit unterschiedlichen Ressourcentypen. 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. Die flexible Wahl der Ressourcen ermöglicht kürzere Berechnungszeiten, ohne on-premise Hardware zu besitzen, die zur Auswertung und Evaluierung von Produktionsanlagen dienen.

Echtzeitkommunikation

Die Anwendung auf den Edge-Devices kommunizieren in Echtzeit miteinander. Und können die Informationen sinnvoll verarbeiten und persistent speichern.

Künstliche Intelligenz

Die Anwendung bringt einen Mehrwert im Bereich der künstlichen Intelligenz und nutzt die Infrastruktur zur Auswertung von Deep Learning Modellen.

3.2 Fachkonzept

Im folgenden Abschnitt werden die Anforderungen und Anwendungsfälle für die spätere Konzeption der Microservice-Architektur erhoben. Dafür wird ein zielorientierter Ansatz gewählt, der Anforderungen aus den Zielen einer Aufgabenstellung herleitet [45, S.47].

3.2.1 Anforderungserhebung

Der vorherige Abschnitt 3.1 beschreibt die Aufgabenstellungen, die mithilfe von Zielen erreicht werden können. Ziele sind erforderlich, um die zukünftigen Anforderungen der Software zu erarbeiten und werden in einer Anforderungstabelle wiedergeben (vgl. Tabelle 3.1).

Nr.	Ziele	Anforderungen
A01	Auf dem Kubernetes-Cluster laufen Anwendungen im Bereich der künstlichen Intelligenz.	Dienste sollen Funktionen der künstlichen Intelligenz ausführen können.
A02	Anwendungen auf einem Kubernetes-Cluster haben die Möglichkeit, Daten persistent zu speichern.	Dienste sollen über einen Speicher verfügen, der Daten persistent speichert.
A03	Die einfache Auslieferung der Anwendung auf dem Kubernetes-Cluster ist möglich.	Die Dienste sollen über einen Auslieferungsmechanismus bereitgestellt werden.
A04	Die Inbetriebnahme der Anwendung auf ein Kubernetes-Cluster ist vorkonfiguriert.	Dienste sollen vor der Inbetriebnahme vorkonfiguriert werden.
A05	Die Anwendung funktioniert ohne die zusätzliche Installation von Abhängigkeiten auf dem Hostsystem.	Dienste sollen bei der Installation bereits über die nötigen Abhängigkeiten verfügen.
A06	Die Anwendung kommuniziert in Echtzeit mit anderen Anwendungen auf dem Hostsystem.	Dienste sollen in Echtzeit miteinander kommunizieren.
A07	Die Anwendung wurde vor der Inbetriebnahme auf dem Kubernetes-Cluster getestet.	Dienste sollen vor der Inbetriebnahme Tests durchlaufen.
A08	Die Anwendung lässt sich für spezifische Hardware vorkonfigurieren.	Dienste sollen bei Vorkonfiguration auf spezifischer Hardware laufen.

Tabelle 3.1: Anforderungstabelle

Die Tabelle gewährt einen groben Überblick der nötigen Systemanforderungen an die Mircoservice-Architektur. Diese werden nun im nächsten Abschnitt für das Konzept der Anwendung verwendet.

3.2.2 Konzept

Die prototypische Anwendung wird containerisiert und über eine Container-Registry verfügbar sein [A03] [A05]. Weiterhin muss die Anwendung unter der Berücksichtigung der Aspekte einer Microservice-Architektur, wie in Abschnitt 2.3 beschrieben, konzipiert und entwickelt werden. Die einzelnen Dienste der Anwendung müssen auf einem Kubernetes-fähigen Cluster ausgeliefert und in Betrieb genommen werden [A04]. Bevor die Anwendung verwendet wird, muss ein fester Testprozess die Funktionalität gewährleisten [A07].

Für die mögliche Auslieferung bei einem Kunden der Krones AG soll die Nutzung bereits vorhandener Hardware mit Grafikkarten möglich sein [A08]. Es ist vorgesehen, dass die Anwendung in einem hybriden Cloud-Szenario die vordefinierte Hardware nutzen kann. Folglich soll die Verwendung der Hardware zu einer verbesserten

Leistungsauswertung von Modellen im Bereich der künstlichen Intelligenz führen. Arbeitslasten, wie dem Auswerten von Deep-Learning-Modellen wie im Falle der Linatronic AI [2], sollen beispielhaft dargestellt werden [A01]. Dafür muss die Kommunikation von Diensten in Echtzeit stattfinden, um Informationen am Zielort schnell zu verarbeiten und eine nahtlose Verarbeitung großer Daten zu ermöglichen [A06].

Anwendungsszenario

Der Schwerpunkt der zu entwickelnden Anwendung soll ein Dashboard mit Authentifizierungsmechanismus sein. Dieser soll Benutzern ermöglichen sich mit ihrem Passwort oder per Gesichtserkennung in ihr Profil einzuloggen. Die Daten sollen persistent gespeichert werden und können bei erneutem Aufruf der Website wieder verwendet werden [A02].

3.2.3 Vorgehen

Darauf aufbauend, wird im nächsten Schritt das fachliche Vorgehen zur Realisierung des Konzepts festgelegt.

Komponententrennung von Diensten

Wie in Abschnitt 2.3 behandelt werden die Kernfunktionalitäten der zu entwickelnden Anwendung in einzelne Dienste aufgeteilt. Im Systementwurf muss definiert werden über welche Schnittstellen die Dienste verfügen und über welche Endpunkte diese Erreichbar sind. Folgend müssen auch die Service-Contracts zwischen den Diensten definiert werden.

Auslieferungsstrategie

Die Dienste müssen unabhängig voneinander auslieferbar sein. Dienste verfügen über einem eigenen Speicherort, die aus dem Kubernetes-Cluster erreichbar ist.

Bereitstellungsstrategie

Vorkonfiguration

3.3 Grobentwurf

Auf der Grundlage von Abschnitt 3.2.2 werden die Grobentwürfe erstellt. Die Konzeptentwürfe gliedern sich in zwei Teile, der Infrastruktur und der Anwendung.

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

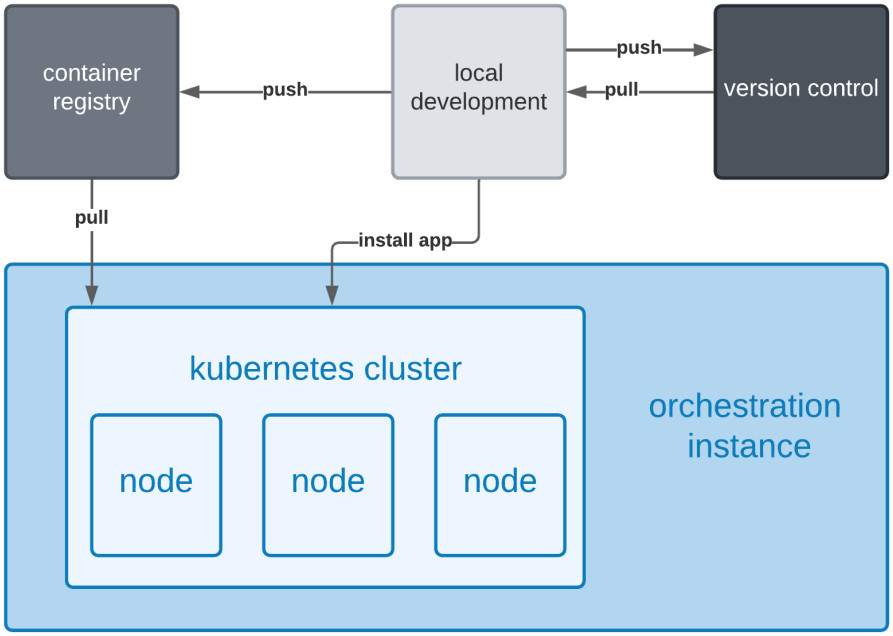

Abbildung 3.1: Grobentwurf der Infrastruktur

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

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

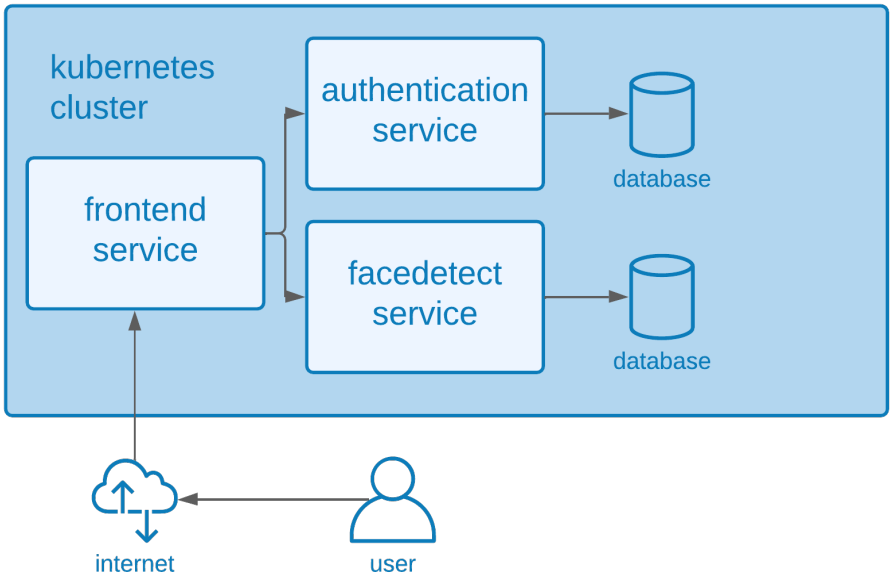

Abbildung 3.2: Grobentwurf der Awendung

Die Abbildung 3.2 stellt das Anwendungszenario aus dem Unterabschnitt 3.2.2 dar. Die Anwendung aus dem Szenario wird in drei Dienste aufgeteilt.

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

Authentication-Service: Die Anmeldung und Registrierung in ein Nutzerkonto erfolgt über den Authentication-Service. Dieser ermöglicht die persistente Speicherung der Nutzerdaten.

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

3.3.1 Entwicklungsprozess

Die Entwicklung der Anwendung wird in vier auf sich aufbauende Schichten eingeteilt (vgl. Abbildung 3.3).

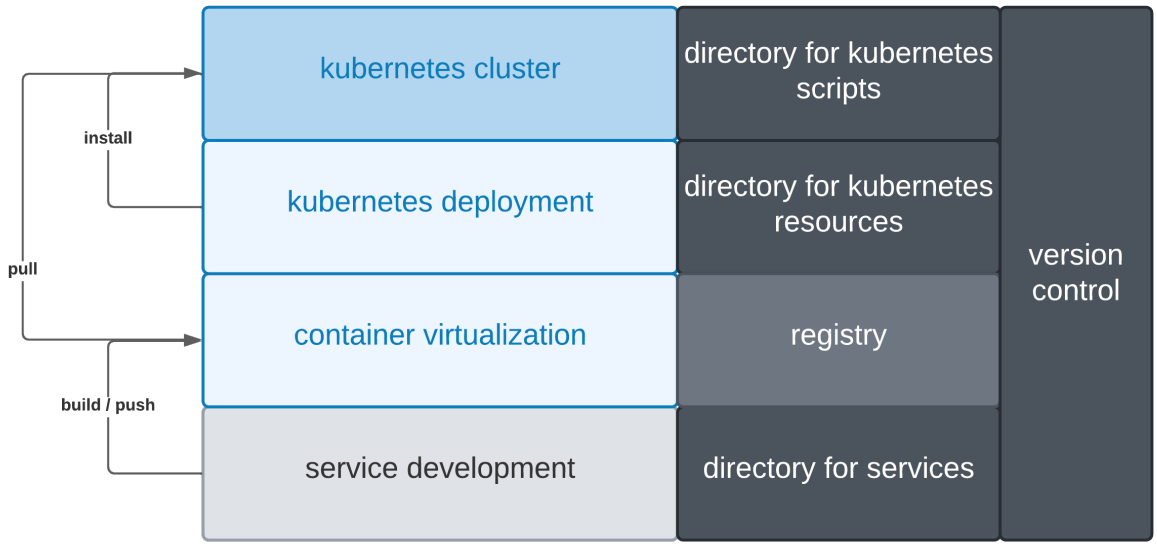

Abbildung 3.3: Vorgehen des Entwicklungsprozesses in Schichten

Anwendungsentwicklung: Ein zentrales Repository beeinhaltet Verzeichnisse für die einzelnen Dienste. Die Dienste 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.

Kubernetes-Deployment: Das zentrale Repository beinhaltet ein weiteres Verzeichnis für die Kubernetes-Ressourcenobjekte in Form von YAML-Dateien. Bei Zugang der Ent-

wicklungsumgebung zu einem Kubernetes-Cluster können diese Dateien ausgeführt werden.

Kubernetes-Cluster: Das Testcluster wird aufgesetzt, installiert und in eine Orchestrierungsinstanz integriert. Skripte für die Vorkonfiguration und Installation des Kubernetes-Clusters, erhalten ein eigenes Verzeichnis im zentralen Repository. Die Auslieferung der Dienste erfolgt über das Image-Registry und werden von dem Kubernetes-Cluster heruntergeladen.

Kapitel 4

Lösungskonzept

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

4.1 Architektur

Dieser Abschnitt befasst sich mit der Architektur der zu entwickelnden Anwendung. Die Ausarbeitung der Software Architektur erfolgt durch das c4model Der folgende Abschnitt besteht aus vier Abstraktionsschichten der Architektur

4.1.1 Authentifizierungs-Service

4.1.2 Backend-Service

4.1.3 Frontend-Service

4.2 Architektur

4.3 Gesichtserkennung

Kapitel 5

Implementierung der Architektur

Das folgende Kapitel beschreibt die Vorgehensweisen der Implementierung. Angefangen mit der Konfiguration und Einrichtung der Knotenpunkte für das Kubernetes-Cluster.

5.1 Konfiguration und Einrichtung

In diesem Abschnitt geht es um die Einrichtung der Kubernetes Infrastruktur. Zuerst die Einrichtung der einzelnen virtuellen privaten Server in Vultr, die als Knotenpunkte in unserem Kubernetes Cluster funktonieren. Danach die Installation der Infrastruktur mit k3s. Zunächst wird eine Domain für den späteren Einsatz der Microservices konfiguriert. Abschließend erfolgt die Bereitstellung von Zertifikaten für die Domain.

5.1.1 Virtueller privater Server

Durch die Einschränkungen, beschrieben in Abschnitt 7.1, werden für die Installation der Kubernetes Plattform virtuelle private Server (VPS) verwendet. Ein VPS ist eine virtuelle Maschine, die von Drittanbietern wie Internet-Hosting-Diensten, als Dienst verkauft wird. Dies ermöglicht das Mieten von Hardware. Die Server dienen als Knotenpunkte für die spätere Kubernetes Installation. Es werden ingesamt drei VPS-Instanzen gemietet auf denen das folgende Betriebssystem installiert wird.

Betriebssystem

Im Rahmen des PoC mit dem Unternehmen SUSE wurde das Betriebssystem SLE-Micro Enterprise 5.1 bereitgestellt und auf den Serverinstanzen installiert. Dieses 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.

Domain

Der Zugang zur Webanwendung wird mithilfe einer öffentlichen Domain ermöglicht. Der DNS-Eintrag einer Domain ist für die Adressierung zuständig. Durch die Veränderung des A-Records leiten alle Anfragen der Domain auf eine IPv4-Adresse um [46]. Die IPv4 Adresse ist in diesem Fall der Cluster Master der späteren k3s-Installation.

5.1.2 Kubernetes Installation

Dieser Abschnitt behandelt die Einrichtung und Installation der Kubernetes-Distribution Lightweight Kubernetes und der Orchestrierungsplattform Rancher.

Lightweight Kubernetes

Für die Installation von k3s auf den Server wurde ein Shell-Skript entwickelt, das mit den nötigen Befehlen aus der Dokumentation geschrieben wurde.

```
1  curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server"
       K3S_CLUSTER_INIT=1 sh -
2  TOKEN=$( cat /var/lib/rancher/k3s/server/node-token )
3
4  USERNAME=root
5  SCRIPT="curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server"
       K3S_TOKEN=$TOKEN K3S_URL=https://$ip4:6443 sh - "
6  for HOSTNAME in ${HOSTS[@]} ; do
7      ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTNAME}
           "hostnamectl set-hostname ${HOSTNAMES[$COUNTER]}; ${SCRIPT}"
8      echo "HOSTNAME CHANGED: ${HOSTNAMES[$COUNTER]}"
9      ((COUNTER++))
10 done
```

Quellcode 5.1: Ausschnitt aus dem installk3s.sh

Das Skript holt mittels *curl* Aufruf das Installationsskript für k3s und führt es mit den vorgegebenen Initialisierungsparametern aus. Der Parameter *INSTALL_K3S_EXEC* bestimmt die Aufgabe der Node. *K3S_CLUSTER_INIT* initialisiert die Node als neuen Cluster-Master. Danach wird ein Token im *../k3s/server* Verzeichnis angelegt. Dieser ist für die Verknüpfung der andern Knoten nötig und wird als Variable gespeichert. In einer Schleife werden die vom Skript vorher abgefragten IP-Adressen und Hostsystemnamen des Benutzer verarbeitet. Über das Netzwerkprotokoll Secure Shell (SSH) verbindet sich das ausführende System mit den restlichen Knotenpunkten und installiert mithilfe der Variablen *TOKEN* und *SCRIPT* k3s.

Rancher

Im Rahmen des PoCs wurde bereits ein Rancher-Server zur Verwaltung von mehreren downstream-Cluster erstellt. Der im vorherigen Abschnitt 5.1.2 eingerichtete k3s-

Cluster wird mit dem Rancher Server verbunden. Über die Rancher-Benutzeroberfläche lässt sich das Kubernetes-Cluster mithilfe des *Add Cluster* Buttons hinzufügen. Die weiteren Schritte ermöglichen das benennen des Cluster und den notwendigen Befehl zum verknüpfen der gewünschten Cluster. Diese werden wegen ihrer Trivialität nicht weiter ausgeführt.

5.1.3 KubeVision

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

5.1.4 Frontend-Service

Das Frontend besteht aus einer HTML5-Benutzeroberfläche und wird für die Interaktion mit dem Benutzer mit JavaScript kombiniert.

5.1.5 Authentifizierungs-Service

5.1.6 Backend-Service

5.2 Gesichtserkennung

5.2.1 Alignment

5.2.2 Training

5.2.3 Model

5.3 Containerisierung

5.3.1 Volumes

5.3.2 Netzwerk

5.3.3 Docker-Compose

5.3.4 DockerHub

5.4 Orchestrierung

5.4.1 SSL-Verschlüsselung

5.4.2 Deployment

5.4.3 Ingress

Nginx-Ingress

5.4.4 Loadbalancer

5.4.5 Taints and Tolerations

5.4.6 Node Affinity

5.4.7 Helm

5.5 Testen der Implementeriung

5.5.1 Service Kommunikation

5.5.2 Loadbalancing

5.5.3 Gesichtserkennung

Kapitel 6

Ergebnisse

6.1 Microservice

6.1.1 Frontend-Serivce

6.1.2 Backend-Serivce

6.1.3 Authentifizierungs-Serivce

6.1.4 Loadbalancer

6.1.5 Kubernetes Cluster

Kapitel 7

Diskussion und Ausblick

7.1 Einschränkungen

7.2 Diskussion

7.3 Ausblick

Abkürzungsverzeichnis

PoC Proof of Concept

VM Virtuelle Maschine

SHA Secure Hash Algorithm

API Application Programming Interface

REST Representational State Transfer

OSI Open Systems Interconnection

YAML Yet Another Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IoT Internet of Things

AWS Amazon Web Services

HMI Human Interface

SSH Secure Shell

Literaturverzeichnis

[1] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[2] "Krones linatronic 735," Feb. 2022. [Online]. Available: https://www.krones.com/de/produkte/maschinen/leerflaschen-inspektionsmaschine-linatronic-735.php

[3] Y. Zhou, Y. Yu, and B. Ding, "Towards mlops: A case study of ml pipeline platform," in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 2020, pp. 494–500.

[4] N. Poulton, *Docker deep dive : zero to Docker in a single book*, 2020th ed. [Germany]: Nigel Poulton, 2020.

[5] "Docker overview," Jan. 2022. [Online]. Available: https://docs.docker.com/get-started/overview/

[6] "About storage drivers," Jan. 2022. [Online]. Available: https://docs.docker.com/storage/storagedriver/

[7] "Best practices for writing dockerfiles," Jan. 2022. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[8] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.

[9] "Are Containers Replacing Virtual Machines?" Aug. 2018. [Online]. Available: https://www.docker.com/blog/containers-replacing-virtual-machines/

[10] "Was ist kubernetes?" section: docs. [Online]. Available: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/

[11] "Kubernetes components," section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[12] "Kubernetes (k8s)," Feb. 2022, original-date: 2014-06-06T22:56:04Z. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.20.md/#urgent-upgrade-notes

[13] "Nodes," section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/

[14] "Don't panic: Kubernetes and docker," Dec. 2020, section: blog. [Online]. Available: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/

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

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

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

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

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

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

[21] "Layer 4 and layer 7 load balancing." [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/k8s-in-rancher/load-balancers-and-ingress/load-balancers/

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

[23] "Raspberry pi documentation - processors." [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html

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

[25] "K3s: Lightweight kubernetes." [Online]. Available: https://k3s.io/

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

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

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

[29] "Overview." [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/

[30] S. Buchanan, J. Rangama, and N. Bellavance, "Deploying and using rancher with azure kubernetes service," in *Introducing Azure Kubernetes Service : A Practical Guide to Container Orchestration*, S. Buchanan, J. Rangama, and

N. Bellavance, Eds. Berkeley, CA: Apress, 2020, pp. 79–99. [Online]. Available: https://doi.org/10.1007/978-1-4842-5519-3_6

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

[32] "Overview." [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/

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

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

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

[36] "Microservices." [Online]. Available: https://martinfowler.com/articles/microservices.html

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

[38] "Softwarecomponent." [Online]. Available: https://martinfowler.com/bliki/SoftwareComponent.html

[39] S. Newman, "Implementing microservice communication." Sebastopol, CA: O'Reilly Media, Sep. 2021.

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

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

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

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

[44] "Boundedcontext." [Online]. Available: https://martinfowler.com/bliki/BoundedContext.html

[45] P. A. Laplante, *Requirements Engineering for Software and Systems*, ser. Applied Software Engineering Series. London, England: Auerbach, Mar. 2009.

[46] J. Belamaric and C. Liu, *Learning coredns*. Farnham, England: O'Reilly UK, Sep. 2019.

Abbildungsverzeichnis

Tabellenverzeichnis