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: 11. 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 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. Ein riesiger Aufwand der Fehleranfälligkeiten wie 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 Linien 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 bestimmt ist. Künftige Produktionsanlagen sollen mittels zusätzlichen 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 beim 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 Architektur zu einer mehr flexibleren microservice Architektur bevorzugt, sind das Hauptmotiv der Auswertung neuer agiler Distributionsmöglichkeiten. Die Containerisierung von Anwendungen ermöglichen erst die Aufteilung großer Projekte in kleine unabängige Services die mittels Orchestrierungsplattformen sinnvoll gebündelt werden können. Namenhafte Unternehmen wie Netflix, Amazon und Uber entwickeln und verwenden bereits robuste und komplexe Microservices die containerisiert auf 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. Verwaltung von Ressourcen wie Speicher, Rechenleistung und Verbindungsgeschwindigkeit für die Datenübertragung, bei der Ausführung einzelner Phasen von der Datenverarbeitung bis hin zur Evaluierung und Entwicklung [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 erneuten Aufruf der Website bestehen zu bleiben. Die Konzeption der Anwendung findet containerisiert auf mehreren Software und Hardware Layern statt. Das ganze System wird auf einem Kubernetes Cluster bereitgestellt und verwaltet. Das bereitstellen von einem Service 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 backend Service.

Kapitel 2

Grundlagen

Dieses Kapitel erläutert die Grundlagen die zum Verständnis dieser Arbeit notwendig sind. Angefangen mit der Erklärung für Technologien Docker und Kubernetes, hinzu Microservices.

2.1 Docker

In diesem Abschnitt wird die Technologie „Docker“ näher erläutert und nicht das Unternehmen „Docker, Inc.“, dass für die maßgebliche Entwicklung dessen verantwortlich ist. Angefangen mit der Terminologie, zum deutlicheren Verständnis der nächsten Abschnitte. Fortgesetzt mit der aufsteigenden Erklärung der Architektur bis 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 [4]. Diese beeinhalten alle nötigen Abhängigkeiten zur Ausführung der vordefinierten Anwendung. Container gewinnen dadurch an Portabilität, die ein bereitstellen auf Infrastrukturen mit der Docker Laufzeit ermöglichen. 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 Container, ziehen von Images, erstellen von Netzwerken und Verwaltung von runc. Die Allgemeine Aufgabe des Docker Daemons ist es eine vereinfachte Schnittstelle für die Abstraktion der unterliegenden Schicht zu gewährleisten, wie zum Beispiel dem verwalten von Images, Volumes und Netzwerken [5]. Auf die Orchestrierung mit Swarm wird nicht weiter eingeganen, da sie zum Verständnis nicht nötig ist.

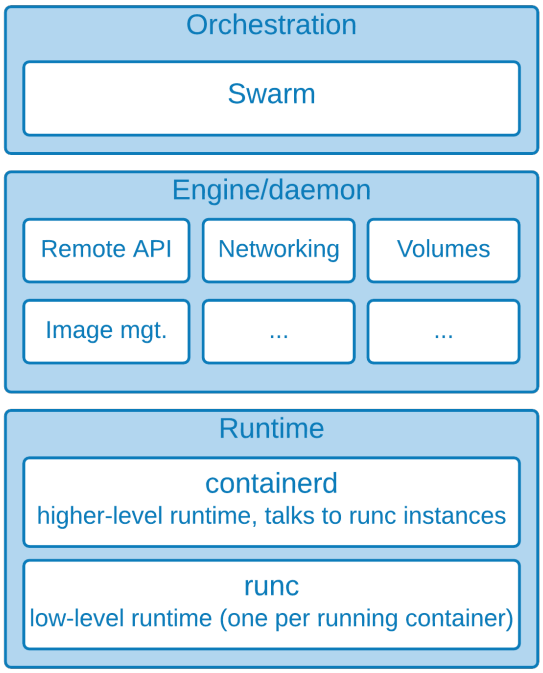

Abbildung 2.1: Docker Architektur [5]

2.1.2 Images und Container

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

Registries

Das beziehen von Images erfolgt über sogenannte „Image Registries“. Bei Docker ist dies standardmäßig `https://hub.docker.com` und das eigene Lokale Registry. Es ist auch möglich eigene zu hosten oder die von Drittanbieter 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“ oder auch „Container layer“ genannt hinzu, dieser gewährt Schreib- und Leserechte bei Laufzeit des Prozesses. Jeder dieser Container hat somit einen

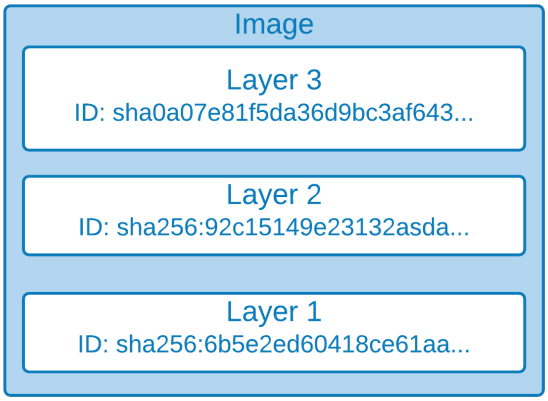

Abbildung 2.2: Image Layers

individuellen Zustand, der unähnlich vom abstammendem Image ist. Bei Löschung des Containers verschwindet auch die dazu gewonne 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 von einem base-image.
- **COPY** hinzufügen von Dateien aus dem derzeitigen Verzeichnis.
- **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 virtuellen Maschine (VM). Denn Images ermöglichen ähnlich wie VM templates, die Erstellung von mehreren Instanzen durch eine Vorkonfiguration. Mit dem großen Unterschied, dass die Einrichtung von VMs müheseliger ist und weitaus mehr Ressourcen beansprucht, da sie ein ganzes Betriebssystem ausführt [8]. Containertechnologien

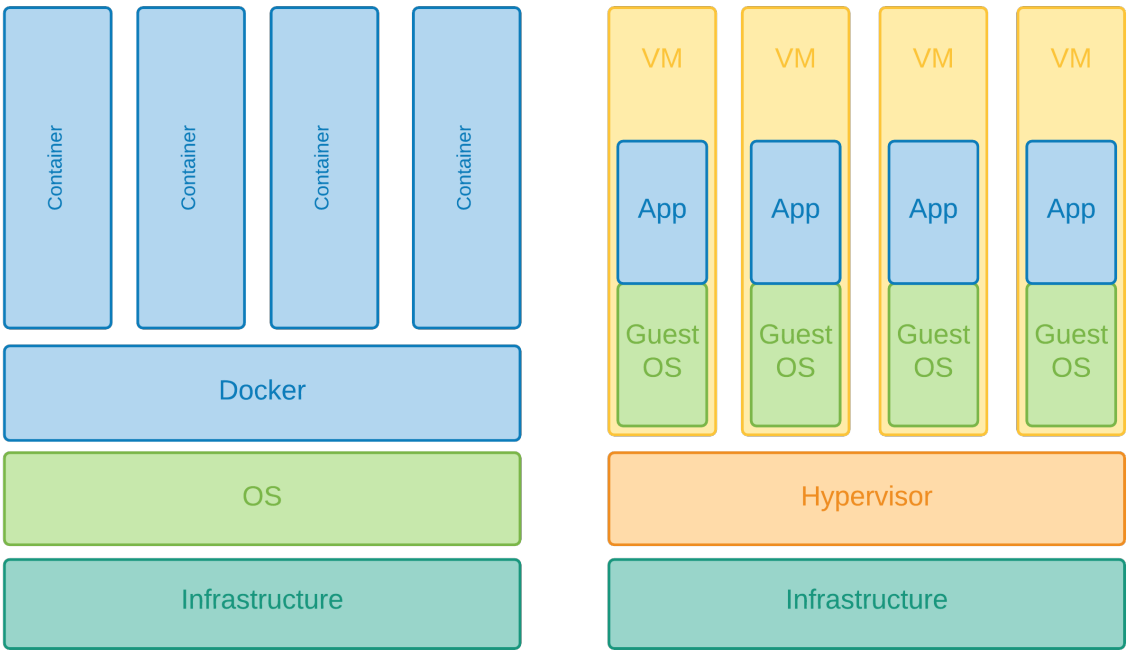

Abbildung 2.3: Virtualisierungsmöglichkeiten

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

Durch die Vorteile eines geteilten Kernels und dessen Betriebssystem abhängigkeiten, erzielen Virtualisierungen basierend auf Container eine höhere Anzahl an virtuellen Instanzen. Images sind auch um einiges kleiner als hypervisor-basierende Ansätze [8].

Die Einsparung von Ressourcen und dem einfachen bereitstellen auf Hostsysteme, prädestinieren containerisierte Anwendungen für die Verwendung von Microservices auf Container Plattformen wie Kubernetes.

2.2 Kubernetes

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 Services. Seit 2014 hat Google das Open-Source-Projekt zur Verfügung gestellt und baut auf 15 Jahre Erfahrungen mit Produktions-Workloads [9].

Namensgebung

„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“ [9].

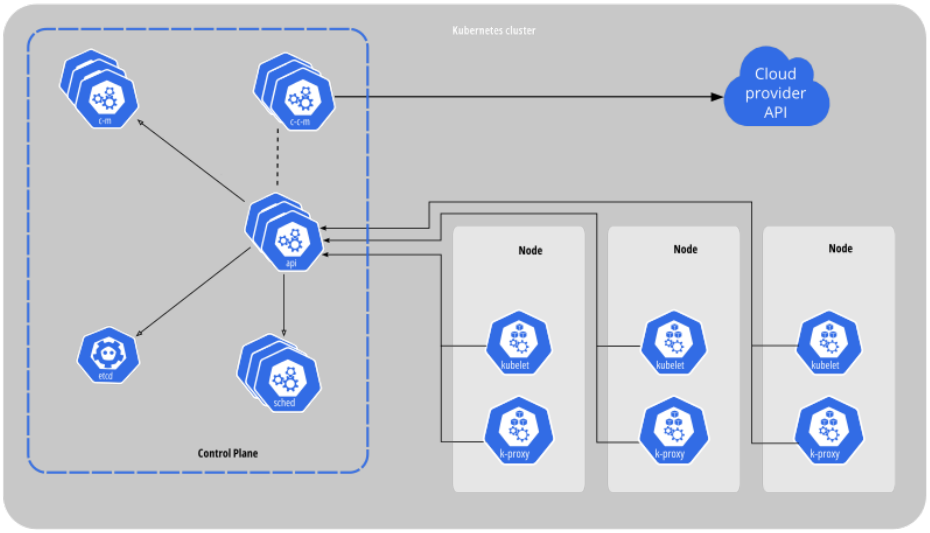

Abbildung 2.4: Komponenten eines Kubernetes Cluster [12]

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 [11]. Die nächsten Unterabschnitte beschreiben diese näher:

API Server

Der API Server operiert über REST und bietet eine Schnittstelle zu Diensten inner- und außerhalb der Master-Komponenten [11].

etcd

etcd ist der primäre Datenspeicher von Kubernetes und sichert alle Zustände eines Cluster [11].

Scheduler

Der Scheduler ist zuständig für die Verteilung und Ausführung von Pods auf Nodes [11].

[1]Seit Kubernetes v1.20, ist die korrekte Bezeichnung für die Master Node Control Plane [23]

Controller Manager

Der Controller Manager reagiert auf Ausfälle von Nodes, erhält die korrekte Anzahl von Replikationen eines Pods und verbindet Services miteinander [11].

Node

Eine Node[2] ist eine Hardware Einheit, die je nach Kubernetes Einrichtung eineVM, physische Maschine oder eine Instanz in einer privaten oder öffentlichen Cloud darstellen kann [13]. Dieser umfasst folgende Komponenten:

Container Laufzeit

Der Abschnitt 2.1 beschreibt bereits alles zu diesem Thema. Ergänzend dazu die Information, dass seit 2020 containerd als Auslaufmodell für die unterliegende Container Laufzeit, für die Kubernetes Versionen nach v1.20 auslaufen. Dies beeinträchtigt die spätere Implementierung dieser Arbeit aber nicht, da k3s containerd als standard Laufzeit verwendet wird.

Kubelet

Kubelet fungiert als „node agent“ und registriert die Node mit dem API-Server eines Clusters, dabei stellt es sicher das Container innerhalb eines Pods funktionieren [13].

Kube-Proxy

Ein Kube-Proxy ist ein Netzwerk Proxy und verwaltet die Netzwerkrechte auf Nodes. Diese erlauben die Kommunikation zwischen Pods inner- und außerhalb des Clusters [13].

2.2.2 Pods

Ein Pod stellt die kleinste Einheit eines Kubernetes Clusters dar und ist eine Gruppe von mindestens einem Container. Dieser erlaubt die gemeinsame Nutzung von Speicher- und Netzwerkressourcen mit Anweisungen zur Ausführung der Container [14].

2.2.3 Deployment

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

- **apiVersion**: definiert die einzelnen workload API Untergruppen und die Version.

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

- **kind**: bestimmt das zu erstellende Kubernetes Objekt.
- **metadata**: deklariert einzigartige Bestimmungsmerkmale.
- **spec**: gewünschte Ausgangszustand des Objekts.

```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 wertvollen Metadaten für die Verwaltung von Skalierbarkeit, Rollouts, Rollbacks und Selbstheilungsprozesse [17]. Der höhere Grad an Abstraktion dient auch zur 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 einer logischen Gruppe 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. Beispiel 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: my-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Quellcode 2.2: zum deutlicheren Verständnis mit dem deployment.yaml, leicht abgewandelte service.yaml [18]

Bei der Erstellung eines Services wird ein REST Objekt erstellt, dass mithilfe eines Controller kontinuierlich nach Pods mit dem passenden Selector sucht, welcher jegliche Updates als POST-Anfragen schickt.

2.2.5 Ingress

Ein Ingress ist ein Kubernetes Ressourcenobjekt, dass die Verfügbarkeit 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 in alle anderen Kubernetes Objekten auch werden vordefinierte Aufgaben des Ingress Controller durch YAML-Dateien abgebildet (vgl. Beispiel 2.3). Im folgenden wichtige Optionen die etwas genauer erklärt werden:

- **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 Service 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: test
            port:
              number: 80
```

Quellcode 2.3: ingress.yaml [19]

2.2.6 Lightweight Kubernetes

Ligthweight Kubernetes auch K3s genannt ist eine Kubernetes Distribution von dem Unternehmen Rancher. Der größte Unterschied der Distribution ist die einnehmende größe auf Hostsystemen mit einer einzelnen Binärdatei von nur 40MB ist auch Platz auf kleineren Geräten. Der hauptsächliche Verwendungszweck von k3s liegt in IoT und Edge-Devices, da unwichtige Kubernetes Inhalte entfernt wurden. [24]. Trotz dieser Reduzierung, bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden so weit wie 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 dessen ist, das Server parallel einen Agent Prozess auf dem selben Knoten starten und somit Arbeitslasten mithilfe von Kubelet ausführen. Weiterhin wird im Gegensatz zu k8s

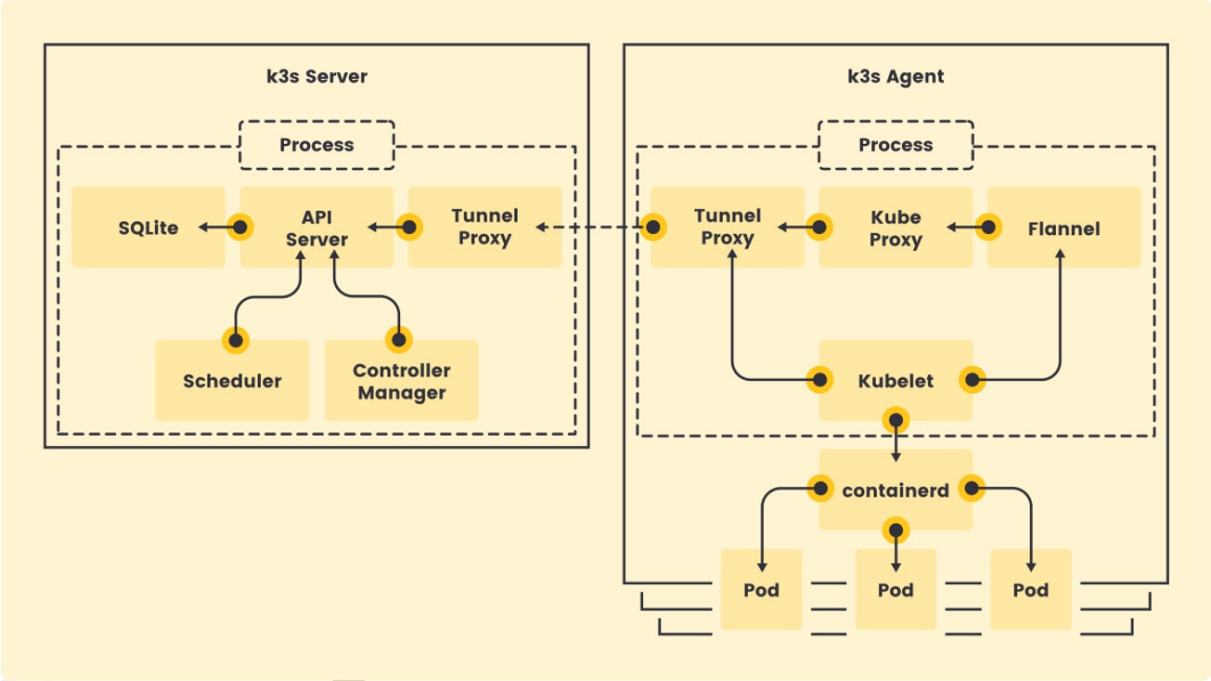

Abbildung 2.5: k3s Architektur [25]

containerd weiterhin unterstüzt und kommt vorinstalliert mit Kubelet. Zwei weitere Besonderheiten werden näher erläutert:

Kine das Akronym steht für 'Kine is not etcd' und ist eine Abstraktionsschicht von etcd, welche sqlite, Postgress, Mysql und dqlite übersetzt [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. Eine Binärdatei startet Agents auf Hostssystemen und 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 Abschnitt wird die Open-Source Lösung Rancher von dem gleichnamigen Unternehmen zur Orchestierungs von Kubernetes Clustern näher behandelt. Diese 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 der selben 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

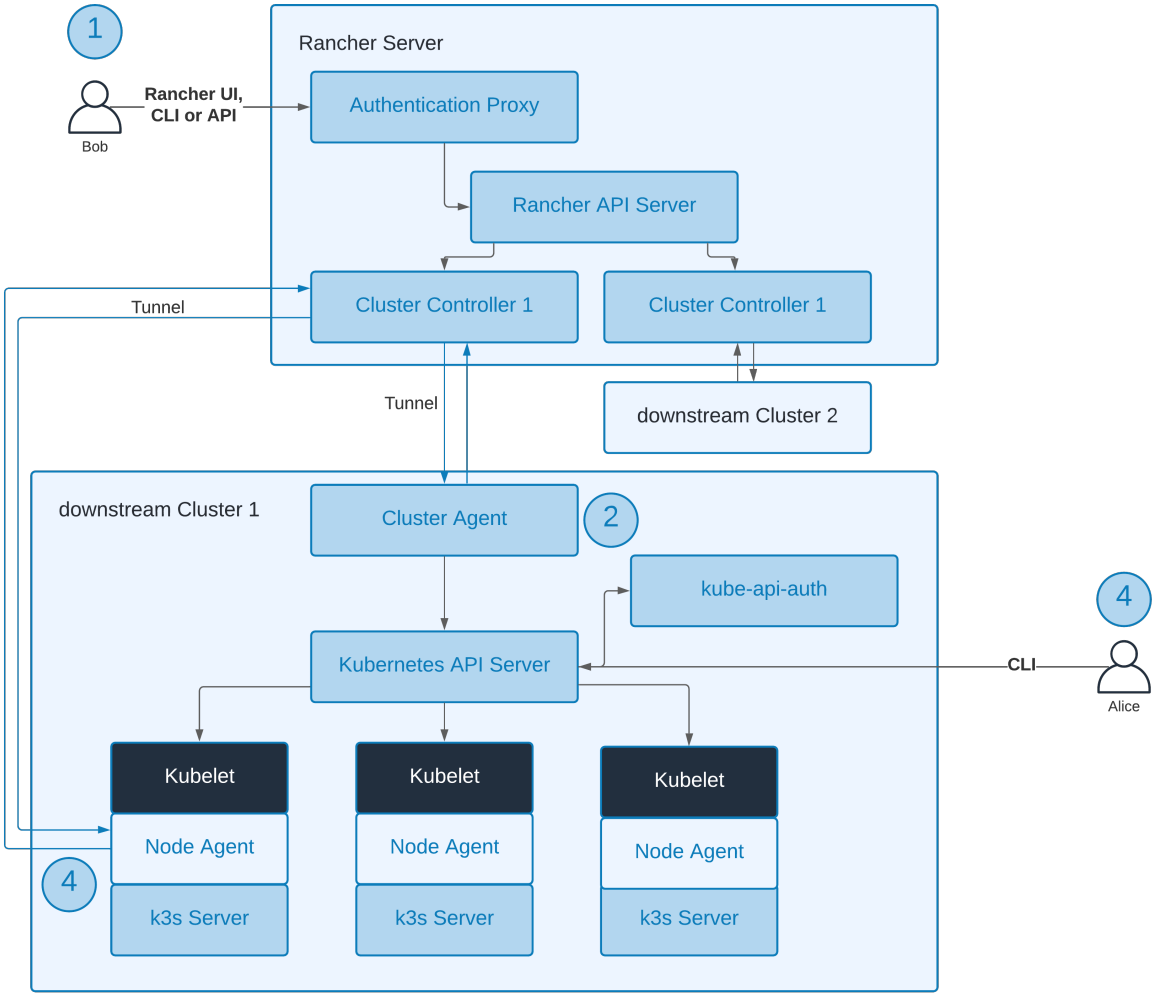

Abbildung 2.6: Rancher Server Kommunikation mit einem downstream k3s Cluster, überarbeitete Abbildung von [31]. (Für die spätere Implementierung nachgebildet)

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. Für komplexere Konfigurationen, kann über die Oberfläche ein Terminal mit Kubectl aufgerufen werden [28, 29].

Die Abbildung 2.6 zeigt den Vorgang von Zwei Benutzern, die auf ein von Rancher verwalteten downstream Cluster[3] zugreifen. Die nachfolgende Beschreibung aus 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 für seine Rancher Instanz. Dieser Proxy leitet den Aufruf mit der ausgewählten downstream-Cluster Instanz weiter und führt diese aus. Dafür wird

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

vor dem weiterleiten des Aufrufs, der angemessene Kubernetes Impersonation Header gesetzt, welcher sich als Service Account der Rancher Instanz ausgibt und je nach Benutzerrecht reagiert.

2. Die Übertragung des Aufrufs erfolgt über einen Cluster-Controller auf dem Rancher Server und dem parallel laufendem Cluster-Agent auf dem downstream-Cluster. Der Controller ist für die Überwachung, Veränderung und Konfiguration von Zuständen auf dem laufendem Cluster zuständig.
3. Wenn der Cluster-Agent nicht erreichbar ist, werden die Aufrufe an den Node-Agent[4] 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-Haken realisiert, der die Verbindung zwischen Alice Rechner ermöglicht. Dies ermöglicht die Verwendung von Befehlszeilentools, wie Kubectl und Helm.

2.2.8 Hybrid Cloud

Eine hybride Cloud erlaubt die Verwendung von Hardware in der Cloud als auch vor Ort in der selben Infrastruktur zu betreiben.

2.3 Microservice

2.3.1 Aufbau

2.3.2 Entwicklung

2.3.3 Dezentrale Datenmanagement

[4] Ein Rancher DaemonSet zur Interaktion mit Nodes, nicht zu verwechseln mit dem k3s-Agent [33].

Kapitel 3

Microservice Architektur

3.1 Anforderungen eigenes kapitel?

3.2 Konzept

3.3 Herausforderungen

3.4 KI-Anwendungsfall

3.4.1 Gesichtserkennung

Kapitel 4

Implementierung der Architektur

4.1 Aufbau der Implementierung

4.1.1 Hardware-Layer

4.1.2 Software-Layer

4.1.3 Betriebssystem

4.2 Konfiguration und Einrichtung

4.2.1 Virtueller Privater Server

4.2.2 Domain

4.2.3 SSL-Verschlüsselung

4.3 Frameworks und Bibliotheken für Microservices

4.3.1 Flask

4.3.2 Gunicorn

4.3.3 SocketIO

4.3.4 OpenCV

4.3.5 MongoDB

4.4 Gesichtserkennung

4.4.1 Alignment

4.4.2 Training

4.4.3 Model

4.5 Containerisierung

Kapitel 5

Ergebnisse

5.1 Microservice

5.1.1 Frontend-Serivce

5.1.2 Backend-Serivce

5.1.3 Loadbalancer

5.1.4 Kubernetes Cluster

Kapitel 6

Diskussion und Ausblick

6.1 Einschränkungen

6.2 Diskussion

6.3 Ausblick

Literaturverzeichnis

[1] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casal- ´ las, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in 2015 10th

[2] Krones Linatronic 735,” Krones.com. [Online]. Available: https://www.krones.com/de/produkte/maschinen/leerflaschen-inspektionsmaschine-linatronic-735.php. [Accessed: 16-Jan-2022].

[3] Y. Zhou, Y. Yu and B. Ding, "Towards MLOps: A Case Study of ML Pipeline Platform,"2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE), 2020, pp. 494-500, doi: 10.1109/ICAICE51518.2020.00102.

[4] Docker Overview,” Docs.docker.com. [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed: 20-Jan-2022].

[5] N. Poulton, “Docker,” in Docker Deep Dive: Zero to Docker in a single book: Nigel Poulton, 2020, pp. 12–13.

[6] About storage drivers,” Docs.docker.com. [Online]. Available: https://docs.docker.com/storage/storagedriver/. [Accessed: 20-Jan-2022].

[7] Best practices for writing Dockerfiles,” Docs.docker.com. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.[Accessed: 20-Jan-2022].

[8] R. Morabito, J. Kjällman and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison,"2015 IEEE International Conference on Cloud Engineering, 2015, pp. 386-393, doi: 10.1109/IC2E.2015.74.

[9] Was ist Kubernetes?,” https://kubernetes.io/. [Online]. Available: https://kubernetes.io/.[Accessed: 20-Jan-2022].

[10] Don’t Panic: Kubernetes and Docker,” https://kubernetes.io/. [Online]. Available: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.[Accessed: 20-Jan-2022].

[11] Kubernetes Komponenten,” https://kubernetes.io/. [Online]. Available: https://kubernetes.io/de/docs/concepts/overview/components/.[Accessed: 21-Jan-2022].

[12] Kubernetes Components," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.[Accessed: 21-Jan-2022].

[13] Nodes," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/de/docs/concepts/architecture/nodes/.[Accessed: 21-Jan-2022].

[14] Pods," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/de/docs/concepts/workloads/pods/.[Accessed: 21-Jan-2022].

[15] Understanding Kubernetes Objects," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/.[Accessed: 21-Jan-2022].

[16] Creating a Deployment," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.[Accessed: 21-Jan-2022].

[17] N. Poulton, "Kubernetes Deployments," in The Kubernetes Book, 2021 edition, Nigel Poulton, 2021, pp. 76–78.

[18] Service," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.[Accessed: 21-Jan-2022].

[19] "Ingress," Kubernetes. https://kubernetes.io/docs/concepts/services-networking/ingress/ (accessed Feb. 08, 2022).

[20] "Ingress Controllers," Kubernetes. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (accessed Feb. 08, 2022).

[21] "Layer 4 and Layer 7 Load Balancing," Rancher Labs. https://rancher.com/docs/rancher/v2.5/en/k8s-in-rancher/load-balancers-and-ingress/load-balancers/ (accessed Feb. 08, 2022).

[22] "What is Kubernetes Ingress?," Apr. 21, 2021. https://www.ibm.com/cloud/blog/kubernetes-ingress (accessed Feb. 08, 2022).

[23] Kubernetes (K8s). Kubernetes, 2022. Accessed: Feb. 10, 2022. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.20.md#urgent-upgrade-notes

[24] K3s - Lightweight Kubernetes. k3s-io.[Online]. Available: https://k3s.io. [Accessed: 07-Feb-2022].

[25] K3s - Lightweight Kubernetes. k3s-io.[Online]. Available: https://k3s.io. [Accessed: 20-Jun-2021].

[26] K3s - Lightweight Kubernetes. k3s-io.[Online]. Available: https://github.com/k3s-io/k3s. [Accessed: 07-Feb-2022].

[27] Cluster Networking," https://kubernetes.io/. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/.[Accessed: 30-Jan-2022].

[28] "Overview," Rancher Labs. https://rancher.com/docs/rancher/v2.5/en/overview/ (accessed Feb. 10, 2022).

[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, Eds. Berkeley, CA: Apress, 2020, pp. 79-99.

[30] "Authenticating," Kubernetes. https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (accessed Feb. 10, 2022).

[31] "Architecture," Rancher Labs. https://rancher.com/docs/rancher/v2.5/en/overview/architectu (accessed Feb. 10, 2022).

[32] "Architecture Recommendations," Rancher Labs. https://rancher.com/docs/rancher/v2.5/en/overview/architecture-recommendations/ (accessed Feb. 10, 2022).

[33] "Rancher Agents," Rancher Labs. https://rancher.com/docs/rancher/v2.5/en/cluster-provisioning/rke-clusters/rancher-agents/ (accessed Feb. 10, 2022).

Abbildungsverzeichnis

Tabellenverzeichnis