

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

Studiengang Medieninformatik

Bachelorarbeit

von

Albert Hahn

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

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

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

Studiengang Medieninformatik

Bachelorarbeit

von

Albert Hahn

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

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

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

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

2. Prüfer: Prof. Dr. Dieter Meiller

Bestätigung gemäß § 12 APO

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

Studiengang: **Medieninformatik**

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel:

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

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

Datum: 8. Februar 2022

Unterschrift:

Bachelorarbeit Zusammenfassung

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

Titel:

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

Zusammenfassung:

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

Schlüsselwörter:

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Einleitung | 2 |
| 1.1 | Motivation | 3 |
| 1.2 | Zielsetzung | 3 |
| 2 | Theoretische Grundlagen | 4 |
| 2.1 | Docker | 4 |
| 2.1.1 | Architektur | 4 |
| 2.1.2 | Images und Container | 5 |
| 2.1.3 | Containervirtualisierung | 6 |
| 2.2 | Kubernetes | 7 |
| 2.2.1 | Cluster | 8 |
| 2.2.2 | Pods | 9 |
| 2.2.3 | Deployment | 9 |
| 2.2.4 | Service | 10 |
| 2.2.5 | Ingress | 11 |
| 2.2.6 | Lightweight Kubernetes | 12 |
| 2.2.7 | Rancher | 13 |
| 2.2.8 | Hybrid Cloud | 13 |
| 2.3 | Microservice | 13 |
| 2.3.1 | Aufbau | 13 |
| 2.3.2 | Entwicklung | 13 |
| 2.3.3 | Dezentrale Datenmanagement | 13 |
| 3 | Microservice Architektur | 14 |
| 3.1 | Anforderungen | 14 |
| 3.2 | Konzept | 14 |
| 3.3 | Herausforderungen | 14 |
| 3.4 | KI-Anwendungsfall | 14 |
| 3.4.1 | Gesichtserkennung | 14 |
| 4 | Implementierung der Architektur | 15 |
| 4.1 | Konfiguration und Einrichtung | 16 |
| 4.1.1 | Virtueller Privater Server | 16 |
| 4.1.2 | Domain | 16 |
| 4.1.3 | SSL-Verschlüsselung | 16 |

| | | |
|----------|---|-----------|
| 4.2 | Frameworks und Bibliotheken für Microservices | 16 |
| 4.2.1 | Flask | 16 |
| 4.2.2 | Gunicorn | 16 |
| 4.2.3 | SocketIO | 16 |
| 4.2.4 | OpenCV | 16 |
| 4.2.5 | MongoDB | 16 |
| 4.3 | Gesichtserkennung | 16 |
| 4.3.1 | Alignment | 16 |
| 4.3.2 | Training | 16 |
| 4.3.3 | Model | 16 |
| 4.4 | Containerisierung | 16 |
| 4.4.1 | Volumes | 16 |
| 4.4.2 | Netzwerk | 16 |
| 4.4.3 | Docker-Compose | 16 |
| 4.4.4 | DockerHub | 16 |
| 4.5 | Orchestrierung | 16 |
| 4.5.1 | Deployment | 16 |
| 4.5.2 | Ingress | 16 |
| 4.5.3 | Loadbalancer | 16 |
| 4.5.4 | Taints and Tolerations | 16 |
| 4.5.5 | Node Affinity | 16 |
| 4.5.6 | Helm | 16 |
| 4.6 | Aufbau der Implementierung | 16 |
| 4.6.1 | Hardware-Layer | 16 |
| 4.6.2 | Software-Layer | 16 |
| 4.6.3 | Betriebssystem | 16 |
| 4.7 | Testen der Implementierung | 16 |
| 4.7.1 | Service Kommunikation | 16 |
| 4.7.2 | Loadbalancing | 16 |
| 4.7.3 | Gesichtserkennung | 16 |
| 5 | Ergebnisse | 17 |
| 5.1 | Microservice | 17 |
| 5.1.1 | Frontend-Service | 17 |
| 5.1.2 | Backend-Service | 17 |
| 5.1.3 | Loadbalancer | 17 |
| 5.1.4 | Kubernetes Cluster | 17 |
| 6 | Diskussion und Ausblick | 18 |
| 6.1 | Diskussion | 18 |
| 6.2 | Ausblick | 18 |
| | Literaturverzeichnis | 19 |
| | Abbildungsverzeichnis | 21 |

Tabellenverzeichnis

22

Kapitel 1

Einleitung

Die Krones AG bietet Anlagen für die Getränkeindustrie als auch Nahrungsmittelhersteller, von der Prozesstechnik bis hin zur IT-Lösung. Die Komplettlinie beinhaltet auch das bereitstellen von Software auf den einzelnen Produktionsanlagen. Hierfür werden eine Vielzahl von Produktionslinienanwendungen auf den Anlagen installiert, gewartet und verwaltet. 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 Softwareunternehmen SUSE GmbH, welches das wartungsfreie Betriebssystem SUSE Linux Enterprise Micro und die multi-cluster Orchestrierungsplattform Rancher anbietet.

Als Grundlage hierfür dient das Open-Source-System Kubernetes, welches zur Automatisierung, Skalierung und Verwaltung von containerisierten Anwendungen 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 unabhängige 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 Flexibilität einer solchen Infrastruktur ist es möglich Anwendungsfälle im Bereich der künstlichen Intelligenz für die Industrie zu testen. Die Anlage Linatronic AI der Krones AG nutzt bereits Deep-Learning-Technologie, um in der Linie mittels Vollinspektion Schäden, Dichtflächen oder Seitenwanddicken zu erkennen und Prozesse zu optimieren [2]. Allgemein sind Anwendungen mit künstlicher Intelligenz durch ihre Komplexität und Vielzahl an Abhängigkeiten schwierig zu entwickeln und bereitzustellen. Eine passende Plattform für Anwendungsfälle mit Bezug zur künstlichen Intelligenz muss eine Vielzahl an Services anbieten. 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 Authentifizierungsverfahren 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 Ein Ingress Controller dient dabei als Loadbalancer und verteilt die Last beim Aufrufen der Website und der Kommunikation zwischen backend Service.

Kapitel 2

Theoretische 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 beinhalten 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 eingegangen, 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 beinhaltet.

Registries

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

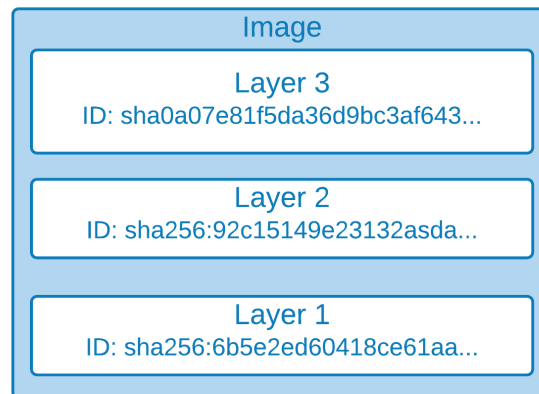


Abbildung 2.2: Image Layers

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 beinhaltet 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].

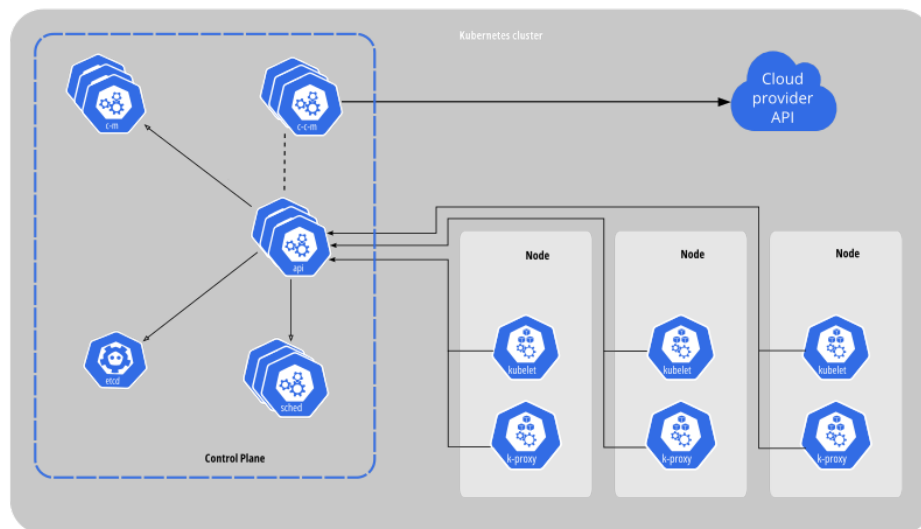


Abbildung 2.4: Komponenten eines Kubernetes Cluster [12]

2.2.1 Cluster

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

Master Node

Master-Nodes 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].

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

Container Laufzeit

Da der Abschnitt zum Thema Docker alles umfassende zur Container Laufzeit bereits beschrieben hat, hier ein kleiner Eintrag zum Thema Docker und Kubernetes. Seit 2020 läuft Docker als Auslaufmodell für die unterliegende Container Laufzeit, nach Version 1.20 von Kubernetes aus. Diese Änderung betrifft, aber nur bestimmte Arbeitsabläufe, die diese Arbeit nicht beeinträchtigen und sollte nur Vollständigkeitshalber erwähnt sein [10].

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


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

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.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-service
5  spec:
6    selector:
7      app: nginx
8    ports:
9      - protocol: TCP
10       port: 80
11       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 beinhaltet die Lastenverteilung durch weiterleiten des Datenverkehrs an Services. Diese Kommunikation findet, wie auch bei dem NGINX Ingress Controller [21] in der Anwendungsschicht des OSI-Schichtenmodells statt und ermöglicht, dadurch die Lastenverteilung von öffentlichen Endpunkten zu internen Pods in einem Cluster [22]. Wie 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 Zusammensetzung 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.

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

Quellcode 2.3: ingress.yaml [19]

2.2.6 Lightweight Kubernetes

Lightweight Kubernetes auch K3s genannt ist eine Kubernetes Distribution von 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 viele Treiber ersetzt wurden [23]. Trotz dieser Reduzierung, bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden so weit wie möglich parallel auf dem neusten Stand gehalten [24].

Unterschiede von k3s

Die Kommunikation zwischen Master Node und Worker Node findet in k3s, über einen Tunnel Proxy auf der Worker Node statt, welche weiter an den Supervisor der Master Node weitergeleitet wird. Desweiteren verfügt auch die Master Node über Kubelet und Kubeproxy (vgl. Beispiel 2.5). Einige Besonderheiten von k3s zu k8s:

Kine

ist eine Abstraktionsschicht von etcd, welche sqlite, Postgress, Mysql und dqlite übersetzt [24].

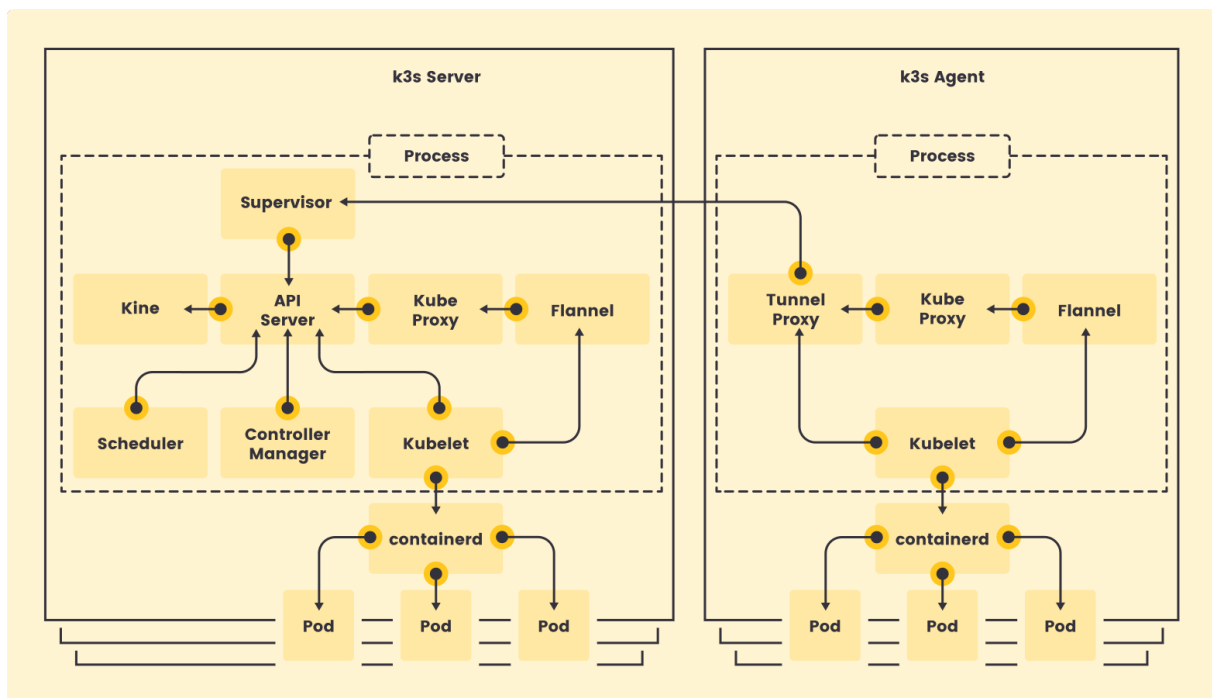


Abbildung 2.5: k3s Architektur [23]

Flannel

verwendet Flannel als überlagerndes Netzwerkmodell [25].

2.2.7 Rancher

Die zuständige Firma hinter k3s ist das Softwareunternehmen Rancher.

2.2.8 Hybrid Cloud

2.3 Microservice

2.3.1 Aufbau

2.3.2 Entwicklung

2.3.3 Dezentrale Datenmanagement

Kapitel 3

Microservice Architektur

3.1 Anforderungen

3.2 Konzept

3.3 Herausforderungen

3.4 KI-Anwendungsfall

3.4.1 Gesichtserkennung

Kapitel 4

Implementierung der Architektur

4.1 Konfiguration und Einrichtung

4.1.1 Virtueller Privater Server

4.1.2 Domain

4.1.3 SSL-Verschlüsselung

4.2 Frameworks und Bibliotheken für Microservices

4.2.1 Flask

4.2.2 Gunicorn

4.2.3 SocketIO

4.2.4 OpenCV

4.2.5 MongoDB

4.3 Gesichtserkennung

4.3.1 Alignment

4.3.2 Training

4.3.3 Model

4.4 Containerisierung

4.4.1 Volumes

4.4.2 Netzwerk

4.4.3 Docker Compose

4.4.4 DockerHub

Kapitel 5

Ergebnisse

5.1 Microservice

5.1.1 Frontend-Service

5.1.2 Backend-Service

5.1.3 Loadbalancer

5.1.4 Kubernetes Cluster

Kapitel 6

Diskussion und Ausblick

6.1 Dikussion

6.2 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] K3s - Lightweight Kubernetes. k3s-io.[Online]. Available: <https://k3s.io>. [Accessed: 07-Feb-2022].
- [24] K3s - Lightweight Kubernetes. k3s-io.[Online]. Available: <https://github.com/k3s-io/k3s>. [Accessed: 07-Feb-2022].
- [25] Cluster Networking," <https://kubernetes.io/>. [Online]. Available: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>. [Accessed: 30-Jan-2022].

Abbildungsverzeichnis

| | | |
|-----|---|----|
| 2.1 | Docker Architektur [5] | 5 |
| 2.2 | Image Layers | 6 |
| 2.3 | Virtualisierungsmöglichkeiten | 7 |
| 2.4 | Komponenten eines Kubernetes Cluster [12] | 8 |
| 2.5 | k3s Architektur [23] | 13 |

Tabellenverzeichnis