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: 13. 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 grundlegenden Begriffe und Konzepte die zum Verständnis dieser Bachelorarbeit notwendig sind. Dabei wird der Technologie-Stack aufsteigend beschrieben. Als Fundament die Container Technologie Docker. Forgesetzt mit der Containerplattform Kubernetes. Abschließend ein Abschnitt zu 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 [4]. Angefangen 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 [5]. Diese beeinhalten alle nötigen Abhängigkeiten zur Ausführung der vordefinierten Anwendungen. 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 [4]. Auf die Orchestrierung mit Swarm wird nicht weiter eingeganen, da sie zum Verständnis nicht nötig ist.

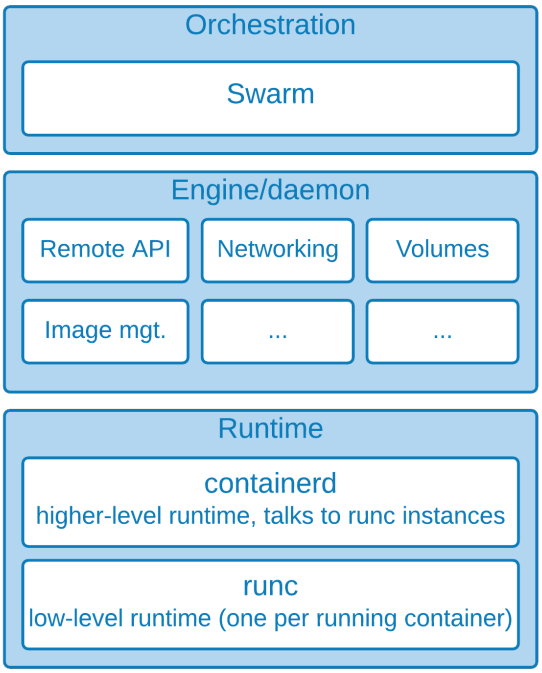

Abbildung 2.1: Docker Architektur in Anlehnung an [4]

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

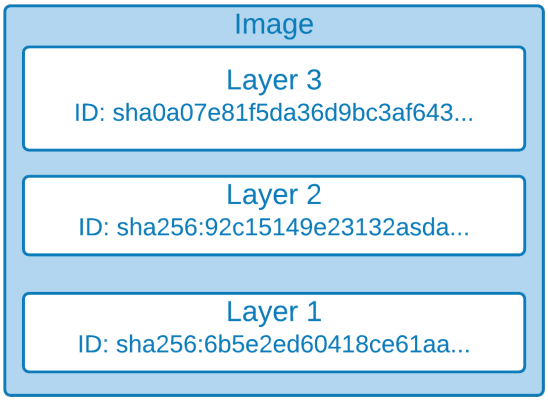

Abbildung 2.2: Image Layers

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

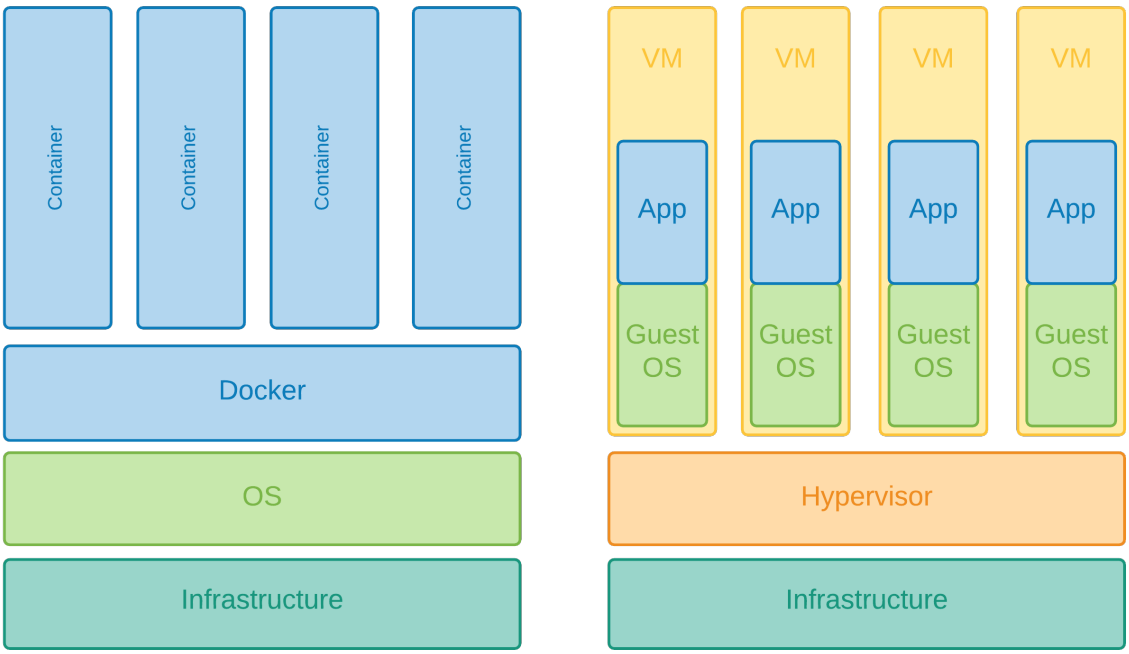

Abbildung 2.3: Virtualisierungsmöglichkeiten

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 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“ [9].

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 [9].

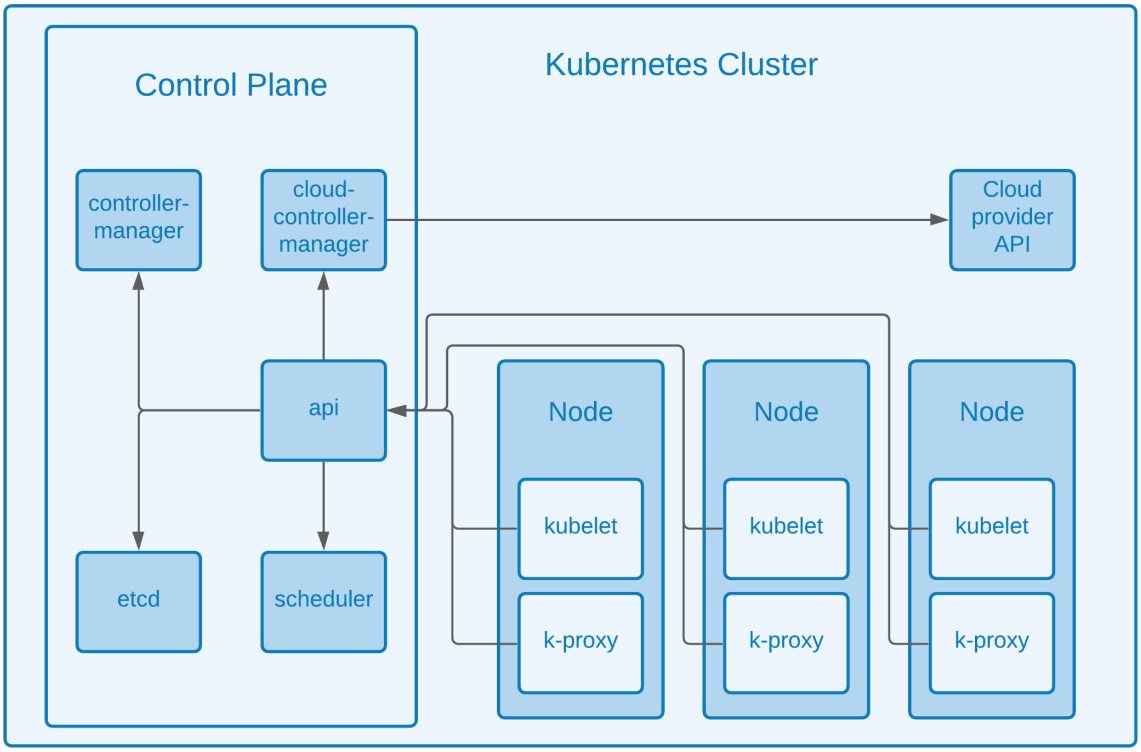

Abbildung 2.4: Komponenten eines Kubernetes Cluster in Anlehnung an [10].

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 beschreiben diese Komponenten wie folgt [10]:

- **API Server**: Der API Server operiert über REST und bietet eine Schnittstelle zu Diensten inner- und außerhalb der Master-Komponenten.
- **etcd** etcd ist der primäre Datenspeicher von Kubernetes und sichert alle Zustände eines Cluster.
- **Scheduler**: Der Scheduler ist zuständig für die Verteilung und Ausführung von Pods auf Nodes.
- **Controller Manager**: Der Controller Manager reagiert auf Ausfälle von Nodes, erhält die korrekte Anzahl von Replikationen eines Pods und verbindet Services miteinander.

[1]Seit Kubernetes v1.20, ist die korrekte Bezeichnung für die Master Node Control Plane [11]

Node

Eine Node[2] ist eine Hardware Einheit, die je nach Kubernetes Einrichtung eine VM, physische Maschine oder eine Instanz in einer privaten oder öffentlichen Cloud darstellen kann. Dieser umfasst folgende Komponenten [12]:

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.

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.

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.

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 [13].

- **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.

[2]Um den Sprachfluss zu wahren wird der englische Begriff Node, als Kubernetes Ressourcenobjekt nicht übersetzt. Die Übersetzung Knoten findet jediglich 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 [14]

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 [15]. 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 [16].

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 [16]:

- **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 [16]

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 [17]. Die Anforderung für die Implementierung eines Ingress ist der Ingress-Controller, eine Vielzahl an Optionen dafür wird in der Dokumentation aufgelistet [18]. 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 [17]. Die übliche Verwendung eines Controllers beeinhaltet die Lastenverteilung durch weiterleiten des Datenverkehrs an Services. Diese Kommunikation findet, wie auch bei dem NGINX Ingress Controller [19] in der Anwendungsschicht des OSI-Schichtenmodells statt und ermöglicht, dadurch die Lastenverteilung von öffentlichen Endpunkten zu internen Pods in einem Cluster [20]. 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 [17]:

- **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: nginx-service
            port:
              number: 80
```

Quellcode 2.3: ingress.yaml [17]

2.2.6 Lightweight Kubernetes

Ligthweight Kubernetes auch K3s genannt ist eine Open-Source 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. Durch die Verschlankung der Distribution ist der ideale Anwendungszweck für IoT oder Edge-Devices mit wenig Rechenleistung. Denn die minimalen Systemanforderungen für Hostsysteme liegen bei einem 512MB-RAM Speicher und einer Pi4B BCM2711, 1.50 GHz CPU[3] [22]. Der hauptsächliche Verwendungszweck von k3s liegt in IoT und Edge-Devices, da unwichtige Kubernetes Inhalte entfernt wurden. [23]. Trotz dieser Reduzierung, bleiben die Kernfunktionalitäten von Kubernetes erhalten und werden so weit wie möglich parallel auf dem neusten Stand gehalten [24].

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 [25]. Weiterhin wird im Gegensatz zu Kubernetes containerd weiterhin unterstüzt und kommt vorinstalliert mit Kubelet [23]. Zwei weitere Unterschiede werden näher erläutert:

[3] Einplatinencomputer Raspberry Pi 4B, basierend auf ARM [21]

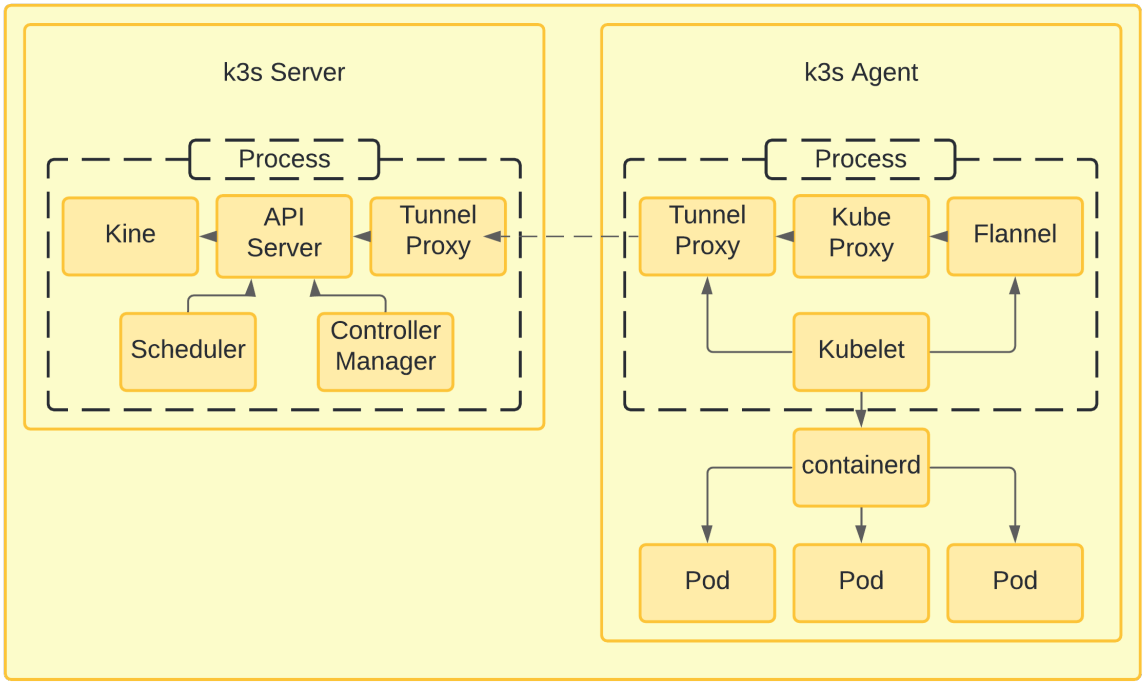

Abbildung 2.5: K3s Architektur in Anlehnung an [23].

Kine das Akronym steht für 'Kine is not etcd' und ist eine Abstraktionsschicht für etcd, welche sqlite, Postgress, Mysql und dqlite übersetzt [24]. 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 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 [26].

2.2.7 Rancher

In diesem Unterabschnitt 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 [27].

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 [27, 28].

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 [29].

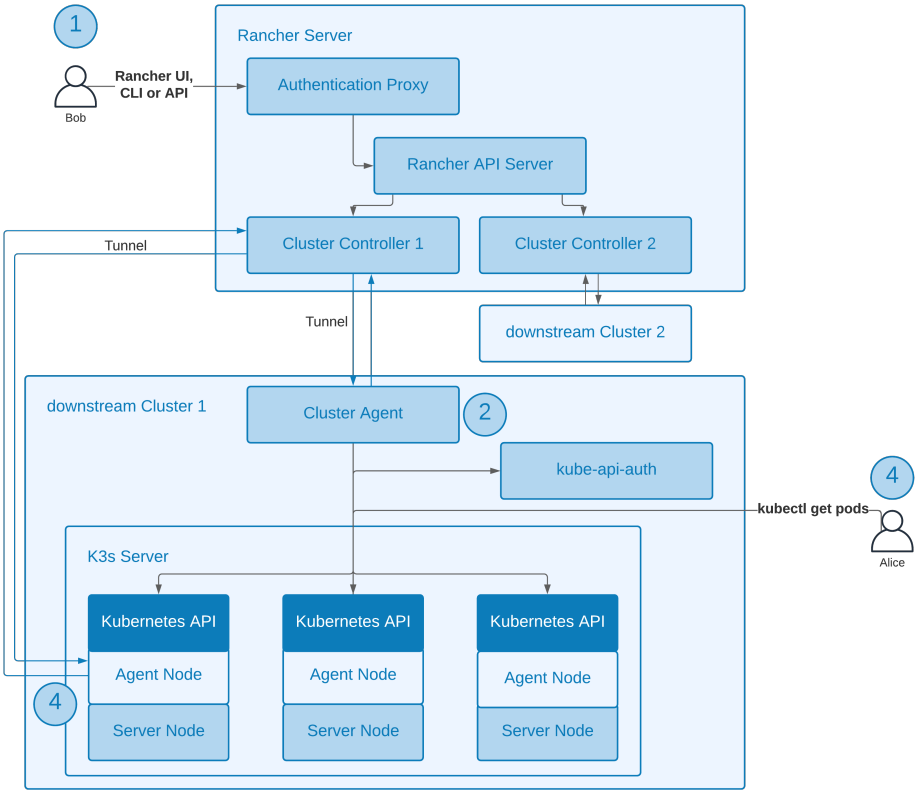

Abbildung 2.6: Rancher Server Kommunikation mit einem downstream k3s Cluster, überarbeitete Abbildung von [30]. (Im Sinne der späteren Architektur nachgebildet)

Die Abbildung 2.6 zeigt den Vorgang von Zwei Benutzern, die auf ein von Rancher verwalteten downstream k3s Cluster[4] zugreifen. Die nachfolgende Beschreibung aus der Dokumentation gibt die einzelnen Schritte mit der in der Abbildung nummerierten Posten wieder [30].

1. Zuerst authentifiziert sich Bob mit seinen Benutzerdaten bei dem Authentifizierungs-Proxy an seinem Rancher Server. Dieser Proxy leitet den Aufruf, über eine Kommandozeile oder der Rancher Benutzeroberfläche mit der ausgewählten downstream-Cluster Instanz weiter und führt diese aus. Dafür wird vor dem weiterleiten des Aufrufs, der angemessene Kubernetes Impersonation Header

[4]Die offiziele Bezeichnung für ein Kubernetes Cluster unter Rancher ist **downstream Cluster** [31]

gesetzt, welcher sich als Service Account der Rancher Instanz ausgibt und je nach Benutzerstatus reagiert.

2. Die Übertragung des Aufrufs erfolgt über einen Cluster-Controller auf dem Rancher Server und dem parallel laufendem Cluster-Agent des 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[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-Haken realisiert, der die Verbindung zwischen Alice Rechner aufbaut. Dies ermöglicht die Verwendung von Befehlszeilentools, wie Kubectl und Helm.

2.2.8 Hybrid Cloud

Eine Hybrid-Cloud ist eine Kombination aus Öffentlichen-Cloud-Diensten 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 [33].

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, die Entwicklern eine Landschaft bietet Hardware flexibel zu verändern. Dabei ist die Nutzung eines einzigen Betriebssystems ratsam, um Abhängigkeiten bei der Automatisierung von cloudnativen Anwendungen zu verhindern. Die Verwaltung erfolgt, dabei mit einer Container-Orchestierungsplattform, wie Kubernetes und ermöglicht die nahtlose Implementierung von Cloud-Umgebungen [33].

2.3 Microservice

Im Folgenden wird der Microservice Architektur-Stil und dessen Eigenschaften näher erläutert. Als Hauptquelle dient der häufig zitierte Artikel [34] von Fowler.

2.3.1 Begriffserklärung

Fowler beschreibt den Microservice Architektur-Stil als eine Entwicklung von einer einzigen Anwendung die aus einer Reihe unabhängiger Dienste besteht. Die Kommu-

[5] Ein Rancher DaemonSet zur Interaktion mit Nodes, nicht zu verwechseln mit dem Node-Agent von k3s [32].

nikation 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. Zusammenhängende Dienste werden minimal zentral gehalten und können in unterschiedlichen Programmiersprachen oder Technologien realisiert werden [34].

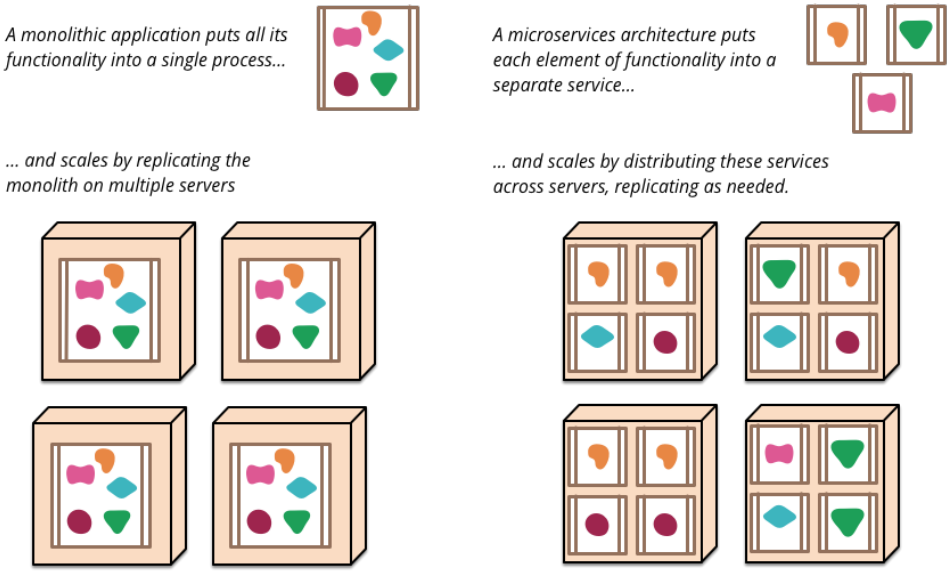

Abbildung 2.7: Gegenüberstellung von Monolithen und Microservices [34]

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 argumentiert Fowler, dass mit dem Zuwachs an Cloud-Technologien dieser Ansatz immer frustrierender für Entwickler ist, denn bereits kleine Änderungen an einem Modul benötigt einen neuen Software-Build und Auslieferungsprozess. Weiterhin merkt Fowler 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 [34].

2.3.2 Charakteristiken

Eine Microservice Architektur prägt sich durch bestimmte Charakteristiken aus. Die Architektur muss nicht zwingend alle, in diesem Abschnitt beschriebenen, Eigenschaften erfüllen. Jedoch sollte ein Großteil der Konzepte in einer Microservice Architektur aufweisbar sein [34]. Die folgenden Unterabschnitte erläutern diese Charakteristiken etwas näher.

Komponententrennung durch Dienste

Fowler definiert Komponenten einer Software wie folgt:

> „Eine Komponente ist eine Softwareeinheit, die unabhängig austauschbar und erweiterbar ist.“ [35]

Dienste einer Microservice Architektur stellen Softwarekomponenten dar, die out-of-process mittels Web-Service-Anfragen oder Remote Procedural Calls interagieren. Bibliotheken hingegen beschrieben einen Verbund aus mehreren Komponenten, die in-memory Funktionsaufrufe nutzen. Der resultierende Vorteil ist das Dienste unabhängig voneinander verändert und ausgeliefert werden können. Denn bei Prozessen mit mehreren eingebundenen Bibliotheken, muss die gesamte Anwendung neu aufgesetzt werden [34].

Dadurch wird der Fokus auf die Entwicklung von unabhängigen Diensten, umso wichtiger, da die Veränderung an kooperierenden Schnittstellen zum Ausfall anderer Dienste führt. Dem Entgegenzuwirken müssen Schnittstellen gut koordiniert werden und eine starke Kohäsion gewährleisten. Service Contracts der jeweiligen Dienste müssen sinnvoll gestaltet werden. Weiterhin müssen Schnittstellen Grobkörniger entworfen werden, um den höheren Ressourcenverbrauch zur in-memory Variante auszugleichen [34, 36].

Ein Dienst kann jedoch trotzdem aus mehreren Prozessen bestehen, beispielsweise einem Anwendungsprozess mit einer Datenbank, die nur von der Anwendung genutzt wird [34, 36].

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 die Organisierung, um die Business Capability. Entwickler arbeiten Funktionsübergreifend in allen Bereichen der Softwareentwicklung und bringen vielfältige Kompetenzen mit. Der Grund dafür ist das bei Konstellationen mit einseitiger Softwarekompetenz, kleinste Änderungen zu Teamübergreifende Projekte und den damit verbundenen Kosten führt. Effiziente Entwickler werden sich immer für den Weg des geringsten Widerstands entscheiden und ihre Logik dort implementieren, zu der das Team Zugang hat [34].

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 “ [37]

Dem Gedanken nach übernimmt das Entwicklungsteam die vollständige Produktion der Software und übergibt diese nicht am Ende 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. Die Mentalität dabei ist, einzelne Funktionen und Aufgaben der Anwendung nicht als eine Liste zu betrachten, sondern vielmehr die Erfüllung des Gedankens zur die strukturierung der Business Capability [34].

Intelligente Endpunkte statt komplexer Infrastruktur

Die Kommunikation von Diensten über Endpunkten soll so weit wie möglich entkoppelt und kohäsiv sein. Anwendungen im Microservice-Stil enthalten ihre eigene Logik und agieren, als ein Filter für das Empfangen, Verarbeiten und Beantworten einer Anfrage. Die Umsetzung erfolgt dabei mit RESTful-Protokollen für die Kommunikation über HTTP. Eine weiterer Ansatz zur Kommunikation ist das Messaging. Die gewählte Infrastruktur muss hier nicht mehr als einen rudimentären Nachrichtenaustausch gewährleisten. Der Schwerpunkt liegt dabei immer auf dem Endpunkt, welcher Nachrichten seltener erhält, aber mit einem gewonnenen Mehrwert umwandelt [34].

Dezentrale Verwaltung

Die dezentralisierung einer Anwendung in Softwarekomponenten, ermöglichen 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 größeren Handlungsspielraum und können bessere Werkzeuge für spezifische Probleme verwenden [34].

Dezentrale Datenmanagement

Die Dezentralisierung von Daten geschieht auf höchster Ebene und Abstrahiert diese für Kontext basierende Modelle. Die Integration solcher Modelle wird durch die unterschiedliche Auffassung verschiedener System erschwert. Attribute werden doppelt oder schlichtweg falsch für Schlüsselbegriffe belegt. Eine Anwendung mit getrennten Softwarekomponenten erschwert dieses Unterfangen nur noch [34]. Weshalb es Sinnvoll ist einen „Bounded Context“ zu definieren, welches zur Darstellung von Wechselwirkungen eines Modells, innerhalb größerer Teams dient [38].

Automatisierung von Infrastruktur

Das testen, ausliefern und bereitstellen von Microservices erfolgt automatisch [34]. Dieses Thema fällt in den Bereich Continuous Delivery und wird in dieser Arbeit nicht weiter behandelt.

„Design for failure “

Softwarekomponenten müssen den Ausfall von anderen Diensten tolerieren. Event basierte Kommunikation führt oft zu Fehlverhalten und kann durch Überwachungstools präventiv verhindert werden [34].

2.4 Zusammenfassung

Kapitel 3

Anforderungen

Das vorherige Kapitel beschreibte die nötigen Technologien für die Realisierung und Ausführung einer Microservice Architektur. Dieses Kapitel beschäftigt sich nun mit der Analyse

3.1 Analyse

3.1.1 Anwendungsszenario

3.2 Spezifikation

3.2.1 Use-Case

Kapitel 4

Konzeption

Das folgende Kapitel beschreibt die Konzeptionellen

4.1 Herausforderungen

4.2 KI-Anwendungsfall

4.2.1 Gesichtserkennung

Kapitel 5

Implementierung der Architektur

5.1 Aufbau der Implementierung

5.1.1 Hardware-Layer

5.1.2 Software-Layer

5.1.3 Betriebssystem

5.2 Konfiguration und Einrichtung

5.2.1 Virtueller Privater Server

5.2.2 Domain

5.2.3 SSL-Verschlüsselung

5.3 Frameworks und Bibliotheken für Microservices

5.3.1 Flask

5.3.2 Gunicorn

5.3.3 SocketIO

5.3.4 OpenCV

5.3.5 MongoDB

5.4 Gesichtserkennung

5.4.1 Alignment

5.4.2 Training

5.4.3 Modell

5.5 Containerisierung

Kapitel 6

Ergebnisse

6.1 Microservice

6.1.1 Frontend-Serivce

6.1.2 Backend-Serivce

6.1.3 Loadbalancer

6.1.4 Kubernetes Cluster

Kapitel 7

Diskussion und Ausblick

7.1 Einschränkungen

7.2 Diskussion

7.3 Ausblick

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

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

[11] "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

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

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

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

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

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

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

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

[19] "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/

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

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

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

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

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

[25] "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

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

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

[28] 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

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

[30] “Overview.” [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/

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

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

[33] “hybrid-cloud,” May 2021. [Online]. Available: https://www.ibm.com/de-de/cloud/learn/hybrid-cloud

[34] “Microservices.” [Online]. Available: https://martinfowler.com/articles/microservices.html

[35] “Softwarecomponent.” [Online]. Available: https://martinfowler.com/bliki/SoftwareComponent.html

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

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

[38] “Boundedcontext.” [Online]. Available: https://martinfowler.com/bliki/BoundedContext.html

Abbildungsverzeichnis

Tabellenverzeichnis