THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Bachelorthesis

Auswahl und Erweiterung eines Container-Orchestrierungs-Systems zum automatischen Deployment einer Kommunikationsplattform

zur Erlangung des akademischen Grades

Bachelor of Science

eingereicht im Fachbereich Mathematik, Naturwissenschaften und Informatik an der Technischen Hochschule Mittelhessen

von

Jonas-Ian Kuche

8. Mai 2022

Referent: Prof. Dr. Frank Kammer

Korreferent: Prof. Thomas Friedl

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Gießen, den 8. Mai 2022 Jonas-Ian Kuche

Zusammenfassung

Server-Anwendungen, welche für eine Vielzahl von Kunden angepasst installiert werden müssen, führen zu einem hohen Administrationsaufwand. Die Verwendung eines *Container-Orchestrierungs-Systems* kann dabei helfen, diesen Aufwand zu reduzieren. So ist es möglich die Installation einer Anwendung zu automatisieren, ohne dabei auf Anpassbarkeit der Installation verzichten zu müssen. Darüber hinaus unterstützen sie die Installation der Anwendung über mehrere Server hinweg, um die Belastbarkeit dieser zu erhöhen. Auch können diese Fehler, welche ansonsten zu Ausfällen führen würden, automatisch etwa durch Neustarten der Anwendung behandeln. Deshalb werden im folgenden am Beispiel der Kommunikationsplattform „Audimax“ Kriterien für die Auswahl eines *Container-Orchestrierungs-Systems* ausgearbeitet, beispielhaft angewendet und die Umsetzung eines automatischen Installationssystems auf Basis dessen beschrieben.

Inhaltsverzeichnis

1 Einführung

In folgendem wird die Auswahl und Erweiterung eines Container-Orchestrierungs-Systems zum automatischen Deployment der Kommunikationsplattform Audimax beschrieben. Audimax vereint verschiedene Funktionen: das Abhalten von Konferenzen, das Kommunizieren über einen Chat, das Erfassen von Terminen über einen Kalender, das Hochladen von Dateien, das Verwalten von Teilnehmern in verschiedenen Gruppen und das Speichern von Informationen in einem Pinboard. Ein Einsatzgebiet von Audimax ist dabei die digitale Lehre, also die Verwendung in Schulen und Hochschulen. Audimax kann aber auch für die Kommunikation in anderen Bereichen wie etwa Unternehmen eingesetzt werden. Audimax besteht dabei aus verschiedenen Softwarekomponenten, welche verschiedene Aufgaben innerhalb des Systems übernehmen.

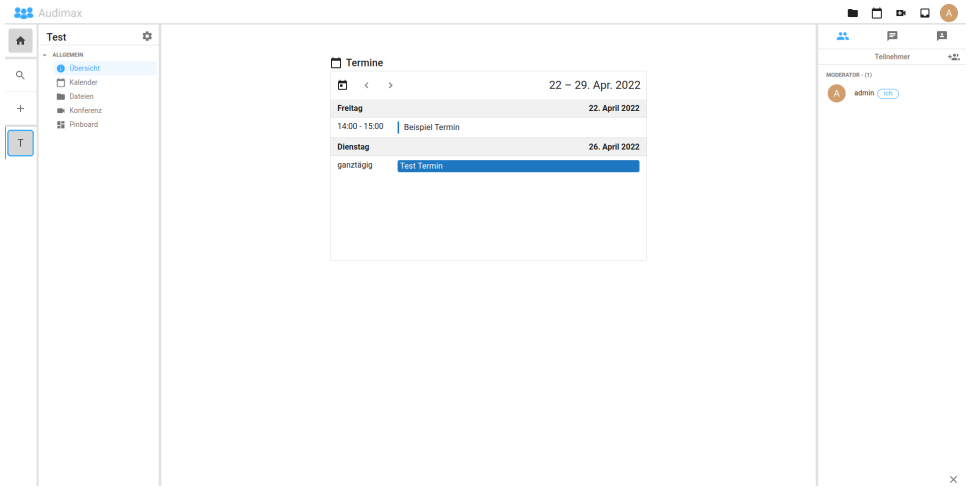

Abbildung 1.1: Screenshot von Audimax, zeigt unter anderen die Termine des Nutzers.

1.1 Motivation

Beim Betreiben von Audimax muss beachtet werden, dass es aus Gründen des Datenschutzes sinnvoll ist, für jeden Kunden, etwa eine Hochschule, eine eigene Audimax-Installation einzurichten. Diese wird im Folgenden auch Audimax-Instanz genannt. So sind alle Nutzerdaten voneinander getrennt. Dies bietet den Vorteil, dass im Falle eines Datenlecks der Schaden begrenzt wird. Der Nachteil wiederum ist, dass alle Audimax

Komponenten für jeden Kunden installiert, aktualisiert und überwacht werden müssen. Die Installation umfasst das Konfigurieren und Starten aller Softwarekomponenten von Audimax. Beim Erscheinen einer neuen Audimax-Version müssen alle bestehenden Audimax-Installationen erneut konfigurieren und gestartet werden. Die Überwachung der Funktionalität dieser Software-Komponenten erfolgt durch das kontinuierliche Beobachten der Programmausgaben, des Ressourcenverbrauchs und sonstiger Aktivitäten von Audimax. Bei einer Abweichung von Normwerten sollen dabei die Administrierenden per E-Mail oder Chat-Nachricht benachrichtigt werden.

Viele Anwender von Audimax haben ähnliche Bedürfnisse. Zum Beispiel benötigen die meisten Schulen einen ähnlichen Funktionsumfang. Daher sind die meisten Audimax-Installationen sehr ähnlich konfiguriert, aber benötigen trotzdem kleine individuelle Anpassungen für jeden Kunden.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, ein System zu konzipieren und dessen Realisierung zu beschreiben, welches die folgenden Anforderungen erfüllt:

Erstes Ziel - Automatisierung: Das System soll die Installation und Aktualisierung einer Audimax-Instanz automatisieren. Dabei sollen die Softwarepakete der jeweiligen Audimax-Komponenten heruntergeladen, die Konfigurationsdateien erstellt und die Komponenten gestartet werden.

Zweites Ziel - Überwachung: Das System soll die Audimax-Instanzen überwachen und Fehler, falls möglich, selbstständig beheben. Sollte ein selbstständiges Beheben nicht möglich sein, sollen die Administrierenden benachrichtigt werden.

Drittes Ziel - Skalierung: Das System soll zur optimalen Nutzung der zur Verfügung stehenden Ressourcen in der Lage sein, mehrere Audimax-Instanzen auf demselben Server auszuführen. Auch soll zur Erhöhung der möglichen gleichzeitigen Nutzeranzahl eine Audimax-Instanz auf mehreren Servern verteilt ausgeführt werden können.

Viertes Ziel - Konfiguration: Die Audimax-Instanzen sollen geteilte Basiskonfigurationen verwenden, welche für einzelne Instanzen angepasst werden können. Dies verhindert, dass die Konfigurationen für kundenspezifische Änderungen kopiert werden müssen, was unnötigen Arbeitsaufwand im Fall einer globalen Konfigurationsänderung verursachen würde.

1.3 Vorgehensweise

Zuerst wird in Kapitel 2 auf grundlegende Begriffe aus dem Bereich der *Container-Virtualisierung*, *Microservices* und *Kubernetes*, welche zum Verständnis der nachfolgenden Kapital notwendig sind, eingegangen. Als Nächstes wird in Kapitel 3 ein *Container-Orchestrierungs-System* ausgewählt. Dies ist ein System, welches eigenständig Softwarepakete, sogenannte *Container*, meist über mehrere Computer hinweg verwaltet. Das ausgewählte *Container-Orchestrierungs-System* soll zum Erreichen der Ziele eins bis drei verwendet werden und sollte das Erreichen von Ziel vier ermöglichen. Dazu werden Anforderungen an dieses definiert. Daraufhin werden einige praktikable Systeme anhand dieser Anforderungen bewertet und ein Vergleich angestellt, mithilfe dessen ein System ausgewählt wird. Danach wird das Vorgehen bei der Erstellung und Durchführung von Lasttests erläutert. Diese helfen bei der Bestimmung der benötigten Rechner-Ressourcen pro Nutzer. In Kapitel 5 wird die Einrichtung des zuvor ausgewählten Container-Orchestrierungs-Systems beschrieben und die notwendigen Erweiterungen, um das vierte Ziel zu erreichen, an diesem vorgenommen. Zuletzt wird in Kapitel 6 die Erfüllung der vier Ziele evaluiert und ein Fazit gezogen.

1.4 Struktur von Audimax

Jeder Kunde erhält eine Audimax-Installation, welche aus mehreren zusammenarbeitenden Software-Komponenten besteht. Bei manchen dieser Komponenten handelt es sich um Standardsoftware, welche von Dritten entwickelt wurde und in Audimax verwendet wird. Bei anderen handelt es sich um Eigenentwicklungen, welche speziell für die Verwendung in Audimax entwickelt wurden. Auch gibt es externe Komponenten, mit welchen Audimax zwar interagiert, die aber nicht Teil einer Audimax-Installation sind.

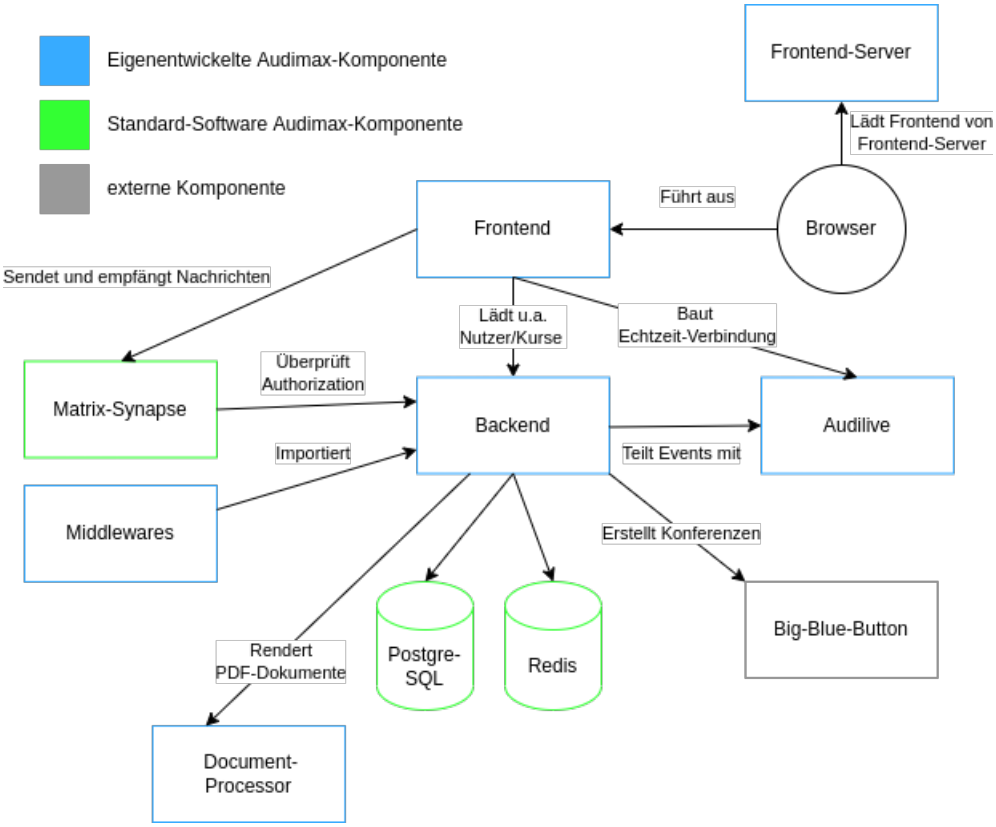

Abbildung 1.2: Audimax Struktur-Diagram

Frontend:

Das Audimax-Frontend ist das Nutzerinterface, über welches die Nutzer mit Audimax interagieren. Es ist mithilfe des *Nutzerinterface-Frameworks* React entwickelt und läuft im Browser des Nutzers. Ein sogenannter *Webserver* wird verwendet, um die Dateien des Frontends an den Browser des Nutzers zu senden, wenn dieser die Adresse einer Audimax-Installation besucht. Das Frontend lädt Informationen von verschiedenen Audimax-Komponenten.

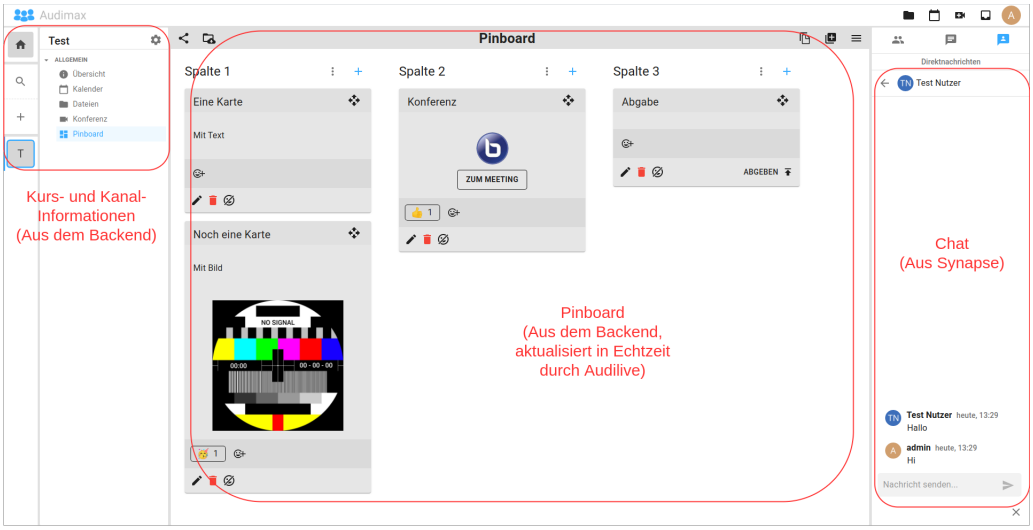

Abbildung 1.3: Screenshot des Audimax-Frontends mit Beschreibung der jeweiligen Datenquelle in Rot

Backend:

Das Audimax-Backend ist in der Programmiersprache PHP geschrieben und verwendet einen Apache Webserver. Es ist eine zentrale Komponente von Audimax, welche große Teile der Betriebslogik von Audimax enthält und die Informationen, welche im Frontend angezeigt werden, wie etwa die Gruppen, Kanäle, Nutzer und Pinboard-Inhalte verwaltet. Auch erstellt das Backend BigBlueButton-Konferenzen und authentifiziert die Interaktion zwischen dem Frontend und den Audimax-Komponenten.

Synapse:

Synapse ist eine Standard-Server-Software für das Chatprotokoll *Matrix*. Matrix definiert zum einen, wie ein *Client*, also ein Programm, was von einem Nutzer zum Versenden und Empfangen von Chat-Nachrichten verwendet wird, und ein *Server*, ein Programm, welches Nachrichten zwischen mehreren Clients vermittelt, miteinander kommunizieren. Zum anderen definiert es auch, wie verschiedene Server, untereinander kommunizieren können. Dies ermöglicht Nutzern verschiedener Server untereinander zu kommunizieren. Das Audimax-Frontend fungiert im Fall von Audimax als Matrix-Client.

Audilive:

Audilive ist der Teil von Audimax, welcher für die Echtzeitkommunikation zwischen den Server-Komponenten und dem Audimax-Frontend verantwortlich ist. Diese Echtzeitkommunikation wird zum Beispiel im Audimax-Pinboard verwendet, um Änderungen am Pinboard durch einen Nutzenden den anderen Nutzenden des Pinboards in Echtzeit, ohne Neuladen des Pinboards, anzuzeigen. Dies kann nicht über das Audimax-Backend erfolgen, da die Programmiersprache, in welcher dieses geschrieben ist, nicht für

Echtzeit-Verbindungen geeignet ist. Audilive ist in der Programmiersprache TypeScript geschrieben und wird mithilfe der JavaScript-Laufzeitumgebung Node.js ausgeführt, welche ideale Voraussetzungen für Echtzeit-Verbindungen bietet.

***-Middleware:**

Middlewares dienen der Integration von Audimax in andere Dienste und können unter anderem Nutzer, Gruppen und Termine von diesen importieren. Eine weitere Aufgabe ist die Authentifizierung der Nutzer. Ein Beispiel für eine Middleware ist die *OIC-Middleware*, welche die Authentifizierung von Nutzern über *Open-ID-Connect* ermöglicht. Open-ID-Connect ist ein System, welches es bei einem *Authorization Server* genannten Dienst angemeldeten Nutzern ermöglicht, sich bei einem anderen Dienst, ohne erneute Angabe von Zugangsdaten anzumelden. Auch können bei dieser Anmeldung Nutzerinformationen wie Gruppenzugehörigkeit abgefragt werden.

Document Processor:

Der *Document Processor* ist eine in der Programmiersprache *Go* geschriebene Anwendung, welche dynamisch PDF-Dateien anhand von Vorlagen erzeugt. Dieser wird zum Beispiel verwendet, um PDF-Dateien mit QR-Codes zu erzeugen, welche mit einer Smartphone-Kamera gescannt werden können und so den Nutzer ohne Eingabe des Passworts anmelden. Dabei kommuniziert das Backend mit dem Document-Processor, um diesem die Informationen, die in die Vorlage eingefügt werden sollen, zu senden.

Datenbanken und Caches:

Das Audimax-Backend und Synapse benötigen zum Operieren jeweils einen Datenbankserver (*PostgreSQL*) und einen Cache-Server (*Redis*). In den Datenbanken werden die Anwendungsdaten von Audimax bzw. Synapse gespeichert. Redis wird vom Audimax-Backend und Synapse dazu verwendet, um zeitaufwendig zu erhaltene Daten für einen gewissen Zeitraum zu speichern. Dies erfolgt, um die Geschwindigkeit der jeweiligen Anwendung zu erhöhen und wird als Caching bezeichnet.

2 Grundlagen

Für ein Verständnis der Container-Orchestrierungs-System-Auswahl ist es wichtig, die zentralen Konzepte der zur Auswahl stehenden Systeme zu verstehen. Daher wird zuerst auf die den meisten Containersystemen zugrundeliegende Softwarearchitektur eingegangen.

Dieses *Microservices* genannte Konzept dient der Modularisierung von Software und teilt die Softwarekomponenten in einzelne Prozesse auf, welche über eine Schnittstelle miteinander kommunizieren. Dies hat den Vorteil, dass die einzelnen Dienste unabhängig voneinander entwickelt, aktualisiert oder sogar durch andere Implementierung derselben Schnittstelle ausgetauscht werden können. Ferner kann jeder Dienst mit unterschiedlichen Technologien implementiert werden. So kann für jede Aufgabe die Programmiersprache und das Framework gewählt werden, welche zu dieser passen. Auch können einzelne Microservices unabhängig voneinander skaliert werden. Dies erlaubt es Komponenten, welche hoher Last ausgesetzt sind, zu skalieren, ohne dabei unnötige Kapazitäten an anderer Stelle zu schaffen. Meist werden Microservice-Prozesse innerhalb von Containern ausgeführt, um deren Isolation voneinander zu gewährleisten. [Wol18, Seite 2]

2.1 Container-Grundlagen

Das am weitesten verbreitete Container-Virtualisierungs-System ist *Docker*. Anhand dessen werden im Folgenden die grundlegenden Konzepte der *Containervirtualisierung* erklärt.

Image

Ein *Image* kann als Blaupause für das Starten eines Containers verstanden werden. In seiner grundlegendsten Form ist ein Image ein Dateisystem-Abbild eines Betriebssystems inklusive Anwendungen und Konfigurationsdateien. [Bur19, Seite 15]

Meist liegt dieses in Form eines speziellen Formates, dem *OIC-Image-Format* vor. Dies ist ein sogenanntes Layer-Format, bei welchem einzelne Dateisystem-Schichten „übereinander“ gelegt sind. Dateien werden immer in die oberste Schicht geschrieben und aus der obersten Schicht, die eine bestimmte Datei enthält, gelesen. Ein solches

Layer-Format ermöglicht es auf Basis eines zugrundeliegenden Betriebssystemimages mehrere Anwendungs-Images zu bauen, ohne dass das Betriebssystem doppelt gespeichert wird. Hierbei ist das Betriebssystem der unterste Layer und die jeweiligen installierten Softwarepakete, Konfigurationsdateien, etc. befinden sich in den darüber liegenden Layern. Dies reduziert den Speicherverbrauch, da der Betriebssystem-Layer nur einmal gespeichert werden muss. [Bur19, Seite 15-16] Container-Images sind oft Anwendungs-Images. Das heißt, sie enthalten nur die Anwendung, die in diesem Container ausgeführt werden soll, und nicht etwa Systemdienste. [Bur19, Seite 16]

Zumeist werden *Images* aus sogenannten *Dockerfiles* gebaut. Diese heißen zwar Dockerfiles, werden aber von den meisten Containersystemen zur Image-Erstellung genutzt. In einem Dockerfile werden die jeweiligen Image-Layer beschrieben. Es wird zuerst ein Start-Image, meistens ein Image nur mit Betriebssystem, angegeben. Danach wird für jeden folgenden Layer ein Kommando, welches in dem vorherigen Layer ausgeführt werden soll, um den darauffolgenden Layer zu erzeugen, angegeben. Des Weiteren ist es möglich, Dateien aus dem bauenden System in den Container zu kopieren und den Nutzer sowie Umgebungsvariablen zu ändern. [Bur19, Seite 16-18]

```
FROM debian:11
RUN apt-get update && apt-get install -y curl
CMD ["curl", "https://example.org"]
```

Listing 2.1: Ein Dockerfile, welches während des Bauens `curl` installiert und dies beim Starten des Conainers ausführt

Mithilfe dieses *Dockerfile*, kann nun mit folgendem Kommando ein Docker-Image erstellt werden: `docker build -t testimagename .`. Dabei muss ein Name angegeben werden, mithilfe dessen das Image beim Starten angegeben werden kann. Dieser ist in diesem Fall `testimagename`.

Container

Ein *Container* kann als eine leichtgewichtige virtuelle Maschine verstanden werden. Dabei können in diesen, wie bei virtuellen Maschinen, unabhängig vom Host-System Dateien gespeichert und Programme gestartet werden. Im Gegensatz zu einer virtuellen Maschine verwendet ein Container den Betriebssystemkern des Hosts-Systems. Dies führt zur effizienteren Nutzung von System-Ressourcen, bringt allerdings den Nachteil mit sich, dass innerhalb von Containern nur Betriebssysteme, welche denselben Betriebssystemkern wie das Host-System verwenden, laufen können [Dei18, Seite 800-801].

Beim Starten eines Containers wird über das Image ein weiterer Dateisystem-Layer gelegt. In diesem Layer werden Dateien gespeichert, welche von dem im Container laufenden

Programmen geändert wurden. Dieser Layer geht verloren, wenn der Container gelöscht wird. Um persistent Daten zu halten werden sogenannte *Volumes* eingebunden. [Doc22i]

Um einen Container auf Basis des Images mit dem Namen `testimagename`, welches zuvor gebaut wurde, zu starten, kann der Befehl `docker run testimagename` verwendet werden.

Volume

Innerhalb von Containern gespeicherte Daten werden gelöscht, sobald der Container gelöscht wird. Diese Eigenschaft wird als *ephemeral* bezeichnet. Dies ist nicht sinnvoll, wenn Daten langfristig gespeichert werden müssen, wie etwa Einträge in einem Datenbanksystem oder das Profilbild eines Nutzers. Für solche Fälle werden Volumes verwendet. Volumes können für Ordner innerhalb eines Containers eingerichtet werden. Dateien, welche unterhalb dieser Pfade abgelegt werden, werden nicht im Container gespeichert, sondern im Volume, und sind so auch nach der Löschung und Neuerstellung eines Containers noch verfügbar. Volumes können sowohl innerhalb des Dateisystems des Systems, welches den Container ausführt, gespeichert werden als auch in Netzwerk-Dateisystemen. Letzteres hat den Vorteil, dass Container nicht an das Host-System, welches ihre Volumes zur Verfügung stellt, gebunden sind, sondern auf jedem System, welches Zugriff auf das Netzwerk-Dateisystem hat, gestartet werden können. [Kub22q] Volumes sind nur ein kleiner Teil der Konfiguration eines Containers, um diese zu vereinfachen und wiederverwendbar zu machen werden sogenannte *Container-Orchestrierungs-Systeme* verwendet.

Container-Orchestrierungs-System

Mithilfe eines *Container-Orchestrierungs-Systems* kann das Erstellen und Verwalten von Containern automatisiert werden. Ein *Container-Orchestrierungs-System* erlaubt es die Konfiguration eines oder mehrerer Container in einer Datei zu speichern. Dabei können unter anderen das zu verwendende Image, das in diesem auszuführende Kommando, die Argumente und Umgebungsvariablen für dieses und die zu mountenden Volumes eingestellt werden. Das Container-Orchestrierungs-System kann dann mithilfe dieser Datei die gewünschten Container erstellen. Viele Container-Orchestrierungs-Systeme bieten Funktionen, welche über das Erstellen und Verwalten von Containern hinausgehen. Sie erlauben etwa das Verwalten von verwendeten Ressourcen wie Volumes und Netzwerke oder das Erstellen von Containern über mehrere Server hinweg. [Red19]

Ein Beispiel für ein solches Container-Orchestrierungs-System ist *docker-compose*. Dieses kann mithilfe einer in der Auszeichnungssprache *YAML* geschriebenen Datei Container erstellen.

```yaml
version: "3"

services:
  test:
    image: testimagename
    command: ["curl", "-o", "/example/index.html",
              "https://example.org"]
    volumes:
    - ./example:/example
```

Listing 2.2: Eine docker-compose-YAML-Datei

Wird in dem Ordner, im welchem sich diese YAML-Datei befindet, `docker-compose up` ausgeführt, startet `docker-compose` einen Container aus dem `testimagename` genannten Image, mountet in diesen den Unterordner `example` und startet innerhalb des Containers einen `curl` Prozess, welcher die example.org Webseite herunterlädt und im gemounteten Volume speichert.

Container Runtime Environment (CRE)

Als Container-Laufzeitumgebung, kurz *CRE*, wird das System bezeichnet, welches den *Container* verwaltet, also die benötigten Aktionen vornimmt, um den Container im Betriebssystem zu erstellen und zu konfigurieren. Es wird vorwiegend ein Kommandozeilenwerkzeug wie zum Beispiel *docker* oder ein *Container-Orchestrierungs-System* wie zum Beispiel *Kubernetes* verwendet, um mit diesem zu interagieren. [Kub20]

2.2 Kubernetes-Grundlagen

Im Folgenden werden einige grundlegenden Konzepte von *Kubernetes*, welche zum Verständnis der nächsten Kapitel notwendig sind, erklärt.

Kubernetes ist ein laufzeitunabhängiges *Container-Orchestrierung-System*. Es ermöglicht das Zusammenschalten von verschiedenen Servern, in der Kubernetes Terminologie *Nodes* genannt, in ein *Cluster*, auf welchen dann *Arbeitslasten* (Container) ausgeführt werden können. Diese Arbeitslasten können auf verschiedene Art und Weise detailliert konfiguriert und verwaltet werden. Dabei wird besonderen Wert auf Stabilität und Skalierbarkeit gelegt. Das heißt, Arbeitslasten können unabhängig vom zugrunde liegenden Node verwendet und im Falle des Ausfalls eines Nodes auf andere verlegt werden. Zusätzlich können bei Fehlern innerhalb der Anwendung auch einzelne Arbeitslasten abgeschaltet, skaliert und Fehlerzustände automatisch behandelt werden. [Bur19, Seite 13]

Dies wird dadurch erreicht, dass Kubernetes fortlaufend den Soll- und Istzustand des Clusters miteinander vergleicht und bei Abweichungen automatisch probiert den Sollzustand zu erreichen. Für eine solche Abweichung gibt es in der Regel zwei mögliche Ursachen. Es kann sich der Sollzustand ändern. Dies ist etwa der Fall, wenn ein neuer Container zum Sollzustand hinzugefügt wird. Oder es kann sich der Istzustand ändern. Dies ist unter anderem der Fall, wenn ein Container abstürzt oder ein Knoten des Clusters ausfällt. In allen diesen beschriebenen Fällen, würde Kubernetes, um den Sollzustand zu erreichen, einen neuen Container starten. Dieses Vorgehen gibt Kubernetes sehr gute Möglichkeiten zur „Selbstheilung“. [Bur19, Seite 4-5] Der Sollzustand eines Clusters wird durch sogenannte *Kubernetes-Objekte* beschrieben.

Kubernetes-Objekt

Jedes *Kubernetes-Objekt* beschreibt einen Teil des Sollzustandes eines Kubernetes-Clusters. Kubernetes-Objekte werden meist in der Auszeichnungssprache *YAML* erstellt und bearbeitet.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: audimax
  labels:
    app: audimax
    component: backend
spec:
#...
```

Listing 2.3: Ein Ausschnitt eines Kubernetes-Objektes vom Typ Deployment

Ein *Kubernetes-Objekt* hat immer folgende Eigenschaften [Kub22j]:

- `apiVersion`: Diese Eigenschaft gibt die vom Objekt verwendete Kubernetes-API Version an, um sicherzustellen, dass das beschriebene Objekt mit der verwendeten Kubernetes Version kompatibel ist.
- `kind`: Mithilfe dieser Eigenschaft wird die Art des Objektes angegeben. Die verschiedenen Arten von Kubernetes Objekten werden in den folgenden Abschnitten erklärt.
- `metadata`: Diese Eigenschaft enthält Metainformationen wie Name der Ressource und Labels, welche genutzt werden können, um das Objekt aus anderen Objekten heraus zu identifizieren.
- `spec`: Unter dieser Eigenschaften werden die von der Objektart abhängigen Eigenschaften aufgeführt.

Kubernetes-Objekte können aus YAML-Dateien über die Kommandozeile mit `kubectl apply -f datei.yml` erstellt werden.

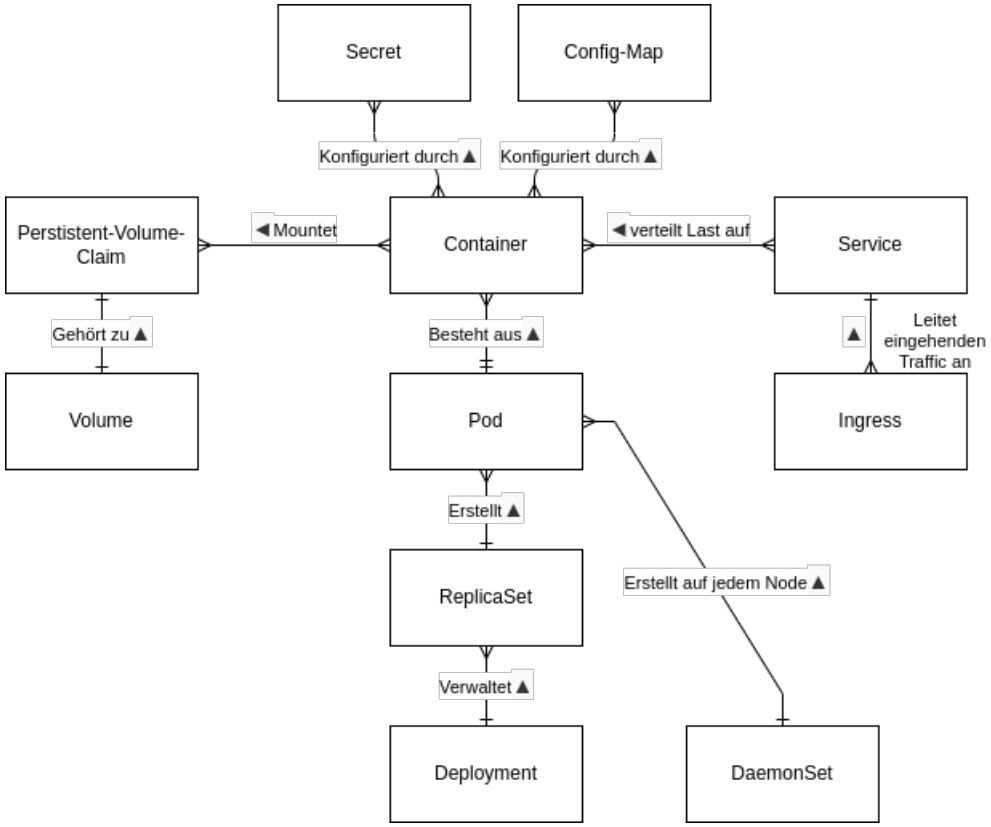

Abbildung 2.1: Diagramm mit Hierarchie von Kubernetes-Objekten

Namespace

Die meisten Kubernetes-Objekte müssen sich innerhalb eines *Namespaces* befinden. Dieser Namespace stellt eine Sammlung von Objekten dar und dient der Gruppierung und gemeinsamen Verwaltung der Objekte. So kann zum Beispiel bei der Rechtevergabe Rechte auf alle Objekten in einen *Namespace* vergeben werden. Auch dient ein *Namespace* der Isolation von Ressourcen voneinander, so kann von *Containern* nur auf *Services* innerhalb desselben *Namespaces* zugegriffen werden. [Kub22i]

Ein Beispiel für die Anwendung von Namespaces in Audimax wäre es, für jede Audimax-Installation einen eigenen Namespace, etwa mit dem Namen `musterschule-audimax`, zu erstellen, um die Trennung der Komponenten verschiedener Installationen voneinander zu gewährleisten.

Pod

Als ein *Pod* wird in Kubernetes ein Zusammenschluss von mehreren *Containern* bezeichnet, welche immer zusammen ausgeführt werden. Dies bedeutet, sie werden immer

auf demselben *Node* ausgeführt. Auch teilen sie sich Ressourcen wie unter anderem ein virtuelles Netzwerk oder Dateisystem, welche genutzt werden können, um Dateien zwischen Containern innerhalb desselben *Pods* zu teilen. Dies ist aber auch nicht notwendig und es können getrennte Ressourcen-Limits für Container innerhalb desselben Pods definiert werden. [Bur19]

In Kubernetes ist ein *Pod* die kleinste skalierbare Einheit. Das bedeutet, einzelne Container innerhalb eines Pods können nicht skaliert werden, nur alle Container im selben Pod als Einheit. Auch muss in Kubernetes jeder Container Teil eines Pods sein, es kann also kein Container existieren, welcher zu keinem Pod gehört. [Kub22l]

Als ein Beispiel für einen *Pod* kann der Backend-Pod in Audimax gesehen werden, dieser beinhaltet zwei Container. Der erste Container beinhaltet das Audimax-Backend und der andere beinhaltet ein Monitoring-Programm, welches die Funktionalität des Backendes überwacht und Statistiken aus diesem sammelt. Zum Skalieren von Pods wird meist ein *ReplicaSet* Objekt verwendet.

ReplicaSet

Ein *ReplicaSet* ist ein Kubernetes-Objekt, welches es ermöglicht, eine frei wählbare Anzahl an identischen *Pods* zu erstellen. Dies wird meist zur Skalierung oder Redundanz-Zwecken verwendet. [Bur19, Seite 103] Wird ein *ReplicaSet* geändert, etwa die Version des Images eines der Container innerhalb der durch das ReplicaSet erstellten Pods, beendet Kubernetes alle Pods mit der alten Version des Container-Images und startet gleichzeitig die Neuen. Dieses Vorgehen hat allerdings den Nachteil, dass in der Zeit zwischen Stoppen der alten und erfolgreichem Starten der neuen Container keine Anfragen bearbeitet werden können. Dies kann mithilfe eines *Deployments* verhindert werden.

Deployment

Ein *Deployment* verwaltet mehrere *ReplicaSets*. Es wird dazu genutzt, um Updates von *ReplicaSets* ohne einen Zeitpunkt, an dem der durch dieses bereitgestellte Dienst nicht zu erreichen ist, zu ermöglichen.

Das *Deployment* erreicht dies, indem, wenn eine Änderung an diesem vorgenommen, wird ein neues *ReplicaSet* erstellt wird und das alte *ReplicaSet* erst nach erfolgreichem start des neuen gelöscht wird. Dies führt dazu, dass zu jedem Zeitpunkt Container zum Beantworten von Anfragen verfügbar sind. [Bur19, Seite 113]

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: audimax
  labels:
    app: audimax
    component: backend
spec:
  replicas: 4
  selector:
      matchLabels:
        app: audimax
        component: backend
  template:
    metadata:
      labels:
        app: audimax
        component: backend
      spec:
        containers:
        - name: backend
          image: images.audimax.digital/am/backend:latest
          #...
        - name: monitoring
          image: images.audimax.digital/am/mointoring:latest
          #...
```

Listing 2.4: Ausschnitt eines Deployment-Objekts

Der oben gezeigte Deployment-Ausschnitt erstellt ein ReplicaSet, welches vier Pods mit jeweils zwei Containern erstellt.

DaemonSet

Ein *DaemonSet* verwaltet mehrere *Pods*, welche auf jedem *Node* des *Clusters* deployt werden sollen. Dabei können auch Kriterien angegeben werden, welche erfüllt werden müssen, damit der *Pod* auf einem Node deployt wird. [Bur19, Seite 131]

DaemonSets könnten zum Beispiel in Audimax dazu verwendet, um Monitoring-Software, welche Statistiken, wie die CPU-, Arbeitsspeicher- und Festplatten-Nutzung, von allen Knoten eines Clusters sammeln soll, und daher auf jedem Knoten genau einmal laufen muss, zu deployen.

Service

Ein *Service* ist ein Kubernetes-Objekt, welches die Kommunikation zwischen *Pods* ermöglicht. Für jeden *Service* können Kriterien definiert werden, welche festlegen, welche

Pods an den *Service* angeschlossen werden sollen. Andere *Pods* können nun, wenn sie im selben *Namespace* sind, Anfragen an den *Service* senden. Dabei muss der Name des *Services* als Hostnamen angeben werden. [Bur19, Seite 76-77] *Services* haben auch eine integrierte Lastverteilung. Diese wird wirksam, wenn mehrere *Pods* mit den Kriterien des Services übereinstimmen. Wenn dies der Fall ist, werden die Anfragen an den *Service* gleichmäßig auf alle *Pods* verteilt. [Kub22m]

```yaml
apiVersion: v1
kind: Service
metadata:
  name: audimax-backend
  labels:
    app: audimax
    component: backend
spec:
  selector:
    app: audimax
    component: backend
  ports:
    - name: web
      protocol: TCP
      port: 80
      targetPort: 80
```

Listing 2.5: Ein Service-Objekt

Das im Beispiel gezeigte Service-Objekt definiert in Audimax den Service für das Backend. Es verteilt Anfragen auf Port 80 an alle Pods mit dem Label `app=audimax` und `component=backend`.

Job

Job ist eine Kubernetes-Objekt-Art zur Definition von kurzlebigen Aufgaben, für welche ein *Pod* erstellt wird. In diesem wird der *Job* ausgeführt, bis dieser erfolgreich beendet wurde. Bei erfolgreicher Ausführung wird der *Job* als erledigt markiert. [Bur19, Seite 139]

Als Beispiel für einen Job kann etwa ein Upgrade-Job in Audimax gesehen werden. Dieser wird bei jedem Audimax-Update gestartet und führt Aufgaben wie etwa das Setzen von Konfigurationswerten durch.

CronJob

CronJobs erlauben es, *Jobs* wiederholt zu bestimmten Zeitpunkten zu starten. [Bur19, Seite 150]

Ein Beispiel für einen *CronJob* ist etwa das tägliche Erstellen von Backups.

ConfigMaps

ConfigMaps sind Kubernetes-Objekte, welche Konfigurations-Optionen/Dateien beinhalten. Diese können bei der Deklaration von *Containern* innerhalb von *Pods* verwendet werden, um zum Beispiel in der *ConfigMap* gespeicherte Werte in Umgebungsvariablen zu setzen oder auch als Dateien zu mounten. [Bur19, Seite 153]

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: audimax-config
data:
  welcomeMessage: Willkommen zu Audimax
  supportMail: support@audimax.digital
  #...
```

Listing 2.6: Ein Teil eines Config-Map-Objektes

Das oben gezeigte Kubernetes-Objekt enthält einen Ausschnitt aus einer Audimax-ConfigMap, welche zum Setzen von Umgebungsvariablen verwendet werden kann.

Secret

Secrets sind ähnlich zu *ConfigMaps*, nur dass diese zur Verwendung mit sensiblen Daten gedacht sind. Dabei gibt es keinen entscheidenden Unterschied zu *ConfigMaps*, außer dass diese von den meisten Werkzeugen wie zum Beispiel `kubectl` nicht automatisch ausgegeben werden. Es gibt aber auch Möglichkeiten, deren Arbeitsweise anzupassen, um zum Beispiel *Secrets* nur verschlüsselt zu speichern. [Bur19, Seite 157-158]

Rolebased-Access-Control

Kubernetes verwendet Rolebased-Access-Control (*RBAC*). Es wird verwendet, um verschiedenen Akteuren, entweder Nutzern oder auch Service-Accounts, die Berechtigungen zu geben, welche sie zum Erfüllen ihrer Aufgabe benötigen. Dies ist insbesondere wichtig, wenn Service-Accounts von innerhalb eines Containerns verwendet werden sollen, um bestimmte Aktionen durchzuführen, wie etwa das Warten auf andere Container. Dies ist eine häufig innerhalb von Containern ausgeführte Aufgabe, um sicherzustellen, dass ein anderer benötigter Container bereit ist. In diesem Fall muss sichergestellt werden, dass der Container die Berechtigung hat, um zu Warten, aber nicht die Berechtigung, um alle *Secrets* auszugeben. Dies kann mit *RBAC* einfach gelöst werden, indem eine Rolle für das Warten erstellt wird, welche nur die Berechtigungen erhält, die für das Warten benötigt werden. Diese Rolle kann dann einem Service-Account zugeordnet und dieser innerhalb eines Containers verwendet werden. [Bur19, Seite 167]

Custom-Resource-Definition

Custom-Resource-Definitions sind *Kubernetes-Objekte*, welche es erlauben, durch das Hinzufügen von neuen Objektarten, die Kubernetes-API zu erweitern. Dafür muss in der Custom-Resource-Definition die API-Version, Benennung der Ressource und ein Schema für den `spec` und `status` Teil der neu zu erstellenden Custom-Ressourcen angegeben werden. Ist dies erfolgt, kann diese als Kubernetes-Objekt erstellt werden, dabei wird das neu erstellte Objekt gegen das Schema in der Custom-Resource-Definition geprüft. Diese neuen Ressourcen können als Erweiterung des Kubernetes-Sollzustandes gesehen werden. [Kub22c]

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: ressource.example.com
spec:
  group: example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                example:
                    type: boolean
            status:
              type: object
              properties:
                ok:
                  type: boolean
  scope: Cluster
  names:
    kind: CustomRessource
    singular: customressource
    plural: customressources
    shortNames:
    - cr
```

Listing 2.7: Eine Custom-Resource-Definition

Ingress

Als *Ingress* (Latein für Betreten) wird der Datenverkehr, welcher aus dem Internet in ein Cluster kommt bezeichnet. Meist werden so eingehende HTTP-Anfragen von Client-Systemen bezeichnet. Diese können entweder direkt an einen Container geleitet oder es kann ein Proxy-Container verwendet werden, welcher Anfragedaten, wie etwa den `Host`-Header, auswertet, um diese an den richtigen Container weiterzuleiten. [Kub22h]

Controller

Controller sind Programme, die eine „Kontroll-Schleife“ darstellen, welche dafür sorgt, dass die in den Kubernetes-Objekten beschriebenen Soll-Zustände im *Cluster* realisiert werden. Der Controller ist damit dafür verantwortlich, den Istzustand des Clusters an den über ein Custom-Objekt angegebenen Sollzustand anzupassen. Dafür beobachtet der Controller Events von Kubernetes-Objekten wie zum Beispiel Erstellen, Ändern oder Löschen und führt als Reaktion darauf Kubernetes-API-Aktionen wie insbesondere das Ändern von anderen Objekten oder System-Aktionen aus, um diese Änderungen zu realisieren. [Kub22g]

Operator

Ein *Operator* ist ein Programm, welches die Verwaltung einer Anwendung in Kubernetes übernimmt. Dabei führt es Aufgaben wie etwa Installation, Update und Backup der Anwendung automatisiert aus. Meistens verwenden *Operatoren Controller*, d. h. die Eigenschaften und der Zustand einer Anwendung, die der Operator verwalten soll, wird als *Custom-Ressource-Objekt* in Kubernetes angelegt. [Kub22k]

3 Softwareauswahl eines Container-Orchestrierungs-Systems

3.1 Kriterien

Die betrachteten Container-Orchestrierungs-Systeme, welche im folgenden Kandidatensysteme genannt werden, werden nach bestimmten Kriterien bewertet. Dabei erhält jedes Kandidatensystem für jedes Kriterium eine Bewertung im Bereich von einem bis zehn Punkten. Wobei bei den ersten zwei Kriterien, welche als besonders wichtig angesehenen werden, bis zu 20 Punkte vergeben werden. Insgesamt kann ein System somit bis zu 100 Punkte erhalten.

Verteilbarkeit: Dieses Kriterium bewertet die Unterstützung des Kandidatensystems für das Verteilen der Container auf mehrere physische Server. Dies ist notwendig, sobald eine große Anzahl an Instanzen vorhanden ist, eine Instanz eine hohe Anzahl von gleichzeitigen Nutzern hat oder Ausfallsicherheit notwendig ist und so mehrere Server verwendet werden müssen. Das Einrichten und Verwalten eines Orchestrierungs-Systems auf jedem einzelnem Server stellt im Vergleich zu einem Orchestrierungs-System, welches auf einem Cluster von Servern arbeiten kann, einen sehr hohen Mehraufwand dar. Dieses Kriterium ist von besonderer Wichtigkeit, da es zum Erreichen des dritten Ziels notwendig ist und Defizite in diesem Kriterium durch Erweiterungen nur schwer verbessert werden können. Daher wird dieses Kriterium mit bis zu 20 Punkten bewertet.

Skalierbarkeit: Bewertet die Fähigkeit des Systems, identische Kopien von Containern zu erstellen und diese zu verwalten. Dies ist notwendig, wenn etwa eine hohe Nutzeranzahl die Erhöhung der Kapazität für eine bestimmte Instanz erfordern und deshalb identische Container auf verschiedenen Servern gestartet werden müssen. Dieses Kriterium ist von besonderer Wichtigkeit, da es zum Erreichen von Ziel eins und drei notwendig ist. Dieses Kriterium wird daher mit bis zu 20 Punkten bewertet.

Überwachbarkeit: Bewertet, welche Möglichkeiten für Administrierende besteht, die Funktionalität des orchestrierten Systems zu beobachten und im Falle eines bevorstehenden oder momentanen System-Ausfalls rechtzeitig informiert zu werden. Dazu zählen Möglichkeiten zum Auslesen und Archivieren von Logdateien sowie zum Sammeln von

Metriken wie etwa des Ressourcenverbrauchs und der Überprüfung von Verfügbarkeit von in Containern laufenden Diensten. Dieses Kriterium ist für Ziel zwei notwendig. Dieses Kriterium und alle Folgenden werden mit zehn Punkten bewertet.

Verbreitung: Das Kandidatensystem sollte von einer angemessenen Anzahl von Organisationen genutzt werden und eine breite Community haben. Dies dient dazu, um sicherzustellen, dass, selbst wenn es vom ursprünglich Entwickler nicht weiterentwickelt wird, eine Weiterführung der Entwicklung sichergestellt ist. Als Metrik zur Verbreitung wird die Anzahl an „Sternen“, welches ein Projekt auf Github hat, verwendet. Diese Metrik gibt einen groben Eindruck über die Verbreitung des jeweiligen Kandidatensystems, sollte aber nicht als perfekt angesehen werden, da diese nur darüber Aussage trifft, wie viele Nutzer von Github das jeweilige System interessant fanden und nicht, ob und in welchen Umfang diese es verwenden.

Erweiterbarkeit: Das Kandidatensystem sollte Schnittstellen bieten, über welche eigens entwickelte Programme eingebunden werden und über welche alle Aufgaben, die auch manuell durch Administrierende ausgeführt werden, programmatisch ausgeführt werden können. Dies ist u.a zum Ermöglichen der automatischen Einrichtung von Audimax-Instanzen notwendig. Dies ist zum Erreichen von Ziel eins und vier notwendig.

Ökosystemunterstützung: Dies bedeutet, dass für das Kandidatensystem eine Vielzahl von Drittanbieter-Erweiterungen und Integrationen existieren sollten, welche es ermöglichen, dieses in andere Anwendungen zu integrieren. Dies ist hilfreich, da so Eigenentwicklungen vermieden und Schwächen des Kandidatensystems mit Drittanbieter-Erweiterungen ausgeglichen werden können.

Effizienz: Das Effizienzkriterium bewertet den Ressourcenverbrauch des Kandidatensystems. Dies ist zum einem von Relevanz, um Verschwendung von Server-Ressourcen durch das System zu vermeiden, und zum anderen, um es potenziellen Kunden mit erhöhten Datenschutz-Bedürfnissen zu ermöglichen, eine Instanz auf ihrer eigenen Hardware zu betreiben, ohne dass aufgrund des Container-Orchestrierungs-Systems erhöhte Anforderungen an diese gestellt werden.

Sicherheit: Es bewertet die Fähigkeit des Kandidatensystems, einzelne Arbeitslasten unabhängig voneinander auszuführen, sodass der Zugriff und die Nutzung aller Betriebsmittel des Gast-Systems limitiert werden können. Konkret bedeutet dies, dass der Zugriff auf Dateisystem, Inhalte und Netzwerk-Ressourcen der Container voneinander getrennt sind und auch die Möglichkeit besteht, um die Ressourcen-Nutzung einzelner Container zu limitieren, um hohen Ressourcenverbrauch von einzelnen Containern durch Softwarefehler oder Angriffe nicht auf die restlichen Container, welche auf demselben Knoten laufen, übergreifen zu lassen.

Die Kandidatensysteme, welche im Folgenden verglichen werden, sind *Docker* mit dem Orchestrierungs-System *Docker-Compose* und der Clustering-Lösung *Docker-Swarm*, und die Kubernetes-Distributionen: *K8s*, *K3s*, *Microk8s*, *OKD* und *Rancher*.

3.2 Docker

Docker ist eine Plattform, welche es erlaubt, einen Container über die Kommandozeile zu bauen, auszuführen und zu verwalten. Docker wurde 2013 veröffentlicht. [Doc22c]

Abbildung 3.1: Das Docker Logo

Docker-Swarm-Mode ist ein in Docker integriertes Container-Orchestrierungs-System.

Verteilbarkeit

Docker operiert grundsätzlich im Single-Host-Betrieb, dies bedeutetet, Docker läuft als Hintergrundprozess, auch Daemon genannt, auf einem Host und auf demselben System können über das `docker`-Kommandozeilenwerkzeug Container gestartet und verwaltet werden. Allerdings unterstützt Docker seit Version `1.12` auch den sogenannten *Swarm-Modus*. In diesem Modus operiert Docker auf mehreren Systemen. Dabei gibt es zwei Typen von Knoten, zum einen *Manager*, welche den Swarm verwalten und zum anderen *Worker*, auf welchen die Container laufen. Neue Knoten können mithilfe des `docker swarm join` Kommandos hinzugefügt werden. Auch ist die Installation von Docker ohne Probleme möglich, da es in den Repositorien der meisten Linux-Distributionen enthalten ist, und es für alle gängigen Distribution Repositorien des Herstellers gibt. Somit lässt sich Docker auf diesen über den Paketmanager installieren. Um Arbeitslasten auf mehreren Workern innerhalb des Swarms zu verteilen, muss ein Swarm-Service über einen Manager-Knoten definiert werden. Ein Swarm-Service besteht aus einer Containerspezifikation, in welcher unter anderem das zu verwendende Container-Image, das in diesem auszuführende Kommando und die dabei zu verwendenden Umgebungsvariablen angegeben werden. [Doc22h]

Skalierbarkeit

Docker-Swarm-Services unterstützen zwei Arten von Skalierung: `global`, wobei der über den Dienst spezifizierte Container auf jedem Knoten des Clusters gestartet wird,

und `replicated`, wobei eine durch Administrierende festgelegte Anzahl an Containern über das Cluster verteilt gestartet werden. [Doc22b]

Überwachbarkeit

Docker-Container können über das `docker` Kommandozeilenwerkzeug überwacht werden. Die Überwachungsmöglichkeiten von Docker-Containern sind mit und ohne Swarm-Modus identisch. Der einzige Unterschied ist, dass sich die Kommandos, über welche diese erfolgen, ändern.

Es existiert die Möglichkeit, mit `docker ps` beziehungsweise `docker service ps` alle Container/Dienste anzuzeigen und deren Status, also ob diese momentan laufen oder nicht, abzufragen. Darüber hinaus werden weitere Informationen wie etwa der Name, der Zeitpunkt der Container-Erstellung und das ausgeführte Kommando angezeigt. [Doc22d] Auch können mit `docker stats <container id>` Informationen zur CPU-, Arbeitsspeicher- und Festplatten- und Netzwerk-IO-Auslastung angezeigt werden. [Doc22e]

Erweiterbarkeit

Docker lässt sich durch Zugriff auf die Docker-API erweitern. Auf diese kann über HTTP oder den Docker-Socket, ein meist unter `/var/run/docker.socket` zu findender Unix-Socket zugegriffen werden. Ein Unix-Socket ist ein spezieller Dateityp, welcher wie eine Netzwerkverbindung verwendet werden kann. Über die Docker-API können alle Aktionen ausgeführt werden, welche auch mit dem `docker` Kommandozeilenwerkzeug ausgeführt werden können. Allerdings findet dabei keine Rechteüberprüfung statt. Dies bedeutet, jede Anwendung, die Zugriff auf die API/den Socket hat, verfügt über vollständige Kontrolle über den Docker-Daemon und kann über diesen vollständigen Uugriff auf den Docker-Host erlangen. Dies kann zur Kompromittierung des gesamten Host-Systems führen, wenn eine Anwendung mit Docker-Socket-Zugang eine Sicherheitslücke aufweist. [Doc22f]

Verbreitung

Docker ist eine der am meisten verbreitet Container-Laufzeitumgebung. Dies wird auch durch die 62.000 Sterne auf Github deutlich. [Git22c]

Ökosystemunterstützung

Es existieren eine Vielzahl von Integrationen und Erweiterungen für Docker. Unter anderem Traefik, zum automatisierten Verwalten von eingehenden HTTP-Anfragen [Tra22b], Prometheus [Doc22a] und Netdata zum Verbessern der Überwachung.

Effizienz

Docker hat einen geringen Ressourcenverbrauch, da der Docker-Daemon leichtgewichtig ist. Daher ist Docker auch für den Einsatz auf leistungsschwächeren System geeignet [Lah20]

Sicherheit

Docker-Container erhalten standardmäßig keinen Zugriff auf das Host-Dateisystem und nur Berechtigungen für ausgehende Netzwerkverbindungen, welche unterbunden werden können. Auch gibt es standardmäßig keine Limitierung der CPU und Arbeitsspeichernutzung. Diese kann aber ebenfalls durch Konfigurationsoptionen bei der Container-Erstellung aktiviert werden. [Doc22g]

3.3 Kubernetes (K8s)

Kubernetes ist ein ursprünglich von Google entwickeltes Container-Orchestration-System, welches mittlerweile zur Cloud-Native-Computing-Foundation gehört. Kubernetes Ziel ist es, das Verteilen, Skalieren und Verwalten von Container-Anwendungen zu ermöglichen.

Abbildung 3.2: Das Kuberentes-Logo

Als Kubernetes-Distributionen werden Softwarepakete bezeichnet, welche Kubernetes enthalten. Diese dienen meist zum Vereinfachen der Installation, bieten zusätzliche Funktionen und übernehmen Entscheidungen wie etwa die verwendete CRE oder CNI. [Toz20]. Im Folgenden wird zuerst auf die „Core-Distribution“ eingegangen, und danach auf verschiede andere Distributionen.

Verteilbarkeit

Kubernetes wird primär für den Multi-Serverbetrieb entwickelt. Dabei gibt es zwei Arten von Nodes: `Control-Plane`- und `Worker`-Nodes. Auf Control-Plane-Nodes laufen die Kubernetes-Komponenten, welche die Schnittstellen zur Interaktion mit dem Cluster bereitstellen und die innerhalb des Clusters laufende Arbeitslasten verwalten. Auf Worker-Nodes werden die Arbeitslasten des Clusters ausgeführt. [Kub22o] Die Installation von Kubernetes-Knoten ist zeitaufwendig und fehleranfällig, da verschiedene Komponenten einzeln installiert werden müssen, etwa die Container-Laufzeitumgebung und das Network-Plugin. Auch müssen verschiedene Einstellungen wie etwa die von `iptables` angepasst werden. Auch das Aktualisieren auf neuere Versionen ist aufwendig,

da Komponenten einzeln und in der richtigen Reihenfolge aktualisiert werden müssen. [Kub22d]

Skalierbarkeit

In Kubernetes existieren mehrere Möglichkeiten zum Skalieren von Arbeitslasten: Das *Replica-Set*, *Daemon-Set* und *Stateful-Set*. Das *Replica-Set* erlaubt es, eine einstellbare Anzahl von identischen Pods im Cluster zu starten. Das *Daemon-Set* startet identische Container auf jedem Knoten des Clusters. Ein *Stateful-Set* operiert ähnlich wie das *Replica-Set*, nur dass jedes Replica eines Pods eine einzigartige ID hat. Dies ermöglicht es zum Beispiel, dass jedes Replica ein anderes Volume einbindet oder andere Pods sicherstellen können, dass sie immer mit demselben Replica kommunizieren. [Kub22n]

Überwachbarkeit

Kubernetes bietet über `kubectl logs` die Möglichkeit, Log-Ausgaben eines Containers oder auch dessen Ressourcenverbrauch über `kubectl describe` anzuzeigen. Weiterhin verfügt Kubernetes auch über ein Webinterface, über welches die Ressourcennutzung von Containern und auch die Auslastung von einzelnen Nodes angezeigt werden kann. [Kub22f]

Eine weitere Möglichkeit, welche Kubernetes zum Überwachen von Pods bietet, sind *Healthchecks*. Es gibt verschiedene Möglichkeiten einen Healthcheck durchzuführen. Es können Shell-Befehle innerhalb des zu testenden Pods ausgeführt werden und wenn ein solcher Befehl mit einem `0` Status-Code beendet wird, wird der Check als erfolgreich angesehen. Auch können HTTP-Anfragen gesendet und TCP-Verbindungen aufgebaut werden, welche bei einem `2xx`-Statuscode beziehungsweise dem erfolgreichen Aufbau einer TCP-Verbindung als erfolgreich angesehen werden. Die Art des *Healthchecks* entscheidet über die Konsequenzen. Es gibt zwei Arten von *Healthchecks*, *liveness* und *readiness*. Wenn ein *liveness* Check fehlschlägt, wird der Pod als fehlgeschlagen angesehen und neu gestartet. Bis der *readiness* Check erfolgreich ist oder sobald dieser fehlschlägt wird ein Pod als nicht bereit angesehen. Dies führt dazu, dass dieser vom Service-Load-Balancing ignoriert wird. [Kub22a]

Verbreitung

Kubernetes ist eines der am häufigsten eingesetzten Orchestrierungs-Systeme für produktive Einsätze. Es hat mit 87.000 Github-Sternen mehr Sterne als Docker. [Git22b]

Ökosystemunterstützung

Kubernetes verfügt über ein weites Ökosystem an Anwendungspaketen, welche einfach im Kubernetes-Cluster installiert werden können, sogenannte *Helm-Charts* [Hel22a].

Auch gibt es eine Vielzahl an Anwendungen, welche in Kubernetes integriert werden können. Es gibt etwa Exporter für *Prometheus* [PO22], welche es erlauben, Metriken über das Kubernetes Cluster zu exportieren, oder Integrationen in bekannte Load-Balancer wie *Nginx* [Ing22] und *Traefik* [Tra22c], welche es erlauben diese als Ingress-Proxies für Kubernetes zu Nutzen.

Erweiterbarkeit

Kubernetes bietet eine HTTP-API, welche zum Verwalten aller Ressourcen innerhalb eines Kubernetes-Clusters genutzt werden kann. Diese unterstützt dabei *Role-Based-Access-Control* (*RBAC*), welches es ermöglicht, einzelnen API-Verwendern, z.B. Administrierenden oder Diensten, bestimmte Berechtigungen in bestimmten Namespaces zu geben. Dies ermöglicht es, den Zugriff auf Ressourcen zu begrenzen und somit Administrierende oder Diensten innerhalb des Clusters nur so viele Rechte zu geben, wie diese zum Erfüllen ihrer Funktion benötigen. Dies reduziert im Falle einer Sicherheitslücke in einer Erweiterung den Schaden. [Kub22p]

Auch ist es möglich, die Kubernetes-API durch die Definition von *Custom-Resource-Definitions* (*CRD*) zu erweitern. Diese können dann so verwendet werden wie die Standard-Kubernetes-Ressourcen, nur dass Kubernetes im Falle von Änderungen an diesen nicht selbst aktiv wird, sondern die erweiternde Anwendung darüber informiert und diese daraufhin tätig werden muss. [Kub21b], [Kub21a]

Effizienz

Kubernetes hat einen hohen Ressourcenverbrauch, deshalb ist es nur für leistungsstärkere Server geeignet. [Kub22e]

Sicherheit

Durch Kubernetes verwaltete Container erhalten standardmäßig keinen Zugriff auf das Host-Dateisystem und können nur mit *Services* innerhalb ihres *Namespaces* und dem Internet kommunizieren. Es gibt standardmäßig keine Limitierung der CPU und Arbeitsspeichernutzung. Es können aber für jeden Container ein Limit und die benötigte Ressourcen-Nutzung definiert werden. Das Limit beschränkt die Ressourcen-Nutzung des Containers, wohingegen die benötigten Ressourcen eine Anforderung an den Node, auf welchem der Container laufen soll, darstellt. Es müssen mindestens so viele Ressourcen verfügbar sein, damit der Container gestartet werden kann. Dies dient dazu, um Abstürze oder Fehler aufgrund von unzureichenden Ressourcen zu verhindern. Kubernetes unterstützt verschiedene Container-Laufzeit-Umgebungen. Dies ermöglicht es bei erhöhten Sicherheitsanforderungen, die Container vollständig virtualisiert auszuführen, statt die Standard-Kernel-Namespace-Isolierung zu nutzen. [Kub22b]

3.4 Kubernetes (K3s)

K3s ist eine ursprünglich von Rancher entwickelte Kubernetes-Distribution, welche mittlerweile ein Cloud-Native-Computing-Foundation-Projekt ist. [K3s22]

Kubernetes-Distributionen erfüllen in der Regel die Kriterien in derselben Weise wie Kubernetes. Deshalb werden im folgenden nur die Kriterien aufgelistet, bei welchen es Unterschiede gibt.

Verteiltheit

K3s kann genauso wie *K8s* mehrere Server in einem Cluster zusammenführen. Standardmäßig wird aber ein Node, welcher sowohl *Control-Plane* als auch *Worker* ist, verwendet. Es können aber auch zusätzliche Worker hinzugefügt und im sogenannten High-Availability-Mode mehrere *Control-Planes* verwendet werden. Dies ist über einen Kommandozeilen-Installer möglich und verursacht einen viel geringeren Aufwand als die Installation von *K8s*. [Ran22f]

Verbreitung

Das K3s Projekt ist mit 20.000 Sternen auf Github nicht so weit verbreitet/bekannt wie Kubernetes an sich. Aber da die allermeisten für Kubernetes entwickelten Erweiterungen auch unter *K3s* funktionieren, profitiert es ebenfalls von der Bekanntheit von Kubernetes. [Ran22c]

Effizienz

Da *K3s* primär für „Edge-Devices“, wie etwa den Raspberry-PI, entwickelt wurde, hat es einen geringeren Ressourcenverbrauch als *k8s*. [Ran22c]

3.5 Kubernetes-Distribution OKD

Die *Origin Kubernetes Distribution*, kurz *OKD*, ist eine von *RedHat* entwickelte Kubernetes-Distribution.

Verteiltheit

OKD unterstützt, wie andere Kubernetes-Distributionen auch, das Verteilen auf verschiedene Cluster-Knoten, dabei wird ebenfalls zwischen Control-Plane und Worker-Nodes unterschieden. *OKD* kann über das Kommandozeilenwerkzeug `openshift-install` eingerichtet werden. Dieses unterstützt sowohl das Einrichten auf demselben Server,

als auch auf automatisch bei einem Infrastructure-as-a-Service-Anbieter, also Firmen, welche IT-Infrastruktur vermitten, erstellten Server. [OKD22a]

Überwachbarkeit

OKD verfügt zusätzlich zu den Kubernetes-typischen Möglichkeiten zur Überwachung noch über ein Web-Interface. Über dieses können verschiedene Cluster-Informationen, abgerufen werden. Zu diesen Informationen gehören allgemeine Cluster-Informationen, wie Anzahl der Knoten, Anzahl der auf diesen laufenden Pods und weitere Statistiken [OKD22c]

Verbreitung

OKD ist mit 1.000 Sternen auf Github von den betrachteten Systemen am unbekanntesten. Allerdings wird es von RedHat als Grundlage für deren Enterprise-Containerlösung OpenShift verwendet. Daher ist davon auszugehen, dass diese es langfristig weiterentwickeln werden. [Git22d]

Effizienz

OKD hat sehr hohe Systemvoraussetzungen. Es benötigt mindestens drei Control-Planes, welche „Fedora Core-OS“ als Betriebssystem verwenden müssen. Dies lohnt sich nur für große Cluster. [OKD22b]

3.6 Kubernetes-Distribution MicroK8s

MicroK8s ist eine von Canonical entwickelte leichtgewichtige Kubernetes-Distribution.

Verteiltheit

MicroK8s unterstützt, wie alle anderen Kubernetes-Distribution auch, das Verteilen auf Knoten. Diese Knoten sind dabei auch in Control-Planes und Worker unterteilt. *MicroK8s* unterstützt primär Ubuntu kann aber auf allen Distribution welche SnapD unterstützen installiert werden. [Mic22a]

Verbreitung

MicroK8s ist mit 6.000 Sternen auf Github nicht so bekannt wie *K3s*, aber immer noch relativ bekannt, und wird von Canonical, den Entwicklern von Ubuntu, unterstützt. Daher ist davon auszugehen, dass die Distribution längere Zeit weiterentwickelt wird. [Git22a]

Effizienz

MicroK8s hat einen geringen Ressourcen-Verbrauch und ist daher auch für den Einsatz auf leistungsschwachen Servern geeignet. [Mic22b]

3.7 Kubernetes-Distribution Rancher

Rancher ist eine vom gleichnamigen Unternehmen entwickelte Kubernetes-Distribution. [Ran22e]

Verteiltheit

Rancher verhält sich vom Gesichtspunkt der Verteiltheit ähnlich wie Standard-Kubernetes. Mit dem großen Unterschied, dass Rancher es erlaubt, über das Webinterface automatisiert neue Knoten zu erstellen. Dafür können Zugangsdaten zu einem Infrastructure-as-a-Service-Provider hinterlegt und die gewünschte Anzahl und Art an Knoten angegeben werden. Diese werden daraufhin automatisiert erstellt und eingerichtet. [Ran22a]

Überwachbarkeit

Rancher bietet zusätzliche Überwachbarkeit über das Web-Interface. [Git22e]

Verbreitung

Das Rancher-Projekt ist mit 18.000 Sternen auf Github nicht so weit verbreitet/bekannt wie Kubernetes an sich. Aber da die allermeisten für Kubernetes entwickelten Erweiterungen auch unter Rancher funktionieren, profitiert es ebenfalls von der Bekanntheit von Kubernetes. [Git22f]

Effizienz

Rancher hat einen höheren Ressourcenverbrauch als die leichtgewichtigen Kubernetes-Distributionen. [Ran22g]

3.8 Resultat

Die Ergebnisse der einzelnen Kandidatensysteme werden im Folgenden zusammengefasst und anhand dieser Punkte vergeben. Die Punkte werden danach in einer Tabelle dargestellt. Dabei wird zuerst der Name des Kriteriums genannt und in Klammern die Tabellenabkürzung angegeben.

Verteilbarkeit (Vt):

Mit zwölf Punkten sind *MicroK8s* und *K8s* auf demselben Platz. Beide Systeme lassen sich über mehrere Server verteilen, werden aber durch unterschiedliche Faktoren zurückgehalten. Bei MicroK8s ist es die Beschränkung auf Ubuntu und andere Snap-unterstützende Distributionen und bei *K8s* der schwierige Installationsvorgang. Darauf folgen mit 14 Punkten *Docker* und *OCD*. Docker ist auf den meisten Distributionen einfach zu installieren und kann dann auch einfach auf mehrere Knoten verteilt werden, bei einigen Linux-Distributionen ist Docker allerdings nicht in den Standardpaketquellen, sondern es müssen externe Paketquellen hinzugefügt werden. *OKD* hat ähnliche Nachteile wie MicroK8s, nämlich, dass es nur bestimmte Betriebssysteme unterstützt. OKD ist nur mit RedHat Enterprise-Linux und verwandten Distributionen kompatibel. Dieser Nachteil wird aber dadurch ausgeglichen, dass OKD die automatische Erstellung von Knoten über IAAS-Provider unterstützt. *Rancher* erhält 16 Punkte, da es auf den meisten Linux-Distributionen installiert werden kann und auch die automatische Erstellung von Knoten über Infrastruktur-as-a Service-Provider ermöglicht. *K3s* erhält 18 Punkte, da es viele Linux-Distributionen unterstützt und sich auf diesen mit einem Kommando installieren lässt.

Skalierbarkeit (Sk):

Bei der Skalierbarkeit gibt es nur zwei Punkte-Gruppen: einmal *Kubernetes* und seine Distributionen und Docker. Docker erzielt durch die Swarm-Services zwölf Punkte, wohingegen Kubernetes 16 Punkte erhält, da es über eine höhere Anzahl an Skalierungsmöglichkeiten verfügt.

Überwachbarkeit (Üb):

Unter dem Gesichtspunkt Überwachbarkeit erhält Docker vier Punkte, da es ohne Erweiterungen nur über Überwachungsmöglichkeiten per Kommandozeile verfügt. Kubernetes und die Distributionen K3s und MicroK8s schneiden mit sechs Punkten besser ab, da sie erweiterte Healthchecks bieten. OKD und Rancher schneiden in dieser Kategorie mit acht Punkten am besten ab, da sie ein eingebautes Webinterface bieten.

Verbreitung (Ve):

Bei der Verbreitung erhält Kubernetes mit zehn die meisten Punkte, gefolgt von Docker mit acht, dann Rancher, K3s und MicroK8s mit sechs und zuletzt OKD mit vier Punkten.

Ökosystemunterstützung (Ök):

Docker hat von den Kandidatensystemen das größte Ökosystem und erhält daher zehn Punkte. Die Kubernetes-Distributionen folgen mit acht Punkten, da es auch für diese ein großes Ökosystem gibt.

Erweiterbarkeit (Er):

Für das Kriterium Erweiterbarkeit erhalten die Kubernetes-Distributionen durch die Möglickeit, Custom-Resource-Definition zu verwenden, um benutzerdefiniert Kubernetes-Objekte zu erstellen, zehn Punkte. Docker erhält im Gegensatz nur sechs, da es nur über eine API verfügt.

Effizienz (Ef):

Für die Effizienz erhält Docker aufgrund des geringen Ressoucenverbrauchs des Docker-Daemons zehn Punkte, die leichtgewichtigen Kubernetes-Distributionen K3s, MicroK8s folgen mit acht Punkten. Die ressourcenintensive Kubernetes-Hauptdistribution und Rancher folgen mit vier Punkten und OKD mit zwei Punkten.

Sicherheit (Si):

Unter dem Sicherheits-Kriterium erhalten die Kubernetes-Distributionen acht Punkte, da diese das Isolieren von Containern in Namespaces unterstützen und durch RBAC auch das sichere Verwenden von Erweiterungen ermöglichen. Docker erhält nur vier Punkte, da es kein Isolationskonzept wie Namespaces gibt und Erweiterungen über die Docker-API vollen Zugriff auf das Host-System erhalten.

	Vt	**Sk**	**Üb**	**Ve**	**Ök**	**Er**	**Ef**	**Si**	**Summe**
Docker	14	12	4	8	10	6	10	4	68
Kubernetes-Core	12	16	6	10	8	10	4	8	74
K3s	18	16	6	6	8	10	8	8	80
Microk8s	12	16	6	6	8	10	8	8	74
OKD	14	16	8	4	8	10	2	8	70
Rancher	16	16	8	6	8	10	4	8	76

Tabelle 3.1: Ergebnisse der Container-Orchestrierungs-System Auswahl

K3s hat mit einer Punkte-Summe von 80 die meisten Punkte.

4 Auswahl einer Lasttest-Software

Eine weitere Auswahl, die getroffen werden muss, ist die einer Lasttestsoftware. Dies ist notwendig, da zum einen überprüft werden muss, ob die Skalierung von Audimax funktioniert, und zum anderen Daten gesammelt werden müssen, um die Hardware-Anforderungen an das Cluster für die geplante Nutzerlast zu ermitteln. Die vier Kriterien für diese Auswahl werden im Folgenden erläutert.

Die Optionen im Bereich des Lasttests lassen sich in drei Bereiche einteilen, zum ersten leichtgewichtige Kommandozeilenwerkzeuge zum Durchführen von Benchmarks wie *Apache Benchmark*, zum zweiten vollständige Benchmarksuites wie *JMeter* und zum Dritten die Selbstimplementierung eines Benchmarkwerkzeuges in einer Programmiersprache wie *Python*.

Die Anforderungen sind zum einen das **einfache Erstellen eines Lasttests**, es soll also möglich sein, ohne viel Aufwand die einzelnen Anfragen des Lasttestszenarios zu definieren. Zum anderen sollte auch das **Durchführen eines Lasttests** ohne viel Arbeitsaufwand und auf eine reproduzierbare Weise möglich sein. Dabei ist besonders darauf zu achten, dass es auch notwendig sein wird, dass derselbe Lasttest mehrmals hintereinander ausgeführt wird, um Ergebnisse zu vergleichen. Auch sollten **Antworten aus vorherigen Anfragen als Parameter** in darauffolgenden Anfragen verwendet werden können. Nur so können realistische Lasttestszenarien, welche oft Folgeanfragen beinhalten, erstellt werden. Die Ergebnisse des Lasttests sollen auch **visuell dargestellt** werden, zum Beispiel als Anfrage-Zeit-Graph.

4.1 Verschiedene mögliche Softwareprodukte

Apache-Benchmark

Apache-Benchmark ist ein Kommandozeilenwerkzeug, welches es ermöglicht, einfache Benchmarks durchzuführen. Dabei können unter anderem die Anzahl der auszuführenden Anfragen, das Limit für gleichzeitige Anfragen und die URL, an welche die Benchmark-Anfragen gesendet werden sollten, über Kommando-Zeilen-Argumente eingestellt werden.

Nach erfolgreichen Ausführen des Benchmarks wird das Ergebnis auf der Kommandozeile ausgegeben. Das Erhalten von einzelnen Anfrage-Ergebnissen zur späteren Verwendung als Argumente für weiterführende Anfragen ist nicht möglich. Und auch das grafische Darstellen von Ergebnissen ist schwierig, da als Ausgabe nur Text und HTML unterstützt werden. [Apa22a]

Python-Skript

Mithilfe der Programmiersprache Python kann ein Benchmark-Skript geschrieben werden. Dabei können Bibliotheken verwendet werden, um die Implementierung zu erleichtern. Ein Beispiel für eine solche Bibliothek ist `request`, welche das Senden von HTTP-Anfragen erleichtert. Dieses Vorgehen hat den Vorteil, dass der volle Funktionsumfang von Python und dessen verfügbarer Bibliotheken genutzt werden kann, um die Anfrage-Ergebnisse auszuwerten und um diese in nachfolgenden Anfragen zu verwenden. Auch können die Anfragezeiten direkt mit Visualisierungs-Bibliotheken wie `Seaborn` grafisch dargestellt werden. Der Nachteil dieser Möglichkeit ist, dass sie mit einem höheren Implementierungsaufwand einhergeht.

JMeter

JMeter ist ein von der Apache Software Foundation entwickeltes Lasttestwerkzeug. Es verfügt über eine grafische Oberfläche, über welche einfach komplexe Lasttests aus verschiedenen Komponenten wie etwa Anfrage, Antwortparsern, Visualisierungen und Kontrollstrukturen erstellt werden können. Auch kann JMeter als HTTP-Proxy fungieren und, wenn dieser in einem Browser als Proxy konfiguriert wird, alle über diesen laufenden Anfragen automatisch als Komponente eines Lasttests erstellen. Diese müssen aber noch zwecks Parametrisierung nachbearbeitet werden. JMeter unterstützt das Verwenden von Werten aus vorherigen Antworten als Parameter in neuen Anfragen, das grafische Darstellen von Antwortzeiten und das Exportieren von Ergebnissen. [Apa22b]

4.2 Resultat

Für Audimax wird eine Kombination aus JMeter und Python verwendet für das Durchführen der Lasttests verwendet, da es mit JMeter einfach möglich ist, Lasttestszenarien zu erstellen und diese wiederholt auszuführen. Insbesondere das automatische Erstellen von Lasttestanfragen mit JMeter vereinfacht die Arbeit, in dem es als HTTP-Proxy fungiert. Für die Visualisierung wird Python verwendet, da die grafische Darstellung der Antwortzeit in JMeter nicht anpassbar genug ist. Die Ergebnisse des Lasttests werden in Kapitel 5 gezeigt.

	Erstel.	Ausführ.	Parameter	Visual.	Summe
Apache-Bench.	5	15	4	10	34
Python-Script	12	15	20	18	60
JMeter	18	19	18	12	67
JMeter + Python	18	19	18	18	73

Tabelle 4.1: Ergebnisse der Lasttestsoftwareauswahl

Lasttests in JMeter sind baumartig aufgebaut. Die Wurzel wird `Test Plan` genannt und repräsentiert den Lasttest. Unter diesen können weitere Knoten eingefügt werden. In erster Ebene sind dies meist `Thread Groups` die Nutzer repräsentieren. In diesen kann eingestellt werden, wie viele Nutzer simuliert werden und wie oft diese die Kindknoten der `Thread Group` ausführen sollen. Unterhalb der `Thread Group` können entweder Logiknoten wie Bedingungen und Schleifen eingefügt werden oder `HTTP-Anfragen`. `HTTP-Anfragen` können vollständig konfiguriert werden. Dabei können unter anderem HTTP-Methode, Pfad, Hostname und Anfrageinhalt festgelegt werden. Diese Werte können auch aus Variablen gesetzt werden. Unterhalb von `HTTP-Anfragen` können sogenannte `Extractoren` hinzugefügt werden, welche konfiguriert werden können, um Informationen aus einer HTTP-Antwort zu extrahieren und diese zur späteren Verwendung in Variablen zu schreiben.

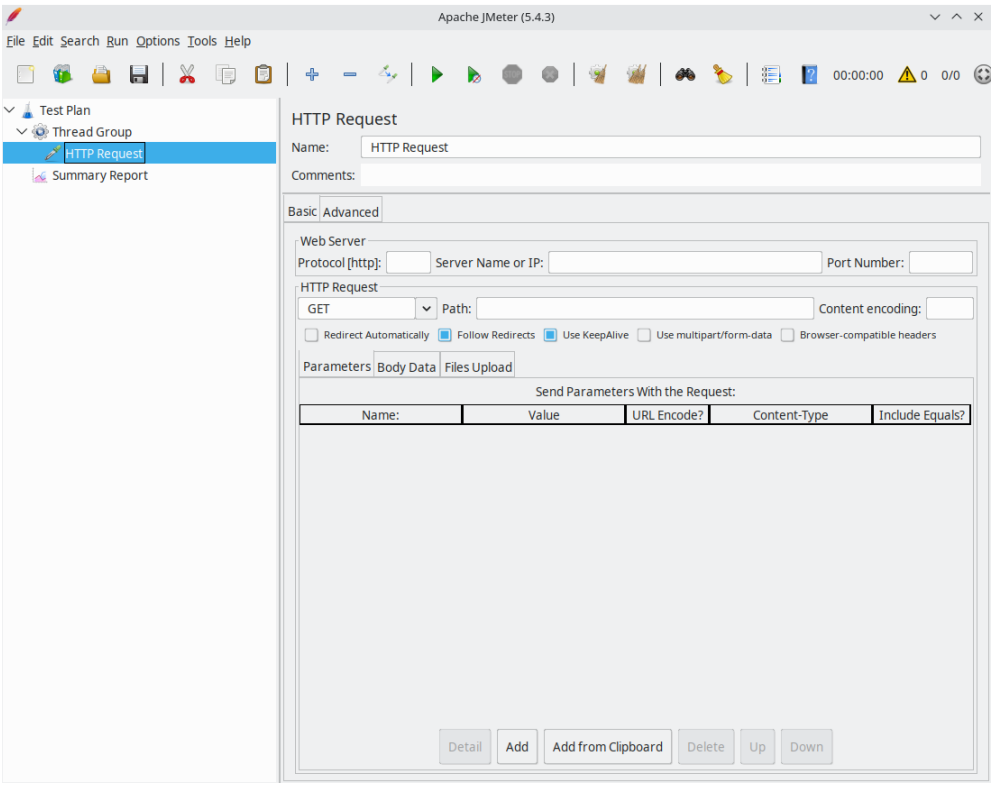

Abbildung 4.1: Ein HTTP-Anfrage Knoten in JMeter

5 Realisierung

Die ausgewählte Kubernetes-Distribution *K3s* ermöglicht es, Audimax verteilt und skalierbar auszuführen. Damit ist ein Teil-Ziel erreicht. Um alle Ziele zu erreichen, müssen aber noch Erweiterungen vorgenommen werden. Dazu wird, um die anpassbare Installation von Audimax zu vereinfachen, die Erstellung eines sogenannten *Helm-Charts* beschrieben. Dann wird die Umsetzung eines *Operators* für Audimax beschrieben, um die Installation weiter zu automatisieren und die Konfiguration zu vereinfachen. Zuletzt wird beispielhaft die Einrichtung eines *K3s*-Clusters zum Ausführen von Audimax beschrieben. In Laufe dessen wird auch auf die Umsetzung eines benachrichtigenden Monitoring-Systems mit *Prometheus* eingegangen.

5.1 Helm-Chart

Ein Helm-Chart kann als eine Art Anwendungs-Paket (wie zum Beispiel `.deb` oder `.rpm`) gesehen werden, welches eine Anwendung mit allen Komponenten, die für ihren Betrieb benötigt werden, in ein Kubernetes-Cluster deployt. Bei diesem handelt es sich um eine Archiv-Datei, welche Template-Dateien für Kubernetes-Ressourcen und Standardwerte für diese Templates enthält. Es kann dann über das `helm` Kommandozeilenwerkzeug in einem Kubernetes-Cluster installiert werden, wobei die im Helm-Chart enthaltenen Standardwerte durch deployment-spezifische Werte aus Kommandozeilenargumenten oder YAML-Dateien ersetzt werden können. Helm ersetzt die Platzhalter innerhalb des Templates und erstellt so eine YAML Datei und wendet diese auf das Kubernetes-Cluster an. [Luk18, Seite 576]

Es gibt eine Vielzahl von Helm-Charts für bekannte Anwendungen wie etwa *Wordpress*, *Postgres* und *Nginx*. Diese werden entweder vom Entwickler der jeweiligen Anwendung oder Mitgliedern der Kubernetes-Community zur Verfügung gestellt. Diese vorgefertigten Helm-Charts lassen sich auch in eigenen Helm-Charts als Abhängigkeiten verwenden, falls diese als Komponente innerhalb einer Anwendung benötigt werden, welche mithilfe eines Helm-Charts deployt werden soll. So müssen die Kubernetes-Ressourcen dieser Komponente nicht selbst beschrieben werden, sondern es kann das jeweilige Helm-Chart als Abhängigkeit hinzugefügt werden. [Hel22b]

Struktur eines Helm-Charts

Um ein Helm-Chart zu erstellen ist es wichtig zu wissen wie ein solches aufgebaut ist. Dabei liegen im Basisordner:

Chart.yaml

Die `Chart.yaml` enthält grundlegende Meta-Informationen über das Helm-Chart wie Name, Version, Version der installierten Anwendung und Abhängigkeiten.

```yaml
apiVersion: v2
name: Audimax
version: 2.4.0
appVersion: 1.10.0
dependencies:
  - name: postgresql
    version: 11.1.15
    repository: https://charts.bitnami.com/bitnami
    condition: audimax.enabled
    ...
```

Listing 5.1: Ein Auschnitt der `Chart.yaml` von Audimax

Values.yaml

Die `Values.yaml` enthält die Standardwerte für das Helm-Chart. Diese können innerhalb von Helm-Templates mit `.Values` abgerufen und beim Deployment des Helm-Charts überschrieben werden. In Audimax enthält die `Values.yaml`-Datei verschiedene Abschnitte, welche den jeweiligen Audimax-Komponenten zugeordnet sind.

```yaml
#...
audimax:
  enabled: true
  noUpgrade: false
  randomUserCreationEnabled: false
  perCourseUserManagementDisabled: true
  #...
  loginScreen:
    oic:
      buttonText: "Login mit externen Account"
      buttonIcon: "oic"
      hideLocalLogin: true
      hiddenText: "Login mit internen Account"
#...
matrix:
  enabled: false
  replicas: 1
  secret: "matrix"
  #...
#...
```

Listing 5.2: Ein Auschnitt der `Values.yaml` von Audimax

templates-Ordner

Der `templates`-Ordner enthält die Helm-Template-Dateien. Bei diesen handelt es sich um Kubernetes-Ressourcen-Beschreibungen im YAML-Format, innerhalb welcher Template-Ausdrücke und -Bedingungen verwendet werden können. Innerhalb der Template-Ausdrücke können verschiedene Variablen verwendet werden. Die am häufigsten verwendeten davon sind `Release` und `Values`. `Release` beinhaltet ein Objekt, welches Informationen zur momentanen Installation beinhaltet, wie etwa der `Name` der Installation und der `Namespace`, in welchem diese ausgeführt wird. `Values` ist ein Objekt, welches die Werte beinhaltet, also entweder die Standardwerte aus der `Values.yaml` oder die mit `-f` bzw. `set` gesetzten installationsspezifischen Werte. Der `template`-Ordner des Audimax-Helm-Charts beinhaltet für jede Komponente eine Datei, welche die Templates, die zum Deployment dieser Kompontente nötig sind, beinhaltet. Es gibt allerdings auch Dateien, welche nur eine bestimmte Art von Kubernetes-Objekten beinhaltet, wie etwa `rbac.yaml`, welche die zum Deployment von Audimax notwendigen RBAC-Kubernetes-Objekte enthält.

```
templates
├── audilive.yaml
├── autodeploy.yaml
├── backend.yaml
├── config.yaml
├── document-processor.yaml
├── frontend.yaml
├── matrix.yaml
├── oic-middleware.yaml
├── rbac.yaml
├── traefik.yaml
└── update-job.yaml
```

Listing 5.3: Ein Ausschnitt der `templates`-Ordner-Struktur aus Audimax

```
{{ if .Values.audimax.enabled }}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-backend
  labels:
    app: {{ .Release.Name }}-backend
spec:
  replicas: {{ .Values.backend.replicas }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}-backend
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}-backend
    spec:
      serviceAccountName: {{ .Release.Name }}-backend
      containers:
      - name: {{ .Release.Name }}-backend
        image: {{ .Values.image.registry }}#...
        #...
---
apiVersion: batch/v1
kind: CronJob
metadata:
  name: {{ .Release.Name }}-backend-cron
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      #...
{{ end }}
```

Listing 5.4: Ein Auschnitt aus der `backend.yaml` Datei

Verwendung des Helm-Charts

Helm-Charts können mit dem `helm package .` Befehl in eine `tar`-Datei gepackt werden. Diese kann dann über `helm install <Installations Name> <Package Datei>` in ein Kubernetes-Cluster installiert werden. Dabei können die Standardwerte, welche in der `Values.yaml` definiert sind, überschrieben werden. Dafür wird ein Wert mittels Kommandozeilen-Argument `-set` übergeben oder eine YAML-Datei mit `-f` angegeben. Hierbei werden diese Werte mit den Standardwerten zusammengeführt.

```
chart-demo on  main [?] is  v2.4.0 via ⎈ v3.7.1
❯ helm package .
Successfully packaged chart and saved it to: /home/jkuche/Projekte/Arbeit/audimax/chart-demo/Audim
ax-2.4.0.tgz

chart-demo on  main [?] is  v2.4.0 via ⎈ v3.7.1
❯ helm install --create-namespace -n demo-audimax -f ../chart/vals.yaml audimax Audimax-2.4.0.tgz
NAME: audimax
LAST DEPLOYED: Sat May  7 15:37:53 2022
NAMESPACE: demo-audimax
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Abbildung 5.1: Packen des Audimax-Helm-Charts und anschließender Installation unter dem Namen `audimax` in den neu erstellten Namespace `demo-audimax` mit dem Setzen von Values aus der Datei `../chart/vals.yaml`

5.2 Operator

Nun, da es mithilfe des Helm-Charts möglich ist Audimax konfigurierbar zu installieren, muss dies nur noch automatisiert und die Konfiguration erweiterbar gestaltet werden. Dafür wird mithilfe des *Operator-Frameworks* ein Operator für Audimax entwickelt.

Operator-Framework

Das *Operator-Framework* ist ein Framework zum einfachen Erstellen von Operatoren, welche Anwendungen innerhalb eines Kubernetes-Cluster verwalten können. [ope22]

Mithilfe des Operator-Frameworks können *Custom-Resource-Definitions* erstellt werden. Dafür werden für jede *CRD* zwei Datenstrukturen definiert: `Spec` und `Status`. Die `Spec` Datenstruktur beschreibt die Felder, welche angegeben werden müssen, um ein Kubernetes-Objekt für die Custom-Resource zu erstellen. `Status` beschreibt den internen Status der Ressource. Diese Datenstrukturen werden durch das Operator-Framework in *Custom-Resource-Definitions* umgewandelt. Wenn Ressourcen von Typ dieser *CRDs* erstellt, geändert oder gelöscht werden, wird eine zu diesen zugeordneten sogenannter *Rekonzilations-Funktion* aufgerufen.

Diese Rekonziliation-Funktion ist die *Controller-Schleife* und hat die Aufgabe, den aus der Spezifikation gewünschten Zustand im Cluster anzuwenden. Ist diese darin erfolgreich, wird sie bis zur nächsten Änderung der Ressource nicht erneut aufgerufen. Sollte die Funktion nicht in der Lage sein, den Zustand wie gewünscht zu ändern oder muss diese auf Aktionen wie etwa die Rekonzilations-Funktion von anderen Ressourcen warten, kann diese einen `Requeue`-Rückgabewert zurückgeben. Dieser führt dazu, dass der Controller die Funktion zu einem späteren Zeitpunkt erneut aufruft. Die Rekonzilation-Funktion kann auch Informationen im Status der von ihr verwalteten

Ressourcen speichern. Dies kann entweder erfolgen, um den Zustand zwischen Aufrufen zu persistieren oder um Informationen an Cluster-Operatoren weiterzugeben.

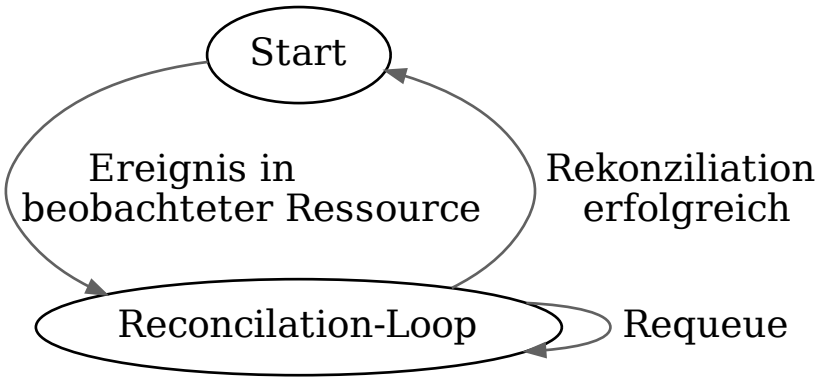

Abbildung 5.2: Kubernetes-Rekonziliations-Schleife

CRD-Struktur

Für den Audimax-Operator werden zwei *Custom-Resource-Definitions* definiert: zum einen `AudimaxConfig` und zum anderen `AudimaxDeployment`. Objekte vom Typ `AudimaxConfig` enthalten Audimax-Konfigurations-Werte und einen optionalen Verweis auf ein anderes `AudimaxConfig`-Objekt von welchem nicht gesetzte Konfigurations-Optionen übernommen werden. `AudimaxDeployment`-Objekte repräsentieren eine Audimax-Installation. Diese Objekte enthalten einen Verweis auf das zu verwendende `AudimaxConfig`-Objekt und bieten aber noch die Möglichkeit in diesem gesetzte Optionen zu überschreiben.

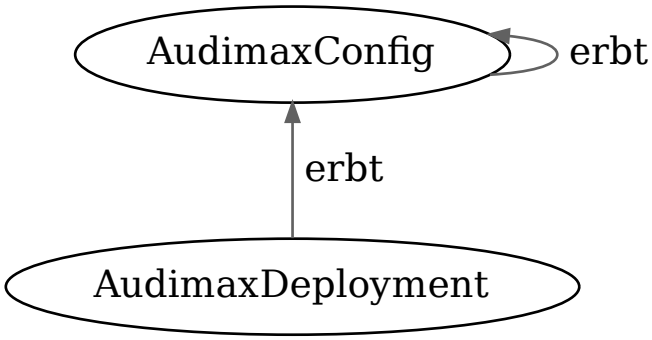

Abbildung 5.3: Verhältnis von Audimax CRDs

Die CRD-Datenstrukturen werden für das `Operator-Framework` als structs in der Programmiersprach GO definiert.

```
type ConfigSpec struct {
      Inherits     string `json:"inherits,omitempty"`
      ChartUrl     string `json:"chartUrl"`
      Values       string `json:"values"`
}
```

Listing 5.5: Das `ConfigSpec` Struct

Das `ConfigSpec` beschreibt die bei der Erstellung eines `AudimaxConfig`-Kubernetes-Objektes benötigen Werte. Das Feld `Inherits` enthält den Namen des zu erbenden `AudimaxConfig`-Objekts. `json:"inherits,omitempty"` bedeutet das Struct-Feld `Inherits` wird bei der Serializierung und Deserialisierung von und nach JSON oder YAML `inherits` genannt und wird bei der Serialisierung, falls es einen leeren String enthält, weggelassen. Das Feld `ChartUrl` enthält die URL von welcher das Helm-Chart, welches für das Erstellen einer Audimax-Instanz verwendet werden soll, heruntergeladen wird. Das Feld `Values` beinhaltet die Werte, welche für das Installieren dieses Helm-Charts verwendet werden sollen.

```
type DeploymentSpec struct {
        Config   string      `json:"config"`
        Override *ConfigSpec `json:"override,omitempty"`
}
```

Listing 5.6: Das `DeploymentSpec` Struct

Das `DeploymentSpec` beschreibt die bei der Erstellung eines `AudimaxDeployment`-Kubernetes-Objekt benötigen Werte. `Config` enthält der Name des `AudimaxConfig` Objekts ist, auf Basis dessen Audimax installiert werden soll. In `Override` können dieselben Werte gesetzt werden wie in einer `ConfigSpec`, sind diese gesetzt, überschreiben sie die Werte, die in `Config` angegebenen `AudimaxConfig`.

```
type DeploymentStatus struct {
        Installed  bool   `json:"installed"`
        ConfigHash string `json:"configHash"`
        Failures   uint8  `json:"failures,omitempty"`
}
```

Listing 5.7: Das `DeploymentStatus` Struct

Der `DeploymentStatus` beschreibt den Statusteil der `AudimaxDeployment`-CRD. `Installed` gibt an, ob das Deployment bereits mit Helm installiert wurde. Diese Information wird von der Rekonzilations-Funktion genutzt, um festzustellen, ob eine Installation notwendig ist. `ValuesHash` beinhaltet den Hashwert der effektiven Konfiguration, welche bei der letzten Installation des `AudimaxDeployments` genutzt wurde. Dies wird verwendet, um festzustellen, ob sich die Konfiguration geändert hat und somit ein Helm-Update notwendig ist.

Objekt-Beispiel

Mithilfe dieser CRD-Struktur können baumartige Konfigurations-Strukturen abgebildet werden, die ausgehend von der Wurzel immer spezifischer werden. Etwa kann eine Basiskonfiguration, welche Standardwerte für alle Instanzen beinhaltet, erstellt werden. Von dieser können dann Konfigurationen für Kundengruppen, welche identische Funktionen benötigen, erstellt werden, und auf Basis dieser, Instanzen, welche nur spezifische Werte wie Hostname und Passwörter setzen.

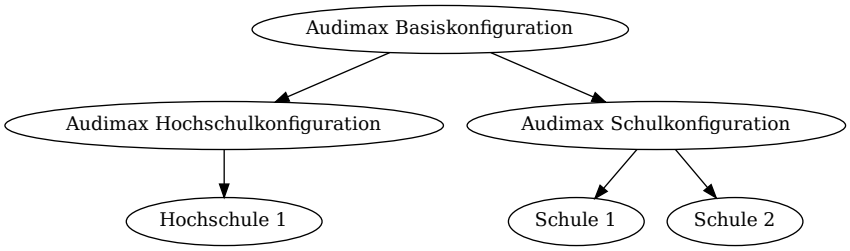

Abbildung 5.4: Verhältnis von Audimax-Objekten

Rekonzilations-Logik

Der Kern des Audimax-Operators bildet die Logik, welche innerhalb der Rekonziliation-Funktion ausgeführt wird.

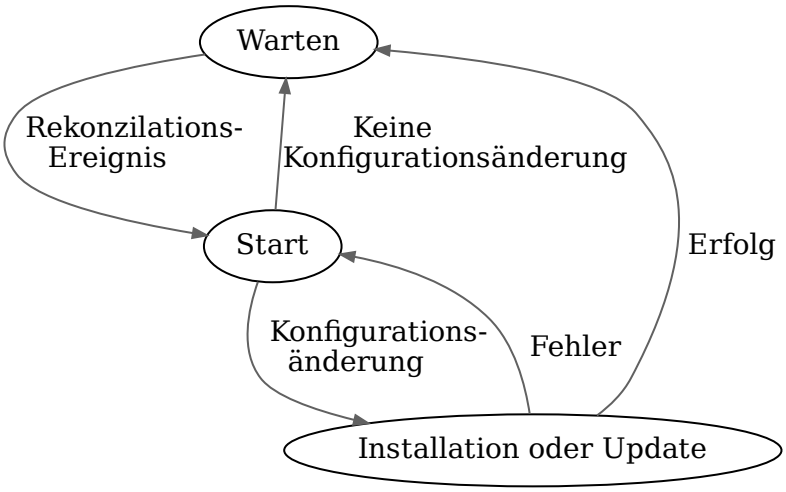

Abbildung 5.5: Die Rekonziliations-Logik

Die Rekonziliations-Funktion wird ausgeführt, sobald sich der Sollzustand ändert, also ein `AudimaxDeployment`-Objekt erstellt oder geändert wird. Zuerst wird sichergestellt, dass die Änderung nicht nur an der Syntax des `AudimaxDeployments` ist, sondern auch eine semantische Änderung an der Konfiguration darstellt. Dazu werden die effektiven Werte aus den geerbeten Konfigurationen und Overrides des Deployments erstellt und gehasht. Dieser Hash wird mit dem im `Status` gespeicherten Hash, des bei der letzten Installation/Upgrade gespeicherten, verglichen. Sind diese gleich gab es keine effektive Änderung an der Konfiguration und somit an dem Sollzustand. Die Rekonzilations-Funktion kann dann erfolgreich beendet werden. Wird eine Änderung festgestellt, wird Helm mit den erzeugten effektiven Konfigurationswerten aufgerufen. Dabei wird, jenachdem ob es sich um die erste Ausführung für dieses Deployment handelt, `helm intall` oder `helm upgrade` verwendet. Wenn dies erfolgreich ist, war die Rekonziliation erfolgreich. Sollte bei der Helm-Operation ein Fehler auftreten, wird die Rekonziliation-Funktion erneut aufgerufen. Sollte sich der Fehler mehr als dreimal wiederholen, wird davon ausgegangen, dass dieser Fehler nur durch manuelles Einschreiten von Administrierenden behoben werden kann und es wird zurück in den Wartezustand übergegangen.

5.3 Cluster-Setup

Sobald alle Vorbereitungen abgeschlossen sind, kann nun das Cluster installiert werden. Dazu wird sowohl das im vorherigen Kapitel ausgewählte Container-Orchestrierungs-System *K3s* installiert als auch zusätzliches Monitoring auf Basis von Prometheus, Grafana und Alert-Manager. Zuletzt wird der Audimax-Operator eingerichtet.

K3s Installation

Die *K3s*-Installation erfolgt über ein Bash-Skript, welches mit `curl` geladen wird. [Ran22b]

```
curl -sfL https://get.k3s.io | sh -
```

Listing 5.8: Der Bash-Befehl, welcher zur *K3s*-Installation genutzt wird

Nachdem dieser Befehl erfolgreich ausgeführt wurde, läuft auf dem momentanen Server ein Single-Node-Cluster, welches sowohl Control-Plane als auch Worker ist. Dies kann mit dem `kubectl get nodes` Kommando bestätigt werden.

```
root@k3s-installation:~# kubectl get nodes
NAME              STATUS   ROLES                  AGE   VERSION
k3s-installation   Ready    control-plane,master   17s   v1.23.6+k3s1
```

Abbildung 5.6: Output von kubectl get nodes

Um einen Worker zu dem Cluster hinzuzufügen, wird vor der Ausführung des Installers auf anderen Servern das auf dem zuerst installiertem Server in der Datei `/var/lib/rancher/k3s/server/node-token` befindliche Token in die `K3S_TOKEN` Umgebungsvariable geschrieben. [Ran22d]

Ingress

Damit eingehende HTTP(s)-Anfragen angenommen werden und an den richtigen Service geleitet werden können, muss ein Ingress-Proxy installiert werden. Auch dies kann mithilfe von Helm erfolgen. [Tra22a]

```
helm install -f traefik.yml traefik traefik/traefik
```

Listing 5.9: Installation von Traefik mit Helm

Dies installiert den *Traefik*-Ingressproxy als Daemon-Set im Cluster, wodurch *Traefik* auf jedem Knoten des Clusters gestartet wird und somit Anfragen von jedem Knoten des Clusters entgegengenommen werden können.

Monitoring

Zwecks Monitoring wird Prometheus und Grafana verwendet. Prometheus ist eine Zeitserien-Datenbank, welche im Folgenden zum Speichern der Monitoringwerte verwendet wird. [Pro22] Grafana wird zur Visualisierung der in Prometheus gespeicherten Werte verwendet. [Gra22]

Auch kann der Prometheus-Alertmanager dazu verwendet werden, im Falle von Fehlern die Administrierenden zu benachrichtigen. In der unten zu sehenden Konfiguration wird im Fehlerfall eine Benachrichtigung über das Chatsystem Slack versendet.

```
#...
grafana:
  adminPassword: admin
#...

alertmanager:
  config:
    global:
      resolve_timeout: 1m
      slack_api_url: 'https://hooks.slack.com/services/...'
    route:
      receiver: 'slack-notifications'
      #..
    receivers:
    - name: 'slack-notifications'
      slack_configs:
      - channel: '#alerts'
        username: 'Alertmanager (audimax.digital)'
        send_resolved: true
    - name: 'null'
```

Listing 5.10: prometheus.yaml

Prometheus wird über helm installiert:

```
helm install -f prometheus.yaml prometheus-stack \
prometheus-community/kube-prometheus-stack
```

Listing 5.11: Helm Install

Ist die Installation abgeschlossen, sammelt Prometheus Daten von Kubernetes und sendet, im Falle eines Problems, eine Warnung per Slack.

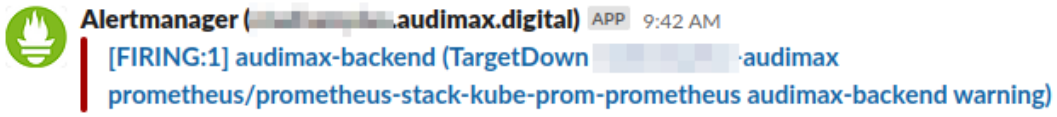

Abbildung 5.7: Ein Alert in Slack, welcher dadurch verursacht wurde, dass kein `audimax-backend`-Pod verfügbar war

Installation des Operators

Der Operator kann dank des Operator-Frameworks einfach installiert werden. Der Quellcode des Operators muss aus git geclont werden und kann mit folgendem Kommando ausgeführt werden, dabei werden alle für den Betrieb des Operators benötigten *RBAC-Regeln*, *CRDs* und *Deployments* erstellt:

```
git clone <operator repository> operator
cd operator
make deploy
```

Listing 5.12: Make zum Deployment

Erstellen von Audimax-Instanzen

Sollen nun Audimax-Instanzen mit dem Operator erstellt werden, muss dafür ein `AudimaxConfig`- und ein `AudimaxDeployment`-Objekte erstellt werden. Diese können mithilfe von `kubectl` aus einer YAML-Datei gelesen werden.

```yaml
apiVersion: audimax.audimax.digital/v1alpha1
kind: AudimaxConfig
metadata:
  name: audimax-config
spec:
  chartUrl: "https://charts.audimax.digital/Audimax-2.4.0.tgz"
  values: |
    tls:
      enabled: true
    bbb:
      enabled: true
      referer: 'https://bbb.audimax.digital/'
      url: "https://bbb.audimax.digital/bigbluebutton/"
      secret: "example"
---
apiVersion: audimax.audimax.digital/v1alpha1
kind: AudimaxDeployment
metadata:
  name: audimax-deployment
spec:
  config: test-audimax/audimax-config
  override:
    values: |
      hostname: example.audimax.digital
```

Listing 5.13: minimal Beispiel mit AudimaxConfig und AudimaxDeploy

Sobald das `AudimaxDeployment`-Kubernetes-Objekt erstellt wurde, wird die Rekonzilations-Funktion aufgerufen und eine entsprechende Audimax-Instanz erstellt. Auf diese Audimax-Instanz kann dann über den Hostname zugegriffen werden.

```
Arbeit/audimax/audimax-operator via  v1.17.5 via ❄  impure (nix-shell)
❯ kubectl create ns example-audimax
namespace/example-audimax created

Arbeit/audimax/audimax-operator via  v1.17.5 via ❄  impure (nix-shell)
❯ kubectl apply -n example-audimax -f audimax.yml
audimaxconfig.audimax.audimax.digital/audimax-config created
audimaxdeployment.audimax.audimax.digital/audimax-deployment created

Arbeit/audimax/audimax-operator via  v1.17.5 via ❄  impure (nix-shell)
❯ kubectl -n example-audimax get pods
NAME                                                READY   STATUS      RESTARTS   AGE
audimax-deployment-api-docs-767c95f8c8-2j2wz        1/1     Running     0          5m57s
audimax-deployment-audilive-76b5d96d7c-qwcm9        1/1     Running     0          5m56s
audimax-deployment-backend-5cffdb9d5d-fjjw4         1/1     Running     0          5m56s
audimax-deployment-backend-cron-27532285--1-wtwrs   0/1     Completed   0          2m18s
audimax-deployment-backend-update-1--1-75w5n        0/1     Completed   0          5m56s
audimax-deployment-document-processor-7dc7c94c-zmz7c   1/1     Running     0          5m56s
audimax-deployment-frontend-c969849f9-dfknd         1/1     Running     0          5m56s
audimax-deployment-help-docs-557788b559-2k8qr       1/1     Running     0          5m56s
audimax-deployment-postgresql-0                     1/1     Running     0          5m56s
audimax-deployment-redis-master-0                   1/1     Running     0          5m56s
audimax-deployment-traefik-response-helper-7cf977668c-hdc2b   1/1     Running     0          5m56s
```

Abbildung 5.8: Konsole mit Audimax-Deployment

5.4 Lasttest

Das Erstellen von Lasttests erfolgt mit JMeter. JMeter erlaubt das automatische Erstellen von Anfragen für ein Lasttest-Szenario, indem es als ein HTTP-Proxy fungiert. Um diese Funktion zu nutzen, muss zuerst ein `Recording Controller` erstellt werden. In diesem erstellt JMeter die aufgezeichneten Anfragen. Dann muss ein `HTTP(S) Test Script Recorder` erstellt werden. Dieser verwaltet die eigentliche Proxy-Funktionalität. Wenn dieser gestartet wird, steht unter einem konfigurierbaren Port ein HTTP-Proxy zur Verfügung. Dieser kann dann über die Browsereinstellung als Proxy eingestellt werden.

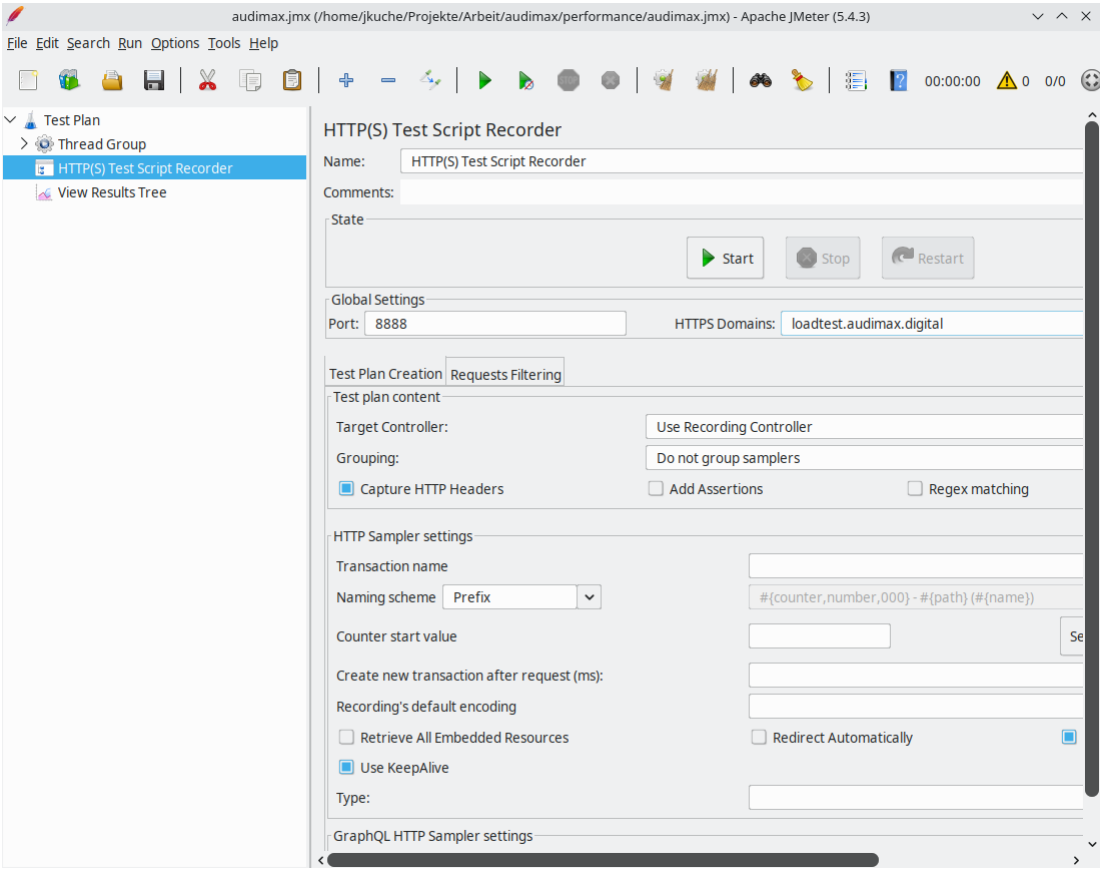

Abbildung 5.9: Ein HTTP(S) Test Script Recorder Knoten in JMeter

Sendet nun der Browser eine Anfrage über diesen Proxy, wird diese in JMeter unterhalb des `Recording-Controllers` angelegt. Diese muss dann nur noch parametrisiert und an die richtige Stelle innerhalb der JMeter-Struktur verschoben werden.

Das Lasttest-Szenario für Audimax wird mithilfe des Proxys erstellt. Dabei wird über diesen die Verwendung von Audimax simuliert. Dies bedeutet, es wird ein Log-In ausgeführt und eine Gruppe ausgewählt. In dieser Gruppe wird dann eine Konferenz ausgewählt und dieser beigetreten. Die von JMeter aufgezeichneten Anfragen werden danach parametrisiert. Dies bedeutet, dass aus den Antworten extrahierte Werte, wie die ID der Gruppe aus der Gruppenliste-Antwort, in Anfragen eingebaut wird, damit das Lasttest-Szenario mit anderen Kurs-IDs funktioniert. Als Letztes wird die Lasttest-ausführung konfiguriert. Dabei wird eingestellt, wie oft und über welchen Zeitraum das Lasttest-Szenario ausgeführt werden soll. Der Zeitraum wird *Ramp-Up-Time* genannt und ist für einen realistischen Lasttest wichtig, da Nutzer nicht alle zur selben Sekunde anfangen ein System zu nutzen, sondern meist innerhalb einer bestimmten Zeit-Periode. An einer Hochschule etwa in den Minuten vor Anfang einer Vorlesung.

Ergebnisse

Die Ergebnisse der von JMeter ausgeführten Lasttests bei der Simulation von 600 Nutzern wurden ins `CSV`-Format exportiert.

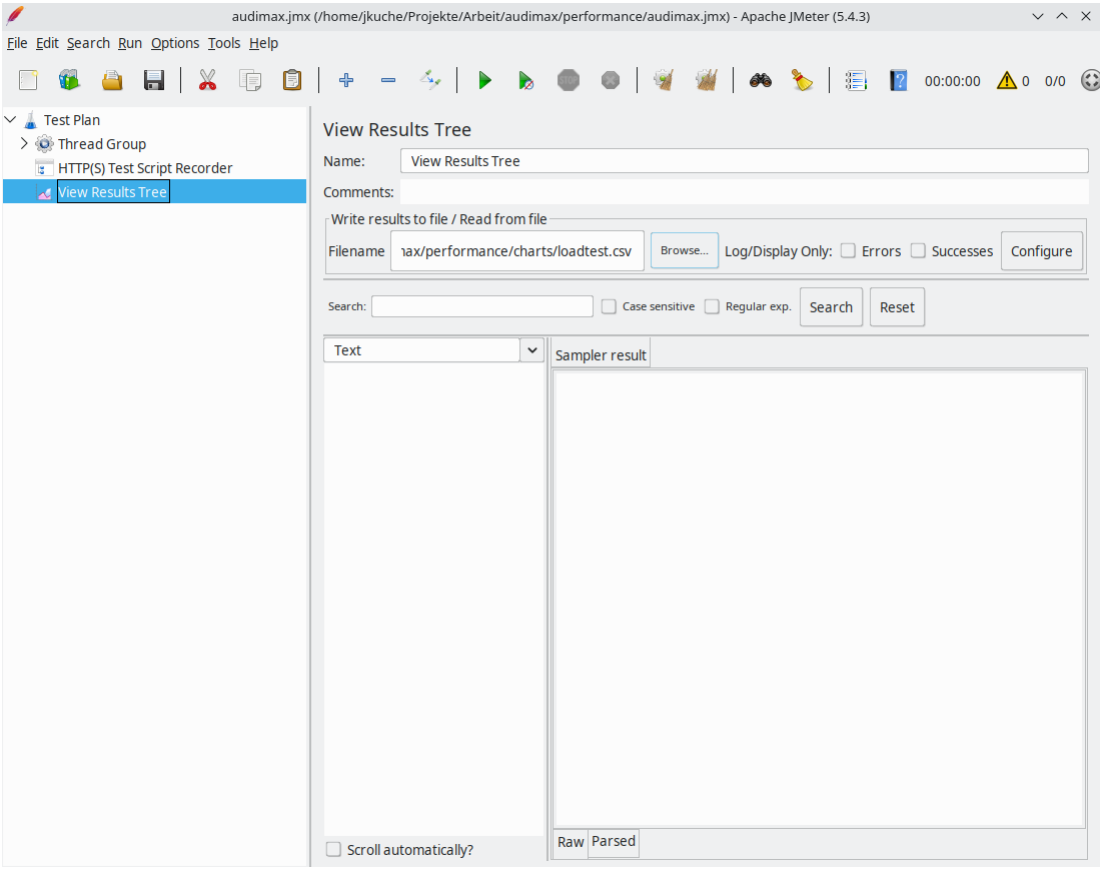

Abbildung 5.10: Ein Result-Knoten in JMeter

Diese können dann durch ein Python-Skript mittels der `pandas`-Bibliothek geladen und mit `seaborn` grafisch dargestellt werden.

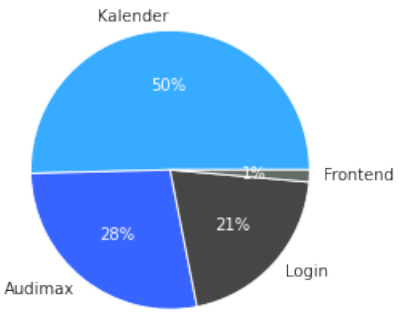

Abbildung 5.11: Anfragezeit Graphen nach Anfragetyp

Zum Beispiel kann anhand dieses Graphen gesehen werden, welche Funktionen von Audimax die meiste Last verursachen. Dies ist für die Priorisierung von Performanceverbesserungen interessant.

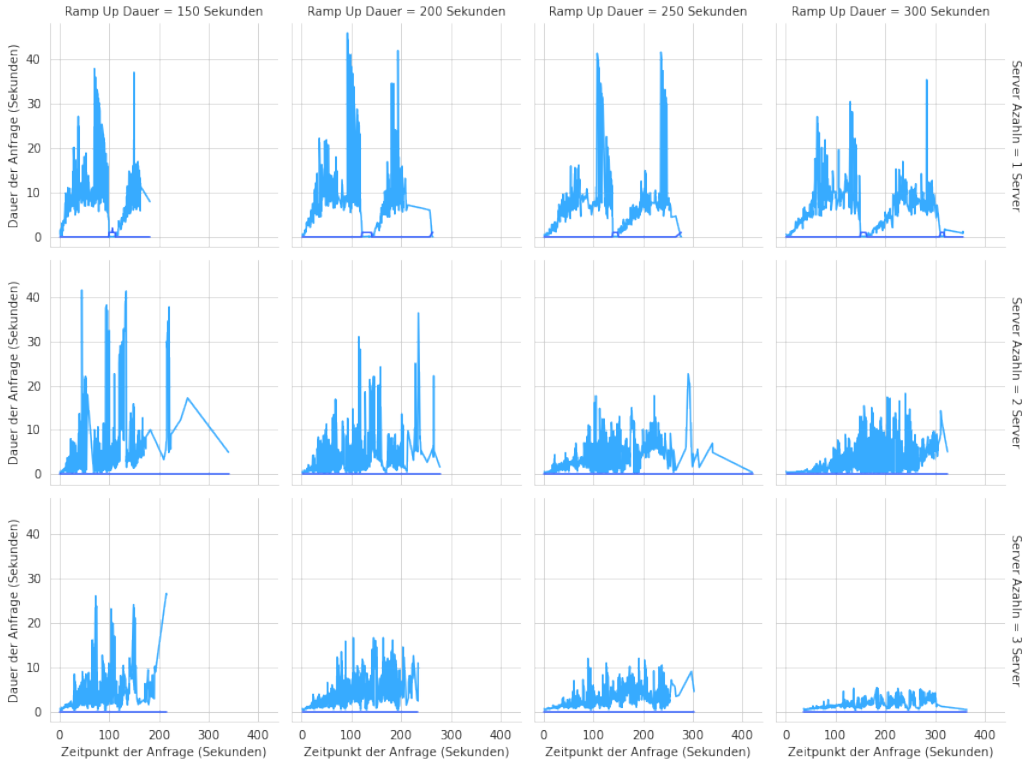

Abbildung 5.12: Antwortzeitgraphen nach Skalierungsfaktor und Anfragezeitfenster

Diese Graphen zeigen die Antwortzeiten des oben beschriebenen Lasttestes. Die Zeilen enthalten geänderte Ramp-Up-Zeit und die Spalten eine geänderte Anzahl von Audimax-Backend-Replicas. Es ist zu sehen, dass die Erhöhung der Replica-Anzahl zu einer deutlichen Verbesserung der Antwortzeiten bei gleicher Ramp-Up-Zeit führt. Auch ist zu sehen, dass die Verwendung von drei Replicas zu einer guten Antwortzeit bei 600 simulierten Nutzern führt.

6 Fazit

Die in Kapitel 1 definierten Ziele konnten durch eine Kombination aus dem ausgewählten *Container-Orchestrierungs-System* „K3s“, einem auf Grafana und Prometheus basierendem *Monitoring-System* und dem entwickelten Kubernetes-Operator für Audimax erfüllt werden.

6.1 Zusammenfassung

Es wurden die Kriterien Verteilbarkeit, Skalierbarkeit, Überwachbarkeit, Verbreitung, Ökosystemunterstützung, Erweiterbarkeit, Effizienz und Sicherheit für die Auswahl eines *Container-Orchestrierungs-Systems* für Audimax aufgestellt und auf Basis dieser *K3s* ausgewählt. Die Erstellung eines Helm-Charts zur automatischen und anpassbaren Installation von Audimax wurde erläutert und die Funktionsweise eines Audimax-Operators erklärt, der in der Lage ist, eine Audimax-Instanz mithilfe dieses *Helm-Charts* automatisch zu installieren. Auch ermöglicht dieser Operator das flexible Konfigurieren von Audimax-Instanzen, durch Erben von Werten aus übergeordneten Konfigurationen. Die Erstellung eines *K3s*-Clusters mit Erweiterung durch *Prometheus*, welches Fehlerfälle erkennt und die betroffene Komponente neu startet oder Benachrichtigungen versendet, wurde erklärt. Auch wurde als Lasttestlösung eine Mischung aus *JMeter* und *Python* ausgewählt. Wobei *JMeter* zum Erstellen und Durchführen des Lasttestszenarios verwendet wird und *Python* zur Visualisierung der Ergebnisse. Die Ergebnisse des Lasttests wurden ausgewertet und festgestellt, dass für das Bewältigen der erwarteten Nutzerlast drei Server notwendig sind.

6.2 Ausblick

In der momentanen Version des Audimax-Operators muss die Erstellung der Audimax-Installation noch durch Administrierende angestoßen und kundenspezifische Änderungen der Konfiguration vorgenommen werden. Diese Notwendigkeit könnte durch Integration in ein Kundenverwaltungssystem, welches Instanzen automatisiert erstellt, eliminiert werden. Auch könnte es Kunden ermöglicht werden, über dieses bestimmte

Konfigurations-Optionen über ein nutzerfreundliches System bei der Ersteinrichtung auszuwählen und eigenständig zu ändern. Ein Beispiel für solche Konfigurationen sind etwa Zugangsdaten für externe Dienste, welche in Audimax eingebunden werden oder externe Authentifikationsserver, welche die Nutzer authentifizieren sollen. Auch unterstützt der Operator momentan nur Einfachvererbung, eine Konfiguration kann also nur von einer anderen Konfiguration erben. Dies könnte, um die Möglichkeit erweitert werden von mehreren Konfigurationen zu erben, um es etwa zu ermöglichen, dass mehrere Basiskonfigurationen miteinander kombiniert werden können.

Literaturverzeichnis

[Apa22a] Apache: AB linux man page (2022), URL https://linux.die.net/man/1/ab, zuletzt abgerufen 2022-05-01

[Apa22b] Apache: JMeter User's Manual (2022), URL https://jmeter.apache.org/usermanual/index.html, zuletzt abgerufen 2022-05-01

[Bur19] Burns, Brendan: *Kubernetes - up and running: Dive into the future of Infrastructure*, O'Reilly Media, Incorporated (2019)

[Dei18] Deimeke, Dirk; Kania, Stefan; Soest, Daniel van; Heinlein, Peer und Miesen, Axel: *Linux-server das umfassende handbuch*, Rheinwerk Verlag (2018)

[Doc22a] Docker: Collect docker metrics with prometheus (2022), URL https://docs.docker.com/config/daemon/prometheus/, zuletzt abgerufen 2022-05-06

[Doc22b] Docker: Deploy services to a swarm (2022), URL https://docs.docker.com/engine/swarm/services/

[Doc22c] Docker: Docker Overview (2022), URL https://docs.docker.com/get-started/overview/, zuletzt abgerufen 2022-04-24

[Doc22d] Docker: Docker PS (2022), URL https://docs.docker.com/engine/reference/commandline/ps/, zuletzt abgerufen 2022-05-01

[Doc22e] Docker: Docker stats (2022), URL https://docs.docker.com/engine/reference/commandline/stats/, zuletzt abgerufen 2022-05-01

[Doc22f] Docker: Dockerd (2022), URL https://docs.docker.com/engine/reference/commandline/dockerd/#examples

[Doc22g] Docker: Protect the docker daemon socket (2022), URL https://docs.docker.com/engine/security/protect-access/

[Doc22h] Docker: Swarm mode key concepts (2022), URL https://docs.docker.com/engine/swarm/key-concepts/

[Doc22i] Docker: What is a container? (2022), URL https://www.docker.com/resources/what-container/, zuletzt abgerufen 2022-04-22

[Git22a] GitHub: Canonical/Microk8s: Microk8s is a small, fast, single-package Kubernetes for developers, IOT and edge. (2022), URL https://github.com/canonical/microk8s, zuletzt abgerufen 2022-05-01

[Git22b] GitHub: Kubernetes/Kubernetes: Production-Grade Container Scheduling and management (2022), URL https://github.com/kubernetes/kubernetes, zuletzt abgerufen 2022-05-01

[Git22c] GitHub: Moby/Moby: Moby Project - a collaborative project for the container ecosystem to assemble container-based systems (2022), URL https://github.com/moby/moby, zuletzt abgerufen 2022-05-01

[Git22d] GitHub: Openshift/OKD: The self-managing, auto-upgrading, Kubernetes Distribution For Everyone (2022), URL https://github.com/openshift/okd, zuletzt abgerufen 2022-05-01

[Git22e] GitHub: Rancher/Dashboard: Rancher New Dashboard UI (2022), URL https://github.com/rancher/dashboard, zuletzt abgerufen 2022-05-01

[Git22f] GitHub: Rancher/Rancher: Complete Container Management Platform (2022), URL https://github.com/rancher/rancher, zuletzt abgerufen 2022-05-01

[Gra22] Grafana: Grafana: The Open Observability Platform (2022), URL https://grafana.com/, zuletzt abgerufen 2022-05-02

[Hel22a] Helm: (2022), URL https://helm.sh/, zuletzt abgerufen 2022-05-06

[Hel22b] Helm: Helm ChartDependencies (2022), URL https://helm.sh/docs/topics/charts/#chart-dependencies, zuletzt abgerufen 2022-05-06

[Ing22] Ingress, Nginx: Nginx Ingress Controller (2022), URL https://kubernetes.github.io/ingress-nginx/, zuletzt abgerufen 2022-05-06

[K3s22] K3s: K3s - Lightweight Kubernetes (2022), URL https://k3s.io/, zuletzt abgerufen 2022-04-24

[Kub20] Kubernetes: Introducing container runtime interface (CRI) in Kubernetes (2020), URL https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/, zuletzt abgerufen 2022-04-22

[Kub21a] Kubernetes: Custom Resources (2021), URL https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[Kub21b] Kubernetes: The Kubernetes Api (2021), URL https://kubernetes.io/docs/concepts/overview/kubernetes-api/

[Kub22a] Kubernetes: Configure liveness, readiness and startup probes (2022), URL https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

[Kub22b] Kubernetes: Container runtime interface (CRI) (2022), URL https://kubernetes.io/docs/concepts/architecture/cri/

[Kub22c] Kubernetes: Custom Resource Definitions (2022), URL https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions, zuletzt abgerufen 2022-04-22

[Kub22d] Kubernetes: Installing kubeadm (2022), URL https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

[Kub22e] Kubernetes: Installing kubeadm (2022), URL https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

[Kub22f] Kubernetes: Kubernetes commands (2022), URL https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands, zuletzt abgerufen 2022-05-01

[Kub22g] Kubernetes: Kubernetes Controller (2022), URL https://kubernetes.io/docs/concepts/architecture/controller/, zuletzt abgerufen 2022-04-22

[Kub22h] Kubernetes: Kubernetes ingress (2022), URL https://kubernetes.io/docs/concepts/services-networking/ingress/, zuletzt abgerufen 2022-04-22

[Kub22i] Kubernetes: Kubernetes Namespace (2022), URL https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/, zuletzt abgerufen 2022-04-22

[Kub22j] Kubernetes: Kubernetes Objects (2022), URL https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, zuletzt abgerufen 2022-04-22

[Kub22k] Kubernetes: Kubernetes Operator (2022), URL https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, zuletzt abgerufen 2022-04-22

[Kub22l] Kubernetes: Kubernetes Pod (2022), URL https://kubernetes.io/de/docs/concepts/workloads/pods/, zuletzt abgerufen 2022-04-22

[Kub22m] Kubernetes: Kubernetes Servie (2022), URL https://kubernetes.io/docs/concepts/services-networking/service/, zuletzt abgerufen 2022-04-22

[Kub22n] Kubernetes: Kubernetes Workload Resources (2022), URL https://kubernetes.io/docs/concepts/workloads/controllers/, zuletzt abgerufen 2022-05-01

[Kub22o] Kubernetes: Nodes (2022), URL https://kubernetes.io/docs/concepts/architecture/nodes/

[Kub22p] Kubernetes: Using RBAC authorization (2022), URL https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[Kub22q] Kubernetes: Volumes (2022), URL https://kubernetes.io/docs/concepts/storage/volumes/, zuletzt abgerufen 2022-04-22

[Lah20] Lahn, Mark: How do I install docker on ubuntu? (2020), URL https://www.servermania.com/kb/articles/install-docker-ubuntu/

[Luk18] Lukša, Marko: *Kubernetes in action: Anwendungen in Kubernetes-Clustern Bereitstellen und verwalten*, Hanser (2018)

[Mic22a] Microk8s: Microk8s - get started: Microk8s (2022), URL https://microk8s.io/docs/getting-started, zuletzt abgerufen 2022-05-01

[Mic22b] Microk8s: Microk8s - get started: Microk8s (2022), URL https://microk8s.io/docs/getting-started, zuletzt abgerufen 2022-05-01

[OKD22a] OKD: OKD Architecture overview (2022), URL https://docs.okd.io/latest/architecture/index.html, zuletzt abgerufen 2022-05-01

[OKD22b] OKD: OKD Prerequisites (2022), URL https://docs.openshift.com/enterprise/3.0/install_config/install/prerequisites.html, zuletzt abgerufen 2022-05-01

[OKD22c] OKD: Using the OKD dashboard to get cluster information (2022), URL https://docs.okd.io/4.10/web_console/using-dashboard-to-get-cluster-information.html, zuletzt abgerufen 2022-05-01

[ope22] Operatorframework (2022), URL https://operatorframework.io/, zuletzt abgerufen 2022-05-06

[PO22] Prometheus-Operator: Kube-Prometheus: Use prometheus to monitor kubernetes and applications running on kubernetes (2022), URL https://github.com/prometheus-operator/kube-prometheus, zuletzt abgerufen 2022-05-06

[Pro22] Prometheus: Prometheus - Monitoring System Time Series Database (2022), URL https://perometheus.io/, zuletzt abgerufen 2022-05-02

[Ran22a] Rancher: Architecture (2022), URL https://rancher.com/docs/rancher/v2.6/en/overview/architecture/, zuletzt abgerufen 2022-05-01

[Ran22b] Rancher: Installation options (2022), URL https://rancher.com/docs/k3s/latest/en/installation/install-options/, zuletzt abgerufen 2022-05-06

[Ran22c] Rancher: Installation requirements (2022), URL https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/, zuletzt abgerufen 2022-05-01

[Ran22d] Rancher: K3s - Quick start (2022), URL https://rancher.com/docs/k3s/latest/en/quick-start/, zuletzt abgerufen 2022-05-02

[Ran22e] Rancher: Rancher (2022), URL https://rancher.com/, zuletzt abgerufen 2022-04-24

[Ran22f] Rancher: Rancher Architecture (2022), URL https://rancher.com/docs/k3s/latest/en/architecture/, zuletzt abgerufen 2022-05-01

[Ran22g] Rancher: Rancher Installation requirements (2022), URL https://rancher.com/docs/rancher/v2.6/en/installation/requirements/, zuletzt abgerufen 2022-05-01

[Red19] Redhat: Was ist container-orchestrierung? (2019), URL https://www.redhat.com/de/topics/containers/what-is-container-orchestration, zuletzt abgerufen 2022-04-22

[Toz20] Tozzi, Christopher: Kubernetes distribution: What it is and what it isn't (2020), URL https://containerjournal.com/topics/container-ecosystems/kubernetes-distribution-what-it-is-and-what-it-isnt/, zuletzt abgerufen 2022-04-24

[Tra22a] Traefik: Traefik/Traefik-Helm-chart: Traefik V2 Helm Chart (2022), URL https://github.com/traefik/traefik-helm-chart#installing, zuletzt abgerufen 2022-05-02

[Tra22b] Traefik.io: Traefik & Docker (2022), URL https://doc.traefik.io/traefik/providers/docker/, zuletzt abgerufen 2022-05-06

[Tra22c] Traefik.io: Traefik & Kubernetes (2022), URL https://doc.traefik.io/traefik/providers/kubernetes-ingress/, zuletzt abgerufen 2022-05-06

[Wol18] Wolff, Eberhard: *Microservices Grundlagen flexibler softwarearchitekturen*, dpunkt.verlag (2018)

Abbildungsverzeichnis

Listings

.