HAW HAMBURG

Bachelorarbeit

Felix Roth

Monitoring eines Multi-Tenancy Kubernetes Clusters am Beispiel der HAW Hamburg Compute Cloud

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Felix Roth

Monitoring eines Multi-Tenancy Kubernetes Clusters am Beispiel der HAW Hamburg Compute Cloud

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Martin Hübner
Zweitgutachter: Prof. Dr. Stefan Sarstedt
Fachlicher Betreuer: M. Sc. Tobias Eichler

Eingereicht am: 17. November 2020

Felix Roth

Thema der Arbeit

Monitoring eines Multi-Tenancy Kubernetes Clusters am Beispiel der HAW Hamburg Compute Cloud

Stichworte

Kubernetes, Informatik Compute Cloud, Monitoring, Prometheus, Prometheus-Operator, Grafana, Docker

Kurzzusammenfassung

Ziel dieser Arbeit ist es mit Hilfe eines Monitoringtools eine Überwachungsstrategie für ein Multi-Tenancy Kubernetes Cluster zu entwickeln. Dabei werden verschiedene Tools auf ihre Einsatzmöglichkeiten in der Informatik Compute Cloud (ICC) der HAW-Hamburg analysiert. Aus dem Ergebnis der Analyse, Monitoringtool Prometheus, wurde mit Hilfe des Projektes Prometheus-Operator eine für die ICC passende Architektur entwickelt.

Felix Roth

Title of Thesis

Monitoring of a multi-tenancy kubernet cluster refering the Compute Cloud of HAW Hamburg

Keywords

Kubernetes, Informatics Compute Cloud, Monitoring, Prometheus, Prometheus-Operator, Grafana, Docker

Abstract

The thesis develops a monitoring strategy for a multi-tenancy kubernetes cluster using a monitoring tool. Different tools are beeing analyzed for their use in the Informatik Compute Cloud (ICC) of HAW-Hamburg. As the result of this analysis is monitoring tool Prometheus, an architecture, suitable for the ICC, was developed with the help of the project Prometheus-Operator.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcode

1 Einleitung

1.1 Problemstellung

In den letzten Jahren hat das Thema Cloud Computing immer mehr an Relevanz gewonnen. Nach Zahlen von Bitkom nutzen drei von vier Unternehmen in Deutschland im Jahr 2018 die Rechenleistung aus der Cloud (vgl. [Bit19]). Zur Unterstützung des Themas in der Lehre hat das Departement Informatik der HAW Hamburg eine eigene Compute Cloud entwickelt. Die sogenannte Informatik Compute Cloud (ICC) ist eine auf Kubernetes basierende Cloud, die von verschiedenen Gruppen und Anwender*innen benutzt wird, wie z.B. von Studierenden für Praktikumsaufgaben oder von Projektgruppen für das Entwickeln intelligenter Systeme.

Das Überwachen (Montitoring) der Cloud oder einer Anwendung in der Cloud stellt sowohl Administrator*innen des Cloud-Systems als auch Entwickler*innen einer Anwendung vor neue Herausforderungen. Die Administrator*innen wollen das Gesamtsystem überblicken um die Nutzung des Systems gewährleisten zu können. Entwickler*innen möchten ihre Anwendung im System überwachen und einen Überblick erhalten, ob die systemrelevanten Prozesse ihrer Anwendung funktionieren. Sowohl möchten beide Usergruppen definierte Benachrichtigungen über ihren Systemstatus bzw. Anwendungsstatus erhalten, als auch über Fehlfunktionen informiert werden.

Hierbei haben sich in den letzten Jahren verschiedene neue Tools und Strategien etabliert. Die Tools unterscheiden sich in ihrer Herangehensweise und Zielsetzung. Da die Informatik Compute Cloud der HAW Hamburg ein sogenanntes Multi-Tenancy Kubernetes Cluster ist, besteht die Herausforderung für viele unterschiedliche Usergruppen passende isolierte Überwachungsstrategien anzubieten. Es darf nicht möglich sein, dass eine fehlerhafte Konfiguration des Monitoringsystem in einem Namespace, Auswirkung auf andere User*in hat.

Da es sich hierbei um eine komplexe Aufgabenstellung handelt, kann gegenwärtig keine Standardlösung angewendet werden. Für diese spezielle Anwendung in der Informatik Compute Cloud der HAW Hamburg muss ein Tool gefunden werden, das auf die individuellen Anforderungen der zahlreichen Usergruppen eingeht.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, eine Monitoring Infrastruktur für die Informatik Compute Cloud (ICC) des Department Informatik der HAW Hamburg zu entwickeln und anschließend in das System zu integrieren. In dieser Arbeit werden Anwendungsszenarien für die unterschiedlichen Usergruppen untersucht und die auf dem Markt momentan zur Verfügung stehenden Tools recherchiert.

Mit Hilfe einer Nutzwertanalyse werden Parameter bestimmt, die die Eignung der Monitoring Anwendung bewerten. Ziel ist es, das Tool zu finden, das am besten auf die ICC zutrifft. Im Weiteren soll für das evaluierte Tool eine, für die ICC passende, System-Architektur und Konfiguration definiert werden. Diese soll mit Hilfe von Anwendungsszenarien, sogenannten User-Storys, getestet und anschließend in die Informatik Compute Cloud (ICC) integriert werden.

1.3 Struktur der Arbeit

Der erste Abschnitt dieser Arbeit dient als Einführung in das Thema, zugleich wird auch die Zielsetzung, sowie der Aufbau der Arbeit erläutert. Im darauf folgenden Kapitel 2 werden die relevanten Grundlagen aufgegriffen, die für das Verständnis der sachlichen Zusammenhänge dieser Arbeit notwendig sind. Es werden verschiedene Überwachungsmethoden vorgestellt, so wie auch das Kubernetes System erläutert.

Im Kapitel 3 wird mit Hilfe einer Anforderungsanalyse und einer Nutzwertanalyse aus verschiedenen Monitoring Tools ein passendes für den speziellen Einsatz in der ICC ausgewertet. Anschließend behandelt Kapitel 4 die Konzeption, die Umsetzung erfolgt in Kapitel 5. Das Kapitel 6 erläutert die Evaluation der Anforderungsanalyse aus Kapitel 3. Das abschließende Kapitel 7 fasst die Arbeit zusammen und gibt ein Ausblick über mögliche Optimierungen.

2 Grundlagen

In diesem Kapitel werden im Folgenden die relevanten Grundlagen der Kubernetes-Thematik sowie des Monitorings aufgegriffen und erläutert.

2.1 Docker-Container

Die Container Technologie in der Softwareentwicklung lässt sich mit der Technologie der Virtuellen Maschinen (VM) vergleichen. In beiden Bereichen können Anwendungen isoliert auf einem Host-System gestartet werden. VM beinhalten in der Regel ein eigenes Betriebssystem, das führt dazu, dass sie deutlich schwergewichtiger und Ressourcen verbrauchender als Docker-Container sind. Container hingegen bilden nur die Applikation und die für die Ausführung benötigen Dateien ab und sind aus diesem Grund deutlich agiler. (vgl. [Red20b])

Docker ist eine Software, die das Erstellen, Starten und Managen zum Zeitpunkt der Laufzeit von Containern übernimmt. Die auszuführende Applikation, die innerhalb eines Docker-Containers laufen soll, beschreibt ein Docker-Image, dieses wird über eine Dockerfile definiert. Über eine Docker-Registry (z.b. Docker Hub) können Docker-Images veröffentlicht werden, bzw. veröffentlichte Docker-Images verwendet werden.

Container können über *localhost* und Port kommunizieren, solange sie auf derselben Maschine laufen.

2.2 Kubernetes

Google hat 2014 das Open-Source-Projekt Kubernetes gestartet. Innerhalb dieses Projekts wurde ein Container-Orchestrierungs-System entwickelt, das von jedem verwendet werden kann. (vgl. S.12 [AD19])

Kubernetes automatisiert die Verwaltung von Container basierten Applikationen bzw. Services. Dabei eliminiert Kubernetes viele manuelle Prozesse, die das Bereitstellen und Skalieren von Applikationen in Containern vereinfacht (vgl. [Red20a]).

2.2.1 Kubernetes-Cluster

Ein Kubernetes-Cluster besteht aus mehreren Nodes, die auf verschiedenen Maschinen laufen. Die Abbildung 2.1 zeigt schematisch den Aufbau eines Kubernetes-Clusters. Das Kubernetes-Cluster besteht immer aus mindestens einer Master-Node die das Verwalten der Kubernetes-Cluster übernimmt. Dies beinhaltet auch den API-Server, der die zentrale Kommunikationsschnittstelle für alle User*innen ist. Auf den Worker-Nodes werden die Applikationscontainer in den einzelne Pods gestartet und verwaltet.

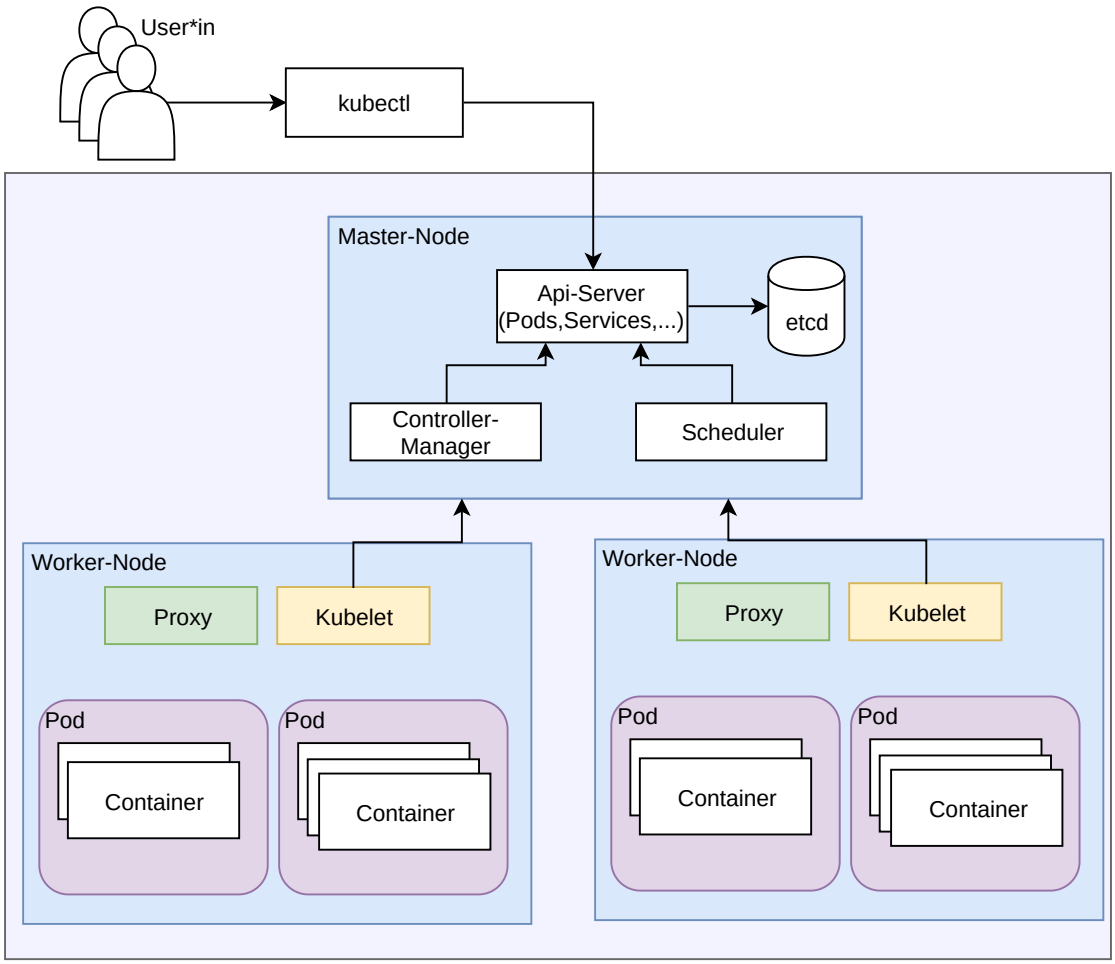

Abbildung 2.1: Kubernetes-Cluster (vgl. [Kot20])

Master-Node

Ein Kubernetes-Cluster besteht immer aus mindestens einem Master-Node. Diese stellt die Steuerungsebene des Clusters dar (vgl. [Kub20l]). Sie verwaltet den Controller-Manager, den Scheduler und den Api-Server. Über den Api-Server können User*innen mit Hilfe von kubectl mit dem Cluster kommunizieren.

Api-Server: Stellt die Kubernetes-API zur Verfügung und ist das Frontend der Kubernetes-Steuerungsebene. (vgl. [Kub20g])

Controller-Manager: Besteht aus dem *Node Controller*, dem *Replication Controller*, dem *Endpoints Controller* und dem *Service Account & Token Controller* (vgl. [Kub20h]). Jeder dieser Controller verwaltet einen Bereich des System.

Scheduler: Überwacht einen neu erstellten Pod und entscheidet auf welcher Node er ausgeführt werden soll (vgl. [Kub20j]).

etcd: Ist ein hochverfügbarer und konsistenter Key-Value Speicher, der Clusterdaten des Kubernetes System speichert (vgl. [Kub20m]).

Worker-Node

Ein Worker-Node stellt die Kubernetes-Laufzeitumgebung bereit. Ein Worker-Node wird von dem Master-Node bzw. den Master-Nodes verwaltet und beinhaltet alle für den Betrieb von Pods notwendigen Dienste, wie z.B. der Container Runtime, das Kubelet und den Kube-Proxy. (vgl. [Kub20p])

Container-Runtime: Ist die Software, die für das Laufen der einzelnen Container zuständig ist. Kubernetes unterstützt unterschiedliche Container-Runtimes, bspw. Docker, containerd und CRI-O. (vgl. [Kub20f])

Pod: Ist in Kubernetes die kleinste einsetzbare Einheit. Ein Pod besteht aus einem oder mehreren Containern, die sich eine Speicher-Ressource bzw. eine Netzwerk-Ressource teilen. Außerdem beinhaltet er die Spezifikation, die definiert, wie die Container gestartet werden sollen. (vgl. [Kub20q])

Kubelet: Stellt sicher, dass ein Container innerhalb eines Pods ausgeführt wird (vgl. [Kub20k]).

Proxy: Ist für die jeweiligen Requests zwischen Pods auf den verschiedenen Worker-Nodes zuständig und verbindet einen Pod mit dem Internet (vgl. S.37 [AD19]). Außerdem wird es für die Kubernetes Service-Abstraktion benötigt (vgl. [Kub20i]).

2.2.2 Kubernetes-Deployment

Ein Kubernetes-Deployment definiert die Konfiguration eines Pods auf einer Worker-Node. Es definiert einen gewünschten Zustand, den eine Applikation haben soll. Der Deployment-Contoller ändert den Ist-Zustand in den gewünschten Zustand. (vgl. [Kub20d])

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-apache-deployment
  labels:
    app: my-apache
spec:
  selector:
    matchLabels:
      app: my-apache
  replicas: 2
  template:
    metadata:
      labels:
        app: my-apache
    spec:
      containers:
      - name: my-apache
        image: httpd:latest
        ports:
        - containerPort: 80
```

Quellcode 2.1: Kubernetes-Deployment

Der Quellcode 2.1 zeigt ein Beispiel-Deployment im YAML-Format. Ein Deployment benötigt immer eine *apiVersion* und wird mit *kind: Deployment* als Deployment definiert. Das Beispiel-Deployment erstellt ein Deployment *my-apache-deployment*, dies definiert der *metadata:* Bereich. Der Bereich *spec:* definiert die Konfiguration des Pods. Mit *replicas:* kann die Anzahl der Repliken eines Pods definiert werden, i.d.F. zwei. Im Bereich *template:* wird der Container definiert und konfiguriert. Das Beispiel-Deployment definiert einen Container mit dem Image *httpd:latest* von Docker Hub. Dabei handelt es sich um einen Apache Server.

2.2.3 Service

Um die Kommunikation zwischen Applikationen im Cluster zu erleichtern, kann ein Service definiert werden. Der Kubernetes Service ist eine Abstraktion der Richtlinie und definiert, wie auf einen Pod bzw. auf mehrere logisch zusammenhängende Pods zugegriffen werden kann (vgl. [Kub20t]).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-apache-service
spec:
  selector:
    app: my-apache
  ports:
  - name: http
    port: 80
    targetPort: 80
```

Quellcode 2.2: Kubernetes-Service

Für das Beispiel-Deployment *my-apache-deployment* definiert der Quellcode 2.2 den Service, mit dem auf die Applikation zugegriffen werden kann. Er bestimmt unter dem Namen *my-apache-service* einen Service, der mit Hilfe des *selector:* auf das im Deployment definierte Label *app: my-apache* verweist. Über *ports:* wird der Port definiert, unter dem die Applikation betrieben wird.

2.2.4 Namespace

Um Namenskonflikte bzw. Ressourcenkonflikte in einem Cluster mit vielen unterschiedlichen Nutzenden zu vermeiden, können virtuelle Cluster unter dem Namen *Namespace* definiert werden. Innerhalb eines Namespaces müssen die Namen einer Ressource eindeutig bestimmt werden, dies gilt jedoch nicht über die Grenzen eines Namespaces hinaus (vgl. [Kub20o]).

2.2.5 Autorisierung über RBAC

Für die Autorisierung innerhalb des Kubernetes-Clusters kann der API-Server die Modes Node, ABAC, RBAC und Webhook (vgl. [Kub20a]) benutzen. Die Informatik Compute Cloud (ICC) der HAW-Hamburg verwendet dem Mode RBAC.

Role-based access control (RBAC) ist eine Methode zur Definition von Zugriffen innerhalb der Cluster. Diese werden mit Hilfe von bestimmten Rollen an Nutzende vergeben (vgl. [Kub20u]). In Kubernetes können mit Hilfe von Role bzw. ClusterRole bestimmte Berechtigungen vergeben werden. Roles sind Berechtigungen, die innerhalb eines Namespaces

definiert werden und auch nur innerhalb dieses Namespaces verwendet werden können. ClusterRole dagegen sind im gesamten Cluster verfügbar. (vgl. [Kub20r])

Mit Hilfe von RoleBilding bzw. ClusterRoleBinding werden einem *User* Account oder einem *ServiceAccount* die, in der Role bzw. ClusterRole definierten, Berechtigungen gewährt (vgl. [Kub20s]).

User: Ist eine Person, die das Kubernetes Cluster nutzt.

ServiceAccount: Ist ein Dienstkonto, das für Prozesse, die in einem Pod laufen, die Identität bietet (vgl. [Kub20b]).

2.2.6 kubectl

kubectl ist ein Kommandozeilen-Tool mit dem Befehle gegen das Kubernetes-Cluster ausgeführt werden kann. Damit können Anwendungen bereitgestellt, Cluster-Ressourcen inspiziert und verwaltet, sowie Protokolle eingesehen werden. (vgl. [Kub20e])

Nachfolgend werden einige wichtige *kubectl* Befehle gezeigt:

```
$kubectl get deploy,pod,service
```

Mit *get* werden die nachfolgenden Ressourcen, in diesem Fall Deployment, Pod und Service, aufgelistet. Für das Beispiel-Deployment bzw. dem Beispiel-Service *my-apache* sieht die Ausgabe wie folgt aus:

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-apache-deployment    2/2     2            2           19s

NAME                                        READY   STATUS    RESTARTS   AGE
pod/my-apache-deployment-5b45d8d84c-9x67d   1/1     Running   0          19s
pod/my-apache-deployment-5b45d8d84c-nr6lf   1/1     Running   0          19s

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP PORT(S) AGE
service/my-apache-service    ClusterIP   10.96.153.121   <none>      80/TCP  19s
```

Mit *apply -f* wird die nachfolgende Datei in das Cluster geladen. In diesem Fall handelt es sich hierbei um die deployment.yaml Datei des *my-apache*-Beispiels.

```
$kubectl apply -f deployment.yaml
```

Um eine Ressource aus dem Kubernetes-Cluster zu löschen wird der Befehl *delete* verwendet. Dies bewirkt, dass die nachfolgenden Ressourcen gelöscht werden, hier das *my-apache-deployment*.

```
$kubectl delete deployment.apps/my-apache-deployment
```

Mit Hilfe von *describe* werden die Details der nachfolgenden Ressource angezeigt.

```
$kubectl describe service/my-apache-service
```

Die Details des Beispiel-Services *my-apache-service* sehen wie folgt aus:

```
Name:              my-apache-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=my-apache
Type:              ClusterIP
IP:                10.96.153.121
Port:              http  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.1.15:80,10.244.1.16:80
Session Affinity:  None
Events:            <none>
```

2.2.7 CustomResourceDefinitions

Mit Hilfe von CustomResourceDefinition (CRD) ist es möglich, der Kubernetes-API eigene Ressourcen hinzuzufügen. Dadurch wird das Kubernetes-Cluster mit diesen Ressourcen erweitert. Die hinzugefügten Ressourcen der CRDs können mit Hilfe von *kubectl* verwendet werden. (vgl. [Kub20c])

2.2.8 Kind

Kind ist ein Tool, mit dem ein lokales Kubernetes-Cluster auf Basis des Docker-Containers *nodes* gestartet werden kann. Es wurde dafür entwickelt Kubernetes zu testen, kann aber auch für lokale Deployment-Tests verwendet werden. (vgl. [Kin20])

2.2.9 Informatik Compute Cloud (ICC)

Die Informatik Compute Cloud der HAW-Hamburg ist ein Kubernetes-Cluster, das vom AI Labor des Departements Informatik zu Verfügung gestellt wird. Alle Mitglieder des Departements können Applikationen in Form von Docker-Containern innerhalb des Clusters betreiben. (vgl. [Use20])

Bei der ICC handelt es sich um ein *multi tenancy* Cluster. Das *multi tenancy* Konzept sieht vor, dass sich Nutzende dieselben Ressourcen teilen, dabei aber datentechnisch immer vollständig getrennt voneinander sind (vgl. [ITW20]). In einem Kubernetes-Cluster werden die einzelnen User*innen mit Hilfe von Namespaces isoliert, zusätzlich bekommen sie nur die Berechtigungen, um ihre Applikation in ihrem Namespace zu integrieren.

2.3 Monitoring

Monitoring in der Informatik bedeutet grundsätzlich die Überprüfung einer Applikation auf ihre korrekte Funktionalität. Antwortet bspw. mein Service auf eine Anfrage oder zeigt meine Webseite die richtigen Informationen an? Im DevOps-Kontext meint Monitoring in der Regel das automatisierte Überwachen einer Applikation, einem Service oder eines Systems. (vgl. S.287 [AD19]) Dabei werden Tools eingesetzt, die eine Applikation überwachen und, die bei bestimmten Ereignissen Mitteilungen an die Entwickler*innen sendet.

2.3.1 Arten des Monitoring

Für das Überwachen von Systemen bzw. einer Applikation gibt es unterschiedliche Methoden: Das Logging, das Messen von Metriken und Tracing. Diese werden nachfolgend erläutert.

Logging

Beim Logging speichern sich Applikation sogenannte Logs in Log-Dateien ab. Bspw. erstellen Webserver bei jeden Request einen Log mit der angeforderten URL, der IP-Adresse der Clients oder dem HTTP-Status Response (vgl. S.291 [AD19]). Diese Logs helfen dabei entstandene Fehler zu dokumentieren und diese Ereignisse bei der Fehlersuche

zu reproduzieren. Die größte Schwäche des Loggings ist es, dass schon während der Entwicklung entschieden werden muss, welche Informationen geloggt werden sollen. Daher können Logs im Anschluss ausschließlich Probleme beantworten, die bereits während der Entwicklung vorhergesehen werden konnten und sind aus diesem Grund recht unflexibel. (vgl. S.291 [AD19])

Metrik

Eine Metrik ist ein numerischer Messwert mit dem Informationen eines Systems gesammelt werden. Metriken können bspw. folgende Informationen sammeln:

- Anzahl der Request in einer Minute
- Durchschnittliche Antwortzeit eines Request
- Auslastung der CPU, des Speichers oder des Netzwerks

Die dabei gesammelten Metriken können dabei behilflich sein, Fehler besser zu verstehen. Bspw. kann eine hohe Auslastung eines Servers der Grund sein, wieso die Antwortzeit eines Service steigt. Diese kann anschließend mit Hilfe einer Metrik, die die CPU bzw. RAM Auslastung ausliest, entdeckt werden. Zusätzlich helfen Metriken dabei mögliche Fehler im System bereits vor Entstehung zu entdecken. Ein volllaufender Speicher kann z.B. zu einem Ausfall der Applikation führen, mit Hilfe einer Metrik kann dieser jedoch schon frühzeitig erkannt werden und das Problem kann zeitnah behoben werden. (vgl. S.292 ff. [AD19])

Tracing

Tracing ist eine Methode mit der Fehler bei der Kommunikation einzelner Komponente in einem verteiltem System entdeckt werden können. Dabei wird bspw. der Request eines Users von Anfang bis Ende verfolgt, um mögliche Engstellen im System zu entdecken. Eine Engstelle kann bspw. eine Netzwerkkomponente im System sein, die für die Kommunikation zu langsam ist.

3 Analyse und Toolauswahl

3.1 Anforderungsanalyse

Die Anforderungsanalyse definiert die Anforderungen der Nutzergruppen an eine Monitoring Strategie.

3.1.1 Nutzergruppen

Für die Anwendungsanalyse werden die folgende Nutzergruppen betrachtet. Dabei muss beachtet werden, dass die User*innen in verschiedenen Nutzergruppen zugeordnet sein können.

- ICC-Admin
  - Mitarbeitende der ICC-Gruppe. Sie sind die Administrator*innen des Gesamt-Systems der ICC
  - Verfügen über einen vollständigen Root-Zugriff auf das System
- ICC-User*in
  - HAW-Mitarbeitende, Professor*innen oder Projektgruppen, deren Anwendung über einen langen Zeitraum laufen (z.B. Dashboard bei den Fahrstühlen)
  - Zugriff nur über die Kubernetes-API in den jeweiligen 'group namespaces'
- ICC-Studierende
  - Studierende, die innerhalb eines Praktikums die ICC nutzen
  - Zugriff nur über die Kubernetes-API in den jeweiligen 'user namespaces'
- ICC-ML-User*in

- Studierende, die die ICC für maschinelles Lernen nutzen
- Zugriff nur über die Kubernetes-API in den jeweiligen 'user namespaces'

3.1.2 Anforderungen der Nutzergruppen (User-Story)

ICC-Admin

Abbildung 3.1: User-Story für Anforderungen der Nutzer ICC-Admin

Als ICC-Admin möchte ich...

- ... einen Überblick der Auslastung meines System (ICC) haben.
  - Messen der aktuellen CPU-, RAM-, Netzwerk- und Speicherauslastung und Ausgabe in einem Dashboard (Metrik). Dabei soll die Möglichkeit bestehen, die Messung auf die verschiedenen Ebenen (Nodes und Pods) des Clusters zu reduzieren bzw. zu erweitern.
- ... eine Mitteilung (Alerting) bekommen, wenn Teile des Systems (ICC) nicht erreichbar sind oder bestimmte Prozesse (z.B. Kubernetes-API) nicht mehr laufen.
  - Automatisiertes Abfragen der einzelnen Prozesse

  - Mitteilung über E-Mail oder Messenger Dienst (z.B. MS Teams)
  - Zu überwachende Prozesse
- ... eine Liste von nicht laufenden Deployments erhalten (möglicherweise Abschalten oder Besitzer*in benachrichtigen)
  - Automatisiertes Überprüfen aller Pods auf Fehler. Fehlerhafte Pods, die nach einer bestimmten Zeit (z.B. einer Woche) noch den selben Fehler aufweisen, abschalten oder Besitzer*in benachrichtigen.

ICC-User*in

Abbildung 3.2: User-Story für Anforderungen der Nutzer ICC-User*in

Als ICC-User*in möchte ich...

- ... einen Überblick der Auslastung meines Systems (z.B Webserver) haben.
  - Metriken über die Auslastung des Service oder des Pods. Für Webserver wären Standard Metriken für Nginx oder Apache2 hilfreich.

- ... eine Mitteilung (Alerting) bekommen, wenn mein System (z.B Webserver) nicht mehr erreichbar ist.
  - Mitteilung über E-Mail
- ... eine Mitteilung (Alerting) bekommen, wenn der Speicherplatz meines Service oder meines persistenten Volumen z.B. zu 90% belegt ist
  - Mitteilung über E-Mail, Überprüfung des Speicherplatzes mit Hilfe einer Metrik

ICC-Studierende

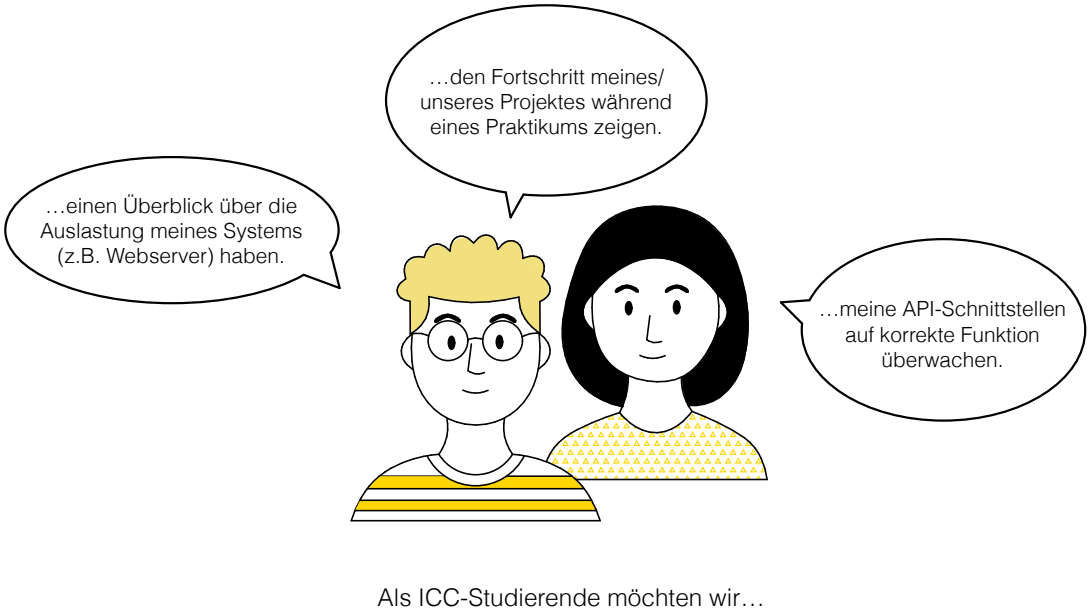

Abbildung 3.3: User-Story für Anforderungen der Nutzer ICC-Student

Als ICC-Student möchte ich...

- ... einen Überblick der Auslastung meines System (z.B Webserver) haben.
  - Metriken über die Auslastung des Service oder des Pods. Für Webserver wären Standard Metriken für Nginx oder Apache2 hilfreich.
- ... meine Fortschritt meines/unseres Projekts während eines Praktikum zeigen.
  - Übersicht aller laufende Pods, sowie die jeweiligen Log-Ausgaben auf einer Webseite (z.B eines Dashboards) anzeigen
- ... meine API-Schnittstellen auf korrekte Funktion überwachen
  - Automatisiertes Überprüfen der API Schnittstellen des Service

ICC-ML-User*in

Abbildung 3.4: User-Story für Anforderungen der Nutzer ICC-ML-User*in

Als ICC-ML-User*in möchte ich...

- ... eine Mitteilung erhalten, wenn ein Trainings-Durchlauf abgeschlossen ist

  - Nach Beendigung eines Jobs eine Mail an den bzw. User*in senden

- ... eine Mitteilung (Alerting) bekommen, wenn der Speicherplatz meines Service z.B. zu 90% belegt ist.
  - Mitteilung über E-Mail. Überprüfung des Speicherplatz mit Hilfe einer Metrik
- ... eine Mitteilung (Alerting) bekommen, wenn mein Job einen Fehler ausgibt
  - Mitteilung über E-Mail, wenn Job mit einem Fehler beendet wird

3.2 Anforderungen an ein Monitoring Tool

Aus der Anwendungsanalyse wurden Anforderungen an ein Monitoring Tool definiert. Dabei teilen sich die Anforderungen in drei zu unterscheidende Bereiche auf. Auf der einen Seite müssen die Anforderungen, die für die Überwachung des Kubernetes System erforderlich sind, definiert werden [3.2.2]. Außerdem müssen die Anforderungen für die Überwachung eines Namespace im Kubernetes Cluster betrachtet werden [3.2.3]. Abschließend müssen die Anforderungen zur Überwachung externer Dienste bzw. externer Prozesse erkannt werden [3.2.4].

3.2.1 Mitteilung (Alerting)

Unabhängig von diesen drei Bereichen gilt die Anforderung, dass das Monitoring Tool den User*innen oder einer Usergruppe über verschiedene Kanäle Mitteilungen senden kann. Es sollte möglich sein E-Mail und auch Messenger-Dienste, wie z.B. MS Teams, als Kommunikationskanäle verwenden zu können.

3.2.2 Überwachen des Kubernetes System

In diesem Abschnitt werden die Anforderungen an ein Tool betrachtet, dass das Kubernetes System überwachen soll.

Ressourcenauslastung

Für die Ressourcenauslastung muss es möglich sein, die verschiedenen Ebenen des Clusters (Gesamtsystem, Nodes und Pods) einzeln zu betrachten. Dabei sollen Ressourcen wie CPU oder Arbeitsspeicher gemessen werden.

Kubernetes Prozesse

Die Kubernetes internen Prozesse, wie z.B der kube-controller-manager oder kubelet, müssen ebenfalls überwacht werden. Dafür bietet Kubernetes einen HTTP-Endpunkt an. Als Anforderung für das Tool gilt es, diesen Endpunkt ebenfalls überwachen zu können.

3.2.3 Überwachen eines Namespaces im Kubernetes Cluster

In diesem Abschnitt werden die Anforderungen an die Überwachung eines Systems im Kubernetes Cluster z.B. einem Webserver betrachtet.

Metriken

Um ein eigenes System zu überwachen, muss die Möglichkeit bestehen, eigene Metriken in das System einzubauen. Hierfür ist es erforderlich, dass das Tool verschiedene Systemumgebungen, als auch verschiedene Client Bibliotheken, für unterschiedliche Programmiersprachen unterstützt.

Systemumgebung

Das Tool sollte für die gängigen Webserver NGINX und Apache Möglichkeiten der Überwachung bereitstellen.

Client Bibliotheken

Für die unterschiedlichen definierten Usergruppen wurden zunächst die wichtigsten Programmiersprachen eruiert. Diese sind einerseits die in den internen Vorlesungen für Programmierungen des Department Informatik der HAW Hamburg vorgestellten Sprachen, wie Ruby, Java, C und C++. Außerdem relevant sind die Sprachen Python, die vor allem in der Entwicklung intelligenter Systeme genutzt wird, und Golang, eine Sprache, die im Bereich der Microservices verwendet wird.

3.2.4 Überwachen eines Prozesses außerhalb des Kubernetes System

Anschließend werden Anforderungen betrachtet, die für externe Prozesse benötigt werden. Folgende Applikationen bzw. Prozesse sollen überwacht werden:

- Es muss möglich sein, eine Applikationen bzw. eine Komponente über das SNMP Protokoll nach deren aktuellen Zustand, Leistung und Fehlern abzufragen. Folgende Applikationen/Komponenten sollen darüber überwacht werden:
  - Netzwerkkomponenten (Nexus Switch)
  - ISCSI Storage (SNMP)
- Für die Überwachung der Temperatur im System muss es möglich sein, die IPMI Sensoren abzufragen (z.B. mit einem IPMI Exporter):
  - Supermicro IPMI
- Es muss möglich sein, das Host System und die darauf laufenden VMs zu überwachen (z.B mit einem Node Exporter):
  - Linux Server
  - Linux VMs
- Für die Überwachung von HAProxy Server muss das Tool die Metriken des HAProxy Server ansprechen können. Dabei kann der Feed direkt im CSV-Format verwendet werden oder die Laufzeit-API kann verwendet werden, um die Daten als JSON zu exportieren.
- Es muss die Möglichkeit bestehen die Applikation Gitlab zu überwachen.

3.3 Nutzwertanalyse

In diesem Abschnitt werden verschiedene Überwachungstools, auf den Einsatz in der ICC hin, analysiert bzw. verglichen. Über eine vorausgehende Definition von Mindestanforderungen wurde zunächst die Toolauswahl begrenzt, um sich ausschließlich mit einer ausgewählten Anzahl dieser zu beschäftigen. Die Mindestanforderungen definieren insbesondere grundlegende Qualitäten in Bezug auf Alerting, Applikations-Metriken und das Überwachen von Kubernetes Prozessen.

3.3.1 Mindestanforderung

Alerting: Es muss möglich sein, mit dem Tool oder einer Erweiterung des Tools, eine Mitteilung an verschiedene User*innen zu senden. Diese könnte z.B. eine Fehlermeldung eines System oder die Benachrichtigung eines abgeschlossenes Kubernetes Jobs beinhalten.

Applikations-Metriken: Es muss möglich sein, für eigene Applikationen Metriken zu erstellen, bspw. für eine in Java geschriebene API-Schnittstelle.

Kubernetes Prozesse: Es muss möglich sein, die unter 3.2.2 erläuterten Prozesse zu überwachen. Diese betreffen insbesondere die für die Nutzung von Kubernetes wichtigen Prozesse wie z.B. kubelet.

Open Source: Das Tool muss Open Source verfügbar sein.

3.3.2 Überblick über die Tools

Mit Hilfe der definierten Mindestanforderungen war es möglich, sich einen genaueren Überblick über die Vielfalt der verschiedenen Tools zu erarbeiten. Mit Hilfe der oben definierten Mindestanforderungen erfolgte eine schnellere Selektion der Tools dahingehend, ob sie für diese Anforderungen zielführend sind oder nicht.

Tool	Alerting	Applikation Metriken	Kubernetes Prozesse	Open Source
Datadog	X	X	X	
Graphite	X	X	X	X
InfluxDB	X	X	X	X
Jaeger				X
Kubewatch	X		X	X
Prometheus	X	X	X	X
Sysdig	X	X	X	

Tabelle 3.1: Auswahl Montitoringtools

Aus der Tabelle 3.1 sind die für diese Analyse zu betrachtenden Monitoring Tools abzulesen. Stellvertretend für den Bereich der kommerziell verfügbaren Tools, die mit einem großen Spektrum an Funktionen punkten können, wurden hier nur Sysdig (vgl. [Sys20]) und Datadog (vgl. [Dat20]) aufgelistet. Die Tools aus dem kommerziellen Bereich sind für andere Anwendungsfälle mit einem nicht eingeschränkten Budget sicherlich empfehlenswert, für den Einsatz in der Hochschule jedoch besteht die Anforderung, ein Open Source Tool zu verwenden.

Kubewatch wurde von Bitnami Labs für die Überwachung von Kubernetes entwickelt und nicht für den Einsatz in verschiedenen Applikationen. Daher verfügt Kubewatch nicht über die Möglichkeit, eigene Metriken für verschiedene Applikation zu entwickeln und wird aus diesem Grund hier nicht genauer betrachtet (vgl. [Lab20]). Jaeger ist ein Tracing basiertes Überwachungstool und bietet keine Möglichkeit Metriken zu überwachen (vgl. [Jea20]). Zusätzlich verfügt es über keinen eigenen Alertmanager. Oft wird Jaeger als Ergänzung zu einem bestehenden Monitoringsystem eingesetzt. Auch dieses Tool ist also für die beschriebene Problematik nicht empfehlenswert.

Aus der ersten Analyse stellten sich Graphite, InfluxDB und Prometheus als Favoriten heraus. Alle drei Tools verfügen über ein Alerting (vgl. [Dav20], [Inf20i], [Pro20b]), können für unterschiedliche Applikationen Metriken erstellen (vgl. [Dav20], [Inf20c], [Pro20c]), zur Überwachung der Kubernetes Prozesse eingesetzt werden (vgl. [Ahu20], [Inf20g], [Kub20n]) und sind darüber hinaus Open Source verfügbar (vgl. [Gra20], [Inf20d], [Pro20e]). Diese Tools wurden nun mit Hilfe einer Nutzwertanalyse genauer miteinander verglichen, um

das Tool zu identifizieren, das am besten auf die Anforderungen in der ICC-Cloud zugeschnitten ist.

3.3.3 Nutzwertanalyse

Mit Hilfe einer Nutzwertanalyse wird aus den drei Favoriten, die im Kapitel 3.3.2 ermittelt wurden, das Tools mit der größten Einsatzfähigkeit eruiert werden.

Sie unterteilt sich in die vier Teilbereiche *Alerting* 3.3.3, *Überwachen des Kubernetes System* 3.3.3, *Überwachung eines Namespaces im Kubernetes Cluster* 3.3.3 und *Überwachung eines Prozesses außerhalb des Kubernetes System* 3.3.3.

Alerting

In diesem Abschnitt werden die Fähigkeiten der drei Tools in Bezug auf den Bereich Alerting verglichen. Dabei werden die Möglichkeiten, eine Nachricht über E-Mail oder über die Messenger-Dienste MS Teams, Slack und Telegram zu senden, genauer betrachtet. Da eine Benachrichtigung über die Hochschulmail alle Usergruppen einschließt, wird dieser Weg als der Wichtigste bewertet. Für diese Möglichkeit werden zwei Punkte für das Tool vergeben, ein Alerting über die o.g. Messenger-Dienste hingegen wird jeweils nur mit einem Punkt bewertet. In der Gesamtevaluation wird diese Punktzahl verdoppelt, da Alerting für alle Usergruppen besonders relevant ist und eine funktionierende Kommunikation eine essenzielle Grundlage für Fehlerbehebungen ist.

Alle drei Tools schließen eine Benachrichtung über E-Mail ein. InfluxDB und Prometheus schnitten im Weiteren bei diesem Vergleich am Besten ab, sie erhalten die volle Punktzahl (fünf), da sie die Möglichkeit bieten, über alle genannten Kanäle, eine Nachricht zu senden (vgl. [Inf20e], [Inf20h], [Pro20d], [Git20q], [Kna20], [Mat20]). Graphite hingegen bietet nicht die Möglichkeit über MS Teams eine Nachricht zu senden, und erhält deswegen nur vier Punkte (vgl. [Kle20]) (s. Tabelle 3.2).

Tool	Gesamt Punkte	E-Mail	Messenger-Dienst		
			MS Teams	Slack	Telegram
Graphite	4	2	0	1	1
InfluxDB	5	2	1	1	1
Prometheus	5	2	1	1	1

Tabelle 3.2: Alerting

Überwachung des Kubernetes System

Dieser Abschnitt beschäftigt sich mit der Überwachung des Kubernetes System. Dabei wird zwischen zwei Bereichen unterschieden, erstens das Überwachen der Auslastung wie CPU oder Speicher, und zweitens in der Überwachung der Kubernetes eigenen Prozessen wie kube-controller-manager oder kubelet.

In diesem Vergleich schneiden alle Tools gleich gut ab. Alle Tools bieten die Möglichkeit, die von Kubernetes bereitgestellten Metriken auszulesen. InfluxDB verfügt über ein Plug-in, mit Hilfe dessen Kubernetes überwacht werden kann (vgl. [Nel20]). Für Graphite existiert ein konfiguriertes Deployment für das Kubernetes Cluster. Mit diesem können die Kubernetes Metriken ausgelesen werden (vgl. [Ahu20]). Prometheus wurde speziell für Kubernetes entwickelt und verfügt daher über die Möglichkeit die, von Kubernetes bereitgestellten Metriken, auszuwerten (vgl. [Kub20n]). Alle drei Tools erhalten je zwei Punkte (s. Tabelle 3.3).

Tool	**Gesamt-Punkte**	Kubernetes Prozesse	Kubernetes Auslastung
Graphite	2	1	1
InfluxDB	2	1	1
Prometheus	2	1	1

Tabelle 3.3: Kubernetes System Überwachung

Überwachung eines Namespaces im Kubernetes Cluster

Für die Überwachung einer eigenen Applikation in einem Kubernetes Namespace ist es einerseits wichtig, dass es für viele unterschiedliche Programmiersprachen eine Client-Bibliothek gibt und andererseits auch Metriken für die gängigen Webserver Systemumgebungen vorhanden sind.

Client-Bibliotheken

Um das Überwachungstool auf die Bedürfnisse der Usergruppen der HAW Hamburg auszurichten, wurde bei der Punkteverteilung verstärkt darauf geachtet, die unter Kapitel 3.2.3 erläuterten Programmiersprachen zu bevorzugen. Für jede verfügbare Client-Bibliothek in den genannten Sprachen gab es einen Punkt. Um Tools mit einem breitem Spektrum an Programmiersprachen nicht zu benachteiligen, wurde jede weitere Programmiersprache mit 0,25 Punkt bewertet.

Bei diesem Vergleich schneidet Prometheus mit neun Punkten am besten ab, es besitzt für jede unter Kapitel 3.2.3 erläuterte Programmiersprache eine Client-Bibliothek, zusätzlich werden noch zwölf andere Programmiersprachen mit einer Client-Bibliothek unterstützt. Es handelt sich hierbei um Bash, Common Lisp, Dart, Elixir, Erlang, Haskell, .NET / C#, Node.js, Perl, PHP, R und Rust (vgl. [Pro20c]). Auf Platz zwei befindet sich InfluxDB mit 4,75 Punkten, es unterstützt die, für die Usergruppen relevanten, Programmiersprachen Java, Golang, Python und Ruby, zusätzlich werden auch die Programmiersprachen C#, JavaScript und PHP unterstützt (vgl. [Inf20c]). Am schlechtesten mit 4,25 Punkten schneidet Graphite ab, es unterstützt nur die Programmiersprachen Java, Golang, Python, C und JavaScript (vgl. [Dav20]) (s. Tabelle 3.4).

Tool	Gesamt-Punkte	Client-Bibliotheken						
		Java	Golang	Python	Ruby	C	C++	Sonstige Sprachen
Graphite	4,25	1	1	1	0	1	0	0,25
InfluxDB	4,75	1	1	1	1	0	0	0,75
Prometheus	9	1	1	1	1	1	1	3

Tabelle 3.4: Client Bibliotheken

Systemumgebung

In diesem Abschnitt wurde bewertet, inwiefern das Tool in der Lage ist, die beiden Webserver NGINX oder Apache zu überwachen. Es gibt je einen Punkt, wenn ein Tool für die jeweiligen Webserver eine Überwachungslösung anbietet. Da diese Webserver als gleichwertig zu betrachten sind, wird hier in der Bewertung kein Unterschied gemacht.

Im Vergleich gewinnen die Tools InfluxDB und Prometheus mit je zwei Punkten. InfluxDB bietet für die Webserver Apache (vgl. [Inf20b]) und für NGINX (vgl. [Inf20j]) jeweils ein Plug-in an. Für Prometheus existieren für unterschiedliche Systemumgebungen Exporter für die Überwachung auf NGINX und Apache Webserver (vgl. [Pro20f]). Graphite hingegen erhält nur einen Punkt für die Möglichkeit, einen NGINX Webserver über ein Modul zu überwachen (vgl. [Git20a]) (s. Tabelle 3.5).

Tool	Gesamt-Punkte	Systemumgebung	
		Nginx	Apache
Graphite	1	1	0
InfluxDB	2	1	1
Prometheus	2	1	1

Tabelle 3.5: Systemumgebung

Gesamtbewertung für die Überwachung eines Namespaces

In der Gesamtbewertung für die Überwachung eines Namespaces im Kubernetes Cluster schneidet Prometheus mit elf Punkten am besten ab. Danach folgt mit 6,75 Punkten InfluxDB. Graphite mit 5,25 Punkten bildet den Schluss (s. Tabelle 3.6).

Tool	**Gesamt-Punkte**	Client Bibliotheken	Systemumgebung
Graphite	5,25	4,25	1
InfluxDB	6,75	4,75	2
Prometheus	11	9	2

Tabelle 3.6: Überwachung einer Applikation im Namespace von Kubernetes

Überwachung eines Prozesses außerhalb des Kubernetes System

In diesem Abschnitt wird untersucht wie gut sich die Überwachungstools für den Einsatz der Überwachung der unter Kapitel 3.2.4 beschrieben Prozessen und Systeme einsetzen lässt. Hierbei wird für jede Prozessgruppe, die überwacht werden kann, ein Punkt vergeben.

Prometheus schneidet in diesem Vergleich mit fünf Punkten am besten ab. Es bietet verschiedene Exporter, mit Hilfe derer man Komponenten über das Simple Network Management Protokoll, IPMI Sensoren, Serversysteme (vgl. [Pro20f]) und GitLab (vgl. [Git20s]) überwachen kann. InfluxDB erhält vier Punkte, es verfügt über Plug-ins, mit Hilfe derer man die unterschiedlichen Prozesse überwachen kann. Es können alle aus Kapitel 3.2.4 beschriebenen Prozesse (außer GitLab) überwacht werden (vgl. [Inf20k] [Git20c]) [Inf20a] [Inf20f]). Graphite hingegen bekommt nur drei Punkte, da hier sowohl die IPMI Sensoren, als auch GitLab nicht überwacht werden können (vgl. [Jen20] [Par20] [Dav20])(s. Tabelle 3.7).

		Externe Prozesse				
Tool	**Gesamt-Punkte**	SNMP	IPMI Sensor	Server system	HA-Proxy Server	Gitlab
Graphite	3	1	0	1	1	0
InfluxDB	4	1	1	1	1	0
Prometheus	5	1	1	1	1	1

Tabelle 3.7: Überwachung von externen Prozesse

Gesamtergebnis

Das Gesamtergebnis setzt sich aus den Teilergebnissen der vorherigen Abschnitte zusammen. Eine gesonderte Rolle erhält der Abschnitt Alerting, dieses Ergebnis wird, wie oben erläutert, doppelt gewertet, da es für alle Usergruppen besonders relevant ist. Alle anderen Teilergebnisse werden einfach gewertet.

Gemäß dem erarbeiteten Vergleich weist das Tool Prometheus die besten Voraussetzungen auf, um für den definierten Anwendungsfall die besten Ergebnisse zu liefern. Mit einer finalen Punktzahl von 28 Punkten erweist sich das Tool als äußerst leistungsfähig in allen Kategorien, und schneidet auch in den Teilbereichen am besten ab. InfluxDB schneidet mit einer Punktzahl von 22,75 Punkten am zweitbesten ab, Graphite hingegen erreicht nur eine Gesamtpunktzahl von 18,25 Punkten. Gerade in der Vielfalt der einzubindenden Programmiersprachen und einzubindenden Applikationen (wie z.B. C++ als Programmiersprache oder GitLab als Applikation) bietet Prometheus hier entscheidende Vorteile (s. Tabelle 3.8).

Tool	Gesamt-Punkte	Alerting (Doppelte Wertung)	Überwachung des Kubernetes System	Überwachung eines Namespaces	Überwachung eines Prozesses außerhalb des System
Graphite	18,25	8	2	5,25	3
InfluxDB	22,75	10	2	6,75	4
Prometheus	28	10	2	11	5

Tabelle 3.8: Nutzwertanalyse Ergebnis

Aufgrund des Ergebnisses der Nutzwertanalyse konzentriert sich die nachfolgende Arbeit des Entwurfs und der Implementierung einer Überwachungsstrategie für die ICC der HAW Hamburg auf die Umsetzung mit Prometheus. Da dieses Tool allen definierten Anforderungen entspricht, kann davon ausgegangen werden, dass die Implementierung Erfolg versprechend ist.

4 Entwurf

Aus Kapitel 3.3.3 Nutzwertanalyse ergab sich eine hohe Eignung des Monitoring Tools Prometheus für den Einsatz in der ICC. Im folgenden Abschnitt steht nun die Ausarbeitung einer geeigneten Architektur für diesen Einsatz im Vordergrund.

Es existieren unterschiedliche Möglichkeiten, Prometheus als Monitoring Tool in ein Kubernetes-Cluster zu integrieren. Es ist grundsätzlich zu unterscheiden zwischen den Varianten, Prometheus außerhalb oder innerhalb eines Clusters zu starten (vgl. [Pro20g]). Die besondere Herausforderung besteht darin, dass der Konfigurationsaufwand des Überwachungstools möglichst gering gehalten werden soll. Aus diesem Grund wird für den Einsatz von Prometheus in der ICC eine Architektur auf Basis des Prometheus-Operator (ein weiteres Open-Source Projekt auf GitHub) entwickelt.

4.1 Prometheus

Prometheus ist ein Open-Source-Projekt, das die "De-factor-Standard-Lösung"(S.326 [AD19]) für das Sammeln von Metriken in einem Kubernetes-Cluster bildet. Dabei handelt es sich um ein Tool, das Metriken sammelt und in einer Zeitreihen-Datenbank speichert. Für den Zugriff auf die gesammelten Daten verfügt Prometheus über eine eigene Abfragesprache (PromQL). Für das Sammeln von Metriken erwartet Prometheus eine metrics-API von den jeweiligen Containern im Cluster. (vgl. S.35 [Bur18])

Für die Integration von Prometheus in ein Kubernetes-Cluster existiert für den Prometheus-Server ein Docker Container auf Docker Hub bzw. Quay.io. Mit Hilfe des Containers und einer Konfigurationsdatei im YAML-Format lässt sich ein Prometheus-Server im Cluster starten. (vgl. [Pro20g])

Das Versenden von Alerts übernimmt bei Prometheus ein zusätzlicher Service, der sogenannte Prometheus-Alertmanager. Für diesen existieren auf Docker Hub bzw. Quay.io

je ein Docker Container für die Integration in einem Kubernetes-Cluster. Die Konfiguration des Prometheus-Alertmanagers erfolgt analog zum Prometheus-Server über eine YAML-Datei.

4.2 Prometheus-Operator

"The Prometheus Operator provides Kubernetes native deployment and management of Prometheus and related monitoring components. The purpose of this project is to simplify and automate the configuration of a Prometheus based monitoring stack for Kubernetes clusters."[Git20i]

Der Prometheus-Operator ermöglicht es, mit geringem Konfigurationsaufwand beliebig viele Prometheus-Server, im Weiteren als Prometheus-Instanzen bezeichnet, in Kubernetes Namespaces zu starten. Dabei erweitert Prometheus-Operator mit Hilfe von CustomResourceDefinitions (vgl. [Kub20c]) die Kubernetes API um z.B. den ServiceMonitor. Dieser ist bspw. für die einfache Konfiguration der Metriken zuständig, d.h. automatisiert den Prozess der Erstellung von Konfigurationsdateien für die einzelnen Metriken. Das Erstellen von großen Konfigurationsdateien für eine Prometheus-Instanz entfällt somit vollständig.

Zusätzliche in diesem Entwurf verwendeten CustomResourceDefinitions sind Prometheus, PrometheusRule, Alertmanager, AlertmanagerConfig, PodMonitor, Probes und ThanosRuler. Diese werden in Kapitel 5.2.3 erläutert.

4.3 Architektur

Die Abbildung 4.1 zeigt die Architektur in vereinfachter Form und stellt die unterschiedlichen Komponenten vor. Diese Komponenten sind in Namespaces gruppiert und werden nachfolgend erläutert. Die Farbgebung der Namespaces zeigt die für diesen Bereich relevante Usergruppe an.

Der Namespace *Monitoring* ist das Grundgerüst der Architektur. In diesem läuft der Prometheus-Operator, der die Prometheus-Instanzen konfiguriert, der Prometheus-Alertmanager, der für das Versenden der Benachrichtigungen zuständig ist, und eine Prometheus-Instanz, die diese Services im Namespace *Monitoring* überwacht. Der Namespace *Kubernetes-*

Monitoring managt die Überwachung des Kubernetes-Systems, das sogenannte Cluster-Monitoring. Innerhalb dieses Namespaces werden alle für das System relevanten Metriken gesammelt und mit Hilfe von Grafana visualisiert. Grafana ist ein Open-Source Visualisierungstool, das aus den gesammelten Daten einer Prometheus-Instanz, Dashboards erstellt. Beide Namespaces *Monitoring* und *Kubernetes-Monitoring* sind nur für die Usergruppe ICC-Admins sichtbar. Für die Überwachung einer Applikation im Cluster steht der Namespace *Example-Application*, dieser zeigt beispielhaft an, wie eine Userin bzw. ein User oder eine Usergruppe eine Applikation mit Hilfe einer Prometheus-Instanz überwachen kann.

Wie erwähnt ist für die Visualisierung der erfassten Metriken der einzelnen Prometheus-Instanzen Grafana als Docker-Container zuständig.

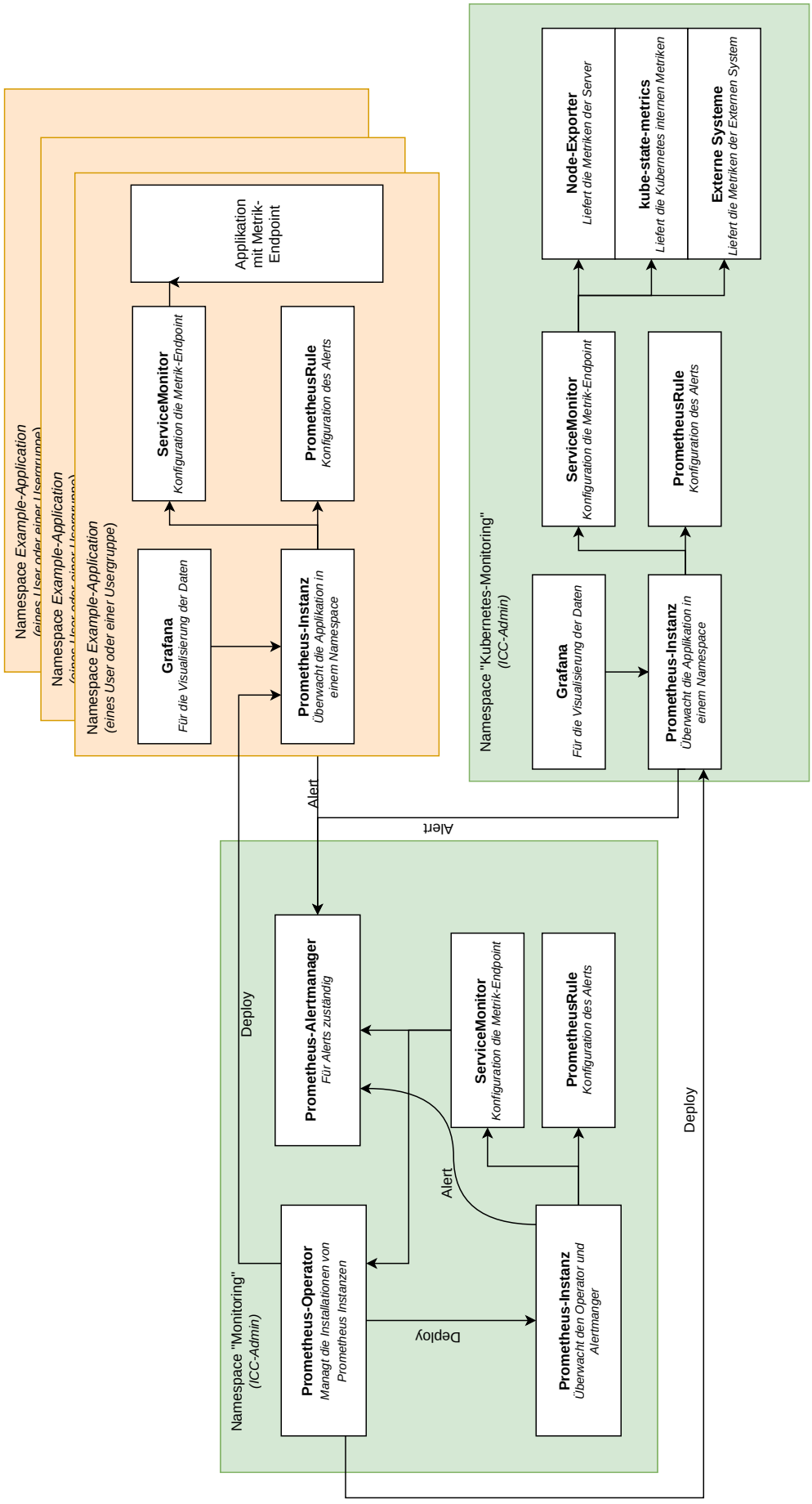

Abbildung 4.1: Entwurf (vgl. [Bai20])

4.3.1 Namespace Monitoring

Die Abbildung 4.2 zeigt vergrößert die Architektur des Namespaces *Monitoring*. Dieser Bereich der Architektur bildet das Grundgerüst des Systems. Der Teil bzw. der Namespace ist nur für einen ICC-Admin sichtbar, bzw. die Ressourcen können nur als ICC-Admin konfiguriert werden. Innerhalb dieses Namespaces wird der Prometheus-Operator gestartet. Dieser ist für die Konfiguration der Prometheus-Instanzen bzw. des Prometheus-Alertmanagers zuständig. Der Prometheus-Operator erweitert die Kubernetes API, mit Hilfe von CustomResourceDefinitions (CRD) (vgl. [Kub20c]) um die Syntax Prometheus, PrometheusRule, Alertmanager, ServiceMonitor, AlertmanagerConfig, PodMonitor, Probes und ThanosRuler. Mit Hilfe dieser Erweiterungen ist es möglich mit der Kubernetes Syntax (YAML-Datei) Objekte zu erstellen, die der Prometheus-Operator, für das Deployment einer Prometheus-Instanz (PrometehusCRD) bzw. eines Prometheus-Alertmanagers (AlertmanagerCRD), benötigt. Mit PrometheusRulesCRD bzw. ServiceMonitorCRD werden Objekte, mit Hilfe der Kubernetes Syntax, (YAML-Datei) erstellt, die die Konfiguration der Alerts bzw. der Metrik beinhalten.

Im Namespace *Monitoring* läuft, neben dem Prometheus-Operator, der Prometheus-Alertmanager, der von allen Prometheus-Instanzen einzelne Alerts erhält und diese an die jeweiligen Kanäle verteilt. Für die Überwachung dieser zwei Services läuft zusätzlich im Namespace *Monitoring* eine Prometheus-Instanz. Diese wird mit Hilfe eines ServiceMonitor-Objekts zur Konfiguration der zu überwachenden Metriken, bzw. zu einem PrometheusRule-Objekt, das die Alerts konfiguriert.

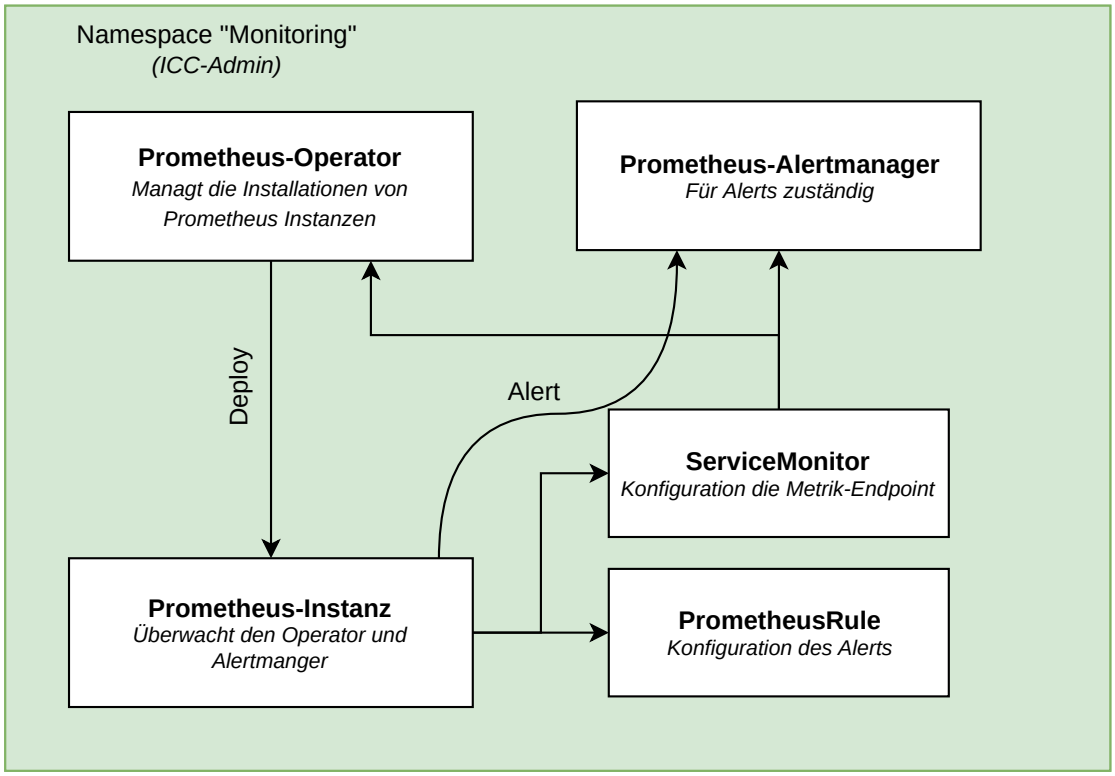

Abbildung 4.2: Namespace Monitoring (vgl. [Bai20])

4.3.2 Namespace Kubernetes-Monitoring

Der Namespace *Kubernetes-Monitoring* dient dem sogenannten Cluster-Monitoring. Hierbei soll das Kubernetes-System, sowie die wichtigen externen Systeme, überwacht werden. Innerhalb des Namespaces, (s. Abb. 4.3), läuft eine Prometheus-Instanz, die mit Hilfe des Prometheus-Operator deployt wird. Um die verschiedenen Metriken einzusammeln, bzw. in das für Prometheus benötigte Format zu bringen, fungieren verschiedene Services.

Der Node-Exporter ist ein auf Github Open Source entwickelter Service für Prometheus (vgl. [Git20g]). Er sammelt für alle Server, auf denen das Kubernetes Cluster läuft, die jeweiligen Ressource-Metriken ein. Der Service Kube-State-Metrics ist auch ein auf Github entwickelter Service für Prometheus bzw. Kubernetes, und sammelt alle Metriken des Kubernetes-Systems ein.

Das Projekt wird beschrieben als "kube-state-metrics [...] that listens to the Kubernetes API server and generates metrics about the state of the objects.[...] It is not focused on

the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods"[Git20f].

Die externen Systeme bzw. Prozesse aus Kapitel 3.2.4, die überwacht werden sollen, können mit denen, in der Analyse 3.3.3 bereits erwähnten, Methoden und Projekten überwacht werden. In der Abbildung sind sie als *Externe Systeme* zusammengefasst.

Der ServiceMonitor und die PrometheusRule sind zwei Objekte, die mit Hilfe der, von Prometheus Operator hinzugefügten, CRD's zu einer Kubernetes API erstellt werden können. Sie sind für die Konfiguration der Metriken bzw. der Alerts zuständig.

Um die gesammelten Daten anzeigen zu können, wird, wie bereits erläutert, auch hier Grafana verwenden. Grafana wird im Namespace deployed und visualisiert die Daten, die Prometheus sammelt.

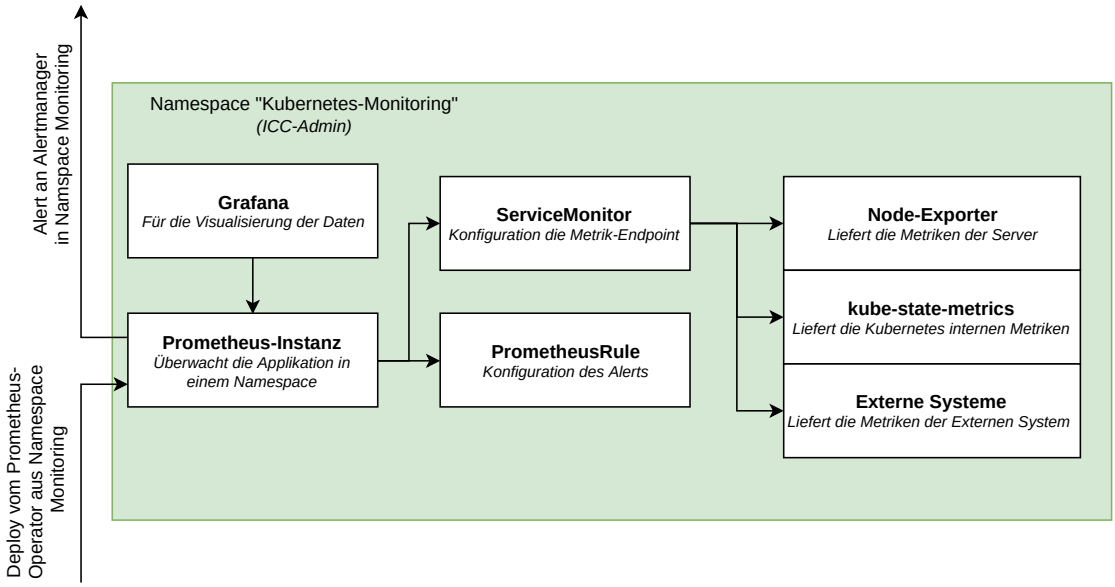

Abbildung 4.3: Namespace Kubernetes-Monitoring (vgl. [Bai20])

4.3.3 Namespace Example-Application

Die Abbildung 4.4 zeigt einen beispielhaften Namespace, den Namespace *Example-Application* einer Userin bzw. eines Users oder einer Usergruppe. In diesem Namespace

läuft eine Applikation, die überwacht werden soll. Diese Applikation muss die Metriken in dem, für Prometheus lesbaren Format, an einem Endpoint abrufbar machen. Standardmäßig sind die Metriken unter {URL/IP}\metrics abrufbar.

Die Überwachung der Applikation läuft simultan zu Überwachung im Namespace *Kubernetes-Monitoring* aus Kapitel 4.3.2. Die Prometheus-Instanz, die vom Prometheus-Operator konfiguriert wird, sammelt alle Metriken ein und managt die Alerts. Die Konfiguration der Metriken bzw. der Alerts übernehmen die Objekte ServiceMonitor und PrometheusRule. Für die Visualisierung der Daten ist Grafana zuständig.

Die Herausforderung dieses Bestandteils der Architektur gilt der Vergabe der Rechte, da eine Userin bzw. ein User oder eine Usergruppe nur begrenzte Rechte speziell für das Erstellen und Managen ihrer Applikation innerhalb ihres Namespaces haben. Für das Erstellen z.B. des ServiceMonitors, benötigen sie aber zusätzlich Zugriff auf die CustomResourceDefinition des Prometheus-Operator aus den Namespace *Monitoring*. Dieser Zugriff muss den Userinnen bzw. dem User und den Usergruppen zuerst vom ICC-Admin gewährt werden.

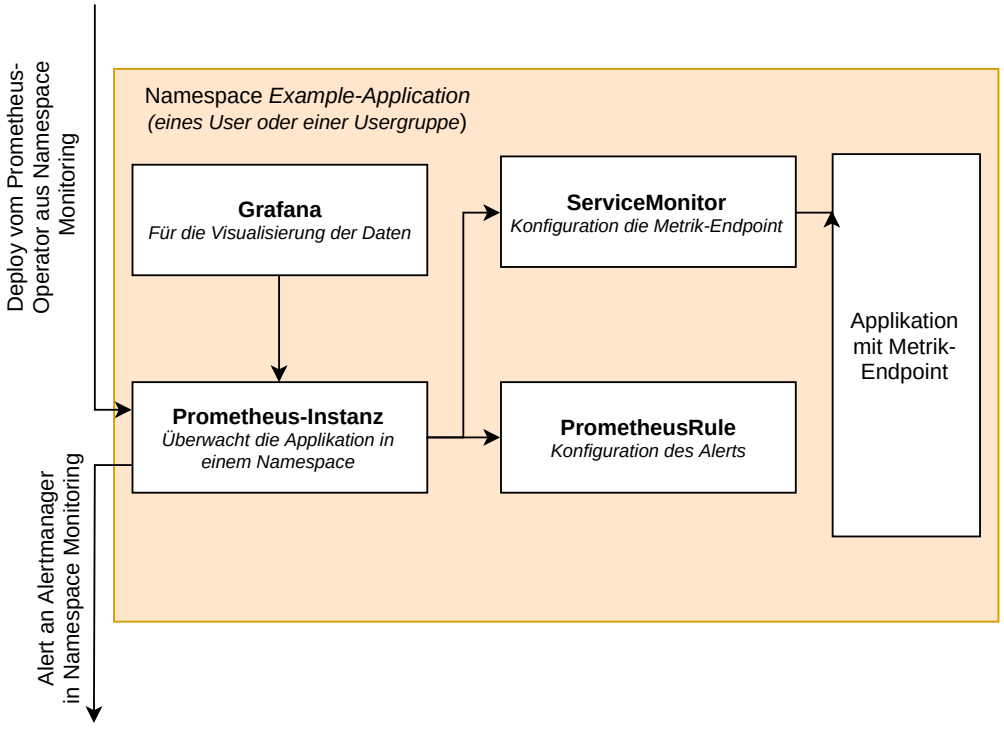

Abbildung 4.4: Namespace Example-Application (vgl. [Bai20])

5 Implementierung

Im Zuge des Kapitels 4 Entwurf wurde die Architektur des Prometheus-Operator für den Einsatz in der ICC erläutert. Im nächsten Schritt wird nun die Implementierung des Prometheus-Operator in das System beschrieben. Anhand eines Beispiels wird die Überwachung innerhalb eines Namespace mit dem Prometheus-Operator aufgezeigt. Alle für die Implementierung erforderlichen YAML-Dateien sind der Arbeit beigefügt, dadurch können die nachfolgenden Arbeitsschritte begleitend nachvollzogen werden.

5.1 Bestehendes System

Die in dieser Arbeit entwickelte Monitoring Strategie mit Anwendung des Prometheus-Operator, wurde für die neu aufgesetzte Informatik Compute Cloud (ICC) der HAW-Hamburg entwickelt. Diese soll im Frühjahr 2021 veröffentlicht werden.

"Die Informatik Compute Cloud ist eine vom AI Labor zur Verfügung gestellte Container Cloud Umgebung, in der Mitglieder des Departments Applikationen in Form von Docker Containern betreiben können."[Use20]

Zum Testen des in dieser Arbeit entwickelten Systems wurde zunächst ein lokales Kubernetes-Cluster, mit Hilfe des Kommandozeilen-Tools *Kind* erstellt und verwendet. Zur Verifizierung der Tests in der realen Umgebung wurde das existierende Test-Cluster, das die neue Compute Cloud abbilden soll, verwendet. Dieses wurde von der ICC-Gruppe des AI-Labors der HAW Hamburg bereitgestellt.

5.2 Integration des Prometheus-Operators

Wie schon in Kapitel 4.3.1 beschrieben, ist der Namespace *Monitoring* das Herzstück der Monitoring Strategie. In diesem Namespace wird der Prometheus-Operator und der

Alertmanager deployed. Für die Integration des Prometheus-Operator innerhalb des Kubernetes-Cluster gibt es grundsätzlich unterschiedliche Möglichkeiten. Im Rahmen dieser Arbeit hat man sich dazu entschieden, den Prometheus-Operator mit Hilfe der auf Github verfügbaren Beispiel-Deployments zu integrieren. Eine weitere Möglichkeit ist es, den Prometheus-Operator mit Hilfe des Paketmanagers Helm zu installieren (vgl. [Git20b]). Man hat sich im Zuge dieser Arbeit für das klassische Deployment entschieden, um die Funktionsweise des Prometheus-Operator zu verstehen. Ein Deployment über den Paketmanager Helm hat zwar den Vorteil, dass die Installation nur wenige Schritte benötigt, jedoch kann als entscheidend nachteilig gewertet werden, dass das System um den Paketmanager erweitert werden muss. Somit würde sich die Komplexität des Gesamtsystems entschieden vergrößern, was schlussendlich gegen diese Variante sprach.

Für den Einsatz des Prometheus-Operator im Rahmen der in Kapitel 4 erläuterten Architektur muss das Kubernetes-Cluster an verschiedenen Punkten angepasst werden. Für den Prometheus-Operator muss die Kubernetes API um verschiedene Ressourcen mit Hilfe von CustomResourceDefinitions erweitert werden. Diese Konfigurationen werden detailliert in Kapitel 5.2.3 erläutert. Damit der Prometheus-Operator im gesamten Cluster funktioniert, benötigt er zusätzlich bestimmte Berechtigungen, sogenannte ClusterRole, diese werden in Kapitel 5.2.2 beschrieben.

Die Architektur sieht vor, den Prometheus-Operator im Namespace *Monitoring* zu integrieren. Das Beispiel Deployment wurde dahingehend mit dem Zusatz des Namespace erweitert. Im nachfolgenden Kapitel werden die wesentlichen Punkte des Deployments aufgezeigt.

5.2.1 Deployment des Prometheus-Operators

Für das Deployment des Prometheus-Operator hat man sich, wie bereits erläutert, für die auf Github zu Verfügung gestelltem Beispiel-Depoloyments in einem Kubernetes-Cluster mit einer RBAC Authentifikation entschieden. Diese Entscheidung wurde auch aus dem Grund getroffen, dass der Prometheus-Operator sich zum Zeitpunkt dieser Arbeit noch im Beta-Status befindet, und daher mit jedem neuen Release um weitere Funktionen erweitert wird. Um ein Update des Prometheus-Operator zu erleichtern, wurde daher das Beispiel-Deployment nur an wenigen Punkten verändert.

Das Deployment des Prometheus-Operator besteht aus einem *Service* (Quellcode A.2) und einem *Deployment* (Quellcode A.1), das dem Kubernetes Cluster hinzugefügt wird (vollständige Quellcode-Dateien s. Anhang).

```
2 kind: Deployment
3 metadata:
4   labels:
5     app.kubernetes.io/component: controller
6     app.kubernetes.io/name: prometheus-operator
7     app.kubernetes.io/version: v0.42.1
8   name: prometheus-operator
9   namespace: monitoring
```

Quellcode 5.1: Ausschnitt des Prometheus-Operators Deployment (vgl. [Git20o])

Der Quellcode 5.1 zeigt einen Ausschnitt des *Deployments*(Quellcode A.1). Das Deployment definiert den Zustand der Applikation des Prometheus-Operator für das Kubernetes-Cluster. Zusätzlich zum Beispiel-*Deployment* von Github wurde Zeile neun *namespace: monitoring* hinzugefügt. Diese weist dem *Deployment* die Definition des Namespace hinzu, d.h. die Applikation Prometheus-Operator wird im Namespace *monitoring* integriert.

```
2 kind: Service
3 metadata:
4   labels:
5     app.kubernetes.io/component: controller
6     app.kubernetes.io/name: prometheus-operator
7     app.kubernetes.io/version: v0.42.1
8   name: prometheus-operator
9   namespace: monitoring
```

Quellcode 5.2: Ausschnitt des Prometheus-Operators Service (vgl. [Git20o])

Der *Service* (Quellcode A.2) definiert die Kommunikation der Applikation. Da der Container, in dem die Applikation läuft, bei jedem Neustart eine andere IP-Adresse zugewiesen bekommt, ist die Kommunikation über diese nicht zu empfehlen. Mit Hilfe des *Service*, der die genaue Adresse der Applikation (IP Adresse: Port) kennt, kann die Applikation mit Hilfe des Kubernetes Labeling erreicht werden.

5.2.2 Berechtigungen des Prometheus-Operators

Damit der Prometheus-Operator Prometheus-Instanzen korrekt für alle Nutzende des Clusters konfigurieren kann, muss er über bestimmte Berechtigungen verfügen. In Kubernetes

werden Berechtigungen mit Hilfe von *Role* oder *ClusterRole*, die an einen *ServiceAccount* mit Hilfe eines *RoleBinding* bzw. *ClusterRoleBinding* gebunden werden, geregelt.

Für den Prometheus-Operator wird ein ServiceAccount mit dem Namen *prometheus-operator* konfiguriert (Quellcode A.3). An diesen ServiceAccount wird mit einem ClusterRoleBinding (Quellcode A.3) eine ClusterRole (Quellcode A.4) gebunden. Mit Hilfe dieser ClusterRole erhält der ServiceAccount alle benötigten Rechte um im Cluster laufen zu können.

5.2.3 CustomResourceDefinitions

Damit der Prometheus-Operator korrekt im Kubernetes Cluster eingesetzt werden kann, müssen der Kubernetes API mit Hilfe von CustomResourceDefinitions (CRDs) neue Ressourcen hinzugefügt werden. Die Abbildung 5.1 soll die neuen Ressourcen veranschaulichen. Hinzugefügt werden die, zum Zeitpunkt dieser Erarbeitung aktuellen, vom Github Projekt Prometheus-Operator verfügbaren CRDs (vgl. [Git20l]). Im zu bezeichnenden Fall sind dies die CRDs: Prometheus, PrometheusRule, Alertmanager, ServiceMonitor, AlertmanagerConfig, PodMonitor, Probes und ThanosRuler.

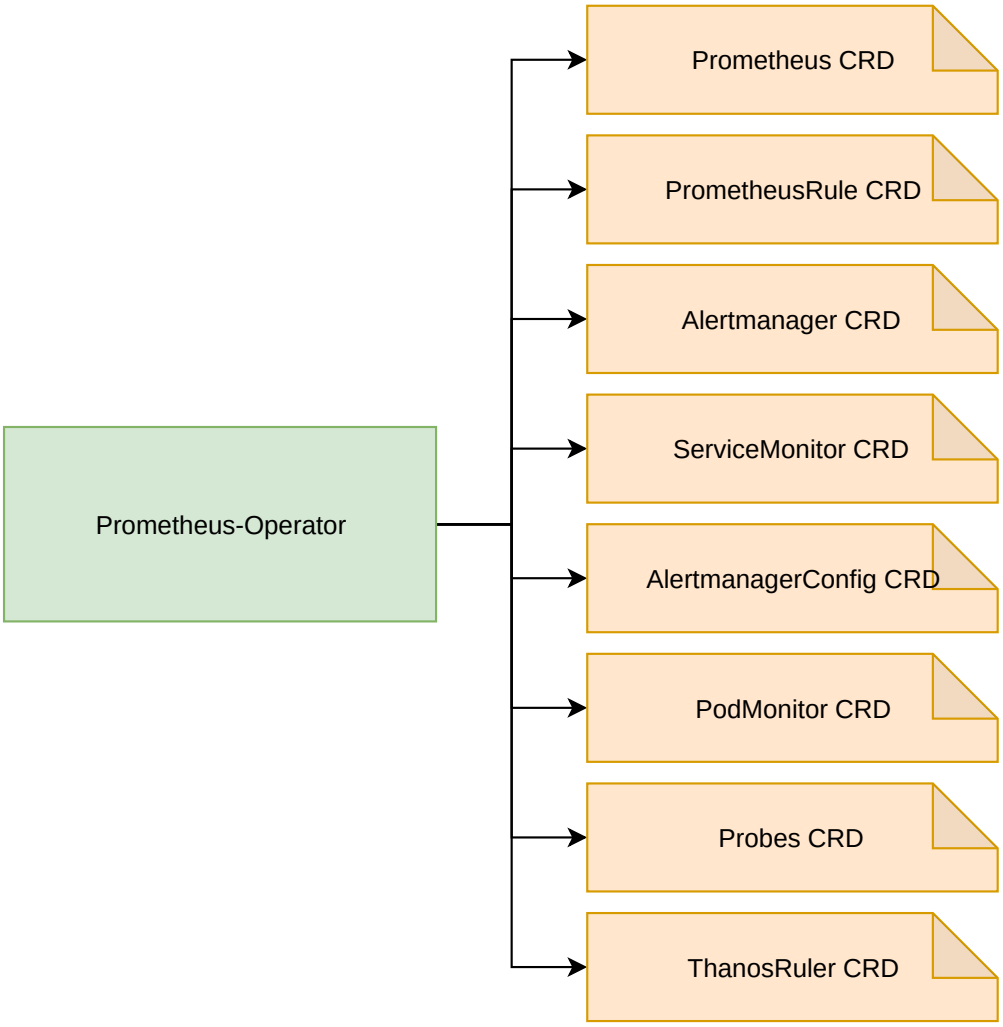

Abbildung 5.1: Prometheus-Operator CRD

Prometheus: Damit lässt sich ein Prometheus Objekt im Namespace definieren, mit Hilfe dessen der Prometheus-Operator das Deployment für eine Prometheus-Instanz erstellt.

Alertmanager: Damit lässt sich ein Alertmanager Objekt im Namespace definieren, mit Hilfe dessen der Prometheus-Operator das Deployment für eine Alertmanager erstellt.

ServiceMonitor: Damit lässt sich ein ServiceMonitor Objekt im Namespace definieren, der die Konfiguration der zu überwachenden Metriken auf Basis der Kubernetes Services darstellt.

PodMonitor: Damit lässt sich ein PodMonitor Objekt im Namespace definieren, das die Konfiguration der zu überwachenden Metriken auf Basis der Kubernetes Pod darstellt.

PrometheusRule: Damit lässt sich ein PrometheusRule Objekt im Namespace definieren, das die Konfiguration der Alerts darstellt.

AlertmanagerConfig: Stellt eine Unterkonfiguration des Alertmanagers für einzelne User*innen dar.

Probes: Damit lässt sich ein Probes Objekt im Namespace definieren, das eine Konfiguration zur Überwachung von statischen Zielen oder von einem Ingress erstellt.

ThanosRuler: Damit lässt sich ein ThanosRuler Objekt im Namespace definieren, mit Hilfe dessen der Prometheus-Operator das Deployment für eine ThanosRuler erstellt.

(vgl. [Git20j])

Für die in dieser Arbeit entstandenen Monitoring Strategie werden nicht alle CustomResourceDefinitions verwendet. Da man sich mit dem Deployment des Prometheus-Operator, wie schon in Kapitel 5.2.1 beschrieben, für das Standard-Deployment entschieden hat, benötigt der Prometheus-Operator für eine korrekte Funktion aller Tools, alle diese CustomResourceDefinitions. Zusätzlich ermöglicht dies später eine einfachere Erweiterung der Monitoring Strategie.

5.2.4 Nachteil der Variante

Ein Nachteil der hier vorgestellten Integration des Prometheus-Operator ist, dass für ein Update des Prometheus-Operators auf eine neue Version, alle Konfigurationsdateien angepasst werden müssen. Die in Kapitel 5.2.1 gezeigten Ausschnitte des Deployments bzw. des Services (Quellcode 5.1 und 5.2) zeigen, dass die Version des Prometheus-Operator fest in die Dateien geschrieben ist.

5.3 Benötigte Berechtigungen

Damit User*innen den Prometheus-Operator und die hinzugefügten Ressourcen nutzen können, benötigen sie bestimmte Berechtigungen, die nachfolgend erläutert werden. Da die

ICC die RBAC Methode von Kubernetes verwendet, kann mit Hilfe von ClusterRole der Userin bzw. dem User die benötigten Berechtigungen zur Verwendung der CRD Ressourcen hinzugefügt werden. Der Quellcode A.11 zeigt diese ClusterRole. Mit Hilfe dieser YAML Datei werden die zwei ClusterRole (*prometheus-crd-view* und *prometheus-crd-edit*) dem Cluster hinzugefügt.

***prometheus-crd-view*:** Erlaubt User*innen mit *view*-Rechten, in ihrem Namespace die CRD Alertmanager, Prometheus, PrometheusRule und ServiceMonitor einzusehen. (vgl. [Git20r])

***prometheus-crd-edit*:** Erlaubt User*innen mit *admin*- und *edit*-Rechten, in ihrem Namespace die CRD Alertmanager, Prometheus, PrometheusRule und ServiceMonitor zu erstellen, zu bearbeiten und einzusehen. (vgl. [Git20r])

Damit eine Prometheus-Instanz korrekt funktioniert, benötigt sie Berechtigungen, um auf Metriken zuzugreifen zu können oder Alerts an den Alertmanager zu senden. Diese Berechtigungen werden einer Prometheus-Instanz über einen ServiceAccount erteilt. Die Quellcodes A.12, A.13 und A.14 zeigen die ClusterRole, bzw. wie diese an einen ServiceAccount gebunden wird.

Für das Überwachen des Kubernetes-Clusters, benötigt eine Prometheus-Instanz weitreichendere Berechtigungen, wie zum Beispiel das Auslesen von den Ressourcen *pod* oder *service* aus anderen Namespaces. Der Quellcode A.15 stellt diese ClusterRole dar.

5.4 Alertmanager

Prometheus verwendet für das Versenden von Alerts den Prometheus-Alertmanager, im Weiteren wird dieser nur Alertmanager genannt. Wie in Kapitel 5.2.3 beschrieben, wurde der Kubernetes API die Ressource Alertmanager hinzugefügt. Resultierend daraus kann nun ein Alertmanager deployed werden. Die Abbildung 5.2 skizziert die Integration des Alertmanagers. Wie in Kapitel 4 herausgearbeitet, wird der Alertmanager im selben Namespace wie der Prometheus-Operator laufen. Für das Deployment und das Erstellen der richtigen Konfiguration ist der Prometheus-Operator zuständig. Mit Hilfe der Ressource Alertmanager kann das Objekt *Alertmanager* im Namespace erstellt werden. Über diese Objekte wird das Deployment des Alertmanager definiert, bspw. kann dadurch die Anzahl der Repliken konfiguriert werden. Die Konfiguration des Alertmanagers wird in einem

Kubernetes Secret Objekt gespeichert. Die Referenzierung auf das Secret erfolgt über die configSecret: Bereich des *Alertmanager* Objekts (s. Abbildung 5.2). Diese Abbildung skizziert das Versenden von Alerts in einen Microsoft Teams Kanal. Da der Alertmanager zum Zeitpunkt dieser Arbeit keine direkte Konfigurationsmöglichkeit bietet, um einen Empfänger für Microsoft Teams zu konfigurieren, erfolgt der Sendevorgang eines Alerts an einen Microsoft Teams Empfänger über einen Service, wie in der Abbildung im Pod MS-Teams dargestellt.

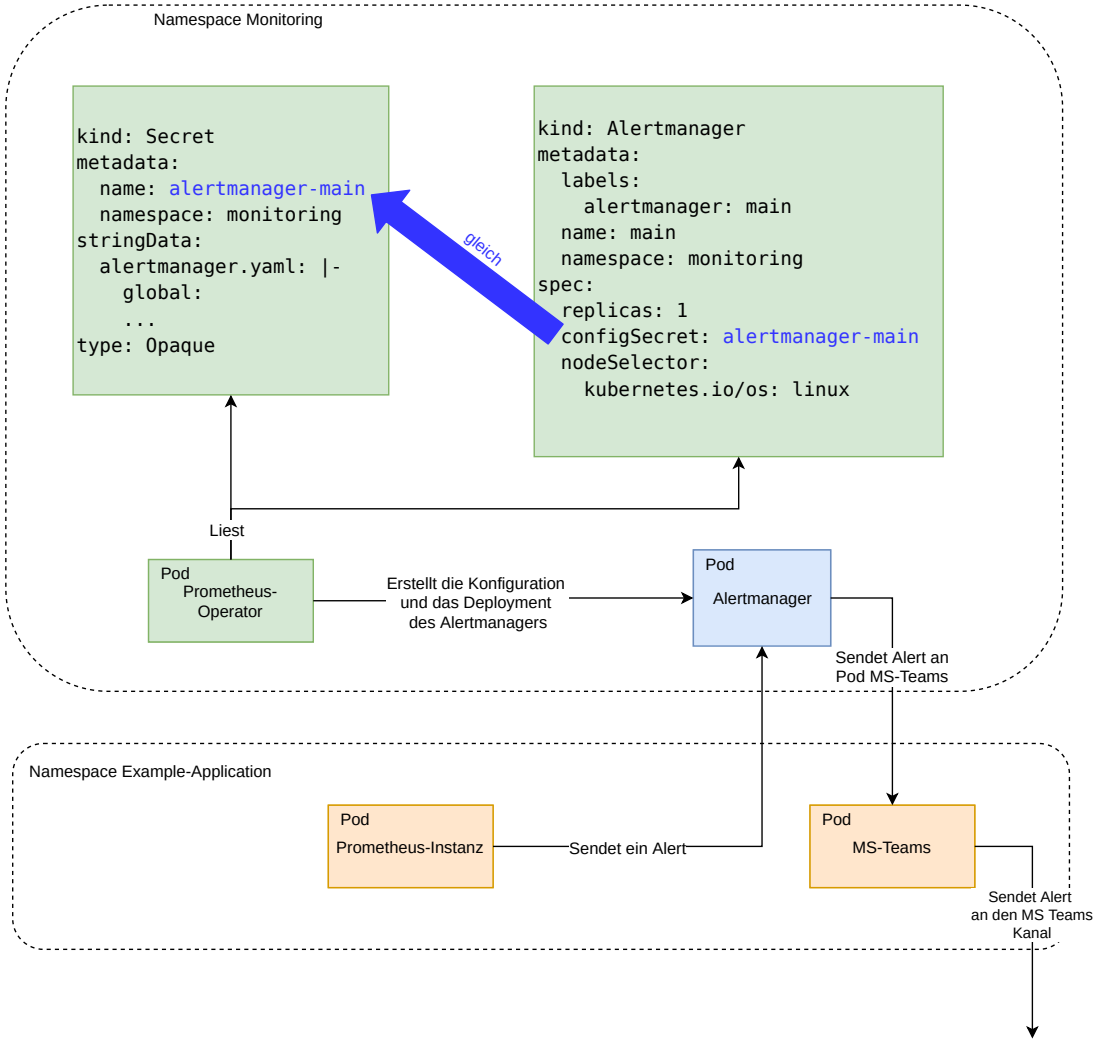

Abbildung 5.2: Der Alertmanager (vgl. [Git20k])

Das Erstellen des *Alertmanager*-Objekts erfolgt über der Quellcode A.6. Um den Alertmanager im Pod zu starten, benötigt er eine korrekte Konfiguration. Diese Konfiguration wird mit Hilfe eines Kubernetes Secret-Objekts sicher im Cluster gespeichert. Die Referenzierung auf das richtige Secret-Objekt erfolgt im *Alertmanager*-Objekt über die Zeile 10 *configSecret:* (s. Quellcode 5.3). Damit der Alertmanager im Cluster gefunden werden kann, wird zusätzlich zum Deployment ein Service benötigt, der im Quellcode A.7 dargestellt wird.

```
8  spec:
9    replicas: 1
10   configSecret: alertmanager-main
```

Quellcode 5.3: Ausschnitt des Alertmanager Objekts

Die Konfiguration des Alertmanagers wird, wie zuvor erläutert, in einem Kubernetes Secret gespeichert. Die für die Implementierung verwendete Konfiguration zeigt der Quellcode A.8. Die Konfiguration erfolgt über das von Prometheus vorgegebene Schema (vgl. [Pro20d]), i.d.F. besteht es aus den drei Bereichen *global*, *route* und *receivers*. Über den Bereich *receivers* werden die Empfänger der Alerts definiert. Der Ausschnitt der Konfiguration Quellcode 5.4 zeigt die, i.d.F. zwei, definierten Empfänger. Zum einen handelt es sich hierbei um eine Weebhook Konfiguration, die über einen Service einen Alert in einen Microsoft Teams Kanal sendet, (*prometheus-msteams*) zum anderen um eine Konfiguration für einen E-Mail Empfänger (*mail-notifications*).

```
23       - receiver: 'mail-notifications'
24         match:
25           team: example-namespace
26     receivers:
27     - name: 'prometheus-msteams'
28       webhook_configs:
29       - url: http://<IP(MSTeamsServiec)>:Port/alertmanager
30     - name: 'mail-notifications'
31       email_configs:
32       - to: ToMail
33         from: FromMail
34         smarthost: smtp<Host>:Port
35         auth_username: user
```

Quellcode 5.4: Ausschnitt des Alertmanager Konfiguration *receivers*

Der Bereich *route* definiert, in welchen zeitlichen Abständen ein Alert gesendet wird, und an welchen Empfänger dieser adressiert wird. Die Auswahl des Empfängers erfolgt über

den *match:* Bereich *team: kubernetes-namespac* bzw. *team: example-namespace* Zeile 19 und 22 des Ausschnittes Quellcode 5.5.

```
11    route:
12      receiver: 'mail-notifications'
13      group_by:
14      - "job"
15      - "namespace"
16      group_wait: 30s
17      group_interval: 5m
18      repeat_interval: 1h
19      routes:
20      - receiver: 'prometheus-msteams'
21        match:
22          team: kubernetes-namespace
```

Quellcode 5.5: Ausschnitt des Alertmanager Konfiguration *route*

5.4.1 Empfänger Microsoft Teams

Da die Kommunikation innerhalb von Projektgruppen bzw. Praktikumsgruppen des Departements Informatik der HAW-Hamburg vermehrt über Microsoft Teams verläuft, hat sich der Fokus der Umsetzung des Versenden von Alerts auf diesen Kommunikationsweg fokussiert. Da, wie erläutert, der Alertmanager keine Möglichkeit bietet, als Empfänger einen Microsoft Teams Kanal zu konfigurieren, muss für das Versenden von Alerts hier ein zusätzlicher Service verwendet werden. Dieser ist ebenfalls ein Open Source Projekt, das auf Github veröffentlicht ist. Dieses Projekt bietet einen in Golang geschriebenen Webserver an, der die Alerts vom Alertmanager an den Microsoft Teams Kanal weiterleitet (vgl. [Git20h]).

Der Webserver wird mit Hilfe der Quellcodes A.9 im Namespace deployed. Mit Hilfe des in Quellcode A.10 dargestellten Service erhält der Webserver einen Endpoint im Kubernetes-Cluster. Mit Hilfe dieses Endpoints kann der Alertmanager einen Alert an diese Webserver senden, der diese wiederum an den konfigurierten Microsoft Teams Empfänger Kanal weiterleitet. Auf diesen Endpoint muss im Bereich *receivers* die Alertmanager-Konfiguration eingetragen sein.

Die Einstellung des individuellen Microsoft Teams Kanal, der verwendet werden soll, erfolgt über die Container Umgebungsvariable *TEAMS_INCOMING_WEBHOOK_-*

URL. Als Value muss hier die Incoming Webhook Adresse des Microsoft Teams Kanal eingetragen werden (s. Quellcode 5.6).

```
16      containers:
17      - env:
18          - name: TEAMS_INCOMING_WEBHOOK_URL
19            value: "https://outlook.office.com/webhook/XXX"
20          - name: TEAMS_REQUEST_URI
21            value: alertmanager
```

Quellcode 5.6: Ausschnitt des Deployment des Go Webservers zum Senden von Alerts an Microsoft Teams

5.4.2 Empfänger E-Mail

Alternativ zur Verwendung von Microsoft Teams wurde das Versenden von Alerts über E-Mails mit Hilfe des Quellcode A.8 konfiguriert. Dafür wurde ein *receiver mail-notifications* zur Konfiguration hinzugefügt (s. Quellcode 5.4). Über die definierte *route* wird konfiguriert, welcher Alert welcher Empfängerin bzw. welchem Empfänger zuzuordnen ist (s. Quellcode 5.5).

5.4.3 Nachteil

Da der Alertmanager im Namespace *monitoring* liegt und dieser nur für Admins einsehbar bzw. Konfigurierbar ist, kann nur ein ICC-Admin die Alertmanager-Konfiguration erstellen bzw. verändern. Dies erzeugt einen hohen Konfigurationsaufwand für die Admins, da sie für jeden Empfänger einen neuen *receivers*-Eintrag in der Alertmanager-Konfiguration hinzufügen müssen. Dieser Nachteil soll mit Hilfe der CRD *AlertmanagerConfig* bereinigt werden. Diese Ressource bietet die Möglichkeit, die Alertmanager-Konfiguration für User*innen in ihrem Namespace zu erweitern. Da diese Erweiterung des Prometheus-Operators zum Zeitpunkt dieser Arbeit nur als Alpha Version zu Verfügung steht und dieser noch Funktionen fehlen um sie in der ICC einsetzen zu können, konnte sie hier nicht eingesetzt werden. Alternativ könnten User*innen einen eigenen Alertmanager im Namespace starten, dies wäre in Summe aber eine Ressourcenverschwendung.

6 Auswertung

In der Auswertung wird analysiert, inwiefern die User-Storys, die in Kapitel 3.1 definiert wurden, durch die ausgearbeitete Monitoring Strategie abgedeckt werden konnten. Dazu wird zunächst anhand eines Beispiels gezeigt wie mit Hilfe des Prometheus-Operators eine Applikation überwacht werden kann (s. Kapitel 6.1). Zusätzlich wird das Thema Cluster-Monitoring durch den Prometheus-Operator erläutert (s. Kapitel 6.2). Abschließend werden die einzelnen Anforderungen der User-Storys der Ausarbeitung gegenübergestellt.

6.1 Anwendung des Prometheus-Operators

Nachfolgend wird erläutert wie, mit Hilfe des Prometheus-Operators, eine kleine Beispiel-Applikation überwacht werden kann. Die Applikation, die überwacht werden soll, ist eine Go-Applikation. Diese liefert unter <URL>/metrics, mit Hilfe der Prometheus Client-Bibliothek für Golang, Metriken im Prometheus Format.

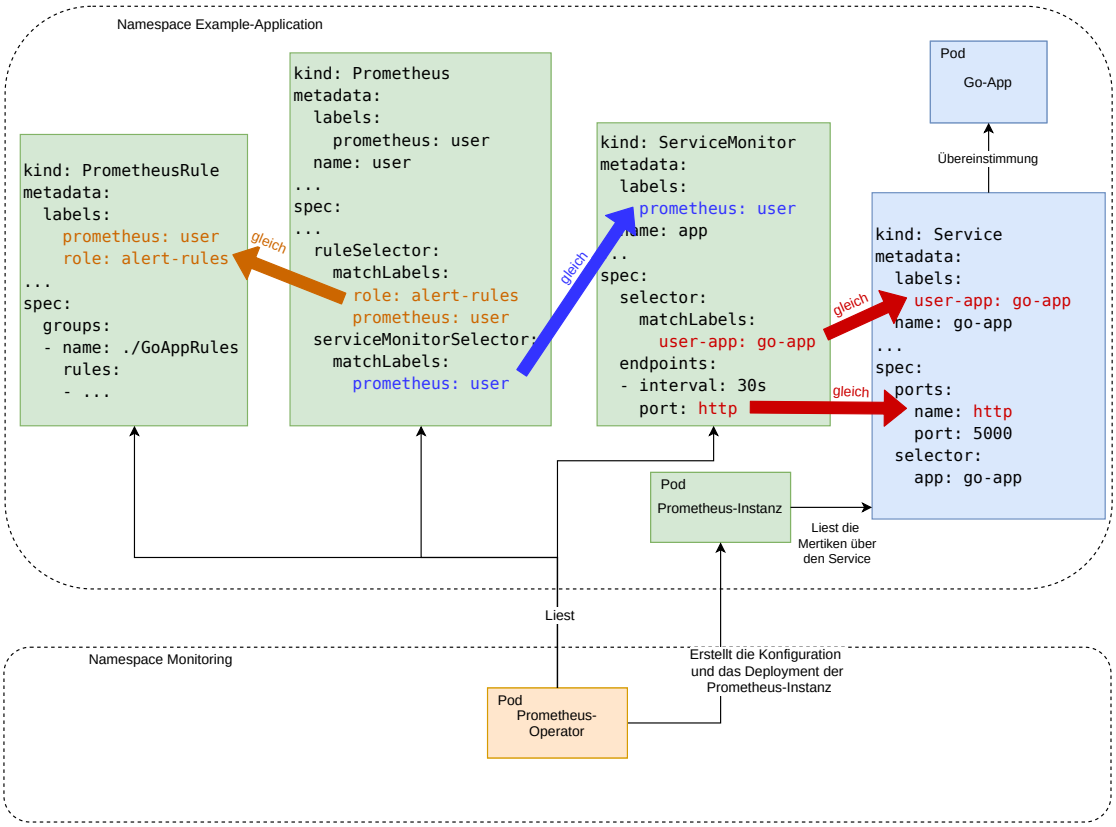

Abbildung 6.1: Überwachung einer Example-Application (vgl. [Git20k])

Die Abbildung 6.1 zeigt zunächst das Zusammenspiel der verschiedenen Objekte. Die *Go-App* stellt Metriken bereit. Über den *Service* kann, mit Hilfe des Labelings innerhalb des Systems, immer auf die Go-App zugegriffen werden. Im Weiteren erleichtert das Labeling von Kubernetes das Referenzieren über verschiedene Objekte, bspw. bezüglich der Verbindung zwischen *Prometheus* und dem *ServiceMonitor* bzw. zwischen dem *ServiceMonitor* und dem *Service* der *Go-App*. Hierbei müssen jeweils die *labels:* (z.b. prometheus: user bzw. user-app: go-app, s. Abbildung 6.1) mit den *match-labels* des *serviceMonitorSelectors* bzw. des *selectors* übereinstimmen. Dadurch ist der Prometheus-Operator imstande, die korrekte Konfiguration für die Prometheus-Instanz erstellen zu können.

Im Weiteren wird in Kapitel 6.1.1 und 6.1.2 erläutert, wie die Objekte *Prometheus* und *ServiceMonitor* aufgebaut werden.

6.1.1 Prometheus-Instanz

Um eine Prometheus-Instanz innerhalb seines Namespaces zu deployen, wird mit Hilfe der CRD Prometheus ein *Prometheus* Objekt im Namespace erstellt. Der Prometheus-Operator verwendet dieses Objekt, um eine Prometheus-Instanz im Namespace zu deployen.

Für das Überwachen der Go-App stellt der Quellcode A.16 die YAML-Datei dar, die benötigt wird, um das *Prometheus*-Objekt zu erstellen. In der Datei werden die Grundkonfigurationen der Prometheus-Instanz definiert.

Der Quellcode 6.1 zeigt die Zeilen 9-13 als Ausschnitt dieser Datei. Dieser Abschnitt definiert, welcher Alertmanager verwendet werden soll. Sollte der unter 5.4 definierte Alertmanager verwendet werden, muss der *Name*, der *Namespace* und der *Port* mit dem definierten Alertmanager übereinstimmen.

```
9  alerting:
10   alertmanagers:
11   - name: alertmanager-main
12     namespace: monitoring
13     port: web
```

Quellcode 6.1: Ausschnitt des Prometheus Objekts (vgl. [Git20m])

Da die Konfiguration des zu überwachenden Ziels, in diesem Beispiel die Go-APP, das ServiceMonitor Objekt übernimmt, muss in der Konfiguration das *Prometheus*-Objekt auf dieses *ServiceMonitor*-Objekt referenziert werden. Der Quellcode 6.2 zeigt einen Ausschnitt des *Prometheus*-Objekts. Für die Auswahl des gewünschten ServiceMonitors muss das *matchLabels:* in Zeile 33, wie Abbildung 6.1 zeigt, mit den *labels:* sowie dem *ServiceMonitor* übereinstimmen.

```
31 serviceMonitorSelector:
32   matchLabels:
33     prometheus: user
```

Quellcode 6.2: Ausschnitt des Prometheus Objekts (vgl. [Git20m])

Damit die Kommunikation der Prometheus-Instanz mit den User*innen bzw. mit dem Visualisierungstool Grafana einfacher ist, wird für die Prometheus-Instanz ein *Service* erstellt, der für die Kommunikation verantwortlich ist. Diesen stellt der Quellcode A.17 dar.

6.1.2 ServiceMonitor

Der ServiceMonitor definiert die Konfiguration der zu überwachenden Metriken. Dieser definiert, welche Applikation überwacht werden und wie die Überwachung aussehen soll, bspw. ein möglicher Überwachungs-Rhythmus. Im Beispiel *Go-App* definiert der Quellcode A.18 den ServiceMonitor. Die Auswahl der zu überwachenden Applikation erfolgt über das *selector: matchLabels:* (s. Quellcode 6.3). Dieses muss, wie in Abbildung 6.1 gezeigt, mit den *labels:* des Services der Applikation übereinstimmen. Zusätzlich muss auch der korrekte Port in Zeile 13 angegeben werden. Mit *interval:* kann die Häufigkeit der Abrufung der Metriken eingestellt werden.

```
8  spec:
9    selector:
10     matchLabels:
11       user-app: go-app
12   endpoints:
13   - interval: 30s
14     port: http
```

Quellcode 6.3: Ausschnitt des ServiceMonitor Objekts (vgl. [Git20m])

6.1.3 PrometheusRule

Mit Hilfe der hinzugefügten CRD Ressource PrometheusRule lässt sich das Objekt *PrometheusRule* erstellen. Dieses Objekt stellt die Konfiguration der Alerts für die Prometheus-Instanz dar. Der Quellcode A.19 zeigt das Objekt für das Beispiel *Go-App*. Damit diese PrometheusRule-Definition verwendet wird, muss das Labeling mit dem *matchLabels:* des *ruleSelector:*-Bereichs des *Prometheus*-Objekt übereinstimmen (s. Abbildung 6.1).

Der Quellcode 6.4 zeigt den Ausschnitt einer Alert-Definition des PrometheusRule-Codes. Dieser Alert trägt den Namen *GoAppDown*. Mit Hilfe von *expr:* und *for:* wird definiert, wann der Alert ausgelöst werden soll. In diesem Beispiel wird ein Alert gesendet, wenn der Container *go-app* für zwei Minuten nicht *up* ist. Über den *labels:* Bereich Zeile 17 werden dem Alert Labels hinzugefügt. Über diese Labels lässt sich der Empfänger des Alerts definieren. Die hier verwendete Syntax für ein Alert wurde von Prometheus definiert (vgl. [Pro20a]).

```
12    - alert: GoAppDown
13      annotations:
14        description: The GoApp pod is Down
15      expr: (absent(up{job="go-app"})) == 1
16      for: 2m
17      labels:
18        severity: warning
19        team: example-namespace
```

Quellcode 6.4: Ausschnitt aus PrometheusRule für Go-App

6.1.4 Grafana

Für die Visualisierung der von einer Prometheus-Instanz gesammelten Metrik wird das Open Source-System Grafana verwendet. Dieses wird im Namespace mit Hilfe des Quellcodes A.20 deployed. Für die Erreichbarkeit wird ein Service erstellt, diesen bildet der Quellcode A.21 ab. Mit Hilfe einer ConfigMap lässt sich die zu verwendete Datenquelle, die Prometheus-Instanz, vorab konfigurieren. Die Quelldatei A.22 zeigt die ConfigMap für das hier gezeigte Beispiel *Go-App*.

6.2 Cluster-Monitoring

In Kapitel 4.3.2 wurde erläutert, dass das Cluster-Monitoring für den Namespace *Kubernetes-Monitoring* vorgesehen ist. In diesem Namespace läuft eine Prometheus-Instanz, die mit Hilfe von verschiedenen ServiceMonitor-Objekten bzw. einem PrometheusRule-Objekt vom Prometheus-Operator konfiguriert wird. Das Bereitstellen der einzelnen Metriken erfolgt, wie bereits erläutert, über verschiedene Hilfssysteme, wie z.B. dem Node-Exporter oder kube-state-metrics. Die Grundlage für die verschiedenen Deployments des Cluster-Monitoring bildet ein, auf GitHub von Marcel Dempers veröffentlichtes, Projekt (vgl. [Dem20a]). Dieses Projekt stellt ein verwendbares Deployment für ein Cluster-Monitoring eines Kubernetes Cluster mit Prometheus-Operator bereit.

Mit Hilfe dieser Grundlage wurde im Namespace *Kubernetes-Monitoring* eine Prometheus-Instanz konfiguriert und deployed, die das Cluster-Monitoring für die ICC übernimmt. Folgende Kubernetes System bzw. Metriken können damit überwacht werden:

- kube-apiserver

- kubelet
- Server auf denen Kubernetes läuft (über den Node-Exporter)
- Kubernetes interne Metriken (über die kube-state-metrics)

Für Benachrichtigungen von möglichen Fehlern im Cluster wurde die, aus dem GitHub Projekt verfügbare, PrometheusRule-Datei für den Einsatz der ICC angepasst (vgl. [Dem20b]). Es wurde ein Beispiel-Empfänger hinzugefügt, anhand diesem werden die Alerts an einen Microsoft Teams Kanal gesendet.

Das GitHub Projekt verwendet zur Visualisierung Grafana. Es liefert dafür ein entsprechendes Deployment mit Dashboard-Konfigurationen der entsprechenden Metriken, das für die ICC in angepasster Form verwendet werden kann.

Die Abbildung 6.2 zeigt abschließend eine mögliche Visualisierung der Metriken des Node-Exporters über Grafana. Hier kann die Auslastung der einzelnen Nodes eingesehen werden.

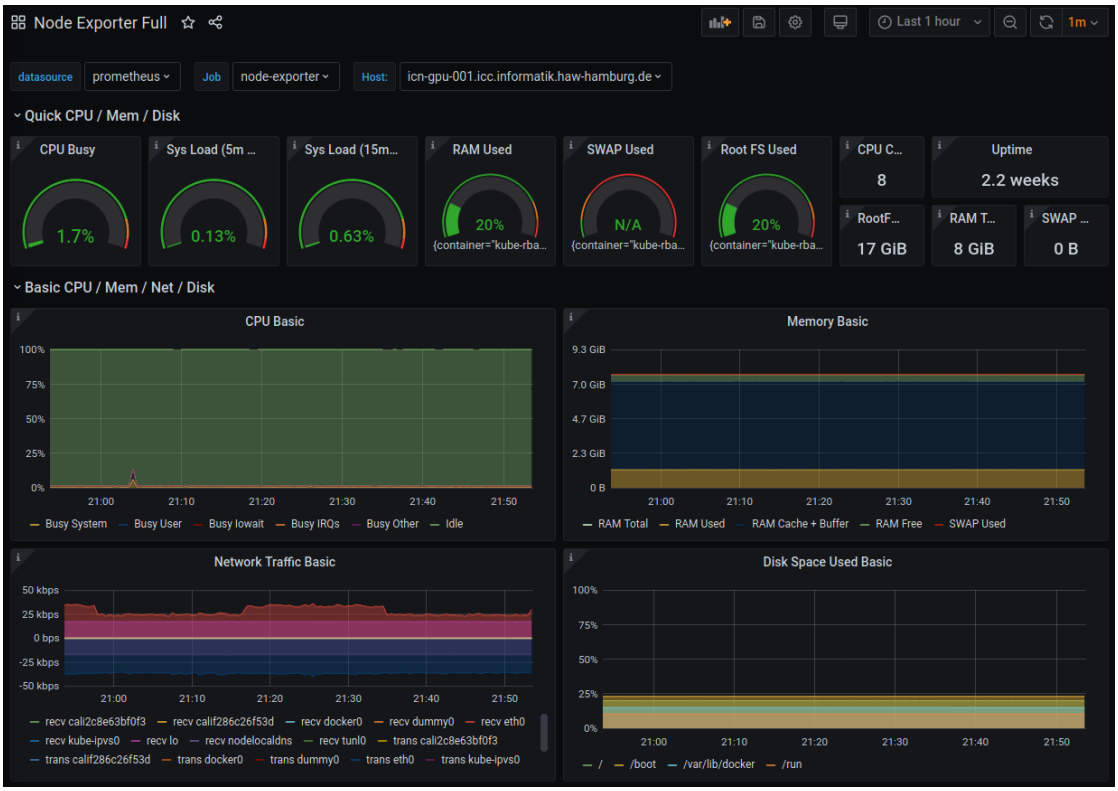

Abbildung 6.2: Datenvisualisierung über Grafana

6.3 Auswertung der User-Storys

6.3.1 ICC-Admin

Abbildung 6.3: ICC-Admin User-Story Auslastung

Dem ICC-Admin ist es möglich, einen Überblick über die Auslastung der ICC zu haben. Das Kapitel 6.2 erläutert wie, mit Hilfe des Node-Exporters, die Auslastung der einzelne Nodes, d.h. Servern, auf denen das Kubernetes-Cluster installiert ist, ausgelesen werden kann. Im Weiteren wurde gezeigt, wie sich die Kubernetes internen Prozesse überwachen lassen. Hierfür wurden die von *kubelet* und *kube-ApiServer* erstellten Metriken direkt abgerufen. Mit Hilfe des von *kube-state-metrics* erstellten Services können weitere Kubernetes interne Metriken ausgewertet werden. Für eine aussagekräftige Visualisierung der Daten kann Grafana verwendet werden (s. Abbildung 6.2).

Abbildung 6.4: ICC-Admin User-Story Alert

Mitteilungen können über den Prometheus-Alertmanager versendet werden. Damit ist es dem ICC-Admin möglich, eine Benachrichtigung zu erhalten, wenn bestimmte Prozesse nicht erreichbar sind. Über die im Kapitel 5.4 erläuterte Konfiguration des Prometheus-Alertmanagers ist es möglich, Alerts über Microsoft Teams (s. 5.4.1) bzw. E-Mail (s. 5.4.2) zu versenden. Über die PrometheusRule-Ressource (s. 6.1.3) können Alert-Regeln aus den gesammelten Metriken erstellt werden. Es ist bspw. möglich eine Regel aufzustellen, die bei einer Fehlfunktion im Kubernetes-Cluster, einen Alert an den Alertmanager auslöst. Bei dieser Fehlfunktion könnte es sich z.B. um eine nicht laufende *kube-API* handeln. Dieser Mechanismus wurde in Kapitel 6.2 erläutert.

Abbildung 6.5: ICC-Admin User-Story Deployment

Es ist möglich die Anzahl der aktuell laufenden bzw. nicht laufenden Pods im Kubernetes-Cluster anzeigen zu lassen. Das Abschalten oder die Benachrichtigung der User*innen ist mit der aktuellen Lösung bislang nicht möglich. Hierfür würde ein Service benötigt werden, der aus den Metrik-Daten der Prometheus-Instanz des Namespaces *kubernetes-monitoring*, die nicht laufenden Pods zu einem User bzw. einer Userin mappen, und anschließend Benachrichtigungen an den entsprechenden Kanal senden kann.

6.3.2 ICC-User*in

Abbildung 6.6: ICC-User*in User-Story Auslastung

Ein Überblick der Auslastung meines Systems als ICC-User ist mit Hilfe der verschiedenen Metrik-Bibliotheken möglich. Prometheus bietet für verschiedene Programmiersprachen Client Bibliotheken direkt an bzw. durch inoffizielle Dritt-Anbieter (vgl. [Pro20c]). Zusätzlich bietet Prometheus für einige Systeme Exporter an, die aus diesen Systemen Metriken in das Prometheus Format exportieren, bspw. für die Webserver Apache2 und Nginx, sowie für Datenbanken und Messaging Systeme (vgl. [Pro20f]). Anhand des Beispiels aus Kapitel 6.1 wurde an einer Go-Application gezeigt, wie die Überwachung mit Hilfe des Prometheus-Operators aussehen kann.

Abbildung 6.7: ICC-User*in User-Story Alert

Als ICC-User ist es möglich über den Prometheus-Alertmanager einen Alert zu versenden. Wie im Kapitel 6.1.3 erläutert ist es außerdem möglich, zu überprüfen, ob ein Service noch läuft und im Falle einer Fehlfunktion, Alerts zu versenden.

Abbildung 6.8: ICC-User*in User-Story persistentes Volumen

Der Prometheus-Operator bietet gegenwärtig keine Möglichkeit, das persistente Volumen innerhalb eines Namespace zu überwachen. Dies könnte über eine neue CRD des Prometheus-Operators ergänzt werden, zum aktuellen Zeitpunkt liegen darüber aber keine Informationen vor.

6.3.3 ICC-Student

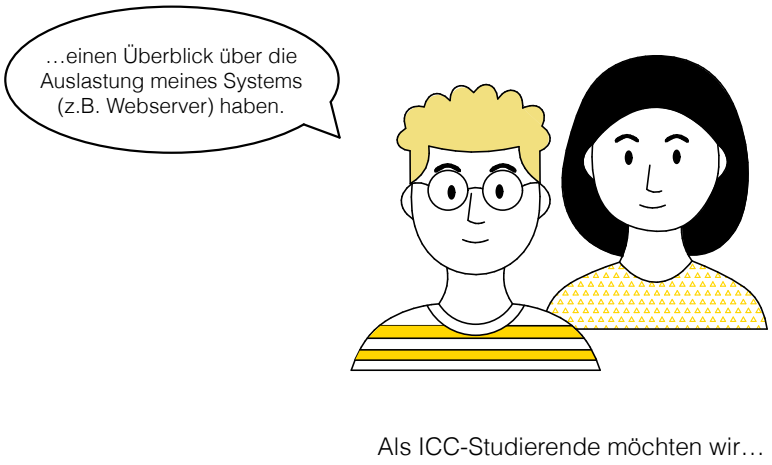

Abbildung 6.9: ICC-Student User-Story Auslastung

Wie bereits in der Auswertung der ICC-User*innen beschrieben, ist es möglich, als ICC-Student die Auslastung des Systems zu überblicken.

Abbildung 6.10: ICC-Student User-Story Fortschritt

Da Prometheus ein auf Metriken basierendes Monitoring Tool ist, können mit dieser Lösung nur Metriken von Applikationen oder Services gesammelt und ausgewertet werden. Verfügt eine Applikation über Metriken, können diese über einen ServiceMonitor mit einer Prometheus-Instanz überwacht und mit Grafana visualisiert werden (s. Kapitel 6.1). Im Praktikum können diese Visualisierung gezeigt werden.

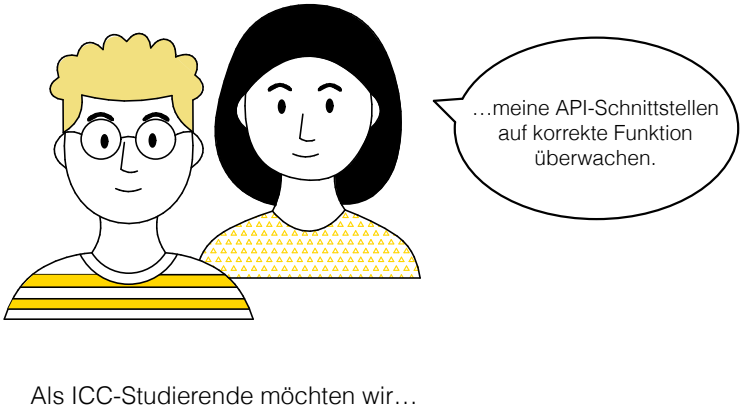

Abbildung 6.11: ICC-Student User-Story API-Schnittstellen

Genau so wie bei der vorherigen Anforderung können mit der in diese Arbeit betrachteten Lösung nur Metriken verarbeitet werden. Daher kann eine API-Schnittstelle nur mit Hilfe der Metriken, die die API-Schnittstelle selbst preisgibt, überwacht werden. Es kann bspw. überprüft werden, ob eine API-Schnittstelle erreichbar ist.

6.3.4 ICC-ML-User*in

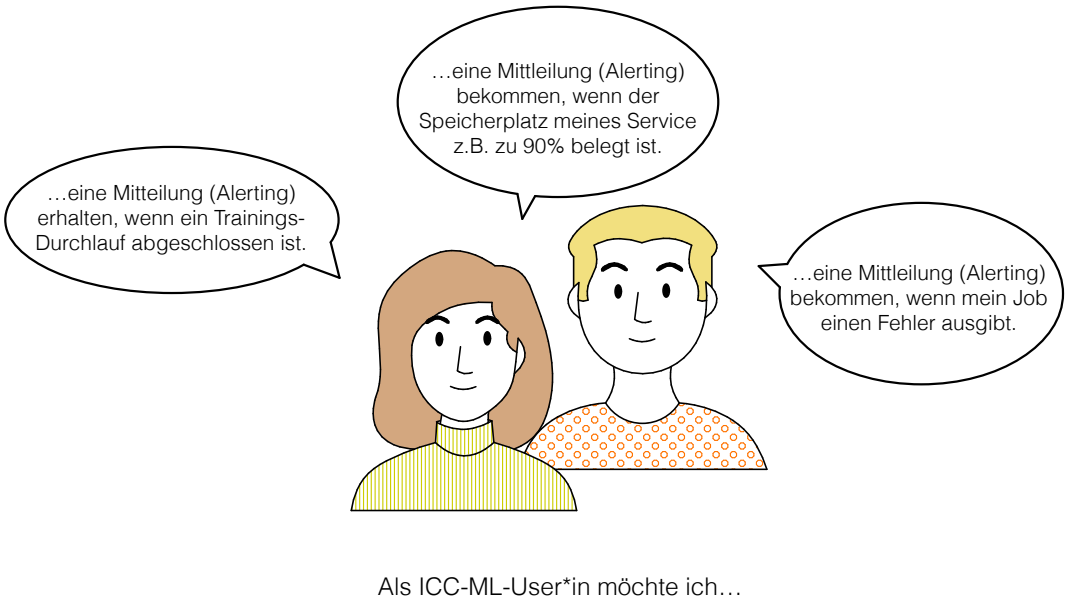

Abbildung 6.12: ICC-ML-User*in User-Story Auswertung

Da der Prometheus-Operator aktuell keine Möglichkeit anbietet einen Kubernetes Job zu überwachen, ist es als ICC-ML-User*in nicht möglich, Alert-Regeln zu erstellen, die auf einem Kubernetes-Job basieren. Wie bereits in der Auswertung der ICC-User*in erläutert, ist es nicht möglich ein persistentes Volumen innerhalb eines Namespaces zu überwachen. Aus diesem Grund ist es nicht möglich einen Alert dahingehend auszulösen.

7 Fazit

Das Ziel dieser Arbeit war es, eine Monitoring Infrastruktur für die Informatik Compute Cloud (ICC) des Departments Informatik der HAW-Hamburg zu entwickeln.

Dabei musste zunächst analysiert werden, welche Anforderungen verschiedene Nutzergruppen an eine mögliche Überwachungslösung haben. Für Admins ist es bspw. wichtig, sich einen Überblick über das Kubernetes-Cluster zu verschaffen, Studierende möchten jedoch nur speziell ihre entwickelte Applikation überwachen. Mit Hilfe der Nutzwertanalyse konnte aus der Vielzahl von potenziell einsetzbaren Monitoring Tools ein Passendes für die Anforderungen der ICC gefunden werden. Dabei war Teil der Problematik, die Fülle an Monitoring Tools zu überblicken, da sich gerade in den letzten Jahren eine Vielzahl von Open-Source Projekten dieser Thematik widmen. Nachdem das Ergebnis der Nutzwertanalyse eindeutig auf das Monitoring Tool Prometheus verwies, war die nachfolgende Herausforderung, eine optimale Integration von Prometheus in die ICC zu entwickeln. Das Projekt Prometheus-Operator hat die meisten der zuvor definierten Anforderungen der Usergruppen erfüllt und bietet zusätzlich für die User*innen eine benutzerfreundliche Methode zur Überwachung einer Applikation.

Die Nutzergruppen ICC-Admin, ICC-User*in, ICC-Studierende können in hohem Maße von den Möglichkeiten des Prometheus-Operators profitieren. Die meisten ihrer Anforderungen konnten in der vorliegenden Arbeit umgesetzt werden, bspw. einen Überblick über die Kubernetes-Prozesse zu behalten (Admins), oder auch Fehlermeldungen bzgl. nicht laufender Applikationen über MS Teams zu erhalten (Admins, User*innen, Studierende). Da das Prometheus-Operator-Projekt selbst noch in der Beta-Phase ist, fehlt zur Zeit der Arbeit noch die Möglichkeit, Kubernetes-Jobs bzw. ein persistentes Volumen zu überwachen. Daraus folgt für die Nutzergruppe ICC-KI-User*in aktuell nur eine eingeschränkte Einsatzmöglichkeit für die in dieser Arbeit beschriebene Monitoring Strategie. Dies könnte sich aber mit einer Weiterentwicklung des Prometheus-Operator innerhalb kurzer Zeit ändern.

Zusammenfassend lässt sich sagen, dass die vorliegende Lösung zur Etablierung einer Monitoring Strategie für die Informatik Compute Cloud (ICC) durch den Prometheus-Operator weitreichende Möglichkeiten bietet, um die beteiligten Nutzergruppen in ihrer Arbeit zu unterstützen. Die Arbeit mit dem Prometheus-Operator überzeugt, da der Konfigurationsaufwand, im Gegensatz zu einer vergleichbaren Strategie, erheblich reduziert ist. Aus diesem Grund war nach der Implementierung des Prometheus-Operators die Erstellung einer Überwachung für eine Applikation mit einfachen Mitteln möglich. Die Einschränkungen, die der Prometheus-Operator zum Zeitpunkt der Arbeit noch hat, wurden nicht mit einem zusätzlichen System behoben, um eine Einheitlichkeit des Systems nicht zu beeinträchtigen. So ist eine Aktualisierung des Prometheus-Operators deutlich einfacher zu handhaben. Durch eine mögliche Erweiterung innerhalb des Beta-Projektes könnten diese Anforderungen in der Zukunft jedoch künftig abschließend erfüllt werden.

Literaturverzeichnis

[AD19] John Arundel and Justin Domingus. *Cloud Native DevOps mit Kubernetes - Bauen, Deployen und Skalieren moderner Anwendungen in der Cloud.* Dpunkt.Verlag GmbH, Heidelberg, 2019.

[Ahu20] Madhur Ahuja. Onlineartikel: Monitoring Kubernetes with Graphite. https://www.metricfire.com/blog/monitoring-kubernetes-with-graphite/, 2020. Stand: 2020-08-14.

[Bai20] Eduardo Baitello. Medium: Trying Prometheus Operator with Helm + Minikube. https://medium.com/faun/trying-prometheus-operator-with-helm-minikube-b617a2dccfa3, 2020. Stand: 2020-11-09.

[Bit19] Bitkom. Onlinepräsenz Bitkom: Cloud Nutzung der Unternehmen. https://www.bitkom.org/Presse/Presseinformation/Cloud-Nutzung-auf-Rekordniveau-bei-Unternehmen, 2019. Stand: 2020-04-03.

[Bur18] Brendan Burns. *Verteilte Systeme mit Kubernetes entwerfen - Patterns und Prinzipien für skalierbare und zuverlässige Services.* O'Reilly, Sebastopol, 2018.

[Dat20] DataDog. Onlinepräsenz DataDog: Modern monitoring and security. https://www.datadoghq.com/, 2020. Stand: 2020-08-15.

[Dav20] Chris Davis. Graphite Docs: Tools That Work With Graphite. https://graphite.readthedocs.io/en/latest/tools.html, 2020. Stand: 2020-08-13.

[Dem20a] Marcel Dempers. GitHub: docker-development-youtube-series/monitoring/prometheus/kubernetes/1.18.4/. https://github.com/marcel-dempers/docker-development-youtube-series/tree/master/monitoring/prometheus/kubernetes/1.18.4, 2020. Stand: 2020-11-03.

[Dem20b] Marcel Dempers. GitHub: docker-development-youtube-series/monitoring/prometheus/kubernetes/1.18.4/prometheus-cluster-monitoring/prometheus.rules.yaml. https://github.com/marcel-dempers/docker-development-youtube-series/blob/master/monitoring/prometheus/kubernetes/1.18.4/prometheus-cluster-monitoring/prometheus.rules.yaml, 2020. Stand: 2020-11-03.

[Git20a] GitHub. GitHub: GitHub-module. https://github.com/mailru/graphite-nginx-module, 2020. Stand: 2020-08-13.

[Git20b] GitHub. GitHub: Helm-Chart - kube-prometheus-stack. https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack#kube-prometheus-stack, 2020. Stand: 2020-10-28.

[Git20c] GitHub. GitHub: IPMI Sensor Input Plugin. https://github.com/influxdata/telegraf/tree/master/plugins/inputs/ipmi_sensor, 2020. Stand: 2020-08-14.

[Git20d] GitHub. GitHub: kube-prometheus/manifests/grafana-deployment.yaml. https://github.com/prometheus-operator/kube-prometheus/blob/master/manifests/grafana-deployment.yaml, 2020. Stand: 2020-11-02.

[Git20e] GitHub. GitHub: kube-prometheus/manifests/grafana-service.yaml. https://github.com/prometheus-operator/kube-prometheus/blob/master/manifests/grafana-service.yaml, 2020. Stand: 2020-11-02.

[Git20f] GitHub. GitHub: kube-state-metrics. https://github.com/kubernetes/kube-state-metrics, 2020. Stand: 2020-10-14.

[Git20g] GitHub. GitHub: Node exporter. https://github.com/prometheus/node_exporter, 2020. Stand: 2020-10-14.

[Git20h] GitHub. GitHub: prometheus-msteams. https://github.com/prometheus-msteams/prometheus-msteams#customise-messages-to-ms-teams, 2020. Stand: 2020-11-1.

[Git20i] GitHub. GitHub: Prometheus Operator. https://github.com/prometheus-operator/prometheus-operator, 2020. Stand: 2020-10-11.

[Git20j] GitHub. GitHub: Prometheus Operator-CustomResourceDefinitions. https://github.com/prometheus-operator/prometheus-operator#customresourcedefinitions, 2020. Stand: 2020-10-25.

[Git20k] GitHub. GitHub: Prometheus Operator Documentation custom-metrics-elements.png. https://github.com/prometheus-operator/prometheus-operator/blob/master/Documentation/custom-metrics-elements.png, 2020. Stand: 2020-10-27.

[Git20l] GitHub. GitHub: Prometheus Operator /example/prometheus-operator-crd/. https://github.com/prometheus-operator/prometheus-operator/tree/master/example/prometheus-operator-crd, 2020. Stand: 2020-10-30.

[Git20m] GitHub. GitHub: Prometheus Operator /example/rbac/prometheus/. https://github.com/prometheus-operator/prometheus-operator/tree/master/example/rbac/prometheus, 2020. Stand: 2020-10-29.

[Git20n] GitHub. GitHub: Prometheus Operator /example/rbac/prometheus/. https://github.com/prometheus-operator/prometheus-operator/tree/master/example/rbac/prometheus, 2020. Stand: 2020-11-01.

[Git20o] GitHub. GitHub: Prometheus Operator /example/rbac/prometheus-operator/. https://github.com/prometheus-operator/prometheus-operator/tree/master/example/rbac/prometheus-operator, 2020. Stand: 2020-10-28.

[Git20p] GitHub. GitHub: Prometheus Operator /example/rbac/prometheus-operator-crd/. https://github.com/prometheus-operator/prometheus-operator/tree/master/example/rbac/prometheus-operator-crd, 2020. Stand: 2020-10-29.

[Git20q] GitHub. GitHub: prometheus_bot. https://github.com/inCaller/prometheus_bot, 2020. Stand: 2020-08-15.

[Git20r] GitHub. GitHub: Rbac for prometheus Operator CRDs. https://github.com/prometheus-operator/prometheus-operator/blob/master/Documentation/rbac-crd.md, 2020. Stand: 2020-11-01.

[Git20s] GitLab. Onlinepräsenz GitLab Docs: Monitoring Gitlab with Prometheus. https://docs.gitlab.com/ee/administration/monitoring/prometheus/, 2020. Stand: 2020-08-14.

[Gra20] Graphite. Graphite Docs: Graphite does three things. https://graphiteapp.org/, 2020. Stand: 2020-08-15.

[Inf20a] InfluxData. Onlinepräsenz influxdata: Telegraf. https://www.influxdata.com/time-series-platform/telegraf/, 2020. Stand: 2020-08-14.

[Inf20b] InfluxData. Onlinepräsenz InfluxDB Docs: Apache HTTP Server Telegraf Plugin. https://www.influxdata.com/integration/apache-http-server/, 2020. Stand: 2020-08-13.

[Inf20c] InfluxData. Onlinepräsenz InfluxDB Docs: Client libraries. https://docs.influxdata.com/influxdb/v1.8/tools/api_client_libraries/, 2020. Stand: 2020-08-13.

[Inf20d] InfluxData. Onlinepräsenz InfluxDB Docs: Get InfluxDB. https://www.influxdata.com/get-influxdb/, 2020. Stand: 2020-08-13.

[Inf20e] InfluxData. Onlinepräsenz InfluxDB Docs: Handlers. https://docs.influxdata.com/kapacitor/v1.5/working/alerts/#handlers, 2020. Stand: 2020-08-15.

[Inf20f] InfluxData. Onlinepräsenz InfluxDB Docs: HAProxy Monitoring Integration. https://www.influxdata.com/integration/haproxy-monitoring/, 2020. Stand: 2020-08-14.

[Inf20g] InfluxData. Onlinepräsenz InfluxDB Docs: Kubernetes monitoring integration. https://www.influxdata.com/integration/kubernetes-monitoring/, 2020. Stand: 2020-08-14.

[Inf20h] InfluxData. Onlinepräsenz InfluxDB Docs: Microsoft Teams event handler. https://docs.influxdata.com/kapacitor/v1.5/event_handlers/microsoftteams/, 2020. Stand: 2020-09-04.

[Inf20i] InfluxData. Onlinepräsenz InfluxDB Docs: Monitor data and send alerts. https://v2.docs.influxdata.com/v2.0/monitor-alert/, 2020. Stand: 2020-08-15.

[Inf20j] InfluxData. Onlinepräsenz InfluxDB Docs: Nginx monitoring tools. https://www.influxdata.com/integration/nginx-monitoring-tools/, 2020. Stand: 2020-08-13.

[Inf20k] InfluxData. Onlinepräsenz InfluxDB Docs: SNMP integration. https://www.influxdata.com/integration/snmp/, 2020. Stand: 2020-08-14.

[ITW20] ITWissen. Onlinepräsenz ITWissen: Multi-Tenancy-Architektur. https://www.itwissen.info/Multi-Tenancy-Architektur-multi-tenancy-architecture.html, 2020. Stand: 2020-11-08.

[Jea20] Jeager. Onlinepräsenz Jeager: open source, end-to-end distributed tracing. https://www.jaegertracing.io/, 2020. Stand: 2020-08-15.

[Jen20] John Jensen. GitHub: graphite-snmp-collector. https://github.com/jensenja/graphite-snmp-collector, 2020. Stand: 2020-08-13.

[Kin20] Kind. Onlinepräsenz Kind: Home. https://kind.sigs.k8s.io/, 2020. Stand: 2020-11-08.

[Kle20] Kirill Klenov. GitHub: graphite-beacon. https://github.com/klen/graphite-beacon, 2020. Stand: 2020-08-15.

[Kna20] Andy Knapp. GitHub: Prometheus Alert Manager for MS Teams. https://github.com/prometheus-msteams/prometheus-msteams, 2020. Stand: 2020-09-04.

[Kot20] Subhakar Kotta. Onlinepräsenz DZone: Deploy a Production-Ready Kubernetes Cluster Using kubespray. https://dzone.com/articles/kubernetes-113-installation-using-kubespray, 2020. Stand: 2020-11-06.

[Kub20a] Kubernetes. Onlinepräsenz Kubernetes: Authorization Modes. https://kubernetes.io/docs/reference/access-authn-authz/authorization/#authorization-modules, 2020. Stand: 2020-11-06.

[Kub20b] Kubernetes. Onlinepräsenz Kubernetes: Configure Service Accounts for Pods. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/, 2020. Stand: 2020-11-06.

[Kub20c] Kubernetes. Onlinepräsenz Kubernetes: CustomResourcesDefinitions. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions/, 2020. Stand: 2020-10-13.

[Kub20d] Kubernetes. Onlinepräsenz Kubernetes: Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2020. Stand: 2020-11-06.

[Kub20e] Kubernetes. Onlinepräsenz Kubernetes: Install and Set Up kubectl. https://kubernetes.io/docs/tasks/tools/install-kubectl/, 2020. Stand: 2020-11-06.

[Kub20f] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - Container runtimes. https://kubernetes.io/docs/concepts/containers/#container-runtimes, 2020. Stand: 2020-11-06.

[Kub20g] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - kube-apiserver. https://kubernetes.io/de/docs/concepts/overview/components/#kube-apiserver, 2020. Stand: 2020-11-06.

[Kub20h] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - kube-controller-manager. https://kubernetes.io/de/docs/concepts/overview/components/#kube-controller-manager, 2020. Stand: 2020-11-06.

[Kub20i] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - kube-proxy. https://kubernetes.io/de/docs/concepts/overview/components/#kube-proxy, 2020. Stand: 2020-11-06.

[Kub20j] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - kube-scheduler. https://kubernetes.io/de/docs/concepts/overview/components/#kube-scheduler, 2020. Stand: 2020-11-06.

[Kub20k] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - kubelet. https://kubernetes.io/de/docs/concepts/overview/components/#kubelet, 2020. Stand: 2020-11-06.

[Kub20l] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten - master-komponenten. https://kubernetes.io/de/docs/concepts/overview/components/#master-komponenten, 2020. Stand: 2020-11-06.

[Kub20m] Kubernetes. Onlinepräsenz Kubernetes: Kubernetes Komponenten-etcd. https://kubernetes.io/de/docs/concepts/overview/components/#etcd, 2020. Stand: 2020-11-06.

[Kub20n] Kubernetes. Onlinepräsenz Kubernetes: Metrics For The Kubernetes Control Plane. https://kubernetes.io/docs/concepts/cluster-administration/monitoring/, 2020. Stand: 2020-08-14.

[Kub20o] Kubernetes. Onlinepräsenz Kubernetes: Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/, 2020. Stand: 2020-11-06.

[Kub20p] Kubernetes. Onlinepräsenz Kubernetes: Nodes. https://kubernetes.io/de/docs/concepts/architecture/nodes/, 2020. Stand: 2020-11-06.

[Kub20q] Kubernetes. Onlinepräsenz Kubernetes: Pods. https://kubernetes.io/docs/concepts/workloads/pods/, 2020. Stand: 2020-11-06.

[Kub20r] Kubernetes. Onlinepräsenz Kubernetes: Role and ClusterRole. https://kubernetes.io/docs/reference/access-authn-authz/rbac/#role-and-clusterrole, 2020. Stand: 2020-11-06.

[Kub20s] Kubernetes. Onlinepräsenz Kubernetes: RoleBinding and ClusterRoleBinding. https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding, 2020. Stand: 2020-11-06.

[Kub20t] Kubernetes. Onlinepräsenz Kubernetes: Service resources. https://kubernetes.io/docs/concepts/services-networking/service/#service-resource/, 2020. Stand: 2020-11-06.

[Kub20u] Kubernetes. Onlinepräsenz Kubernetes: Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/, 2020. Stand: 2020-11-06.

[Lab20] Bitnami Labs. GitHub: kubewatch. https://github.com/bitnami-labs/kubewatch, 2020. Stand: 2020-08-15.

[Mat20] Mattermost. Onlinepräsenz Mattermost: Prometheus Alertmanager Plugin. https://integrations.mattermost.com/alertmanager-plugin/, 2020. Stand: 2020-08-15.

[Nel20] Daniel Nelson. GitHub: Kubernetes Input Plugin. https://github.com/influxdata/telegraf/tree/master/plugins/inputs/kubernetes, 2020. Stand: 2020-08-14.

[Par20] Pavlos Parissis. GitHub: haproxystats. https://github.com/unixsurfer/haproxystats#graphite-section, 2020. Stand: 2020-08-13.

[Pro20a] Prometheus. Onlinepräsenz Prometheus: Alerting rules. https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/, 2020. Stand: 2020-11-02.

[Pro20b] Prometheus. Onlinepräsenz Prometheus: Alertmanager. https://prometheus.io/docs/alerting/latest/alertmanager/, 2020. Stand: 2020-08-15.

[Pro20c] Prometheus. Onlinepräsenz Prometheus: Client libraries. https://prometheus.io/docs/instrumenting/clientlibs/, 2020. Stand: 2020-08-13.

[Pro20d] Prometheus. Onlinepräsenz Prometheus: Configuration. https://prometheus.io/docs/alerting/latest/configuration/, 2020. Stand: 2020-08-15.

[Pro20e] Prometheus. Onlinepräsenz Prometheus: Download. https://prometheus.io/download/, 2020. Stand: 2020-08-13.

[Pro20f] Prometheus. Onlinepräsenz Prometheus: Exporters and integrations. https://prometheus.io/docs/instrumenting/exporters/#http, 2020. Stand: 2020-08-13.

[Pro20g] Prometheus. Onlinepräsenz Prometheus: Installation. https://prometheus.io/docs/prometheus/latest/installation/, 2020. Stand: 2020-11-09.

[Red20a] RedHat. Onlinepräsenz RedHat: Kubernetes (k8s) erklärt. https://www.redhat.com/de/topics/containers/what-is-kubernetes, 2020. Stand: 2020-11-06.

[Red20b] RedHat. Onlinepräsenz RedHat: Vergleich: Container oder vms? https://www.redhat.com/de/topics/containers/containers-vs-vms, 2020. Stand: 2020-11-06.

[Sys20] Sysdig. Onlinepräsenz sysdis: How It Works. https://sysdig.com/products/monitor/prometheus-monitoring/, 2020. Stand: 2020-08-15.

[Use20] Userdoc. Onlinepräsenz Userdoc HAW Department Informatik: Informatik Compute Cloud. https://userdoc.informatik.haw-hamburg.de/doku.php?id=docu:informatikcomputecloud, 2020. Stand: 2020-11-03.

A Anhang

A.1 Dateien zum Kapitel 5.2

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/name: prometheus-operator
    app.kubernetes.io/version: v0.42.1
  name: prometheus-operator
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/component: controller
      app.kubernetes.io/name: prometheus-operator
  template:
    metadata:
      labels:
        app.kubernetes.io/component: controller
        app.kubernetes.io/name: prometheus-operator
        app.kubernetes.io/version: v0.42.1
    spec:
      containers:
      - args:
        - --kubelet-service=kube-system/kubelet
        - --logtostderr=true
        - --prometheus-config-reloader=quay.io/prometheus-operator/prometheus
    -config-reloader:v0.42.1
        image: quay.io/prometheus-operator/prometheus-operator:v0.42.1
        name: prometheus-operator
        ports:
        - containerPort: 8080
```

```
32          name: http
33        resources:
34          limits:
35            cpu: 200m
36            memory: 200Mi
37          requests:
38            cpu: 100m
39            memory: 100Mi
40        securityContext:
41          allowPrivilegeEscalation: false
42      nodeSelector:
43        beta.kubernetes.io/os: linux
44      securityContext:
45        runAsNonRoot: true
46        runAsUser: 65534
47      serviceAccountName: prometheus-operator
```

Quellcode A.1: Prometheus-Operators Deployment (vgl. [Git20o])

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    labels:
5      app.kubernetes.io/component: controller
6      app.kubernetes.io/name: prometheus-operator
7      app.kubernetes.io/version: v0.42.1
8    name: prometheus-operator
9    namespace: monitoring
10 spec:
11   clusterIP: None
12   ports:
13   - name: http
14     port: 8080
15     targetPort: http
16   selector:
17     app.kubernetes.io/component: controller
18     app.kubernetes.io/name: prometheus-operator
```

Quellcode A.2: Prometheus-Operators Service (vgl. [Git20o])

```
1  apiVersion: v1
2  kind: ServiceAccount
3  metadata:
4    labels:
5      app.kubernetes.io/component: controller
```

```
    app.kubernetes.io/name: prometheus-operator
    app.kubernetes.io/version: v0.42.1
  name: prometheus-operator
  namespace: monitoring
```

Quellcode A.3: Prometheus-Operators ServiceAccount (vgl. [Git20o])

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/name: prometheus-operator
    app.kubernetes.io/version: v0.42.1
  name: prometheus-operator
rules:
- apiGroups:
  - monitoring.coreos.com
  resources:
  - alertmanagers
  - alertmanagers/finalizers
  - alertmanagerconfigs
  - prometheuses
  - prometheuses/finalizers
  - thanosrulers
  - thanosrulers/finalizers
  - servicemonitors
  - podmonitors
  - probes
  - prometheusrules
  verbs:
  - '*'
- apiGroups:
  - apps
  resources:
  - statefulsets
  verbs:
  - '*'
- apiGroups:
  - ""
  resources:
  - configmaps
  - secrets
  verbs:
  - '*'
```

```
39 - apiGroups:
40   - ""
41   resources:
42   - pods
43   verbs:
44   - list
45   - delete
46 - apiGroups:
47   - ""
48   resources:
49   - services
50   - services/finalizers
51   - endpoints
52   verbs:
53   - get
54   - create
55   - update
56   - delete
57 - apiGroups:
58   - ""
59   resources:
60   - nodes
61   verbs:
62   - list
63   - watch
64 - apiGroups:
65   - ""
66   resources:
67   - namespaces
68   verbs:
69   - get
70   - list
71   - watch
72 - apiGroups:
73   - networking.k8s.io
74   resources:
75   - ingresses
76   verbs:
77   - get
78   - list
79   - watch
```

Quellcode A.4: Prometheus-Operators ClusterRole (vgl. [Git20o])

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/name: prometheus-operator
    app.kubernetes.io/version: v0.42.1
  name: prometheus-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus-operator
subjects:
- kind: ServiceAccount
  name: prometheus-operator
  namespace: monitoring
```

Quellcode A.5: Prometheus-Operators ClusterRoleBinding (vgl. [Git20o])

A.2 Dateien zum Kapitel 5.4

```yaml
apiVersion: monitoring.coreos.com/v1
kind: Alertmanager
metadata:
  labels:
    alertmanager: main
  name: main
  namespace: monitoring
spec:
  replicas: 1
  configSecret: alertmanager-main
  nodeSelector:
    kubernetes.io/os: linux
```

Quellcode A.6: Alertmanager Objekts

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    alertmanager: main
  name: alertmanager-main
  namespace: monitoring
```

```
spec:
  ports:
  - name: web
    port: 9093
    targetPort: web
  selector:
    alertmanager: main
    app: alertmanager
  sessionAffinity: ClientIP
```

Quellcode A.7: Alertmanager Service

```
apiVersion: v1
data: {}
kind: Secret
metadata:
  name: alertmanager-main
  namespace: monitoring
stringData:
  alertmanager.yaml: |-
    global:
      resolve_timeout: 5m
    route:
      receiver: 'mail-notifications'
      group_by:
      - "job"
      - "namespace"
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 1h
      routes:
      - receiver: 'prometheus-msteams'
        match:
          team: kubernetes-namespace
      - receiver: 'mail-notifications'
        match:
          team: example-namespace
    receivers:
    - name: 'prometheus-msteams'
      webhook_configs:
      - url: http://<IP(MSTeamsServiec)>:Port/alertmanager
    - name: 'mail-notifications'
      email_configs:
      - to: ToMail
        from: FromMail
```

```
34         smarthost: smtp<Host>:Port
35         auth_username: user
36         auth_identity: user
37         auth_password: <PW>
38         send_resolved: true
39 type: Opaque
```

Quellcode A.8: Alertmanager Secret

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    labels:
5      app: teams
6    name: teams
7  spec:
8    selector:
9      matchLabels:
10       app: teams
11   template:
12     metadata:
13       labels:
14         app: teams
15     spec:
16       containers:
17       - env:
18           - name: TEAMS_INCOMING_WEBHOOK_URL
19             value: "https://outlook.office.com/webhook/XXX"
20           - name: TEAMS_REQUEST_URI
21             value: alertmanager
22         name: teams
23         image: quay.io/prometheusmsteams/prometheus-msteams:latest
24         ports:
25         - name: http
26           containerPort: 2000
```

Quellcode A.9: Deployment des Go Web Servers zum Senden von Alerts an Microsoft Teams

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    labels:
5      app: teams
6    name: teams
```

```
spec:
  selector:
    app: teams
  ports:
  - name: http
    port: 2000
    targetPort: http
```

Quellcode A.10: Service des Go Web Servers zum Senden von Alerts an Microsoft Teams

A.3 Dateien zum Kapitel 5.3

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: prometheus-crd-view
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
rules:
- apiGroups: ["monitoring.coreos.com"]
  resources: ["alertmanagers", "prometheuses", "prometheusrules", "
    servicemonitors"]
  verbs: ["get", "list", "watch"]
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: prometheus-crd-edit
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
rules:
- apiGroups: ["monitoring.coreos.com"]
  resources: ["alertmanagers", "prometheuses", "prometheusrules", "
    servicemonitors"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

Quellcode A.11: Benutzerrollen der CRDs (vgl. [Git20p])

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
```

```
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
```

Quellcode A.12: ClusterRole für User Prometheus-Instanzen(vgl. [Git20n])

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: example-application
```

Quellcode A.13: ClusterRoleBinding für User Prometheus-Instanzen(vgl. [Git20n])

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: example-application
```

Quellcode A.14: ServiceAccount für User Prometheus-Instanzen(vgl. [Git20n])

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-admin
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/metrics
```

```
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
  - configmaps
  verbs: ["get"]
- apiGroups:
  - networking.k8s.io
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
```

Quellcode A.15: ClusterRole für Admin Prometheus-Instanzen(vgl. [Git20n])

A.4 Dateien zum Kapitel 6.1

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  labels:
    prometheus: user
  name: user
  namespace: example-application
spec:
  alerting:
    alertmanagers:
    - name: alertmanager-main
      namespace: monitoring
      port: web
  image: quay.io/prometheus/prometheus:v2.19.2
  nodeSelector:
    kubernetes.io/os: linux
  replicas: 1
  resources:
    requests:
      memory: 400Mi
  serviceAccountName: prometheus
  version: v2.19.2
  ruleSelector:
    matchLabels:
```

```
      role: alert-rules
      prometheus: user
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
  serviceMonitorSelector:
    matchLabels:
      prometheus: user
```

Quellcode A.16: Prometheus Objekts (vgl. [Git20m])

```
apiVersion: v1
kind: Service
metadata:
  labels:
    prometheus: user
  name: prometheus-user
  namespace: example-application
spec:
  ports:
  - name: web
    port: 9090
    targetPort: web
  selector:
    app: prometheus
    prometheus: user
  sessionAffinity: ClientIP
```

Quellcode A.17: Prometheus-Instanz Service (vgl. [Git20m])

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    prometheus: user
  name: app
  namespace: example-application
spec:
  selector:
    matchLabels:
      user-app: go-app
  endpoints:
  - interval: 30s
```

```
    port: http
```

Quellcode A.18: ServiceMonitor Objekts (vgl. [Git20m])

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: user
    role: alert-rules
  name: prometheus-goapp-rules
spec:
  groups:
  - name: ./GoAppRules
    rules:
    - alert: GoAppDown
      annotations:
        description: The GoApp pod is Down
      expr: (absent(up{job="go-app"})) == 1
      for: 2m
      labels:
        severity: warning
        team: example-namespace
```

Quellcode A.19: PrometheusRule für Go-App

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
  namespace: example-application
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      name: grafana
      labels:
        app: grafana
    spec:
      containers:
      - name: grafana
        image: grafana/grafana:latest
```

```
        ports:
        - name: http
          containerPort: 3000
        resources:
          limits:
            cpu: 200m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 100Mi
        volumeMounts:
          - mountPath: /var/lib/grafana
            name: grafana-storage
          - mountPath: /etc/grafana/provisioning/datasources
            name: grafana-datasources
            readOnly: false
      volumes:
        - name: grafana-storage
          emptyDir: {}
        - name: grafana-datasources
          configMap:
              defaultMode: 420
              name: grafana-datasources
```

Quellcode A.20: Grafana Deployment (vgl. [Git20d])

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: grafana
  name: grafana
  namespace: example-application
spec:
  ports:
  - name: http
    port: 3000
    targetPort: http
  selector:
    app: grafana
```

Quellcode A.21: Grafana Service (vgl. [Git20e])

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: grafana-datasources
  namespace: example-application
data:
  prometheus.yaml: |-
    {
        "apiVersion": 1,
        "datasources": [
            {
               "access":"proxy",
                "editable": true,
                "name": "prometheus",
                "orgId": 1,
                "type": "prometheus",
                "url": "http://prometheus-user:9090",
                "version": 1
            }
        ]
    }
```

Quellcode A.22: Grafana Dashboard Konfiguration

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Monitoring eines Multi-Tenancy Kubernetes Clusters am Beispiel der HAW Hamburg Compute Cloud

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original