

Inhaltsverzeichnis

1	Einleitung	3
1.1	Struktur	4
2	Grundlagen	5
2.1	Kubernetes	5
2.1.1	Master Worker Prinzip	5
2.1.2	Relevante Komponenten	6
2.2	Metriken	7
2.3	Anomalie	8
2.4	Prometheus	8
2.4.1	Scraping	9
2.4.2	Abfragen	9
2.4.3	Alerting	9
3	Stand der Technik	11
3.1	Anomaly Detection	11
3.2	Cluster-Skalierung	12
4	Datenaggregation	14
4.1	Toolauswahl	14
4.2	Grafana	16
4.3	Datenquellen	16
4.3.1	'USE'-Methode	16
4.3.2	Metriken	19
5	Auswerten der Metriken	20
5.1	Klassifizierung	21
5.2	Logische Auswertung	21
5.3	Graphische Aufbereitung	22
6	Automatisierte Aktionen	23
6.1	Architektur des Systems	24
6.1.1	Prometheus	25
6.1.2	Alertmanager	26
6.1.3	Regeln	26
7	Alert-Action-Manager	29
7.0.1	Aufgabe	29
7.0.2	Anforderungen	29
7.1	Wahl der Sprache	29
7.2	Funktionen	31
7.2.1	Erweiterung	32

7.3	Aktionen	32
7.3.1	Skalierung	32
7.3.2	Anomalie-Detection	33
7.3.3	Metriken	34
7.3.4	Grenzwerte	34
8	Evaluation	34
8.1	Messaufbau	34
8.2	Regeln	34
8.3	Grenzwerte	34
8.4	Komponenten und Architektur	34
9	Diskussion	34
10	Fazit und Ausblick	34

1 Einleitung

Anbieter von Cloud-Services haben den Anspruch, möglichst geringe und kurze Ausfallzeiten mit ihren Services zu erreichen. Um eine maximal lange, störungsfreie Servicelaufzeit zu erreichen, ist es notwendig jederzeit den Servicestatus einsehen und mögliches Fehlverhalten frühzeitig erkennen zu können. Da für eine dauerhafte Kontrolle eines Services ein oder sogar mehrere Mitarbeiter benötigt würden, welche eine eintönige Kontrollaufgabe übernehmen müssten, ist es sinnvoll möglichst viele Teile der Kontrolle zu automatisieren. Diese Automatisierung bringt einerseits den Vorteil der Kosteneinsparung, da keine Mitarbeiter für diese Aufgabe benötigt werden und andererseits einen Geschwindigkeitsvorteil durch die wesentlich geringere Reaktionszeit, die durch die Geschwindigkeit von Computern gegenüber dem Menschen einhergeht.

Hierbei sollen vor allem Engpässe bei Ressourcen ausfindig gemacht werden, sowie Anomalien, also Fehlverhalten, in einzelnen Komponenten der Infrastruktur gefunden und behoben werden, bestenfalls noch bevor sich größere Auswirkungen auf die restlichen Komponenten ergeben. Sofern Ressourcenengpässe, also hohe Last, auftritt und dies frühzeitig erkannt wird, können einzelne Services gezielt skaliert und so Beeinträchtigungen auf die Funktion verhindert werden. Da Engpässe oft temporär auftreten, werden Services sowohl hoch- als herunterskaliert um so immer die ideale Zahl an Ressourcen bereitgestellt. Des Weiteren soll Fehlverhalten detektiert werden. Dies liegt dann vor, wenn hohe Ressourcenlast ohne erkennbaren Grund vorliegt. Das wäre beispielsweise dann der Fall, wenn die Prozessorlast oder Speicherlast eines Services auf einem sehr hohen Wert läuft, gleichzeitig aber keine hohe Netzwerklast durch Nutzer vorliegt, die dieses Verhalten begründet. In diesem Fall kann von einer anomalen Funktion ausgegangen und ein Service neu, im besten Fall unterbrechungsfrei, bereitgestellt werden.

Um eine automatisierte Erkennung zu ermöglichen, werden Daten sog. Metriken benötigt, die eine Entscheidung auf Basis des vorliegenden Verhaltens treffen lassen. Metriken müssen erhoben und ausgewertet werden, um eine Aktion aus ihnen schließen zu können, welche die vorliegende Anomalie oder den vorliegenden Engpass beheben kann.

Die erhobenen Metriken müssen einerseits für Menschen lesbar sein, um aktuelle Zustände widerspiegeln und entsprechend darauf reagieren zu können, andererseits ebenso für Computer auswertbar sein, um die Automatisierung durch diese zu ermöglichen.

Neben der Möglichkeit automatisierte Aktionen auszuführen, ist es auch sinnvoll entsprechende verantwortliche Administratoren über das Fehlverhalten in Kenntnis zu setzen und diese zu benachrichtigen, um ihnen die Möglichkeit zu geben dem Verhalten auf den Grund zu gehen.

Diese Arbeit setzt sich das Ziel die Durchführbarkeit der automatisierten Anomalie- und Engpasserkennung nachzuweisen und die erste Implementierung innerhalb eines schon bestehenden Kubernetes-Clusters. Im Rahmen dieser Arbeit werden die passenden Komponenten gewählt, die zur Umsetzung der Anforderungen benötigt werden, die Infrastruktur geplant, erstellt und die korrekte Funktion evaluiert.

Des Weiteren wird die Relevanz verschiedener erhobener Metriken in Bezug auf ihre Verwendbarkeit beim automatisierten Detektieren von Anomalien und Engpässen dargestellt und geklärt.

Es werden die weit verbreiteten Tools Prometheus und Grafana verwendet und durch Eigenentwicklungen ergänzt und so eine Infrastruktur geschaffen, welche die Anforderungen erfüllen kann.

1.1 Struktur

Diese Arbeit ist in neun Kapitel strukturiert. Im zweiten Kapitel werden die Grundlagen und für das Verständnis der weiteren Arbeit benötigtes Wissen vermittelt. Das Kapitel vermittelt grundlegendes Wissen zu den verwendeten Tools. Im dritten Kapitel wird der Status Quo der Technik ermittelt und somit die technische Ausgangslage der Arbeit geklärt. Mit dem vierten Kapitel beginnt der praktische Teil des Projekts. Am Anfang dessen steht die Vorgehensweise beim Aggregieren von Daten aus einem Kubernetes-Cluster. Nachdem Daten aggregiert wurden, müssen diese verarbeitet und ausgewertet werden. Das Vorgehen hierbei wird in Kapitel fünf erläutert. Die ausgewerteten Daten haben den Zweck automatisierte Aktionen auszulösen. Wie passende Aktionen gewählt und angewandt werden, damit befasst sich das sechste Kapitel. Hier werden verwendete Komponenten, Architektur, Aktionen, Regeln sowie die passende Programmiersprache erläutert. Nachdem die geplanten Features implementiert sind, müssen diese auf ihre korrekte Funktion getestet werden. Im siebten Kapitel wird geklärt, wie die korrekte Funktion evaluiert wird, wie der Messaufbau gestaltet wurde und welche Komponenten auf welche Art und Weise getestet wurden.

Nach der Evaluation werden die Ergebnisse der Arbeit diskutiert. Hier werden Fragen geklärt ob alle Ziele erreicht wurden oder aus welchem Grund Entscheidungen getroffen wurden. Diese und weitere Fragen werden in der Diskussion in Kapitel acht diskutiert. Zum Schluss, in Kapitel neun, wird ein Fazit aus dem zurückliegenden Projekt gezogen und ein möglicher Ausblick in die Zukunft gestellt.

2 Grundlagen

2.1 Kubernetes

Kubernetes, kurz "k8s", ist eine ursprünglich von Google entwickelte, mittlerweile aber quelloffene Software zur Orchestrierung und Deployment von containerisierten Anwendungen. Seit seiner Einführung 2014 hat Kubernetes ein starkes Wachstum erlebt und ist zum Quasistandard bei der Entwicklung von Cloud-Native Applikationen geworden. Diese Übersicht der Grundlagen spricht die für dieses Projekt benötigten Komponenten an, für weitergehende Informationen gibt es die Dokumentation[Referenz auf Kube-Doku]

Die Relevanz von Kubernetes lässt sich an der halbjährlichen Befragung der Community der Cloud-Native Computing Foundation ablesen: In der Umfrage stellte sich heraus, dass über 78 Prozent der 1337 Befragten Kubernetes verwendet.[Umfrage CN-CF 2020 ergänzen]

Kubernetes ist eine mittlerweile bewährte Infrastruktur und bietet Software die nötig ist um zuverlässige und skalierbare verteilte Systeme zu entwickeln. [1]

2.1.1 Master Worker Prinzip

Der Zweck eines Kubernetes-Clusters besteht darin viele, einzelne Computer als eine einzige Einheit zusammenarbeiten zu lassen. Ein Cluster besteht aus zwei verschiedenen Arten von Komponenten, die zusammenarbeiten: dem Node und dem Master. Der Master stellt den Verwalter im Cluster dar. Er koordiniert alle Vorgänge, trifft also Entscheidungen die von globaler Bedeutung für das gesamte Cluster sind. Beispielsweise startet er Komponenten oder schaltet sie ab, ist aber auch für Tasks wie die Zeitplanung zuständig. Er wird über die Kubernetes-API angesprochen und steht in direkter Verbindung mit den Nodes.[2] [3] Nodes wiederum stellen die Arbeiter dar, weshalb sie auch "worker" genannt werden. Deren Aufgaben bestehen darin Pods[siehe Relevante Komponenten] aufrecht zu erhalten und die Laufzeitumgebung bereitzustellen. Nodes halten auch die Container-Runtime bereit, welche dafür zuständig ist Containerisierte Anwendungen auszuführen und so das Verteilen einer Anwendung auf beliebige Hardware möglich macht.

<evtl. hier nochmal Grafik Master/Node>

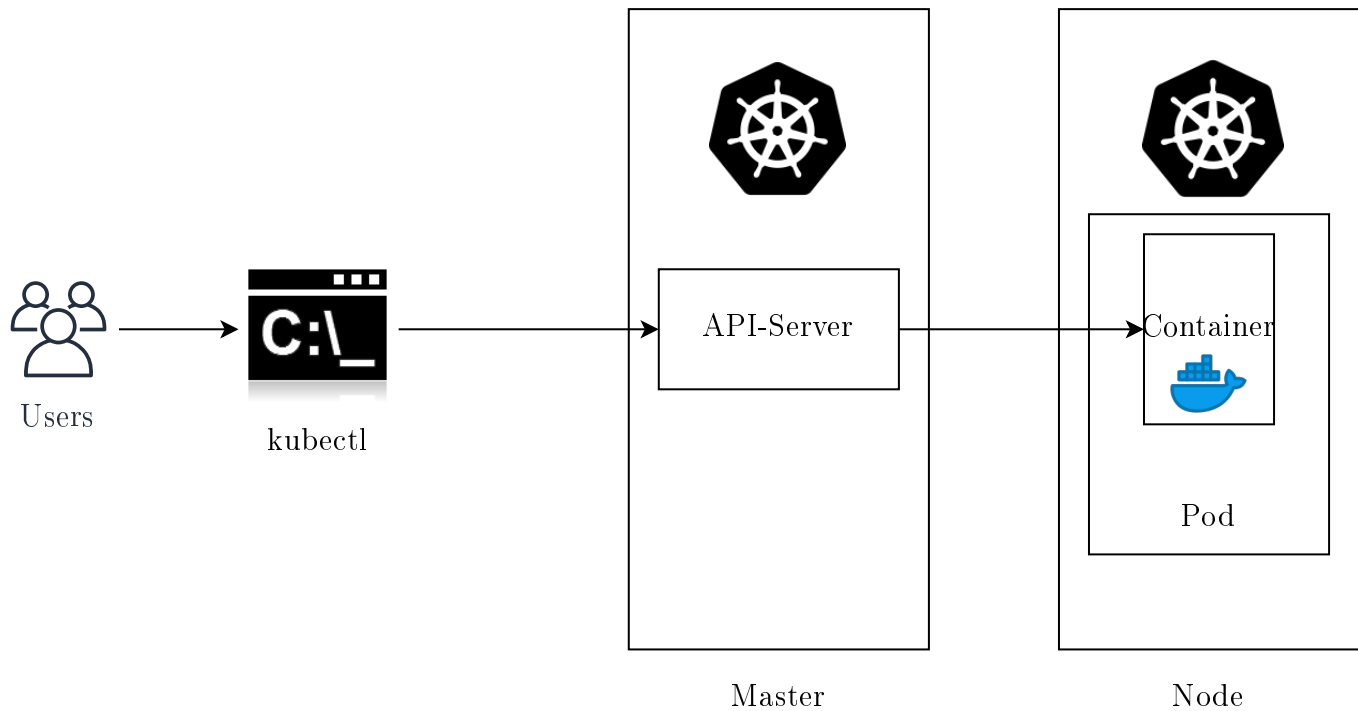


Abbildung 1: einfache Darstellung eines Kubernetes-Cluster (eigene Zeichnung)

2.1.2 Relevante Komponenten

Kubernetes bietet eine Vielzahl von Komponenten für unterschiedliche Aufgaben. Die für dieses Projekt Relevanten werden hier erklärt:

kubelet :

Der kubelet ist der primäre 'node-agent'. Er ist dafür zuständig die Nodes beim Kubernetes API-Server zu registrieren. Des Weiteren verwaltet er Pods anhand einer Podspezifikation(PodSpec) und sorgt dafür, dass die Pods im Rahmen der Spezifikation "gesund"laufen.

Über den kubelet können diverse Metriken gesammelt werden, die über den Status des Nodes oder der darin laufenden Pods und Container Auskunft geben.

Kubernetes bietet verschiedene Organisationskonzepte mit deren Hilfe sich die Kubernetes-Struktur umsetzen lässt, die für dieses Projekt wichtigsten werden im Folgenden erläutert:

Pod :

Ein Pod ist eine 'execution unit' und repräsentiert einen Prozess, der in einem Cluster läuft. Ein Pod kapselt einen oder mehrere Anwendungs-Container, ein eigener Speicherbereich, eine eigene IP-Adresse sowie dessen Konfigurationsoptionen. Die meistverwendete Container-Runtime ist, wie auch in diesem Projekt, Docker, es

gibt aber auch Unterstützung für Weitere wie beispielsweise Rocket.

Deployment :

Das Deployment ist eine Beschreibung des Zustands, in dem Pods ausgeliefert werden sollen. Hier werden beispielsweise der Name, Namensraum, Ressourcenlimits oder auch die Größe der Skalierung definiert.

Service :

Nach der offiziellen Kubernetes-Dokumentation sind Services Eine abstrakte Möglichkeit, eine Anwendung, die auf einer Reihe von Pods läuft, als Netzwerkdienst bereitzustellen". Kubernetes Pods können dynamisch in ihrer Anzahl skalieren und so auch ihre IP-Adresse wechseln. Daher ist es sinnvoll die Pods über einen Service anzusprechen, der mit einem DNS ähnlichen System funktioniert und so ein Deployment über einen lesbaren Namen ansprechbar macht.

kubectl :

kubectl ist eine Kontrollanwendung für Kubernetes. Es ist eine direkte Schnittstelle zwischen User und dem Kubernetes API-Server und kann mittels Konsolenbefehlen bedient werden. Das Tool stellt eine einfache Möglichkeit zur Bedienung des Kubernetes-Clusters dar.

Ein kubectl-Befehl besteht aus der Aktion und dessen Ziel und sieht folgendermaßen aus:

```
kubectl <Aktion><Ziel>
```

Kubernetes bietet systemseitig Funktionen, die beim ermitteln von Cluster- und Systeminformationen helfen, eine davon ist der cAdvisor

cAdvisor :

Da Container von sich aus keine Informationen zu ihrem Ressourcenstatus nach außen preisgeben oder exportieren, bedarf es eines Hilfsmittels, das genau dies macht. cAdvisor(Container Advisor) ist ein Daemon, der Ressourcen-Informationen aus Containern sammelt, verarbeitet und exportiert.[4]

2.2 Metriken

Eine Metrik ist eine Funktion, die einen Zustand oder eine Eigenschaft als Maßzahl abbildet. Metriken in der Informatik lassen sich im Grunde in 3 Bereiche einteilen:

- Service-Metriken, welche die Performance eines Service bemessen, zum Beispiel die Unterbrechungsfreie Laufzeit.
- Prozess-Metriken, die für die Quantifizierung des Entwicklungsprozesses einer Software verwendet werden

- Technologie-Metriken, welche die zugrunde liegende Technologie quantifizieren, zum Beispiel die Speicherauslastung

Wenn Metriken über einen Zeitraum beobachtet werden und nach Messzeit strukturiert werden, werden sie Zeitreihen-Metriken genannt.

In dieser Arbeit werden Service-Metriken erstellt und verwendet, welche die Performance der in einem Kubernetes-Cluster laufenden Services beziffert.

<hier Quelle finden !>

2.3 Anomalie

Eine Anomalie oder anormales Verhalten bezeichnet das Verhalten eines Programms oder Service, das stark von dessen Normalzustand abweicht. Dies kann beispielsweise eine hohe Ressourcenlast verursacht durch fehlerhaften Code sein.

<Quelle bezieht sich auf 'A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures'>

2.4 Prometheus

Prometheus ist ein Open-Source Monitoring-Toolkit. Es wurde ursprünglich von SoundCloud entwickelt, ist aber mittlerweile ein Open-Source Projekt, das der Cloud Native Computing Foundation (CNCF) beigetreten ist. Die primären Funktionen des Toolkits sind das Aufzeichnen von Zeitreihen-Metriken und das Alarmieren bei Überschreitungen von Grenzwerten der Metriken. Des weiteren bietet Prometheus:

- Ein WebUI zum Visualisieren der aufgezeichneten Daten
- Eine eigene Abfragesprache(PromQL) für aufgezeichneten Metriken Regeln, Visualisierungen oder ähnliches erstellen zu können
- Einen Alertmanager um Alerts entgegen zu nehmen und weiter verwalten zu können
- Eine 'Target-Discovery' um selbstständig sinnvolle und parametrisierbare Ziele zu entdecken

Prometheus bietet eine Vielzahl an Komponenten, die das Überwachen von Systemen und Alarmieren unterstützen. Die grundlegende Komponente ist hierbei der zentrale Prometheus Server. Dieser ist dafür zuständig Metriken zu sammeln, im Folgenden scrapen genannt, und zentral zu speichern, sofern es gewünscht ist.

2.4.1 Scraping

Der Begriff Scraping bezeichnet das Sammeln von Metriken durch den Prometheus-Server. In Prometheus funktioniert dies folgendermaßen: Ein Scraping-Target, beispielsweise ein Kubernetes-Node, besitzt einen sog. Exporter, der Metriken aus dem System ausliest und diese an einem HTTP-Endpunkt '<IP-Adresse/DNS-Name>/metrics' bereitstellt. Der Prometheus-Server findet entweder per automatischem Target-Discovery Mechanismus das Target oder wird per Konfiguration darauf eingestellt. Der Exporter auf dem Target hat im Normalfall einen Aktualisierungszeitraum ebenso wie der Prometheus-Server, sodass die Metriken automatisch aktualisiert werden und eine Zeitreihenmetrik erzeugt wird.

2.4.2 Abfragen

Das Abfragen von Metriken wird mittels der prometheuseigenen, an SQL angelehnten Query-Language PromQL durchgeführt. PromQL-Requests werden an den Prometheus-Server gestellt, der die Requests prüft, verarbeitet und entsprechende Werte als Antwort zurückgibt. Die Abfragen können sehr einfach sein, indem beispielsweise nur der Name einer Metrik angegeben wird und so der entsprechende Wert zurückgeliefert wird. Requests können aber auch, ähnlich zu SQL, miteinander kombiniert werden um voneinander abhängige Werte abzufragen oder Werte über unterschiedliche Zeiträume zu erhalten.

Ein beispielhafter Request: Dieser Request bildet den Mittelwert der HTTP-Codes 401 über die letzten 5 Minuten ab:

```
rate ( http_status { code = '401' } [5m] )
```

In diesem Projekt werden PromQL-Abfragen vor allem für zwei verschiedene Zwecke verwendet:

- zum erstellen von Regeln, bei deren Erfüllung der Prometheus-Server einen Alert verschickt (siehe nächstes Kapitel Alerting)
- zum Visualisieren der Metriken in der Prometheus Web-UI oder in Grafana-Boards

2.4.3 Alerting

Das Alerting in Prometheus funktioniert mittels festgelegten Regeln. Diese werden in der PromQL-Sprache auf dem Prometheus-Server definiert und gespeichert. Sobald eine Regel erfüllt ist sendet der Server einen Alert an den Alertmanager, der diesen dann weiter verarbeiten kann.

Der Alert-Vorgang besteht im Grunde aus zwei Teilen. Der erste Teil ist die PromQL-Bedingung, die erfüllt sein muss. Sobald diese erfüllt ist erhält der Alert den Status 'pending'. In diesem Status wird der Alert noch nicht versendet, sondern wartet auf das Erfüllen eines vorgegebenen Zeitwertes. Erst nach erfüllen des Wertes wird der Alert an

den Manager versendet.

Der Alertmanager hat mehrere Möglichkeiten mit Alerts umzugehen. Eine der Möglichkeiten ist das Weiterleiten an definierbare Ziele, beispielsweise an bestimmte E-Mail Adressen, Chat-Programme wie Slack oder auch an Webhooks bzw. HTTP-Endpunkte. Außerdem stellt der Alertmanager eingetroffene Alerts auf dem eigenen Web-UI dar. Hier kann eingesehen werden welchen Status ein Alert zum aktuellen Zeitpunkt hat, 'pending' oder 'active', außerdem können Alerts stillgelegt werden, welche dann nicht mehr durch den Manager weiterverarbeitet werden.

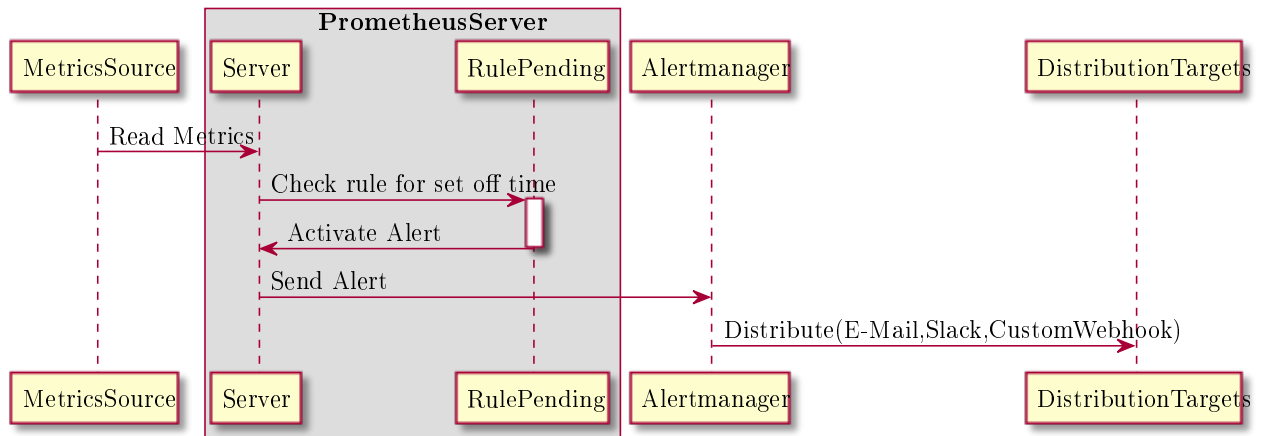


Abbildung 2: Alertmanager Aufbau (eigene Zeichnung)

In Abbildung 2 ist der sequenzielle Ablauf eines Alarmierungsvorgangs dargestellt. Nachdem Metriken vom Server von einer Metrikquelle gelesen wurden und eine Alertinregel erfüllen, wird der Zeitraum, in welcher die Regeln erfüllt ist gemessen, der Alert wird mit dem Status 'pending' an den Alertmanager gesendet. Besteht der Zustand des Alerts für die konfigurierte Offset-Zeit, wird der Alert in den Status 'active' versetzt und an alle im Alertmanager konfigurierten Ziele versendet.

3 Stand der Technik

3.1 Anomaly Detection

In ihrem Paper 'Anomaly Detection and Diagnosis for Container-based Microservices with Performance Monitoring' beschreiben Qingfeng Du et al. eine Möglichkeit mithilfe der Performance und Hardware-Metriken von Container basierten Microservices und Machine Learning Techniken Anomalien zu erkennen und so Service Level Agreement valuations (SLAV) zu reduzieren bzw. zu verhindern. Das Anomaly Detection System, folgend ADS abgekürzt, besteht aus drei verschiedenen Modulen. Diese Module sind das Monitoring-Modul, das dafür zuständig ist Performance Daten aus dem Zielsystem auszulesen, das Data-Processing Modul das die ausgelesenen Daten auf Anomalien zu prüfen sowie zuletzt das 'Fault-Injection' Modul, das Fehlerfälle im System erzeugt und so einen Datensatz erhält, der dafür genutzt wird das Machine-Learning Model anzulernen und zu validieren.

Die verschiedenen beschriebenen Arten der Anomalien, die das System abdeckt sind 'CPU-Hog's, Memory Leaks oder der Package-Loss von Containern. Die beiden Hauptaufgaben des Systems sind daher, die Klassifizierung, ob in einem Microservice eine Anomalie vorliegt und falls dies der Fall ist zu lokalisieren, wo diese stattfindet.

Dieses Paper bezieht sich ausschließlich auf die Detektion der Anomaly, nicht aber auf das Beheben dieser.

Weitere Paper verwenden ebenfalls Machine Learning Modelle. In dem Paper 'A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures' von Areeg Samir et al., wird eine Controller-Architektur vorgestellt, die autonom Anomalien erkennt und diese selbstständig behebt. Der Controller ermittelt mögliche Anomalien in der zugrundeliegenden Providerinfrastruktur. Um Performance Anomalien zu entdecken und identifizieren werden sogenannte 'Hidden Markov'-Machine Learning Modelle verwendet.

'KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches' Chin-Wei Tien et al., das in dem Journal 'Engineering Reports' veröffentlicht wurde. In diesem Paper wird eine Methode untersucht, mit deren Hilfe sich anormales Verhalten von Containern in Kubernetes-Clustern aufspüren lassen soll. Hierfür wird ein neuronaler Ansatz verwendet und ein Klassifikationsmodell trainiert, der das Containerverhalten selbstständig untersucht.

In diesem Projekt wird ein anderer Ansatz gewählt, welcher auf Regeln mit statischen Thresholds basiert. Dieser verspricht gegenüber den Machine-Learning Ansätzen einige Vorteile die es zu untersuchen vor Allem die leichte Erweiterbarkeit des System durch einfaches Erweitern von Regeln und die Einsatzmöglichkeit für Firmen, die keine Expertise in Machine Learning besitzen.

3.2 Cluster-Skalierung

In dem Paper 'ACCRS: autonomic based cloud computing resource scaling' geschrieben von Ziad A. Al-Sharif et al. wird ein Skalierungssystem für Cloud-Ressourcen beschrieben. Dieses arbeitet mithilfe statischer Regeln und Thresholds.

Das System besteht aus mehreren Komponenten, die Erste, welche die Basis des Ganzen bildet ist die 'System State Monitoring' Komponente. Sie zeichnet unter anderem CPU, RAM, Netzwerk Utilization auf. Um aus den aufgezeichneten Monitoringdaten Entscheidungen und entsprechende Funktionen auszuführen gibt es das Modul 'System state analyses and decision making algorithm'(SSA-DMA). Das SSA-DMA Modul bietet die zwei Algorithmen, der Erste stellt den Systemdurchsatz der Anzahl der verwendeten VMs gegenüber um so Probleme mit der Hardware zu finden. Im Falle eines erkannten Problems wird ein Root-Cause-Analysis Algorithmus ausgeführt um den fehlerhaften Host auszutauschen.

Der zweite Algorithmus im SSA-DMA Modul ist der 'Workload Classification Algorithm'. Dieser sorgt dafür, dass ein System immer die optimale Anzahl an VMs und Ressourcen zur Verfügung hat.

Al-Sharif et al. beschreiben die Funktionsweise ihres Algorithmus so, dass er die Auslastung eines Systems als hoch oder niedrig einstuft. Mittels Messen der Utilization von CPU, RAM und Netzwerk und Abgleichen mit den Thresholds wird identifiziert, ob die Systemressourcen skaliert werden müssen. Das Skalieren hat den Zweck ein System durch aufwenden zusätzlicher Ressourcen die Auslastung in die sogenannte 'Safe-Zone' zu bringen. Die Safe-Zone beschreibt den Bereich zwischen 70%-80% Utilization der Ressourcen. Dieser Bereich wird als Bereich der idealen Auslastung beschrieben, da hier weder die Ressourcenlast zu Nahe am Leistungslimit liegt, noch so niedrig ist, dass zu viele Ressourcen verwendet werden, die im Zweifelsfall vermeidbare Kostenaufwände bedeuten können.

In dem Paper werden so in Summe 5 Zustände beschrieben in denen sich das System durch die beiden Algorithmen befinden kann, diese sind:

Safe zone Die Safe-Zone ist der Idealzustand, in dem das System sich befinden kann. Dieser Zustand ist erreicht, wenn sich die Utilization zwischen 70%-80% befindet. Hier ist der Utilization Level, der Energieverbrauch und die 'Quality of Service' im Optimum. Alle weiteren Zustände zielen darauf ab, den Zustand der Safe-Zone herzustellen.

Under-utilization (UU) Der Zustand 'Under-utilization' tritt ein, wenn das System eine geringe Auslastung seiner Ressourcen feststellt. Der Energieverbrauch ist in diesem Zustand hoch, während der Durchsatz gering ist.

Under-utilization with fault (UUF) Dieser Zustand tritt ein, wenn die Bedingungen eines UU-Zustandes erfüllt sind, außerdem aber auch noch ein fehlerhafter Zustand vorliegt(bspw. defekte Hardware)

Over-utilization (OU) Der Zustand der 'Over-utilization' tritt ein, wenn das System eine Überlastung seiner Ressourcen feststellt. Dieser Zustand wird bei einer Ressour-

cenauslastung über 80% erreicht und kann dafür sorgen, dass eingehende Workloads verzögert oder sogar verworfen werden.

Over-utilization with Fault (OUF) Dieser Zustand wird erreicht, wenn die Voraussetzungen des OU-Zustands erfüllt sind, außerdem aber auch noch ein fehlerhafter Zustand vorliegt (bspw. defekte Hardware)

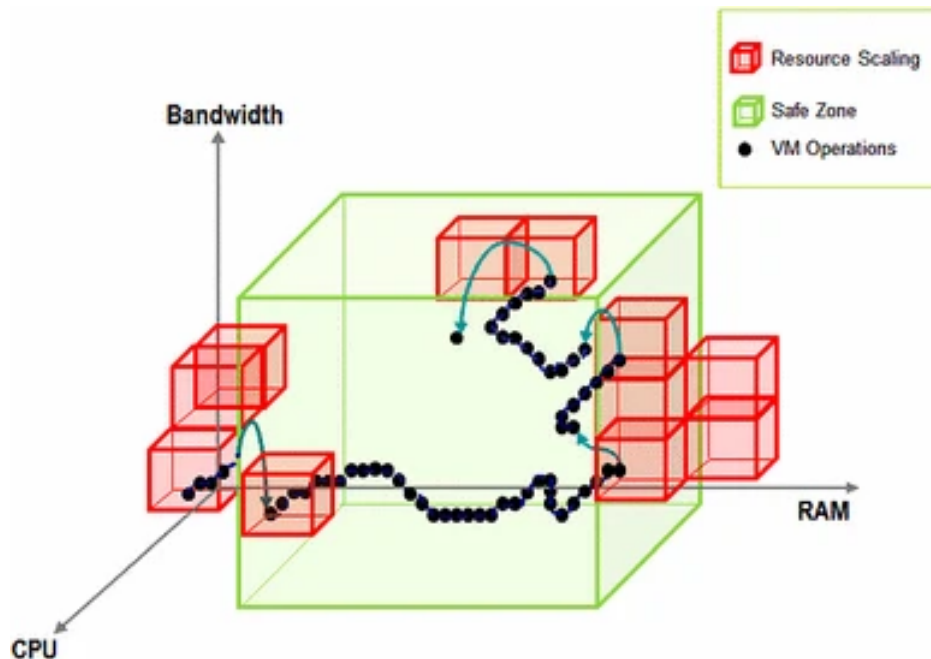


Abbildung 3: Safe-Zone and Resource Scaling Thresholds (from Paper ACCRS: autonomic based cloud computing resource scaling)

Die Grafik stellt die Safe-Zone als grünen Kasten, die Over- bzw. Under-Utilization als rote Kästchen, die an die Grenzen der Safe-Zone grenzen dar. Die X,Y und Z-Achsen, repräsentieren die Auslastung von RAM, Netzwerk und CPU. Anhand von Sprüngen, die der Graph der VM-Operations nach überschreiten der Safe-Zone Grenze zurück in die Safe-Zone macht, kann nachvollzogen werden, dass skaliert wurde und so die Last pro VM gesunken ist.

Die Skalierung in diesem Projekt orientiert sich an dem Verfahren, welches im Paper 'ACCRS: autonomic based cloud computing resource scaling' eingesetzt wird. Die Idee der Zustände Safe-Zone, Under-Utilization sowie Over-Utilization und deren Erkennung mittels statischer Regeln werden adaptiert und auf das Kubernetes-Konzept der Deployment-Skalierung übertragen, die Thresholds werden allerdings abgewandelt.

4 Datenaggregation

Die Datenaggregation, also das Erheben und Sammeln von Daten, ist zentraler Bestandteil des Projektes und der erste grundlegende Schritt. Das Ziel der Aggregation ist es Daten sog. Metriken über den Zustand eines Kubernetes-Clusters(siehe Kapitel Grundlagen/Kubernetes) zu erhalten. Kubernetes stellt selbst Metriken zur Verfügung, mittels eines passenden Tools werden diese ausgelesen und so für die weitere Verarbeitung nutzbar gemacht.

4.1 Toolauswahl

<evtl. hieraus eigenes Kapitel machen,für Prometheus, Grafana,(Loki)>

Das Monitoringtool, welches in diesem Projekt zur Datenaggregation genutzt wird, ist Prometheus. Prometheus wird als Quasi-Standard im Cloud-Native Umfeld und für das Monitoring von Kubernetes verwendet <Hierfür belegende Quelle finden>. Für dieses Projekt bietet es zentrale Features, welche die Umsetzung ermöglichen.

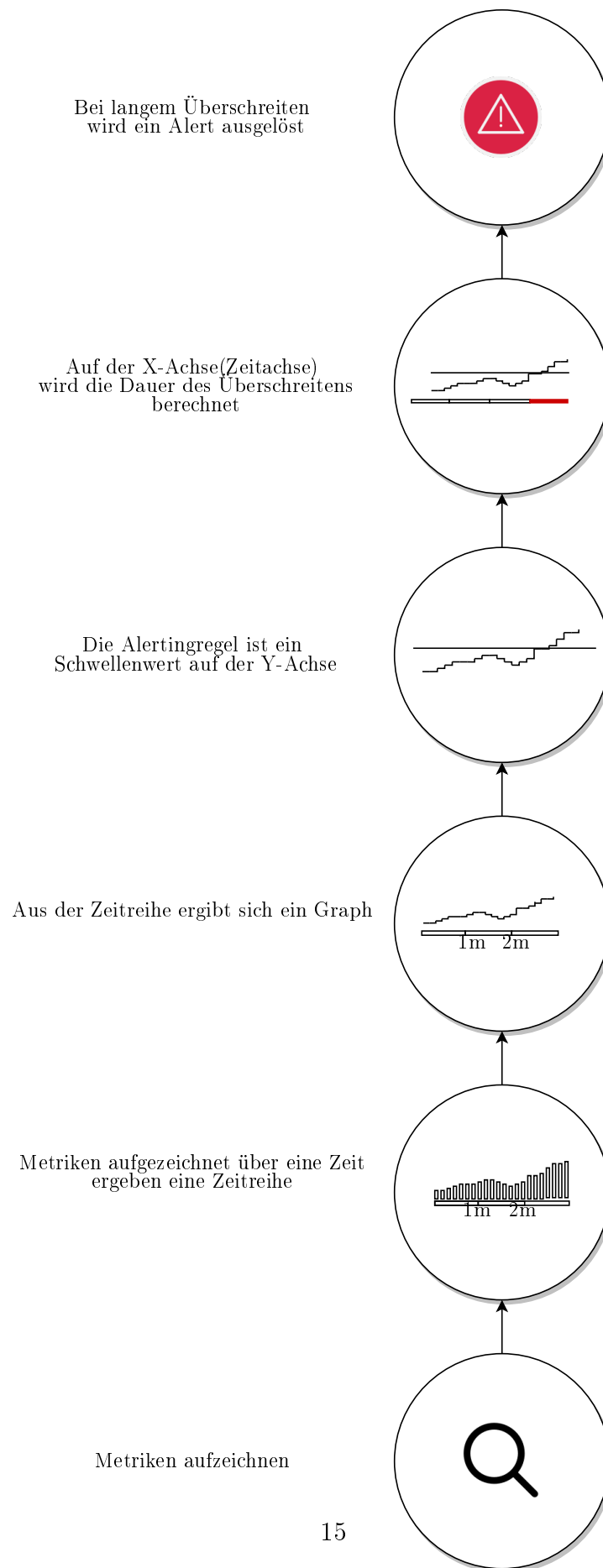
Automatische Service-Discovery Das Feature der automatische Service-Discovery ist sehr entscheidend, da sich die Services im Cluster häufig verändern und neue Services hinzugefügt werden. Mithilfe dieser Funktion werden Services automatisch gefunden, Metriken direkt erhoben und Regeln auf diese angewendet, ohne konfiguriert werden zu müssen.

Echtzeitmetriken Prometheus zeichnet Metriken in Echtzeit auf. Das ermöglicht es jederzeit den aktuellen Status des Clusters einzusehen und in Echtzeit darauf reagieren zu können.

Zeitreihen Prometheus zeichnet Metriken als Zeitreihen auf, was das Beobachten, Abfragen und die Definition von Regeln über Zeiträume möglich macht.

Regeln Regeln spielen eine zentrale Rolle in diesem Projekt. Mithilfe dieser lassen sich verschiedene Features von Prometheus regulieren, in diesem Projekt werden sie für das Regulieren von Alerts verwendet. Regeln werden über Thresholds und optional über Zeiträume von Metriken definiert, sodass bei Überschreiten eines Thresholds für eine bestimmte Zeit eine Regel als erfüllt gilt.

Alerting Mithilfe der Alerts ist es möglich anhand von Regeln Alarme zu versenden, die über den Zustand des Clusters informieren. Sie können an Ziele wie E-Mail-Adressen oder Chat-Tools wie Slack oder Telegram gesendet werden, aber auch an selbst definierte Endpunkte um sie dort zu empfangen oder sie möglicherweise weiterzuverarbeiten.



4.2 Grafana

Grafana ist das Tool, welches zur Visualisierung verwendet wird. Das Open-Source Tool bietet Möglichkeiten Metriken in Form von Graphen, Zählern, Histogrammen und vielem mehr über Zeiträume zu visualisieren. Des Weiteren können ganze 'Dashboards' erstellt werden, die den Überblick über das ganze Cluster auf einmal erlauben.

Aufgrund der bereits vorhandenen Prometheusintegration ist eine Verwendung bei bereits bestehendem Prometheus-Server einfach und innerhalb kurzer Zeit umzusetzen. Der Prometheus-Server, von welchem Grafana Metriken erhalten soll muss als Datenquelle mittels seiner Adresse konfiguriert werden, danach stehen alle Metriken, welche Prometheus kennt auch in Grafana zur Verfügung.

Um Metriken zu visualisieren, wird die aus Prometheus bereits bekannte Abfragesprache PromQL verwendet, was den Vorteil bietet, Abfragen die für Prometheus funktionieren, funktionieren ebenso in der Grafana Oberfläche zum Erstellen von Visualisierungen.

Da Grafana eines der gängigsten Tools ist und teilweise als Quasi-Standard zur Visualisierung von Metriken behandelt wird, findet eine kontinuierliche Weiterentwicklung statt und eine große Community bietet Unterstützung und Erfahrungen.

<hier Quelle Logz.io einfügen>

4.3 Datenquellen

Als Datenquelle dienen potenziell alle Ziele, die von Prometheus mittels der Target-Discovery innerhalb des Kubernetes-Cluster gefunden werden und Daten bereitstellen. Von besonderem Interesse für dieses Projekt sind hierbei alle system- und hardware-nahen Targets, wie der kubelet (siehe Kapitel Grundlagen) und cAdvisor in Pods und Containern. Prinzipiell eignen sich alle Pods als Datenquelle, Container die produktive Services bereitstellen weisen hierbei ein erhöhtes Interesse auf, gleiches gilt ebenso für Nodes

4.3.1 'USE'-Methode

Das Monitoring der Systemressourcen wird nach der 'USE'-Methode durchgeführt. Diese Methode wurde von Brendan Gregg, einem Netflix-Ingenieur im Bereich Cloud-Performance, mit Schwerpunkt auf das Monitoring von Systemressourcen entwickelt, weshalb sie sich für dieses Projekt eignet. Ziel der Methode ist es die Utilization, der Zeitdurchschnitt der Arbeit, die eine Ressource beschäftigt erledigt, die Saturation, die 'Zusatzarbeit' die eine Ressource verrichten muss, im Moment aber nicht leisten kann sowie die Error-Rate, die Häufigkeit in der Fehler auftreten von Systemkomponenten abzubilden. Systemressource, die es sich lohnt zu überwachen, sind unter anderem:

- CPUs: Sockets, Kerne, Hardware-Threads (virtuelle CPUs)
- RAM: Kapazität
- Netzwerkinterfaces

- Speicher(Festplatte,SSD): I/O, Kapazität
- Controller: Storage, Netzwerkkarten

Laut Gregg ist die Methode für Geräte besonders sinnvoll und effektiv, die bei erhöhter Utilization oder Saturation einen Performanceeinbruch erleiden.

Gregg will mit dieser Methode eine Möglichkeit bieten serverseitige Performanceprobleme schnell zu erkennen, deren Ursache zu finden um diese dann eliminieren zu können. Sein Ziel ist es mit dieser Methode eine einfache und unkomplizierte aber trotzdem schnelle und komplette Methode zu liefern. Laut Gregg hat sich diese Methode schon in vielen Bereichen mehrfach bewährt in verschiedenen Unternehmen, als Lehrmittel in Schulklassen oder vor Allem aktuell im Cloud-Computing Bereich.

Die Reihenfolge des Vorgehens wird in diesem Diagram dargestellt. Um die Geschwindigkeit zu optimieren sollten Errors als erstes überprüft werden, da sie typischerweise schneller und einfacher zu interpretieren sind. Die Utilization oder Saturation können in beliebiger Reihenfolge beobachtet werden.

<Quelle Brendan Gregg einfügen>

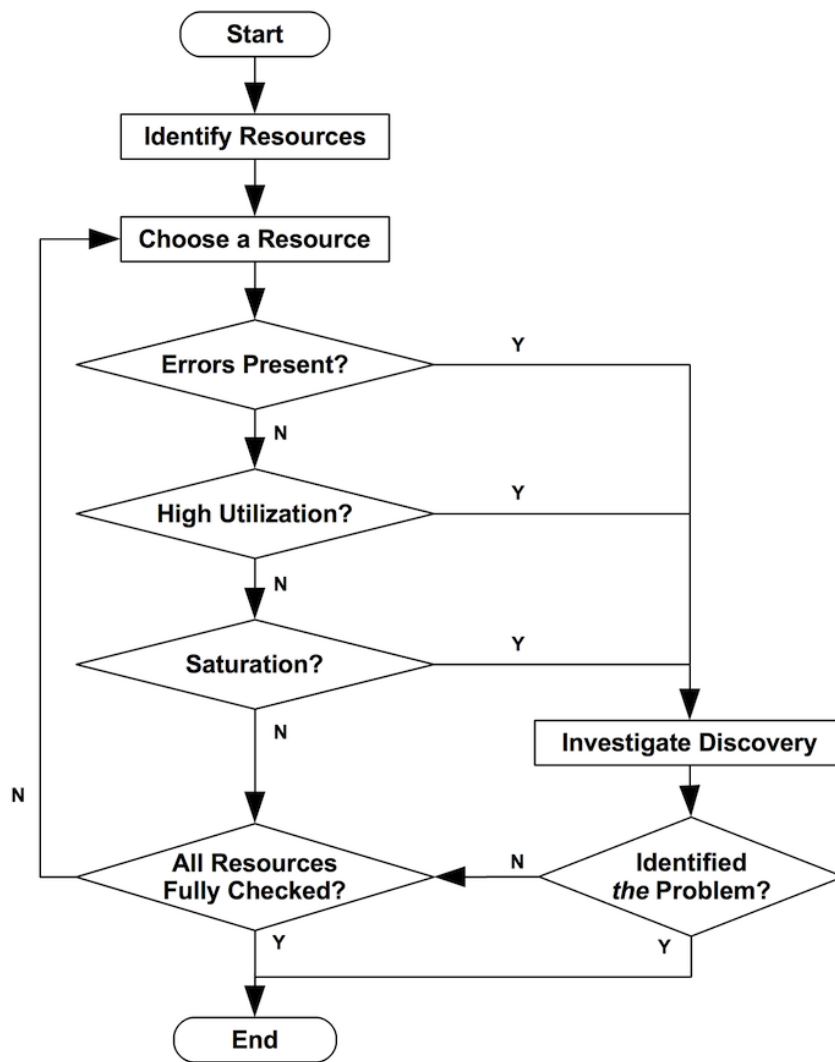


Abbildung 4: Darstellung der Vorgehensweise bei Anwendung der USE-Methode

4.3.2 Metriken

Die Metriken, mit denen die USE-Methode umgesetzt wird, werden aus den Systemmetriken, die durch Prometheus gesammelt werden, zusammengesetzt. Die Metriken Utilization und Saturation werden für CPU, RAM und Netzwerkauslastung folgendermaßen berechnet:

CPU :

Utilization:

Die CPU Utilization berechnet sich aus der Summe der Millisekunden, welche die CPU beschäftigt war, geteilt durch eine ganze Sekunde. So erhält man die Utilization pro Sekunde, welche über aufsummiert und der Mittelwert über den beobachteten Zeitraum gebildet wird, um so die durchschnittliche Utilization zu erhalten.

Summe der Zeit in Millisekunden, welche die CPU beschäftigt war:

J_{max} entspricht 1000 Millisekunden

$$U = \frac{1}{J_{max}} \sum_{j=1}^{J_{max}} u_j$$

Der gleitende Mittelwert der aufsummierten Zeit U über den Beobachteten Zeitraum von 5 Minuten:

I_{max} entspricht 300 Sekunden

$$\frac{1}{N} \sum_{i=1}^{I_{max}} U_{(t-i)}$$

Der gesamte Wert eines Utilization Fensters lässt sich dann über die zusammengesetzte Formel berechnen:

$I_{max} = 300; J_{max} = 1000$

$$\frac{1}{J_{max} * N} \sum_{i=1}^{I_{max}} \sum_{j=1}^{J_{max}} u_{(t-i)j}$$

Saturation:

Die CPU Saturation kann anhand des Ressourcenlimits berechnet werden. In UNIX basierten Betriebssystemen gibt es die Metrik 'load-average', welche die Anzahl der laufenden sowie wartenden Prozesse enthält. Sofern man diese durch die Anzahl der zur Verfügung stehenden CPU-Kerne teilt, erhält man eine Kennzahl für die Saturation. Diese aufsummiert und über die Zeit gemittelt, ergibt die durchschnittliche Saturation.

Ein weiterer Weg die Saturation zu berechnen ist das aufsummieren der Zeit in der ein Prozess 'throttelt' läuft. Dies funktioniert dann, wenn ein Prozess ein Ressourcenlimit besitzt, übersteigt er dieses wird der Prozess gedrosselt.

RAM :

Utilization:

Die RAM Utilization berechnet sich aus der Menge des vom System reservierten Speichers, der durch die Menge des gesamten Speichers geteilt wird. Beide stehen in UNIX basierten Betriebssystemen zur Verfügung, sodass die Rechnung zur RAM-Utilization $1 - \text{reservierterSpeicher} / \text{gesamterSpeicher}$ ist.

Saturation:

Die RAM Saturation kann am besten berechnet werden, wenn Ressourcenlimits existieren. Unter diesen Voraussetzungen kann die Saturation berechnet werden, indem die Summe des aktuell verwendeten Speichers durch das Ressourcenlimits geteilt wird.

Netzwerk-Bandbreite :

Utilization:

Um die Netzwerk Utilization zu messen werden die Bytes aufsummiert, die versendet sowie empfangen werden.

Saturation:

Die Netzwerk Saturation kann nur dann sinnvoll gemessen werden, wenn das Bandbreitenlimit bekannt ist. Sofern dies der Fall ist kann die Saturation berechnet werden indem die versendeten bzw. empfangenen Bytes durch das Bandbreitenlimit geteilt wird.

Sofern keine Information zum Limit zur Verfügung stehen, können die verworfenen Pakete (dropped Packages) als Indikator dafür dienen, wie hoch die Saturation ist.

5 Auswerten der Metriken

Das Auswerten der Metriken erfolgt für das Monitoring durch das Umsetzen der USE-Methode, beschrieben in Kapitel vier, statt.

5.1 Klassifizierung

Die Metriken werden in Node-Level sowie Pod- und Containerlevel Metriken klassifiziert. Nodelevelmetriken repräsentieren den Zustand eines ganzen Nodes, das bedeutet, dass die Metriken die Summe aller Pods, Container, Services, etc. des Nodes darstellen.

Podlevelmetriken hingegen repräsentieren den Status eines einzelnen Pods sowie des Containers bzw. den Containern die sich in diesem Pod befinden.

Die Containerlevelmetriken werden visualisiert um Zustände einzelner Pods und Container einsehen zu können, finden aber vor allem auch bei der Automatisierung von Aktionen ihren Nutzen. Für diese Metriken werden Regeln erstellt, die zur automatisierten Anwendung von Aktionen auf einen Pod oder Container dienen.

Nodelevelmetriken werden in diesem Projekt vor allem zum Überwachen des Zustandes des gesamten Nodes verwendet und dienen dazu kritische Zustände eines ganzen Nodes zu erkennen. Daher werden diese visualisiert, finden aber bei der Automatisierung und dem Alerting keine weitere Verwendung.

5.2 Logische Auswertung

Die logische Auswertung der Metriken findet an zwei Stellen in diesem Projekt Anwendung. Einmal zum Erstellen von Alerting-Regeln auf dem Prometheus-Server und einmal zur grafischen Darstellung der Metriken in der Grafana-Oberfläche. In beiden Fällen wird zur Auswertung die PromQL-Sprache verwendet. Zur logischen Auswertung werden die von Prometheus aggregierten Daten verwendet. Für das Monitoring werden die Schritte der USE-Methode Utilization, Saturation und Error-Rate jeweils für CPU, RAM sowie Netzwerk umgesetzt(siehe Kapitel vier, Metriken). Diese Metriken werden mittels eines 'Moving Average'-Filters mit einer Fensterbreite von fünf Minuten gefiltert und ausgewertet. Dies hat zur Folge, dass Ausreißer in der Last nicht schwer ins Gewicht fallen und so starke Schwankungen verhindert werden, was zur Verbesserung der Beobachtbarkeit beiträgt.

Der eingesetzte 'Moving-Average'-Filter sieht folgendermaßen aus:

$$m_{MA}^{(n)}(t) = \frac{1}{n} \sum_{i=0}^{n-1} x(t-i)$$

Der Zeitraum von fünf Minuten wurde so gewählt, da bei einem längerem Zeitraum, von beispielsweise 10 Minuten, die beobachtete Menge an Metriken größer wird und so die Reaktionszeiten auf Leistungsschwankungen durch langsamer ansteigende Mittelwerte länger werden. Der umgekehrte Fall tritt ein, wenn ein zu kurzer Zeitraum gewählt wird, hierdurch werden Mittelwertschwankungen möglicherweise überbewertet und wodurch Ausreißer eine zu große Bedeutung einnehmen. Der gewählte Wert von fünf Minuten basiert auf Eigenerfahrungen, die diesen Wert als gutes Mittel herausgefunden haben.

Für die logische Auswertung der Daten werden neben dem reinen Auslesen von Metriken auch Grenzwerte benötigt. Deshalb wurde hierbei keine Methode wie USE eingesetzt, aber es wurde sich an ihr orientiert.

Zu diesem Zweck ist in diesem Projekt die 'Metric-to-Action'-Methode entwickelt worden, welche die Ansätze der USE-Methode mit Grenzwerten verknüpft.

Diese Methode verwendet die Utilization-Metriken von USE, in diesem Projekt, zu CPU, RAM und Netzwerklast, welche mit Grenzwerten vereint werden.

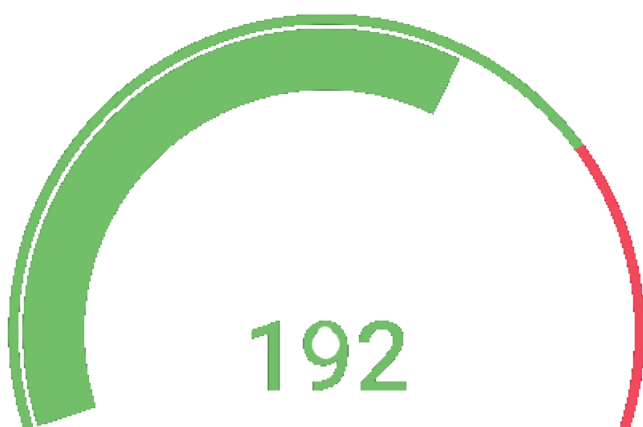
Das Ziel ist es über ein beliebiges Zeitfenster Metriken anhand von Grenzwerten zu beobachten, die zusammen einen binären Wert ergeben. Die Methode ist dann sinnvoll, wenn anhand von Metriken regelbasiert entschieden werden soll, wie auf diese reagiert werden kann.

5.3 Graphische Aufbereitung

Zur Visualisierung der Metriken stehen in der Grafana-Oberfläche verschiedene Möglichkeiten zur Verfügung. Da es das Ziel ist möglichst auf einen Blick, den Systemressourcenzustand des ganzen Clusters sehen zu können, wird ein Dashboard mit allen USE-Metriken erstellt. Die Metriken können als Graph, Zähler, Balkendiagramm oder Heat-Map dargestellt werden.

Da schnell erkennbar sein soll, an welcher Stelle ungewöhnliche oder kritische Werte auftreten, werden die Zähler als Visualisierungsmethode für alle Metriken gewählt. Hierbei steht pro Metrik ein eigener Zähler zur Verfügung, womit sich der Zustand jeder Resource einzeln einsehen lässt.

Für die Counter werden grüne Bereiche für unkritische Auslastung und rote Bereiche für kritische Auslastung mit einen Grenzwert von 75% eingerichtet, um kritische Zustände hervorzuheben.



6 Automatisierte Aktionen

Das Kernthema dieser Arbeit ist die Entwicklung eines Systems, welches das Ausführen automatisierter Aktionen auf ein Kubernetes-Cluster ermöglicht. Mithilfe der Komponenten und Tools, Prometheus und dessen Alert-Verwaltungstool 'Alertmanager' (siehe Grundlagenkapitel) sowie einer Eigenentwicklung, genannt 'Alert-Action-Manager', welcher Alerts empfängt und auf diese mit einer passenden automatisierten Aktion reagiert, wird dieses Vorhaben in diesem Kapitel umgesetzt. Mittels automatischer Skalierung und Anomalie-Detection wird die Funktion der Architektur und Komponenten nachgewiesen.

6.1 Architektur des Systems

Das System, welches die automatisierten Aktionen möglich macht, besteht aus mehreren, voneinander abhängigen Komponenten. Das Konzept nutzt das Alerting-Feature von Prometheus und setzt mithilfe dessen automatisierte Aktionen um.

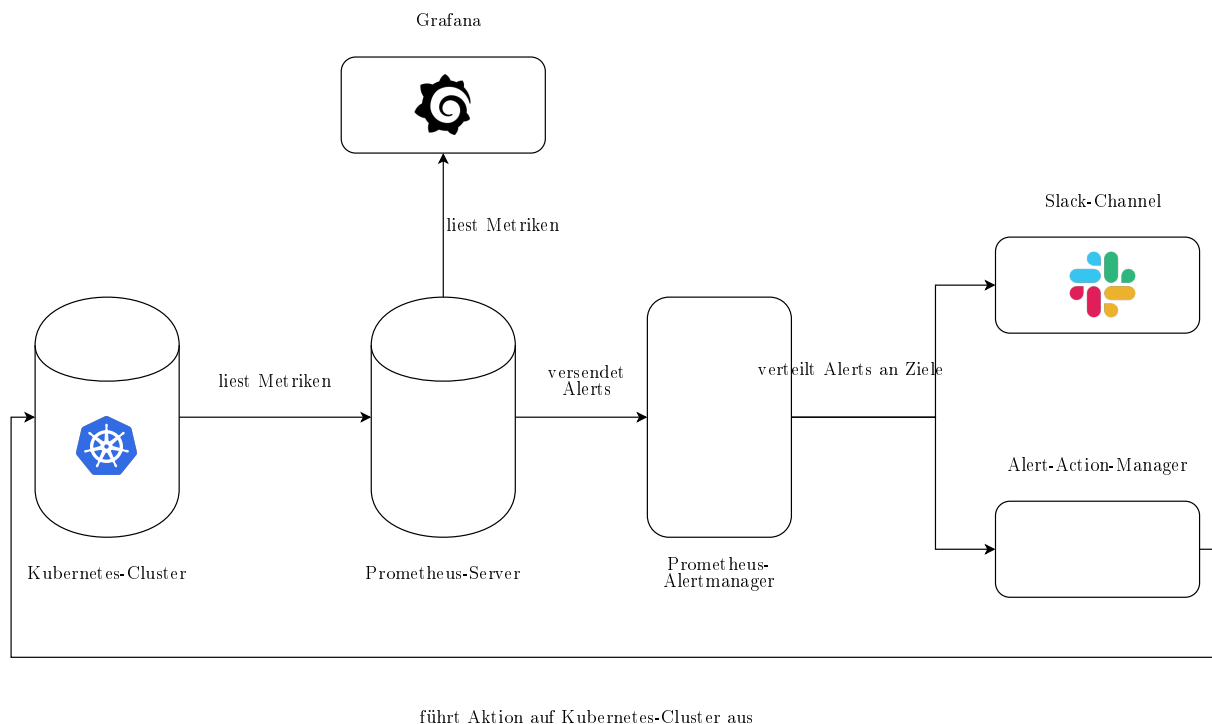


Abbildung 5: Aufbau der Architektur (eigene Zeichnung)

Die Architektur, welche in diesem Projekt entwickelt wurde, besteht aus sechs Komponenten, vier davon, Grafana, Prometheus Server und Alertmanager sowie der Alert-Action-Manager, wurden im Laufe des Projekts erstellt sowie konfiguriert bzw. entwickelt, zwei davon, das Kubernetes-Cluster, auf dem dieses Projekt basiert, sowie Slack, bestanden schon vorher und wurden in das Projekt miteinbezogen. Die vier für Komponenten Grafana, Prometheus Server und Alertmanager sowie der Alert-Action-Manager laufen selbst als Deployment im Kubernetes-Cluster.

Die Architektur nach Bestandteilen erklärt:

Kubernetes-Cluster: Das Kubernetes-Cluster, auf dem Webservices sowie andere Backendsoftware läuft, ist das Ziel des Monitorings sowie der automatisierten Aktionen. Kubernetes stellt Metriken von sich aus bereit, welche ausgelesen und verarbeitet werden. Die Architektur läuft losgelöst vom restlichen Cluster und dessen Software, sodass durch dieses Projekt keine zusätzliche Anpassung oder Konfiguration von bestehender Software benötigt wird.

Prometheus-Server: Der Prometheus-Server bildet das Herzstück der Architektur. Er aggregiert die durch das Kubernetes-Cluster bereitgestellten Metriken und wendet Regeln, wie das Alerting, auf diese an. Außerdem macht er diese Metriken, entweder durch das eigene Web-UI oder durch weitere Tools, mittels der Abfragesprache PromQL wieder abrufbar.

Grafana: Grafana ist das in dieser Architektur eingesetzte Visualisierungstool, mit dem Administrator und andere Benutzer den Zustand des Clusters graphisch aufbereitet einsehen können. Das Tool bietet gegenüber der nativen Prometheus Web-UI deutlich bessere Darstellungsmöglichkeiten (siehe hierzu Kapitel Auswertung der Metriken/Graphische Aufbereitung). Grafana ruft Metriken direkt vom Prometheus-Server ab und unterstützt auch dessen Query-Language PromQL.

Prometheus-Alertmanager: Der Prometheus-Alertmanager empfängt Alerts direkt durch den Prometheus-Server. Seine Aufgabe ist es empfangene Alerts an seine Ziele 'Alert-Action-Manager' sowie 'Slack' und darauf eingerichteten Alert-Channel weiterzuverteilen.

Alert-Action-Manager: Der 'Alert-Action-Manager' empfängt Nachrichten durch den Prometheus-Alertmanager. Dieses Tool ist eine komplette Eigenentwicklung und dafür zuständig, dass auf empfangene Alerts mit entsprechenden Aktionen auf das Kubernetes-Cluster reagiert wird.

Slack: Slack ist die Benachrichtigungsplattform für Administratoren und interessierte Benutzer. Hier wurde ein eigener Channel für den Zweck der Benachrichtigung beim Auftreten von Alerts eingerichtet.

Auf wichtige Komponenten wird hier nochmal gesondert eingegangen.

6.1.1 Prometheus

Wie bereits beschrieben, bildet Prometheus die zentrale Komponente in diesem Projekt und dessen Architektur. Auf dem Prometheus-Server werden alle Regeln definiert und konfiguriert, die für das Alerting benötigt werden und damit Aktionen auslösen können. Außerdem können auf dem Server die Zeiten konfiguriert werden, in deren Abständen der Server Metriken scraped, Regeln prüft und Alerts sendet. Diese Zeiten bestimmen maßgeblich in welcher Geschwindigkeit die Aktionen ausgeführt werden.

Zusätzlich zum zentralen Server besitzt Prometheus auch Komponenten, die direkt im Kubernetes-Cluster auf den Nodes laufen und dafür sorgen, dass von Kubernetes mittels kubelet und cAdvisor (siehe Kapitel Grundlagen/Kubernetes) veröffentlichte Metriken für den Server bereitgestellt werden. Diese Aufgabe übernehmen die Exporter, welche im Cluster installiert und konfiguriert sind.

Der Prometheus-Server, wie auch der Alertmanager, stellen ein eigenes Deployment dar und laufen als Pod im Kubernetes-Cluster. Sie werden in der für Kubernetes optimierten Variante über den Kubernetes Paket-Manager 'Helm' installiert und konfigu-

riert. Dies hat den Vorteil, dass für die grundsätzliche Konfiguration im Kubernetes-Cluster deutlich weniger Ressourcen und Zeit in Anspruch genommen werden müssen und die Vollständigkeit und Fehlerwahrscheinlichkeit verbessert werden, da benötigte Exporter direkt mit installiert und automatisch konfiguriert werden.

Dieser Ansatz bedingt, dass Konfigurationen nicht mehr direkt in Prometheus selbst, sondern über den Paket-Manager konfiguriert werden, was eine gewisse Kenntnis der Funktion sogenannter 'Helm-Charts' voraussetzt.

6.1.2 Alertmanager

Der Prometheus-Alertmanager, der in der Architektur als 'Verteiler' von Alerts dient, ermöglicht die grundsätzliche Kommunikation zwischen Prometheus und externen Tools. Die Möglichkeit mittels des Nachrichtensystems neben vorkonfigurierten Drittanbieter-tools wie Slack auch eigene sog. Webhooks zu adressieren, erlaubt die Möglichkeit einen eigenen HTTP-Endpunkt als Nachrichtenziel zu verwenden. Da sich für einen Alert mehrere Ziele festlegen lassen, kann so zu jeder automatischen Aktion auch eine Warnung an den für Alerts konfigurierten Slack-Channel gesendet werden. So kann in zukünftigen Alerts auch zwischen rein informativem Alert, der nur an Slack gesendet wird und einem Alert der eine Aktion zur Folge hat unterschieden werden und diese entsprechend konfiguriert.

6.1.3 Regeln

Die Regeln, die zum Aktivieren der Alerts verwendet werden, bestehen aus einem Label, einem Ausdruck, der Metriken abgleicht und einem Erfüllungszeitrahmen. Der Ausdruck ist, wie die Queries zur Visualisierung auch, in der PromQL-Sprache geschrieben. Ein Ausdruck muss immer als Vergleich geschrieben werden und somit einen BOOL-Wert repräsentieren können, also erfüllt(true) oder nicht erfüllt(false). Jeder Ausdruck besitzt einen Zeitrahmen über den er erfüllt sein muss um einen Alert auszulösen, womit vorzeitiges oder zu häufiges Alarmieren eingeschränkt oder verhindert werden kann. Um mehrere Bedingungen in einem Ausdruck zu verknüpfen kann auf logische Operatoren zurückgegriffen werden, diese sind AND, OR und UNLESS(Komplement). Zum Inkludieren oder Exkludieren einzelner Metrikquellen, können mittels der Anwendung von regulären Ausdrücken auf die Namen der Quellen Filter für diese erstellt werden.

Die grobe Struktur eines Alerts sieht folgendermaßen aus:

Label:	"Musterlabel"
Regel:	$a > b$
Offsetzeit:	5 Minuten
Anmerkungen:	In Musterlabel ist a kleiner b

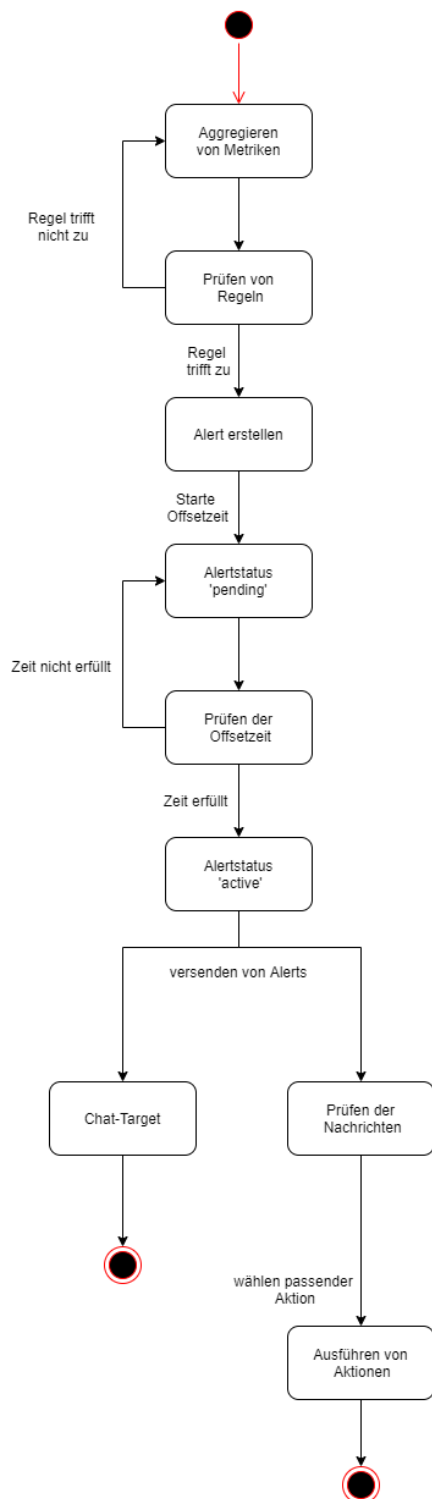
Ein Alert muss identifiziert werden können, hierfür wird das Label verwendet, das den Namen eindeutig repräsentiert.

Alerts sollen nur dann ausgelöst werden, wenn ein entsprechender Zustand erfüllt ist.

Um diesen Zustand zu spezifizieren wird die Vergleichsregel zum bemessen gebraucht. Um kurzzeitige Ausreißer im Zustand nicht überzubewerten, muss dieser eine gewisse Dauer vorliegen. Diese Dauer, auch Offsetzeit genannt, wird zusätzlich zur Regel spezifiziert.

Je nach Alert oder auszuführender Aktion werden Zusatzinformationen benötigt. Diese können als Anmerkung beigefügt werden. Unter anderem kann hier beschrieben werden, wie eine Alert-Nachricht als Chatnachricht aussehen soll.

Der Ablauf des Konzepts ist eine Kette von Aktionen, bei der ein Schritt den nächsten auslöst. Die Kette der Schritte, welche zum Auslösen und Ausführen einer Aktion durchlaufen wird, ist hier beschrieben:



Schritt 1

Im ersten Schritt aggregiert der Prometheus-Server aus dem Kubernetes-Cluster Systemressourcen bezogene Metriken von Pods und Containern.

Schritt 2

In Schritt zwei prüft der Prometheus-Server anhand konfigurierter Alerting-Regeln, ob ein Alert ausgelöst werden soll. Falls dies der Fall ist, versendet der Server einen Alert an den Prometheus-Alertmanager, zuerst mit dem Status 'pending', nach Ablauf der Offset-Zeit mit dem Status 'active' (siehe Kapitel Grundlagen/Alerting).

Schritt 3

Der Alertmanager versendet, sobald ein Alert den Status 'active' erhält, an alle konfigurierten Ziele eine Alert-Nachricht. Diese sind ein Channel des Chatprogramms 'Slack', der alleine für Alert-Nachrichten zur Verfügung steht und den Zweck besitzt Administratoren und andere Interessierte zu informieren, sowie das in diesem Projekt entwickelte Tool 'Alert-Action-Manager', welches den Zweck besitzt anhand eintreffender Alert-Nachrichten Aktionen auf das Kubernetes-Cluster auszuführen.

Schritt 4:

Beim Eintreffen von Alerts prüft der 'Alert-Action-Manager' die eingehenden Nachrichten auf deren Inhalt um festzustellen, welche Aktion ausgeführt werden soll.

Schritt 5:

Nachdem eine Nachricht durch den Alert-Action-Manager geprüft wurde, wird eine entsprechende Aktion auf das Kubernetes-Cluster ausgeführt.

7 Alert-Action-Manager

Der Alert-Action-Manager (hier abgekürzt: AAM) ist eine Entwicklung für die Architektur dieses Projekts. Dieser stellt den Part dar, der Aktionen aus empfangenen Alerts ableitet und diese dann auf das Kubernetes-Cluster ausführt.

7.0.1 Aufgabe

Da Alerts mit einer kritischen Situation im Kubernetes-Cluster einhergehen, auf die so schnell wie möglich reagiert werden muss, wird ein automatisiertes Hilfsmittel zur schnellstmöglichen Reaktion benötigt. Dieses Hilfsmittel stellt der Alert-Action-Manager dar. Seine Aufgabe ist es zu bestimmen, welche Aktion aus einem ausgelösten Alert folgen muss um die kritische Situation zu entschärfen.

7.0.2 Anforderungen

An den AAM sind mehrere Anforderungen geknüpft, die er erfüllen muss, um seine Aufgabe adäquat umsetzen zu können.

- von Prometheus versendete Alerts müssen empfangen werden können
- der Manager muss mit dem gegebenen Format JSON umgehen können
- ein fest konfigurierbarer HTTP-Endpunkt wird benötigt
- der Manager muss aus den gegebenen Informationen eines Alerts Aktionen schließen können
- um Aktionen auf das Cluster ausführen zu können, muss der Manager eine Schnittstelle zu Kubernetes besitzen

7.1 Wahl der Sprache

Die Wahl der Programmiersprache für die 'Alert-Action-Manager'- sowie die Evaluations-Komponente fiel auf die von Google entwickelte Open-Source Sprache Go(-lang).

Google beschreibt die Sprache selbst 'Go is a compiled, concurrent, garbage-collected, statically typed language developed at Google. It is an open source project: Google imports the public repository rather than the other way around.' Go findet vor allem in der Web-Entwicklung eine zunehmend starke Verbreitung, im Ranking der Website tiobe.com beispielsweise, steht Go, Stand 19.07.2020, auf Platz 12 der meist verwendeten Programmiersprachen <Quelle für Verbreitung und Wachstum einfügen>, wodurch es eine große und hilfreiche Community, viele Erfahrungen in einschlägigen Foren sowie eine stete Weiterentwicklung bietet.

Vorteile der Sprache die auch in diesem Projekt hilfreich waren und so zur Entscheidung dieser Sprachwahl beigetragen haben, sind folgende:

Kompilierung :

Go ist im Gegensatz zu Sprachen JavaScript oder PHP eine kompilierte Sprache. Dadurch bietet Go einen Performancevorteil, den kompilierte Sprachen gegenüber interpretierten typischerweise haben. Des Weiteren wirkt sich die Kompilation positiv auf die Fehlerrate aus, da 'compile time errors' von vornherein durch das Kompilieren detektiert werden.

Typsicherheit :

Die Sprache ist statisch typisiert, was bedeutet, dass Typen zur Kompilierzeit überprüft werden und so Fehler durch falsche Typen vermieden werden.

Speichermanagement :

Go hat eine integrierte Garbage-Collection, was das Speichermanagement vereinfacht und Fehler in dessen Zusammenhang verhindert. Trotzdem bietet die Sprache die Möglichkeit mit Pointern zu arbeiten.

Zahlreiche Packages, einfache Integration :

Go bietet die Möglichkeit Libraries, in Go Packages genannt, mittels ihres Pfads in ein Projekt zu integrieren. Die Standard JSON-Bibliothek wird beispielsweise mittels des Statements

```
import "encoding/json"
```

eingebunden. Mittels des Package-Names können dann Elemente aus dem Package verwendet werden.

```
var dec = json.NewDecoder(reader)
```

Ein großer Vorteil ist, dass das selbe Prinzip ebenso bei externen Packages, beispielsweise von Github, funktioniert.

Mit dem mitgelieferten Package-Downloader 'go get' kann ein Package heruntergeladen und installiert werden und ist so direkt einsatzbereit. Dadurch stehen eine Vielzahl an Packages auf Github und anderen Plattformen bereit, die ohne großen Aufwand verwendet werden können.

C-Integration(CGO) :

Mit CGO bietet die Sprache eine Integration der kompletten Sprache C bereit. Dies erlaubt es C-Libraries zu importieren und zu verwenden oder C-Funktionen innerhalb eines Go-Programms zu benutzen. Dies ist beispielsweise praktisch, wenn

trotz der automatischen Speicherverwaltung Speicher manuell allokiert werden soll. So kann mittels CGO die C-Funktion 'malloc()' zum reservieren des Speichers verwendet werden.

Go bietet zahlreiche weitere nützliche Features, für weitere Informationen dazu wird die offizielle Dokumentation und FAQ empfohlen:

https://golang.org/doc/faq#What_is_the_purpose_of_the_project

7.2 Funktionen

Um die beschriebenen Anforderungen umzusetzen, besitzt der AAM diese grundlegenden Funktionen:

HTTP-Endpunkte: Für den AAM wird ein eigener Kubernetes-Service bereitgestellt, der als DNS-Service fungiert. Um Alerts zu empfangen wird ein HTTP-POST Endpunkt bereitgestellt. Der Prometheus-Alertmanager wird so konfiguriert, dass er Alerts an den Kubernetes-Service des AAM sendet. Die Requests werden im JSON-Format empfangen, welches auch das Standardformat ist, in dem die Alerts versendet werden. Weitere Endpunkte sind der 'health' Pfad, über den sich überprüfen lässt, ob der AAM erreichbar ist und der 'podstatus' Pfad, über den sich der Podstatus des AAM und allen Pods in dessen Namespace, in dem sich alle Komponenten der Architektur befinden, einsehen und der korrekte Zugriff des Service auf das 'kubectl' überprüfen lässt.

Alert-Analyse: Alerts die über den HTTP-POST Endpunkt empfangen wurden, werden vom AAM analysiert und eingeordnet. Aus dem POST-Request wird zuerst das Label geparkt, was die Einordnung des Alerts ermöglicht. Dieses wird mit den bekannten Labels verglichen, falls es nicht bekannt ist, wird der Alert verworfen, sofern es bekannt ist wird die Funktion des passenden Labels ausgeführt. Der erste Schritt einer Funktion ist das Analysieren des zugehörigen Deployments, also der Pod oder das Deployment auf welches eine Aktion ausgeführt werden soll. Hierzu wird der Name eines Pods verwendet oder vorher der Deployment-Name aus dem Namen des Pods geparkt. Nach erfolgreicher Analyse wird die passende Aktion ausgeführt.

Ausführen von Aktionen: Das Ausführen von Aktionen wird mittels 'kubectl' durchgeführt. Die erforderlichen Daten die zum Durchführen einer kubectl-Aktion gebraucht werden, wurden vor dem Ausführen durch die Analyse ermittelt und nun in die Aktion eingebaut.

Zusatzfunktionen: Es werden zum Ausführen einiger Aktionen zusätzliche Funktionen benötigt. Zum Skalieren der Replicas eines Deployments (beschrieben im folgenden Unterkapitel), ist die Anzahl der Replicas zum Zeitpunkt des Ausführens der Aktion erforderlich. Hierfür besitzt der AAM eine Funktion, welche die Replikaanzahl zu einem Deployment analysieren kann.

7.2.1 Erweiterung

Sofern zusätzliche Aktionen in der Infrastruktur benötigt werden, ist die Erweiterung mit relativ geringem Aufwand möglich. Es muss einerseits eine neue Regel bzw. neue Regeln mit neuem Label in Prometheus definiert werden sowie die zugehörige Aktion im AAM.

7.3 Aktionen

Die Aktionen, welche durch den Alert-Action-Manager automatisiert auf das Kubernetes-Cluster ausgeführt werden, sind in diesem Abschnitt erklärt. Das Ausführen dieser Aktionen wird anhand von Regeln mit statischen Grenzwerten sog. Thresholds entschieden. Die beiden Aktionen, welche in diesem Projekt eingesetzt werden sind die Skalierung der Pod-Replikationen eines Deployments und die Anomalie-Detection innerhalb eines Pods bzw. Containers.

7.3.1 Skalierung

Die Aktion der Skalierung ermöglicht das Hochskalieren, also das Erweitern eines Deployments um eine höhere Anzahl an Replikationen (Replicas) seiner Pods, sowie die Gegenaktion, das Herunterskalieren eines Deployments.

Diese Aktion wird durch zwei Regeln realisiert:

PodLowResource Diese Regel wird ausgelöst, sobald in einem Pod eine hohe Last der Ressourcen CPU-Utilization oder RAM-Utilization festgestellt wird und dies in Verbindung mit einer hohen Nutzerlast, also einer ausreichend hohen Netzwerkbandbreiten-Utilization steht. Die Thresholds für CPU- und RAM-Utilization liegen bei 75%, die Auslastung der Netzwerkbandbreite muss bei mindestens einem Megabit pro Sekunde liegen. Diese Last muss für 10 Sekunden bestehen um die die 'PodLowResource'-Regel und damit den Alert zu aktivieren.

PodHighResource 'PodHighResource' ist die Gegenaktion zu 'PodLowResource' und wird dann ausgelöst, wenn eine niedrige Ressourcenlast in einem Deployment detektiert wird. Die geprüften Metriken sind die CPU-Utilization sowie die RAM-Utilization. Diese müssen über ein Zeitfenster von 10 Sekunden einen Wert von 25% Auslastung oder niedriger aufweisen um diese Regel zu erfüllen.

Nach Auslösen einer der Alerts und Aktivieren der Aktion im 'Alert-Action-Manager' setzt dieser die Aktion folgendermaßen um:

Für diese Aktion wird einerseits der Name des Deployments benötigt sowie die Anzahl der Pod-Replikation die zum Ausführungszeitpunkt der Regel existieren.

Der Name betroffener Pods wird mit einem Alert mitgeschickt, durch entfernen des Pod-Tags kann daraus der Name des Deployments ermittelt werden.

Die Anzahl der Replikationen wird durch eine dafür bereitgestellte Methode aus dem Cluster ermittelt und zurückgeliefert. Sobald die benötigten Informationen vorhanden

sind, wird aus Ihnen die passende kubectl-Aktion erstellt.

Dieser sieht für diese Regel schlussendlich folgendermaßen aus:

Die Variable 'deployment_name' enthält den analysierten Deploymentname, die Variable actualCount enthält die Anzahl der aktuellen Deployments. Für das Hochskalieren wird wie im Beispiel gezeigt die Replicaanzahl +1 verwendet zum Herunterskalieren die Replicaanzahl -1.

```
kubectl scale deployment deployment_name --replicas= actualCount+1
```

7.3.2 Anomalie-Detection

Die Anomalie-Detection Aktion funktioniert, ähnlich wie die Skalierungsaktion, mit Hilfe von Thresholds. Diese wird dann ausgeführt, wenn hohe Ressourcenlast scheinbar unbegründet, das bedeutet ohne hohe Nutzerinteraktion von außen, auftritt. Diese Aktion wird durch eine Regel realisiert:

PodAnomalyBehaviour Die Thresholds werden so gesetzt, dass entweder die RAM-Auslastung, die CPU-Auslastung oder beide, einen Wert von 75% Last oder mehr aufweisen müssen, bei einer Netzwerkbandbreitenlast von unter einem Megabit. Dieser Zustand muss 10 Sekunden bestehen, um diese Regel auszulösen.

Nach Auslösen des 'PodAnomalyBehaviour' Alerts wird die zugehörige Aktion folgendermaßen ausgeführt:

Für die Aktion wird der zugehörige Deploymentname des anormalen Pods benötigt, dieser wird aus dem im Alert mitgesendeten Podnamen analysiert. Sobald der Name des Deployments bekannt ist wird die zugehörige kubectl-Aktion ausgeführt. Diese nennt sich 'rollout restart' und sorgt dafür, dass ein Deployment unterbrechungsfrei neu gestartet wird, was den Vorteil bietet für den Rest des Clusters sowie auch für Nutzer nicht bemerkbar zu sein. Dies bedeutet, dass zuerst ein neues Deployment gestartet wird und sobald dieses arbeitsbereit ist, wird das alte Deployment heruntergefahren. Die resultierende Aktion sieht folgendermaßen aus:

Die Variable 'deployment_name' enthält den analysierten Deploymentname.

```
kubectl rollout restart deployment deployment_name
```

7.3.3 Metriken

7.3.4 Grenzwerte

8 Evaluation

8.1 Messaufbau

8.2 Regeln

8.3 Grenzwerte

8.4 Komponenten und Architektur

9 Diskussion

10 Fazit und Ausblick

Literatur

- [1] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and running : dive into the future of infrastructure*. O'Reilly Media, Incorporated, Sebastopol, CA, second edition edition, 2019.
- [2] Kubernetes komponenten, 30.05.2020.
- [3] Using minikube to create a cluster, 16.03.2020.
- [4] Github-User: dashpole. google/cadvisor, 05.07.2020.