https://www.overleaf.com/project/63ac82b9b7e2e97ef8e7eaff

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik–

Performance Testen einer Kubernetes Umgebung unter Betrachtung verschiedener Metriken

Abschlussarbeit zum absolvieren der zweiten Praxisphase

vorgelegt von

Muhammet Sen

Matrikelnummer: 769225

Referent	:	Herr Kai Naschinski
Korreferent	:	

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 21.03.2023

Muhammet Sen

ZUSAMMENFASSUNG

Viele Web-Anwendungen, APIs oder andere Services, welche dauerhaft vielen Anfragen ausgesetzt sind, werden häufig in einer containerisierten Umgebung aufgesetzt, damit die Anwendungen skalierbar und weniger anfällig für Ausfälle sind. Zwar sind die Anwendungen, welche beispielsweise im Kontext einer Kubernetes-Umgebung laufen, skalierbarer, jedoch können Kubernetes-Cluster auch ein Kapazitätslimit haben und unter bestimmten Umständen sogar ausfallen oder nicht anständig nutzbar sein. Solche Ausfälle sind vor allem aus finanzieller Sicht nicht wünschenswert, da Ausfälle beispielsweise zu Datenverlust führen können.
Um die Qualität eines Clusters zu sichern und solche Ausfälle zu verhindern, bieten sich verschiedene Tests an, um das Verhalten des Clusters in bestimmten Situationen oder auch das Kapazitätslimit des Clusters unter extremer Last zu ermitteln. Die Grenzen eines Clusters zu kennen, ermöglicht es präventiv Maßnahmen zu ergreifen, um besagte Ausfälle zu verhindern. In dieser wissenschaftlichen Arbeit wird eine Kubernetes-Umgebung auf Grundlage verschiedener Metriken getestet. Die betroffenen Metriken werden auf einer grafischen Oberfläche überwacht und das Verhalten des Clusters anschließend gedeutet. Hauptsächlich werden Last-Tests mithilfe von verschiedenen Tools und Skripten durchgeführt.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface

OOMkill Out Of Memory Kill

CPU Core Processing Unit

RAM Random Access Memory

MTTF Mean Time To Failure

MTBF Mean Time Between Failure

MTTR Mean Time To Recovery

Teil I

THESIS

1 EINLEITUNG

Kubernetes-Cluster werden inzwischen von vielen nennenswerten Konzernen verwendet, um ihre Online-Dienste in einer skalierbaren Umgebung zu betreiben. Kubernetes-Cluster haben viele verschiedene Metriken, welche als Indikatoren für die Qualität und Gesundheit einer Umgebung verwendet werden können. Um zu ermitteln, ob eine Cluster-Umgebung für einen bestimmten Anwendungsfall ausreicht, bietet sich das Durchführen verschiedener Tests an, um präventiv gegen Ausfälle vorgehen zu können.

1.1 MOTIVATION

Die Präsenz von Microservices in containerisierten Umgebungen nahm in den letzten Jahren zu. Große Unternehmen wie Google, Spotify oder Pinterest verwenden Kubernetes-Cluster, um ihre Anwendungen effizient zu betreiben. Denn auch nur ein kurzer Ausfall dieser Anwendungen würde einer großen Anzahl von Nutzern auffallen und bei den betroffenen Unternehmen für finanzielle Verluste sorgen. Um solche Verluste zu vermeiden, ist es wichtig, die Grenzen des eigenen Clusters zu kennen. Das Testen der verfügbaren Leistung ist daher eine wichtige Qualitätssicherungsmaßnahme, welche bei korrekter Durchführung einen guten Einblick in die Kapazitäten einer Kubernetes-Anwendung gibt.

1.2 ZIEL DER ARBEIT

Ziel der Arbeit ist es, den Testvorgang einer Kubernetes-Umgebung aufzuzeigen. Hierbei geht es primär darum, den Leser darüber zu informieren, welche verschiedenen Metriken wichtig für die Qualität eines Clusters sind und welche Tests uns Auskunft über diese Metriken geben können. Außerdem sollen die Testergebnisse ausgewertet und gedeutet werden. Das Testen mithilfe verschiedener Tools soll dabei unkompliziert und einfach wiederholbar sein.

1.3 GLIEDERUNG

Am Anfang werden die verschiedenen Metriken erklärt und die für diese Arbeit wichtigen Testing-Typen aufgezeigt. Dieser Teil soll die Grundlagen aufarbeiten, welche für den späteren Teil der Arbeit benötigt werden. Daraufhin wird die Vorgehensweisen aufgegriffen, die für das Testen eines Clusters verwendet werden. Danach wird die zu testende Umgebung vorgestellt und getestet. Im Anschluss werden die Testergebnisse ausgewertet und gedeutet. Zum Schluss folgt das Fazit dieser Arbeit.

1.4 ABGRENZUNG

Diese Arbeit beschäftigt sich ausschließlich mit dem Last-Testen und Stress-Testen von Kubernetes-Anwendungen sowie Metriken, welche für die Auswertung jener Tests benötigt werden. Diese Arbeit befasst sich nicht mit den Kubernetes-Grundlagen sowie dem Aufsetzen von Anwendungen auf einer Kubernetes-Umgebung. Des Weiteren werden Sicherheitsaspekte einer Kubernetes Umgebung ebenfalls nicht behandelt.

1.5 VORRAUSSETZUNGEN

Diese Arbeit setzt grundlegendes Wissen über die Installation, Nutzung und Infrastruktur von Kubernetes voraus. Der Aufbau der Testumgebung sowie das Aufsetzen der Monitoring Dashboards sind protokolliert. Das Protokoll kann für weitere Informationen herangezogen werden. [10]

2 DEFINITIONEN UND GRUNDLAGEN

In diesem Kapitel werden Begriffe erklärt, welche im Laufe dieser Arbeit vermehrt auftauchen werden. Diese Begrifflichkeiten sind außerdem die theoretischen Grundlagen, welche für das Testen eines Kubernetes-Clusters benötigt werden.

2.1 METRIKEN

Im Folgenden werden Metriken beschrieben, welche wichtige Indikatoren für den Zustand und die „Gesundheit" eines Kubernetes Clusters sind.

2.1.1 *Node Level Metriken*

Unter Node Level Metriken sind Metriken zu verstehen, welche von einem Kubernetes-Knoten abhängen. Da es sich bei einem Kubernetes-Knoten oft um ein physisches Gerät handelt, sind diese Metriken eher darauf ausgelegt, Hardware-Komponenten zu überwachen. Hierbei wird nur die allgemeine Belastung der einzelnen Nodes betrachtet. Für die Arbeit werden diese Metriken hauptsächlich verwendet, um zu überprüfen, dass die Tests korrekt verlaufen und sind daher Kontroll-Metriken.[1, vgl.]

2.1.1.1 *Node Memory Pressure/ Arbeitsspeicher Auslastung*

Unter der Node Memory Pressure wird die allgemeine Auslastung des Arbeitsspeichers eines Knotens oder des gesamten Clusters verstanden [1, vgl.]. Diese Metrik ist ein Indikator dafür, ob das Cluster in absehbarer Zeit hoch skaliert werden muss. Wenn ein Pod mehr Arbeitsspeicher verlangt als zur Verfügung steht, findet ein Out Of Memory Kill (OOMkill) statt, der Pod wird dadurch zwanghaft beendet [11, vgl.]. Durch das Kontrollieren dieser Metrik können unnötige Eviktionen (erzwungenes Beenden von Pods und Prozessen) vermieden werden [1, vgl.].

2.1.1.2 *Node CPU Utilization/ CPU Auslastung*

Die Core Processing Unit (CPU) Auslastung ist ähnlich zur der Node Memory Pressure, bezieht sich jedoch auf die Auslastung der Prozessoren eines Knotens. Es ist wichtig zwischen beiden Metriken zu unterscheiden, da beide Indikatoren für unterschiedliche Probleme sind. Während ein Pod bei Überschreitung des Arbeitsspeichers erzwungener maßen beendet wird, können Anwendungen in den Pods bei der Überschreitung des CPU-Limits in ihrer Funktionalität einbüßen und werden nicht zwangsläufig immer beendet. [1, vgl.]

2.1.1.3 *Network In and Out /Netzwerk Verkehr*

Der Netzwerkverkehr einzelner Knoten kann ein Indikator für die Belastung dieser Knoten sein. Wenn einige Knoten kaum Netzwerkverkehr aufweisen, ist es möglich, dass die Last auf dem Cluster nicht optimal verteilt ist [1, vgl.]. Daher kann diese Metrik auch Fehlkonfigurationen aufdecken, welche beispielsweise das Scheduling auf einigen Worker-Knoten verhindern [9, vgl.][2, vgl.].

2.1.2 *Pod Level Metriken*

Pod Level Metriken geben uns Auskunft über die Gesundheit von Pods in einem Kubernetes Cluster. Sie können Indikatoren für Fehler auf der Anwendungsebene (in den Pods) sein.

2.1.2.1 *OOMkilled Pods*

OOMkills sind Events, die stattfinden, wenn Pods nicht ausreichend Zugriff auf den Arbeitsspeicher haben. Dies kann verschiedene Gründe haben. Es ist beispielsweise gut möglich, dass die Pod-Konfiguration ein zu niedriges Ressourcen-Limit für einen Pod festgelegt hat. OOMkilled Pods können aber auch bedeuten, dass die Pods wesentlich mehr Ressourcen benötigen als sie sollten, dies könnte ein Indiz dafür sein, dass etwas mit der Anwendung in den betroffenen Pods nicht stimmt. Grundsätzlich bedeutet es jedoch, dass die verfügbare Random Access Memory (RAM) nicht für die vorgegebenen Pods ausreicht und die Pods daher terminiert werden.[11, vgl.]

2.1.2.2 *API Responsiveness*

Kubernetes verwendet Application Programming Interface (API)-Anfragen, um mit verschiedenen Komponenten zu kommunizieren. Die Schnittstelle dieser Kommunikation ist die Kubernetes API. Responsiveness ist daher eine wichtige Metrik für die Kubernetes-Infrastruktur. Über API Requests werden beispielsweise Informationen zum Status des Clusters abgefragt, um auf dieser Grundlage Pods, Services und weitere Komponenten zu erstellen. Eine schlechte Responsiveness hat daher eine Vielzahl von Folgen, zum Beispiel eine schlechtere Skalierbarkeit durch hohe Latenzen oder langsame Anwendungen. [7, vgl.]

2.1.3 *Disaster Recovery Metriken*

Disaster Recovery Metriken sind wichtig um zu ermitteln wie sich ein System nach einem Ausfall verhält. Zu diesen Metriken gehören beispielsweise die Zeit, die ein System benötigt, um nach einem Ausfall wieder lauffähig zu sein, auch genannt Mean Time To Recovery (MTTR). Die Zeitspanne zwischen zwei reversiblen Ausfällen (Mean Time Between Failure (MTBF)), sowie die Zeit, die ein System benötigt, um nach der Initialisierung einen irreversiblen

Fehler zu produzieren (Mean Time To Failure (MTTF)), gehören ebenfalls zu den Disaster Recovery Metriken.[4, vgl.]

2.2 LASTTESTS

Lasttests im Kontext einer Kubernetes-Umgebung beschreiben einen Prozess, in dem die Anwendungen in einem Cluster, plötzlich vielen Anfragen in kürzester Zeit ausgesetzt sind. Diese Anfragen werden von den Testern generiert, um das Verhalten des Clusters in Extremfällen zu simulieren [13, vgl.]. Die Durchführung der Tests hängt von den Anwendungen ab, welche auf dem Cluster laufen sowie der Infrastruktur des zu testenden Clusters und den Anforderungen der Tester. Lasttests bieten sich an, um die API-Responsiveness und den Netzwerkverkehr zu testen. Ziel ist es, die Reaktion des Systems auf eine bestimmte Menge an Anfragen oder Benutzern zu ermitteln und die Leistung des Systems in Bezug auf Faktoren wie Reaktionszeit, Durchsatz und Fehlerquote zu bewerten.

2.3 STRESSTESTS

Ähnlich wie Lasttests belasten Stresstests das Kubernetes-Cluster, um das Verhalten des Clusters in extremen Situationen zu ermitteln. Anders als bei Lasttests ist hier das Ziel, die Belastungsgrenze des Clusters zu ermitteln [14, vgl.] . Das bedeutet, dass die Last auf dem Cluster so lange erhöht wird, bis die Anwendungen auf dem Cluster nicht mehr lauffähig sind und wir somit den Breaking Point des Clusters erreicht haben. Damit können wir nicht nur ermitteln, wie viel Last die Kubernetes-Umgebung aushält, sondern auch wie sich das Cluster nach dem Test verhält. Stresstests sind wichtig, um Disaster Recovery-Metriken zu ermitteln. Die Überwachung der Pod-Level-Metriken helfen uns dabei, das Ergebnis dieser Tests zu bestimmen [4, vgl.].

3 METHODIK

In diesem Kapitel wird ausgeführt, welche Methodiken und Techniken für den praktischen Teil dieser Arbeit verwendet werden.

3.1 APIS IN KUBERNETES

Kubernetes verwendet APIs, um intern mit verschiedenen Komponenten zu kommunizieren. Bei diesen APIs handelt es sich meist um HTTP-Endpunkte, welche unterschiedliche Aufgaben übernehmen. Die Kubernetes Metrics API beispielsweise, liefert Informationen über den Status der verschiedenen Ressourcen eines Kubernetes-Clusters. Diese Schnittstelle ist essentiell um Monitoring zu betreiben. Das Erstellen, Löschen und Verändern von Pods, Services oder anderen Komponenten findet im Hintergrund ebenfalls mit API-Anfragen statt. [8]

3.2 API LASTTESTS

API Last Tests können verwendet werden, um Endpunkte einer Kubernetes-Umgebung isoliert zu testen. Die Grundidee hierbei ist es, eine geeignete Schnittstelle zu finden, über welche die generierte Last auf das Cluster gelangen kann. Wie der Name bereits verrät, handelt es sich bei diesen Schnittstellen oft um APIs. Mit solchen Anfragen können sowohl Anwendungen auf dem Cluster als auch interne APIs des Kubernetes-Clusters getestet werden.[6][12]

3.3 ALLGEMEINE VORGEHENSWEISE

Zunächst wird eine Kubernetes-Umgebung aufgebaut, auf der eine API gehostet wird, welche zufällige Daten aus einer Datenbank ausgibt. Daraufhin wird ein Lasttest ausgeführt, welcher diese API für ungefähr eine Stunde belastet, um zu überprüfen, wie sich die Anwendung und das Cluster im Zeitraum einer hohen Belastung verhält. Der Testvorgang wird hauptsächlich mithilfe von API Testing Tools angegangen. Eine Monitoring-Komponente wird verwendet, um die verschiedenen Metriken im Auge zu behalten. Die Last wird in diesem Fall innerhalb des Clusters generiert, kann jedoch, abhängig von verwendeten Technologien, auch extern generiert werden. Das Testen einer selbst erstellten API wurde dem Testen einer Kubernetes internen API vorgezogen, um einen gängigeren Usecase abzudecken. Grundsätzlich ist davon auszugehen, dass eine gut besuchte Web-Anwendung in der Produktion mehr Last generiert als die Kubernetes internen APIs.

3.4 TOOLS ZUM TESTEN EINES KUBERNETES-CLUSTERS

Um die unterschiedlichen Tests erfolgreich auf einem Kubernetes Cluster durchführen zu können, gibt es eine Vielzahl von Möglichkeiten. Um das Testen so unkompliziert wie möglich zu gestalten, wird sich an verschiedenen Tools bedient, welche das Testen vereinfachen sollen und den ganzen Prozess einfacher zu replizieren machen. Die Installation der verschiedenen Programme wird hier nicht behandelt, kann jedoch im Versuchsprotokoll eingesehen werden.[10]

3.4.1 *Locust*

Locust ist ein Open Source Performance Test Framework, das in Kombination mit Python verwendet wird. Locust lässt das Schreiben von Lasttest-Skripten in Python zu. Außerdem besitzt das Framework eine grafische Oberfläche, über welche die verschiedenen Testparameter angegeben werden können. In dieser grafischen Oberfläche werden auch wichtige Metriken zur Bestimmung der API Responsiveness dargestellt [5]. Im späteren Verlauf wird Locust für die Ausführung von Lasttests in einer zu testenden Kubernetes verwendet.

3.4.2 *Alternativen für das Performance Testing*

Abgesehen von Locust gibt es viele andere Tools, welche das Last- oder Stresstesten einer Kubernetes Umgebung vereinfachen können. Der praktische Teil wird zwar nur mit Locust ausgeführt, jedoch können ähnliche Ergebnisse auch mit anderen Tools erlangt werden. Im Folgenden werden zwei Alternativen angeführt.

3.4.2.1 *K6*

K6s ist ebenfalls ein Open Source Performance Testing Tool. Ähnlich wie Locust können hier Skripte geschrieben und ausgeführt werden, um die Endpunkte eines Kubernetes-Clusters zu belasten. Die Skripte werden dabei in JavaScript geschrieben. Anders als bei Locust kann die Last im Kubernetes Server generiert werden. [13][14]

3.4.2.2 *Apache JMeter*

Apache JMeter ist eine Desktop Anwendung, welche von der Apache Foundations als Open Source Projekt entwickelt wurde. Ursprünglich war diese Anwendung darauf ausgelegt ausschließlich die Performance von Web-Anwendungen zu testen, daher lassen sich auch APIs damit testen. Für die Nutzung von Apache JMeter werden keine zusätzlichen Skripte benötigt, die Tests werden alle mithilfe der grafischen Oberfläche von Apache JMeter ausgeführt.[3]

4 PRAKTISCHER TEIL

In diesem Kapitel wird eine zu testende Kubernetes-Umgebung aufgebaut und anschließend mithilfe verschiedener Tools getestet. Der Schritt-für-Schritt-Aufbau des Clusters sowie die Installation verschiedener Tools wird in diesem Kapitel nicht behandelt, ist jedoch auf einem GitHub-Repository dokumentiert und kann daher nach gebaut werden. [10]

4.1 INFRASTRUKTUR DER TESTUMGEBUNG

4.1.1 *Ressourcen*

Für das Kubernetes-Cluster wird ein Zwei-Node-Cluster (1 Master und 1 Worker) verwendet. Jeder Knoten hat hierbei jeweils zwei physische und vier logische Kerne. Außerdem besitzt jeder Knoten 16 GB DDR3-RAM. Jeder Knoten besitzt ein Ubuntu Server-Betriebssystem. Die verwendeten Server stammen in diesem Fall von der Open Telekom Cloud. Es kann jedoch auch jede Art von virtueller Maschine verwendet werden, solange eine beliebige Kubernetes-Engine auf dem Gerät lauffähig ist und sich die Nodes gegenseitig erreichen können.

4.1.2 *Technologien und Anwendungen*

Als Lightweight Kubernetes Engine wurde K3s benutzt, der Testvorgang sollte jedoch mit jeder anderen Kubernetes Engine möglich sein. K3s erlaubt das Scheduling von Pods auf den Master Knoten, um dies zu verhindern, kann ein Taint auf den Master Knoten gesetzt werden, welcher das Scheduling auf diesem Knoten nicht zulässt. Diese Einschränkung sorgt dafür, dass weniger Last generiert werden muss, um die Obergrenze des Clusters zu erreichen [2, vgl.].
Im zu testenden Cluster wird eine Flask API gehostet, welche auf Anfrage zufällige Daten aus einem PostgreSQL Server als JSON-Datei übermittelt. Daher läuft ebenfalls ein PostgreSQL Server im Cluster. Die Last wird durch API-Requests auf die Flask-Anwendung generiert. Das Scheduling des Datenbank-Servers sowie der Flask-API finden auf dem Worker-Knoten statt. Die Ausführung der verschiedenen Skripte findet auf dem Master-Knoten statt, die Last kann aber auch theoretisch von außerhalb des Clusters generiert werden [10, vgl.].

```
root@mse:/home/mse# kubectl get all
NAME                                                   READY   STATUS    RESTARTS        AGE
pod/prometheus-prometheus-node-exporter-sdh98          1/1     Running   1 (8m49s ago)   20h
pod/postgresql-0                                       1/1     Running   0               8m51s
pod/prometheus-prometheus-node-exporter-c4d4x          1/1     Running   1 (6m33s ago)   20h
pod/prometheus-prometheus-node-exporter-lwk9f          1/1     Running   2 (3m5s ago)    20h
pod/prometheus-kube-prometheus-operator-84d9478dcc-7lhj9   1/1     Running   1 (3m5s ago)    10m
pod/alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   0               2m8s
pod/prometheus-prometheus-kube-prometheus-prometheus-0   2/2     Running   4 (3m5s ago)    20h
pod/prometheus-kube-state-metrics-f4fc57bd5-w7757      1/1     Running   2 (2m ago)      10m
pod/api-deployment-8465b8dff4-z9gvv                    1/1     Running   23 (2m ago)     18h
pod/prometheus-grafana-6fbc8d96f4-6kbf7                3/3     Running   6 (3m5s ago)    20h

NAME                                              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
service/kubernetes                                ClusterIP   10.43.0.1       <none>        443/TCP                      43h
service/postgresql                                ClusterIP   10.43.139.5     <none>        5432/TCP                     37h
service/postgresql-hl                             ClusterIP   None            <none>        5432/TCP                     37h
service/api-service                               NodePort    10.43.245.201   <none>        5000:30069/TCP               35h
service/prometheus-kube-state-metrics             ClusterIP   10.43.45.97     <none>        8080/TCP                     20h
service/prometheus-kube-prometheus-operator       ClusterIP   10.43.89.131    <none>        443/TCP                      20h
service/prometheus-kube-prometheus-prometheus     NodePort    10.43.115.241   <none>        9090:30218/TCP               20h
service/prometheus-prometheus-node-exporter       ClusterIP   10.43.220.191   <none>        9100/TCP                     20h
service/prometheus-grafana                        NodePort    10.43.110.104   <none>        80:30143/TCP                 20h
service/prometheus-kube-prometheus-alertmanager   ClusterIP   10.43.149.69    <none>        9093/TCP                     20h
service/alertmanager-operated                     ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UDP   20h
service/prometheus-operated                       ClusterIP   None            <none>        9090/TCP                     20h

NAME                                                 DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/prometheus-prometheus-node-exporter   3         3         3       3            3           <none>          20h

NAME                                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/prometheus-kube-prometheus-operator   1/1     1            1           20h
deployment.apps/prometheus-kube-state-metrics       1/1     1            1           20h
deployment.apps/api-deployment                      1/1     1            1           24h
deployment.apps/prometheus-grafana                  1/1     1            1           20h

NAME                                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/prometheus-kube-prometheus-operator-84d9478dcc   1         1         1       20h
replicaset.apps/prometheus-kube-state-metrics-f4fc57bd5   1         1         1       20h
replicaset.apps/api-deployment-8465b8dff4                 1         1         1       24h
replicaset.apps/prometheus-grafana-6fbc8d96f4             1         1         1       20h

NAME                                                          READY   AGE
statefulset.apps/postgresql                                   1/1     37h
statefulset.apps/alertmanager-prometheus-kube-prometheus-alertmanager   1/1     20h
statefulset.apps/prometheus-prometheus-kube-prometheus-prometheus       1/1     20h

NAME                              COMPLETIONS   DURATION   AGE
job.batch/postgres-random-data    1/1           90s        36h
```

Abbildung 4.1: Laufende Komponenten

4.1.3 *Monitoring*

Für das Monitoring der verschiedenen Metriken wird Prometheus in Kombination mit Grafana verwendet. Prometheus dient dabei als Datasource, welche die nötigen Daten der einzelnen Metriken vom betroffenen Kubernetes Cluster anfordert. Grafana wird für die Visualisierung verwendet und lässt den Aufbau von Monitoring-Dashboards auf Grundlage der Daten, die durch Prometheus angefragt werden, zu.[10, vgl.]

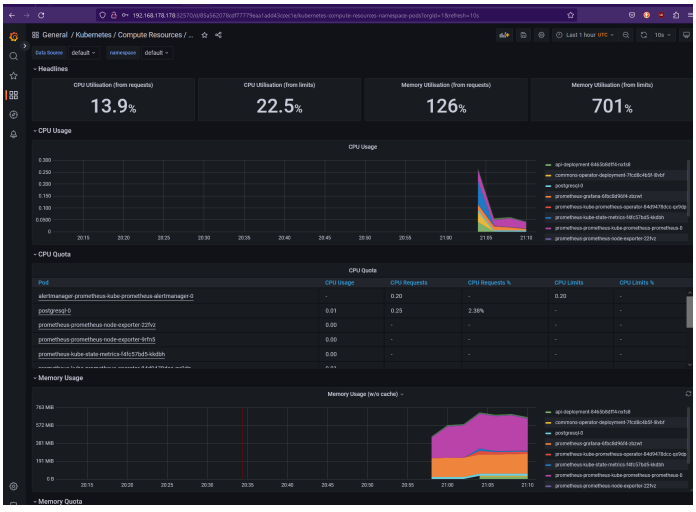

Abbildung 4.2: Monitoring Dashboard

4.2 TESTEN DES CLUSTERS

In diesem Abschnitt werden die Tests mithilfe des vorgestellten Locust vollzogen. Die Last für die Tests wird mithilfe von automatisierten Skripten im Masterknoten generiert. Locust lässt jedoch auch das Testen eines Kubernetes-Clusters von außerhalb zu. In diesem Fall wird darauf verzichtet, da der Masterknoten kein Scheduling betreibt und daher genügend freie Ressourcen besitzt. Es wird zwar nur das Lasttesten mittels Locust realisiert, jedoch ist der Testvorgang mit anderen Tools wie K6 oder Apache JMeter auch möglich.

4.2.1 *Testen mit Locust*

Um Locust verwenden zu können, sollte Python 3 bereits installiert sein. Für die Verwendung von Locust wird zunächst ein Python-Skript benötigt, welches die zu testenden Cluster-Endpunkte angibt. [5, vgl.]
Der Einfachheit halber wird die Datei „locust.py" erstellt, welche lediglich für jeden angelegten Test-Nutzer die API aufruft.

```
1 from locust import HttpUser, task, between
2
3 class MyUser(HttpUser):
4     wait_time = between(1, 5)
5
6     @task
7     def get_endpoint(self):
8         self.client.get("http://192.168.178.178:30069/random-row")
```

Die Klasse„MyUser" repräsentiert einen virtuellen Nutzer. Der Dekorator „@task" kennzeichnet eine Aufgabe, welche die Nutzer ausführen, um die Kubernetes-Umgebung zu belasten. Wie bereits erwähnt wird in diesem Test nur die API aufgerufen [10, vgl.]. Dieser Aufruf wird spätestens nach fünf Sekunden von jedem Nutzer wiederholt.
Nun muss das Skript in der Kommandozeile ausführt werden:

```
locust -f locust.py --host=http://192.168.178.178:30069/random-row --master
```

Daraufhin sollte es eine Ausgabe geben, welche standardmäßig auf den Port 8089 verweist. Unter diesem Port kann die grafische Oberfläche von Locust erreicht werden.

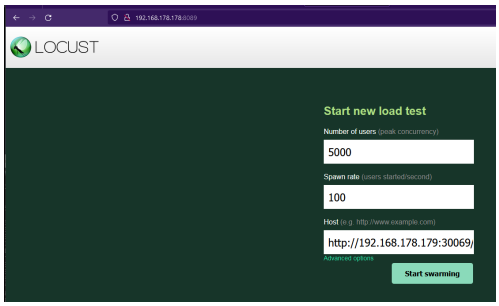

Abbildung 4.3: Locust Parameter Eingabe

Die grafische Oberfläche lässt den Testenden eine maximale Besucheranzahl festlegen. Außerdem kann eine Spawnrate gewählt werden, welche bestimmt, wie viele Besucher jede Sekunde hinzugefügt werden sollen, bis die maximale Besucherzahl erreicht ist. In diesem Fall gibt es insgesamt 5000 Nutzer, dabei werden jede Sekunde 100 Nutzer hinzugefügt. Es ist nicht zu empfehlen, eine Spawnrate größer als 100 auszuwählen, da es zu Fehlern in Locust kommen kann. Die „Advanced options" lassen den Nutzer eine Dauer für den Testlauf festlegen, dies ist nützlich für Endurance-Tests, ist jedoch für diesen Usecase nicht relevant. Nachdem alle 5000 Nutzer dem Last Test hinzugefügt werden, senden die Nutzer alle eins bis fünf Sekunden eine Anfrage an die betroffene API.
Wenn der Test ausgeführt wird, werden alle Aufgaben des virtuellen Besuchers aufgelistet. Locust bietet eine Graphenansicht an, welche verschiedene Details zum Test ausgibt.

4.2.1.1 *API Responsiveness*

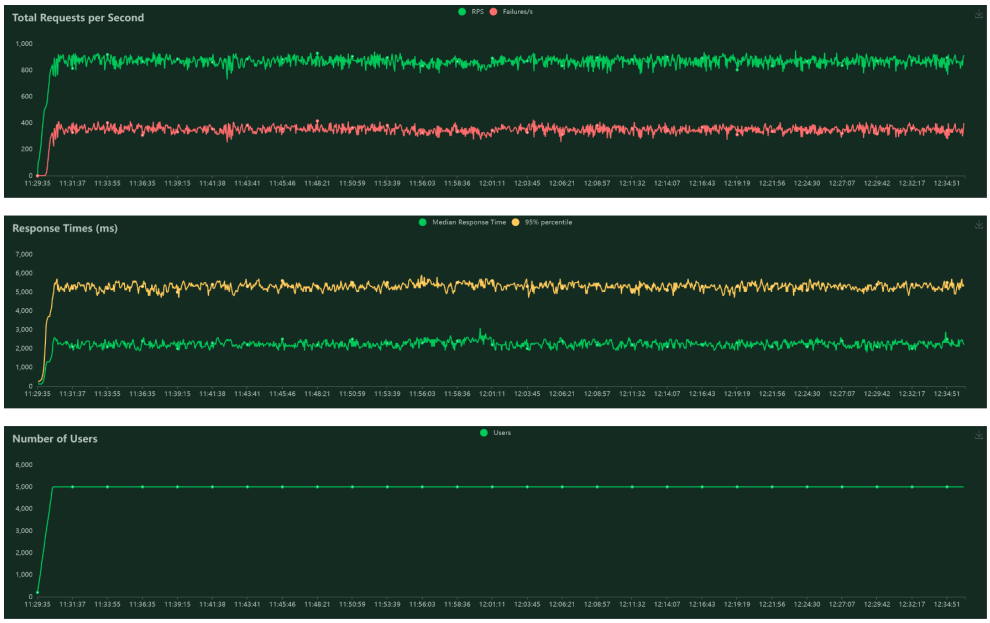

Abbildung 4.4: Locust API Responsiveness Dashboard

Diese grafische Ansicht ermöglicht es die API-Responsiveness der getesteten Endpunkte zu bestimmen. Das obere Diagramm zeigt, dass die API ab ungefähr 500 Anfragen pro Sekunde schon beginnt, die ersten Fehlschläge zurückzugeben. Mit dem Anstieg der Anfragen häufen sich gleichermaßen auch die Fehlschläge. Zwischen 900 und 1000 Anfragen pro Sekunde hat die API eine durchgehende Fehlerrate von 40%. Ein Breaking Point wurde noch nicht erreicht, da die API zu einem gewissen Grad noch nutzbar ist.
In der zweiten Grafik kann auch gesehen werden, dass die Antwortdauer zusammen mit den gesendeten Anfragen ansteigt. Bei rund 900 Anfragen pro Sekunde beträgt die durchschnittliche Antwortzeit (grüner Graph des zweiten Diagramms) ungefähr 2400 ms, umgerechnet wären das 2,4 Sekunden. Der gelbe Graph gibt die Dauer an, welche die API mindestens benötigt, um auf 95% der Anfragen zu antworten. Dieser Wert liegt bei ungefähr 5 Sekunden. Während des Tests ist es möglich auch manuell Anfragen an die API zu senden. Dabei würde man feststellen, dass die Antwortzeit tatsächlich zwischen 3 und 6 Sekunden liegt.
Um sicherzustellen, dass dieser Test tatsächlich das Cluster belastet hat, bietet sich an, einen Blick auf die anfangs vorbereiteten Kontrollmetriken zu werfen, welche über Grafana eingesehen werden können.

4.2.1.2 *Ressourcen Auslastung*

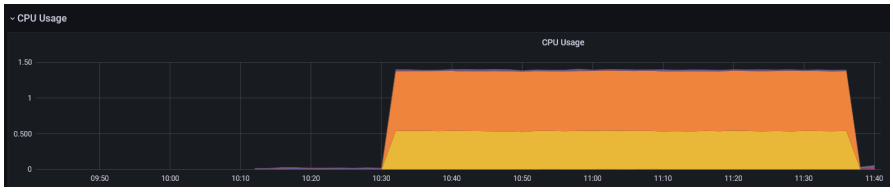

Abbildung 4.5: Grafana Dashboard CPU Auslastung

In dieser Grafik wird in Gelb die CPU Nutzung der API und in Orange die CPU Nutzung des Datenbankservers dargestellt. Hierbei haben während des Lasttests beide Anwendungen rund 1,5 Kerne in Anspruch genommen. Das entspricht fast der gesamten CPU-Leistung eines Nodes. Der Großteil des Verbrauchs stammt in diesem Fall vom Datenbankserver. Der Grundverbrauch beider Anwendungen ohne jegliche Belastung ist hierbei nicht signifikant.

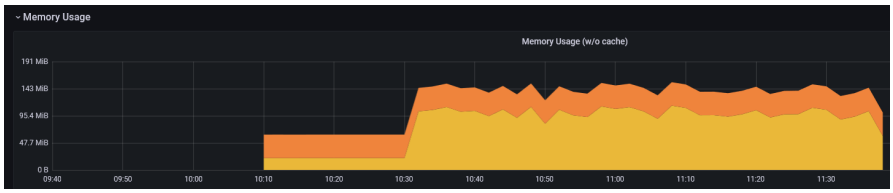

Abbildung 4.6: Grafana Dashboard RAM Auslastung

Die Arbeitsspeichernutzung beider Pods zeigt ebenfalls an, dass die Tests eine gewisse Last auf das Cluster ausgeübt haben, jedoch ist die Last auf

den Arbeitsspeicher nicht so hoch wie die Last auf den Prozessor. Die beiden Anwendungen hatten vor dem Test einen Grundverbrauch von ungefähr 60 MiB. Während des Tests hat sich der Verbrauch fast verdreifacht, beide Anwendungen haben hier rund 160 MiB in Anspruch genommen. Nach dem Test ist der Grundverbrauch auf 90 MiB angestiegen, wohlmöglich wurden viele Datenbank-Einträge durch die ganzen Lese-Anfragen in den Arbeitsspeicher geladen und danach nicht mehr freigegeben.

4.2.1.3 *Netzwerk Auslastung*

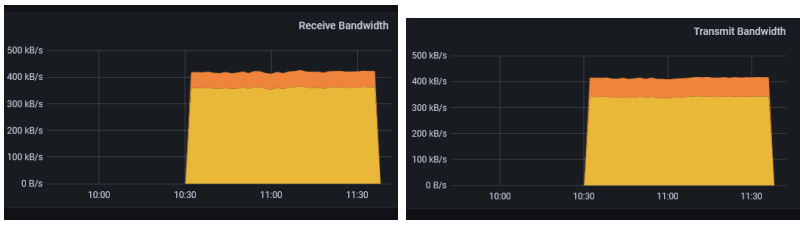

Abbildung 4.7: Grafana Dashboard Netzwerk Input und Output

Der empfangene und gesendete Netzwerkverkehr der API fiel insgesamt gering aus und machte während des Last-Tests fast nur ein Sechstel des gesamten Verkehrs der Kubernetes Umgebung aus.

5 AUSWERTUNG DER ERGEBNISSE

In diesem Kapitel werden die Testergebnisse gedeutet. Außerdem werden Maßnahmen aufgegriffen, welche erfolgen können, um nach diesen Tests die Qualität der Kubernetes-Umgebung sicherzustellen. Des Weiteren werden mögliche Aussichten für weitere Tests erläutert.

5.1 DEUTUNG DER TESTERGEBNISSE

Die Testergebnisse zeigen, dass die getestete API bei unter 500 Anfragen pro Sekunde fehlerfrei funktioniert. Bei mehr als 500 Anfragen treten die ersten Fehler auf, daher ist festzuhalten, dass das Cluster bei knapp unter 500 Anfragen pro Sekunde hoch skaliert werden sollte, um die Verfügbarkeit der API nicht einzuschränken und die API Responsiveness nicht zu verringern. Durch die Überwachung der Ressourcenauslastungen kann auch festgestellt werden, dass die problembereitende Ressource in diesem Fall die CPU ist. Sowohl die Arbeitsspeicherauslastung als auch die Netzwerkauslastung fielen verglichen mit der Prozessorauslastung relativ gering aus. Während des Lasttests nahmen die API und der Datenbankserver fast die gesamte Leistung eines Prozessors in Anspruch. Deshalb ist bei der Skalierung dieser Anwendung zu empfehlen, die CPU Leistung zu erhöhen und die Arbeitsspeicherleistung aus Kosten- oder Effizienzgründen runterzuschrauben. Das Hochskalieren der Kubernetes-Umgebung ist jedoch nicht die einzige Möglichkeit, um die Verfügbarkeit der Anwendung zu steigern. Durch das Verwenden einer Ingress-Komponente können beispielsweise mehrere Requests der gleichen IP-Adresse gefiltert werden, um Last zu sparen und absichtliches Überlasten der Anwendung zu verhindern. Solche Tests können auch Probleme der Anwendung selbst offenbaren. Ein Mechanismus zur Limitierung von empfangenen Anfragen könnte in der API selbst realisiert werden.

5.2 WEITERE TESTS

Während des Lasttests konnte zwar eine Fehlerquote generiert werden, jedoch war die API weiterhin nutzbar. Es kam zu keinen Pod-Eviktionen. Für das weitere Testen der API bieten sich eine Reihe von Methoden an. Um die Grenzen der API zu ermitteln würden sich Stresstests anbieten. Eventuell könnte man die Datenbank mit weiteren Daten befüllen, um zu schauen, wie sehr die Anzahl der Datenbankeinträge die Obergrenze für mögliche Anfragen auf die API beeinflusst. Pod Eviktionen bieten die Möglichkeit, Disaster Recovery Metriken zu überprüfen. Dies ist vor allem wichtig, um zu identifizieren, ob die Anwendung nach einer hohen Belastung noch fehlerfrei funktioniert oder eventuell neu gestartet werden muss.

6 FAZIT

Das Peformance-Testen einer Kubernetes Umgebung ist eine wichtige Qualitätssicherungsmaßnahme. Die Tests können viele verschiedene Probleme offenbaren, welche auf dem ersten Blick nicht sichtbar sind. Ermittelt werden können von Fehlkonfigurationen bis hin zu Sicherheitslücken, oder Problemen in der Skalierung, ein sehr breiter Umfang an Fehlern. Die Performance Tests geben außerdem Auskunft über die Lastenverteilung der gehosteten Anwendungen. Einer der Hauptziele des Performance Tests ist es, die Probleme ausfindig zu machen, welche die Verfügbarkeit der betriebenen Anwendungen einschränken können. Des Weiteren ist es möglich herauszufinden, wie schnell die verschiedenen Ressourcen (CPU, RAM, HDD/SSD) die Obergrenze erreichen. Mit dieser Information kann in der Skalierung Geld und Zeit gespart werden, da es nur notwendig ist die mangelnden Ressourcen nachzurüsten. Das Kennen des Kapazitätslimit des eigenen Clusters ist außerdem wichtig um zu einem geeigneten Zeitpunkt zu skalieren.
Eine Vielzahl von Tools ermöglichen unkompliziertes und qualitatives Testen von Kubernetes Endpunkten. Mit einer Monitoring Komponente so wie Grafana in Kombination mit Prometheus, ist das Einsehen vieler wichtiger Metriken möglich.
Das Performance Testen einer Kubernetes Umgebung bringt viele Vorteile und benötigt einen geringen Aufwand, sodass es sich nicht lohnt darauf zu verzichten.

LITERATUR

[1] Arie Bregman. *Kubernetes Metrics to Monitor for Optimal Cluster Performance*. https://medium.com/@bregman.arie/18-kubernetes-metrics-to-monitor-for-optimal-cluster-performance-ca9458869e52: Medium.com, 2023.

[2] Halim Fathoni. *Performance Comparison of Lightweight Kubernetes in Edge Devices*. https://link.springer.com/chapter/10.1007/978-3-030-30143-$9_2$5: Springer, 2019.

[3] Apache Foundations. *Apache JMeter Dokumentation*. https://jmeter.apache.org/: Apache Foundations.

[4] Damon M. Garn. *5 IT Disaster Recovery Measurements to Know* . https://www.comptia.org/blog/disaster-recovery-measurements: Comptia, 2022.

[5] Jonatan Heyman. *Locust Dokumentation*. https://docs.locust.io/en/stable/what-is-locust.html: Locust, 2023.

[6] Nicole van der Hoeven. *What is API Load Testing?* https://www.flood.io/blog/what-is-api-load-testing: flood, 2019.

[7] Kubernetes IO. *Kubernetes Performance Measurements and Roadmap* . http://kubernetes.io/blog/2015/09/kubernetes-performance-measurements-and/: Kubernetes, 2015.

[8] Kubernetes. *API Overview*. https://kubernetes.io/docs/reference/using-api/: Kubernetes, 2022.

[9] Tarek Menouer. *Containers Scheduling Consolidation Approach for Cloud Computing*. https://link.springer.com/chapter/10.1007/978-3-030-30143-$9_2$5: Springer, 2019.

[10] Muhammet Sen. *Kubernetes Performance Testing* . https://github.com/msen7437/Kubernetes-Performance-Testing: Muhammet Sen, 2023.

[11] Natan Yellin. *What everyone should know about Kubernetes memory limits, OOMKilled pods, and pizza parties*. https://home.robusta.dev/blog/kubernetes-memory-limit: Robusta, 2022.

[12] k6. *API load testing*. https://k6.io/docs/testing-guides/api-load-testing/: k6, 2022.

[13] k6. *Load Testing*. https://k6.io/docs/test-types/load-testing/: k6, 2022.

[14] k6. *Stress Testing*. https://k6.io/docs/test-types/stress-testing/: k6, 2022.