HAW HAMBURG

Masterarbeit

Alina Böttcher

Automatisierte und verbesserte Ressourcenzuweisung in einem Kubernetes-Cluster mittels Maschinellen Lernens

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Alina Böttcher

Automatisierte und verbesserte Ressourcenzuweisung in einem Kubernetes-Cluster mittels Maschinellen Lernens

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 31. März 2023

Alina Böttcher

Thema der Arbeit

Automatisierte und verbesserte Ressourcenzuweisung in einem Kubernetes-Cluster mittels Maschinellen Lernens

Stichworte

Kubernetes, Operator, Operator-SDK, Golang, verändernder WebHook, Maschinelles Lernen, bestärkendes Lernen, Ressourcenmanagement, Q-Lernen

Kurzzusammenfassung

Die Ressourcenanfragen von Pods in einem Kubernetes-Cluster erfolgen meist manuell und beruhen lediglich auf Schätzungen der Nutzer. Da diese ungenau und häufig zu hoch sind, ist die Ressourcenauslastung auf den Servern nicht optimal.
Ressourcenanfragen von laufenden Pods können nur um den Preis eines Abbruchs mit anschließendem Neustart verändert werden. Da das vermieden werden soll, muss eine Optimierung der Ressourcenanfragen vor dem Start der jeweiligen Pods erfolgen. Diese Masterthesis beschäftigt sich mit der Frage, wie und unter welchen Bedingungen Ressourcenanfragen von Pods in einem Kubernetes-Cluster vor deren Start automatisiert verbessert werden können.
Dazu wurde ein Operator mit Hilfe des Operator-SDKs und Golang geschrieben. Dieser überwacht die Pods und speichert den jeweiligen tatsächlichen Ressourcenverbrauch ab. Mit Hilfe des Q-Lernen-Algorithmus aus dem Bereich des Bestärkenden-Lernens werden die tatsächlichen Ressourcenverbräuche gelernt. Anschließend bestimmt der Algorithmus neue Ressourcenanfragen, um damit zukünftige Ressourcenanfragen automatisiert anzupassen und zu verbessern.
Die automatisierte Anpassung erfolgt mittels eines verändernden WebHooks, der die manuellen Anfragen durch die vom Algorithmus ermittelten Anfragen ersetzt. Dieses erfolgt, während der Pod durch Kubernetes erstellt und bereitgestellt wird. In dieser Phase führt eine Veränderung der Ressourcenanfrage noch nicht zu einem Abbruch mit anschließendem Neustart.
Es kann gezeigt werden, dass eine Anpassung von Ressourcenanfragen unter Nutzung des Operators möglich ist, ohne dass ein Abbruch mit anschließendem Neustart erfolgen muss. Es gibt jedoch einige Kriterien, die die Pods erfüllen müssen, damit diese Anpassung möglich ist.

Alina Böttcher

Title of Thesis

Automated and improved resource allocation in a Kubernetes cluster using machine learning

Keywords

Kubernetes, operator, Operator-SDK, Golang, mutating webhook, machine learning, reinforcement learning, resource management, q-learning

Abstract

Resource requests from pods in a Kubernetes cluster are mostly manual and based only on user estimates. Since these are inaccurate and often too high, resource utilization on the servers is not optimal.
Resource requests from running pods can only be modified at the cost of aborting followed by a restart. Since this is to be avoided, optimization of resource requests must be done before the respective pods are started. This master thesis addresses the question of how and under what conditions resource requests from pods in a Kubernetes cluster can be automatically improved before they are started.
For this purpose, an operator was written using the Operator SDK and Golang. This monitors the pods and stores the actual resource consumption in each case. Using the Q-learning algorithm from reinforcement learning, the actual resource consumptions are learned. Then, the algorithm determines new resource requests to use to automatically adjust and improve future resource requests.
The automated adaptation is done using a mutating WebHook that replaces the manual requests with the requests determined by the algorithm. This is done while the pod is being created and deployed by Kubernetes. At this stage, a change in the resource request does not yet result in an abort followed by a restart.
It can be shown that it is possible to adjust resource requests using the operator without having to abort followed by a restart. However, there are some criteria that the pods must fulfill for an adaptation to be possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit Ressourcenanfragen in Kubernetes. Wie in anderen Bereitstellungssystemen auch, ist es in Kubernetes eine Herausforderung, die benötigten Ressourcen richtig einzuschätzen. In [4] wird gezeigt, dass es sich bei der Ressourcenanfrage um ein NP-Vollständiges-Problem handelt, das sich nicht nur auf Kubernetes beschränkt.
Vor dieser Herausforderung stehen vielmehr all jene, die ein Cluster administrieren und für die optimale Nutzung der vorhandenen Ressourcen verantwortlich sind. Nur ein möglichst optimal ausgelasteter Cluster kann wirtschaftlich betrieben werden und verursacht keine unnötigen Kosten.
In dieser Arbeit wird ein Ansatz vorgestellt, wie die Ressourcenvergabe mittels Maschinellen Lernens optimiert und automatisiert werden kann.

1.1 Problemstellung und Abgrenzung

Die Ressourcenvergabe in einem Kubernetes-Cluster geschieht auf Basis von Ressourcenanfragen, welche aktuell manuell im Deployment angegeben werden. Diese angegebenen Ressourcenanfragen beruhen auf Abschätzungen und weichen daher oft stark von den tatsächlich verbrauchten Ressourcen ab.
In Kubernetes sind Ressourcenanfragen optional. Wenn keine Anfragen existieren, dann werden der Anwendung so viele Ressourcen bereitgestellt, wie diese anfordert. Je nach Anwendungsgröße kann das zu Problemen führen, wenn z.B. keine freien Ressourcen mehr für andere Anwendungen zur Verfügung stehen.
Gut zu sehen ist das im Bild 1.1 auf Seite 2. Die Differenz zwischen der oberen und der unteren Kurve enthällt die CPU-Kerne, die angefragt, aber nicht benötigt werden. Die untere Kurve zeigt die CPU-Kerne, die laut Anfragen noch frei sein sollten. Die obere Kurve stellt die CPU-Kerne da, die tatsächlich noch frei sind. Im Schnitt handelt es

sich hier um etwa 150 CPU-Kerne, die angefragt, aber ungenutzt sind. Dieses führt im Ergebnis dazu, dass die Ressourcen eines Kubernetes-Clusters nicht optimal ausgenutzt werden können.
Da es sich im hier behandelten Szenario um tausende automatisierte Deployments täglich handelt, ist eine genauere manuelle Abschätzung zu aufwendig. Daher soll die Optimierung der Ressourcenanfragen automatisiert werden.
Grundsätzlich gibt es zwei Möglichkeiten, den Ressourcenverbrauch anzupassen: Die horizontale und die vertikale Skalierung. Die horizontale Skalierung variiert die Anzahl der Pods. Wenn die Anwendung hochskaliert werden soll, dann werden mehr Pods hinzugefügt, andererseits wieder gelöscht. Das ist notwendig, wenn die Last auf der Anwendung zunimmt bzw. abnimmt.
Bei der vertikalen Skalierung werden die Ressourcen, die einem einzelnen Pod zugewiesen werden, erhöht oder verringert. In dem Paper [33] wird als Lösungsansatz ein Programm

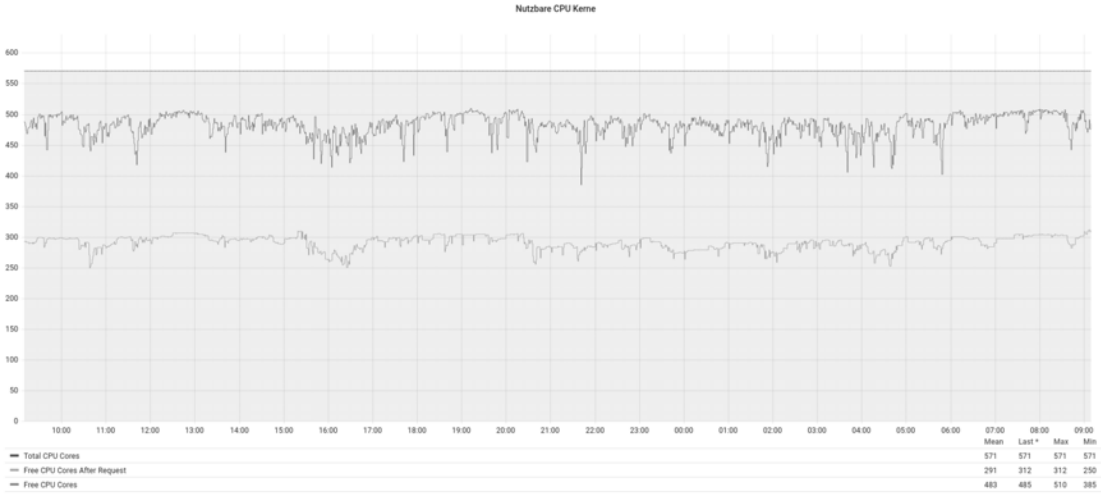

Abbildung 1.1: Nutzbare CPU-Kerne in einem Test-Cluster

beschrieben, welches im laufenden Betrieb den Verbrauch misst und die Pods anschließend anpasst. Auch [1] beschäftigt sich mit der vertikalen Skalierung, setzt aber als Lösungsansatz bei den Docker-Containern an.
In beiden Fällen wird versucht, die Auswirkungen des erzwungenen Neustarts der Pods, nach einer Veränderung der Ressourcen, abzumildern, indem der Zustand der Pods zwischengespeichert wird.
Wie in der Umfrage [39] zu unterschiedlichen Skalierungen gut beschrieben ist, gibt es keine weiteren Untersuchungen zur vertikalen Skalierung. Entsprechend befassen sich die folgenden Untersuchungen mit einer Optimierung der horizontalen Skalierung bzw. mit

der Optimierung der Scheduler.
In [12] wird ein Ansatz mit Maschinellem Lernen verwendet, um containerisierte Anwendungen horizontal zu skalieren. Das dort verwendete neuronale Netz mit Kurzzeitgedächtnis (LSTM) konnte eine Verbesserung in der horizontalen Skalierung zeigen. Auch in [3] wird eine automatisierte horizontale Skalierung vorgeschlagen. Hier soll ein Programm namens Libra automatisiert die Skalierung übernehmen.
In dem hier behandelten Anwendungsfall ist die Ressourcenanfrage der einzelnen Pods relevant, da eine Replizierung nicht möglich ist. Aus diesem Grund bieten die vorgeschlagenen Verbesserungen von [3] und [12] keinen geeigneten Lösungsansatz.
In dem Paper [10] wird eine alternative Art des Container-Schedulings vorgestellt. Wenn der Scheduler einen Arbeiter-Knoten für einen Container sucht, dann schaut der Scheduler, wo genügend Platz vorhanden ist. In dem Paper wird ein Ansatz vorgeschlagen, bei dem die Restlaufzeit der vorhandenen Pods mit in die Entscheidung einfließt. Dieses soll die Auslastung auf den Arbeiter-Knoten erhöhen und die Ausführzeit deutlich verringern.
Ein weiterer alternativer Scheduler wird in [36] vorgestellt. Hier werden Performance Level Objectives (PLOs) erstellt. Das ist eine Sammlung von Metriken, wie Flexibilität, Verfügbarkeit etc. anhand derer die Bedürfnisse der Benutzer bezüglich der Ressourcen geschätzt werden. Auf Grundlage der PLOs findet dann die Ressourcenanfrage statt. Dieser Scheduler ist für den Einsatz bei Cloudanbietern ausgelegt. Die Optimierung der Scheduler liefert zwar eine Optimierung der Auslastung auf den Arbeiter-Knoten, aber keine Lösung für die Herausforderung der zu hohen manuellen Ressourcenanfrage der Pods.

In dieser Arbeit wird als Lösungsansatz ein Operator vorgestellt, welcher mittels Maschinellen Lernens erlernt, welche Container mit welchen Ressourcen optimal laufen und der dann die manuellen Ressourcenanfragen entsprechend optimiert. Bei diesem Ansatz werden die angefragten Ressourcen über einen WebHook noch vor dem Start der Pods angepasst. Dieses hat zur Folge, dass kein Neustart benötigt wird.
Trotz allem sollte bedacht werden, dass es laut [13] in der Ressourcenverwaltung von Kubernetes ein Problem hinsichtlich der zugesicherten Ressourcen gibt. Die CPU-Last des Netzwerkverkehrs von netzwerkintensiven Pods wird nicht ausreichend bei der Ressourcenverteilung berücksichtigt. Dieses kann dazu führen, dass die Leistung deutlich abfallen kann, wenn Netzwerk- und CPU-intensive Pods auf dem gleichen Arbeiter-Knoten laufen.

1.2 Ziele

Das Ziel dieser Ausarbeitung ist es, die Ressourcenvergabe an die kurzlebigen und häufig wiederkehrenden Deployments zu optimieren. Dadurch soll die Auslastung im Kubernetes-Cluster verbessert werden.
Dieses Ziel soll mithilfe eines Operators erreicht werden. Dieser soll mittels Maschinellen Lernens den tatsächlichen Ressourcenverbrauch lernen und die Ressourcenanfrage mittels WebHook während der Erstellung der Pods anpassen.

1.3 Struktur der Arbeit

In den Grundlagen in Kapitel 2 erfolgt in Abschnitt 2.1 zuerst eine Einführung in Kubernetes und die wichtigsten Bestandteile und Abläufe dort. Anschließend wird in Abschnitt 2.2 das Operator-Framework inklusive seiner Vorteile erläutert. Daran anschließend wird dann in Abschnitt 2.3 das Tool Prometheus inklusive seines Einsatzbereiches vorgestellt. In Abschnitt 2.4 wird eine Übersicht über die verwendete Datenbank PostgreSQL gegeben und die Möglichkeiten der langfristigen Speicherung in Kubernetes erläutert. Danach wird in Abschnitt 2.5 noch eine Einführung in die wichtigsten Algorithmen des Maschinelles Lernens gegeben und im letzten Abschnitt 2.6 wird dann der Q-Lernen-Algorithmus genauer erläutert.

In der Konzeptionierung in Kapitel 3 wird in Abschnitt 3.1 der Aufbau des Operators vorgestellt. Danach folgt in Abschnitt 3.2 das Verhalten des Operators zur Laufzeit und in Abschnitt 3.3 der Ablauf innerhalb des WebHooks beim Start eines neuen Pods.
Daran anschließend folgt in Abschnitt 3.4 die Erläuterung der Tabellenschemata der Datenbank und in 3.5 die Daten, die in Prometheus vorhanden sind. Hier wird auch erläutert, welche Daten davon im Operator genutzt werden. Als letztes folgt in Abschnitt 3.6 die Erläuterung zur Auswahl des Algorithmus.

In Kapitel 4 wird die Realisierung des Operators beschrieben. Zuerst erfolgt in Abschnitt 4.1 die Erläuterung über die Nutzung des Operator-SDKs. Dazu werden in Abschnitt

4.1.1 die wichtigsten YAML-Dateien der Operator-Bestandteile und notwendigen Ressourcen erklärt. Anschließend wird dann im Abschnitt 4.1.2 der Inhalt der Reconcile-Funktion gezeigt.
Im Abschnitt 4.1.3 wird die Umsetzung des WebHooks erläutert. Danach folgt im Abschnitt 4.1.4 der Prometheus-Client und in Abschnitt 4.1.5 der Aufbau des PostgreSQL-Clients.
Des Weiteren werden in Abschnitt 4.2 die Abfragen an Prometheus behandelt. Zum Schluss wird in Abschnitt 4.3 noch beschrieben, welche Herausforderungen während der Durchführung der Arbeit aufgetreten sind und wie diese gelöst wurden.

In den Ergebnissen in Kapitel 5 wird zuerst in den Abschnitten 5.1 und 5.2 geprüft, ob eine Verbesserung der Anfragen erreicht wurde. Anschließend wird in den Abschnitten 5.3 und 5.4 erläutert, worauf beim Bereitstellen des Operators geachtet werden muss.

Zum Schluss wird in Kapitel 6 noch ein Fazit zu der Arbeit und ein Ausblick auf eine mögliche weiterführende Untersuchung gegeben.

2 Grundlagen

In diesem Kapitel werden in Abschnitt 2.1 zuerst die Grundlagen und die für diese Arbeit wichtigsten Bestandteile von Kubernetes beschrieben. Anschließend wird in Abschnitt 2.2 das Operator-Framework erläutert. In Abschnitt 2.3 folgt dann die Erläuterung des Tools Prometheus inklusive der Einsatzbeschreibung. In Abschnitt 2.4 wird kurz auf die Datenbank PostgreSQL und die Herausforderung eines sicheren Speicherplatzes eingegangen. Im letzten Teil dieses Kapitels, in Abschnitt 2.5 wird auf die Grundlagen des Maschinellen Lernens eingegangen. Dabei wird in Abschnitt 2.6 der Algorithmus des Q-Lernens vertiefend behandelt.

2.1 Kubernetes

Kubernetes ist ein Open-Source-System, das für die Orchestrierung von Containern entwickelt wurde. Unter Orchestrierung von Containern wird die Automatisierung der Bereitstellung, Skalierung und Verwaltung von Containern verstanden. Genaue Informationen über Kubernetes und die im folgenden beschriebenen Bestandteile stehen in der offiziellen Kubernetes-Dokumentation [17].
Alles, was im Cluster bereitgestellt werden soll, muss vorher spezifiziert werden. Dafür werden API-Objekte verwendet. In diesen wird z.B. festgelegt, welches Container-Image verwendet werden soll und welche Ressourcen genutzt werden sollen bzw. dürfen.
Nachdem genau beschrieben wurde, wie die Objekte aussehen sollen, sorgt die Control-Plane dafür, dass die Objekte erstellt werden und den gewünschten Status auch beibehalten. Dafür werden verschiedene Aufgaben, wie etwa das Erstellen und Löschen von Containern, automatisiert durchgeführt.
Kubernetes ist dafür gedacht, dass es in einem Cluster läuft. Es gibt immer mindestens einen Kubernetes-Master und je nach Clustergröße noch zusätzliche Master- und

Arbeiter-Knoten. Die zusätzlichen Master-Knoten sind notwendig, wenn der Kubernetes-Cluster hochverfügbar sein soll.
In Abbildung 2.1 ist die Architektur von Kubernetes schematisch abgebildet. Auf den Master-Knoten läuft die Control-Plane, die auch die Kommunikation mit kubectl und dem Benutzer entgegennimmt. Auf den Arbeiter-Knoten gibt es jeweils zwei Prozesse: Den kubelet-Prozess, der sich um die Kommunikation mit dem Master kümmert und den kube-proxy, der sich um die Netzwerkdienste kümmert.
Darüber hinaus existiert auf allen Knoten ein Programm zur Containerisierung. Das kann, wie auf der Abbildung 2.1, Docker sein. Es existieren aber auch die Alternativprogramme CRI-O [6] und containerd [5].
Auf den Arbeiter-Knoten sorgt das Programm zur Containerisierung dafür, dass die Pods und Container der Benutzer laufen. Auf den Master-Knoten geht es um die Bereitstellung der Control-Plane-Komponenten, welche in der Regel auch in Containern ausgeführt werden.
Im Folgenden werden die einzelnen Bestandteile der Control-Plane genauer beschrieben:

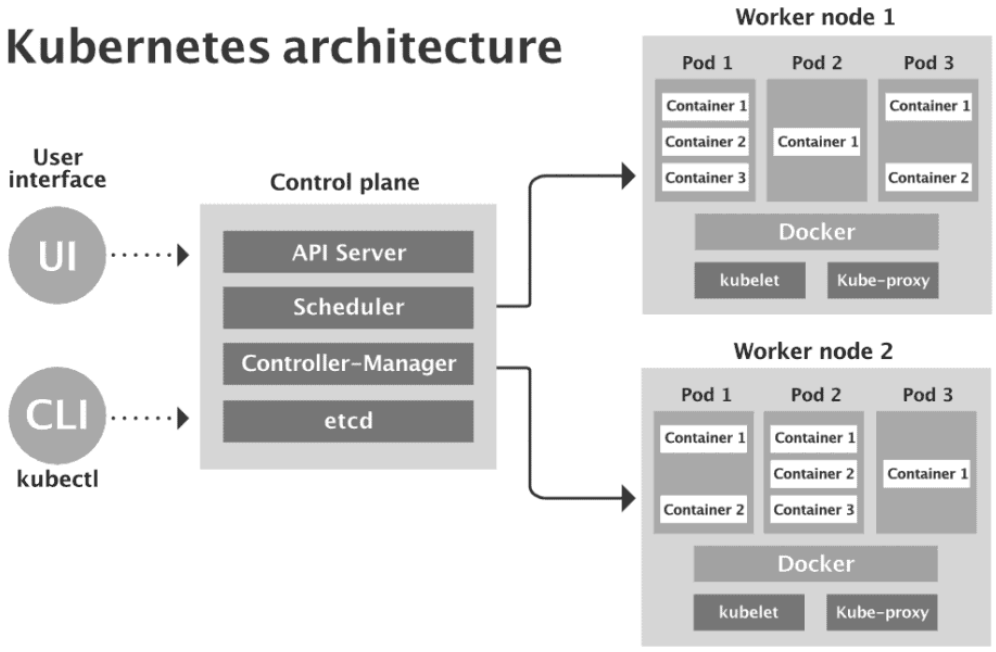

Abbildung 2.1: Kubernetes Architektur [37]

Api-Server

Der Api-Server validiert und konfiguriert die Objekte. Darüber hinaus werden REST-Operationen bearbeitet und der Status der Objekte kommuniziert.
Eine REST-Schnittstelle unterstützt die Anwendungen in ihrer Kommunikation untereinander. Dafür wird meistens das HTTP-Protokoll verwendet und die Daten mithilfe von JSON übertragen. Die Adressierung erfolgt mithilfe eines Pfades, der sich wie folgt zusammensetzt:
Bei den API-Kern-Gruppen beginnt der Pfad mit "/api/v1". Danach folgt die entsprechende Ressource wie auf Abbildung 2.2 dargestellt. Alternativ gibt es noch die benannten Gruppen, bei denen der Pfad mit "/apis/GRUPPEN-NAME/VERSION" beginnt. Hier können unterschiedliche Gruppen mit verschiedenen Versionen stehen.

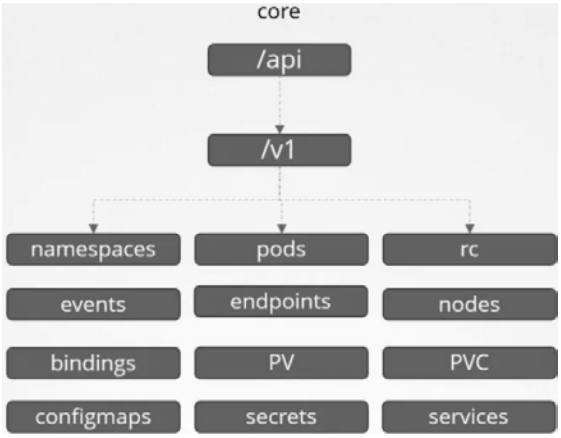

Abbildung 2.2: API-Kern-Gruppen [2]

Controller-Manager

Der Controller-Manager beinhaltet den zentralen Kontrollablauf (Control-Loop) und regelt so den Zustand des Gesamtsystems. Dadurch wird sichergestellt, dass der Ist-Zustand dem spezifizierten Soll-Zustand entspricht. Weitere Informationen über den genauen Ablauf folgen in Kapitel 2.1.2 auf Seite 11.

Scheduler

Der Scheduler ist für die Verteilung der Pods auf die unterschiedlichen Arbeiter-Knoten zuständig und ist dafür verantwortlich, dass dort die benötigten Ressourcen vorhanden sind.

etcd

Bei etcd handelt es sich um einen hochverfügbaren Key-Value-Speicher. Hier speichert Kubernetes alle relevanten Daten zum Cluster ab.

Zusätzlich zu der bisher beschriebenen physischen Trennung zwischen Arbeiter- und Master-Knoten gibt es noch die logische Trennung mithilfe der Namensräume.

Namensräume

Namensräume bilden eine logische Trennung innerhalb eines Kubernetes-Clusters. Wenn eine Ressource bereitgestellt wird, dann muss normalerweise ein Namensraum mitgegeben werden, in dem das Objekt abgelegt werden soll. Wenn dieses nicht passiert, dann wird der Standardnamensraum "default" verwendet.
Es gibt Ausnahmen von dieser Regel. Beispiele sind WebHooks, die in Abschnitt 2.1.3 beschrieben werden oder permanenter Speicherplatz, welcher in Abschnitt 2.4.1 erläutert wird. Diese beiden Ressourcen besitzen keinen Namensraum und sind clusterweit gültig.
Bei sehr kleinen Clustern reicht meistens der Standardnamensraum "default". Bei größeren Clustern macht es Sinn, mit mehreren Namensräumen zu arbeiten und unterschiedlichen Teams und Projekten unterschiedliche Namensräume zuzuweisen.
Dieses bietet zum einen den Vorteil, dass die Namen von Ressourcen dann nur noch innerhalb ihres Namensraumes eindeutig sein müssen und zum anderen können die Rechte gezielter gesetzt werden.

2.1.1 Pods

Ein Pod ist die kleinste bereitstellbare Recheneinheit, die in Kubernetes erstellt und verwaltet werden kann. Häufig befindet sich in einem Pod genau ein Container. Es gibt aber auch die Möglichkeit, dass sich mehrere Container in einem Pod befinden. Das macht Sinn, wenn Ressourcen gemeinsam genutzt werden müssen oder der Kommunikationsaufwand zwischen den Containern groß ist. Daher werden auch alle Container und die benötigten Ressourcen eines Pods immer auf dem gleichen Knoten bereitgestellt.

In der YAML-Datei eines Pods befinden sich neben der Information, wie viele Container mit welchen Images gestartet werden sollen, zusätzlich noch Spezifikationen für die Speicher- und Netzwerkressourcen sowie für die CPU- und Arbeitsspeicherressourcen.
Wenn der Pod von einem Deployment kontrolliert wird, dann wird nicht der Pod spezifiziert, sondern alles im Deployment beschrieben.

Deployment

Ein Deployment ist in Kubernetes für das Managen neuer Versionen von Pods zuständig. Es nutzt dafür das Rolling-Update, das schrittweise Updates der Instanzen durchführt und dabei auch kontrolliert, ob die neuen Instanzen funktionieren. Dadurch wird sichergestellt, dass die Anwendung die ganze Zeit erreichbar bleibt. Genauere Einzelheiten können in der Dokumentation [8] und der Beschreibung über Rolling-Updates [35] nachgelesen werden.
Über Deployments wird auch geregelt, wie viele Replikate von einem Pod erstellt werden. Soll die Anzahl der Replikate für einen Pod verändert werden, dann muss dieses nur in der YAML-Datei des Deployments geändert werden und das Deployment kümmert sich um die Ausführung.

In der Abbildung 2.3 auf Seite 11 ist übersichtlich dargestellt, welche Auswirkungen Limit- und Anfrage-Spezifikationen haben und welche Folgen eine zu geringe Angabe haben kann. So kann zu wenig CPU zu verlängerten Wartezeiten und zu wenig Arbeitsspeicher sogar zu einem Ende des Containers führen. Entsprechend werden meistens zu viele statt zu wenige Ressourcen angefragt bzw. die Limits deutlich höher angesetzt oder gar nicht erst angegeben.
Allerdings wird beim Erstellen eines Pods nur garantiert, dass die angefragten Ressourcen auf dem Knoten vorhanden sind. Das Limit ist nur relevant, wenn sich noch ungenutzte Ressourcen auf dem Knoten befinden. Stehen auf einem stark ausgelasteten Knoten nur die garantierten Ressourcen zur Verfügung, dann ist die Limit-Angabe ohne Bedeutung. Aus diesem Grund kann es auch zu einem "Out-of-Memory-Fehler" und einem Beenden des entsprechenden Containers kommen, wenn das angegebene Limit noch nicht erreicht ist.
Die Entscheidung, wie mit Anfrage und Limit von CPU und Arbeitsspeicher umgegangen wird, hat grundlegende Auswirkungen auf die Stabilität des Systems. In [41] und [42] beschreibt Herr Yellin, warum es sinnvoll ist, bei der CPU das Limit nicht zu setzen und beim Arbeitsspeicher das Limit gleich der Anfrage zu setzen.
Pods sind nicht dazu gedacht, verändert zu werden. Werden Spezifikationen verändert,

beispielsweise mehr Ressourcen zugewiesen, dann werden neue Pods nach der Vorlage erstellt und die alten im Anschluss entfernt. Wird ein Pod entfernt, dann werden auch alle Ressourcen, die zu diesem Pod gehören, mit entfernt. Existieren in einem Container Daten, die nach einem Neustart noch verfügbar sein müssen, dann muss dem Pod ein ständiger Speicher zugewiesen werden. Weiterführende Informationen zu ständigen Speichern werden in Abschnitt 2.4.1 erläutert.

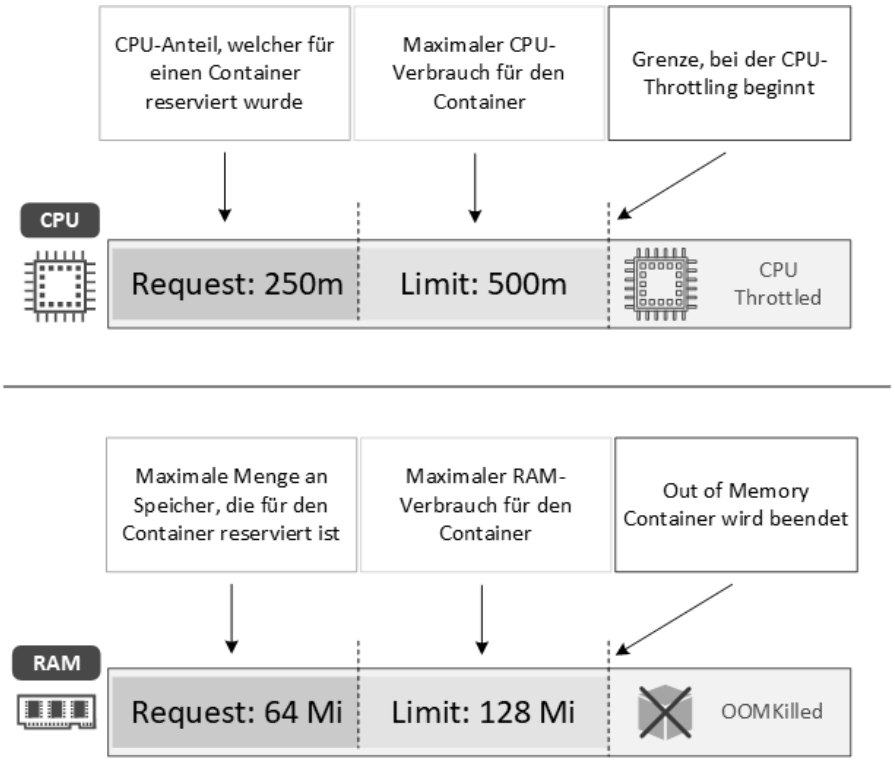

Abbildung 2.3: Vergleich der Auswirkung von Anfrage und Limit [34]

2.1.2 Controller

Ein Controller überwacht eine Ressource in Kubernetes. Er ist dafür verantwortlich, dass der Zustand dieser Ressource der Spezifikation entspricht. In Kubernetes existieren viele verschiedene Ressourcen. Entsprechend existieren dort auch viele verschiedene Controller, die vom Controller-Manager überwacht werden.
Mehr Details zu den Controllern können in der Dokumentation [15] nachgelesen werden.

Die Controller sind in der Lage, selber Aktionen wie z.B. das Erstellen oder Löschen von Ressourcen auszuführen. Im Regelfall senden sie aber eine Nachricht an den API-Server, damit dieser die gewünschte Aktion ausführt. Durch die vielen verschiedenen Controller, die die unterschiedlichen Ressourcen überwachen, wird sichergestellt, dass sich ein verändernder Cluster immer wieder dem gewünschten Zustand annähert.
Zusätzlich zu den kubernetes-internen Controllern gibt es auch die Möglichkeit, eigene Controller zu erstellen, die dann spezielle Ressourcen überwachen. Ein häufiges Beispiel ist eine Datenbankanwendung, die im Cluster laufen soll. Die dafür benötigte Überwachung ist zu komplex für die kubernetes-eigenen Controller. Diese benutzerdefinierten Controller werden dann Operatoren genannt. Zum Erstellen eigener Controller gibt es die "Controller-Runtime-Bibliothek" von Kubernetes, die verschiedene Möglichkeiten zum Überwachen und Abgleichen von Ressourcen enthält.

Operatoren sind Software-Erweiterungen für Kubernetes, die benutzerdefinierte Ressourcen überwachen können, um Anwendungen zu verwalten. Parallel zu den benutzerdefinierten Ressourcen können auch kubernetes-interne Ressourcen überwacht werden.
Die benutzerdefinierten Ressourcen werden "Custom Resources" (CR) genannt und sind Objekte von "Custom Resource Definitions" (CRD). CRDs erweitern die vorhandene API um eigene REST-Schnittstellen. Um die Prinzipien von Kubernetes nicht zu verletzen, nutzen Operatoren dafür den Control-Loop, den auch die kubernetes-eigenen Controller nutzen. Der Ursprungsgedanke dahinter war, dass Kubernetes auch komplexe Anwendungen automatisiert überwachen kann.

Control-Loop

Der Control-Loop ist eine Schleife, die in den Controllern und Operatoren die Reconcile-Funktion aufruft. Diese Reconcile-Funktionen vergleichen den Ist-Zustand ihrer Ressource mit dem spezifizierten Soll-Zustand. Werden Differenzen festgestellt, dann wird versucht, den Soll-Zustand wieder herzustellen.
Die Reconcile-Funktion wird immer dann vom Control-Loop aufgerufen, wenn ein Ereignis an einer zu überwachenden Ressource des Controllers oder Operators eintritt oder ein Zeitmesser abgelaufen ist. Zu überwachende Ressourcen können CRs sein, oder alternativ kubernetes-interne Ressourcen, wie z.B. Pods.

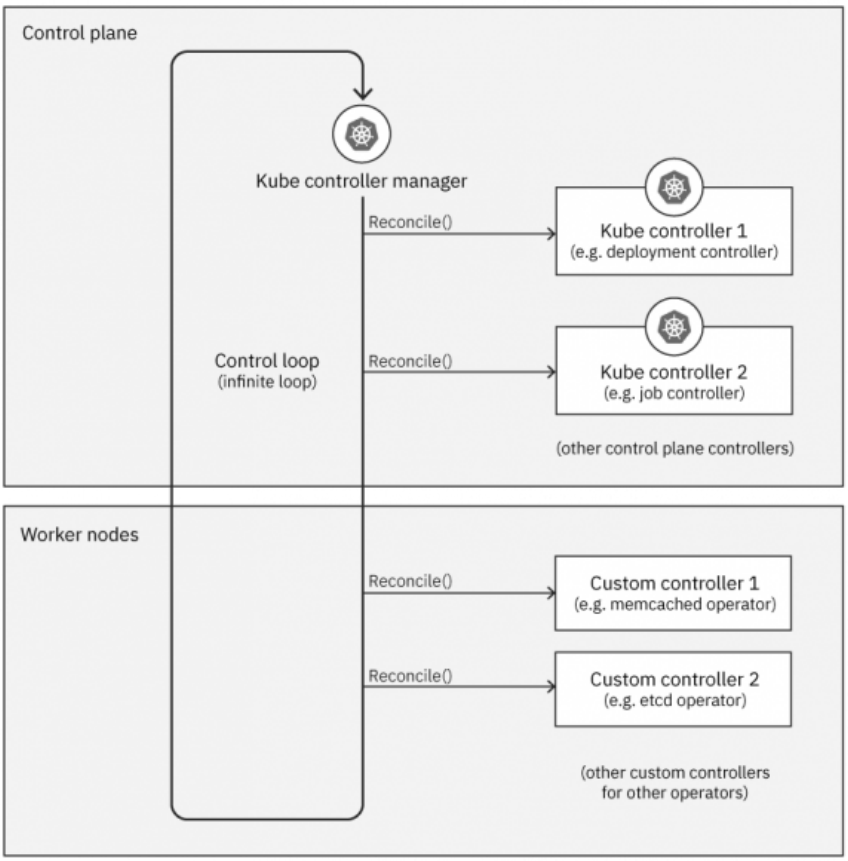

Abbildung 2.4: Control-Loop [40]

In der Abbildung 2.4 ist der Control-Loop, der die Reconcile-Funktionen steuert, schematisch dargestellt. In der Control-Plane auf dem Master-Knoten laufen die Reconcile-Funktionen für die kubernetes-internen Controller. Auf den Arbeiter-Knoten sind die Reconcile-Funktionen der Operatoren aktiv.

2.1.3 Kubernetes-WebHooks

Ein WebHook ist ein HTTP-Callback, welcher sich am API-Server anmelden kann. Dort wird festgelegt, wann er benachrichtig werden soll. Dieses könnte z.B. erfolgen, wenn ein neuer Pod erstellt wurde. Genauere Informationen zu WebHooks in Kubernetes befinden sich in der Dokumentation [18].

In Kubernetes gibt es zwei Arten von WebHooks: den verändernden und den validierenden. Der verändernde WebHook wird immer zuerst aufgerufen. Erst nach der Schema-Validierung wird der validierende WebHook aufgerufen.

Verändernder WebHook:

Verändernde WebHooks können Einstellungen an den Objekten verändern, beispielsweise Ressourcenanfragen oder -limits. Da dadurch das Schema verändert werden kann, muss anschließend eine Schema-Validierung stattfinden.

Validierender WebHook:

Der validierende WebHook kontrolliert die Einhaltung der gestellten Anforderungen. Bei Nichteinhaltung, kann ein Objekt abgelehnt werden.
Da die validierenden WebHooks immer zuletzt aufgerufen werden, können nur diese garantieren, dass sie den endgültigen Zustand eines Objektes kennen.

2.2 Operator-SDK

Das Operator-SDK ist Bestandteil des Operator-Frameworks, welches die Erstellung und Verwaltung von Operatoren vereinfachen soll. Dieses geschieht durch die Abstraktion der Erstellung und Konfiguration im Operator-SDK, sowie die Möglichkeiten zur späteren Verwaltung mithilfe des Operator-Lifecycle-Managers (OLM).
Genauere Informationen dazu können in der Operator-SDK-Dokumentation [23] und der OLM-Dokumentation [22] nachgelesen werden. Einen allgemeinen Überblick gibt es in der Dokumentation des Operator-Frameworks [21]. In dieser Arbeit wird nur das Operator-SDK verwendet und nicht das OLM.

Das Operator-SDK nutzt das Kubernetes-Controller-Runtime-Projekt. Dieses Projekt garantiert die Kompatibilität zur Kubernetes-API und stellt unter anderem den Manager zum Aufsetzen des Operators zur Verfügung. Genauere Informationen zum Controller-Runtime-Projekt können in der Dokumentation unter [16] nachgelesen werden.
Neben dem Kubernetes-Controller-Runtime-Projekt wird noch das Tool Kubebuilder vom Operator-SDK verwendet. Dieses Tool wird für die Generierung der YAML-Dateien genutzt. Es handelt sich bei Kubebuilder ebenfalls um ein Framework, das aber für die Erstellung von CRDs für die Kubernetes-API entwickelt wurde. Genauere Informationen

zu Kubebuilder gibt es in der Kubebuilder-Dokumentation unter [14].
Um das Operator-SDK nutzen zu können, muss dieses zuerst installiert werden. Anschließend werden mithilfe von Terminal-Befehlen alle benötigten Dateien für einen Operator erstellt. Der so erstellte Operator kann bei Bedarf um einen WebHook erweitert werden. Eine Besonderheit des Operator-SDK ist, dass mehrere Sprachen für die Erstellung eines Operators angeboten werden. Zum einen ist das Golang, in welcher auch Kubernetes geschrieben wurde, zum anderen wird eine Erstellung in Ansible und mithilfe von HELM-Charts angeboten. In dieser Ausarbeitung wurde sich nur mit der Golang-Variante beschäftigt. Die Konfiguration des Operators erfolgt in den go-Dateien. Durch Generatoren werden aus diesen die benötigten YAML-Dateien generiert. Diese YAML-Dateien enthalten die Spezifikationen und werden später an Kubernetes übergeben.
Das Operator-SDK hat darüber hinaus noch die Funktion, die Erstellung von Operatoren zu standardisieren. Dieses soll auch eine Vereinheitlichung auf der OperatorHub.io-Internetseite [24] schaffen. Diese Internetseite bietet eine Vielzahl von bereits fertigen Operatoren für viele unterschiedliche Problemstellungen, die kostenlos von der Community zur Verfügung gestellt werden. In der Abbildung 2.5 ist ein Ausschnitt der OperatorHub.io-Internetseite abgebildet, auf dem einige der Operatoren zu sehen sind.

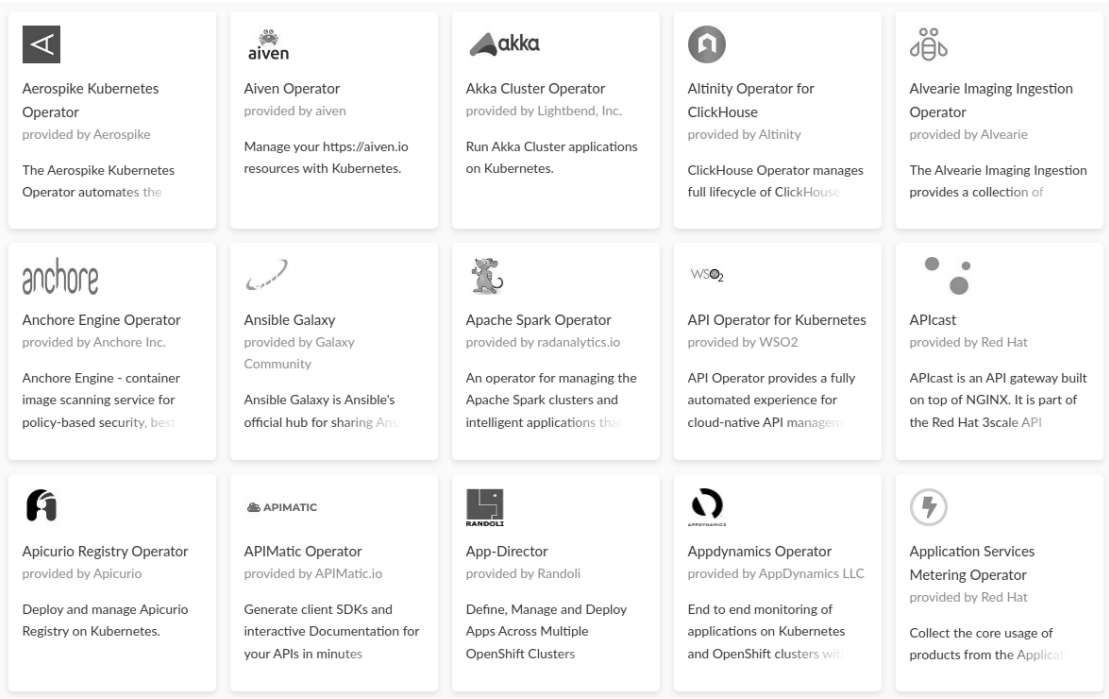

Abbildung 2.5: Internetseite OperatorHub.io [24]

Golang-Kontext

Für die komplette Kommunikation innerhalb von Kubernetes werden Golang-Kontexte genutzt. Ein Golang-Kontext hat die Aufgabe einem HTTP-Request Informationen oder eine Deadline mitzugeben. Darüber hinaus kann er genutzt werden, um komplette Aufrufketten abzubrechen. Ein Kontext ist eine Goroutine. Eine Goroutine hat Ähnlichkeiten mit einem Thread, ist aber deutlich leichtgewichtiger.
Der Kontext wird am Beginn des Aufrufes erstellt und dann durch die ganze Aufrufkette, über API-Grenzen hinweg, propagiert. Jeder Beteiligte in dieser Kette hat die Möglichkeit, den Kontext zu übernehmen oder einen Unterkontext davon zu bilden und diesem dann weitere Informationen mitzugeben. Wird der ursprüngliche Kontext beendet, enden automatisch auch alle Unterkontexte.

2.3 Prometheus

Um die Daten in dem gegebenen Kubernetes-Cluster zu sammeln, wird Prometheus genutzt. Die von Prometheus gesammelten Metriken werden als Zeitreihendaten gespeichert. Pods können weitere spezielle Metriken bereitstellen, die über das hinausgehen, was Kubernetes bereits bekannt ist. Allgemeine Metriken, wie beispielsweise die angefragten Ressourcen, werden bereits durch Kubernetes veröffentlicht. Weiterführende Informationen stehen in der Prometheus-Dokumentation [29] und im GitHub-Repository [30].

Wenn Prometheus im Cluster installiert ist, kann der Operator über einen Prometheus-Golang-Client auf Prometheus zugreifen und die Informationen von dort abfragen. Dafür kommuniziert der Client mit der Prometheus-HTTP-API. Der Client ist Bestandteil des Prometheus-Projektes, welches Open-Source ist und von der Cloud Native Computing Foundation verwaltet wird. Weitere Informationen sind unter [28] zu finden.
Für die Anfrage an die API verwendet der Client eine prometheus-eigene Abfragesprache, die Prometheus Query Language (PromQL). Diese Abfragesprache kann auch über eine Weboberfläche oder von Visualisierungstools, wie beispielsweise Grafana, genutzt werden.

Prometheus garantiert nicht, dass immer alle Metriken existieren. Bei Ausfall von Ressourcen fehlen die entsprechenden Metriken in den Zeitreihendaten. Das Ignorieren fehlender Daten hat den Vorteil, dass Prometheus stabiler läuft, jedoch sorgt es für eine gewisse Ungenauigkeit in den Daten. Daher ist Prometheus gut geeignet, um einen Überblick über den Cluster zu bekommen, aber ungeeignet, wenn es auf Genauigkeit oder Details ankommt.

2.4 PostgreSQL

Zum Speichern von Informationen über Pods wird die Datenbank PostgreSQL verwendet. Genauere Informationen können auf der PostgreSQL-Internetseite [27] nachgelesen werden.
PostgreSQL ist eine objekt-relationale Datenbank. Das bedeutet, dass die Datenbank intern mit Objekten arbeitet, was das Speichern von beispielsweise Bildern, Texten oder JSON ermöglicht. Andererseits bietet PostgreSQL auch die Möglichkeit, Daten wie in klassischen relationalen Datenbanken abzuspeichern. Dieses hat den Vorteil, dass eine SQL-ähnliche Anfragesprache unterstützt wird. Durch diese Mischung ist die Datenbank vielfältig einsetzbar.

2.4.1 Ständiger Speicher

Die Datenbank benötigt zum Speichern der Daten einen Speicherplatz, der dauerhaft existiert und unabhängig vom Datenbank-Pod verwaltet wird. Nur so kann sichergestellt werden, dass ein Neustart des Datenbank-Pods keine Datenverluste zur Folge hat.
Dafür gibt es in Kubernetes die API-Ressourcen Persistent-Volume und Persistent-Volume-Claim. Genauere Informationen zu den beiden Ressourcen gibt es in der Dokumentation [31]. Im Folgenden werden die beiden Ressourcen genauer beschrieben:

Persistent-Volume:

Das Persistent-Volume (PV) ist ein Speicherplatz im Cluster, der bereitgestellt werden muss. Dieses passiert üblicherweise mittels "dynamic volume provisioning". Diese dynamische Volumen-Bereitstellung löst die bisher übliche Vorgehensweise

ab, dass Administratoren Speicherplatz manuell bereitstellen müssen. Es handelt sich beim PV um eine eigenständige Cluster-Ressource, die unabhängig von Pods ist. Daher kann der Datenbank-Pod neu gestartet werden, ohne dass dieses Auswirkungen auf das PV hat.

Persistent-Volume-Claim:

Das Persistent-Volume-Claim (PVC) ist eine Anfrage nach Speicherplatz und bietet unterschiedliche Zugriffsmodi. Die häufigsten Modi sind Read-Write-Once, Read-Only-Many und Read-Write-Many. Benötigt ein Pod oder Deployment einen PV, dann wird zunächst ein PVC erstellt und diesem Pod oder Deployment zugewiesen. Diese Zuweisung erfolgt in der entsprechenden YAML-Datei. Kubernetes teilt diesem PVC dann ein passendes PV zu. PV und PVC sind damit eindeutig einander zugeordnet und gehören zu dem anfordernden Pod oder Deployment. Kubernetes garantiert, dass die angefragte Speichermenge mindestens in dem PV vorhanden ist. Es kann aber sein, dass das PV größer ausfällt.

2.5 Maschinelles Lernen

Der Bereich des Maschinellen Lernens (ML) gehört zur künstlichen Intelligenz. Es handelt sich um ein Teilgebiet der Informatik und soll es Maschinen ermöglichen, komplexere Problemstellungen zu lösen. Die Hintergründe und Grundlagen sind gut in der Fraunhofer Publikation [9] beschrieben.
Der Grundgedanke hinter ML ist, dass die Maschinen nicht nur den explizit programmierten Lösungsweg nutzen, sondern auch durch Lernen neues Wissen hinzugewinnen. Dieses Vorgehen kommt immer dann zum Einsatz, wenn die existierenden Prozesse zu komplex sind, um sie durch explizites Programmieren zu erfassen.
Allerdings sind auch ML-Anwendungen nur auf einen bestimmten Kontext bezogen und immer für eine ganz bestimmte Anwendung entworfen und trainiert. Ein Modell, das z.B. für die Gesichtserkennung entwickelt wurde, ist nicht in der Lage, Texte zu übersetzen.
Die drei wichtigsten Bereiche des Maschinellen-Lernens sind überwachtes Lernen, unüberwachtes Lernen und bestärkendes Lernen. Die drei Bereiche werden im Folgenden genauer beschrieben:

Überwachtes Lernen:

Das Modell lernt anhand von gelabelten Daten. Gelabelte Daten enthalten für jeden Datenpunkt die gewünschte Lösung. Das heißt, das Modell bekommt nach jedem Lernschritt gesagt, wie das gewünschte Ergebnis lautet. Das gewünschte und das tatsächliche Ergebnis werden verglichen und die Unterschiede generieren den Lernprozess. Die Herausforderung ist häufig, dass die Daten für das Training erst mühsam erstellt werden müssen und nicht immer zur Verfügung stehen. Der Vorteil hingegen ist, dass eine gute Kontrolle darüber existiert, was das Modell erlernt.

Unüberwachtes Lernen:

Beim unüberwachten Lernen sind die Daten nicht gelabelt. Das hat den Vorteil, dass die Daten dahingehend nicht aufwendig vorverarbeitet werden müssen. Auf der anderen Seite ist die Kontrolle über Gelerntes deutlich geringer. Gegebenenfalls fällt nicht auf, wenn etwas Falsches gelernt wird. Dieses Lernen ist beispielsweise gut geeignet, um große Datenmengen auf Strukturen hin zu untersuchen, die vorher unbekannt waren, um die Daten zu sortieren und zu gruppieren. Die Kontrolle, ob die gefundenen Strukturen relevant sind, liegt im Aufgabenbereich des Menschen.

Bestärkendes Lernen:

Das bestärkende Lernen ist der natürlichste Algorithmus. Es wird etwas ausprobiert und wenn die Entscheidung gut war, dann gibt es eine Belohnung, andererseits folgt eine Bestrafung. Das Ziel des Algorithmus ist es, die Belohnung zu maximieren. Dabei wird unterschieden, ob der kurzfristige Gewinn oder der langfristige Erfolg relevanter ist.

In der Tabelle Abbildung 2.6 sind die wichtigsten Lernverfahren abgebildet und zu welchem der drei beschriebenen Lernstile sie gehören. Das Lernverfahren Rückwärtspropagierung kann, je nach Modell, sowohl überwacht als auch unüberwacht durchgeführt werden. Weitere Informationen zur Rückwärtspropagierung folgen auf Seite 25.

Lernstiel	Lernaufgabe	Lernverfahren	Modell
überwacht	Regression	lineare Regression	Regressionsgerade
	Klassifikation	logistische Regression	Trennlinie
		Iterative Dichotomizer (ID3)	Entscheidungsbaum
		Stützvektormaschine (SVM)	Hyperebene
unüberwacht	Clustering	K-Means	Clustermittelpunkte
	Dimensionsreduktion	Kern-Hauptkomponentenanalyse	Zusammengesetzte Merkmale
überwacht	Bildanalyse	Rückwärtspropagierung	Faltende Neuronale Netze
	sequenzielle Daten		Rückgekoppelte Neuronale Netze
unüberwacht	Erstellung von Bildern		Erzeugende gegnerische Netzwerke
bestärkend	sequentielles Entscheiden	Q-Lernen	Strategien

Abbildung 2.6: Gängige Lernverfahren und ihre Modelle angelehnt an [9]

2.5.1 Lernverfahren

In diesem Abschnitt werden die in Abbildung 2.6 genannten Lernverfahren kurz vorgestellt.

Lineare Regression

Genauere Informationen zur linearen Regression können in [19] nachgelesen werden. Eine Regression hat zum Ziel, eine unbekannte Variable anhand einer bekannten Variablen vorherzusagen. Bei der linearen Regression müssen die beiden Variablen linear voneinander abhängig sein. Je höher die lineare Abhängigkeit, desto genauer ist die Vorhersage. Um eine möglichst genaue Vorhersage treffen zu können, muss eine Gerade so positioniert

werden, dass die Summe der quadratischen Abweichung der gegebenen Punkte minimal ist. Dieses ist in Abbildung 2.7 schematisch dargestellt. So wird erreicht, dass die Gerade die Werte möglichst genau wiedergibt. Anhand der Geraden lassen sich dann die y-Werte zu den bekannten x-Werten berechnen.

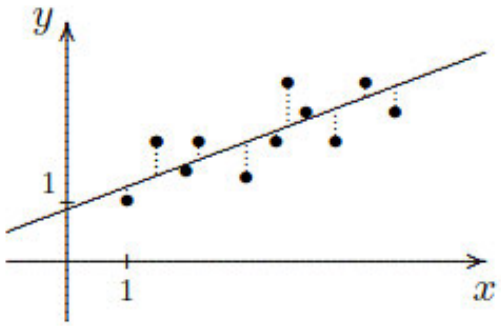

Abbildung 2.7: Regressionsgerade [19]

Logistische Regression

Genauere Details zur logistischen Regression werden unter [20] erläutert. Genau wie bei der linearen Regression ist auch bei der logistischen Regression die unbekannte Variable abhängig von der bekannten Variable. Die Abhängigkeit ist aber nicht linear, sondern nominalskaliert. Das heißt, es gibt keine Reihenfolge, sondern Kategorien, in die die Variablen einsortiert werden können.
Eine Fragestellung könnte z.B. sein, ob der Kaufentscheid für ein bestimmtes Produkt vom Alter abhängig ist. Die Kategorien wären hier "kaufen" oder "nicht kaufen" bzw. 0 oder 1.
Um die Frage zu lösen, wird die lineare Regression als Grundlage genommen und mit der logistischen Funktion erweitert. Diese ist in Abbildung 2.8 dargestellt. Das Problem der linearen Regression ist, dass dort auch Werte größer 1 und kleiner 0 auftreten können, was nach der Erweiterung mit der logistische Funktion nicht mehr möglich ist.
Die Kombination beider Funktionen ergibt dann Abbildung 2.9. So ist der Algorithmus in der Lage vorherzusagen, zu welcher Kategorie der y-Wert wahrscheinlich gehört. Im Beispiel dieser Abbildung wäre die Vorhersage für niedrige x-Werte die Kategorie 0 und

für hohe x-Werte die Kategorie 1.

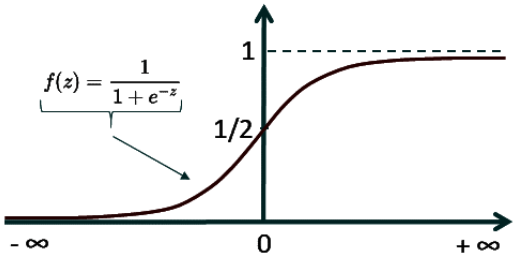

Abbildung 2.8: Logistische Funktion [20]

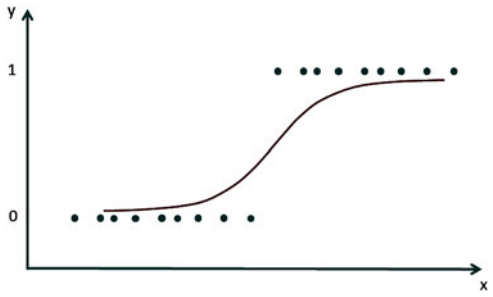

Abbildung 2.9: Kombination der logistischen und linearen Funktion [20]

Iterative Dichotomiser 3

Der Iterative Dichotomiser 3 oder ID3 ist ein Algorithmus, der Entscheidungsbäume zur Entscheidungsfindung erstellt. Entscheidungsbäume veranschaulichen hierarchische Entscheidungen. Weiterführende Informationen zu dem Algorithmus sind unter [7] zu finden. In den Knoten werden die Attribute getestet. Je nachdem wie das Attribut lautet, wird dem entsprechendem Ast gefolgt, bis am Ende in den Blättern des Baumes die Entscheidung für eine Kategorie getroffen wird.
Um so einen Baum zu erstellen, wird das Attribut mit den meisten Informationen als Wurzel genommen. Die Äste sind die Werte, die dieses Attribut annehmen kann. Für jeden Ast wird dieser Vorgang mit der Untermenge der Daten wiederholt. An den Blättern ist die Klassifikation dann eindeutig. Wichtig ist, dass jedes Attribut auf einem Pfad nur einmal vorkommen darf.

Grundsätzlich sind flache Bäume zu bevorzugen. Da der ID3-Algorithmus den Baum so lange wachsen lässt, bis die Trainingsdaten abgedeckt sind, ist die Auswahl der Trainingsdaten ein entscheidender Faktor. Es ist nicht garantiert, dass der beste kleinste Baum gefunden wird. Möglicherweise gibt es einen besseren Baum mit einer anderen, besseren Wurzel.

Stützvektormaschine

Stützvektormaschinen oder Support Vector Machines (SVM) werden meist für die Klassifikation verwendet, können aber auch für die Regression verwendet werden. SVM unterstützen sowohl die lineare als auch die nicht-lineare Klassifikation. Weitere Details können unter [32] nachgelesen werden.
Bei der nicht-linearen Trennung werden Hyperebenen erstellt, die den Raum in unterschiedliche Dimensionen trennen. Durch Hinzufügen neuer Dimensionen entstehen neue Hyperebenen. Das ist z.B. in Abbildung 2.10 gut zu erkennen. Nach der Einführung einer weiteren Dimension lassen sich die Beispieldaten durch eine Hyperebene trennen. Typische Anwendungsfelder sind in Bild- oder Texterkennungen zu finden.

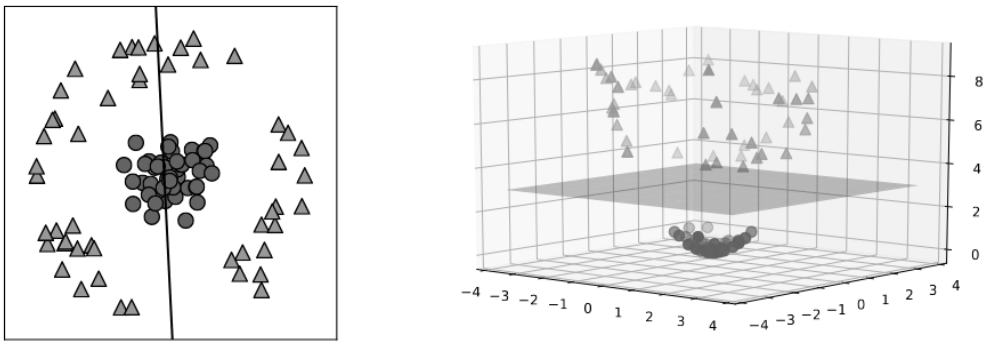

Abbildung 2.10: Erweiterung um eine Dimension für eine Hyperebene [32]

K-Means

Der K-Means-Algorithmus wird zur Clusterbildung eingesetzt. Weiterführende Details können unter [26] nachgelesen werden.
Dem Algorithmus wird die Anzahl an Gruppen mitgegeben und dieser sortiert die vorhandenen Daten in entsprechend viele Gruppen. Als Ergebnis werden die Zentren der Gruppen ausgegeben. Der Algorithmus ist linear zur Anzahl der Daten und damit sehr effizient. Aufgrund dieser Effizienz wird er gerne zur Datenanalyse im Big-Data-Umfeld eingesetzt.
Der Algorithmus legt die Anfangszentren anhand der mitgegebenen Gruppenmenge fest. Danach werden alle Daten den Gruppen zugeordnet. Anschließend werden die neuen Zentren der Gruppen berechnet. Der Zyklus mit Zuordnung der Daten und Neuberechnung der Clusterzentren wird so lange wiederholt, bis sich die Clusterzentren nicht weiter verändern.
In Abbildung 2.11 ist der Zyklus dargestellt. In **a** sind die Daten noch in ihrem ursprüng-

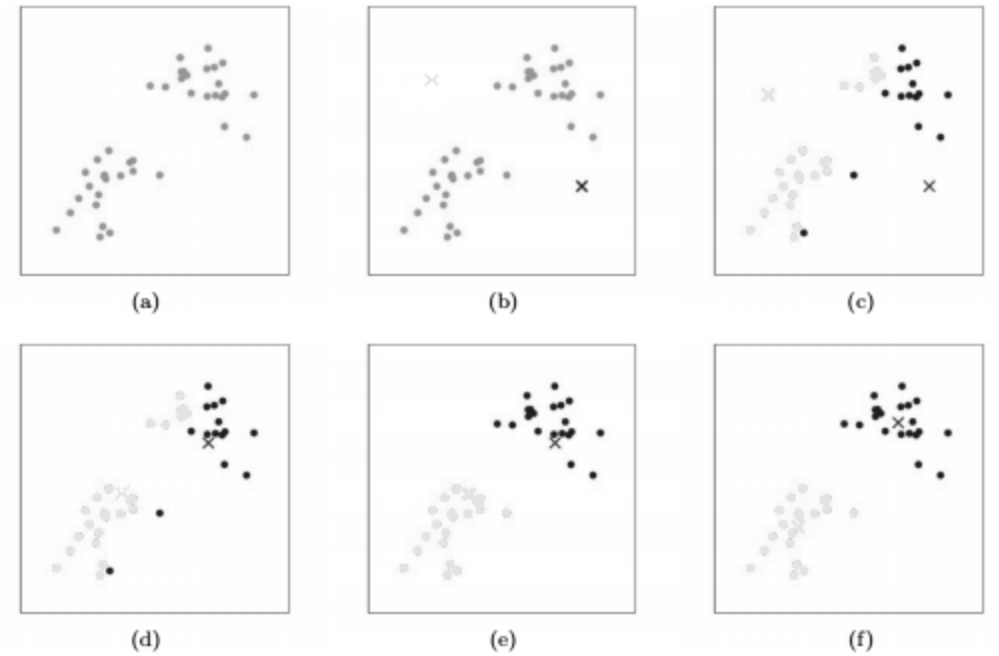

Abbildung 2.11: Ablauf des K-Means-Zyklus [26]

lichen Zustand. In **b** werden dann die Anfangszentren zufällig gesetzt. Jetzt werden die Punkte in **c** dem am nächsten gelegenen Zentrum zugeordnet. In **d** werden die Clusterzentren neu berechnet, um dann in **e** die Punkte wieder dem am nächsten gelegenen

Zentrum zuzuweisen. In **f** werden dann die Clusterzentren wieder neu berechnet. In dieser Abbildung **f** sind die Cluster jetzt schon deutlich zu erkennen.

Kern-Hauptkomponentenanalyse

Die Kern-Hauptkomponentenanalyse (KPCA) wird genutzt, um die vorhandenen Dimensionen von gegebenen Daten zu reduzieren. Dabei muss darauf geachtet werden, dass so viele Informationen wie möglich erhalten bleiben. Details können unter [11] und [43] nachgelesen werden.
Der Informationsgehalt von Daten wird durch die Varianz bestimmt. Je kleiner die Varianz, desto besser werden die Daten durch den Mittelwert bestimmt. Je größer die Varianz, desto mehr erhaltenswerte Informationen sind enthalten.
Um die Dimensionen zu reduzieren, werden die Hauptkomponenten gesucht. Die erste Hauptkomponente ist eine Linie, die den Abstand zu den Datenpunkten minimiert, so wie bei der linearen Regression. Die zweite Hauptkomponente steht senkrecht zur ersten und versucht ebenfalls den Abstand zu den Datenpunkten zu minimieren.
Je weniger Komponenten existieren, desto mehr wurden sowohl Dimensionen als auch Informationsgehalt reduziert. Als Richtwert sollen nur Komponenten mit einer Varianz größer 1 gewählt werden.

Rückwärtspropagierung

Die Rückwärtspropagierung oder auch Fehlerrückführung wird zum Trainieren von künstlichen neuronalen Netzen (KNN) verwendet. Weitere Details können in [25] nachgelesen werden.
Wie in Abbildung 2.12 zu sehen ist, findet beim Training eines KNN zuerst eine Vorwärtspropagierung statt. In dieser entscheidet das KNN, wie die Antwort auf die Aufgabenstellung lautet. Dieses Ergebnis wird mit der Zielvorgabe abgeglichen und die Differenz ermittelt. Diese Differenz wird jetzt durch das Netz zurück propagiert und die Werte in den Funktionen damit angepasst. Es gibt unterschiedliche Arten von KNN, die ihre Stärken in unterschiedlichen Bereichen haben.

Faltende neuronale Netze:

Faltende neuronale Netze arbeiten nicht mit einem Eingabewert, sondern mit einer Eingabematrix, was sich besonders zur Verarbeitung von Bildern eignet.

Rückgekoppelte neuronale Netze:

In rückgekoppelten neuronalen Netzen ist der Informationsfluss nicht strikt von vorne nach hinten, sondern enthält interne Verzweigungen. Sie sind daher besonders für sequentielle Daten geeignet.

Erzeugende gegnerische Netze:

Erzeugende gegnerische Netze bestehen aus zwei KNN, die sich gegenseitig steuern und oft für die Generierung von z.B. Bildern genutzt werden.

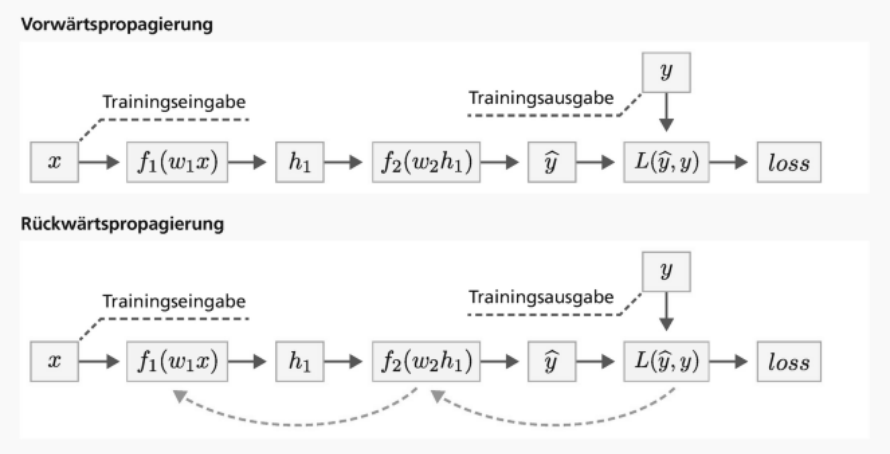

Abbildung 2.12: Ablauf der Vorwärts- und Rückwärtspropagierung [25]

2.6 Q-Lernen

In diesem Abschnitt werden die Grundlagen zum Q-Lernen-Algorithmus erläutert. In dieser Arbeit wird der Q-Lernen-Algorithmus verwendet, um eine optimierte Ressourcenanfrage zu ermitteln. Wie in Abbildung 2.6 gezeigt, handelt es hierbei um ein Verfahren des Bestärkenden-Lernens.
Der Grundgedanke von Q-Lernen ist, dass ein Agent anhand vorgegebener Grundregeln

ein bestimmtes Verhalten erlernen soll. Da der Agent nicht weiß, wie das gewünschte Verhalten aussieht, probiert er alle möglichen Aktionen aus und erhält dafür eine Belohnung oder eine Bestrafung. Mit der Zeit lernt der Agent, für welche Aktionen er in welcher Situation welche Belohnung bzw. Bestrafung bekommt und versucht nun die Belohnung zu maximieren.
Da der Agent bei jeder Entscheidung weiterlernt, kann er sich, in einem begrenzten Rahmen, an eine verändernde Umgebung anpassen.

Mathematisch stecken im Q-Lernen zwei wichtige Methoden:

- Markov-Entscheidungsprozess
- Lernen mit Zeitlicher-Differenz

2.6.1 Markov-Entscheidungsprozess

Der Markov-Entscheidungsprozess stammt vom Mathematiker Andrey Markov und ist eine Weiterentwicklung der Markov-Ketten. Diese werden verwendet, um die Wahrscheinlichkeit für das Eintreten bestimmter Ereignisse zu berechnen.

Markov-Ketten

Angenommen, in einer Beispiel-Umgebung existieren die drei Zustände Z1, Z2 und Z3, wie in Abbildung 2.13 dargestellt.
Der Wechsel zwischen den Zuständen erfolgt nach einer festgelegten Wahrscheinlichkeit. Wenn Z1 aktiv ist, dann ist der nächste Zustand mit einer Wahrscheinlichkeit von 20% Z2 und mit jeweils einer Wahrscheinlichkeit von 40% Z3 oder weiterhin Z1.
Die Fragestellung für Markov-Ketten wäre, mit welcher Wahrscheinlichkeit ist welcher Zustand nach einer bestimmten Anzahl von Zustandswechseln, aktiv. Wichtig ist hierbei, dass nur der aktuelle Zustand interessant ist. Ohne Bedeutung sind die vorher aktiven Zustände oder der Weg zum aktuellen Zustand. Um das Problem mathematisch zu lösen, werden die Informationen zunächst in einer Matrix erfasst.

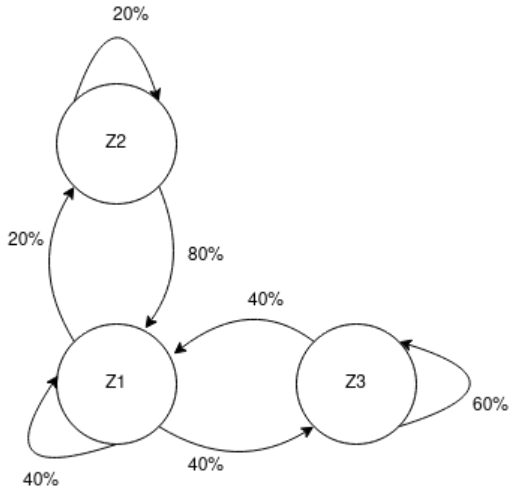

Abbildung 2.13: Beispiel einer Markov-Kette

Der Punkt m_{12} steht für die Wahrscheinlichkeit, dass von Zustand 2 zu Zustand 1 gewechselt wird. In diesem Beispiel sind das 80% oder anders ausgedrückt $\frac{4}{5}$.

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} = \begin{pmatrix} \frac{2}{5} & \frac{4}{5} & \frac{2}{5} \\ \frac{1}{5} & \frac{1}{5} & 0 \\ \frac{2}{5} & 0 & \frac{3}{5} \end{pmatrix} \tag{2.1}$$

In einem Spaltenvektor $\vec{z_t}$ wird jetzt die Wahrscheinlichkeit eingetragen, mit dem der Zustand 1 bis 3 zum Zeitpunkt t erreicht wird.

$$\vec{z_t} = \begin{pmatrix} z_{1,t} \\ z_{2,t} \\ z_{3,t} \end{pmatrix} \tag{2.2}$$

Die Wahrscheinlichkeiten von $\vec{z}$ zum Zeitpunkt t werden durch Multiplikation von M und $\vec{z_{t-1}}$ errechnet.

$$\vec{z_t} = M \cdot \vec{z_{t-1}} \tag{2.3}$$

Wobei sich $\vec{z_0}$ auf den Startzustand bezieht. Bei der Annahme, dass der Startzustand Z1 ist, würde $\vec{z_0}$ wie folgt lauten:

$$\vec{z_0} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \tag{2.4}$$

$\vec{z_1}$ würde dann wie folgt aussehen:

$$\vec{z_1} = M \cdot \vec{z_0} = \begin{pmatrix} \frac{2}{5} & \frac{4}{5} & \frac{2}{5} \\ \frac{1}{5} & \frac{1}{5} & 0 \\ \frac{2}{5} & 0 & \frac{3}{5} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{2}{5} \\ \frac{1}{5} \\ \frac{2}{5} \end{pmatrix} \tag{2.5}$$

Dieses kann jetzt für beliebig viele Zeitpunkte wiederholt werden. Die folgende Tabelle 2.1 enthält die Wahrscheinlichkeiten für die ersten Zeitpunkte des beschriebenen Beispiels.

	t=1	t=2	t=3	t=4	t=5
$z_{1,t}$	100%	40%	48%	45%	45%
$z_{2,t}$	0	20%	12%	12%	11%
$z_{3,t}$	0	40%	40%	43%	44%

Tabelle 2.1: Wahrscheinlichkeiten für die ersten fünf Zeitpunkte

Endlicher Markov-Entscheidungsprozess

Für eine endliche Anzahl von Zeitschritten mit festen Wahrscheinlichkeiten liefern die Markov-Ketten einen schnellen Überblick. Da viele Probleme dynamischer sind, wurde die Markov-Kette erweitert zum Markov-Entscheidungsprozess (MDP) (Engl. Markov decision process). Weiterführende Informationen zum MDP sind im Buch [38] ab Seite

69 nachzulesen. Wichtig ist, dass es auch beim MDP eine Sequenz von diskreten, also endlichen Zeitschritten gibt.
$t = 1, 2, 3, 4, 5, ...$
Des Weiteren gilt auch beim MDP die Markov-Eigenschaft, dass der Zustand s_{t+1} und die Belohnung r_{t+1} zum Zeitpunkt (t+1) nur von Zustand s und Aktion a zum Zeitpunkt t abhängig sind und nicht von vorhergegangenen Zuständen. Darüber hinaus haben die Mengen der Zustände, Aktionen und Belohnungen (S, A und R) alle eine endliche Anzahl an Elementen.

Im Unterschied zu den Markov-Ketten gibt es beim MDP einen Agenten. Dieser befindet sich in einem Zustand s_t. Um von einem in den nächsten Zustand zu wechseln, gibt es hier keine vorher festgelegten Wahrscheinlichkeiten mehr. Stattdessen gibt es Aktionen, die ausgewählt werden können und die bestimmen, welches der nächste Zustand s_{t+1} sein wird.
Die Auswahl der Aktion erfolgt anhand der hinterlegten Belohnungsinformationen oder nach dem Zufallsprinzip. Nach dem Zustandswechsel bekommt der Agent eine Belohnung oder Bestrafung von seiner Umgebung. Diese Belohnung bzw. Bestrafung wird in dem entsprechenden Zustands-Aktions-Paar hinterlegt.

$$Q(s, a) \leftarrow Q(s, a) + r \tag{2.6}$$

Zusammengefasst existiert in einem MDP:

- ein Agent
- eine Menge von Zuständen s
- eine Menge von Aktionen a
- eine Belohnungsfunktion r(s,a), die die Belohnung r abhängig von s und a bestimmt
- eine Wahrscheinlichkeitsfunktion $p(s_{t+1}, r|s, a)$

Gut zu sehen ist das in Abbildung 2.14. Der Agent macht eine Aktion A_t und bekommt von der Umgebung die Belohnung bzw. Bestrafung R_{t+1} und den neuen Zustand S_{t+1} mitgeteilt. Jede Aktion mit anschließender Reaktion ergibt zusammen einen Zeitschritt

t.

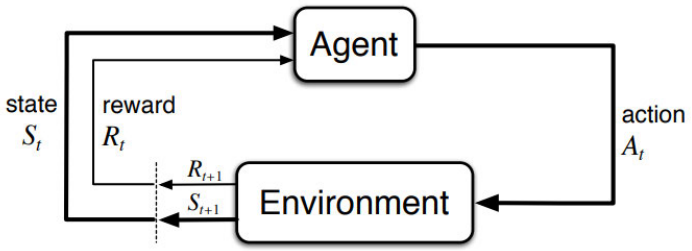

Abbildung 2.14: Die Agenten-Umgebungs-Interaktion im MDP [38]

Dabei muss unterschieden werden, ob sich der Agent nur für die sofortige Belohnung interessiert, also gierig ist oder auch für spätere Belohnungen. Letzteres kann wichtig sein, wenn nur langfristiges Planen die höchsten Belohnungen bringt. Um diesen Gierigkeits-Faktor mit in die Formel 2.6 aufzunehmen, wird γ eingefügt.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + r + \gamma \cdot Q(s_{t+1}, a_{t+1}) \tag{2.7}$$

γ kann Werte zwischen 0 und 1 annehmen und zeigt an, wie gierig der Agent ist. Wenn $\gamma = 0$ ist, dann ist der Agent sehr gierig und beachtet die zukünftigen Belohnungen nicht. Bei einem Wert von $\gamma = 1$ ist der Agent nicht gierig und misst den zukünftigen Belohnungen eine sehr hohe Bedeutung bei.

2.6.2 Lernen mit Zeitlicher-Differenz

Es gibt unterschiedliche Methoden, mit Zeitlicher-Differenz (TD) zu lernen. In diesem Kontext ist nur die TD(0) oder auch "ein Schritt TD" relevant. Die Zeitliche-Differenz bedeutet in diesem Zusammenhang, dass versucht wird, die Belohnung zwischen zwei t-Schritten zu ermitteln. Auf der einen Seite ist der bereits vorhandene Belohnungswert und auf der anderen Seite der neue Belohnungswert.
Mittels der Lernrate α kann gesteuert werden, wie viel Prozent des neuen Belohnungswertes mit wie viel Prozent des bereits existierenden Belohnungswertes addiert werden.

Bei einem $\alpha = 0,5$ werden der vorhandene Belohnungswert und der neue Belohnungswert jeweils zur Hälfte genutzt. Genauere Informationen können im Buch [38] ab Seite 141 nachgelesen werden.
TD(0) unterscheidet sich von den anderen TD-Methoden darin, dass das Lernen nach jedem Schritt erfolgt. Daher kommt auch der Beiname "ein Schritt TD".
Die Formel 2.7 des Markov-Entscheidungsprozesses sieht um TD(0) erweitert wie folgt aus:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1})] \tag{2.8}$$

Diese Formel wird für die Realisierung des Maschinellen Lernens im Operator genutzt.

3 Konzeptionierung

Im folgenden Kapitel wird das Konzept für den Operator vorgestellt. Dieser wird in einem Kubernetes-Cluster laufen und mit diesem interagieren. Daher wird in Abschnitt 3.1 zuerst auf den Operator im Kontext des Kubernetes-Clusters eingegangen. Danach folgt in Abschnitt 3.2 das Verhalten des Operators zur Laufzeit eines Pods. Im Anschluss folgt in Abschnitt 3.3 der Ablauf im WebHook, nachdem dieser durch die Kubernetes-API aufgerufen wurde.
In Abschnitt 3.4 wird auf die Datenbank und die Tabelenschemata eingegangen. Danach folgen in Abschnitt 3.5 die vorhandenen Daten in Prometheus und welche Daten davon für den Operator relevant sind. Zum Schluss folgen in Abschnitt 3.6 die Überlegungen zu den Anforderungen an den Algorithmus. Hier wird auch das Konzept für die Umsetzung des Q-Lernens-Algorithmus erläutert.

3.1 Aufbau des Operators

In Abbildung 3.1 auf Seite 34 ist der Operator im Kontext von Kubernetes zu sehen. Die Information über die Erstellung eines Pods wird an die Kubernetes-API auf dem Master-Knoten weitergeleitet. Dort wird anschließend kontrolliert, ob es einen Eintrag für einen verändernden WebHook gibt. Hier ist das der Fall. Der verändernde WebHook wird aufgerufen. Dieser kontrolliert, ob er die Ressourcenanfragen anpassen muss und sendet eine Antwort zurück an den Master. Genaueres zur Anpassung in Abschnitt 3.3 auf Seite 39.
Nachdem der Pod gegebenenfalls angepasst wurde, wird das Schema im Master validiert, um sicherzugehen, dass der Pod noch funktionsfähig ist. Anschließend wird überprüft, ob es einen Eintrag für einen validierenden WebHook gibt. Danach werden alle Informationen in die Datenbank der Control-Plane, den etcd, geschrieben.

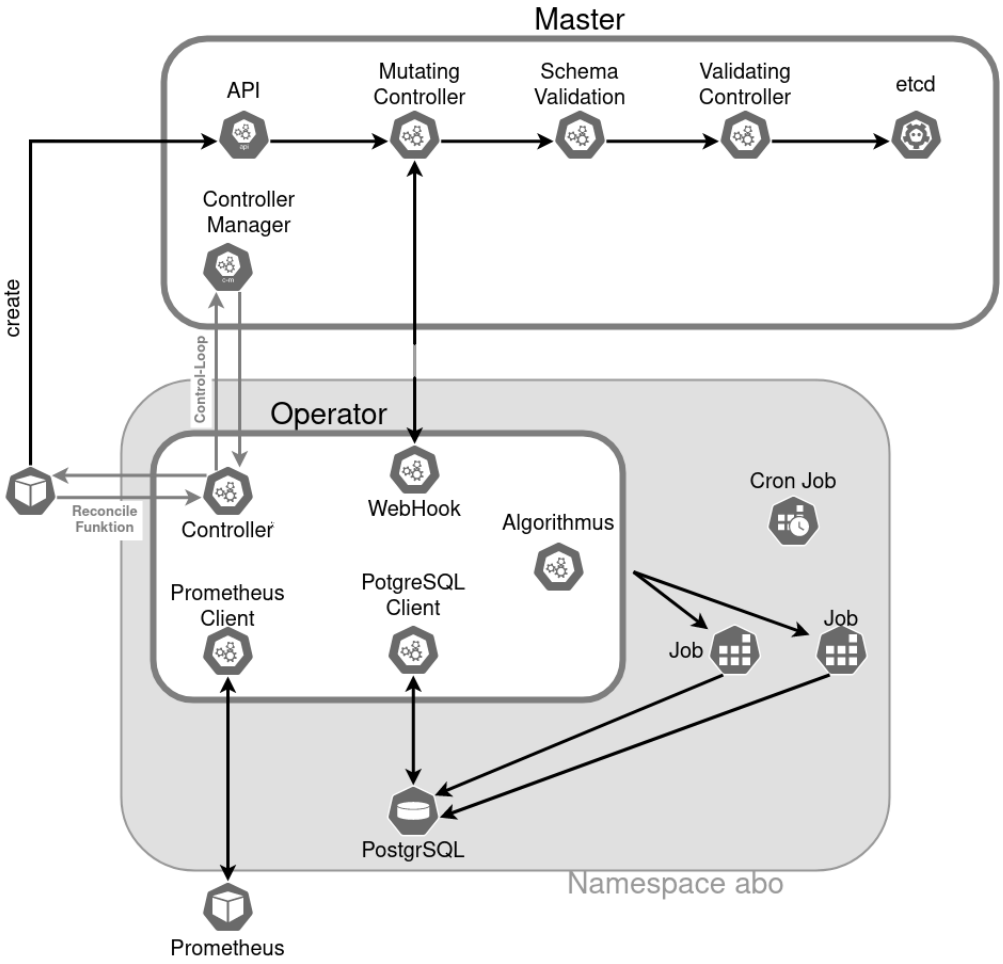

Abbildung 3.1: Der Operator im Kubernetes-Cluster

Dann wird der Pod gestartet. Nach der Erstellung des Pods wird dieser in regelmäßigen Abständen durch die Reconcile-Funktion des Operators überwacht. Der Control-Loop, der durch den Controller-Manager gesteuert wird, ruft die Reconcile-Funktion auf. Jedes Mal, wenn die Reconcile-Funktion im Operator den Pod überprüft, wird über den Prometheus-Client Prometheus aufgerufen und die aktuell verbrauchten CPU- und Arbeitsspeicher-Ressourcen abgefragt. Diese Informationen werden über einen PostgreSQL-Client in die Datenbank geschrieben.
Wenn der Pod am Beenden ist, beginnt die Verarbeitung der vorher gespeicherten Daten. Hat der Algorithmus bereits Erfahrungen mit dem entsprechenden Pod, dann werden die gemessenen Verbräuche aus der Datenbank geladen, ausgewertet und ggf. weitere Anpassungen durch den Algorithmus vorgeschlagen. Hat der Pod noch keinen Eintrag in der

Datenbank, dann werden die gemessenen Informationen genutzt, um den Algorithmus erstmalig zu trainieren.
Da das erstmalige Training des Algorithmus rechenaufwändiger ist als eine spätere Nutzung, wird das Training in einen Job ausgelagert. Dadurch werden extreme Schwankungen im Ressourcenverbrauch des Operators vermieden. Nach dem Training wird das Gelernte direkt vom Job in die Datenbank gespeichert. In der hier benutzten Kubernetes-Version kann dem Job noch keine Zeit für das automatische Abräumen mitgegeben werden. Aus diesem Grund läuft zusätzlich noch ein Cron-Job, der zyklisch die beendeten Jobs abräumt.

3.2 Verhalten des Operators zur Laufzeit

In diesem Abschnitt wird der Ablauf in der Reconcile-Funktion zur Laufzeit eines Pods genauer erläutert. Um die Abläufe besser darstellen zu können, sind Sequenzdiagramme erstellt worden. Es werden die drei Optionen "erstellen", "laufend" und "beendet" unterschieden.

Der Ablauf in der Reconcile-Funktion

Im Sequenzdiagramm 3.2 auf Seite 36 ist der Ablauf in der Reconcile-Funktion des Operators zu sehen. Als Erstes müssen die Informationen über den Pod vom Golang-Kontext geladen werden, damit darauf zugegriffen werden kann. Im zweiten Schritt wird überprüft, ob der Pod am Beenden ist. Nach Beendigung des Pods dauert es eine Weile bis er aus dem Control-Loop entfernt ist.
Während der Terminierung des Pods werden die Daten ausgewertet. Genaueres dazu im Sequenzdiagramm 3.4 auf Seite 38. Wenn der Pod nicht beendet wird, wird geprüft, ob es einen Datenbankeintrag zu diesem Pod gibt. Ist das der Fall, dann werden die aktuellen Ressourcen dieses Pods gesammelt. Genaueres dazu in Sequenzdiagramm 3.3 auf Seite 37. Existiert noch kein Eintrag, dann wird ein neuer angelegt.

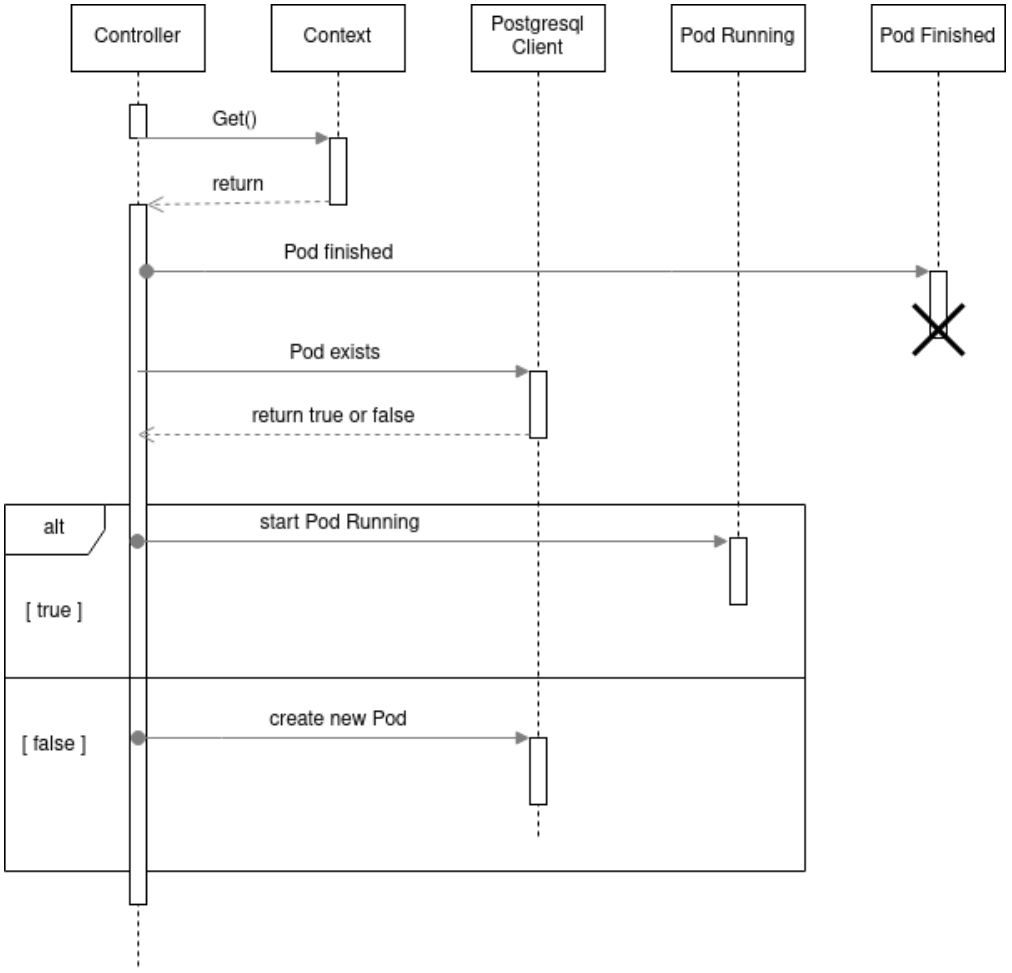

Abbildung 3.2: Die Reconcile-Funktion im Controller

Der regelmäßige Reconcile-Aufruf

Während der Pod läuft, wird er regelmäßig vom Operator überwacht. Das ist im Sequenzdiagramm 3.3 auf Seite 37 dargestellt. Existiert für die Container eines Pods ein Eintrag in der Datenbank, dann durchlaufen diese jede Minute die Reconcile-Funktion des Operators. In diesen regelmäßigen Aufrufen wird der tatsächliche Verbrauch der Container von Prometheus abgefragt und zusammen mit einem Zeitstempel in der Datenbank gespeichert.

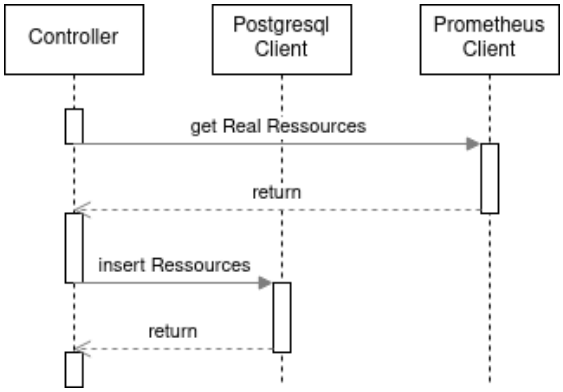

Abbildung 3.3: Der regelmäßige Reconcile-Aufruf eines Pods

Der letzte Reconcile-Aufruf

Während der Terminierung, entspricht der Ablauf dem Sequenzdiagramm 3.4 auf Seite 38. Da von einem beendeten Pod keine Informationen mehr von Prometheus gesammelt und publiziert werden, wird zunächst abgefragt, ob beim Start zu wenig Ressourcen angefragt wurden. Dafür gibt es zwei Metriken, die jeweils für CPU- und Arbeitsspeicher die Sekunden zählen, in denen die Ressourcen nicht ausgereicht haben. Anschließend werden aus der Datenbank die Informationen geladen, welche Informationen die einzelnen Container über die Zeit durchschnittlich und maximal verbraucht haben.
Um entscheiden zu können, ob der Algorithmus erst trainiert werden muss oder für den entsprechenden Container bereits trainiert wurde, wird in der Datenbank in der Tabelle qtable nach der id des Containers gesucht. Existiert diese, dann fand das Training bereits statt. In diesem Fall wird nur kontrolliert, ob der Algorithmus eine Anpassung vorschlägt, die dann in der Datenbank gespeichert wird. Wurde der Algorithmus noch nicht trainiert, dann wird ein Job gestartet, der das Training durchführt. Aus dem Job heraus werden die Ergebnisse anschließend in der Datenbank gespeichert.

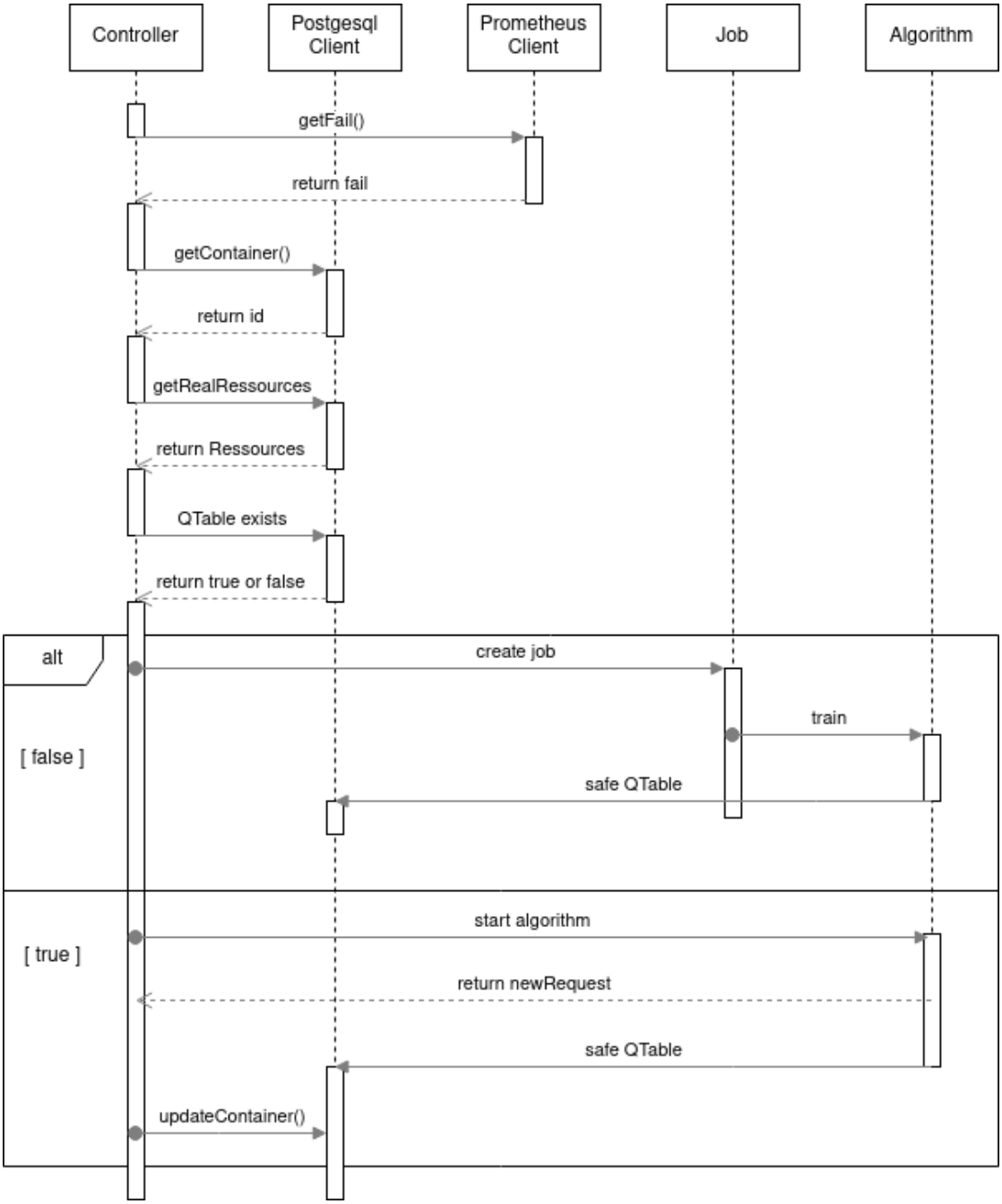

Abbildung 3.4: Der letzte Reconcile-Aufruf während der Pod beendet wird

3.3 WebHook

In diesem Abschnitt wird der Ablauf im WebHook erläutert. Dieses ist in Abbildung 3.5 zu sehen. Zuerst wird versucht, den Container aus der Datenbank zu laden. Wird kein entsprechender Container gefunden, dann wird der WebHook wieder beendet. Andernfalls wird ein "ResourceRequirement" erstellt und mit den Anfrage- und Limitinformationen aus der Datenbank gefüllt. Nachdem die neuen Ressourcenanfragen im Pod gespeichert sind, wird dieser zurück zum Master gesendet.

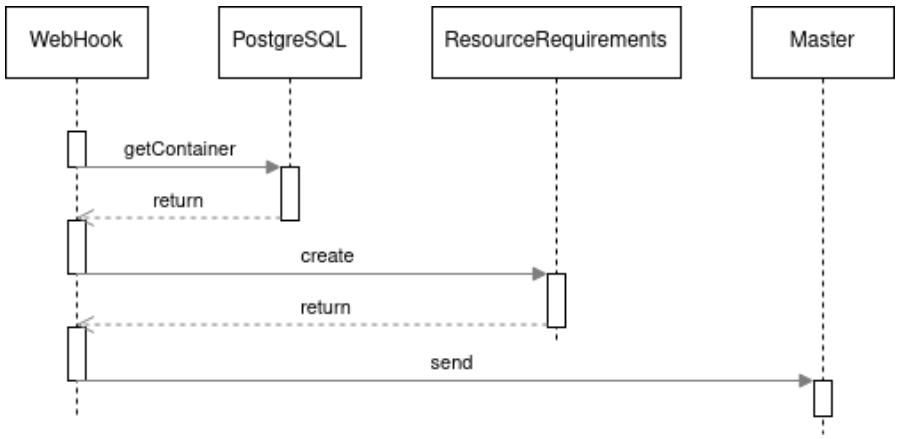

Abbildung 3.5: Der Ablauf im WebHook

3.4 Datenbank

In diesem Kapitel wird die Datenbank beschrieben. In dieser Ausarbeitung wurde PostgreSQL als Datenbank ausgewählt, da diese bereits in mehreren anderen Projekten der Firma als Datenbank genutzt wird und auch die in dieser Arbeit gestellten Anforderungen erfüllt.
Wie in Bild 3.6 dargestellt, existieren in der Datenbank drei Tabellen für die Sicherung der benötigten Daten. Die erste Tabelle ist Tabelle "pod". Die Container jedes eingehenden Pods werden anhand der Kombination eines eindeutigen Labels, dem Containernamen und dem Namensraum identifiziert. Existiert noch kein entsprechender Eintrag, dann wird beim Anlegen eines neuen Eintrages automatisch eine "ID" zugewiesen. Diese generierte "ID" findet sich in den anderen Tabellen als "pod_id" wieder. Dieses wird über

einen Fremdschlüssel realisiert. So werden die Einträge der restlichen Tabellen eindeutig einem bestimmten Container zugewiesen.
Des Weiteren werden in der Tabelle "pod" die Ressourcenanfragen gespeichert. Diese

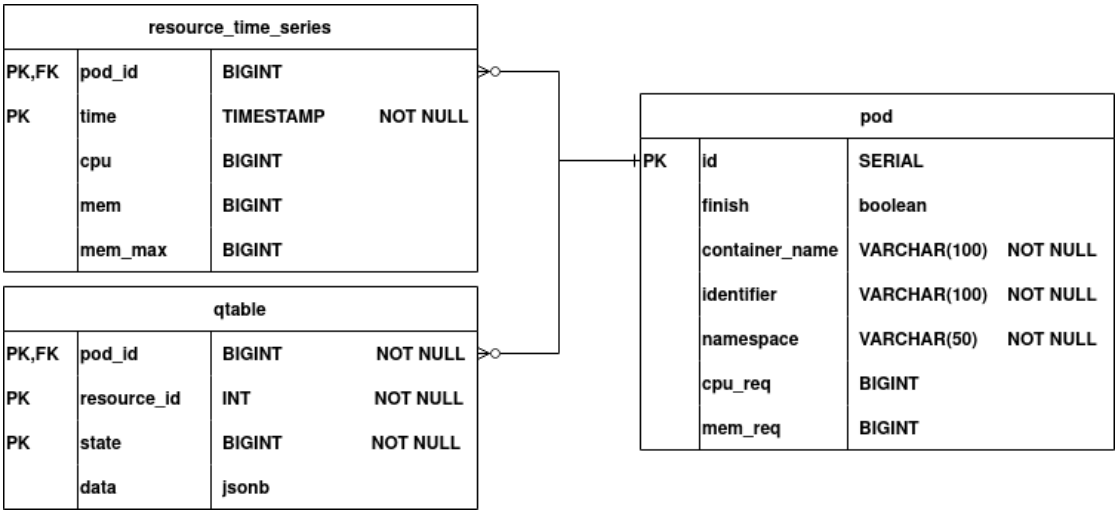

Abbildung 3.6: Tabellenschemata in der PostgreSQL

Anfragen werden vom Algorithmus optimiert und dienen dem WebHook als Grundlage für die neuen Ressourcenanfragen. Um zu verhindern, dass der Container während des Terminierens mehrfach aufgerufen wird, gibt es zusätzlich den Tabelleneintrag "finish" , in dem notiert wird, ob der Container nach dem Beenden bereits verarbeitet wird.

In der Tabelle "resource_time_series" werden die tatsächlich benötigten Ressourcen der laufenden Container gespeichert. Diese werden einmal pro Minute bei Prometheus angefragt. Da nicht zu jedem Zeitpunkt garantiert werden kann, dass Informationen über Arbeitsspeicher- und CPU-Ressourcen vorhanden sind, muss mit leeren Einträgen gerechnet werden. Nach Beendigung des Pods, werden die Einträge der entsprechenden Container aus der Tabelle "resource_time_series" ausgelesen, verarbeitet und anschließend aus der Tabelle gelöscht.
Die Ressource "Arbeitsspeicher" besitzt zwei unterschiedliche Einträge in dieser Tabelle. In der Spalte "mem" wird der aktuell genutzte Arbeitsspeicherwert gespeichert. In der Spalte "mem_max" wird der zu dem Zeitpunkt höchste genutzte Arbeitsspeicherwert gespeichert. Dadurch können auch sehr kurze Lastspitzen abgefangen werden, die sonst nicht zu sehen wären.

Die dritte Tabelle ist "qtable". In dieser Tabelle wird der Lernfortschritt des Algorith-

mus gespeichert. Die "resource_id" steht mit 0 oder 1 für CPU oder Arbeitsspeicher. Da die beiden Ressourcen unabhängig voneinander sind, werden diese auch unabhängig voneinander gelernt und gespeichert. In die Spalte "state" werden die Ressourcenmengen geschrieben. Dieser Eintrag entspricht dem Zustand s aus dem Algorithmus. Da die Menge s viele Elemente beinhaltet, gibt es für CPU und Arbeitsspeicher jeweils viele verschiedene "state"-Einträge. In der Spalte "data" wird in JSON der Lernfortschritt zu dem entsprechenden Zustand aus der Spalte "state" hinterlegt.

3.5 Vorhandene Daten in Prometheus

Prometheus sammelt von allen Ressourcen, die dieses unterstützen, die Ressourcen und stellt diese über eine API zu Verfügung. Im Folgenden werden die wichtigsten verfügbaren Informationen aufgezählt. Die für diese Arbeit relevanten Informationen sind hervorgehoben.

Informationen über die Knoten

- Maximale CPU-Frequenz in Hertz
- Minimale CPU-Frequenz in Hertz
- Name des zuständigen Masters
- Name des Arbeiter-Knotens
- Aktiver Arbeitsspeicher in Byte
- Inaktiver Arbeitsspeicher in Byte
- Port-ID
- Port-Name
- Port-Typ
- Port-Status

Informationen über Deployments

- Deployment-Name
- Namensraum
- Datum / Uhrzeit der Erstellung
- Status

Informationen über die Pods

- Pod-Name
- Container-Name
- Namensraum
- Datum / Uhrzeit der Erstellung
- Name des Arbeiter-Knotens
- Pod-IP-Adresse
- Host-IP-Adresse
- Phase

Informationen über die Container

- **Container-Name**
- Container-ID
- Container-Image
- Container-Image-ID
- **Namensraum**
- Name des Arbeiter-Knotens
- **Pod-Name**

- **CPU-Limit**
- **Arbeitsspeicherlimit in Byte**
- **CPU-Anfrage**
- **Arbeitsspeicheranfrage in Byte**
- **Status**
- Restarts
- **CPU-Nutzung**
- **Arbeitsspeichernutzung in Byte**
- Dateisystemlimit in Byte
- Dateisystemnutzung in Byte
- Speicher im Cache in Byte
- SWAP-Arbeitsspeicher in Byte
- **Throttling CPU in Sekunden**

Die Überwachung des Operators läuft auf Containerebene. Zur Identifizierung des Containers ist neben dem Identifizierungslabel auch der Namensraum und der Container-Name relevant. Darüber hinaus sind die Ressourceninformationen zu CPU und Arbeitsspeicher sowie zum Status des Containers relevant.

3.6 Auswahl des Lernverfahrens

In diesem Abschnitt wird beschrieben, welcher Algorithmus für den Operator gewählt wurde und welche Vorteile er hat. Die Anforderung an den Algorithmus ist, dass dieser die Ressourcenanfragen der Container anhand der reellen Verbräuche anpasst. Aufgrund der Vielzahl unterschiedlicher Pods mit meist mehreren Containern, gibt es eine entsprechend hohe Anzahl unterschiedlicher Verbräuche.
Für einen überwachten Algorithmus wären sehr viele gelabelte Datensätze nötig. Zur

Erstellung der gelabelten Daten könnte jedem Container der gemessene Ressourcenverbrauch zugewiesen wird. Das ergäbe einen Datensatz, der den Algorithmus für die aktuellen Container trainiert. Da die Container aber unabhängig voneinander sind und ihr Ressourcenverbrauch maßgeblich von den Anwendungen im Inneren abhängen, lassen sich bei einem neuen Pod keine Rückschlüsse auf den zu erwartende Ressourcenverbrauch ziehen. Es müsste für jeden neuen Container erneut ein Training gestartet werden. Ein überwachter Algorithmus ist daher in diesem Fall ungeeignet.
Die unüberwachten Algorithmen sind zum Sortieren und Reduzieren von Daten hilfreich. Da die Daten in diesem Zusammenhang nicht sortiert oder zusammengefasst werden müssen, ist auch diese Vorgehensweise hier ungeeignet.

Aus diesem Grund wird ein Algorithmus aus dem Bereich des bestärkenden Lernens gewählt. Das wird der Q-Lernen-Algorithmus sein.
Dem Q-Lernen werden Regeln mitgegeben, die den Rahmen festlegen. Anhand der Regeln lernt der Algorithmus, welche Ressourcenanfragen der jeweilige Container benötigt. Da nur grundlegende Regeln vorgegeben sind, ist der Algorithmus in der Lage, sich anzupassen, wenn sich die Ressourcenanforderungen eines Containers in der Zukunft ändern.

3.6.1 Q-Lernen

In diesem Abschnitt geht es um das Design des Q-Lernens. Zuerst werden die Mengen der Aktionen und Zustände beschrieben. Anschließend folgt eine Auflistung der Regeln, anhand derer die Entscheidungen getroffen werden. Zum Schluss wird der Ablauf im Algorithmus erläutert.

Mengen

Als mögliche Aktionen existieren "more", "less" und "stay".

$$a = \{more, less, stay\} \tag{3.1}$$

Die Ressourcenmengen sind die Zustände s. Um die Menge der Zustände nicht zu groß werden zu lassen, besteht die Menge s aus den Vielfachen von 64.

$$s = \{n \cdot 64 | 64, n \in \mathbb{N}\} \tag{3.2}$$

Bei der Ressource "Arbeitsspeicher" wurde die Einheit MB gewählt, damit die Zahlen nicht zu groß werden. Bei den CPU-Werten werden Milli-CPU verwendet. Dieses wurde erreicht, indem die Float-Zahl mit 1000 multipliziert wurde. Eine Ressourcenanfrage von 1000 entspricht einem Kern.

Regeln

Der Agent bekommt Regeln mitgegeben, anhand derer sich die Belohnung bzw. Bestrafung berechnet. Diese Regeln müssen das angestrebte Ziel widerspiegeln und werden für Arbeitsspeicher und CPU getrennt betrachtet.

Arbeitsspeicher:
Beim Arbeitsspeicher muss sichergestellt werden, dass der angepasste Container nicht aufgrund von zu wenig Arbeitsspeicher beendet wird. Wie in den Grundlagen beschrieben, wird der Ansatz verfolgt, beim Arbeitsspeicher das Limit gleich der Anfrage zu setzen. Aus diesem Grund sollte eine Anfrage angestrebt werden, die mindestens den maximalen Verbrauch abdeckt.
Da die Container nicht bei jedem Durchlauf den exakt gleichen Verbrauch aufweisen, muss ein Puffer eingeplant werden. Durch den Puffer ist die Anfrage immer etwas oberhalb des maximalen Verbrauchs und kann auch schwankende maximale Verbräuche zwischen den Durchläufen abfangen. Aus diesem Grund wurden für die Arbeitsspeicheranfrage folgende Regeln formuliert:

- nicht mehr Ressourcen als der maximale Verbrauch plus einen Puffer
- nicht weniger Ressourcen als der maximale Verbrauch
- Fehler aufgrund von zu wenig angefragten Ressourcen sind nicht erlaubt

CPU:
Die Menge der vorhandenen CPU-Ressourcen ist meistens geringer, weshalb es hier besonders wichtig ist, eine möglichst hohe Auslastung zu erreichen. Darüber hinaus ist die Anfrage nach mehr CPU deutlich dynamischer als die Anfrage nach mehr Arbeitsspeicher. Sollte ein Container trotzdem zu wenig CPU zugewiesen bekommen, dann wird dieser nur verlangsamt, aber nicht beendet.
Aus diesem Grund wird eine CPU-Anfrage unterhalb des maximalen Verbrauchs angestrebt. Der Container soll im Durchschnitt genügend Ressourcen zur Verfügung haben und bei Lastspitzen zusätzliche Ressourcen nachfordern. Die angestrebte Anfrage befindet sich also zwischen dem durchschnittlichen und maximalen Verbrauch.
Die Anfrage sollte jedoch hoch genug sein, um eine Drosselung der CPU zu verhindern. Eine CPU kann gedrosselt werden, wenn das Limit erreicht wurde. Da hier kein Limit gesetzt wird, kann eine Drosselung alternativ noch erfolgen, wenn die kompletten CPU-Ressourcen verteilt sind. Wenn dieser Fall eintritt, dann waren die Anfragen der Pods zu niedrig und es wurden zu viele Pods auf einem Knoten bereitgestellt. Wenn die CPU gedrosselt wurde, muss beim nächsten Start mehr CPU angefragt werden. Entsprechend wurden für die CPU-Anfrage folgende Regeln festgelegt:

- nicht mehr Ressourcen als der maximale Verbrauch
- nicht weniger Ressourcen als der durchschnittliche Verbrauch
- CPU-Drosselung ist nicht erlaubt

Ablauf

In Abbildung 3.7 ist der Ablauf grob dargestellt. Zuerst muss der Agent erstellt werden. Damit dieser weiß, um welchen Container es sich handelt, wird die ID aus der Datenbank mitgegeben. Des Weiteren muss der Agent wissen, um welche Ressource es sich handelt. Die beiden Ressourcen CPU und Arbeitsspeicher werden getrennt voneinander betrachtet. Dadurch ist der Operator nicht davon abhängig, dass immer Informationen zu beiden Ressourcen in Prometheus existieren.
Im nächsten Schritt wird der Algorithmus gestartet. Beim Start werden die Informationen zum durchschnittlichen und maximalen Verbrauch mitgegeben. Des Weiteren findet eine Abfrage statt, ob beim letzten Durchlauf des Containers zu wenig Ressourcen zugewiesen wurden. Auch diese Information wird beim Start mitgegeben.
Für den normalen Durchlauf werden als ersten Schritt die vorhandenen Informationen

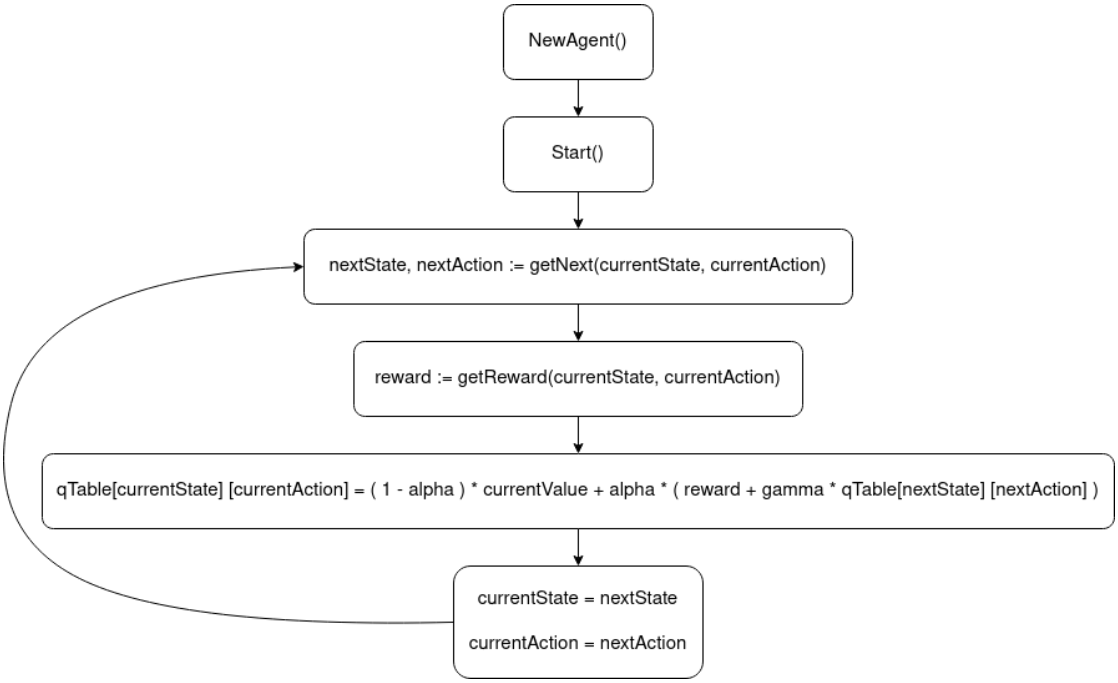

Abbildung 3.7: Ablauf des Q-Lernen-Algorithmus

aus der Datenbank in eine Map geladen. Für das Training im Job entfällt dieses. Als Einstiegspunkt in den Algorithmus wird die angefragte Ressourcenmenge genommen. Ab da entscheidet der Map-Eintrag, wohin sich die Ressourcenanfrage weiterentwickelt. Anhand der ausgewählten Aktion wird der nächste Zustand ausgewählt. Bei diesem Zustand wird dann die Aktion gewählt, die am meisten Gewinn verspricht.
Als nächster Schritt wird anhand des gewählten Zustandes, der Aktion und eventueller Fehler aufgrund von Ressourcenmangel zur Laufzeit des Containers, die Belohnung bzw. Bestrafung ermittelt. Alle Informationen werden dann in die Formel 2.8 von Seite 32 eingesetzt. Das Ergebnis wird in die Map eingetragen. Als Abschluss eines Durchlaufs werden die neuen State- und Aktion-Werte als aktuelle Werte genommen.
Nachdem der Agent ein paar Durchläufe gemacht hat und dabei neue Erfahrungen gesammelt hat, wird die veränderte Map wieder in der Datenbank abgespeichert. Der letzte Zustand s, den der Agent hat, wird als neue Ressourcenanfrage ebenfalls in der Datenbank abgelegt. Diese Information wird der WebHook beim erneuten Start des entsprechenden Containers nutzen, um die Ressourcen anzupassen.

4 Realisierung

In diesem Kapitel wird die Umsetzung des Operators beschrieben. Dabei wird zunächst in Abschnitt 4.1 auf allgemeine Informationen zur Operatorerstellung eingegangen. Darauf folgt in Abschnitt 4.1.1 eine Beschreibung der YAML-Dateien des Operators.
In Abschnitt 4.1.2 wird dann detailliert die Reconcile-Funktion des Operators erläutert und anschließend in Abschnitt 4.1.3 der WebHook beschrieben. Danach folgen in den Abschnitten 4.1.4 und 4.1.5 die Details der Clients für Prometheus und die PostgreSQL-Datenbank.
Nach den Informationen über den Operator geht es in Abschnitt 4.2 um die Abfragen an Prometheus. Als Abschluss des Kapitels wird in Abschnitt 4.3 noch auf die verschiedenen Herausforderungen eingegangen, die während der Realisierung aufgetaucht sind.

4.1 Bauen eines Operators

Zum Bauen des Operators wurde das Operator-SDK für Golang genutzt. Wie bereits in den Grundlagen in Abschnitt 2.2 auf Seite 14 beschrieben, besteht die Hauptaufgabe darin, in einem bestehenden Grundgerüst den Operator an die gestellten Anfoderungen anzupassen.
So soll der Operator nicht alle, sondern nur bestimmte Namensräume überwachen. Es muss möglich sein diese Namensräume unkompliziert anzupassen, weshalb in der YAML-Datei 4.7 des Deployments die Umgebungsvariable "WATCH_NAMESPACE" eingefügt wurde. So können die Namensräume unkompliziert angepasst werden, ohne den Code des Operators verändern zu müssen. Ein Neustart des Operator-Pods genügt, um die neuen Namensräume zu übernehmen.
Bei der Konfiguration der Namensräume im Manager muss darauf geachtet werden, dass bei mehr als einem Namensraum, diese über einen Cache übergeben werden müssen und

nicht einfach in eine Variable gespeichert werden können. Sollte das Einlesen der Namensräume fehlschlagen, dann überwacht der Operator automatisch alle Namensräume. Auf den WebHook hat diese Konfiguration keinen Einfluss. Für welche Namensräume dieser benachrichtigt werden soll, wird in seiner eigenen Spezifikation festgelegt. Die beiden Bestandteile des Operators werden unabhängig voneinander konfiguriert.

Die YAML-Dateien werden von Kubebuilder generiert. Dafür müssen Konfigurationen in den GO-Dateien angelegt werden. Die wichtigsten Kubebuilder-Konfigurationen werden vom Operator-SDK angelegt und müssen nur noch angepasst werden. Im Folgenden ein Beispiel für die Generierung der Rollen-YAML-Datei 4.4 von Seite 53:

Code 4.1: Kubebuilder Generator-Anweisung

```
//+kubebuilder:rbac:
        groups="",namespace=abo,resources=pods,verbs=get
```

Dieser Befehl wird von Kubebuilder ausgewertet und die entsprechende Rollen-YAML-Datei generiert. Statt einer kompletten YAML-Datei muss also nur diese eine Zeile Code geschrieben werden.

4.1.1 YAML-Dateien

In diesem Abschnitt werden die YAML-Dateien des Operators beschrieben. In Kubernetes wird jede Ressource über eine YAML- oder JSON-Datei beschrieben. Diese Dateien werden dann an Kubernetes übergeben, damit dort die entsprechenden Ressourcen erstellt werden können.
Zunächst werden in den YAML-Dateien 4.2 und 4.3 die Spezifikationen für den WebHook und den dazugehörenden Service beschrieben. In den YAML-Dateien 4.4 und 4.5 kommen dann die Spezifikationen für die Rolle und die Zuweisung der Rolle an den Operator. In der letzten YAML-Datei 4.7 folgt noch ein Ausschnitt der Spezifikation für das Deployment des Operators.

Verändernder WebHook und Service

In der WebHook-YAML-Datei wird der verändernde WebHook konfiguriert. Unter **1** wird der API mitgeteilt, dass es sich bei dieser Spezifikation um einen verändernden WebHook handelt, mit dem unter **2** festgelegten Namen.
Die Namen von Ressourcen müssen innerhalb ihres Namensraumes eindeutig sein. Durch den Namenszusatz "resource-allocation-", der vor alle Ressourcen des Operators gesetzt wird, wird sichergestellt, dass sämtliche Namen eindeutig sind und sich gleichzeitig gut zuordnen lassen.
Unter **3** wird der zuständige Service festgelegt. Des Weiteren wird dort dem Service mitgeteilt, in welchem Namensraum sich der WebHook befindet und unter welchem Pfad er dort anzutreffen ist.
Die failurePolicy von **4** regelt, was passiert, wenn der WebHook nicht erreichbar sein sollte. Mögliche Optionen sind "Ignore" und "Fail". "Ignore" bedeutet, dass der WebHook ignoriert wird, "Fail" bedeutet, dass das Erstellen der anfragenden Ressource fehlschlägt. Dieser WebHook wurde mit "Ignore" konfiguriert.
Unter operations: bei **5** werden alle Operationen genannt, die der WebHook ausführen darf. Dieser WebHook darf Ressourcen erstellen. Unter **6** sind alle Ressourcen aufgelistet, die verändert werden dürfen. In diesem Fall sind das "Pods". Durch diese Spezifikation kann der WebHook die Pods beim Start verändern.
Unter **7** wird festgelegt, welche Pods vom WebHook überprüft werden sollen. Sind keine Einschränkungen angegeben, dann werden alle Pods überprüft. Zur Einschränkung können hier Namensräume angegeben werden, die überwacht werden solle.

Code 4.2: YAML-Datei des verändernden WebHooks

```
    apiVersion: admissionregistration.k8s.io/v1
1   kind: MutatingWebhookConfiguration
    metadata:
      creationTimestamp: null
2     name: resource-allocation-mutating-webhook-configuration
    webhooks:
    - admissionReviewVersions:
      - v1
      clientConfig:
```

```
3         service:
            name: resource-allocation-webhook-service
            namespace: abo
            path: /mutate-v1-pod
4       failurePolicy: Ignore
        name: mpodresourceallocation.kb.io
        rules:
        - apiGroups:
          - ""
          apiVersions:
          - v1
5         operations:
          - CREATE
6         resources:
          - pods
7         scope: Namespaced
        sideEffects: None
        namespaceSelector:
          matchLabels:
            resource-allocation-operator: "true"
```

Der Service, der in der YAML-Datei 4.2 unter Punkt **3** genannt wird, wird in der folgenden YAML-Datei 4.3 spezifiziert. Grundsätzlich werden Services für die Netzwerkkommunikation innerhalb von Kubernetes genutzt. Dieser Service ist für die Netzwerkkommunikation mit dem Deployment zuständig, zu dem sowohl der WebHook, als auch der Operator gehört.

Bei **1** wird der API mitgeteilt, dass es sich um einen Service handelt. Bei **2** wird der Name des Services angegeben, der zur korrekten Zuordnung dem Namen in der WebHook-YAML-Datei 4.2 unter **3** entsprechen muss. Ein Service muss, im Gegensatz zum WebHook, in einem Namensraum bereitgestellt werden, der unter **3** angegeben werden muss. Bei diesem Service wurde der Namensraum "system" zugeordnet. An Position **4** wird dem Service mitgeteilt, für welche Pods er zuständig ist und auf welchen Port diese Pods lauschen. In diesem Fall wurde angegeben, dass er für Pods mit dem Label "control-plane: controller-manager" zuständig ist. Das sind alle Pods des Deployments.

Nachdem der Service erstellt worden ist, wird diesem vom Kubernetes eine IP-Adresse zugewiesen. Durch die Abstraktion mit den Services ist es so auch möglich einer Gruppe von Pods eine IP-Adresse zuzuteilen.

Code 4.3: YAML-Datei des Service

```
    apiVersion: v1
1   kind: Service
    metadata:
2     name: resource-allocation-webhook-service
3     namespace: system
    spec:
      ports:
4       - port: 443
          protocol: TCP
          targetPort: 9443
      selector:
4       control-plane: controller-manager
```

Rolle, Rollenzuweisung und Service-Account

Die folgende YAML-Datei 4.4 erstellt eine neue Rolle. In Kubernetes sind die Berechtigungen über Rollen geregelt, weshalb diese, wie bei **1** zu sehen, in der Autorisierung der API angesiedelt sind.
Die apiGroups: von **2** beschreiben, auf welche API-Gruppen die Rolle Zugriff hat. In diesem Fall darf die Rolle auf die Kern-API-Gruppe "/api/v1" zugreifen, zu der auch Pods gehören.
Unter Punkt **3** wird geregelt, auf welche Ressourcen die Rolle Zugriff hat. Da der Operator Pods bearbeiten soll, sind an dieser Stelle "Pods" angegeben. Unter **5** wird dann definiert, was die Rolle mit den Ressourcen alles machen darf. In diesem Fall darf die Rolle die Ressourcen vom Kontext holen.

Code 4.4: YAML-Datei der Rolle

```
1   apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
      creationTimestamp: null
      name: resource-allocation-manager-role
      namespace: abo
    rules:
2   - apiGroups:
      - ""
3     resources:
      - pods
4     verbs:
      - get
```

Nach der Definition der Rolle muss noch die Zuordnung zu einem Rolleninhaber erfolgen. Die folgende YAML-Datei 4.5 ist ein "RoleBinding". Hier wird die Rolle einem Objekt oder Service-Account zugeordnet. Eine Rolle kann mehreren Objekten bzw. Service-Accounts zugewiesen sein und ein Objekt bzw. Service-Account kann Inhaber mehrerer Rollen sein.
Im der Rollen-Bindung wird unter **1** die Rolle genannt, die zugewiesen wird und unter **2** der Rolleninhaber. In diesem Fall ist das der Service-Account des Operators. Dieser Service-Account wird in der YAML-Datei 4.6 genauer beschrieben.

Code 4.5: YAML-Datei der Rollen-Bindung

```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: RoleBinding
    metadata:
      name: resource-allocation-manager-rolebinding
1   roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: Role
      name: resource-allocation-manager-role
2   subjects:
```

```
  - kind: ServiceAccount
    name: resource-allocation-controller-manager
    namespace: system
```

Der Service-Account ist ein Bindeglied zwischen den Rollen-Bindung und dem Deployment. Er ist so etwas wie die Identität des Operators innerhalb des Clusters. Aus diesem Grund wird dem Service-Account auch die Berechtigung zugewiesen. Die Zuweisung des Service-Accounts an den Operator findet im Deployment des Operators statt.

Code 4.6: YAML-Datei des Service-Accounts

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: resource-allocation-controller-manager
  namespace: system
```

Deployment

In der folgenden YAML-Datei 4.7 wird das Deployment des Operators beschrieben. Da die tatsächliche Spezifikation sehr lang ist, handelt es sich nur um einen Auszug aus der Datei. Bei Nummer **1** steht, dass das Deployment den Namen "resource-allocation-controller-manager" trägt und im Namensraum "abo" bereitgestellt wird. Der Operator darf nicht in einem Namensraum bereitgestellt werden, den der WebHook überwacht. Dieses hätte zur Folge, dass der WebHook auch den Operator überwachen möchte und somit auch sich selber, was zu Fehlern führt.
Das Schlüsselwort "replicas:" bei **2** beschreibt, dass es den Operator-Pod nur einmal gibt. An den Positionen, die mit **3** gekennzeichnet sind, wird ein Label mitgegeben, um dem Service zu zeigen, dass er für die Pods des Deployments zuständig ist. Die Details des Containers werden unter **4** "container:" beschrieben. Dort steht unter anderem bei Nummer **5**, welches Image genutzt werden soll und woher es geladen werden soll: Aus dem Cache oder aus dem Repository
Unter **6** ist beschrieben, welcher Port für die Kommunikation zwischen dem Service und dem Operator genutzt wird. Dieser Port muss identisch sein mit den Informationen im Service. Unter Punkt **7** stehen die Ressourcenanfragen für den Container. Eine Anfrage

von 100m-CPU bedeutet, das 0,1 oder 10% einer CPU angefragt werden. Ein Wert von 1000 steht für einen Kern bzw. 100%. Entsprechend sind auch Werte von über 1000 bzw. über 100% möglich, wenn mehr als ein Kern benötigt wird.
Eine Anfrage von 20Mi steht für 20 Mebibytes bzw. für 20971520 Bytes Arbeitsspeicher. Alternativ sind auch andere Angaben wie beispielsweise MB oder Byte zulässig. In der Umgebungsvariablen unter "env" bei Punkt **8** können die Namensräume konfiguriert werden, die der Operator überwachen soll. Die Variable "WATCH_NAMESPACE" wird vom Operator beim Start geladen und für die Konfiguration des Managers genutzt. Unter Punkt **9** wird der Service-Account an das Deployment gebunden.

Code 4.7: YAML-Datei des Deployments

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      labels:
        control-plane: controller-manager
1     name: resource-allocation-controller-manager
      namespace: abo
    spec:
2     replicas: 1
      selector:
        matchLabels:
3         control-plane: controller-manager
      template:
        metadata:
          labels:
3           control-plane: controller-manager
        spec:
4         containers:
5           image: controller:latest
            imagePullPolicy: Always
            name: manager
6           ports:
            - containerPort: 9443
```

```
                name: webhook-server
                protocol: TCP
7             resources:
                limits:
                  memory: 250Mi
                requests:
                  cpu: 100m
                  memory: 250Mi
8             env:
              - name: WATCH_NAMESPACE
                value: ""
9         serviceAccountName:resource-allocation-controller-manager
```

4.1.2 Reconcile-Funktion

In diesem Abschnitt wird die Reconcile-Funktion des Controllers genauer beschrieben. Wie in den Grundlagen dargelegt, wird die Reconcile-Funktion vom Control-Loop des Controller-Managers aufgerufen. Bei welchen Ressourcen der Controller benachrichtigt werden soll, wird, im Gegensatz zum WebHook, beim Operator im Code statt in der YAML-Datei konfiguriert.

```
ctrl.NewControllerManagedBy(mgr).For(&corev1.Pod{}).Complete(r)
```

Nachdem die Reconcile-Funktion aufgerufen wird, muss diese mithilfe des mitgegebenen Kontextes den verursachenden Pod in der Controller-Laufzeit finden. Um sicher zu gehen, dass nur bestimmte Pods überwacht werden, wird als nächster Schritt überprüft, ob der gefundene Pod mit einem entsprechenden Label versehen ist. Ist dieses nicht der Fall, dann wird die Funktion sofort wieder beendet und dem Controller-Manager mitgeteilt, dass der Operator für diesen Pod nicht erneut aufgerufen werden soll.
Anschließend findet eine Überprüfung statt, ob der Pod bereits fertig erstellt wurde. Dazu wird der Status überprüft. Wenn der Pod den Status "running" hat, wird eine der drei Optionen aus Kapitel 3.2 durchlaufen.
Ist der Pod noch nicht im Prozess der Terminierung, dann wird zum Abschluss der Reconcile-Funktion eine Nachricht an den Controller-Manager gesendet, dass der Pod

nach einer Minute erneut überprüft werden soll.

4.1.3 WebHook

In diesem Abschnitt werden die Details des WebHooks erläutert. Bevor dieser Container manipulieren kann, muss er erst beim WebHook-Server angemeldet werden. Dort wird er unter seiner konfigurierten Adresse registriert. In diesem Fall ist das "/mutate-v1-pod". Damit der WebHook später die Pods verändern kann, wird ihm ein Decoder zugewiesen, der Informationen zur Controller-Laufzeit enthält.
Bei Aufruf des WebHooks wird seine Handle-Funktion gestartet. Diese Funktion implementiert das Handler-Interface aus der Controller-Runtime-Bibliothek.
Genau wie der Controller muss auch der WebHook den Pod zuerst von der Controller-Laufzeit holen. Anschließend wird auch hier überprüft, ob der Pod mit dem entsprechenden Label versehen ist, um keine falschen Pods zu manipulieren.
Die Informationen, welche Ressourcen dem Container per Spezifikation zugewiesen wurden, können von den Containern abgefragt werden. Die optimierten Ressourcenanfragen werden aus der Datenbank geladen. Liegt für den Container noch kein Datenbank-Eintrag vor, dann werden die ursprünglichen Anfragen beibehalten und der Container bleibt unverändert.
Um dem Container neue Ressourcen zuweisen zu können, gibt es in der Kubernetes-API-Kernbibliothek sogenannte ResourceRequirements-Objekte. Das sind die Objekte, in denen Kubernetes die Ressourceninformationen aus den Spezifikationen speichert. Zum Verändern der Ressourcen im Container werden dort die bestehenden ResourceRequirements mit neuen überschrieben.
Damit die Änderung umgesetzt werden kann, muss eine Korrekturantwort mit den gewünschten Änderungen an den Master gesendet werden. Dafür werden veränderter und originaler Pod zurückgesendet. Da die komplette Kommunikation innerhalb von Kubernetes über REST-Schnittstellen läuft, müssen beide Pods in einen JSON-String umgewandelt werden, bevor sie zurückgesendet werden können.
Wie bereits in den Grundlagen beschrieben, wird der veränderte Pod nach dem Zurücksenden noch validiert und eventuell weiterbearbeitet. Möglicherweise wird er dabei noch weiter verändert.

4.1.4 Prometheus-Client

In diesem Abschnitt wird der Prometheus-Client genauer beschrieben. Zur Erstellung eines Clients muss bei diesem zuerst die Adresse zur Prometheus-API konfiguriert werden. Danach wird eine prometheus-kompatible API für den Client erstellt. Des Weiteren wird ein neuer Kontext erstellt, über den die gesamte Kommunikation zwischen Client und Prometheus läuft. Mithilfe dieser beiden Objekte können die PromQL-Anfragen und die Antworten zwischen Prometheus und dem Client ausgetauscht werden. Die hier benutzten PromQL-Anfragen werden in Abschnitt 4.2 genauer beschrieben.
Die Antwort von Prometheus erfolgt als String, in dem sämtliche angefragten Informationen enthalten sind. Die relevanten Informationen müssen anschließend herausgefiltert werden. Da es sich bei den Informationen, um numerische Werte handelt, muss nach dem Filtern noch eine Umwandlung in Integer-Werte erfolgen.

4.1.5 PostgreSQL-Client

In diesem Abschnitt wird der Client für die Datenbank PostgreSQL erläutert. Um einen Client für die Datenbank zu konfigurieren, müssen der Host und der zu benutzende Port, ein Benutzer mit einem Password sowie der Name der gewünschten Datenbank gesetzt werden. Nachdem die Verbindung zur Datenbank aufgebaut ist, überprüft der Client, ob die drei benötigten Tabellen vorhanden sind und erstellt sie gegebenenfalls neu.
Der CPU-Wert ist in Prometheus als Float-Wert hinterlegt. Für den Algorithmus wird aber ein Integer-Wert benötigt. Daher wird der Float-Wert vor der Sicherung in der Datenbank in einen Integer-Wert umgewandelt. Weitere Informationen zu der Umwandlung folgen in Abschnitt 4.3.
Bei jedem Informationsausstausch mit der Datenbank muss vorher anhand des Identifizierungsstrings, des Container-Namens und des Namensraumes die entsprechende ID herausgefunden werden. Daher benötigen die Insert-Funktionen immer die drei Informationen oder die ID als Übergabeparameter.
Um ein Volllaufen der Datenbank zu verhindern, werden nach jedem Auslesen aus den Tabellen die entsprechenden Daten gelöscht. Einzige Ausnahme ist die Tabelle "pod", deren Informationen erhalten bleiben müssen.

4.2 PromQL-Abfragen

Im Folgenden werden die in dieser Arbeit verwendeten PromQL-Abfragen kurz vorgestellt. Für diese Arbeit sind CPU- und Arbeitsspeicheranfrage sowie CPU- und Arbeitsspeichernutzung relevant. Aufgrund von z.B. Replika kann es sein, dass sich die Ressourcen von einem Pod über mehrere Knoten verteilen. Um pro Container einen Wert zu erhalten, wird in den Anfragen mit der Summe gearbeitet.
Prometheus ist so konfiguriert, dass es seine Werte etwa alle 30 Sekunden aktualisiert. Daher wurde im Controller ein Zeitraum von einer Minute für das Aufrufen der Reconcile-Funktion gewählt. So wird sichergestellt, dass ein Wert nicht doppelt von Prometheus abgefragt wird.

Tatsächliche CPU-Nutzung

Die durchschnittliche Nutzung von CPU-Kernen in der letzten Minute pro Container:

```
sum(
    rate(
        container_cpu_usage_seconds_total{
            namespace="NAMENSRAUM_NAME",
            container!="",
            container!="POD",
            pod="POD_NAME"
        }[1m]
    )
) by (container)
```

Tatsächliche Arbeitsspeichernutzung

Die aktuelle Arbeitsspeichernutzung eines Containers:

```
sum(
    container_memory_working_set_bytes{
```

```
        namespace="NAMENSRAUM_NAME",
        container!="",
        container!="POD",
        pod="POD_NAME"
    }
) by (container)
```

Tatsächliche maximale Arbeitsspeichernutzung

Die bis dahin höchste Arbeitsspeichernutzung:

```
sum(
    container_memory_max_usage_bytes{
        namespace="NAMENSRAUM_NAME",
        container!="",
        container!="POD",
        pod="POD_NAME"
    }
) by (container)
```

Zu wenig zugewiesene CPU

Die Anzahl der Sekunden, die ein Container insgesamt seine CPU-Grenze überschritten hat und gedrosselt werden musste:

```
sum(
    rate(
        container_cpu_cfs_throttled_seconds_total{
            namespace="NAMENSRAUM_NAME",
            container!="",
            container!="POD",
            pod="POD_NAME"
        }[1m]
    )
) by (container)
```

Zu wenig zugewiesener Arbeitsspeicher

Die Anzahl der Sekunden, die ein Container insgesamt das Speichernutzungslimit erreicht hat:

```
sum(
    container_memory_failcnt{
        namespace="NAMENSRAUM_NAME",
        container!="",
        container!="POD",
        pod="POD_NAME"}
) by (container)
```

Maximal zulässige Ressourcenzuteilung

Das Arbeitsspeicher- bzw. CPU-Ressourcenlimit eines Containers über alle Knoten hinweg:

```
sum(
    kube_pod_container_resource_limits{
        resource="RESOURCE",
        namespace="NAMENSRAUM_NAME",
        container!="",
        container!="POD",
        pod="POD_NAME"
    }
) by (container)
```

CPU- bzw. Arbeitsspeicher-Ressourcenanfrage

Existiert für eine Ressource ein Ressourcenlimit aber keine Ressourcenanfrage, dann verwendet Kubernetes das angegebene Limit als Anfragewert. Gibt es eine Ressourcenanfrage aber kein Ressourcenlimit, dann lässt Kubernetes die Anfrage unverändert. Dem Container werden in diesem Fall alle verfügbaren Ressourcen zur Verfügung gestellt.
CPU- bzw. Arbeitsspeicher-Ressourcenanfrage eines Containers über alle Knoten hinweg:

```
sum(
    kube_pod_container_resource_requests{
        resource="RESOURCE",
        namespace="NAMENSRAUM_NAME",
        container!="",
        container!="POD",
        pod="POD_NAME"
    }
) by (container)
```

4.3 Herausforderungen

In diesem Abschnitt werden die größeren Herausforderungen beschrieben, die während der Realisierung des Operators aufgetreten sind.

Schneller Versionsfortschritt bei Kubernetes

In den letzten Jahren hat Kubernetes eine enorme Weiterentwicklung erfahren. Das hat zur Folge, dass sich zwischen den Versionen häufig die Schnittstellen zwischen Kubernetes und dem Operator geändert haben.
Bei einem auftretenden Fehler konnte nicht einfach die Lösung eines Forums genutzt werden, da die dortige Antwort häufig veraltet oder zu neu war. Daher gibt es auf eine Fragestellung häufig viele Antworten, welche in den entsprechenden Versionen richtig sind, abseits ihrer Kubernetes-Version aber nicht funktionieren. Der Kubernetes-Cluster, in dem in dieser Arbeit gearbeitet wird, hat nicht die neueste Version. Entsprechend konnte auch nicht einfach nach den neuesten Foreneinträgen gesucht werden. Da mehrere Kubernetes-Versionen parallel existieren und in Nutzung sind, war auch das Datum in den Foreneinträgen nur ein grober Filter.

Unübliche Nutzung des Operators

Wie in den Grundlagen beschrieben, wurde die Möglichkeit eines personalisierten Controllers ursprünglich eingerichtet, um komplexere Ressourcen, wie beispielsweise Datenbank-Anwendungen, zu überwachen. In dieser Arbeit wurde der Operator verwendet, um fremde Pods zu überwachen und die gesammelten Informationen in einer Datenbank zu speichern.
Folglich sind viele Beispiele zu finden, in denen erklärt wird, wie der Operator seine eigenen Ressourcen erstellt und überwacht. Der in dieser Arbeit verwendete Einsatz des Operators zur Überwachung fremder Ressourcen ist dagegen nur selten zu finden.

Unübliche Nutzung des Prometheus-Clients

Die übliche Verwendung des Prometheus-Clients scheint sich auf die Veröffentlichung von Metriken an Prometheus zu beschränken. Entsprechend schwierig war es, herauszufinden, wie der Client genutzt werden muss, um Informationen von Prometheus abzufragen. Nachfolgend ist ein Codeausschnitt aus dem Prometheus-Client des Operators zu sehen.

In der Konstante "promAddress" muss die richtige Adresse eingetragen werden, unter der Prometheus in Kubernetes erreichbar ist. Zum Ausführen des Clients wird schließlich noch ein Abfrage-String in PromQL benötigt.
Zuerst wird ein Client mit der Adresseninformation erstellt. Mithilfe dieses Clients kann eine neue API für den Operator erstellt werden, über die die Kommunikation mit Prometheus läuft. Anschließend kann eine Anfrage mit einem neu erstellten Kontext an Prometheus gesendet werden.

```
import (
    "context"
    "github.com/prometheus/client_golang/api"
    "github.com/prometheus/client_golang/api/prometheus/v1"
    "time"
)

const promAdress = "Address"

promClient, err := api.NewClient(
                       api.Config{Address: promAddress})
promContext,cancel := context.WithTimeout(
                                    context.Background(),
                                    10*time.Second)
promApi := v1.NewAPI(promClient)
result,warnings,err := promApi.Query(
                                     promContext,
                                     "PromQL-String",
                                     time.Now())
```

Prometheus-Angaben für die CPU

Die Antwort von Prometheus erfolgt als String, aus dem die Werte herausgefiltert werden müssen. Der Arbeitsspeicherwert liegt in Byte vor, was einfach weiter zu verarbeiten war. Der CPU-Wert wird als Float- oder Integer-Wert zurückgegeben. So würde die Ausgabe für einen CPU-Kern `1` lauten und für eineinhalb CPU-Kerne `1.5`.
Eine grundsätzliche String-Umwandlung in Float-Werte ist ungünstig, da der Algorithmus mit Integer-Werten arbeitet. Hier wird als Lösung mit Milli-CPU gearbeitet. Zuerst findet eine String-Umwandlung in Float-Werte statt und anschließend wird das Ergebnis mit 1000 multipliziert. Das Ergebnis ist ein Integer-Wert, bei dem aber beachtet werden muss, dass immer 1000 Einheiten für einen Kern stehen.

Erster und letzter Aufruf der Reconcile-Funktion

Die Reconcile-Funktion des Operators wird das erste Mal bereits vor Abschluss der Create-Phase aufgerufen. Zu diesem Zeitpunkt liegen noch nicht alle benötigten Informationen für den Controller vor. Aus diesem Grund wird innerhalb des Controllers zuerst überprüft, ob der Pod bereits fertig erstellt wurde und sich in der Phase "running" befindet.

Der letzte Aufruf erfolgt, sobald der "DeletionTimestamp" gesetzt ist. Die Herausforderung ist, dass ca. 30 Sekunden später ein erneuter Aufruf erfolgt. Da dieser zu einer erneuten Auswertung führen würde und damit ein großes Risiko für Fehler darstellt, muss dieser letzte Aufruf rausgefiltert werden.
Eine Lösung über Label ist aufgrund der Rollen und Rechtevergabe nicht praktikabel. Daher wurde der Weg über einen Datenbankeintrag gewählt. Wenn jetzt im Zuge der Auswertung nach der ID des Containers gesucht wird, dann wird automatisch kontrolliert, ob der Container bereits ausgewertet wird oder noch ausgewertet werden muss.

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Ausarbeitung vorgestellt. Dazu wird zuerst in den Abschnitten 5.1 und 5.2 gezeigt, inwiefern sich die Ressourcennutzung verbessern lässt. Im dritten Abschnitt 5.3 werden dann die für die Nutzung des Operators nötigen Voraussetzungen erläutert. Im letzten Abschnitt 5.4 wird anschließend beschrieben, was zusätzlich beachtet und eingestellt werden muss, wenn Pods durch den Operator überwacht werden sollen.
Für die Vergleiche werden exemplarisch drei unterschiedliche Pods gewählt, die häufig gelaufen sind. Aus diesen Pods werden jeweils einige der acht bis elf Container genommen. Nicht alle Container laufen in allen Pods.
Die Pods werden durchnummeriert, da die Namen hier irrelevant sind. Da sämtliche Pods Bestandteil von ähnlichen Testpipelines sind, heißen die Container gleich. Allerdings unterscheiden sich die Container hinsichtlich der erzeugten Last.
Die zugrundeliegenden Daten und die Daten weiterer Pods befinden sich in dem Ordner "Daten" auf der beiliegenden CD.

5.1 Anfrageentwicklung über die Zeit

In diesem Abschnitt werden die Entwicklungen der Ressourcenanfragen über die Zeit betrachtet, um den Lernfortschritt des Algorithmus darstellen zu können.
Im Folgenden, in Tabelle 5.1, ist die Entwicklung des Recall-Archive-Simulation-Containers von Pod 3 zu sehen. Es ist gut zu erkennen, wie sich die Ressourcenanfragen über die Zeit verändern. Der maximale Arbeitsspeicherverbrauch dieses Containers liegt durchschnittlich bei 515MB. Die neuen Ressourcenanfragen nähern sich langsam dem maximalen Verbrauch plus einem kleinen Puffer an.
Im Gegensatz dazu ist in Tabelle 5.2 der Cucumber-Test eines anderen Pods dargestellt.

neue Anfrage in MB	Uhrzeit
976	0:26:41
896	0:27:10
768	0:39:58
704	0:58:51
640	0:59:20
704	1:09:27
640	1:19:24
576	1:27:52
640	1:28:19
576	1:52:44
640	1:53:01
576	1:53:17

Tabelle 5.1: Arbeitsspeicheranfrage über die Zeit des Recall-Archive-Simulation-Containers von Pod 3

Bei diesem Pod schwanken die Verbräuche stark, weshalb der Algorithmus Schwierigkeiten bei der Anpassung der Anfragen hat.
Von Zeile Zwei auf Zeile Drei verändern sich die Verbräuche stark und liegen danach deutlich oberhalb der vorherigen Anfragen. In so einem Fall greift ein Sicherheitsmechanismus im Operator, welcher sicherstellt, dass die Anfragen beim Arbeitsspeicher immer oberhalb der maximalen Verbräuche bleiben. Aus diesem Grund sind die neuen Anfragen in der dritten Zeile deutlich oberhalb der vorhergehenden Anfragen.
Danach versucht der Algorithmus sich wieder anzupassen und sich auf die neue Situation einzustellen.

Maximaler Verbrauch	neue Anfrage in MB	Uhrzeit
2704	3008	1:12:28
2563	2944	1:12:40
3427	4214	1:20:14
3757	4288	1:33:02
3877	4160	1:41:48

Tabelle 5.2: Arbeitsspeicheranfrage über die Zeit eines Cucumber-Tests eines weiteren Pods

5.2 Vergleich der Ressourcenverbräuche

In diesem Abschnitt wird untersucht, ob die Ressourcennutzung verbessert werden konnte. Dazu werden in den folgenden Tabellen die Ressourcenanfragen jeweils vor und nach der Optimierung des Operators gegenübergestellt.

5.2.1 Arbeitsspeicher-Vergleich

In diesem Abschnitt werden die Arbeitsspeicheranfragen und -verbräuche verglichen. Diese sind in den Tabellen 5.3 bis 5.5 dargestellt. Auffällig ist, dass sowohl die ursprünglichen Anfragen als auch die ursprünglichen Limits bei den jeweiligen Containern der drei Pods identisch sind.
Nach der Anpassung durch den Operator ist der Anfrage-Wert identisch mit dem jeweiligen Limit-Wert. Dieses Vorgehen sorgt für mehr Stabilität beim Ausführen, da der Scheduler eine genauere Vorgabe zum tatsächlichen Verbrauch hat und der Pod bis zum Limit die Ressourcen garantiert bekommt.
Vor der Anpassung durch den Operator lagen die Anfragen meist unterhalb der tatsächlichen maximalen Verbräuche. Der Container musste sich also zur Laufzeit zusätzlichen Speicher besorgen. Es ist nicht garantiert, dass der Container diesen zusätzlichen Speicher tatsächlich erhält. Da die optimierte Anfrage keine spätere Erhöhung der Ressourcen zulässt, sind die Anfragen nach der Optimierung im Regelfall höher als vor der Anpassung. Lediglich beim Container Cucumber-Test ist eine Verringerung der Ressourcen zu sehen, da die ursprüngliche Anfrage oberhalb des tatsächlichen Verbrauches lag.
Bei den Containern ist gut erkennbar, dass der Algorithmus in der Lage ist, die Anfragen an die tatsächlichen Verbräuche anzupassen, unabhängig davon, ob die ursprünglichen Verbräuche zu hoch oder zu niedrig angesetzt waren. Darüber hinaus müssen die angepassten Container ihre Ressourcen nicht nachträglich an ihren Verbrauch anpassen.

Containername	neue Anfrage in MB	altes Limit in MB	alte Anfrage in MB	maximale Verbräuche in MB
Database	11392	11534	9437	10665
Databasefiller	2688	4194	2621	2292
Cucumber-Test	4160	7340	5242	3403
Instant-Server	2048	2621	1887	1688
Interface-Simulation	1216	2621	838	1161
Webservices	2752	4718	2202	2632

Tabelle 5.3: Arbeitsspeicheranfrage und -verbrauch von Pod 1

Containername	neue Anfrage in MB	altes Limit in MB	alte Anfrage in MB	maximale Verbräuche in MB
Database	10368	11534	9437	10159
Databasefiller	2752	4194	2621	2284
Cucumber-Test	2432	7340	5242	1961
Instant-Server	1920	2621	1887	1726
Interface-Simulation	1088	2621	838	980
Recall-Archive-Simulation	832	2621	536	521
Webservices	3072	4718	2202	2711

Tabelle 5.4: Arbeitsspeicheranfrage und -verbrauch von Pod 2

Containername	neue Anfrage in MB	altes Limit in MB	alte Anfrage in MB	maximale Verbräuche in MB
Database	12032	11534	9437	11003
Databasefiller	3072	4194	2621	2369
Cucumber-Test	4992	7340	5242	4474
Instant-Server	1984	2621	1887	1771
Interface-Simulation	2752	2621	838	2208
Recall-Archive-Simulation	576	2621	536	522
Webservices	3072	4718	2202	2865

Tabelle 5.5: Arbeitsspeicheranfrage und -verbrauch von Pod 3

5.2.2 CPU-Vergleich

In diesem Abschnitt werden Anfrage und tatsächlicher Verbrauch der CPU betrachtet. Diese sind in den Tabellen 5.6 bis 5.8 dargestellt. Auch bei dieser Ressource zeigt sich, dass die ursprüngliche Anfrage der jeweiligen Container bei allen Pods identisch ist.
Im Gegensatz zum Arbeitsspeicher wurde hier kein Limit gesetzt. Dieses Vorgehen nutzt der Operator bei der Zuweisung der neuen CPU-Anfragen ebenfalls. Im Gegensatz zum Arbeitsspeicher, wo die Meisten der alten Anfragen unterhalb der tatsächlichen Nutzung liegen, sind die CPU-Anfragen meistens zu hoch angesetzt.
Da die nachträgliche Veränderung der CPU-Nutzung sehr dynamisch ist, macht es Sinn, weniger als den tatsächlichen Verbrauch zuzuweisen. Der Operator weist einen Wert zwischen der maximalen und der durchschnittlichen Nutzung zu. Das hat den Vorteil, dass weniger CPU auf dem Knoten für den Container reserviert wird. Im schlimmsten Fall wird die Anwendung dadurch zeitweise etwas langsamer. Im Normalfall werden die nicht reservierten CPUs dynamisch von den bereitgestellten Pods genutzt.
Anhand der neuen Anfragen kann gezeigt werden, dass der Operator auch bei der CPU die Ressourcenanfrage an den tatsächlichen Verbrauch anpassen kann. In diesem Fall ist die Anfrage auch reduziert worden.

Containername	neue Anfrage	altes Limit	alte Anfrage	maximale Nutzung	durchschn. Nutzung
Database	0,512	0	1	0,963	0,488
Databasefiller	0,064	0	1	0,069	0,005
Cucumber-Test	0,960	0	4	3804	0,930
Instant-Server	0,768	0	1	0,281	0,195
Interface-Simulation	0,064	0	1	0,049	0,033
Webservices	0,064	0	1	0,064	0,015

Tabelle 5.6: Genutzte CPU-Kerne von Pod 1

Containername	neue Anfrage	altes Limit	alte Anfrage	maximale Nutzung	durchschn. Nutzung
Database	0,791	0	1	1,561	0,533
Databasefiller	0,682	0	1	1,874	0,284
Cucumber-Test	0,896	0	4	1,691	0,097
Instant-Server	0,576	0	1	1,869	0,385
Interface-Simulation	0,614	0	1	1,666	0,263
Recall-Archive-Simulation	0,064	0	1	0,007	0,005
Webservices	1,029	0	1	2,444	0,556

Tabelle 5.7: Genutzte CPU-Kerne von Pod 2

Containername	neue Anfrage	altes Limit	alte Anfrage	maximale Nutzung	durchschn. Nutzung
Database	1,024	0	1	0,512	0,510
Databasefiller	0,064	0	1	0,068	0,004
Cucumber-Test	2,240	0	4	1,459	1,395
Instant-Server	0,256	0	1	0,140	0,159
Interface-Simulation	0,064	0	1	0,039	0,034
Recall-Archive-Simulation	0,064	0	1	0,069	0,005
Webservices	0,064	0	1	0,070	0,006

Tabelle 5.8: Genutzte CPU-Kerne von Pod 3

5.3 Voraussetzungen für die Nutzung

In diesem Abschnitt wird beschrieben unter welchen Voraussetzungen der Operator eine Verbesserung herbeiführen kann und "Out-of-Memory-Fehler" vermieden werden.

Wiederholungsrate

Der Operator, der aus den tatsächlichen Verbräuchen der Vergangenheit lernt, zukünftige Ressourcenanfrage besser anzupassen, ist darauf angewiesen, dass der Pod möglichst oft ausgeführt wird.

Dabei ist der Lerneffekt umso größer, je weniger Abweichung die tatsächlichen Verbrauchsdaten der einzelnen Durchläufe voneinander haben. Bei stark schwankenden Verbräuchen ist eine sinnvolle Vorhersage der benötigten Ressourcen nicht möglich.

Die Anwendung innerhalb des Images

Der Operator weiß nicht, welche Anwendung innerhalb eines Containers läuft. Daher es wichtig ist, dass dort keine zusätzlichen Ressourcenanfragen gestellt werden, die der Operator nicht sehen und auch anhand der Verbräuche nicht erkennen kann. Ein Beispiel für solche Ressourcenanfragen wäre die "Java heap size". Mit den Parametern Xmx und Xms können Ressourcenanforderungen festgelegt werden, die für den Operator nicht sichtbar sind und die sich auch nicht zwingend in den Ressourcenverbräuchen widerspiegeln.

Im Fall von Java existieren alternative Parameter, die extra für die Verwendung in Containern eingeführt wurden. Mit dem Parameter "-XX:+UseContainerSupport" werden diese Parameter aktiviert. Mit den folgenden zwei Parametern kann angegeben werden, wieviel Prozent der Containerressourcen für den "Java heap size" zur Verfügung stehen:

"-XX:InitialRAMPercentage"

"-XX:MaxRAMPercentage"

Dadurch kann sich die Java-Anwendung an verändernde Containerressourcen anpassen. Allerdings besitzen Java-Anwendungen einen Overhead, der je nach Anwendung unterschiedlich groß ist und der zusätzlich zur "Java heap size" noch zusätzliche Ressourcen verbraucht.

Wenn dem Pod mehr Ressourcen zugewiesen werden, dann wird die Anwendung durch den damit größeren "Java heap size" zwar schneller, aber der Overhead bleibt gleich, wodurch ungenutzte Ressourcen entstehen. Auf der anderen Seite braucht eine Java-Anwendung eine Mindestmenge an Ressourcen, die sich nur abschätzen

lässt, wenn bekannt ist, was die Anwendung genau macht und die bei Unterschreitung zu einem OOM-Fehler führt.

Aufgrund dieser Einschränkungen ist die Ressourcenanpassung von Containern, in denen eine Java-Anwendung läuft, nicht ohne weiteres möglich. Der hier vorgestellte Operator hat bei Containern mit Java-Anwendungen keine Möglichkeit, Informationen über die Mindestmenge und den Overhead zu bekommen und ist daher nicht in der Lage, Ressourceanfragen für diese Pods anzupassen, ohne einen OOM-Fehler zu riskieren.

5.4 Nutzungsänderung der Pods

In diesem Abschnitt wird erläutert, welche Änderungen es bei der Nutzung der Pods gibt, wenn diese vom Operator überwacht werden sollen. Es müssen zwei Bereiche betrachtet werden. Zum einen die Konfigurationen, die für den WebHook vorgenommen werden müssen und zum anderen die Konfiguration für den Operator.

WebHook

Der WebHook ist grundsätzlich im kompletten Cluster gültig. Um dieses einzuschränken, wurde der WebHook so gestaltet, dass zwei Label gesetzt werden müssen. Sowohl der Namensraum als auch der Pod müssen mit dem Label "resource-allocation-operator: "true" " gekennzeichnet werden.
Nur gelabelte Namensräume werden vom WebHook geprüft und nur gelabelte Pods werden dann auch bearbeitet. So ist es möglich, Pods in gelabelten Namensräumen laufen zu lassen, die nicht verändert werden sollen.

Operator

Im Gegensatz zum WebHook muss für den Operator kein Label an die Namensräume gesetzt werden. In der YAML-Datei des Operator-Managers werden alle Namensräume in der "env"-Variable gespeichert, die der Operator überwachen soll.
Zusätzlich dazu ist auch hier das Label "resource-allocation-operator: "true" " am Pod notwendig, damit innerhalb der Namensräume nicht automatisch alle Pods überwacht werden.

Zusammengefasst müssen das Label an die entsprechenden Namensräume gesetzt, die Namensräume in die "env"-Variable gespeichert und die entsprechenden Pods mit einem Label versehen werden.
Zusätzlich zu den Labeln für die Überwachung und Manipulation ist noch eine eindeutige Identifizierung der einzelnen Pods notwendig, damit diese in der Datenbank wiedergefunden werden können. Dieser eindeutige Name wird ebenfalls mittels Label übergeben. Darüber hinaus sollte darauf geachtet werden, dass nur Pods für die Anpassung gewählt werden, die bei jedem Aufruf einen ähnlichen Ressourcenverbrauch haben und häufig gestartet werden.

6 Fazit und Ausblick

In diesem Kapitel wird im ersten Abschnitt 6.1 ein Fazit gezogen und anschließend in Abschnitt 6.2 ein Ausblick auf mögliche weiterführende Untersuchungen gegeben.

6.1 Fazit

Es kann kein durchweg positives Fazit aus dieser Arbeit gezogen werden. Unter ganz bestimmten Voraussetzungen ist es möglich, mittels dieses Operators die Ressourcennutzung zu verbessern. Allerdings sind die Voraussetzungen erheblich und fordern aufwändige Vorbereitungen.
Im Vorfeld muss zunächst geprüft werden, welche Images innerhalb der Container laufen und ob eventuelle Ressourcenanfragen versteckt sind. Auch muss vorher geprüft werden, ob die entsprechenden Pods jedes Mal ungefähr die gleichen Ressourcen benötigen. Pods mit einem außergewöhnlich hohen Ressourcenverbrauch würden mit einem OOM-Fehler beendet werden.
Darüber hinaus müssen die Pods in der Datenbank eindeutig erkennbar sein. Handelt es sich, wie in dieser Arbeit, um Testpipelines, dann kann es sein, dass Pods automatisiert mit einem Namen versehen werden. In dem Fall kann nicht einfach der Name des Pods oder Containers genommen werden. Es muss ein zusätzlicher, eindeutiger Name gefunden werden.
Im aktuellen Zustand ist noch sehr viel manueller Aufwand nötig, um die zum Operator passenden Pods herauszufiltern. Grundsätzlich konnte aber gezeigt werden, dass es möglich ist, die Ressourcenanpassung zu verbessern.
Positiv ist anzumerken, dass die Pods stabiler laufen, weil der Operator das Arbeitsspeicherlimit mit der Anfrage gleichsetzt und damit verhindert, dass Pods nachträglich wieder Arbeitsspeicher abgeben müssen.

6.2 Ausblick

Der in dieser Arbeit vorgestellte Operator kann nur einen kleinen Teil der Pods sinnvoll anpassen. Bei den meisten Pods reicht die Information, die Prometheus über den Ressourcenverbrauch ermitteln kann, nicht aus, um eine Anpassung sinnvoll zu ermöglichen. Da der überwiegende Teil davon eine Java-Anwendung enthält, liegt es nahe, als nächsten Schritt ein verbessertes Monitoring für Java-Anwendungen zu untersuchen.
Beispielsweise könnte in einer weiterführenden Untersuchung geprüft werden, ob die Prometheus-Erweiterung "Java Management Extensions" (JMX) eine Lösung darstellen könnte. JMX soll ein Bindeglied zwischen Prometheus und dem nativen Metrik-Erfassungssystem von Java sein. Möglicherweise bekommt der Operator so Zugriff auf weiterführende Informationen der Java-Anwendungen. Damit wäre dann eventuell eine sinnvolle und sichere Ressourcenanpassung auch bei diesen Pods möglich.

Literaturverzeichnis

[1] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. *IEEE 10th International Conference on Cloud Computing*, 2017.

[2] Api groups in kubernetes. https://www.waytoeasylearn.com/learn/api-groups-in-kubernetes/. Accessed: 02-02-2023.

[3] David Balla, Csaba Simon, and Markosz Maliosz. Adaptive scaling of kubernetes pods. *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[4] Fangzhe Chang, Jennifer Ren, and Ramesh Viswanathan. Optimal resource allocation in clouds. *IEEE 3rd International Conference on Cloud Computing*, 2010.

[5] containerd container runtime für kubernetes. https://containerd.io/. Accessed: 19-12-2022.

[6] Cri-o container runtime für kubernetes. https://cri-o.io/. Accessed: 19-12-2022.

[7] Vera Demberg. Iterative dichotomiser 3. https://www.coli.uni-saarland.de/courses/mathe3/SS12/Vorlesungen/decisiontree.pdf. Accessed: 17-02-2023.

[8] Deployment dokumentation. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Accessed: 14-12-2022.

[9] Inga Döbel, Dr. Miriam Leis, Manuel Molina Vogelsang, Dmitry Neustroev, Dr. Henning Petzka, Annamaria Riemer, Dr. Stefan Rüping, Dr. Angelika Voss, Martin Wegele, and Dr. Juliane Welz. *Maschinelles Lernen Eine Analyse zu Kompetenzen,Forschung und Anwendung*. Fraunhofer-Gesellschaft, 2018.

[10] Yuqi Fu, Shaolun Zhang, Jose Terrero, Ying Mao, Guangya Liu, and Sheng Li Dingwen Tao. Progress-based container scheduling for short-lived applications in a kubernetes cluster. *IEEE International Conference on Big Data*, 2019.

[11] Hauptkomponentenanalyse. https://databasecamp.de/statistik/principal-component-analysis. Accessed: 21-02-2023.

[12] Mahmoud Imdoukh, Imtiaz Ahmad, and Mohammad Gh. Alfailakawi. Machine learning-based auto-scaling for containerized applications. 2019.

[13] Eunsook Kim, Kyungwoon Lee, and Chuck Yoo. On the resource management of kubernetes. *International Conference on Information Networking*, 2021.

[14] Kubebuilder dokumentation. https://github.com/kubernetes-sigs/kubebuilder. Accessed: 16-12-2022.

[15] Kubernetes controller. https://kubernetes.io/docs/concepts/architecture/controller/. Accessed: 24-11-2022.

[16] Kubernetes-controller-runtime dokumentation. https://github.com/kubernetes-sigs/controller-runtime. Accessed: 16-12-2022.

[17] Kubernetes dokumentation. https://kubernetes.io/de/. Accessed: 02-11-2022.

[18] Kubernetes webhook. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/. Accessed: 24-11-2022.

[19] Lineare regression. https://www.math.uni-bielefeld.de/~sek/funktion/leit03.pdf. Accessed: 17-02-2023.

[20] Logistische regression. https://datatab.de/tutorial/logistische-regression. Accessed: 17-02-2023.

[21] Operator framework dokumentation. https://operatorframework.io/. Accessed: 16-12-2022.

[22] Operator lifecycle manager dokumentation. https://olm.operatorframework.io/. Accessed: 16-12-2022.

[23] Operator sdk. https://sdk.operatorframework.io/docs/building-operators/golang/quickstart/. Accessed: 24-11-2022.

[24] Operatorhub.io. https://operatorhub.io/. Accessed: 16-12-2022.

[25] Gerhard Paaß and Dirk Hecker. *Künstliche Inteligenz: Was steckt hinter der Technologie der Zukunft?* Springer Viewweg, 2021.

[26] Chris Piech. K-means. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html. Accessed: 20-02-2023.

[27] Postgresql. https://www.postgresql.org/about/. Accessed: 16-11-2022.

[28] Prometheus client repository. https://github.com/prometheus/client_golang. Accessed: 02-11-2022.

[29] Prometheus dokumentation. https://prometheus.io/docs/introduction/overview/. Accessed: 02-11-2022.

[30] Prometheus git repository. https://github.com/prometheus/prometheus. Accessed: 02-11-2022.

[31] Pv und pvc dokumentation. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Accessed: 14-12-2022.

[32] Jens Pönisch. Support vector machines. https://www.tu-chemnitz.de/urz/ittime/documents/Vortrag_Jens_Poenisch_SVM.pdf. Accessed: 20-02-2023.

[33] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. *IEEE 12th International Conference on Cloud Computing*, 2019.

[34] Limit und request im vergleich. https://www.kreyman.de/index.php/others/vmware/248-optimale-nutzung-von-cpu-und-memory-ressourcen-in-vsphere-with-tanzu. Accessed: 12-12-2022.

[35] Rolling update dokumentation. https://kubernetes.io/de/docs/tutorials/kubernetes-basics/update/update-intro/. Accessed: 14-12-2022.

[36] Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. Skynet: Performance-driven resource management for dynamic workloads. *IEEE 14th International Conference on Cloud Computing*, 2021.

[37] Jef Spaleta. Kubernetes architektur. https://www.cncf.io/blog/2019/08/19/how-kubernetes-works/, 2019. Accessed: 24-11-2022.

[38] Richard S. Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 2018.

[39] Minh-Ngoc Tran, Dinh-Dai Vu, and Younghan Kim. A survey of autoscaling in kubernetes. *Thirteenth International Conference on Ubiquitous and Future Networks*, 2022.

[40] Bobby Woolf. Reconcile loop. https://developers.redhat.com/articles/2021/06/22/kubernetes-operators-101-part-2-how-operators-work#reconcile_states, 2021. Accessed: 24-11-2022.

[41] Natan Yellin. Cpu limit. https://home.robusta.dev/blog/stop-using-cpu-limits. Accessed: 24-02-2023.

[42] Natan Yellin. Ram limit. https://home.robusta.dev/blog/kubernetes-memory-limit. Accessed: 24-02-2023.

[43] Tugba Yilmaz. Kern-hauptkomponentenanalyse. http://www.mi.uni-koeln.de/wp-znikolic/wp-content/uploads/2021/02/20201218_Machine_Learning_Basics_Handout_Yilmaz.pdf. Accessed: 21-02-2023.

A Anhang

A.1 CD

Ordnername	Inhalt des Ordners
Operator Code	Der Code für das Image des Operators
Job Code	Der Code für das Image des Jobs
Daten	Die Daten für die Auswertung als CSV Datei
Internetseiten	Die Internetseiten des Literaturverzeichnises
Masterarbeit	Die Masterarbeit als pdf

Erklärung zur selbstständigen Bearbeitung

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

Ort	Datum	Unterschrift im Original