Fakultät Informatik/Mathematik

Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Master-Arbeit

Analyse von Verfahren zur Optimierung des Container-Schedulings in Kubernetes

Erik Schubert
Studiengang: Angewandte Informatik

Gutachter

Prof. Dr.-Ing. Robert Baumgartl
Dipl.-Inf. Fabian Ruff

Eingereicht am: 31. August 2023

Abstrakt

Kubernetes ist ein etablierter Container-Orchestrator. Dennoch stellt die Zuordnung von Containern zu Knoten Anwender immer wieder vor Herausforderungen und erschwert eine effiziente und erfolgreiche Adaption. In dieser Arbeit werden die Cluster der SAP Converged Cloud mittels Design Science untersucht, um verschiedene Verbesserungsmöglichkeiten im Container-Scheduling aufzuzeigen. Es wird mittels einer KANO-Umfrage bestimmt, welche Ressourcen und Nebenbedingungen im Scheduling relevant sind. Dann werden verschiedene Vector-Bin-Packing-Heuristiken und Methoden zur Vorhersage des Ressourcenbedarfs mittels Simulationen verglichen. Weiterhin wird die Interaktion zwischen verschiedenen Containern auf einem Knoten analysiert. Die gewonnenen Erkenntnisse bezüglich des Vector-Bin-Packings werden in einem Cluster experimentell validiert. Es wird aufgezeigt, dass die Standardkonfiguration des Schedulers und des Vertical-Pod-Autoscalers bezüglich der Auslastungsmaximierung suboptimal ist. Weiterhin lassen sich in der Literatur sowohl für das Bin-Packing als auch für die Ressourcenvorhersage Heuristiken und Modelle finden, welche weitere Verbesserungen versprechen. Trotz dessen bringt die Anpassung der Konfiguration des Schedulers, welche wenig Investition erfordert, eine Steigerung der Auslastung. Das Container-Scheduling ist ganzheitlich zu betrachten, da Aspekte der Ressourcenspezifikation, der Zuordnung von Containern zu Knoten und der lokalen Isolation der Ressourcen miteinander interagieren.

Inhaltsverzeichnis

Akronyme

API Application-Programming-Interface. 3, 4, 6, 16, 17, 18, 52

ARIMA-Modell Autoregressive-Integrated-Moving-Average-Modell. 37, 44, 45, 46, 47, 48, 49, 58, 59

CFS Completly-Fair-Scheduler. 10, 11, 19, 21, 24, 50, 51, 53

cgroup control group. 2, 5, 10, 11, 18, 19, 20, 46, 50, 51, 53, 60

FPGA Field-Programmable-Gate-Array. 5

GPU Graphics-Processing-Unit. 5

NIC Network-Interface-Controller. 5

OOM Out-of-Memory. 5, 10, 19, 21, 38, 44, 46, 58, 67

PID Process-ID. 2, 3

PRNG Pseudo-Random-Number-Generator. 31

QoS-Klasse Quality-of-Service-Klasse. 5, 10, 11, 12, 25, 53

RPC Remote-Procedure-Call. 12, 13, 50, 51, 52

UID User-ID. 2

VPA Vertical-Pod-Autoscaler. 37, 44, 45, 46, 47, 48, 57, 58, 59

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Unternehmen digitalisieren zunehmend ihre Produkte und Geschäftsmodelle [45, S. 1]. Informationssysteme unterstützen nicht mehr ausschließlich die internen Geschäftsprozesse, sondern werden fundamentaler Bestandteil der Produkte [45, S. 26]. Ein wesentlicher Treiber dieser digitalen Revolution ist das Cloud-Computing, welches den Bezug von informationstechnischen Dienstleistungen über das Netzwerk beschreibt [45, S. 4]. Vorteile durch das Cloud-Computing entstehen in der schnellen, flexiblen und risikoarmen Bereitstellung dieser Dienstleistungen [2, S. 115], welche eine schnelle Reaktion auf ein verändertes Marktumfeld erlaubt. Die Dienstleistungen beziehungsweise Applikationen werden durch den Cloud-Anbieter verwaltet. Dessen Kernkompetenz in der Informationstechnik und realisierte Skaleneffekte werden dadurch in die eigene Wertschöpfung einbezogen [45, S. 5]. Die unternehmensinternen Landschaften können vereinfacht werden [2, S. 3–4].

Unternehmen verlagern daher den Betrieb betrieblicher Informationssysteme zunehmend in die Cloud, welche niedrigere Kosten und größere Flexibilität bei sich ständig ändernden Geschäftsanforderungen verspricht. Die Realisierung dieses Potenzials hängt von der Umsetzung der IT-Landschaften in der Cloud ab. Bestandteil dieser Bewegung in die Cloud ist die Containerisierung der Applikationen. Dadurch ist Kubernetes zu einem weit verbreiteten Container-Orchestrator avanciert, der bei allen großen Cloudanbietern verfügbar ist [28, S. 2–3]. Im Jahr 2019 setzten mindestens 71% der Unternehmen auf der Fortune-100-Liste[1] Kubernetes produktiv ein [6, S. 1, 7]. Bedeutsam für eine erfolgreiche Adaption von Kubernetes ist die Implementierung des Schedulings, also die Zuordnung von Containern zu den Servern, die sie ausführen. „The topic of scheduling in Kubernetes is of paramount importance, since it can have negative consequences if not performed in an optimal manner“ [28, S. 3].

Die theoretischen Ausarbeitungen zum Scheduling in der Cloud sind umfangreich, allerdings oft noch auf virtuelle Maschinen bezogen [5, S. 2] und kaum in Kubernetes integriert. Bisher haben wenige Arbeiten das sogenannte Scheduling-Framework verwendet, um besseres Scheduling umzusetzen [28, S. 31]. Aufgrund der wachsenden Popularität von Kubernetes, lohnt sich die Entwicklung besserer Algorithmen für das Scheduling und die Ressourcenverwaltung [29, S. 23]. Ein Ansatz zur Opti-

[1] Die 100 größten amerikanischen Unternehmen nach Umsatz.

mierung letzterer besteht in der Nutzung von Vorhersagen über das Verhalten von Containern, welche mittels maschinellen Lernens generiert werden können [28, S. 31]. Auch in diesem Bereich besteht weiterer Untersuchungsbedarf [5, S. 31]. Diese Arbeit soll einen Beitrag dazu leisten, die aufgezeigten Lücken zu schließen. Welche Methoden zur Anpassung des Schedulings in Kubernetes-Clustern existieren, wie aufwendig ist deren Implementierung und wie gut verbessern diese die Zuordnungen von Containern zu Knoten?

Wirtschaftlich betrachtet führt eine Verbesserung im Scheduling zu einer erhöhten Auslastung der Hardware und damit zu einer Kostenreduktion, da weniger Hardware für die Ausführung der gleichen Applikationen erforderlich ist [28, S. 1, 3]. „In der Praxis ist overprovisioning und underutilization in Rechenzentren der Regelfall und eine optimale Auslastung ist kaum möglich. [...] Die finanziellen Folgen sind nicht unerheblich“ [2, S. 118]. Weitere Einsparungen ergeben sich potenziell im Kontext der operativen Betriebsprozesse durch die Vereinfachung jener Infrastruktur. Ebenso wird die Angriffsfläche als auch der Energieverbrauch durch den Abbau überschüssiger Kapazität reduziert. Vorteilhaft an Optimierungen, die sich in Software implementieren lassen, ist, dass jene verhältnismäßig einfach in die Praxis überführt werden können. Aus den genannten Gründen bezüglich der Verbreitung, Effizienz und Umsetzbarkeit ist die Thematik eine genaue Untersuchung wert.

1.2 Kubernetes

Container sind eine Form der Betriebssystemvirtualisierung, welche der Isolation verschiedener Prozesse dient. Dies fördert die Sicherheit und Vertraulichkeit der Prozesse, die sich das Betriebssystem teilen. Weiterhin werden Einflüsse einer geteilten Ausführungsumgebung minimiert [2, S. 13–14]. Für die Prozessisolation werden folgende drei Funktionen des Linux-Kernels verwendet:

- Control groups (cgroups), welche den Zugriff auf Ressourcen, wie CPU-Zeit, Arbeitsspeicher und Netzwerkbandbreite beschränken [20, S. 11–12].

- Namespaces, welche die Sicht eines Prozesses auf Ressourcen, wie beispielsweise Process-IDs (PIDs), User-IDs (UIDs), und Netzwerk-Interfaces, anpassen und begrenzen [20, S. 11].

- Union-Filesystems, welche die Überlagerung und Wiederverwendung von Dateibäumen ermöglichen und die Grundlage für Containerabbilder darstellen [18, S. 95].

Die voranschreitende Paketierung von kleinen Applikationskomponenten in Containern generiert den Bedarf nach einem System zur Verwaltung von Containern auf verschiedenen Servern. Ein solcher quell-offener Container-Orchestrator ist Kuber-

netes [28, S. 4]. Das System bietet umfangreiche Funktionalitäten zur Verwaltung und Automatisierung des Lebenszyklus von Containern [5, S. 4].

Der Einsatz von Kubernetes löst nicht ausschließlich technologische Herausforderungen bezüglich der verteilten Ausführung von Containern, sondern schafft auch eine klar definierte Schnittstelle zwischen Systemadministratoren und Anwendungsentwicklern. Die Administratoren müssen die Applikationen und deren Abhängigkeiten nicht mehr installieren und verwalten, da jene in Containern verpackt sind und von Kubernetes automatisch ausgeführt werden. Entwickler müssen Spezifika der Infrastruktur nicht mehr berücksichtigen, da diese lediglich mit dem abstrahierten Application-Programming-Interface (API) von Kubernetes interagieren. Durch die klar spezifizierte Ausführungsumgebung innerhalb der Container werden Fehler durch abweichende Umgebungen ausgeschlossen [20, S. 22–23].

Ein Kubernetes-Cluster besteht aus einer Menge an Servern, welche auch als Knoten bezeichnet werden, die sowohl die Container als auch die Systemkomponenten ausführen. Jeder Knoten verfügt über ein sogenanntes Kubelet, welches vom zentralen API-Server informiert wird, sobald Container zu starten oder zu stoppen sind. Diese Kommandos werden an die Container-Runtime, wie `containerd` oder `cri-o`, weitergereicht [5, S. 5]. Zusätzlich stellt das Kubelet die Schnittstelle für die Verwaltung von persistentem Speicher für Container bereit. Um den Netzwerkverkehr innerhalb des Clusters zu verteilen, wird auf jedem Knoten die `kube-proxy` Komponente ausgeführt. Diese konfiguriert auf Basis der Informationen vom zentralen API-Server verschiedene Regelketten mittels `iptables` [20, S. 326–327].

Es gibt weitere zentrale Systemkomponenten, welche oft als Controlplane bezeichnet werden. Dazu gehört der API-Server, welcher Informationen über im Cluster verwaltete Objekte entgegennimmt, validiert und verteilt. Weiterhin speichert der API-Server den Zustand des Clusters in der Schlüssel-Wert-Datenbank `etcd`. Grundlegende Funktionen bezüglich der Verwaltung des Lebenszyklus von Containern werden vom Controller-Manager, Container-Storage-Interface Komponenten und dem Kubernetes-Scheduler (kurz Kube-Scheduler) implementiert [5, S. 7]. Von allen eben genannten Komponenten lassen sich mehrere Replikate zwecks Hochverfügbarkeit erstellen [20, S. 342].

Die kleinste von Kubernetes verwaltetet Einheit ist der sogenannte Pod. Ein Pod besteht aus einem oder mehreren Containern, welche gemeinsam auf einem Knoten platziert werden und den gleichen Netzwerk-, Inter-Process-Communication- und (optional) PID-Linux-Namespace teilen [20, S. 57]. Prinzipiell ist anzustreben, pro Pod genau einen Container zu verwenden. Mehrere Container pro Pod dienen der Abbildung von Anwendungsfällen, welche eine solche enge Kopplung benötigen [28, S. 4–5]. Ein übliches Beispiel ist ein Applikationsserver dessen Logs von einem dedizierten Logshipper in einem weiteren Container eingesammelt und weitergeleitet werden.

Alle Typen von Objekten, welche Kubernetes verwaltet, teilen denselben grundlegenden Aufbau bestehend aus Metadaten, der Spezifikation des gewünschten Zustands und dem aktuellen Status des Objekts. Die Metadaten umfassen mitunter den Namen, den Namensraum innerhalb des Clusters (kein Linux-Namespace) und eine Menge von Labels. Ein Label ist ein willkürliches Schlüssel-Wert-Paar. Deren Semantik wird von verschiedenen Controllern festgelegt. Der API-Server indiziert Labels und ermöglicht damit ein effizientes Durchsuchen der im Cluster verwalteten Objekte [20, S. 67–69].

Pods sind im Kontext des Gesamtsystems als kurzlebige Ressourcen zu betrachten. Wird die Spezifikation eines Pods aus dem System entfernt, so werden die zugehörigen Container gestoppt. Dies kann beispielsweise bei Ausfall oder Wartung eines Knotens im Cluster auftreten. Container eines Pods können nicht zur Laufzeit auf einen anderen Knoten migriert werden. Es gibt mehrere übergeordnete Objekte, die in einem solchen Fall einen neuen Pod erstellen, um die Funktionsfähigkeit der Applikation zu gewährleisten [20, S. 84–85]. Für zustandslose Applikationen sind dies ReplicaSets und Deployments [20, S. 251–254] und für zustandsbehaftete Applikationen sogenannte StatefulSets [20, S. 280–285]. Weiterhin sind DaemonSets verwendbar, die einen Pod auf jedem Knoten im Cluster instanziieren [20, S. 108–109]. Diese übergeordneten Objekte finden die Pods, die sie verwalten, mit Hilfe von Labels. Außerdem wird in den Pods eine Referenz auf deren Besitzer gepflegt. Eine exemplarische Podspezifikation ist in Abbildung 1.1 gegeben.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: primetime
spec:
  containers:
  - name: primetime
    image: keppel.eu-de-1.cloud.sap/ccloud/primetime
    resources:
      requests:
        cpu: 500m # 500 MilliCPU = 0.5 Kerne
        memory: 128Mi
        nvidia.com/gpu: 1 # Erweiterte Ressource
      limits:
        cpu: 700m # 700 MilliCPU = 0.7 Kerne
        memory: 256Mi
        nvidia.com/gpu: 1 # Erweiterte Ressource
```

Abbildung 1.1: Beispielhafte Spezifikation eines Pods in der Burstable-QoS-Klasse

Pods werden durch den Kube-Scheduler den ausführenden Knoten zugeordnet. Im Rahmen dieser Arbeit sei der Prozess des Schedulings zunächst durch zwei Schritte, die Bewertung des Ressourcenbedarfs der Container und die Einplanung auf Knoten, beschrieben. Als Teil jeder Containerspezifikation können Requests und Limits für Ressourcen angegeben werden. Die entsprechenden Grenzen für einen Pod ergeben sich jeweils aus der Summe der Requests und Limits der enthaltenen Container [5,

S. 8]. Requests beschreiben die Minima an Ressourcen, welche einem Container zugeordnet und zur Laufzeit garantiert werden, und Limits beschreiben die Maxima, welche ein Container zu Laufzeit nicht überschreiten darf. Kubernetes unterstützt direkt CPU-Kerne, Arbeitsspeicher, Hugepages und Ephemeral-Storage (knotenspezifische Festplattenkapazität) als begrenzbare Ressourcen. Prinzipiell sollen sowohl Requests als auch Limits durch eine entsprechende Konfiguration der cgroups abgesichert oder erzwungen werden. Bei Überschreitung des Arbeitsspeicherlimits ist die cgroup Out-of-Memory (OOM), woraufhin der OOM-Killer innerhalb der cgroup aufgerufen wird. Die Überschreitung des CPU-Limits führt zu Throttling. Die tatsächliche Implementierung zur Konfiguration der cgroups besitzt noch Lücken. Carrión [5, S. 28] identifiziert ebenfalls die Verbesserung der Virtualisierung der Ressourcen als offene Herausforderung. Die Thematik der Ressourcenisolation wird in den Abschnitten 1.4 und 5.4 ausführlich bearbeitet.

Die Typen an Ressourcen für das Scheduling sind erweiterbar gestaltet, wobei zwischen knotenspezifischen und clusterweiten Ressourcen unterschieden wird. Anwendungsfälle für knotenspezifische Ressourcen können Ressourcen von Graphics-Processing-Units (GPUs), Network-Interface-Controllers (NICs) und Field-Programmable-Gate-Arrays (FPGAs) sein. Erweiterte Ressourcen werden in der `capacity` Spezifikation der Knoten und in den Requests und Limits der Container angegeben. Diese werden dann bei der Zuordnung der Pods zu Knoten zusätzlich vom Kube-Scheduler berücksichtigt [40]. Die Angabe der Kapazität von erweiterten Ressourcen der Knoten kann mittels Device-Plugins automatisiert werden [36].

Basierend auf der Ressourcenspezifikation ordnet Kubernetes den Pods eine Quality-of-Service-Klasse (QoS-Klasse) zu. Wenn der Festplatten- oder Arbeitsspeicher auf einem Knoten knapp wird, führt das Kubelet eine sogenannte Eviction durch. Dabei werden so lange Pods gelöscht, bis die Ressourcenknappheit beseitigt ist [38]. Zuerst werden Pods entfernt deren aktueller Verbrauch die Requests übersteigt, wobei Pods mit niedriger Priorität[2] und hoher Differenz zu den Requests bevorzugt entfernt werden. Sollte dies unzureichend sein, werden die restlichen Pods auch betrachtet, wobei ebenfalls Pods mit niedriger Priorität bevorzugt entfernt werden [38]. Ein Pod ist

- in der Guaranteed-Klasse, wenn jeder Container CPU- und Arbeitsspeicherrequests und -limits besitzt, die identisch sind,
- in der Burstable-Klasse, sobald ein Container ein CPU- oder Arbeitsspeicherrequest oder -limit besitzt,
- sonst in der Best-Effort-Klasse [39].

Basierend auf dem Auswahlverfahren für die zu entfernenden Pods nährt die QoS-Klasse die Häufigkeit an, in der Pods durch das Kubelet bei Ressourcenknappheit entfernt werden. Pods in der Best-Effort-Klasse werden vor Pods in der Burstable-

[2]Die Priorität ist als Teil der Spezifikation eines Pods frei wählbar.

Klasse entfernt, welche vor Pods in der Guaranteed-Klasse gelöscht werden [38]. Da Pods möglichst unterbrechungsfrei ausgeführt werden sollen, ist anzustreben, geeignete identische Requests und Limits zu setzen [5, S. 9]. Sind alle Pods auf einem Knoten in der Guaranteed-Klasse, ist eine vom Kubelet initiierte Eviction ausgeschlossen, da der Kube-Scheduler Knoten nicht überbelegt.

Wenn eine neue Spezifikation für einen Pod am API-Server eintrifft, ist dieser zunächst keinem Knoten zugeordnet. Kubernetes unterstützt die Verwendung verschiedener Scheduler. Diese sind Teil der Controlplane und werden üblicherweise als Pod im Cluster ausgeführt. Die Scheduler werden vom API-Server über neue Pods informiert [5, S. 9]. Der gewünschte Scheduler für einen spezifischen Pod wird durch das `schedulerName` Feld der Spezifikation definiert. Die Scheduler können eintreffende Pods nach diesem Feld filtern und wenn nötig ignorieren.

Der Kube-Scheduler ist die Referenzimplementierung [5, S. 9]. Dieser wählt im ersten Schritt den Pod mit der höchsten Priorität aus der Menge der zu planenden Pods aus. Pods werden nicht gemeinsam eingeplant. Dann werden Knoten aus der Gesamtmenge aller Knoten herausgefiltert, die bestimmte harte Bedingungen nicht erfüllen. Die verbleibenden Knoten werden dann bewertet. Der Pod wird auf dem Knoten mit der höchsten Bewertung eingeplant. Bei Gleichstand mehrerer Knoten wird einer dieser Knoten zufällig ausgewählt [28, S. 5].

Die internen Strukturen im Kube-Scheduler sind erweiterbar gestaltet, sodass neue Funktionalität integriert werden kann. Dafür gibt es verschiedene Erweiterungspunkte, die im Rahmen des Scheduling-Frameworks angeboten werden [42]:

- Sort: Auswahl des nächsten Pods, der geplant wird, aus der Menge der zu planenden Pods.
- PreFilter: Überprüfen von Vorbedingungen bezüglich des Pods oder des Clusters.
- Filter: Überprüfen der Erfüllung von harten Bedingungen bezüglich der Knoten. Mehrere Knoten können nebenläufig evaluiert werden.
  - PostFilter: Plugins werden aufgerufen, wenn die Knotenmenge nach dem Filtern leer ist. Wird für die Implementierung von Präemption verwendet.
- PreScore: Generieren von shared State für die Scoring-Plugins.
- Score: Bewerten der Knoten bezüglich eines Attributes.
- NormalizeScore: Normalisieren des Ergebnisses des zugehörigen Scoring-Plugins in das Intervall, welches alle Plugins nutzen.

Die Standardfunktionalität wird durch eine Reihe an Plugins realisiert, welche gemeinsam mit dem Scheduler kompiliert werden müssen [5, S. 11]. Deren detailliertes Verhalten kann mittels einer Konfigurationsdatei adjustiert werden. Prinzipiell kann jedes Plugin mehrere Erweiterungspunkte implementieren. Eine Betrachtung der Plugins bietet eine Übersicht über die Funktionalität, die angepasste Scheduler implementieren sollen. Der größere Teil der Plugins entfernt ungeeignete Knoten vor der Scoring-Phase. Oft können harte und weiche Bedingungen spezifiziert werden. Weiche Bedingungen werden im Scoring berücksichtigt [41].

Taints und Tolerations erlauben das Filtern nach Einschränkungen (Taints) von Knoten, wie beispielsweise fehlende Konnektivität. Die Resistenz gegenüber einer Einschränkung kann mit der entsprechend konfigurierten Toleranz ausgedrückt werden [20, S. 457–458]. Das NodeName-Plugin entfernt Knoten deren Name nicht dem Knotennamen in der Spezifikation des Pods entspricht. Bei Bedarf kann damit ein Pod einem vorher spezifizierten Knoten zugeordnet werden. Das NodePorts-Plugin stellt sicher, dass nur Knoten berücksichtigt werden, welche ausreichend viele freie Netzwerkports besitzen [41]. In der Spezifikation eines Pods können Labels integriert sein, welche geeignete Knoten besitzen müssen. Dies wird als Node-Affinity bezeichnet [20, S. 462–463]. Ein ähnlicher Mechanismus zur Verteilung über selbstgewählte Verfügbarkeitsdomänen sind Topology-Spread-Constraints, welche ebenfalls durch ein respektives Plugin implementiert sind. Abhängigkeiten zwischen Pods werden durch das InterPodAffinity-Plugin gelöst. Pods können Präferenzen oder Abneigungen für Pods auflisten, mit denen diese auf einem Knoten geplant werden [20, S. 468–474]. Ebenso werden Knoten mit unzureichenden Ressourcen gefiltert. Weiterhin werden verschiedene Vorbedingungen der Knoten bezüglich persistentem Speicher durch eine Reihe an Plugins sichergestellt [41].

Neben weichen Bedingungen werden die verbliebenen Knoten auch nach der lokalen Verfügbarkeit der Containerabbilder bewertet. Bezüglich der Ressourcennutzung werden in der Standardkonfiguration die Knoten bevorzugt, die über alle Ressourcen die kleinste gesamte Auslastung aufweisen. Die Container werden daher auf möglichst viele Knoten aufgeteilt. Außerdem gibt es ein Scoring-Plugin, das Knoten präferiert, deren Auslastung pro Ressourcentyp durch Ausführung eines neuen Pods ausgeglichener wird. Bei der Betrachtung von Ressourcen werden lediglich die Requests der bereits geplanten und des zu planenden Pods genutzt [41].

1.3 Vector-Bin-Packing

Die Scoring-Phase approximiert eine Lösung für das Vector-Bin-Packing-Problem (seltener Multi-Capacity-Bin-Packing-Problem). Leinberger, Karypis und Kumar [19, S. 404] geben folgende Definition dafür:

$$\vec{C} = (C_1, C_2, \ldots, C_j, \ldots, C_d) \tag{1.1}$$
$$C_j \geq 0 \tag{1.2}$$
$$\vec{X_i} = (X_{i1}, X_{i2}, \ldots, X_{ij}, \ldots, X_{id}) \tag{1.3}$$
$$C_j \geq X_{ij} \geq 0 \tag{1.4}$$
$$L = \{\vec{X_0}, \ldots, \vec{X_n}\} \tag{1.5}$$

$\vec{C}$ beschreibt die Kapazität von Knoten bezüglich d verschiedener Ressourcentypen und L den Ressourcenbedarf verschiedener Pods. Gesucht wird das kleinste m, welches $L = B_1 \cup \cdots \cup B_m$ partitioniert, sodass $\sum_{\vec{X_i} \in B_j} \vec{X_i} \leq \vec{C}, 1 \leq j \leq m$ gilt. B_j repräsentiert einen Knoten. Eindimensionales Bin-Packing ist bereits NP-hart [14, S. 94]. Um zeitnah eine qualitativ vertretbare Lösung zu finden, werden sowohl spezifische Heuristiken und Metaheuristiken als auch Modelle aus dem Bereich des maschinellen Lernens eingesetzt [5, S. 21–24].

Es kann zwischen der offline und der online Variante des Problems unterschieden werden. Bei der online Variante sind einzelne Elemente aus L gegeben, die nacheinander platziert werden müssen. Im Kontext von Kubernetes ist diese Variante naheliegender zu implementieren, da die Spezifikationen von Pods zu beliebigen Zeitpunkten eintreffen. In der offline Variante ist die ganze Menge L gegeben. Durch das Sortieren von L lassen sich bessere Heuristiken nutzen als für die online Variante [14, S. 95]. Dies lässt sich prinzipiell umsetzen, wenn alle bereits ausgeführten Pods ebenfalls erneut eingeplant werden. Je nach Frequenz der eintreffenden Spezifikationen ist dies jedoch nicht praktikabel, da Pods erst nach einer gewissen Verzögerung funktionstüchtig sind, besonders wenn persistenter Speicher involviert ist.

Bei ausschließlicher Betrachtung von Plugins in Kubernetes 1.27.0 [37], die Ressourcen bewerten, wird in der Standardkonfiguration des Kube-Schedulers folgende Heuristik genutzt, um ein neuen Pod $\vec{X_s}$ einem Knoten B_j zuzuordnen. Die verschiedenen bereits geplanten Pods $\vec{X_i}$ werden durch die Requests spezifiziert. Beschreibe

$$\vec{U_j} = (\vec{X_s} + \sum_{\vec{X_i} \in B_j} \vec{X_i}) \oslash \vec{C} \tag{1.6}$$

die Auslastung jedes Ressourcentyps eines Knotens B_j, falls der zu planende Pod auf dem jeweiligen Knoten ausgeführt wird[3]. Dann ist die Heuristik beschrieben durch:

[3]Durch $\oslash$ sei die Hadamard (elementweise) Division ausgedrückt.

$$S1_j = \frac{1}{d} \sum_{u \in \vec{U}_j} (1 - u) \tag{1.7}$$

$$S2_j = 1 - \sqrt{\frac{1}{d} \sum_{u \in \vec{U}_j} (u - \bar{U}_j)^2} \quad \text{mit} \quad \bar{U}_j = \frac{1}{d} \sum_{u \in \vec{U}_j} u \tag{1.8}$$

$$S_j = \frac{S1_j + S2_j}{2} \tag{1.9}$$

Der zu planende Pod $\vec{X}_s$ wird dem Knoten B_j mit dem größten Score S_j zugeordnet [37]. $S1_j$ bevorzugt Knoten mit geringer Auslastung[4]. $S2_j$ bevorzugt Knoten mit kleiner Standardabweichung zwischen den Auslastungen der Ressourcentypen, was aus Huang, Li und Qian [10, S. 316] entnommen wurde. Für S_j insgesamt ist dem Autor keine Quelle bekannt. Um Knoten möglichst stark auszulasten, kann $S1_j$ per Konfiguration zu

$$S1_j = \frac{1}{d} \sum_{u \in \vec{U}_j} u \tag{1.10}$$

geändert werden[5].

Wenn die Anforderungen an die verschiedenen Ressourcentypen ausreichend positiv linear korreliert sind, sollte das Problem als eindimensional aufgefasst werden [19, S. 412]. Gleiches gilt, wenn die Platzierung der $\vec{X}_i$ von lediglich einer Ressource entscheidend beeinflusst wird, da die anderen nicht knapp sind. Beides ist mit stärkeren Garantien des eindimensionalen Falls zur Annäherung an die optimale Lösung zu begründen [25, S. 4].

Das Nicht-Planen einer Ressource schafft keine Auslastung für diese. Im Cloud-Umfeld ist es üblich, dass Knoten von virtuellen Maschinen bereitgestellt werden. Sollte eine Ressource das Vector-Bin-Packing dominieren, sollte die Größe der virtuellen Maschinen angepasst werden, sodass zunächst nicht relevante Ressourcen knapper werden. Ein geeignetes Verfahren zum Finden einer Näherungslösung für das Vector-Bin-Packing ist für Cluster mit geeigneter Knotengröße folglich relevant.

[4] Später als Kubernetes-Least bezeichnet.
[5] Später als Kubernetes-Most bezeichnet.

1.4 Control-Groups

Abschnitt 1.2 hat aufgezeigt, dass die Requests eines Pods zur Zuordnung auf die Knoten verwendet werden. Sowohl Requests als auch Limits werden auf Ebene des Knotens durch das Kubelet und die Container-Runtime in verschiedenem Maße durchgesetzt.

Das Limit für den Arbeitsspeicher wird durch den `memory.max` cgroup-Schlüssel implementiert [46]. Überschreitet ein Container diese Grenze wird eine Memory-Reclaimation durchgeführt. Ist die Grenze immer noch überschritten wird der OOM-Killer innerhalb der cgroup aufgerufen [9]. Bezüglich des Requests für den Arbeitsspeicher fragt das Kubelet keine Konfiguration bei der Container-Runtime an. Dies bedeutet, dass der Request nur beim Scheduling berücksichtigt wird [46]. Zur Laufzeit ist diese Menge an Speicher nicht unbedingt garantiert belegbar [48, S. 2]. Werden Pods in der Best-Effort- oder Burstable-QoS-Klasse auf dem gleichen Knoten ausgeführt, können diese mehr Arbeitsspeicher belegen als in ihren Requests spezifiziert ist.

Das `MemoryQoS`-Feature, welches mit Kubernetes 1.22 mit dem Alpha-Reifegrad eingeführt wurde [46] und in 1.27 weiterhin im Alphazustand ist [37], konfiguriert den `memory.min` und `memory.high` Schlüssel, um der Problematik entgegenzuwirken. Die als `memory.min` angegebene Menge kann vom System nicht zurückverlangt und daher nicht für andere Container verwendet werden [9]. Überschreitet der Verbrauch den `memory.high` Wert bekommen die Prozesse innerhalb der cgroup weniger CPU-Zeit, die stattdessen für die Memory-Reclaimation verwendet wird [9]. Der `memory.high` Schlüssel wird ausschließlich für Pods in der Burstable- und Best-Effort-QoS-Klasse nahe dem Limit[6] gesetzt [46].

Der Completly-Fair-Scheduler (CFS) ist der Taskscheduler im Linux-Kernel[7]. Dessen Mechanismen werden von den Container-Runtimes und damit respektiv Kubernetes genutzt, um die CPU-Zeit von Containern zu verteilen und zu beschränken. Grundlage des CFS ist das Modell einer „ideal multi-tasking CPU“ [33], welche jeden Task exakt parallel jeweils mit der Geschwindigkeit des Reziproken der Prozessanzahl ausführt. Für die Ausführung auf tatsächlicher Hardware wird das Konzept der virtuellen Laufzeit genutzt. Diese entspricht der bisherigen Ausführungszeit geteilt durch die Anzahl aller Tasks. Der als nächstes auszuführende Task ist derjenige mit der kleinsten virtuellen Laufzeit. Dadurch wird die Zeit möglichst entsprechend der „ideal multi-tasking CPU“ aufgeteilt [33].

CPU-Requests werden durch den `cpu.weight` cgroup-Schlüssel umgesetzt. Jeder Prozess bekommt dabei Zeit entsprechend seines Anteils an der Summe der Gewichtungen zugeordnet. Zu jener Summe tragen nur Prozesse bei, welche gerade CPU-Zeit beanspruchen, weshalb eine untere Schranke bezüglich der CPU-Zeit ga-

[6] Ist kein Limit spezifiziert, wird stattdessen die Kapazität des Knotens verwendet.
[7] Abseits der Echtzeit-Scheduler sind alternative Scheduler nicht im Kernel enthalten.

rantiert wird. Nicht genutzte Zeit kann von anderen Prozessen verwendet werden [9].

Die obere Schranke von CPU-Zeit wird durch die CFS-Bandwidth-Control realisiert. Dabei gibt es zwei relevante konfigurierbare Variablen, die Periode und die innerhalb der Periode nutzbare CPU-Zeit. Letztere wird als Quota bezeichnet [44, S. 337–338]. Beide Werte werden im `cpu.max` cgroup-Schlüssel abgelegt. Das Quota kann auf mehrere CPU-Kerne verteilt werden. Beispielsweise kann ein Quota von 50 Millisekunden vollständig auf einen Kern verteilt werden oder jeweils zehn Millisekunden auf fünf Kerne, je nachdem auf welchen Kernen verschiedene Threads ausgeführt werden. Ist das Quota verbraucht, werden die Threads bis zum Beginn der nächsten Periode pausiert. Dies wird als Throttling bezeichnet [32].

Die verschiedenen cgroup-Einstellungen sind hierarchisch organisiert. Beschreibe R_C die CPU-Requests in MilliCPU (Eintausendstel Kern), R_M die Memory-Requests in Bytes, L_C, L_M die respektiven Limits eines Containers c, C die Menge der Container in einem Pod p, P die Menge der Pods auf einem Knoten. Schlüssel auf Ebene der Container werden für einen Pod in der Guaranteed-QoS-Klasse in Kubernetes 1.27 folgendermaßen konfiguriert:

$$\text{memory.min} = \begin{cases} 0, \text{ohne MemoryQoS} \\ R_M, \text{mit MemoryQoS} \end{cases} \tag{1.11}$$

$$\text{memory.max} = L_M \tag{1.12}$$

$$\text{cpu.weight} = \frac{R_C}{\sum_{p \in P} \sum_{c \in C_p} R_{Cpc}} \tag{1.13}$$

$$\text{cpu quota} = \frac{L_C}{10}\ ms \quad \text{cpu period} = 100\ ms \tag{1.14}$$

Auf der Ebene der Pods gilt:

$$\text{memory.min} = \begin{cases} 0, \text{ohne MemoryQoS} \\ \sum_{c \in C} R_{Mc}, \text{mit MemoryQoS} \end{cases} \tag{1.15}$$

$$\text{memory.max} = \sum_{c \in C} L_{Mc} \tag{1.16}$$

$$\text{cpu.weight} = \frac{\sum_{c \in C} R_{Cc}}{\sum_{p \in P} \sum_{c \in C_p} R_{Cpc}} \tag{1.17}$$

$$\text{cpu quota} \approx \frac{1}{10} \sum_{c \in C} L_{Cc}\ ms \quad \text{cpu period} = 100\ ms \tag{1.18}$$

Die Interaktion von Applikationen, welche mehrere Threads nutzen, mit der Bandwidth-Control erfordert eine besondere Beachtung [44, S. 339]. Kubernetes wird zum Betrieb verteilter Systeme eingesetzt, bei denen die Bearbeitungszeit von Remote-Procedure-Calls (RPCs) für die Performance entscheidend ist. Die benötigte CPU-Zeit eines RPC, welcher von genau einem Thread abgehandelt wird, sei konstant modelliert und mit t_c bezeichnet. Ohne Begrenzung der Periode und des Quotas beträgt die Bearbeitungszeit t_w des RPC $t_w = t_c$. Sei t_p die Periode, t_q das Quota und die Bearbeitung beginne immer am Anfang der Periode. Auf einem einzelnen Kern ist die Bearbeitungszeit eines RPC, wenn $t_q \leq t_p$, gegeben durch:

$$t_w = t_c + (\left\lceil \frac{t_c}{t_q} \right\rceil - 1) * (t_p - t_q) \tag{1.19}$$

Der zweite Summand beschreibt die Zeit, welcher der Prozess aufgrund von Throttling warten muss. Dessen erster Faktor drückt die Anzahl der Perioden aus, in denen Throttling auftritt, der zweite Faktor die Wartezeit pro Periode. Die Bearbeitungszeit eines RPC steigt, sobald $t_c > t_q$ gilt, aufgrund der Einführung der CPU-Begrenzung. Die Bearbeitungszeit wächst, umso kleiner t_q und umso größer t_p wird. t_p wird durch die Container-Runtime konfiguriert und ist in Kubernetes nicht weiter konfigurierbar. t_q ist in Kubernetes als CPU-Limit konfigurierbar. Das Festlegen eines CPU-Limits zwecks besserer Planbarkeit und Einordnung in die Guaranteed-QoS-Klasse (siehe Abschnitt 1.2) führt zu einer schlechten von außen wahrgenommenen Performance der Anwendung. Dies gestaltet die Spezifikation eines Limits unattraktiv und führt zu einer Überbewertung, um die Verschlechterung der Performance abzuschwächen [47, S. 217].

Der zeitliche Ablauf des Throttlings ist in Abbildung 1.2 beispielhaft für einen RPC mit $t_c = 500\ ms$ skizziert, wobei ein RPC pro Sekunde eintreffe. 500 MilliCPU beschreiben diesen Bedarf exakt, wodurch sich $t_q = 50\ ms$ und $t_p = 100\ ms$ ergibt. Während der orangenen Blöcke wird der RPC bearbeitet. In den Weißen wird der Prozess nicht ausgeführt. Nach Gleichung 1.19 ergibt sich $t_w = 950\ ms$. Die Bearbeitung dauert durch die Einführung des CPU-Limits 450 ms länger.

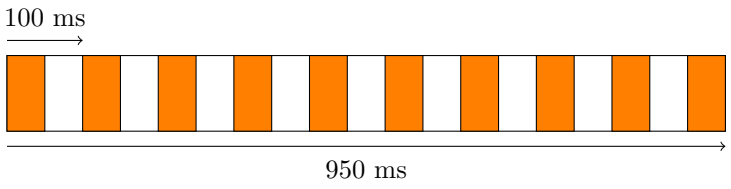

Abbildung 1.2: Schematische Darstellung des CPU-Throttlings

Dieser Effekt tritt bei Anwendungen, die mehrere Threads benutzen, verstärkt auf. Es bezeichne n die Anzahl gleichzeitig eintreffender RPCs und gleichzeitig die Anzahl an Kernen, auf denen diese bearbeitet werden. Das Quota sei gleichmäßig auf alle

Kerne verteilt $t'_q = t_q/n$. Die Bearbeitungszeit der RPCs, wenn $t_q \leq n * t_p$, ist dann gegeben durch:

$$t_w = t_c + (\left\lceil \frac{t_c * n}{t_q} \right\rceil - 1) * (t_p - \frac{t_q}{n}) \tag{1.20}$$

Umso paralleler die Anwendung, desto länger dauert der einzelne RPC. CPUs mit zweistelliger Anzahl an Kernen sind in Rechenzentren bereits länger etabliert. Webserver bearbeiten Anfragen häufig parallelisiert, da Anfragen oft unabhängig voneinander bearbeitet werden können. Die Kapazität für parallele Anwendungen steht bereit. Das Throttling erschwert aber eine optimale Nutzung. Das Warten auf I/O-Operationen verbraucht kein Quota, da in dieser Zeit andere Prozesse auf der CPU ausgeführt werden.

Prinzipiell kann $t_q > t_p$ konfiguriert werden. Dann kann mehr als ein voller Kern pro Periode verwendet werden. Das Verhalten, das zunächst die Threads auf möglichst viele Kerne verteilt werden, was das Quota schnell aufbraucht, bleibt aber erhalten. Damit tritt weiterhin der eben beschriebene Effekt der verlängerten Bearbeitungszeit bei der Einführung eines CPU-Limits auf.

2 Methodik

Scheduling ist ein umfangreich betrachtetes Themengebiet in der Informatik, welches sich gut mittels quantitativer Methoden untersuchen lässt. Verschiedene Algorithmen können auf eine identische Menge an Jobs angewendet und mittels einer Metrik, wie beispielsweise der Auslastung, verglichen und bewertet werden. Informationssysteme sind in einen sozialen Kontext eingebettet, da diese von Organisationen und Individuen eingesetzt werden, um einen Nutzen zu erhalten [27, S. 3]. Eine rein quantitative Betrachtung aktiv eingesetzter Informationssysteme ist nur begrenzt nützlich, da sozio-technische Aspekte vernachlässigt werden. Um diesen Aspekten gerecht zu werden, sind vielfältigere Methoden zu berücksichtigen [27, S. 4]. Das theoretisch beste Verfahren wird in der Praxis möglicherweise keine Anwendung finden, da es zu kostspielig in der Implementierung sein kann oder dessen Entscheidungen für Administratoren nicht mehr nachvollziehbar sind.

Die Converged Cloud ist ein internes Infrastrukturangebot des Unternehmens SAP. Mit der Converged Cloud ist für diese Arbeit ein konkreter Untersuchungsgegenstand gegeben, bei dem ein Bedarf zur Verbesserung des Schedulings in den betriebenen Kubernetes-Clustern besteht. Daher bietet sich Design Science als Forschungsmethodik an. Diese legt den Fokus auf die Schaffung eines Artefakts, welches bisherige Lösungen zu einem Problem verbessert. Design Science verbindet die konkrete Untersuchungsumgebung mit dem wissenschaftlichen Stand des Themengebiets, um die Verbesserung zu realisieren [27, S. 106].

Johannesson und Perjons [13, S. 76] schlagen ein Rahmenwerk mit fünf Aktivitäten vor, welchen in dieser Arbeit gefolgt wird. Die erste Aktivität ist die Bestimmung einer detaillierten Problembeschreibung, welche in Kapitel 3 vorgenommen wird. Diese wird mittels Beobachtung der Zustände in den Clustern und semi-strukturierten Experteninterviews gewonnen.

Als Zweites sind Anforderungen an das Artefakt zu definieren, welche vor allem die zu planenden Ressourcen umfassen. Kubernetes schafft eine Abstraktionsschicht zwischen den auszuführenden Applikationen und der informationstechnischen Infrastruktur. Ebenso zwischen dem respektiven Personal. Administratoren bieten Entwicklern eine Plattform als Dienstleistung für den Betrieb der zugehörigen Container an. In diesem Rahmen sind die Administratoren Anbieter des IT-Services und die Entwickler Kunden. Für die Erfassung von Anforderungen an das Scheduling bietet sich die Kano-Methode an. Das Kano-Modell beschreibt fünf verschiedene Zusammenhänge zwischen der Anforderungserfüllung und der Kundenzufriedenheit [11,

S. 81–83]. Durch eine Umfrage, die entsprechend dem Kano-Modell aufgebaut ist und in Kapitel 4 ausgewertet wird, lassen sich die Anforderungen den verschiedenen Zusammenhängen zuordnen und priorisieren [11, S. 111–112].

Der dritte Schritt ist die Entwicklung des Artefakts. Dafür werden verschiedene Verfahren in Kapitel 5 zur Beeinflussung des Schedulings in Kubernetes identifiziert. Dies ist ein innovativer Prozess und muss daher nicht direkt einer etablierten Forschungsmethodik zugeordnet werden [13, S. 125]. Fokussiert werden Heuristiken für das Vector-Bin-Packing und die Ableitung des Ressourcenbedarfs von Containern aus historischen Daten. Zwecks Identifikation bereits implementierter Ansätze liegt eine Literaturrecherche nahe. Für die Bewertung werden Simulationen genutzt. Bisher hat sich kein standardisiertes Verfahren für die Simulation des Schedulings in Kubernetes etabliert [5, S. 28]. Die Bestimmung der Auslastungsgrenzen von Knoten und die Vermeidung von CPU-Throttling finden als angrenzende Themen Berücksichtigung.

Die identifizierten Anpassungen werden im vorletzten Schritt in einer Testumgebung angewendet, um den praktischen Nutzen zu erfassen. Dabei kann der Zustand eines Clusters vor und nach der Anpassung des Schedulings verglichen werden. Die Ergebnisse werden in Kapitel 6 präsentiert. Zuletzt werden die Anpassungen in Kapitel 7 evaluiert. Zusammengefasst ist dies im Design Science Canvas in Tabelle 2.1 dargestellt.

<table>
<tr><td colspan="2">Problem
Suboptimales Scheduling führt zu ungenutzten Ressourcen und Störungen der Applikationen.</td><td colspan="2">Artefakt
Instanziierung: Anpassung oder Erweiterung des Kube-Schedulers, um zu besseren Plänen zu gelangen.</td><td>Wissensbasis
Literatur zu Scheduling im Cloud-Kontext.</td></tr>
<tr><td colspan="2">Praktik
Bereitstellung von Applikationen in Kubernetes an der Administratoren als auch Entwickler teilhaben.</td><td colspan="2">Anforderungen
Bestimmung der zu planenden Ressourcen und weiterer Einschränkungen.</td><td>Constructs
Vector-Bin-Packing, Zeitreihenanalyse, automatische Skalierung</td></tr>
<tr><td>Beschreibung des Problems
Experten-interview</td><td>Anforderungs-analyse
Kano-Methode, Umfrage</td><td>Entwicklung des Artefakts
Simulation, Experiment</td><td>Demonstration
Experiment</td><td>Evaluation
Diskussion</td></tr>
<tr><td colspan="2">Struktur
Konfigurationen und Erweiterung des Kubernetes Schedulers</td><td colspan="2">Funktion
Anpassung des Scheduling Algorithmus</td><td>Effekt
Bessere Platzierung von Containern auf Knoten</td></tr>
</table>

Tabelle 2.1: Design Science Canvas

3 Problembeschreibung

3.1 Converged Cloud

Die SAP Converged Cloud ist eine interne Cloud-Infrastruktur. Diese ist der Infrastructure-as-a-Service-Schicht zuzuordnen, welche eine abstrahierte Sicht auf die Hardware bereitstellt [2, S. 31–32]. Eine eindeutige Einordnung in die Bereitstellungsmodelle des Cloud-Computings ist nicht möglich, da externe Kundensysteme in der Converged Cloud ausgeführt werden, die APIs jedoch nicht öffentlich zugänglich sind. Dies ist eine Mischung der Konzepte der Private und Public Cloud [2, S. 27–28]. Converged Cloud ist in 15 verschiedenen produktiven Regionen in Europa, Asien, Nord- und Südamerika verfügbar.

Die Infrastrukturdienste werden durch OpenStack [24] realisiert. Die verschiedenen OpenStack-Dienste werden in Kubernetes ausgeführt. Um die Verfügbarkeit der Kubernetes-Controlplane zu gewährleisten, wird in jeder Region eine Hierarchie von Kubernetes-Clustern installiert, die in Abbildung 3.1 dargestellt ist.

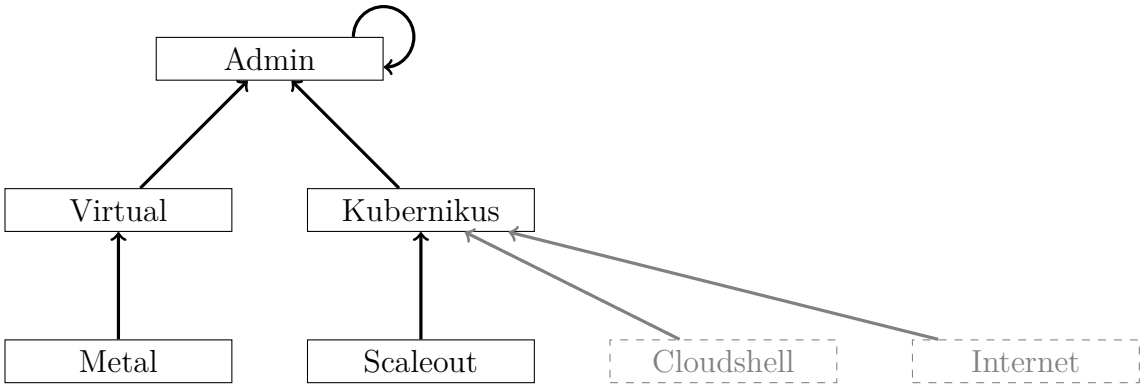

Abbildung 3.1: Architektur der Cluster in Converged Cloud

Die Pfeile zeigen auf das Cluster, welches die Controlplane-Komponenten des ausgehenden Clusters ausführt. In jeder Region sind die vollumrahmten Cluster aufgesetzt, die gestrichelten nur in ausgewählten Regionen. Die Trennung der Controlplane von deren Clustern dient der Isolation. Die Controlplane-Komponenten können nicht von den Anwendungen oder Wartungsarbeiten in den zugeordneten Clustern beeinflusst werden.

- Das Admin-Cluster besteht aus einer virtuellen Maschine deren Verfügbarkeit durch den ausführenden Hypervisor gewährleistet wird. Dieses beinhaltet die Controlplanes für das Virtual und Kubernikus Cluster einer Cloud-Region.

- Das Virtual-Cluster besteht aus vier virtuellen Maschinen und führt die Controlplane des Metal-Clusters aus. Im Gegensatz zum Admin-Cluster führt das Fehlen einer virtuellen Maschine nicht zu einem Ausfall der Controlplane, da diese auf einen anderen Knoten umziehen können.

- Das Metal-Cluster beinhaltet die verschiedenen OpenStack-Dienste, welche „die Cloud bereitstellen“, weshalb die Rechenkapazität lediglich beschränkt skalierbar ist. Die zwei- bis dreistellige Anzahl an Knoten wird durch wenige Cluster von Hypervisors realisiert.

- Das Kubernikus-Cluster führt den Kubernikus-Dienst aus, welcher eine API zur automatischen Einrichtung weiterer Cluster bereitstellt. Dieses Cluster enthält daher mehrere Controlplanes verschiedener untergeordneter Cluster.

- Das Scaleout-Cluster führt verschiedene Dienste aus, die nicht für eine grundlegende Installation von OpenStack benötigt werden. Im Gegensatz zum Metal-Cluster ist durch den Zugriff auf die OpenStack-APIs die Rechenkapazität simpel zu skalieren.

- Das Cloudshell-Cluster stellt den internen Nutzern eine per Browser erreichbare vorkonfigurierte Shell bereit, welche mit den OpenStack-Diensten interagieren kann. Diese werden zwecks Isolation in einem zusätzlichen Cluster ausgeführt

- Das Internet-Cluster beinhaltet Dienste, welche direkt aus dem öffentlichen Internet erreichbar sind.

Produktiv werden insgesamt 80 Cluster betrieben. Im Rahmen dieser Arbeit wird nur eine Teilmenge dieser betrachtet. Rejiba und Chamanara [28, S. 11] nutzen in ihrer Metaanalyse zum Scheduling eine Kategorisierung der untersuchten Arbeiten. Eine Kategorie ist dabei die Art der zu planenden Anwendung. Die Zielumgebung ist eine weitere Kategorie, wobei lediglich Cluster in der Cloud- und im Edge-Computing untersucht wurden. Der Großteil der Arbeiten bezieht sich auf diese Anwendungsdomäne [28, S. 7]. Carrión [5, S. 20–21] nennt ebenso lediglich Cloud-, Edge- und Fog-Computing als Anwendungsdomänen für Kubernetes. Der lokale Einsatz (on-premise) von Kubernetes, wie bei den Metal-Clustern scheint wenig Berücksichtigung zu finden. Zusammen mit der Anzahl der Regionen, in denen ein Clustertyp installiert ist, und der Größe dieser Cluster lassen sich die für diese Arbeit relevanten Cluster bestimmen, wie in Tabelle 3.1 dargestellt.

Clustertyp	Regionen	Knotenanzahl	Workload
Admin	Alle	$= 1$	homogen (2 Controlplanes)
Virtual	Alle	$= 4$	homogen (1 Controlplane)
Metal	Alle	≈ 45 bis ≈ 235	heterogen, zustandsbehaftet (OpenStack-APIs und Datenbanken)
Kubernikus	Alle	$= 4$ bis $= 45$	homogen (mehrere Controlplanes)
Scaleout	Alle	≈ 20 bis ≈ 120	heterogen (weitere Dienste mit Datenbanken)
Cloudshell	1	$= 5$	homogen (Cloudshells)
Internet	1	$= 3$	zustandslos (verschiedene Dienste, welche von externen Diensten aus dem Internet aufgerufen werden)

Tabelle 3.1: Clustertypen in Converged Cloud

In bisherigen Untersuchungen zum Scheduling in Kubernetes wurden bisher tendenziell kleine Cluster betrachtet [5, S. 28–29]. 70 Prozent der Untersuchungen, die von Rejiba und Chamanara [28, S. 31] betrachtet wurden, evaluieren das Scheduling mit weniger als zehn Knoten. Die Analyse der großen Clustertypen, also der Metal- und Scaleout-Cluster, ist daher aus Sicht des Forschungstandes lohnender. Weiterhin wird in der Literatur oft das Scheduling für bestimmte Arten von Workloads behandelt. Generische Lösungen für heterogene Applikationen in Clustern mit verschiedenen Nutzergruppen werden seltener betrachtet [5, S. 26].

3.2 Analyse der Auslastung

Die Bewertung der Zuordnung von Containern zu Knoten erfordert die Erfassung der Auslastung von Containern als auch Knoten. Relevant für die Problembeschreibung sind lediglich die CPU-Zeit und der Arbeitsspeicher, weil diese die einzigen Ressourcen sind, die bisher überhaupt Berücksichtigung fanden und erfasst wurden. Aktuelle Informationen zur Nutzung dieser Ressourcen werden in den Interface-Dateien der cgroups ausgegeben [9]. Das in das Kubelet integrierte Programm cAdvisor [4] stellt diese Metriken über eine API bereit [20, S. 430].

Es werden mehrere Metriken bezüglich der Nutzung von Arbeitsspeicher innerhalb eines Containers exportiert. Der Linux-Kernel stellt Prozessen virtuelle Speicheradressen zur Verfügung. Die Speicherbereiche, welche von einem Prozess verwendet werden, sind in sogenannten Pages organisiert. Eine Page ist dabei von physischem Arbeits- oder Festplattenspeicher gedeckt, wenn Optimierungen bezüglich der erstmaligen Initialisierung von Speicherbereichen außer Acht gelassen werden. Der Kernel kann bei Bedarf Pages zwischen den zwei Medien verschieben [34]. Die Menge aller Pages eines Prozesses, welche durch physischen Arbeitsspeicher gedeckt sind, wird als Resident-Set bezeichnet. Die Menge aller aktiv verwendeten Pages eines

Prozesses wird als Working-Set bezeichnet. Das Working-Set umfasst das Resident-Set. Allerdings beschreibt das Working-Set nicht die Speichermenge, welche durch den `memory.max` cgroup-Schlüssel beschränkt wird. Für Lese- und Schreibzugriffe auf reguläre Dateien verwaltet der Kernel einen Page-Cache, welcher diese Dateien im Arbeitsspeicher zwecks schnellen Zugriffs vorhält. Zusätzlich werden weitere Kernelstrukturen, wie beispielsweise Socket-Buffers und Metainformationen des Dateisystems, dem besitzenden Prozess zugerechnet. Die Menge an Speicher, welcher der `memory.max` Schlüssel limitiert, ist das Working-Set zuzüglich des Page-Cache und solcher Kernelstrukturen [9]. Bei Knappheit an Speicher innerhalb einer cgroup wird zunächst der Page-Cache geleert. Sollte dies nicht ausreichen, wird ein Prozess innerhalb der cgroup vom OOM-Killer beendet.

Im Rahmen dieser Arbeit sei das Optimum für den Arbeitsspeicherbedarf derart zu bestimmen, dass das Limit im `memory.max` Schlüssel möglichst nahe aber über dem tatsächlichen Working-Set liegt. Das Abschätzen einer oberen Grenze für die Größe des Working-Sets wird als herausfordernd beschrieben [9]. Die Entwicklung der Größe des Working-Sets wird in der `container_memory_working_set_bytes` Metrik nachverfolgt. Es sei darauf hingewiesen, dass die Performance I/O-intensiver Prozesse von einem mangelnden Page-Cache negativ beeinflusst wird [48, S. 2].

Die CPU-Zeit wird durch die `container_cpu_usage_seconds_total` Metrik erfasst, welche die verwendete CPU-Zeit seit Start des Containers angibt. Durch Differenzbildung kann die verbrauchte CPU-Zeit innerhalb eines Zeitintervalls bestimmt werden. Die nutzbare Zeit kann mittels der cgroups auf eine bestimmte Zeit innerhalb einer definierbaren Periode beschränkt werden, wie in Abschnitt 1.4 erläutert. Sind Threads ausführbar, die Zeitschranke jedoch überschritten, werden diese nicht ausgeführt [32]. Diese Wartezeit wird in der `container_cpu_cfs_throttled_seconds_total` Metrik erfasst. Durch Betrachtung dieser Metrik kann ein Mangel an CPU-Zeit innerhalb einer cgroup erkannt werden[1]. Alternativ kann auch das Verhältnis der CFS-Perioden mit Throttling zu allen CFS-Perioden betrachtet werden.

cAdvisor [4] stellt die eben diskutierten Metriken für jeweils einen Knoten bereit. Um die Anzahl der direkten Abfragen auf die Knoten zu minimieren, werden die Metriken von wenigen cluster-weiten Komponenten regelmäßig abgefragt, welche die Daten dann für die Weiterverarbeitung bereitstellen. Für die kurzfristige Speicherung dient der Metrics-Server [21], welcher die Daten im Arbeitsspeicher vorhält und nicht persistiert. Dieser stellt die Informationen über die Ressourcennutzung für die verschiedenen Autoscaling-Projekte bereit. Converged Cloud betreibt mehrere Instanzen der Zeitreihendatenbank Prometheus [26], welche die Metriken historisiert und persistiert. Thanos [31] bietet die Funktionalität mehrere Prometheus-Instanzen auf einmal abzufragen. Für die Analyse der aktuellen Auslastung der Cluster stammen die Informationen aus den verschiedenen Prometheus-Instanzen beziehungsweise Thanos. Der gesamte Informationsfluss ist in Abbildung 3.2 visualisiert.

[1] Verschiedene weitere Nuancen der Interaktionen zwischen dem Kernel und cgroups werden in [32] und [48] diskutiert.

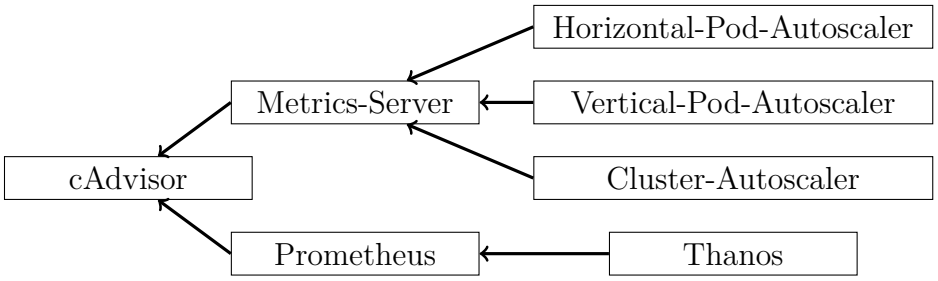

Abbildung 3.2: Informationsfluss Containermetriken

Basierend auf eben vorgestellten Metriken lässt sich die Ressourcennutzung in den Clustern evaluieren. Die folgenden Diagramme stellen die Situation in den verschiedenen Scaleout-Clustern dar. Zunächst sei die tatsächliche Nutzung einer Ressource, wie CPU-Zeit oder Arbeitsspeicher, der Kapazität gegenübergestellt. Ebenso können die Bedarfsinformationen (Requests und Limits) von Pods in Prometheus abgelegt und pro Knoten oder Region aufsummiert werden. Dadurch kann ein Vergleich zur Auslastung und Kapazität durchgeführt werden, wie in Abbildung 3.3 zu sehen ist.

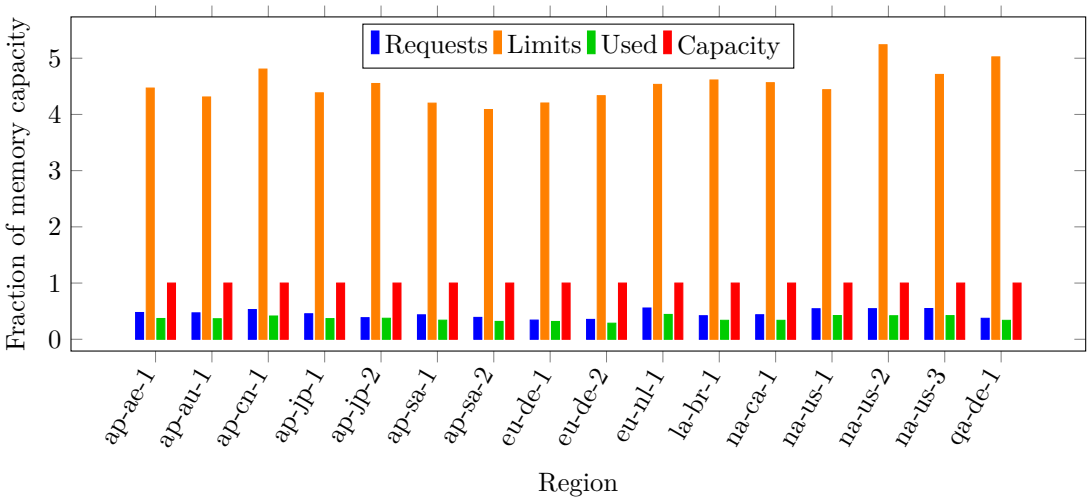

Abbildung 3.3: Durchschnittlicher Arbeitsspeicherbedarf für die Scaleout-Cluster am 13. März

Die absoluten Zahlen sind auf die Gesamtkapazität im jeweiligen Cluster normalisiert, da sich die Gesamtkapazität entsprechend der Aktivität der Anwender in den Regionen nennenswert unterscheidet. Wenn für einen Container keine Limits angegeben sind, wurde der gesamte verfügbare Arbeitsspeicher des Knoten als Limit angenommen. Dieser kann von solchen Containern theoretisch allokiert werden, da dann keine Begrenzung in der cgroup konfiguriert wird. Ersichtlich ist, dass die Cluster aufgrund der nicht konfigurierten Limits bezüglich des Arbeitsspeichers mehrfach überbelegt sind. Dies zeigt einen Verbesserungsbedarf auf, führt aber nicht direkt zu operativen Problemen, solange die Container nicht die entsprechende Menge an Arbeitsspeicher allokieren. Ebenso lässt sich erkennen, dass die Cluster lediglich zu rund 50% ausgelastet sind. Die Requests sind leicht höher als der Arbeitsspeicher

der tatsächlich allokiert ist, was positiv zu bewerten ist. Auf Ebene der einzelnen Knoten zeigt sich dennoch ein Ungleichgewicht, wie in Abbildung 3.4 dargestellt.

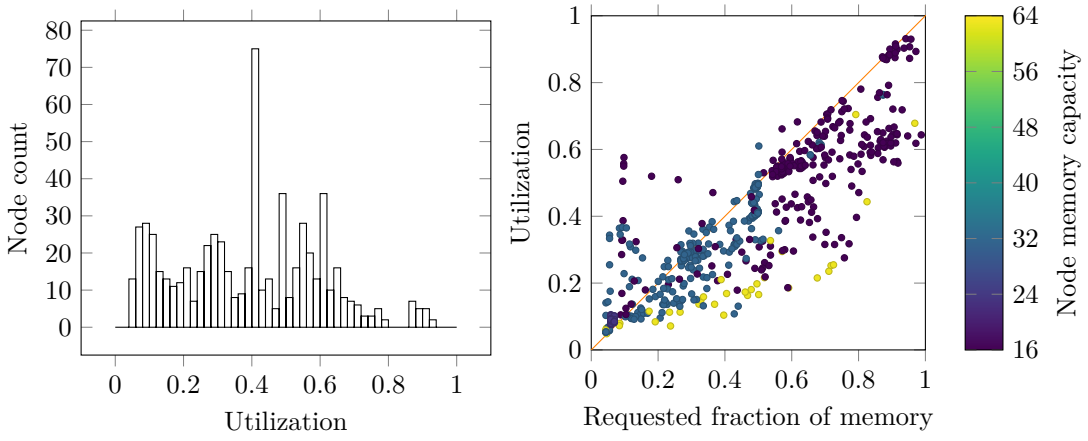

Abbildung 3.4: Histogramm und Streudiagramm der durchschnittlichen Auslastung des Arbeitsspeichers für alle Knoten der Scaleout-Cluster am 13. März

Der Großteil der Knoten ist zu weniger als 80% ausgelastet, rund die Hälfte zu weniger als 50%, wie dem Histogramm entnommen werden kann. Im Allgemein sollte die Ressourcenspezifikation der Container recht nahe am tatsächlichen Bedarf sein. Ist die Spezifikation zu niedrig wird der OOM-Killer aufgerufen, ist diese zu hoch bleiben Ressourcen ungenutzt. Dieser Zusammenhang ist im Streudiagramm durch die orangene Linie $y = x$ visualisiert, welche den optimalen Zustand angibt. Ein loser Zusammenhang zwischen den aufsummierten Spezifikationen der Container pro Knoten mit der tatsächlichen Nutzung ist gegeben. Steigt der spezifizierte Arbeitsspeicher steigt auch die Auslastung. Abweichungen existieren in beide Richtungen, wobei zu niedrig angesetzte Spezifikationen überwiegen. Knoten mit 64 GiB Arbeitsspeicher sind tendenziell am schlechtesten ausgelastet, Knoten mit 16 GiB Arbeitsspeicher am meisten. Die Pods in den Scaleout-Clustern haben oft Nebenbedingungen bezüglich der Knoten, auf denen diese eingeplant werden können. Es lässt sich vermuten, dass ein Großteil der Pods nicht auf den 64 GiB Knoten eingeplant werden darf.

Vergleichbare Diagramme lassen sich auch für die CPU-Auslastung erstellen, welche geringer ist als die Auslastung des Arbeitsspeichers. Dargestellt ist dies in Abbildung 3.5. Gleichzeitig sind die aufsummierten CPU-Limits im Verhältnis zu der Anzahl verfügbarer Kerne noch höher. Insgesamt zeigt sich eine starke Diskrepanz zwischen der Spezifikation des CPU-Bedarfs an den Containern und der tatsächlichen Nutzung. Möglicherweise trägt das in Abschnitt 1.4 diskutierte Verhalten der CFS-Bandwidth-Control dazu bei.

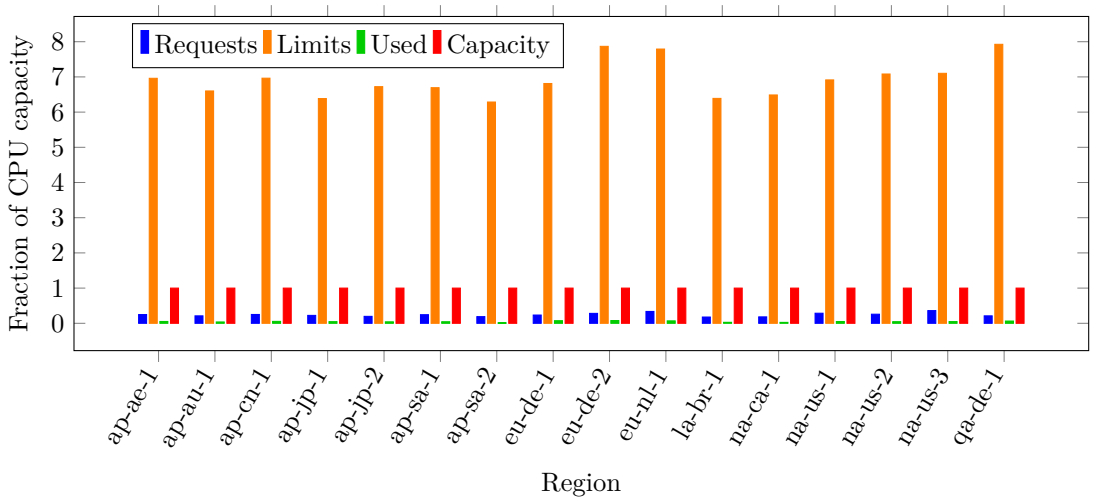

Abbildung 3.5: Durchschnittlicher CPU-Bedarf für die Scaleout-Cluster am 13. März

Die Abweichung ist auch auf der Ebene einzelner Knoten zu sehen, welche in Abbildung 3.6 abgetragen ist. Die Mehrheit der Knoten hat eine CPU-Auslastung von unter 20%. Weiterhin ist kein Zusammenhang zwischen den CPU-Requests und der durchschnittlichen Auslastung zu erkennen. Knoten mit mehr Kerne zeigen eine geringere Auslastung als Knoten mit wenig Kernen.

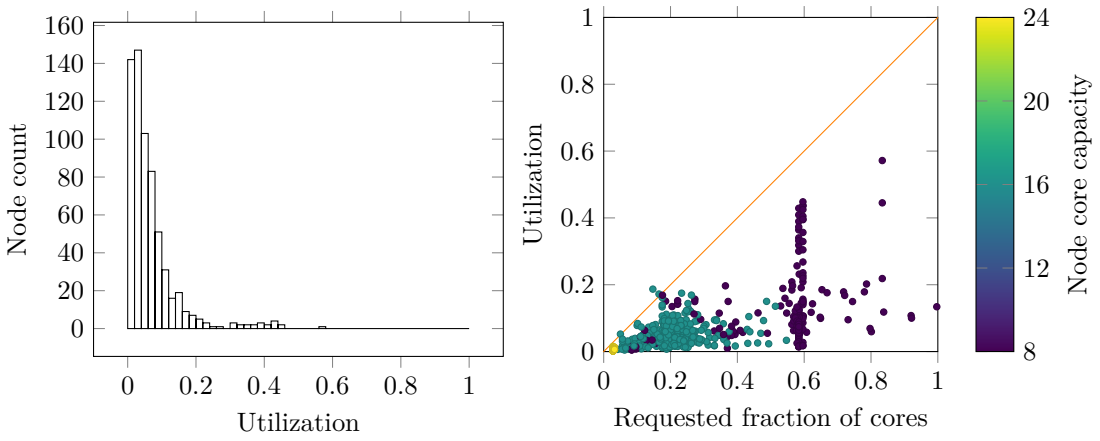

Abbildung 3.6: Histogramm und Streudiagramm der durchschnittlichen Auslastung der CPUs für alle Knoten der Scaleout-Cluster am 13. März

Anzumerken ist, dass alle Darstellungen zur Auslastung den Tagesdurchschnitt basierend auf Stichproben, welche alle 15 Minuten erfasst wurden, nutzen. Starke Beanspruchungen der Ressourcen von kurzer Dauer finden daher lediglich einen geringen Einfluss. Eben genanntes Beanspruchungsmuster trifft aber auf manche in den Clustern enthaltenen Anwendungen zu. Diese Anwendungen bestehen aus Webservern und Datenbanken, die weitgehend nur agieren, wenn eine Anfrage eintrifft. Insofern kann eine Abweichung zwischen der tatsächlichen CPU-Nutzung und der statisch spezifizierten in Form von Requests und Limits vertretbar sein.

3.3 Experteninterviews

Die in Abschnitt 3.2 durchgeführte Analyse zeigt eine schlechte Auslastung der Cluster auf. Die Summe der Requests aller Container eines Clusters, welche der Kube-Scheduler berücksichtigt, ist deutlicher kleiner als die Kapazität, die in dem jeweiligen Cluster bereitsteht. Dennoch ist diese Arbeit durch Schwierigkeiten in den Bereichen der Ressourcenverwaltung und des Schedulings motiviert. Um diese zu erfassen, wurden semi-strukturierte Interviews durchgeführt. Dabei wurden die drei Administratoren für die Metal- und Scaleout-Cluster und zwei verantwortliche Personen für Applikationen in diesen Clustern befragt. Die erste Person ist dabei für den Betrieb der Prometheus-Instanzen zuständig. Die Zweite für den Betrieb und die Entwicklung von Neutron, der OpenStack-Komponente für virtualisierte Netzwerke. Alle Befragten sind seit mehreren Jahren bei Converged Cloud angestellt und arbeiten entsprechend lange mit Kubernetes und den betreuten Applikationen. Die Fragen und der Aufbau des Interviews wurden vor der eigentlichen Befragung mit einer weiteren Person von Converged Cloud validiert und sind in Anlage A.1 aufgeführt.

Den verschiedenen Gesprächen sind mehrere Erkenntnisse gemeinsam. Zunächst haben alle Teilnehmer bereits Störungen erlebt, die durch suboptimales Scheduling zumindest verlängert wurden. Dafür konnten zwei Ursachen identifiziert werden. Eine Ursache sind gelegentliche Probleme mit Noisy-Neighbours, welche durch die Lücken in der Isolation der Ressourcen begründet sein können. Alternativ können Pods mit unzureichenden Requests bei gleichzeitig großen Limits einem stark ausgelasteten Knoten zugeordnet werden. Die Container des Pods bekommen allerdings, aufgrund zu kleiner Requests, keine ausreichenden Ressourcen mehr.

Die zweite Ursache ist, dass Pods mit einem hohen Bedarf nach Ressourcen oft bei der Durchführung von Wartungsarbeiten nicht mehr planbar sind. Bei Wartungsarbeiten werden nacheinander einzelne Knoten aus dem Cluster entfernt und später wieder hinzugefügt. Wenn ein Pod mit großen Requests bei mindestens einem Ressourcentyp entfernt wird, ist es möglich, dass dessen Ersatz mit identischen Requests nicht mehr einplanbar ist. Dies tritt bei verschiedenen Workloads, wie großen Prometheus-Instanzen, Neutron-Agenten und MariaDBs für OpenStack, auf. Das Entfernen der Pods gibt Ressourcen frei. Diese werden im Wartungsfall allerdings nicht dem Cluster zurückgeben, da der Knoten aus dem Cluster entfernt wird. Trotz insgesamt ausreichend freier Kapazität im Cluster verbleiben Pods mit großen Requests als nicht planbar, da kein anderer Knoten allein die erforderlichen freien Ressourcen besitzt. Der Grund dafür ist, die in Abschnitt 1.3 angemerkte Standardkonfiguration des Kube-Schedulers Knoten möglichst gleichmäßig auszulasten.

Ebenso wird mehr Automatisierung in der Bewertung der Pods bezüglich ihrer Requests und Limits gewünscht. Dies ist damit zu begründen, dass verschiedene Cloud-Regionen unterschiedlich stark verwendet werden und die Applikationen selbst in kleine Dienste zerlegt sind. Eine Konfiguration von Hand ist entsprechend aufwen-

dig. Die dynamische Anpassung zur Laufzeit verspricht zusätzlich eine Verbesserung der Auslastung der Knoten, da zu große Ressourcenspezifikationen automatisch angepasst werden. Kubernetes kann jedoch vor Version 1.27 und ohne Aktivierung des `InPlaceVerticalPodScaling`-Features Requests und Limits nicht ändern, ohne den Pod neuzustarten. Daher gibt es Sorgen bezüglich der kontinuierlichen Erbringung der IT-Dienstleistungen, aufgrund häufiger Neustarts.

Das in 1.4 beschriebene Verhalten der Quotaverwaltung des CFS war eher unbekannt. Falls doch mit dem wagen Stand, dass sich die Spezifikation CPU-Limits prinzipiell nicht lohne, da dies das Throttling bedingt. Dies erklärt partiell die in Abschnitt 3.2 dargestellte schlechte Verwaltung der CPU-Zeit. Bezüglich zu planender Ressourcen wurde Interesse an CPU-Zeit, Arbeitsspeicher, Netzwerkbandbreite und I/O-Geschwindigkeit geäußert.

4 Anforderungsanalyse

Ein wirtschaftlicher Grund für den Einsatz von Virtualisierungstechnologien sind Kostenersparnisse, welche durch simplere Betriebsprozesse und bessere Auslastung der Hardware erreicht werden sollen [2, S. 10–11]. Dies gilt auch für die Motivation dieser Arbeit, weshalb die Auslastung der Knoten zu erhöhen ist. Für die Scaleout-Cluster ist eine ausführliche Analyse diesbezüglich in Abschnitt 3.2 vorgenommen worden. In den Metal-Clustern ist die Situation ähnlich.

Die Auslastungsoptimierung unterliegt der Nebenbedingung, dass sich die Qualität der angebotenen Dienstleistungen nicht verschlechtern darf. Daraus resultieren vor allem folgende drei Einschränkungen:

- Die Hardware lässt sich nicht vollständig auslasten, ohne an Reaktionszeit einzubüßen. Desto voller der Arbeitsspeicher der Knoten wird, desto mehr Zeit muss in die Verwaltung der Pages investiert werden. Ebenso gibt es Grenzen bei der CPU-Auslastung. Diese Grenzen sind zu ermitteln.

- Für die Aktualisierung des Betriebssystems, der Hypervisors oder von Kubernetes selbst müssen Knoten aus den Clustern entfernt werden. Für diesen Bedarf ist Kapazität vorzuhalten.

- Die Isolation zwischen Pods sollte möglichst stark sein. Dieses Thema wurde in Abschnitt 1.2 bezüglich der QoS-Klassen diskutiert. Im Rahmen der Optimierung werden ausschließlich Pods in der Guaranteed-QoS-Klasse, also mit identischen Requests und Limits, betrachtet. Lösungsansätze für das CPU-Throttling sind zu identifizieren.

Um Pods geeigneten Knoten zuzuordnen, ist es erforderlich festzulegen, welche Ressourcen zu planen und welche weiteren Einschränkungen zu berücksichtigen sind. Diese Informationen werden mittels einer Umfrage für die Converged Cloud Organisationseinheit erfasst. Der Umfrage liegt die Kano-Methode zugrunde, welche auf Kanos-Theorie zur Kundenzufriedenheit basiert. Dabei werden für jedes zu untersuchende Attribut zwei Fragen gestellt. Die funktionale Frage erfasst, wie Anwender die Erfüllung eines Attributs wahrnehmen, wohingegen die dysfunktionale Frage die Reaktion erfasst, wenn das Attribut fehlt. Die Antwortmöglichkeiten reichen von „unakzeptabel“ über „gerade so akzeptierbar“, „egal“, „erwartet“ bis zu „beeindruckend“ [11, S. 111–112].

Aus den Experteninterviews (Abschnitt 3.3) ergeben sich vier Typen an Ressourcen, an denen Interesse bezüglich der Berücksichtigung beim Scheduling besteht. Diese sind CPU-Zeit, Arbeitsspeicher, Netzwerkbandbreite und I/O-Geschwindigkeit[1]. Wie in Abschnitt 1.2 ausgeführt, besitzt Kubernetes optionale Mechanismen, welche die Scoring-Phase des Kube-Schedulers beeinflussen. Das Ziel ist, die Auslastung zu maximieren. Insofern ist die Ermittlung des Bedarfs für optionale NodeAffinities, InterPodAffinities und Lokalität von Containerabbildern relevant, da diese die Bewertung von Knoten bezüglich optimaler Auslastung verzerren. Möglicherweise kann auf diese verzichtet werden. Zuletzt wurde noch ermittelt, ob im Rahmen des Scheduling bereits platzierte Pods verschoben werden dürfen.

Die Umfrage wurde online durchgeführt und ist vor der Veröffentlichung von zwei Personen getestet worden. Dann wurde die Umfrage in einem internen Kommunikationskanal für Mitarbeiter der Converged Cloud Organisationseinheit bereitgestellt. Dieser hat 233 Mitglieder, wobei anzumerken ist, dass einige der erreichten Personen nicht mit Kubernetes arbeiten. Eine weitere Auswahl der Teilnehmer wurde nicht vorgenommen. Elf Personen haben Antworten eingereicht. Respektiv beträgt die Antwortrate damit rund 4.7%. Die Auswertung ist folglich nur begrenzt repräsentativ. Die Fragen und Antworten sind in Anhang A.3 angegeben.

Das KANO-Modell unterscheidet fünf Arten von Anforderungen [11, S. 82–83]:

- Basisanforderungen (**M**ust-be-Quality) werden vom Nutzer vorausgesetzt. Eine Erfüllung vermeidet lediglich Unzufriedenheit.

- Leistungsanforderungen (**O**ne-dimensional-Quality) besitzen einen monoton steigenden Zusammenhang zwischen dem Erfüllungsgrad und der Zufriedenheit des Nutzers.

- Begeisterungsanforderungen (**A**ttractive-Quality) erzeugen eine hohe Zufriedenheit bei Erfüllung. Ein Fehlen wirkt sich nicht auf die Zufriedenheit aus.

- Indifferente Anforderungen (**I**ndifferent-Quality) haben keinen Einfluss auf die Zufriedenheit unabhängig des Erfüllungsgrads.

- Entgegengesetzte Anforderung (**R**everse-Quality) besitzen einen monoton fallenden Zusammenhang zwischen dem Erfüllungsgrad und der Zufriedenheit des Nutzers.

Die ersten drei Typen lassen sich, wie in Abbildung 4.1 dargestellt, skizzieren. Indifferente Anforderungen würden auf der Abszisse liegen.

[1] Kubernetes unterstützt direkt das Planen von Festplattenkapazität. Diese Ressource ist in Converged Cloud allerdings nicht knapp.

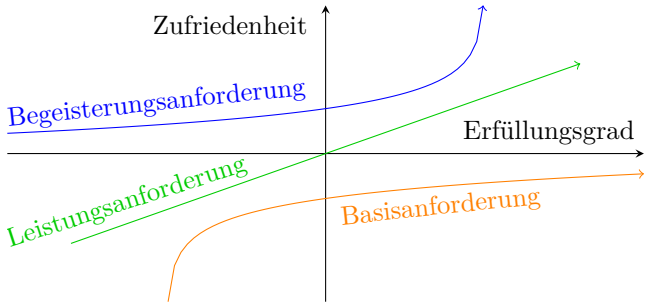

Abbildung 4.1: Typen von Anforderungen im KANO-Modell [11, S. 87]

Basierend auf den Antworten bezüglich der Fragenpaare lassen sich anhand von Tabelle 4.1 die verschiedenen Kategorien je Anforderung und Teilnehmer klassifizieren. Die Kürzel beziehen auf die vorherige Auflistung der Anforderungskategorien. **Q** markiert widersprüchliche Antworten.

Dysfunktional / Funktional	Beeindruckend	Erwartet	Egal	Gerade so akzeptierbar	Unakzeptabel
Beeindruckend	Q	A	A	A	O
Erwartet	R	I	I	I	M
Egal	R	I	I	I	M
Gerade so akzeptierbar	R	I	I	I	M
Unakzeptabel	R	R	R	R	Q

Tabelle 4.1: Auswertung KANO-Umfrage [11, S. 116]

Pro Anforderung werden die Kategorien zusammengezählt und dann die häufigste Kategorie für die Endauswertung benutzt [11, S. 121–122]. Bei Gleichstand wurde nach der Ordnung $M > O > A > I$ entschieden. Diese Rangfolge ist darauf ausgelegt Unzufriedenheit zu vermeiden [11, S. 123]. Damit ergibt sich die in Tabelle 4.2 angegebene Zuordnung.

Anforderung	**Kategorie**
Der Scheduler plant CPU-Zeit.	A
Der Scheduler plant Arbeitsspeicherauslastung.	A
Der Scheduler plant Netzwerkbandbreite.	A
Der Scheduler plant I/O-Geschwindigkeit.	A
Der Scheduler plant bei Bedarf bereits laufende Pods um.	A
Der Scheduler bevorzugt Knoten, auf denen das Containerabbild bereitliegt.	I
Der Scheduler berücksichtigt optionale Topology-Spread-Constraints.	M
Der Scheduler berücksichtigt optionale (Anti-)Affinitäten bezüglich der Knoten.	I

Tabelle 4.2: Kategorisierung von Anforderungen an den Scheduler

Die ersten vier Anforderungen beziehen sich auf Ressourcen, welche der Scheduler berücksichtigen kann. Diese werden als Begeisterungsanforderungen eingeordnet. Das Fehlen der Ressourcenplanung beeinflusst die Nutzererfahrung also weder positiv noch negativ, während die Existenz zu einer besseren Erfahrung beiträgt. Möglicherweise kann dies damit erklärt werden, dass die Berücksichtigung von Ressourcen lediglich negative Effekte vermeidet und Kubernetes als Gesamtsystem auch ohne jene nutzbar ist. Zusätzlich ist ersichtlich, dass Interesse an der Planung weiterer Ressourcen, neben der CPU-Zeit, dem Arbeitsspeicher und der Festplattenkapazität, welche Kubernetes bereits einbezieht, besteht. Ebenso wird das erneute Einplanen von bereits ausgeführten Pods akzeptiert. Dies ist bemerkenswert, da in den Experteninterviews (Abschnitt 3.3) die Sorge vor Diskontinuitäten bei automatischer Skalierung thematisiert wurde.

Die letzten drei Anforderungen beziehen sich auf optionale Nebenbedingungen beim Scheduling. Optionale Topology-Spread-Constraints werden erwartet. Dies ist nachvollziehbar, da mit diesen Pods über mehrere virtuelle oder physische Knoten bis hin zu verschiedenen Rechenzentren verteilt werden können. Das ist aus Sicht der Verfügbarkeit wünschenswert. Auf die Berücksichtigung der älteren (Anti-)Affinitäten für Knoten kann verzichtet werden, da sie nicht zur Nutzererfahrung beitragen. Ebenfalls kann die Lokalität von Containerabbildern vernachlässigt werden. Kleine Abbilder werden zureichend schnell heruntergeladen und bei kritischen Applikationen sind bereits Mechanismen im Einsatz, die bestimmte Abbilder auf allen Knoten eines Clusters vorhalten.

5 Artefaktentwicklung

5.1 Vector-Bin-Packing-Heuristiken

In Abschnitt 1.3 wurde das Vector-Bin-Packing-Problem vorgestellt, welches die Zuordnung von Pods zu Knoten beschreibt. Ebenso wurde die von Kubernetes eingesetzte Heuristik angegeben. Es bietet sich an verschiedene Heuristiken zur Maximierung der Auslastung zu vergleichen, da

- dem Autor keine wissenschaftliche Quelle zur Qualität der von Kubernetes genutzten Heuristik vorliegt und
- weitere in der Literatur vorgeschlagene Heuristiken möglicherweise bessere Zuordnungen finden.

Der Vergleich wird mittels einer Monte-Carlo-Simulation durchgeführt. Dabei wird das in Abschnitt 1.3 beschriebene Problem gelöst. Die Eingabe ist folglich eine Menge von Vektoren X, also eine Liste an Pods mit spezifizierten Ressourcen, und die Ausgabe die Partitionierung L, wobei die Anzahl der ausgegebenen Knoten erfasst wird.

Die einfachste Heuristik ist gewöhnliches First-Fit. Für jedes zu planende $\vec{X_i}$ wird jeder nicht leere Knoten in der Reihenfolge der Erstellung berücksichtigt. Das aktuell zu planende $\vec{X_i}$ wird dem ersten Knoten, dessen Kapazität nicht überschritten wird, zugeordnet. Kann die Zuordnung nicht erfolgen, wird ein neuer Knoten erstellt [19, S. 405].

Die Auswahl der weiteren implementierten Heuristiken basiert auf einer Literaturrecherche. Suchbegriffe dafür waren „multi capacity bin packing“, „vector bin packing“, „container placement“ und „virtual machine placement“. Letzterer Begriff bietet sich an, da das Problem der Zuordnung von virtuellen Maschinen auf physische auch als Vector-Bin-Packing aufgefasst werden kann [25, S. 1]. Algorithmen, welche zwingend eine offline Variante des Vector-Bin-Packings erfordern oder nicht ausschließlich bezüglich der Auslastung optimieren, wurden nicht betrachtet. Der von Mishra und Sahoo [22, S. 279–282] vorgeschlagene Algorithmus „Planar Resource Hexagon“ wurde aufgrund des hohen Aufwands der Implementierung nicht umgesetzt. Viele der vorgeschlagenen Ansätze setzen Best-Fit mit verschiedenen Kriterien zur Sortierung der Knoten ein. Anstatt die Knoten in der Reihenfolge der Erstellung bezüglich der

Zuordnung eines $\vec{X}_i$ zu überprüfen, wird die Liste der bisherigen Knoten anhand einer Heuristik sortiert. Bezüglich der Optimierung der Laufzeit ist das Sortieren nicht erforderlich. Es ist zureichend den Knoten mit geeigneter Kapazität und dem besten Wert der Heuristik per einfacher Schleife zu ermitteln. Die Heuristiken streben oft an, den einzelnen Knoten gleichmäßig bezüglich seiner Ressourcentypen auszulasten, sodass ein Erreichen der Kapazität in einem Typ nicht zu schlechter Auslastung in anderen Ressourcentypen führt. Das Verfahren ist in Abbildung 5.1 skizziert. `pods` sei eine Liste, deren Elemente einzeln und unabhängig in der Schleife eingeplant werden.

```
def best_fit(pods):
    nodes = []
    for pod in pods:
        indices = range(len(nodes))
        # Indecies von Knoten, die Kapazität für den Pod besitzen
        fitting = [i for i in indices if fits(pod, nodes[i])]
        if len(fitting) == 0:
            # Kein Knoten hat Kapazität => neuer Knoten
            nodes.append([pod])
            continue
        # Auswahl des "besten" Knoten anhand der Heuristik
        best = max(fitting, key=lambda i: score(pod, nodes[i]))
        # Pod platzieren
        nodes[best].append(pod)
    return nodes
```

Abbildung 5.1: Skizzierung Best-Fit

Leinberger, Karypis und Kumar [19] schlagen mehrere Algorithmen vor, deren Grundgedanke für das online Vector-Bin-Packing genutzt werden kann. Permutation-Pack versucht einem $\vec{X}_i$ für dessen Komponenten $X_{i1} \geq X_{i2} \geq \cdots \geq X_{i(d-1)} \geq X_{id}$ gilt einen Knoten $\vec{B}_k$ mit entgegengesetzter Auslastung $B_{k1} \leq B_{k2} \leq \cdots \leq B_{k(d-1)} \leq B_{kd}$ zuzuordnen, um die Auslastung des Knotens zu balancieren. Wenn dies nicht gelingt, wird die Reihenfolge der Komponenten in $\vec{X}_i$ permutiert, wobei bei den kleinen Komponenten begonnen wird. Im Rahmen der Erläuterung, die gegeben wird, ist dafür die lexikografische Permutation geeignet. Die nächste Permutation bezüglich Knoten ist dann $B_{k1} \leq B_{k2} \leq \cdots \leq B_{kd} \leq B_{k(d-1)}$. Erfüllen mehrere Knoten die beschriebene Bedingung ordnet die originale Implementierung das $\vec{X}_i$ diesen Knoten per First-Fit zu [19, S. 406]. In dieser Arbeit wird der Knoten mit der höchsten Auslastung bevorzugt. Die originale Formulierung ist für die offline Variante des Problems vorgesehen, bei der die Menge X vor dem Platzieren nach dem Ressourcenbedarf sortiert werden kann. Die Knoten nach Auslastung zu sortieren, führt zu einem besseren Ergebnis. Hat kein Knoten die Kapazität für das aktuelle $\vec{X}_i$ wird ein neuer Knoten angelegt.

Die Anzahl der Permutationen beträgt $d!$. Wird d ausreichend groß, dominiert die Partitionierung die Laufzeit des Algorithmus. Um diese zu begrenzen, wird die Window Size w eingeführt. Dann werden nur die größten w Komponenten bezüglich ihrer

Permutation berücksichtigt. Bei $w = 2$ zum Beispiel $X_{i1} \geq X_{i2} \geq \ldots, X_{i(d-1)}, X_{id}$ Die Anzahl der Permutationen ändert sich zu $(d - w)!$ [19, S. 406].

Choose-Pack lockert die Anforderungen an die Reihenfolge der Komponenten weiter, sodass die ersten w Komponenten keine Ordnungsrelation mehr erfüllen müssen. Exemplarisch für $w = 2$ also $X_{i1}, X_{i2} \geq \ldots, X_{i(d-1)}, X_{id}$ [19, S. 406]. Choose-Pack ist für unsortierte Eingaben ähnlich gut wie Permutation-Pack [19, S. 408]. Letzteres war simpler in der Implementierung, weshalb Choose-Pack nicht implementiert wurde.

Mishra und Sahoo [22, S. 276–278] diskutieren mehrere Ansätze zur Platzierung von virtuellen Maschinen. Bis auf das Vector-Dot-Verfahren, wird an diesen Ansätzen die Vernachlässigung der Mehrdimensionalität der Problematik kritisiert, was zu einer suboptimalen Platzierung führe. Deshalb wurde nur die Vector-Dot-Heuristik für das Best-Fit implementiert. Wie der Name bereits andeutet, werden die Knoten in aufsteigender Reihenfolge des Skalarprodukts $\vec{U_k} \cdot \vec{X_i}$ sortiert. Dabei stellt $\vec{U_k} = (\sum_{\vec{X_k} \in B_k} \vec{X_k}) \oslash \vec{C}$ die aktuelle Auslastung eines Knotens dar[1]. Daran wird die fehlende Normalisierung der Ressourcen kritisiert. Als Verbesserung wird die Minimierung des Winkels zwischen $\vec{1} - \vec{U_k}$ und $\vec{X_i}$ vorgeschlagen [22, S. 278]. Die Minimierung dieses Winkels wurde für die Simulation implementiert.

Die Simulation umfasst als weiteren Parameter die Anzahl der Ressourcendimensionen. Minimal sind die CPU-Zeit und der benötigte Arbeitsspeicher zu berücksichtigen. Netzwerkbandbreite und Ressourcen bezüglich persistentem Speicher sind für Converged Cloud optional. Je nach konkretem Anwendungsfall können weitere erweiterte Ressourcen interessant sein, wie in Abschnitt 4 ausgeführt. Daher wurde die Simulation mit zwei, vier und acht Dimensionen durchgeführt.

Die Simulation wurde in der Programmiersprache `Go` umgesetzt. Die Eingabemengen an Pods werden zufällig mittels verschiedener Verteilungen erzeugt. Jede Dimension bekommt dafür einen eigenen Pseudo-Random-Number-Generator (PRNG) zugeordnet, um eventuelle Abhängigkeiten zwischen Dimensionen zu vermeiden. Die Implementierung des PRNG in der `Go` Standardbibliothek verfügt über keine ausführliche wissenschaftliche Untersuchung und basiert auf dem Linear-Feedback-Shift-Register-Verfahren für das bereits statistische Schwächen bekannt sind [23, S. 10, 23]. Aufgrund dessen wurde eine Implementierung des Verfahrens von O'Neill [23] zur Erzeugung der Zufallszahlen genutzt, welche auch das alte Verfahren in der Standardbibliothek ersetzen soll.

Die optimale Anzahl an Knoten ist aufgrund der NP-Schwere des Problems nicht bekannt. Daher können die verschiedenen Heuristiken lediglich untereinander verglichen werden. Die Vergleichslinie bildet die durchschnittliche Anzahl der Knoten nach First-Fit. Es ist zu erwarten, dass First-Fit im mehrdimensionalen Fall schlecht packt, da Knoten in lediglich einer Dimension gut ausgelastet werden, da keine Maßnahmen für eine ausgeglichene Auslastung der Dimensionen innerhalb eines Knotens

[1] Durch $\oslash$ sei die Hadamard (elementweise) Division ausgedrückt.

angewendet werden. Das Verhältnis zwischen der durchschnittlichen Anzahl an Knoten des untersuchten Verfahrens und jener Größe nach First-Fit wird an der Ordinate abgetragen. Die Auswertung von Leinberger, Karypis und Kumar [19, S. 409] zeigt, dass die Betrachtung für verschiedene durchschnittliche Gewichtungen der $\vec{X}_i$ interessant ist. Der durchschnittliche Bedarf ist an der Abszisse notiert. Aufgrund eines spezifischen Verfahrens zur Generierung der $\vec{X}_i$ werden als durchschnittliche Gewichte $\frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{10}, \frac{1}{12}, \frac{1}{16}, \frac{1}{20}, \frac{1}{25}$ verwendet. Für jedes Verfahren zur Generierung von Pods und deren durchschnittliche Gewichtung werden je 1500 Listen an Pods erzeugt. Diese werden jeweils von allen Heuristiken für die Simulation genutzt. Zusammengefasst werden folgende Heuristiken simuliert:

- First-Fit (FF)
- Kubernetes-Least (KL): Standardkonfiguration des Kubernetes Schedulers, welche wenig ausgelastete Knoten präferiert (Abschnitt 1.3).
- Kubernetes-Most (KM): Konfiguration des Kubernetes Schedulers, welche stark ausgelastete Knoten präferiert (Abschnitt 1.3).
- Permutation-Pack (PP): Zuordnung eines Pods zu Knoten mit möglichst entgegengesetzter Reihenfolge bezüglicher der Auslastung einzelner Ressourcen.
- Vector-Dot (VD): Minimierung des Winkels zwischen $\vec{1} - \vec{U}_k$ und $\vec{X}_i$.

Weiterhin müssen die Eingabevektoren erzeugt werden. Trivial ist die zufällige Erzeugung der Komponenten der $\vec{X}_i$ anhand unabhängiger Gleichverteilungen. Die Ergebnisse der Simulation mit gleichverteilten Eingabevektoren sind in Abbildung 5.2 abgebildet. Permutation-Pack wird ab $d \geq 5$ spürbar langsam, weshalb im achtdimensionalen Fall $w = 4$ genutzt wird. Die genaue durchschnittliche Anzahl an erforderlichen Knoten ist in Anhang A.4 gegeben.

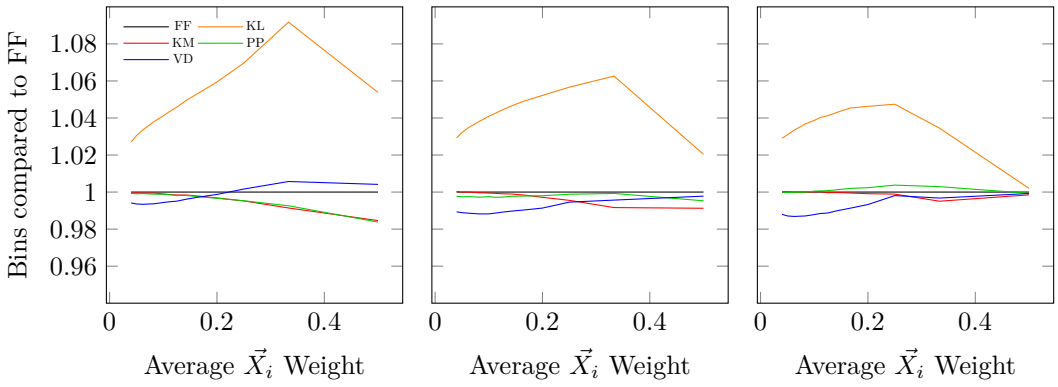

Abbildung 5.2: Verschiedene Bin-Packing-Heuristiken gegenüber First-Fit bei gleichverteiltem Bedarf in 2, 4 und 8 zu planenden Dimensionen

Kubernetes-Most (KM) und Permutation-Pack (PP) benötigen die wenigsten Knoten im zwei- und vierdimensionalen Fall bei größerem durchschnittlichem Ressourcenbedarf. In allen Dimensionen ist Vector-Dot (VD) die Heuristik, welche am wenigsten Knoten bei kleinerem Bedarf benötigt.

Um die Anzahl der benötigten Knoten der Verfahren mit der optimalen Lösung zu vergleichen, ist folgendes Vorgehen umgesetzt worden. Zunächst wird die Anzahl der Knoten der optimalen Lösung festgelegt. Im Rahmen der Simulation werden 100 Knoten verwendet. In Bezug auf den Untersuchungsgegenstand ist das eine geeignete Größe, wie Tabelle 3.1 aufgezeigt. Die Knoten lassen sich dann in eine bestimmte Anzahl $\vec{X}_i$ anhand unabhängiger Gleichverteilungen pro Dimension aufteilen. Da jeder Knoten nur ganzzahlig in Pods geteilt werden kann, begründet dies die oben genannte Einschränkung der durchschnittlichen Gewichte bezüglich $\frac{1}{a}, a \in \mathbb{N}^+$. Die entstehenden $\vec{X}_i$ werden dann gemischt. Durch das Verfahren entstehen $100a$ zu planende Pods. Sowohl das Verfahren, welches sich direkt der Gleichverteilung bedient, als auch das, welches die Exponentialverteilung nutzt, sind derart gestaltet, dass der durchschnittliche Bedarf $1/a$ und die Anzahl der Pods $100a$ entspricht. Die Ergebnisse für das Aufteilen von 100 Knoten sind in Abbildung 5.3 dargestellt. Die Rangfolge der Heuristiken ist vergleichbar zu jener bei direkter Anwendung der Gleichverteilung. Allerdings sind die Heuristiken abseits von Kubernetes-Least nochmals effizienter gegenüber First-Fit. Dies kann damit begründet werden, dass die Heuristiken eine ausgeglichene Auslastung anstreben, die in den generierten Pods, aufgrund der Aufteilung von Knoten, tatsächlich gegeben ist.

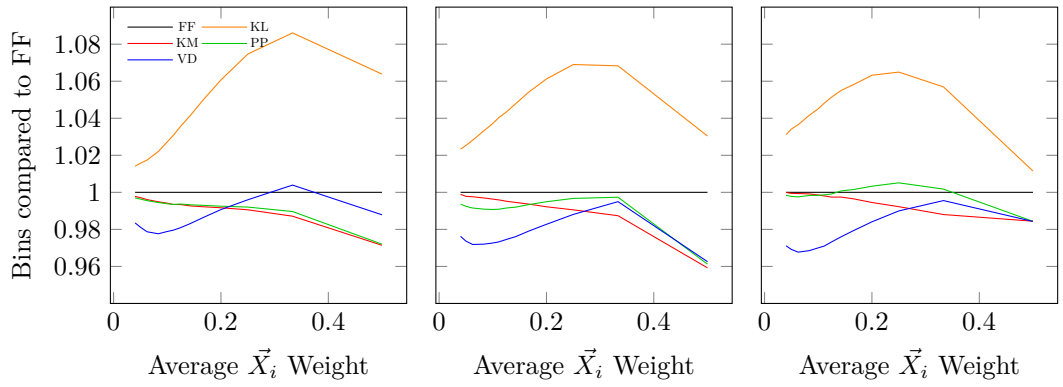

Abbildung 5.3: Verschiedene Bin-Packing-Heuristiken gegenüber First-Fit nach Aufteilung von 100 Knoten in 2, 4 und 8 zu planenden Dimensionen

Leinberger, Karypis und Kumar [19, S. 408] nutzen für die Komponenten der $\vec{X}_i$ zufällige Werte, die exponentialverteilt sind. Dadurch werden wenige Pods mit großem Ressourcenbedarf erzeugt und viele Pods mit kleinem Bedarf. Die Simulationsergebnisse sind in Abbildung 5.4 visualisiert. Komponenten des Bedarfs, die 1 übersteigen, was aufgrund der zugrundeliegenden Exponentialverteilung vorkommt, werden auf 1 limitiert. Im vier- und achtdimensionalen Fall sind alle Heuristiken abseits von Kubernetes-Least (KL) bei größerem durchschnittlichem Bedarf etwa gleich gut.

Unabhängig der Dimensionalität benötigt Vector-Dot (VD) nochmals die wenigsten Knoten bei kleinem durchschnittlichem Bedarf.

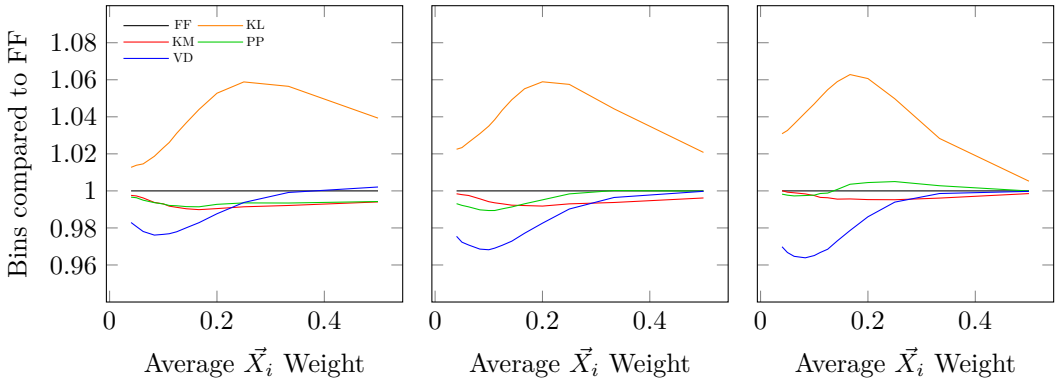

Abbildung 5.4: Verschiedene Bin-Packing-Heuristiken gegenüber First-Fit bei exponentialverteiltem Bedarf in 2, 4 und 8 zu planenden Dimensionen

Die vorkonfigurierte Heuristik im Kube-Scheduler ist, wie in Abschnitt 1.3 ausgeführt, Kubernetes-Least (KL). Jede andere Heuristik benötigt in allen untersuchten Fällen weniger Knoten. Kubernetes-Most (KM) kann bereits durch Änderung der Konfiguration des Kube-Schedulers eingesetzt werden. Je nach Verteilung des Bedarfs und dessen Durchschnitt werden $2-7\%$ weniger Knoten gegenüber Kubernetes-Least (KL) benötigt. Bei einem durchschnittlichen Bedarf kleiner als 0.2 kann sich die Implementierung von Vector-Dot (VD) lohnen, wobei dafür der Kube-Scheduler zu erweitern ist. Kubernetes-Most (KM) ist zwar nicht die in allen untersuchten Kombinationen beste Heuristik, allerdings ist es nie schlechter als First-Fit (FF).

In allen untersuchten Fällen ist Vector-Dot (VD) bei vielen Pods mit durchschnittlich kleinem Bedarf deutlich die beste Heuristik. Vector-Dot (VD) priorisiert lediglich nach ausgeglichener Auslastung zwischen den Ressourcentypen auf einem Knoten und ignoriert die bisherige Auslastung des Knotens. Dies ist ein Indikator dafür, dass bei kleinem Bedarf die ausgeglichene Auslastung innerhalb eines Knotens relevanter ist, als einen Pod auf einem möglichst gut ausgelasteten Knoten zu platzieren. Die Heuristik des Kube-Schedulers besteht aus eben diesen zwei Komponenten. Deren Gewichtung kann angepasst werden.

Abbildung 5.5 betrachtet dies für das Verfahren der gleichverteilten Aufteilung von Knoten. Die Referenz bildet erneut First-Fit (FF). Kubernetes-Most (KM) ist bezüglich der Vergleichbarkeit ebenfalls abgebildet. Kubernetes-Reweighted (KR) sei beschrieben durch $S1_j + 2 * S2_j$ (siehe Abschnitt 1.3). Kubernetes-Vector-Dot (KVD) beschreibe die Kombination aus der Kubernetes-Auslastung-Heuristik und dem Vector-Dot-Verfahren mit $S1_j + 2 * \sphericalangle(\vec{1} - \vec{U_j}, \vec{X_i})$.

Eine stärkere Gewichtung der balancierten Auslastung der Ressourcendimensionen innerhalb eines Knotens lohnt sich. Kubernetes-Reweighted ist mindestens so gut wie Kubernetes-Most und bei manchem durchschnittlichen Bedarf besser. Die Kom-

bination von Heuristiken in Kubernetes-Vector-Dot (KVD) gleicht die Schwäche von Vector-Dot (VD) bei großem durchschnittlichem Bedarf aus und führt bei kleinem Bedarf weiterhin zu einer besseren Zuordnung gegenüber den anderen Heuristiken, abseits von der reinen Vector-Dot-Heuristik. Der Autor schätzt die Implementierung des Vector-Dot-Verfahrens als Plugin im Kontext des Scheduling-Frameworks als umsetzbar ein.

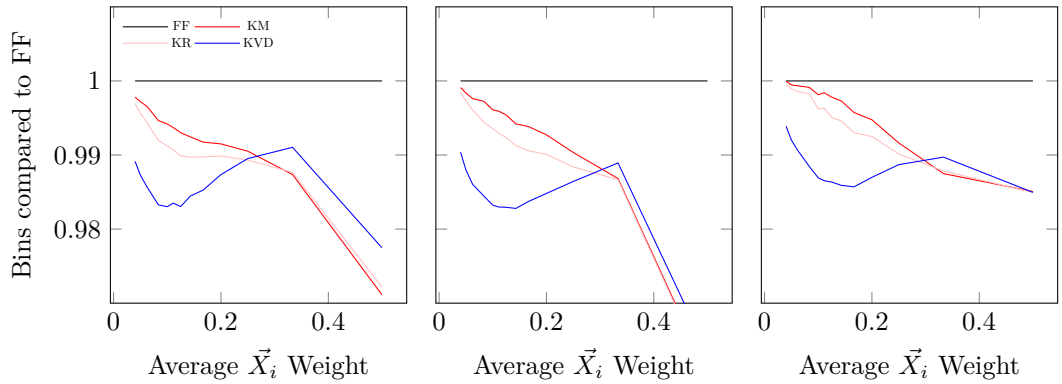

Abbildung 5.5: Gewichtete Bin-Packing-Heuristiken gegenüber First-Fit nach Aufteilung von 100 Knoten in 2, 4 und 8 zu planenden Dimensionen

Zur Einordnung der Ergebnisse sei angemerkt, dass die Anzahl der Knoten ohne Cluster-Autoscaling in einem Cluster vorgegeben ist. Daher ist es sinnvoll, eine Heuristik zu verwenden, welche eine Komponente besitzt, die Knoten präferiert auf denen bereits Pods platziert sind. Sonst werden die Pods möglicherweise recht gleichmäßig auf die Knoten verteilt. In realen Clustern kann es weiterhin vorkommen, dass die Knoten verschiedene Kapazitäten aufweisen, wie auch in der Analyse in Abschnitt 3.2 ersichtlich.

Eine weitere Verbesserung der diskutierten Heuristiken lässt sich potenziell durch das Verfolgen der eintreffenden Ressourcenspezifikationen erzielen. Leinberger, Karypis und Kumar [19, S. 410] stellen fest, dass die Verfahren Permutation-Pack und Choose-Pack für leichtgewichtige $\vec{X}_i$ mit großen w und für schwergewichtige $\vec{X}_i$ mit kleinen w besser funktionieren. Das vorgeschlagene Verfahren Adaptive-Pack verfolgt die durchschnittlichen Komponentengewichte und passt w entsprechend an. Tantawi und Steinder [30, S. 189–190] schlagen ein Verfahren vor, welches die eintreffenden Ressourcenspezifikationen nachverfolgt und versucht die Knoten mit einer ähnlichen Verteilung auszulasten. Dies führt dazu, dass für Komponenten mit großer Varianz Bin-Packing durchgeführt wird, für andere nicht. Der Ansatz ist aus der Perspektive der Dimensionsreduktion interessant.

In der Simulation wird unabhängig vom durchschnittlichen Bedarf pro Pod ein identischer durchschnittlicher Gesamtbedarf erzeugt. Bei Betrachtung der absoluten Anzahl der erforderlichen Knoten in Anhang A.4 fällt auf, dass ein kleinerer durchschnittlicher Ressourcenbedarf von Pods weniger Knoten benötigt. Das Auf-

teilen ressourcenintensiver Applikationen in kleinere Komponenten beeinflusst das Scheduling positiv.

Die Verteilung der Requests von Pods in den realen Clustern folgt keiner Verteilung, die in der Simulation verwendet wird. Abschließend soll angedeutet werden, dass die Ergebnisse der Simulation auch auf eine empirische Verteilung übertragbar sind. Abbildung 5.6 stellt links die Nutzung von CPU-Kernen und Arbeitsspeicher für 6487 Pods im Metal-Cluster in eu-de-2 am 4. Juli um 12 Uhr dar. Die rechte Darstellung stellt die Ressourcennutzung abzüglich der Requests dar. Eine Nutzung, welche die Requests überschreitet, wird dann durch eine positive Zahl ausgedrückt.

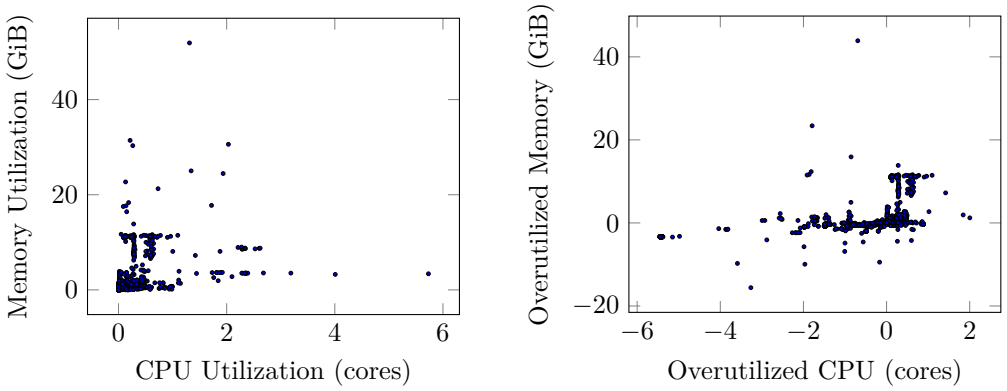

Abbildung 5.6: CPU- und Arbeitsspeichernutzung (links) abzüglich der Requests (rechts) der Pods im Metal-Cluster in eu-de-2 am 4. Juli

Würden die Requests der tatsächlichen Nutzung entsprechen, müssten sich die Punkte im rechten Streudiagramm um den Koordinatenursprung gruppieren. Dadurch ist die Diskrepanz zwischen den spezifizierten Requests und der tatsächlichen Nutzung ersichtlich. Ein Einplanen basierend auf den spezifizierten Requests ist daher nicht zielführend. Unter der Annahme, dass die Requests der Ressourcennutzung entsprechen, wird eine Simulation mit $\vec{X}_i$ durchgeführt, deren Verteilung der im linken Streudiagramm entspricht. Abschnitt 5.2 diskutiert, wie sich die Differenz zwischen der Ressourcennutzung und den Requests verringern lässt. Nachteilig an der Betrachtung an dieser Stelle ist, dass die Ressourcennutzung vom Messzeitpunkt abhängt.

Die Reihenfolge des Eintreffens wird durch zufälliges Mischen der fest definierten $\vec{X}_i$ variiert. Es werden die Daten aus einem Metal-Cluster genutzt, da die Knotengröße als konstant angenommen werden kann[2]. Durch die Simulation von 1500 Listen ergibt sich die in Tabelle 5.1 angegebene durchschnittliche Anzahl an erforderlichen Knoten pro Heuristik. Ähnlich zur restlichen Simulation benötigen die Heuristiken, die Vector-Dot (VD) nutzen die wenigsten Knoten. Kubernetes Least (KL) erfordert erneut die meisten Knoten.

[2]In den Metal-Clustern gibt es zwei verschiedene Knotengrößen. Die Pods jedes größeren Knoten könnten auch auf genau einem kleineren Knoten ausgeführt werden.

Heuristik	Knoten	Heuristik	Knoten
FF	45.8473	VD	45.4727
KL	46.2487	KR	45.8220
KM	45.8440	KVD	45.4047
PP	45.7173		

Tabelle 5.1: Durchschnittliche Anzahl an Knoten für verschiedene Heuristiken bei empirisch-verteiltem Bedarf

5.2 Ressourcenmanagement

Zentral für das Bin-Packing ist die Spezifikation der Ressourcen, die Pods zu ihrer Laufzeit benötigen. Bereits in den Experteninterviews in Abschnitt 3.3 hat sich ergeben, dass das manuelle Festlegen der Ressourcen aufwendig und nach Abschnitt 3.2 auch ungenau ist. Erstgenanntes kostet Arbeitszeit, letztgenanntes führt zu einer schlechten Auslastung. Das Kubernetes-Projekt bietet mit dem Vertical-Pod-Autoscaler (VPA) eine Komponente, um den zukünftigen Bedarf von Pods abzuschätzen und diesen Bedarf, wenn konfiguriert, in deren Spezifikation einzutragen. Ähnlich wie bei den Bin-Packing-Heuristiken wird zunächst das Verfahren des VPA erfasst und dann mit dem deutlich besser untersuchten Autoregressive-Integrated-Moving-Average-Modell (ARIMA-Modell) verglichen.

Vertical-Pod-Autoscaler

Der VPA generiert für Pods eine Abschätzung der benötigten CPU-Zeit und des maximalen Bedarfs an Arbeitsspeicher. Mangels Dokumentation wurde die Funktionsweise des VPA anhand des Quellcodes [43] nachvollzogen. Konkrete Zahlen basieren in Folgendem auf der Standardkonfiguration des VPA. Beschreibe X eine Zufallsvariable, welche eine geeignete Metrik erfasst, und t die Dauer der Aufzeichnung in Tagen. Weiterhin sei X_i das zugehörige Perzentil. Dann sind Abschätzungen des VPA beschrieben durch:

$$Lower(X,t) = 1.15 * X_{50\%} * (\frac{t}{t * 10^{-3}})^2 \tag{5.1}$$

$$Target(X,t) = 1.15 * X_{90\%} \tag{5.2}$$

$$Upper(X,t) = 1.15 * X_{95\%} * \frac{1+t}{t} \tag{5.3}$$

Lower und *Upper* repräsentieren eine Art „Konfidenzintervall“ für $Target$. Der Faktor 1.15 wird als zeitunabhängiger und der dritte Faktor als zeitabhängiger Puffer

hinzugefügt. Die Verteilung von X wird mittels eines Histogramms mit exponentiellem Verfall approximiert. Dabei ist die Gewichtung eines Eintrags w_s im Zeitpunkt s, welche zur Ermittlung des Perzentils beiträgt, gegeben durch:

$$w_s = w_t * 2^{(t-t_r)/t_{1/2}} \tag{5.4}$$

mit w_t als Ausgangsgewichtung zum Zeitpunkt t, t_r als Referenzzeitpunkt und $t_{1/2}$ als Halbwertszeit. Neue Werte werden mit einer höheren Gewichtung einbezogen als bisherige, wodurch die Verteilung von X basierend auf aktuellen Daten angenähert wird. Die Halbwertszeit beträgt 24 Stunden. In Abschnitt 1.2 ist erwähnt, dass Kubernetes mit Deployments, StatefulSets und DaemonSets höhere Konstrukte besitzt, welche mehrfach eine identische Podspezifikation instanziieren. Metriken von Containern, die ein Teil eines Pods sind, deren höheres Konstrukt identisch ist, werden in einem gemeinsamen Histogramm verfolgt. Diese Zuordnung erfolgt über das Abgleichen von Labels und Verweisen auf Besitzer von Pods.

Für das Histogramm zur Abschätzung des Bedarfs an Arbeitsspeicher wird die maximale Arbeitsspeichernutzung pro Container in nicht überlappenden 24-Stunden-Fenstern nachverfolgt. Dafür wird die `container_memory_working_set_bytes` Metrik benutzt, die in Abschnitt 3.2 diskutiert wurde. Bei einem exemplarischen Deployment mit drei Pods, die je einen Container enthalten, werden alle 24 Stunden drei Werte in das zugehörige Histogramm aufgenommen. Innerhalb eines Zeitfensters werden die Werte aktualisiert, falls ein neues Maximum erreicht wird. Der Einfügezeitpunkt ist dabei immer das Ende des aktuellen Zeitfensters. Wenn Container vom OOM-Killer beendet werden, wird dies besonders berücksichtigt. Es wird ein künstlicher Messwert eingefügt, falls dieser das Maximum innerhalb des aktuellen Zeitfensters ist. Dieser künstliche Messwert *value* wird folgendermaßen ermittelt:

$$mem_{used} = max(mem_{currentMax}, mem_{req}) \tag{5.5}$$

$$value = max(mem_{used} + 100MiB, 1.2 * mem_{used}) \tag{5.6}$$

Dabei beschreibt $mem_{currentMax}$ das gemessene Maximum im aktuellen Zeitfenster und mem_{req} den aktuellen Request für den Arbeitsspeicher.

Das Histogramm für die Abschätzung der CPU-Zeit wird über die vom Container verwendete CPU-Zeit aufgebaut. Genutzt werden Differenzen der `container_cpu_usage_seconds_total` Metrik. Dabei gehen die Werte mit einer Gewichtung entsprechend der Anzahl der Kerne in den CPU-Requests ein, jedoch mindestens mit einer Gewichtung von 0.1. Sollten die CPU-Requests erhöht werden, werden alle vorher erfassten Werte für die Bestimmung des Perzentils geringer gewichtet.

Dieser Mechanismus soll CPU-Starvation zusätzlich entgegenwirken. Ein Pod, welcher adäquate CPU-Requests und -Limits besitze und diese nahezu vollständig auslaste, benötige kurzfristig ein Vielfaches der üblichen CPU-Zeit. Die zunächst generierte Vorhersage beträgt $1.15 * X_{90\%}$. Nach Anwendung dieser auf die Requests und Limits können überhaupt erst Werte in das Histogramm aufgenommen werden, welche das alte Limit überschreiten. Durch die höhere Gewichtung der neuen Werte wird das Perzentil schneller nach oben korrigiert als ohne, sodass eine weitere Vorhersage nach $1.15 * X_{90\%}$ bald nennenswert größer als die erste Vorhersage ist.

Insgesamt erscheint das Vorgehen durch praktischen Bedarf entstanden anstatt mathematisch fundiert. Zusätzlich erschweren die diversen Sonderregeln eine Analyse. Der Arbeitsspeicherbedarf wird bei real schwankender Nutzung tendenziell zu groß abgeschätzt, aufgrund der 24-stündigen Zeitfenster und des Aufschlags von 15% Puffer.

Ohne die Zeitfenster ist bereits die Verwendung der Histogramme zur Bestimmung der Perzentile fraglich, da unterstellt wird, dass der nächste Messwert unabhängig von den vorherigen ist. Die Autokorrelation beschreibt die Abhängigkeit der Werte einer Zeitreihe von den vorherigen Werten bei einer gegebenen Zeitdifferenz [3, S. 16]. Per Ermittlung jener lässt sich für die CPU-Zeit und den Arbeitsspeicher unter Vernachlässigung des 24-Stunden-Zeitfensters zeigen, das Abhängigkeiten zu vorherigen Werten besteht. Die Histogramme bestimmen dann Perzentile einer Wahrscheinlichkeitsverteilung, die nicht vorliegt. Die Graphen für die Autokorrelation bei verschiedenen Zeitdifferenzen eines exemplarischen Containers sind in Abbildung 5.7 gegeben.

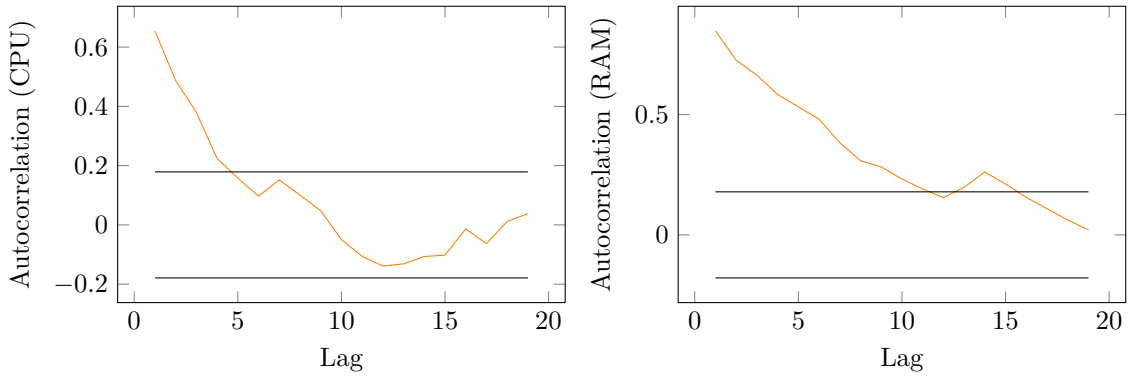

Abbildung 5.7: Autokorrelation für einen Elasticsearch-Container für CPU-Zeit und RAM

Für lange Zeitreihen, die durch unabhängige und identisch verteilte Zufallsvariablen entstehen, sind die Autokorrelationen normalverteilt mit $\mu = 0$ und $\sigma^2 = 1/T$. Wenn eine untersuchte Zeitreihe dem entspricht, müssen 95% der Autokorrelationen innerhalb der Grenze $\pm 1.96/\sqrt{T}$ mit T als Anzahl der Messungen liegen. Diese Grenze ist durch die schwarzen Linien in Abbildung 5.7 abgetragen und wird ausreichend oft überschritten [3, S. 30].

Zeitreihenanalyse

Möglicherweise können bessere Vorhersagen durch Berücksichtigung der zeitlichen Abhängigkeit getroffen werden. Die Analyse und Vorhersage von Zeitreihen ist ein ausführlich untersuchtes Wissenschaftsgebiet, da jene auch außerhalb des hier diskutierten Anwendungsfalls, exemplarisch in der Ökonometrie, relevant sind [16, S. 2–4]. Grundlegend werden Muster aus der aufgenommenen Zeitreihe extrahiert und anhand dieser eine Vorhersage erstellt. Ursprünglich wurden dafür statistische Methoden verwendet. Mit dem Aufkommen von Deep-Learning finden darauf basierende Modelle zunehmend Anwendung. Beide Verfahren lassen sich dem Oberbegriff des maschinellen Lernens zuordnen. Im Gegensatz zu Deep-Learning-Modellen benötigen die statistischen Modelle kleinere Datenmengen, weshalb diese im Folgendem näher untersucht werden [17, S. 1364].

Die lineare Abhängigkeit eines Werts x_t einer Zeitreihe von den vorherigen Werten x_{t-i} kann mittels autoregressiver Modelle ausgedrückt werden. Ein autoregressiver Prozess der Ordnung p wird beschrieben durch [16, S. 49]:

$$AR(p) : x_t = u_t + \sum_{i=1}^{p} \alpha_i x_{t-i} \tag{5.7}$$

Der nächste Wert der Zeitreihe ist die gewichtete Summe der p vorherigen Werte mit α_i als Gewichten und einem zufälligen Fehler u_t. Da in den Metriken für die Abschätzung der Ressourcen kein beziehungsweise lediglich ein marginaler Messfehler vorliegt, ist u_t besser als nicht durch Linearkombination der vorherigen Messwerte erklärte Komponente aufzufassen. Der nächste Messwert kann alternativ durch Linearkombination der q nicht-erklärbaren vorangegangenen Komponenten modelliert werden. Ein solcher Moving-Average-Prozess ist beschrieben durch [16, S. 65]:

$$MA(q) : x_t = u_t + \sum_{i=1}^{q} \beta_i u_{t-i} \tag{5.8}$$

Beide Ansätze lassen sich in einem Autoregressive-Moving-Average-Modell kombinieren [16, S. 75]:

$$ARMA(p,q) : x_t = u_t + (\sum_{i=1}^{p} \alpha_i x_{t-i}) + \sum_{i=1}^{q} \beta_i u_{t-i} \tag{5.9}$$

Dabei wird die erfasste Zeitreihe als eine konkrete Realisierung eines stochastischen Prozesses verstanden. Der stochastische Prozess kann als ein T-dimensionaler Vektor von Zufallsvariablen $(X_1, X_2, \ldots, X_T)$ interpretiert werden, wobei T die Länge der Zeitreihe beschreibt. Damit kann der stochastische Prozess durch eine T-

dimensionale Verteilungsfunktion vollständig beschrieben werden. Bei ausreichend großem T ist dies jedoch impraktikabel, weshalb auf die Beschreibung durch das erste und zweite Moment, also Erwartungswert, Varianz und Kovarianz, zurückgegriffen wird. Um diese zu schätzen, sind eigentlich mehrere Realisierungen pro Zeitpunkt nötig. Daher wird angenommen, dass die Zeitreihe ergodisch ist. Dies bedeutet, dass sich die Schätzer für den Erwartungswert und die Varianz basierend auf der endlichen erfassten Zeitreihen bei großem T den tatsächlichen Werten annähern. Da letztere Aussage nur sinnvoll ist, wenn alle X_t denselben Erwartungswert und Varianz besitzen, wird dies ebenfalls unterstellt [16, S. 12–13].

Damit ein stochastischer Prozess ergodisch ist, muss dieser stationär sein. Ein stochastischer Prozess ist strikt stationär, wenn die Verteilungsfunktion der X_t nicht von der Zeit abhängt. Dies ist praktisch jedoch schwierig anzuwenden. Ein stochastischer Prozess ist schwach stationär, wenn $E[X_t] = \mu_t = \mu, \mu \quad const.$ und $Cov[X_t, X_s] = E[(X_t - \mu)(X_s - \mu)] = \gamma(|s - t|)$, also zu jedem Zeitpunkt der Erwartungswert identisch und die Kovarianz eine Funktion γ des zeitlichen Abstands zweier Werte ist [16, S. 13–14].

Bevor ein ARMA-Modell angewendet werden kann, muss Stationarität hergestellt werden. Dies kann mitunter durch Differenzbildung zwischen den Messwerten der Zeitreihe erfolgen, z.B. $\dot{x}_t = x_t - x_{t-1}$. Bei Bedarf kann die Differenzbildung auch mehrfach angewandt werden. Ein Prozess der durch d-faches Differenzbilden zu einem $ARMA(p, q)$-Prozess transformiert werden kann, welcher nun die Änderungsrate beschreibt, wird als $ARIMA(p, d, q)$-Prozess bezeichnet [16, S. 159–160]. Exemplarisch ist dieses Vorgehen in Abbildung 5.8 dargestellt, wobei links die ursprüngliche Zeitreihe und rechts jene nach einmaliger Differenzbildung (nun stationär) zu sehen ist. Der lineare Trend ist bereinigt.

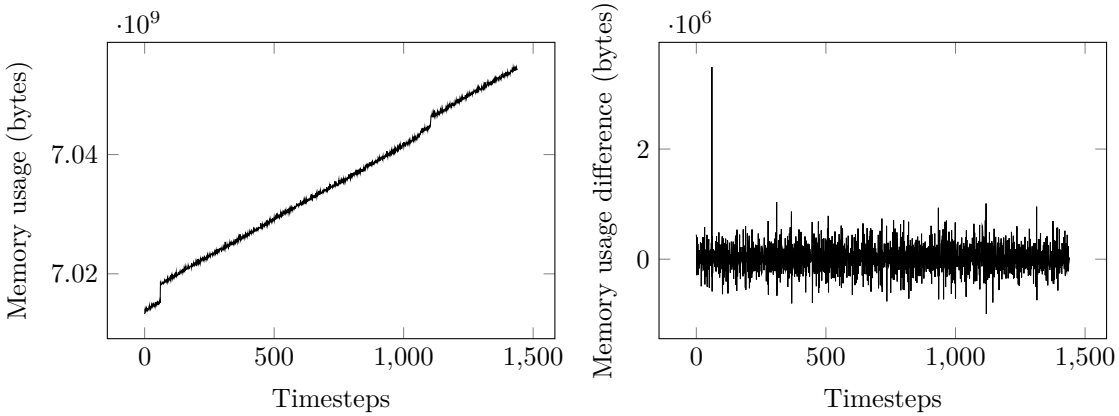

Abbildung 5.8: Erzeugung von stationären Zeitreihen durch Differenzbildung

Der lineare Schätzer für die Ein-Schritt-Vorhersage eines $ARMA(p, q)$-Modells ab dem Zeitpunkt t ist gegeben durch:

$$\hat{x}_{t+1} = (\sum_{i=1}^{p} \alpha_i x_{t-i+1}) + \sum_{i=1}^{q} \beta_i u_{t-i+1} \tag{5.10}$$

Vorhersagen für fernere Zeitpunkte lassen sich durch rekursives Anwenden dieser Gleichung erhalten [16, S. 82–85]. Für $ARIMA(p, 1, q)$-Modelle, die später verwendet werden, müssen die Vorhersagen der Änderung auf den letzten Wert der ursprünglichen Zeitreihe addiert werden, um die finale Vorhersage zu erhalten:

$$\hat{x}_{t+1} = x_t + \hat{\hat{x}}_t \tag{5.11}$$

Der τ-Schritt Vorhersagefehler $f_t(\tau)$ eines ARMA-Modells folgt einem $MA(\tau - 1)$-Prozess mit einem Erwartungswert von $E(f_t(\tau)) = 0$ und Varianz von $V(f_t(\tau)) = (1 + \sum_{i=1}^{\tau-1} \psi_i^2)\sigma^2$. Letztere Gleichung kann mit einem Schätzer für σ^2 aus der Parameterschätzung zur Konstruktion von Konfidenzintervallen verwendet werden. Diese sind allerdings zu eng, da Fehler in der Abschätzung der Parameter $\vec{\alpha}$ und $\vec{\beta}$ nicht berücksichtigt werden [16, S. 80–81]. Die ψ_i ergeben sich aus der Überführung des $ARMA(p, q)$-Modells in ein $MA(\infty)$-Modell mittels der Methode der unbestimmten Koeffizienten [16, S. 40–41]. Für $\tau \to \infty$ konvergiert die Varianz des Vorhersagefehlers gegen die Varianz der zugrundeliegenden Zeitreihe [16, S. 81]. ARMA-Modelle sind daher nur für einen kurzen Vorhersagehorizont geeignet.

Um eine Vorhersage berechnen zu können, müssen die Parametervektoren $\vec{\alpha}$ und $\vec{\beta}$ für eine gegebene Zeitreihe abgeschätzt werden. Zu Beginn ist problematisch, dass die Residuen u_t nicht bekannt sind. Die Verfahren zur Parameterbestimmung bei gegebenen p und q gliedern sich daher in zwei Teile:

- Zunächst wird ein $AR(k)$-Prozess mit $k >> p$ abgeschätzt. Es gilt, dass jeder (invertierbare) $ARMA(p, q)$-Prozess identisch mit einem $AR(\infty)$-Prozess ist [16, S. 75]. Basierend auf den Daten der Zeitreihe und der Ein-Schritt-Vorhersage (Gleichung 5.10) lassen sich per $\hat{u}_t = \hat{x}_t - x_t$ die geschätzten Residuen ermitteln. Für die Ermittlung der Koeffizienten können die Yule-Walker-Gleichungen oder multiple lineare Regression genutzt werden [16, S. 56].

- Mittels der geschätzten Residuen lassen die Parameter des $ARMA(p, q)$-Modells schätzen. Dafür kann der Algorithmus von Hannan und Rissanen [3, S. 137–139] oder alternativ ein Maximum-Likelihood-Schätzer verwendet werden. Letzterer Ansatz führt zu einem nicht linearen Gleichungssystem, welches numerisch zu lösen ist [3, S. 139–142]. Weiterhin lässt sich die lineare Regression iterativ anwenden, bis die geschätzten Residuen konvergieren [15, S. 305–306].

Sind die Ordnungen p und q nicht bekannt, können mehrere ARMA-Modelle mit jeweils verschiedenen Ordnungen berechnet werden und ein finales Modell anhand

eines Informationskriteriums ausgewählt werden [16, S. 75–76]. Das in der Literatur häufig genutzte Vorgehen ist die Anwendung der Yule-Walker Gleichungen und die Verwendung des Maximum-Likelihood-Schätzers [3, S. 121–122]. Dieses ist oft in den üblichen Umgebungen für Statistical-Computing implementiert.

Die Implementierung in dieser Arbeit soll dem Anspruch genügen, potenziell in Kubernetes einsetzbar zu sein. Die Umgebungen für Statistical-Computing sind gewöhnlich umfangreich und benötigten viele Abhängigkeiten, was den Wartungs- und Paketierungsprozess erschwert. Aus Sicht der Implementierung fällt bei der Diskussion der Parameterschätzung auf, dass diese vollständig mittels multipler linearer Regression umsetzbar ist. Die multiple lineare Regression ist in `LAPACK`, einer Fortran-Bibliothek für lineare Algebra, effizient umgesetzt [1]. Prinzipiell kann jede Sprache mit Bindings zu `LAPACK` verwendet werden. Die Simulation ist in `Rust` geschrieben, da jenes zu nativem Code kompiliert, was bei den Berechnungen, abseits der Regression, eine höhere Geschwindigkeit bringt. `Rust` bietet auch Bibliotheken für eine potenzielle Integration mit Prometheus und Kubernetes.

Die Simulation nutzt bereits erfasste Zeitreihen, welche in Prometheus bereitliegen (siehe Abschnitt 3.2). Ein zweistündiger Ausschnitt mit einem einmütigen Messintervall dient der Parametrisierung der Modelle, um die folgenden fünf Minuten vorherzusagen. Die Vorhersagen können dann mit den real erfassten Daten verglichen werden.

Es wird unterstellt, dass einmaliges Differenzbilden hinreichend ist, um die jeweilige Zeitreihe als stationär betrachten zu können. Ebenso, dass die Residuen normalverteilt sind. Dadurch lässt sich die Varianz des Vorhersagefehlers auf die originale Zeitreihe zurückrechnen, da die Summe zweier normalverteilter Zufallsvariablen erneut normalverteilt ist. Für die Varianz der Summe gilt dabei: $\sigma^2 = \sigma_1^2 + \sigma_2^2$.

Nach der Differenzbildung wird für die Zeitreihe ein $AR(48)$-Modell parametrisiert, um die Residuen initial abzuschätzen. Dann wird iterativ ein $ARMA(p, q)$ per linearer Regression parametrisiert. Anstatt die Konvergenz der Residuen zu prüfen, wird nach acht Iterationen abgebrochen, um eine Terminierung dieser Schleife zu garantieren. Für die Ordnung p und q werden alle Kombinationen aus $2 \leq p, q \leq 9$ probiert. Das Modell, welches das Akaike Informationskriterium (Gleichung 5.12) minimiert, wird für die Vorhersage verwendet. T beschreibt die Anzahl der Messungen, $m = p + q$ die Anzahl der Parameter des Modells und $\hat{u}_t$ das im Zeitpunkt t geschätzte Residuum [16, S. 56].

$$AIC = ln(\frac{1}{T}\sum_{t=1}^{T}\hat{u}_t) + \frac{2m}{T} \tag{5.12}$$

Das Akaike Informationskriterium schätzt die Anzahl der Parameter tendenziell zu groß ab [16, S. 57]. Applikationen, welche in einer Programmiersprache mit Garbage Collection umgesetzt sind, weisen durchaus zyklische Muster im Arbeitsspeicherver-

brauch auf. Im Kontext der Zeitreihenanalyse werden solche zyklischen Muster als Saisonalität bezeichnet und durch saisonale ARIMA-Modelle behandelt [3, S. 177]. Saisonalität wird in der Implementierung in dieser Arbeit nicht explizit berücksichtigt, was durch die Aufnahme von „zu vielen“ Parametern, also durch längeres „in die Vergangenheit schauen“, etwas kompensiert werden soll.

Das ausgewählte Modell wird schließlich für die fünfminütige Vorhersage mittels Gleichung 5.10 eingesetzt. Durch das ARMA-Modell wird pro Schritt die Änderung mit einem Fehler geschätzt und dieser Fehler ist in jedem weiteren Schritt enthalten. Aufgrund der Akkumulation des Vorhersagefehlers ist nur die Schätzung von wenigen Schritten in die Zukunft aussagekräftig.

Die τ-Schritt Vorhersagen $\hat{x}_{t+\tau}$ werden von den tatsächlichen Realisierungen abweichen. Übersteigt die Realisierung die Vorhersage führt dies entweder zu unzureichenden CPU-Kapazitäten oder OOM-Situationen. Übersteigt die Vorhersage die Realisierung werden Ressourcen reserviert, aber nicht genutzt. Beide Fälle sind zu minimieren. Aus Gründen der Kontinuität, wie in den Experteninterviews in Abschnitt 3.3 angemerkt, ist es naheliegend den Mangel an Ressourcen zu minimieren und die Verschwendung von Ressourcen zu akzeptieren.

Diese Überlegung führt zu dem gewählten Vergleich zwischen dem VPA und dem ARIMA-Modell, welcher der Anzahl an Pods mit einem Mangel an Ressourcen als unabhängige Variable die nicht benötigten Ressourcen als abhängige Variable gegenüberstellt. Basierend auf diesem Vergleich kann dann eine Konfiguration mit akzeptabler Sicherheit vor Mangelsituationen gewählt werden.

Ein Vorteil der ARIMA-Modelle ist, dass die Unsicherheit der Vorhersage durch die Varianz des Vorhersagefehlers ausgedrückt ist, welcher von der Schwankung in der ursprünglichen Zeitreihe abhängt. Durch Aufschlag eines Vielfachen der Standardabweichung des Vorhersagefehlers kann die Wahrscheinlichkeit eines Mangels an Ressourcen gesenkt werden. Gleichzeitig nimmt die Verschwendung an Ressourcen zu. Die angepasste Vorhersage sei ausgedrückt durch:

$$\hat{x}^*_{t+\tau} = \hat{x}_{t+\tau} + n * \sqrt{\sum_{i=1}^{\tau} V(f_t(i))} + n * 2MiB \tag{5.13}$$

$\hat{x}_{t+\tau}$ ergibt sich rekursiv nach Gleichung 5.11. Unter der Wurzel steht die aufsummierte Varianz der Vorhersagefehler der Änderungsraten, welche als normalverteilt angenommen werden. n beschreibe einen Sicherheitsfaktor. Quasi konstante Zeitreihen haben kaum Schwankung und daher lediglich einen kleinen Vorhersagefehler. Eine leichte Abwärtsbewegung am Ende der Zeitreihe führt sofort zu einer marginalen Mangelsituation, welche nicht durch die Varianz des Vorhersagefehlers korrigiert werden kann. Dies soll durch die lineare Komponente in Gleichung 5.13 kompensiert werden. Letztendlich muss genau ein Wert bestimmt werden, der in die Requests und

Limits des Containers eingetragen wird. Naheliegend ist die maximale angepasste Vorhersage $max(\hat{x}^*_{t+1}, \hat{x}^*_{t+2}, \hat{x}^*_{t+3}, \hat{x}^*_{t+4}, \hat{x}^*_{t+5})$, welche in den Vergleich eingeht.

Exemplarisch ist dieses Vorgehen für eine Zeitreihe in Abbildung 5.9 visualisiert. Die schwarze Linie repräsentiert die Zeitreihe, wie sie gemessen wurde. Die orange Linie stellt die Vorhersage dar, welche sich direkt aus dem ARIMA-Modell ergibt, und die blaue Linie die angepasste Vorhersage mit $n = 2$. Die Abschätzung für den gesamten fünfminütigen Vorhersagehorizont ist das Maximum der blauen Linie und wird durch die grüne Linie dargestellt. Den erfassten zu viel reservierten Arbeitsspeicher, welcher zwischen dem Maximum der angepassten Vorhersage und dem Maximum der Realisierung im Vorhersagehorizont besteht, stellt die gestrichelte Linie dar.

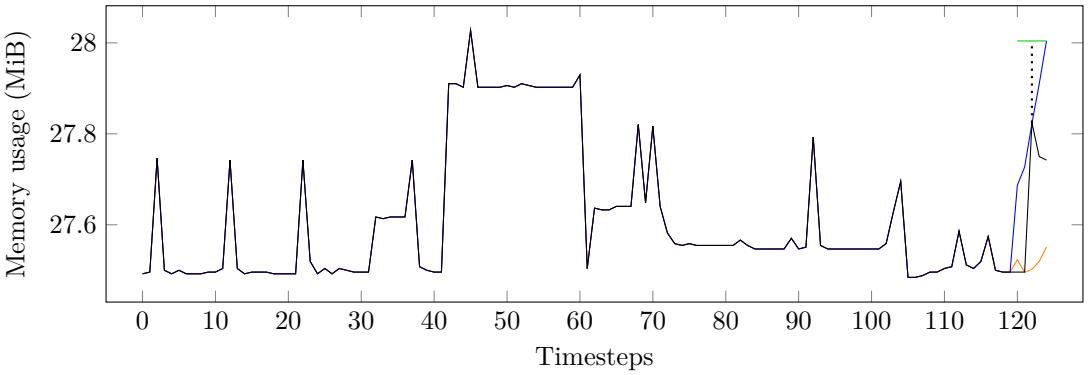

Abbildung 5.9: Simulationsverfahren zur Zeitreihenvorhersage

Für die Vorhersage des VPA lässt sich ebenso ein Sicherheitsfaktor einfügen, wie in Gleichung 5.14 dargestellt. Der Faktor ist konfigurierbar, der lineare Zusammenhang willkürlich gewählt.

$$\hat{v}^*_{t+\tau} = (0.04n + 0.99) * X_{90\%} \tag{5.14}$$

Es wird ein ARIMA-Modell je Container parametrisiert. Für den Vergleich zu dem Verfahren des VPA wird dessen Aggregation für Deployments, StatefulSets und DaemonSets nicht berücksichtigt. Im Nachgang müssen die Schätzungen für Container zusammengefasst werden oder das `InPlaceVerticalPodScaling`-Feature genutzt werden, welches die Anpassung von Requests und Limits zur Laufzeit des Containers erlaubt. Da lediglich ein Maximum vom VPA alle 24 Stunden berücksichtigt wird und ein zweistündiger Zeitraum untersucht wird, ergibt sich vereinfacht:

$$\hat{v}^*_{t+\tau} = (0.04n + 0.99) * max(x_1, x_2, \ldots, x_t) \tag{5.15}$$

Wendet man Gleichung 5.15 auf die Zeitreihe aus Abbildung 5.9 an, ergibt sich eine Abschätzung von rund $3.15 * 10^7$ Bytes, die außerhalb des dargestellten Bereichs

liegt. Damit diese Vereinfachung nicht mehr gilt, muss das Histogramm mehr als einen Wert umfassen, also das Zeitfenster deutlich kleiner als 2 Stunden sein. Die genaue Länge hängt von der Halbwertszeit des Histogramms ab. Das Argument, dass ein zeitlich grobes Verfahren mit einem Hochauflösenden verglichen wird, trifft nur begrenzt zu. Falls sich eine höhere zeitliche Auflösung lohnt, wurde das Verfahren des VPA mit dem 90%-Perzentil und ohne Zeitfenster zusätzlich auf den Arbeitsspeicher angewandt. Basierend auf allen 1664 nicht-konstanten Zeitreihen[3] für das Working-Set im Scaleout-Cluster in eu-de-2 am 22. Mai von 10:00 Uhr bis 11:59 Uhr ergibt sich für $n = 1, 1.5, 2, \ldots, 4$ der in Abbildung 5.10 dargestellte Zusammenhang.

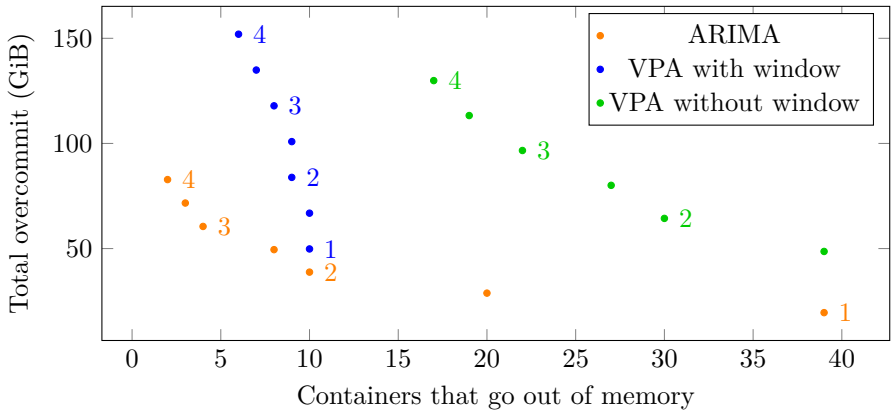

Abbildung 5.10: Vergleich zwischen den Vorhersagen des VPAs und den ARIMA-Modellen bezüglich Arbeitsspeicher

Entlang beider Achsen sind kleinere Werte vorteilhaft. Der Sicherheitsfaktor n ist im Diagramm durch die Ziffern an den Punkten angegeben. Für den simulierten Datensatz sind damit ARIMA-Modelle bei großem Sicherheitsniveau besser zur Vorhersage geeignet als das Verfahren des VPA für Arbeitsspeicher. Das Verfahren des VPA ohne Zeitfenster ist das Schlechteste aus den untersuchten Verfahren.

Die Abschätzung einer Zeitreihe per ARIMA-Modell benötigt auf der Hardware des Autors (Apple M1 Max) rund zehn Millisekunden. Damit ergibt sich für die Abschätzung aller Zeitreihen ohne Parallelisierung eine Laufzeit von rund 16 Sekunden, was bezogen auf den fünfminütigen Vorhersagehorizont zureichend ist. Zur Interpretation ist anzumerken, dass Container, welche in der Simulation unzureichend Arbeitsspeicher erhalten würden, real nicht unbedingt in die OOM-Situation kommen, da einige Laufzeitumgebungen die cgroups berücksichtigen und eine Garbage Collection durchführen können.

Das Bestimmtheitsmaß R^2 gibt an, wie gut eine lineare Regression die vorgelegten Daten erklärt. Ein Wert nahe 1 beschreibt eine gute Anpassung. Werte nahe 0 drücken aus, dass die unabhängigen Variablen keinen Einfluss auf die abhängige Variable haben. R^2 wurde für jedes Modell ermittelt und die Verteilung in Abbildung 5.11 mittels eines Boxplots dargestellt. Die Länge der Antennen ist auf

[3]Die Entwicklung von Zeitreihen, bei denen alle Messwerte identisch sind, ist trivial abschätzbar.

$1.5 * IQR$ beschränkt, wobei IQR den Interquartilsabstand bezeichnet. Das 75%-Perzentil liegt unter 0.5. Die Mehrheit der Zeitreihen ist durch die angewendete Modellierung schlecht beschrieben. Ebenso wurde angenommen, dass die Residuen normalverteilt sind, damit bei Invertierung der Differenzbildung, auch die Varianz des Vorhersagefehlers analytisch ausgedrückt werden kann. Die Verteilung der Residuen kann mittels Hypothesentests überprüft werden. Bei Anwendung eines χ^2-Tests auf einem Konfidenzniveau von 95% muss die Nullhypothese, dass die Residuen normalverteilt sind, bei 1499 Zeitreihen abgelehnt werden.

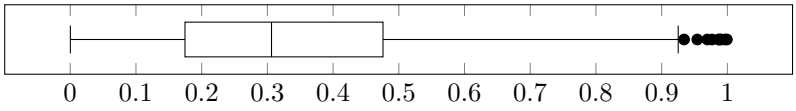

Abbildung 5.11: R^2 für die parametrisierten Modelle der Simulation

Insgesamt passen die $ARIMA(p, 1, q)$-Modelle eher schlecht zu den Zeitreihen. Trotz dessen sind die Vorhersagen besser als jene, die sich durch Verwendung des VPA ergeben. Dies lässt sich in zwei Richtungen interpretieren:

- Die angepasste Abschätzung in der Simulation wird von der Standardabweichung des Vorhersagefehlers dominiert, welche von der Varianz der Datenreihe abhängt. Möglicherweise ist es zureichend die zeitliche Komponente zu vernachlässigen und die Varianz der Datenreihe in die Vorhersage anstatt eines konstanten Faktors, wie in Gleichung 5.15, einzubeziehen.

- Es sind komplexere Modelle aus dem Bereich der Zeitreihenanalyse nötig, um die Zeitreihen besser zu beschreiben. Angedeutet wurde bereits die Bedeutung von Saisonalität und saisonalen ARIMA-Modellen. Bezogen auf die Zeitreihe in Abbildung 5.9 gibt es rund alle $2+10*i, i \in \mathbb{N}$ einen Peak. Diese Information wird bisher nicht genutzt. Bezüglich der Fehler, die als identisch normalverteilt angenommen werden, und der Residuen, die dies nicht bestätigen, gibt es mit den autoregressive conditional heteroscedastic residual (ARCH) Modellen, solche die diese Annahme lockern [16, S. 284–285].

Das erläuterte Vorgehen kann auf die Vorhersage der CPU-Zeit übertragen werden. Beim Verfahren des VPA kann direkt das 90%-Perzentil verwendet werden, da die Implementierung die Messwerte ebenfalls minütlich erfasst. Es wird angenommen, dass die CPU-Requests konstant sind. Weiterhin werden andere Parameter mit $n = 0, 1 \ldots, 7$ für die Anpassung der Vorhersage genutzt, zwecks eines geeigneten Wertebereichs in der Ausgabe der Simulation.

$$\hat{x}^*_{t+\tau} = \hat{x}_{t+\tau} + (0.15n + 0.35) * \sqrt{\sum_{i=1}^{\tau} V(f_t(i))} + 0.0005n + 0.0015 \tag{5.16}$$

$$\hat{v}^*_{t+\tau} = (1 + 0.15n) * X_{90\%} \tag{5.17}$$

Die Ergebnisse der Simulation bezüglich des CPU-Bedarfs für den identischen Zeitraum sind in Abbildung 5.12 dargestellt. Es gehen 1719 nicht-konstante Zeitreihen ein. Da konstante Zeitreihen gefiltert werden und die Zeitreihen bezüglich der CPU-Zeit anscheinend volatiler sind als jene bezüglich des Arbeitsspeichers, stimmt die Anzahl der Zeitreihen nicht mit der vorhergehenden Simulation überein. Die Güte der Regression ist mit der bezüglich der Simulation des Arbeitsspeichers vergleichbar. Da ein Mangel an CPU-Zeit nicht unmittelbar die Terminierung eines Containers bedingt [5, S. 26], ist neben der absoluten Anzahl an Containern mit einer zu kleinen Abschätzung auch die Größe des Mangels interessant, welche im rechten Diagramm an der Abszisse abgetragen ist.

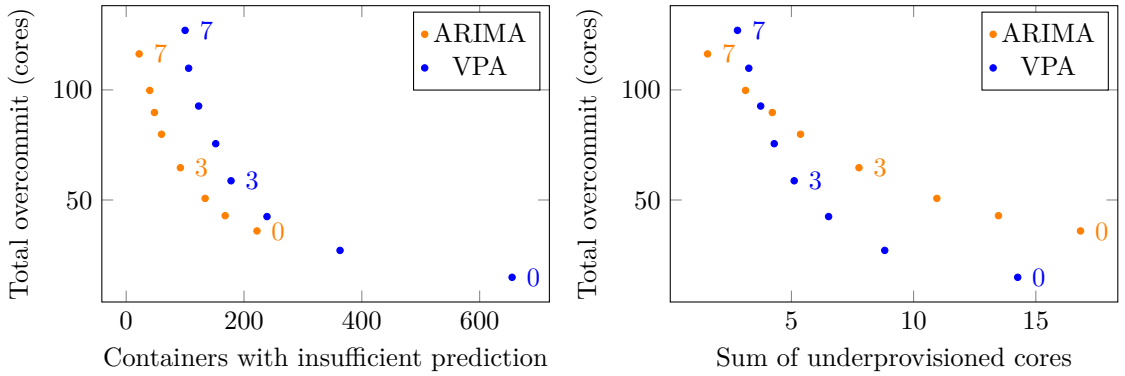

Abbildung 5.12: Vergleich zwischen den Vorhersagen des VPAs und den ARIMA-Modellen bezüglich CPU-Zeit

Bezüglich der Minimierung der Anzahl der Container, die unzureichend CPU-Zeit zugeordnet bekommen, sind die ARIMA-Modelle bei hohem Sicherheitsniveau besser geeignet als das Verfahren des VPA. Das rechte Diagramm stellt die Differenz zwischen der zu klein abgeschätzten CPU-Zeit und der tatsächlichen Realisierung summiert über alle Container an der Abszisse dar. Diese Summe ist bei den ARIMA-Modellen weitgehend größer als beim VPA. Zusammengefasst sind bei den ARIMA-Modellen weniger Container von einem Mangel betroffen. Jene die betroffen sind, weisen allerdings größere Differenzen zur tatsächlichen Realisierung auf. Damit kann nicht eindeutig beurteilt werden, welche Methodik zu einem besseren Ergebnis führt. Bezogen auf die in Abschnitt 3.2 dargestellte Situation bezüglich CPU-Requests und -Limits, welche nicht der Auslastung in den Clustern entspricht, werden beide Verfahren eine akkuratere Spezifikation bringen.

Um einen Vergleich zwischen verschiedenen Konfigurationen der ARIMA-Modelle und dem Verfahren des VPA zu ermöglichen, wurde der Sicherheitsfaktor n eingeführt. Wird die Zuordnung von n zu den Konfigurationsparametern geändert, lässt sich jedes Verfahren beliebig schlecht darstellen. Wenn anstatt $0.04n + 0.99$ in Gleichung 5.15 exemplarisch die Zuordnung $0.1n + 2$ verwendet wird, schiene der VPA einen sehr großen Overcommit zu erzeugen. Die Festlegung der Zuordnung basiert auf einer manuellen Suche nach Konfigurationen, die dem beschriebenen Ziel nahekommen. Da alle Verfahren Ergebnisse in derselben Größenordnung erzeugen, sollte

keines stark bevorteilt oder benachteiligt sein. Ein exakteres Vorgehen ist die Ermittlung der optimalen Konfiguration jedes Verfahrens bezüglich eines analytisch formulierten Optimierungsziels. Die optimal konfigurierten Verfahren können dann mittels der Zielfunktion bewertet werden. Für diese Arbeit ist die Lösung dieses Optimierungsproblems allerdings zu umfangreich.

Durch die Simulation ließ sich zeigen, dass die Modelle der Zeitreihenanalyse Vorhersagen mit geeigneter Qualität generieren. Zu untersuchen bleibt, ob dies auch für weitere Auslastungsdaten gilt. Bis die ARIMA-Modelle potenziell in Kubernetes benutzbar sind, sind zusätzlich einige operative Herausforderungen zu lösen. So muss ein Umgang mit Lücken in Zeitreihen gefunden werden. Weiterhin führt ein Erreichen der strikten Limitierung möglicherweise dazu, dass sich Aufwärtstrends nicht etablieren können, die aber nötig sind, um die Limitierung nach oben zu korrigieren. Vielleicht müssen entsprechende Puffer berücksichtigt werden, anstatt möglichst knappe Vorhersagen zu erzeugen. Die Simulation wurde mit Containern durchgeführt, welche in Pods verwaltet werden. Für die übergeordneten Konstrukte, wie Deployments, die mehrere Pods verwalten und Ressourcen spezifizieren, müssen die Vorhersagen geeignet aggregiert werden. Bezüglich der Verwaltung der Ressourcen im Allgemeinen lohnt sich der Ausbau der Funktionalität von Kubernetes bezüglich vertikaler Skalierung einzelner Pods und Container.

5.3 Auslastungsgrenzen

In Kapitel 4 wurde festgestellt, dass sich die Hardware vermutlich nicht vollständig auslasten lässt, ohne dass die Performance von Pods, aufgrund von Effekten im Prozessscheduling und der Speicherverwaltung, negativ beeinflusst wird. Dieser Abschnitt versucht eine Aussage darüber zu treffen, wo die Grenzen bezüglich der CPU- und Arbeitsspeicherauslastung liegen. Benchmarking, also die Bewertung der Performance, ist im Cloud-Umfeld schwierig, da zusätzliche Faktoren die Messung gegenüber traditionellem Benchmarking beeinflussen [12, S. 85]. Iosup, Prodan und Epema [12, S. 90–95] listen dafür zehn Herausforderungen auf, die von der aufwendigen Durchführung komplexem Lastverhaltens über die vielen Abstraktionsschichten und Änderbarkeit der Infrastruktur bis hin zur Wahl geeigneter Metriken reichen. Die folgende kurze Untersuchung im Rahmen dieser Arbeit kann der Komplexität der Thematik nicht gerecht werden, aber einen Anhaltspunkt bieten.

Zunächst sei der Untersuchungsgegenstand auf einen Kubernetes Knoten beschränkt. Dieser wird durch eine virtuelle Maschine mit 16 CPU-Kernen und 32 GiB Arbeitsspeicher bereitgestellt. Solche Knoten sind in den Scaleout-Clustern vorzufinden. Es wird die Geschwindigkeit eines Prozesses in einem Pod untersucht während in einem anderen Pod mittels `stress-ng` CPU- oder Arbeitsspeicherauslastung erzeugt wird.

Zur Erzeugung von CPU-Auslastung überprüft der zu untersuchende Prozess, ob eine zur Laufzeit definierte Zahl eine Primzahl ist, in dem eine Division durch alle kleineren natürlichen Zahlen durchgeführt wird. Das ist kein effizientes Verfahren, um zu überprüfen, ob eine Zahl eine Primzahl ist. Der Zweck besteht lediglich darin, durchgängig die CPU auszulasten. Bezüglich Arbeitsspeicher wird ein Block mit einer zur Laufzeit definierten Größe allokiert. Dessen Bytes werden periodisch mit allen Zahlen zwischen 0 und 112 befüllt und dann der XXH3-Hash dieses Blocks ermittelt. XXH3 ist ein Hashalgorithmus, dessen Laufzeit primär von der Geschwindigkeit des Arbeitsspeichers abhängt [7].

Die zu untersuchende Zahl 141464683 und die 1 GiB große Allokation werden zur Laufzeit definiert, sodass die respektiven Aufgaben nicht zum Zeitpunkt des Kompilierens gelöst werden. Die Messung erfolgt per RPC von außen. Dieser RPC enthält auch ebengenannte Parameter. Dann wird die aktuelle Systemzeit erfasst, die Aufgabe gelöst, deren Dauer bestimmt und zuletzt die Antwort gesendet. Die gemessene Dauer umfasst daher nicht den Netzwerkverkehr. Es wurden 60 Messungen pro konfigurierter Belastung im Hintergrund durchgeführt. Zwischen einzelnen Messungen wurde jeweils eine Sekunde gewartet.

Bezüglich der CPU-Auslastung wurde zunächst das Verhalten getestet, wenn sowohl der `stress-ng` Pod als auch der untersuchte Pod lediglich geeignete Requests spezifizieren. Für den `stress-ng` Pod sind das 16000 MilliCPU multipliziert mit der prozentualen konfigurierten Auslastung. Für den zu untersuchenden Prozess haben sich 900 MilliCPU durch Messung bei sonstigem Leerlauf des Knotens ergeben. In einem zweiten Versuch wurden auch den Requests identische Limits gesetzt. Bei beiden Varianten ließ sich der zu untersuchende Pod ab 92% Auslastung nicht mehr gemeinsam mit dem `stress-ng` Pod auf einem Knoten platzieren. Diese Grenze ist in Abbildung 5.13 durch die orangene Linie gekennzeichnet. Um das Verhalten bei höheren Auslastungen zu erfassen, wurden die Requests des `stress-ng` Pods bei 91% Auslastung festgesetzt, der Prozess selbst aber mit stärkerer Auslastung konfiguriert.

Gesucht wird die Auslastung, bevor die Streuung der Bearbeitungszeit als auch der Median jener ausreichend zunimmt. Mit steigender CPU-Auslastung ist aufgrund der CPU-Contention eine Verschlechterung der Bearbeitungszeit zu erwarten. Eine weitere Erwartung ist, dass bei sehr hoher Auslastung ($\geq 95\%$) und ausschließlicher Anwendung von Requests die Bearbeitungszeit deutlich schlechter wird, da keine obere Grenze für die CPU-Nutzung in den cgroups konfiguriert wird, während bei der Nutzung von Limits keine Verschlechterung zu erwarten ist, da der `stress-ng` Prozess durch die CFS-Bandwidth-Control unterbrochen wird. Die Ergebnisse sind in Abbildung 5.13 abgetragen. An der Abszisse ist die durch den `stress-ng` Prozess erzeugte Auslastung notiert.

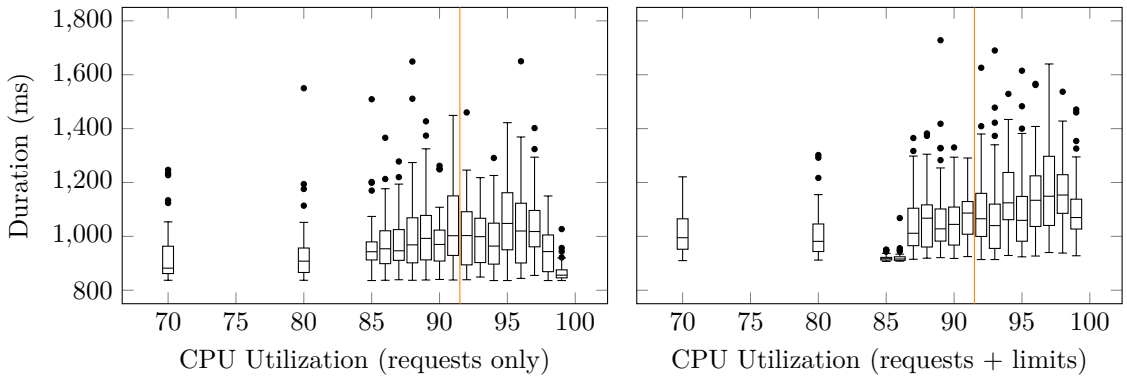

Abbildung 5.13: Boxplot der Berechnungszeit bei prozentualer CPU-Auslastung im Hintergrund

Beide oben formulierten Erwartungen können nicht durch die Messung bestätigt werden. Im Fall der ausschließlichen Anwendung von Requests verbessert sich die Bearbeitungszeit des untersuchten Pods ab 98% Auslastung. Wie in diesem Abschnitt eingangs angemerkt, erschweren die zahlreichen Abstraktionsschichten (cgroups, Betriebssystem, Hypervisor, unbekannte physische Hardware) die Interpretation und Validität. Tatsächlich schwanken die Messwerte in Abhängigkeit des Zeitpunkts der Durchführung. Ein Ansatz zur Erklärung der Verbesserung ab 98% Auslastung kann in der Bestrebung des CFS liegen, jedem Prozess gleich viel CPU-Zeit zuzuordnen. Da zwischen den Messungen eine Sekunde gewartet wird, lässt sich vermuten, dass der `stress-ng` Prozess innerhalb dieser Sekunde viel CPU-Zeit akkumuliert. Trifft der RPC nun am untersuchten Prozess ein, kann dieser ungestört bearbeitet werden, da viel CPU-Zeit „aufzuholen“ ist.

Weiterhin wurde eine Messung für die Arbeitsspeicherauslastung durchgeführt. Die erfassten Bearbeitungszeiten sind in Abbildung 5.14 dargestellt. Ähnlich der CPU-Auslastung ist die Annahme, dass mit steigender Auslastung die Bearbeitungszeit steigt, da mehr Zeit in die Verwaltung der Pages investiert werden muss.

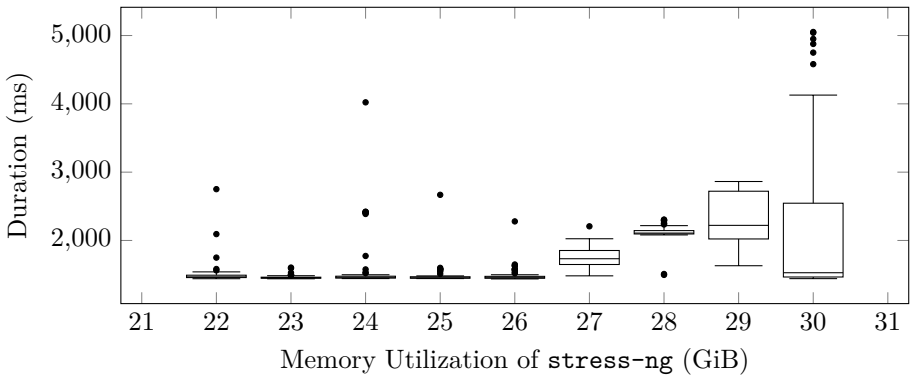

Abbildung 5.14: Boxplot der Berechnungszeit bei Arbeitsspeicherauslastung im Hintergrund

An der Abszisse ist die Belegung des Arbeitsspeichers durch den `stress-ng` Prozess abgetragen. Abseits der Verbesserung des Medians bei 30 GiB Belegung, nimmt ab

einschließlich 27 GiB Belegung der Median und die Streuung zu. Der Messreihe nach liegt die maximale tolerable Belegung durch `stress-ng` zwischen 26 und 27 GiB. Zuzüglich der 1 GiB Allokation des untersuchten Prozesses ergibt sich damit die maximale tolerable Auslastung von $27 - 28$ GiB. Offen bleibt, ob in Abhängigkeit von der Größe des Arbeitsspeichers eines Knotens absolut $4 - 5$ GiB oder relativ $12.5\% - 15.6\%$ nicht verplant werden sollten. Unabhängig von der Antwort kann in Kapitel 6 nur eine relative Angabe verwendet werden, weshalb auf eine weitere Analyse verzichtet wird.

5.4 CPU-Throttling

In Abschnitt 1.4 wurde das CPU-Throttling als Effekt aufgezeigt, welcher durch die Konfiguration von Limits bezüglich der CPU-Zeit hervorgerufen wird. Es bietet sich daher an kurz die Relevanz des Throttlings zu diskutieren und Lösungsideen aufzuzeigen. Der messbare Einfluss des Throttlings ist, dass die Ausführung von RPCs mehr Zeit beansprucht als ohne Throttling, was auch in Abbildung 5.13 zu erkennen ist. Ob sich die Minimierung von Throttling lohnt, hängt davon ab, wie mit den betroffenen Containern interagiert wird. Interagiert der Endanwender mit einem in Kubernetes bereitgestellten IT-Service, sollten dessen APIs und zugehörige Datenbanken möglichst wenig Throttling erfahren, da die Verzögerung durch das Throttling die Nutzererfahrung negativ beeinflusst. Bezogen auf Converged Cloud betrifft das die OpenStack APIs. (System-)Komponenten, welche Arbeit im Hintergrund verrichten, müssen bezüglich Throttling eher nicht optimiert werden, da mit diesen selten interaktiv interagiert wird. Beispielsweise gibt es in den Clustern der Converged Cloud eine Komponente, die vollautomatisch Wartungsarbeiten an Knoten plant und durchführt. Solange genug CPU-Zeit zur Erfüllung dieser Aufgabe bereitsteht, sind Verzögerungen durch Throttling irrelevant.

Aus Gleichung 1.20 lässt sich ableiten, dass die Reduktion von Threads beziehungsweise Prozessen innerhalb eines Containers zu weniger Throttling führt. Serielles Abarbeiten führt zu einer kürzeren durchschnittlichen Bearbeitungsdauer aller RPCs, anstatt langer Bearbeitungszeiten aller RPCs bei paralleler Abarbeitung mit starkem Throttling. Schematisch ist dies in Abbildung 5.15 für zwei RPCs, deren Bearbeitung mit orangenen und blauen Blöcken visualisiert ist, mit $t_c = 500ms, t_p = 100ms$ und $t_q = 50ms$ dargestellt.

Der Vorteil der seriellen Bearbeitung besteht darin, dass das Quota unfair auf die RPCs aufgeteilt wird. Das verkleinert die Wartezeit pro Periode und beendet den ersten RPC früher als bei paralleler Bearbeitung. Die Umsetzbarkeit dieser Empfehlung hängt davon ab, ob Zugriff auf den Quellcode der enthaltenen Applikation besteht und falls dies der Fall ist, wie praktikabel die Anpassung ist[4].

[4]Für Anwendungen, welche in Go geschrieben sind, ist beispielsweise das Setzen der Umgebungsvariable `GOMAXPROCS=1` zureichend, um Parallelität zu reduzieren.

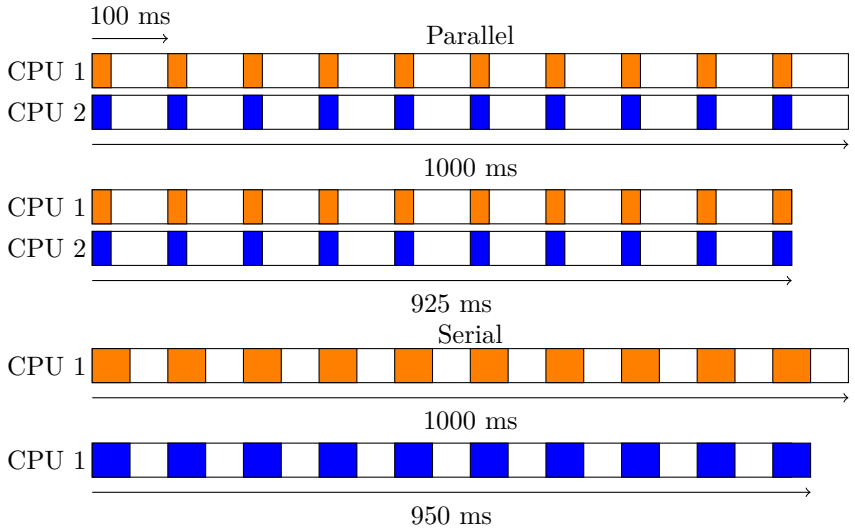

Abbildung 5.15: CPU-Throttling bei paralleler und serieller Bearbeitung zweier RPCs

Alternativ kann auf dem Kubelet die sogenannte „Static CPU Management Policy“ konfiguriert werden. Container, die ganzzahlige CPU-Kerne beanspruchen, und Teil eines Pods in der Guaranteed-QoS-Klasse[5] sind, erhalten dann entsprechend viele dedizierte Kerne, welche von anderen Pods nicht genutzt werden können [35]. Dies wird mittels des `cpuset.cpus` cgroup-Schlüssels realisiert [9]. Es gibt die Möglichkeit per Konfiguration CPU-Kerne zu spezifizieren, welche nicht mit dem `cpuset.cpus` Schlüssel an die Container distribuiert werden, um diese Systemprozessen vorzuhalten. Jene sind aus Perspektive des Kube-Schedulers nicht beplanbar, weshalb diese Reservierung die Kapazität im Cluster senkt [35].

Die CFS-Bandwidth-Control kann aufgrund des Pinnings entfallen und damit auch das Throttling. Die Isolation ist jedoch auf die vom Kubelet verwalteten Container beschränkt, da das CPU-Pinning innerhalb der vom Kubelet verwalteten cgroups erfolgt. Andere Prozesse können weiterhin auf den dedizierten Kernen geplant werden [35]. Praktisch setzt Kubernetes 1.27 noch den `cpu.max` cgroup-Schlüssel, wodurch die Bandwidth-Control aktiviert bleibt. Trotz dessen wird die Anzahl der nutzbaren Kerne beschränkt, was das verstärkte Throttling durch hohe Parallelität innerhalb eines Containers reduziert.

Zuletzt bleibt noch die rein vom Bedarf her unnötige Erhöhung des CPU-Limits, um Throttling zu minimieren. Die nicht genutzten Ressourcen müssen dann in Kauf genommen werden. Die Anpassung der Periodendauer ist nicht zielführend. Eine Verkürzung erhöht den Aufwand zur Verwaltung des Quotas und bedingt mehr Prozesswechsel. Eine große Verlängerung untergräbt den Zweck der Isolation, da ein Prozess erst nach längerer Zeit durch die CFS-Bandwidth-Control entfernt werden kann.

[5] Alle Requests und Limits aller Container sind identisch.

6 Demonstration

Abschnitt 5.1 hat verschiedene Bin-Packing-Heuristiken ausführlich betrachtet. Das Ergebnis der Simulation war, dass die Aktivierung von Bin-Packing im Kube-Scheduler bei doppelter Gewichtung der ausgeglichenen Auslastung zwischen Ressourcendimensionen innerhalb eines Knotens eine effektive Kombination darstellt. Um diese in einem Cluster zu nutzen, ist lediglich die Konfiguration des Schedulers zu ändern. Aus Abschnitt 5.3 ist ebenso bekannt, dass Knoten nur bis etwa 85% ausgelastet werden können, bevor negative Effekte aufgrund der hohen Auslastung auftreten. Ebenso wurde in Kapitel 4 diskutiert, dass auf die Berücksichtigung von lokal vorliegenden Containerabbildern verzichtet werden kann. Alle diese Erkenntnisse lassen sich in folgenden Ausschnitt der Kube-Scheduler Konfiguration überführen:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
- plugins:
    score:
      disabled:
      - name: ImageLocality
      enabled:
      - name: NodeResourcesFit
        weight: 1
      - name: NodeResourcesBalancedAllocation
        weight: 2
  pluginConfig:
  - name: NodeResourcesFit
    args:
      scoringStrategy:
        type: RequestedToCapacityRatio
        requestedToCapacityRatio:
          shape:
          - utilization: 0
            score: 1
          - utilization: 85
            score: 10
          - utilization: 86
            score: 0
          - utilization: 100
            score: 0
```

Abbildung 6.1: Ausschnitt der optimierten Konfiguration des Kube-Schedulers

Der Schlüssel `shape` definiert eine stückweise lineare Funktion $f(util)$, welche der Auslastung einer Dimension einen Score zuordnet. Für einen Knoten ist der Score dieses spezifischen Plugins dann gegeben durch den Durchschnitt des Funktionswerts für alle Ressourcendimensionen: $S1 = \frac{1}{d} \sum_{i=0}^{d} f(util_i)$ [37]. Damit steigt die „Attraktivität“ eines Knotens bis zu einer Auslastung von 85% in einer Dimension an und nimmt danach ab. Dadurch wird Bin-Packing durchgeführt, wobei die Knoten weitgehend nur bis zur Auslastungsgrenze Pods zugeordnet bekommen.

Die Effektivität dieses Verfahrens soll überprüft werden, in dem die aufgezeigte Konfiguration in einem Cluster eingesetzt wird. Dabei wird die Auslastung der Knoten vor und nach der Konfigurationsänderung erfasst. Untersucht wird die Änderung im Metal-Cluster in der Region qa-de-1, welche der Qualitätssicherung dient. Im Gegensatz zu den anderen Kapiteln dieser Arbeit wird nicht das Scaleout-Cluster untersucht, da während des Zeitraums der Durchführung des Experiments eine ressourcenintensive Applikation und entsprechend einige Knoten entfernt wurden, was einen Vergleich erschwert.

Die Daten für die Standardkonfiguration des Kube-Schedulers sind am 15. Juni in 15 Minuten Intervallen erfasst worden. Die dargestellte Auslastung bezieht sich auf den resultierenden Tagesdurchschnitt. Danach wurde das Bin-Packing mit der Konfiguration aus Abbildung 6.1 aktiviert und jeder Knoten im Cluster mindestens einmal neugestartet. Das Neustarten führt dazu, dass alle auf dem betreffenden Knoten ausgeführten Pods gelöscht werden und deren Ersatz neueingeplant wird. Die Auslastungsdaten für das Bin-Packing stammen von 27. Juni. Die Verteilung der Knoten bezüglich ihrer Auslastung ist in Abbildung 6.2 dargestellt.

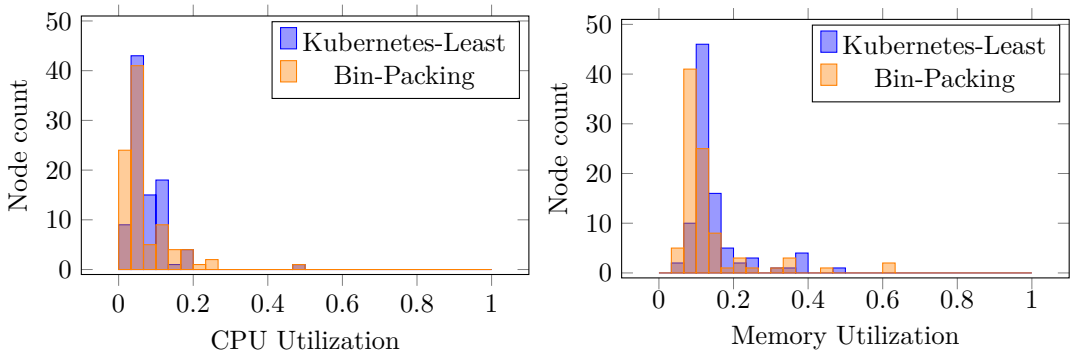

Abbildung 6.2: Durchschnittliche CPU- und Arbeitsspeicherauslastung im Metal-Cluster in qa-de-1

Nach der Aktivierung des Bin-Packings nimmt die Anzahl gering ausgelasteter Knoten zu. Die Pods werden stattdessen auf wenigen stärker ausgelasteten Knoten ausgeführt. Eine Auslastung, die sich 0 annähert kann aufgrund von Pods in DaemonSets, welche immer auf allen Knoten instanziiert werden, nicht erreicht werden. Da die Auslastung des untersuchten Clusters insgesamt klein ist, sind die Effekte des Bin-Packings nicht sonderlich deutlich. Der Scheduler nimmt die Zuordnung anhand der Requests der Pods vor, deren Verteilung in Abbildung 6.3 dargestellt ist.

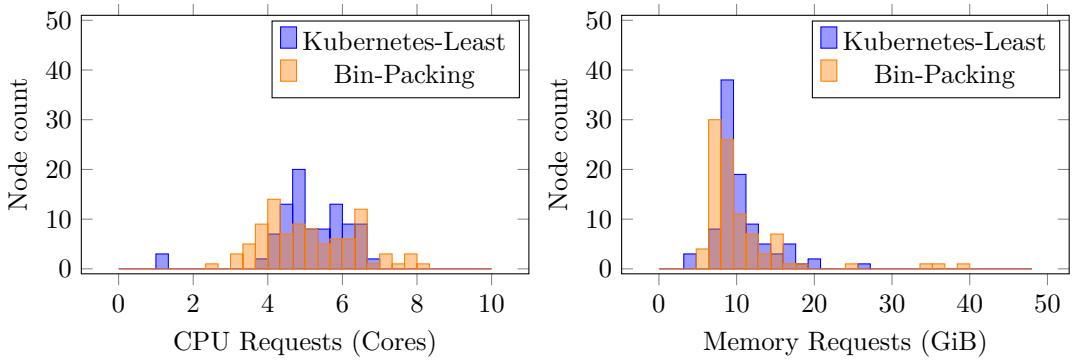

Abbildung 6.3: Durchschnittliche CPU- und Arbeitsspeicherrequests im Metal-Cluster in qa-de-1

Bezogen auf die Requests ist ebenso ersichtlich, dass die Anzahl gering ausgelasteter Knoten zunimmt, da wenige Knoten bis zur Auslastungsgrenze Pods zugeordnet bekommen. Die Auslastungsgrenze bezüglich Arbeitsspeicher, hier 40.8 GiB, wird eingehalten. Gerade bezüglich der CPU-Zeit zeigt sich auch die in Abschnitt 3.2 diskutierte Diskrepanz zwischen dem tatsächlichen und spezifizierten Bedarf, weshalb sich die Verteilung der tatsächlichen Auslastung nach Aktivierung des Bin-Packings nur geringfügig ändert. Dies ist deutlich in den Streudiagrammen in Abbildung 6.4 ersichtlich. Damit das Bin-Packing Auslastung auf Knoten bündelt, muss die Ressourcenspezifikation der Nutzung entsprechen.

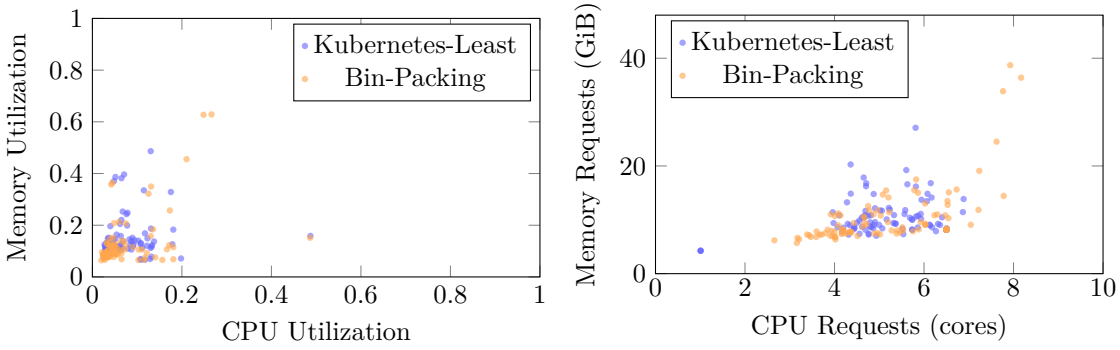

Abbildung 6.4: Streudiagramm der Auslastung und Requests im Metal-Cluster in qa-de-1

Die Auslastungsgrenze bezüglich der CPU wird von wenigen Knoten überschritten. Die ausführenden Knoten verfügen über zehn Kerne. Zum Zeitpunkt der Durchführung wurde auch die „Static CPU Management Policy“ evaluiert, weshalb nur acht Kerne im Scheduling berücksichtigt wurden. Folglich ergeben sich 6.8 Kerne als Grenze. Die Knoten weisen rund drei Kerne als kleinste Summe der CPU-Requests der Pods auf. Allerdings enthält das Cluster 13 Pods, welche vier Kerne in ihren Requests spezifizieren. Deshalb akkumuliert ein Knoten mit einem solchem Pod etwa sieben Kerne an CPU-Requests, um den Pod überhaupt auszuführen. Letztlich werden genauere Requests auch dieses Verhalten reduzieren.

7 Evaluation

7.1 Handlungsempfehlung

Eingangs wurde die Frage gestellt, welche Methoden zur Anpassung des Schedulings in Kubernetes-Clustern existieren, wie aufwendig deren Implementierung ist und wie gut diese die Zuordnungen von Containern zu Knoten verbessern. Dafür wurde die Thematik des Ressourcenmanagements (Abschnitt 5.2), welches die Eingabedaten für das Vector-Bin-Packing (Abschnitt 5.1) bereitstellt, und die Isolationsmechanismen zur Laufzeit auf den Knoten (Abschnitte 5.3 und 5.4) diskutiert, um die Auslastung der Knoten zu erhöhen, ohne die Performance der Applikationen negativ zu beeinflussen. Abschließend sollen die gewonnenen Erkenntnisse in einer Handlungsempfehlung zusammengefasst werden.

Die akkurate Spezifikation von Requests und Limits ist entscheidend für die Performance der Container selbst als auch für die Wirksamkeit des folgenden Bin-Packings. In Abschnitt 3.2 wurde aufgezeigt, dass die manuelle Spezifikation nicht dem realen Verbrauch an Ressourcen entspricht. Abweichungen existieren in beide Richtungen. Pods, die weniger Ressourcen spezifizieren als sie verbrauchen, können zu einer Überbeanspruchung von Knoten führen. Pods, welche mehr Ressourcen spezifizieren als sie real benötigen, belegen unnötig Kapazität, welche weitere Pods nicht mehr nutzen können. Zusätzlich wird die manuelle Spezifikation von den Personen, die Applikationen in Kubernetes bereitstellen, als aufwendig und herausfordernd wahrgenommen.

Demnach ist eine Automatisierung des Ressourcenmanagements sehr empfehlenswert. Kubernetes bietet mit dem VPA eine Komponente, welche diese Funktionalität implementiert. Eine Bereitstellung des VPA ist aus technischer Perspektive überschaubar. Allerdings ist der Adaptionsprozess anspruchsvoll, da den Personen, welche Dienste in Kubernetes betreiben, die Kontrolle über einen kritischen Bestandteil der Bereitstellung entzogen wird. In Bezug auf die Stabilität der Dienste, ergibt sich die geäußerte Sorge vor Diskontinuitäten, sowohl durch zu kleine Abschätzungen als auch Abschätzungen, welche die Kapazität der Knoten übersteigen. Ebenso darf die automatische Skalierung nicht zu träge reagieren, damit abweichende Abschätzungen hinreichend schnell korrigiert werden können. Für den VPA bietet es sich daher konkret an, die 24-stündige Halbwertszeit der Histogramme mit exponentiellem Verfall zu kürzen. Um der Überbewertung des Arbeitsspeicherbedarfs

entgegenzuwirken, bietet es sich an den Pufferfaktor von 1.15 leicht zu senken, sofern die Häufigkeit von OOM-Situationen vertretbar bleibt.

Im Rahmen der Zeitreihenanalyse konnte gezeigt werden, dass bessere Modelle für die Vorhersage des Ressourcenbedarfs existieren, als jenes welches der VPA implementiert. Aufgrund der großen Diskrepanz zwischen der realen Nutzung und der Ressourcenspezifikation der Container ist zunächst jede Automatisierung besser als ein Verbleib beim manuellen Prozess. Langfristig lohnt sich die Untersuchung und Entwicklung besserer Modelle für große Cluster, trotz des hohen Entwicklungsaufwands. Bereits die in dieser Arbeit untersuchten ARIMA-Modelle erzeugen rund 50% weniger Überbewertung bei gleichzeitig weniger Mangelsituationen gegenüber der Abschätzung des VPA bezogen auf den untersuchten Datensatz beim Arbeitsspeicher. Gegenüber den anderen untersuchten Bereichen ist im automatischen Ressourcenmanagement das größte Potential zur Einsparung von Ressourcen.

An die Spezifikation der Ressourcen schließt sich die Zuordnung von Containern zu Knoten an. Der Kube-Scheduler führt in der Standardkonfiguration kein Bin-Packing durch, welches sich durch eine entsprechende Konfiguration aktivieren lässt. Aus den durchgeführten Simulationen ergibt sich die in Abbildung 6.1 dargestellte als zu empfehlende Konfiguration. Die balancierte Auslastung der Ressourcen innerhalb der Knoten ist maßgebend, um eine hohe Auslastung zu erreichen. Der Aufwand zur Anpassung der Konfiguration ist sehr gering und deshalb unbedingt durchzuführen. Ein positiver Effekt dieser Maßnahme auf die Auslastung hängt zwar stark von der Genauigkeit der Ressourcenspezifikationen ab, jedoch hat der experimentelle Einsatz keine operativen Probleme bei abweichenden Spezifikationen hervorgerufen. Das mit der Rekonfiguration verbundene Risiko ist gering. Ein operativer Vorteil des Bin-Packings ist, dass überflüssige Knoten einfacher erkennbar sind. Überflüssige Knoten weisen nach Aktivierung des Bin-Packing eine kleine nahezu identische Auslastung auf, welche durch Pods hervorgerufen wird, welche DaemonSets auf allen Knoten instanziieren. Ohne Bin-Packing werden die restlichen Pods gleichmäßig verteilt, was die Hintergrundauslastung durch DaemonSets „verschleiert“, wodurch die Bestimmung der erforderlichen Kapazität erschwert wird.

Auch beim Vector-Bin-Packing konnte mit der Vector-Dot-Heuristik ein Verfahren identifiziert werden, welches bei bestimmten Verteilungen der Ressourcen etwas besser packt als die im Kube-Scheduler implementierten Heuristiken. Da die Verbesserung im niedrigen einstelligen Prozentbereich liegt, lohnt sich die Entwicklung, Pflege und Paketierung eines Scheduler-Plugins nur, wenn die Cluster viele hundert Knoten umfassen. Gleiches gilt für komplett eigenentwickelte Scheduler. Insgesamt schätzt der Autor das weitergehende Optimierungspotential in Bezug auf das Vector-Bin-Packing als gering ein, weshalb nach Aktivierung des Bin-Packings das Ressourcenmanagement fokussiert werden sollte.

Zuletzt sind die Isolationsmechanismen auf den Knoten zu beachten. Das Kubelet stellt nicht sicher, dass die in den Requests spezifizierte Menge an Arbeitsspeicher eines Containers tatsächlich genutzt werden kann. Die Aktivierung des `MemoryQoS`-

Features schließt diese Lücke, was abseits der Tatsache, dass an der Funktionalität noch entwickelt wird, zu empfehlen ist. Die „Static CPU Management Policy“ kann Throttling reduzieren, ohne das CPU-Limit erhöhen zu müssen. Dies unterstützt die stärke Auslastung der Knoten, womit eine Nutzung für ressourcenintensive Pods, bei denen Throttling relevant ist, zu empfehlen ist. Durch Aufrunden eventueller Vorhersagen bezüglich der CPU-Zeit, können diese gemeinsam mit der „Static CPU Management Policy“ genutzt werden.

7.2 Ausblick

Die in dieser Arbeit durchgeführte Untersuchung zu den ARIMA-Modellen und den Vorhersagen des VPA lässt sich weiter ausbauen. So ist die Wahl der Faktoren zur Anpassung der Vorhersagen anhand eines analytisch formulierten Optimierungsziels durchzuführen. Damit kann bezüglich des konkreten Optimierungsziels das beste Verfahren exakt identifiziert werden. Zusätzlich ist die Analyse auf weitere Datensätze auszuweiten, um eine allgemeinere Anwendbarkeit nachzuweisen. Um Vergleichbarkeit zwischen Untersuchungen in diesem Bereich zu gewährleisten, sollten repräsentative und standardisierte Datensätze geschaffen werden [5, S. 31]. Ebenso ist der Code weiter auszubauen, um die Vorhersagen mittels der ARIMA-Modelle in Kubernetes zu integrieren. Dann kann das in dieser Arbeit vorgeschlagene Verfahren in realen Clustern experimentell evaluiert werden.

Allgemein stammen die in der Zeitreihenanalyse verwendeten ARIMA-Modelle aus der Zeit um 1970 [16, S. 1]. Seit einigen Jahren gibt es ein starkes Forschungsinteresse im Bereich des maschinellen Lernens, weshalb weitere Verfahren im Bereich der Zeitreihenanalyse relevant werden. Kumar und Singh [17] zeigen ein Verfahren auf, welches die wiederholte Parametrisierung von ARIMA-Modellen vermeidet, indem die Vorhersagen anhand des bisher realisierten Vorhersagefehlers adjustiert werden. Dadurch kann mehr Zeit für die initiale Modellwahl und Parametrisierung verwendet werden. Fundamental verschiedene Verfahren, wie neuronale Netze und Gradient-Boosted-Trees, sind auch Untersuchungen bezüglich der Anwendbarkeit in Kubernetes und im Cloud-Umfeld wert.

Eine zu lösende Herausforderung ist die ganzheitliche Zusammenarbeit der verschiedenen Komponenten. Für die in dieser Arbeit thematisierten Aspekte gibt es bereits vertretbare Lösungen, jedoch funktionieren diese isoliert. So berücksichtigt der VPA nicht eigenständig die Kapazität der Knoten im Cluster und setzt durchaus Ressourcenspezifikationen, welche deren Kapazität übersteigen. Die betroffenen Pods verbleiben dann in einem nicht-planbaren Zustand. Um die zulässigen Knoten zu ermitteln, müsste der VPA mit dem Kube-Scheduler zusammenarbeiten. Im Rahmen der Vorhersage ist auch eine Einbeziehung des CPU-Throttlings oder weiterer anwendungsspezifischer Metriken wünschenswert, falls jene für den spezifischen Pod relevant sind. Interessant ist auch eine Abschätzung für die Größe des erforderlichen

Page-Cache eines Prozesses, welche in der Grenze für den Arbeitsspeicher berücksichtigt werden kann [48, S. 3].

Weiterhin betrachtet der Kube-Scheduler nur statische Requests und Limits. Änderungen im zukünftigen zeitlichen Verlauf, die mittels der Zeitreihenanalyse zumindest kurzfristig gewonnen werden können, finden keinen Einfluss bei der Zuordnung von Pods zu Knoten. Das Ausnutzen entgegengesetzter Schwankungen kann die Auslastung weiter optimieren. Um Grenzen der cgroups zur Laufzeit anzupassen, wird am `InPlaceVerticalScaling`-Feature gearbeitet. Dieses funktioniert allerdings nur auf der Ebene von Pods, welche kurzlebige Entitäten sind. Nach der Löschung eines angepassten Pods wird ein neuer mit den ursprünglichen Requests und Limits erzeugt, da das übergeordnete Konstrukt, wie Deployments, StatefulSets und DaemonSets, nicht geändert wird.

Um Georedundanz zu gewährleisten oder Kosten weiter zu optimieren, kann der Betrieb mehrerer Cluster in verschiedenen Rechenzentren bei verschiedenen Infrastrukturanbietern relevant sein. Das Scheduling ist dann nicht mehr ausschließlich lokal für einen Cluster zu optimieren, sondern ganzheitlich über alle Cluster hinweg. Dabei sind die Abrechnungsmodelle, Kapazität und Hardwareperformance der Anbieter einzubeziehen [5, S. 30].

Zuletzt sind die behandelten Themen nicht nur in Kubernetes relevant. Jedes System zur Container-Orchestrierung benötigt einen Prozess zur Bedarfsabschätzung, um die Zuordnung von Containern zu Knoten zu gestalten und Isolationsmechanismen auf den Knoten. Eine nahezu identische Problematik besteht bei der Zuordnung von virtuellen Maschinen zu physischen Servern. Verändert man die Perspektive noch weiter nach außen, lassen sich die Themen auch bei der Gestaltung von Rechenzentren finden. Je nach Quelle wird die durchschnittliche Auslastung eines Rechenzentrums zwischen 6% und 20% geschätzt. An der oberen Grenze werden CPU-Auslastungen zwischen 25% und 35%, bezüglich Arbeitsspeicher um die 40%, genannt [8, S. 127–128]. In diesem Bereich ist die Zeitreihenanalyse zur langfristigen Kapazitätsplanung als auch eine geeignete Zuordnung von Großkunden zu Rechenzentren relevant. Obwohl die Forschung bezüglich Scheduling in der Cloud nun über zehn Jahre zurückreicht, sind Themen in diesem Umfeld aufgrund der Kosten- und Energieeffizienz weithin bedeutsam und die Erkenntnisse in die Praxis zu überführen.

Literatur

[1] E. Anderson u. a. *LAPACK Users' Guide*. Third Edition. Philadelphia, 1999.

[2] Christian Baun u. a. *Cloud Computing. Web-basierte dynamische IT-Services*. Heidelberg, 2011.

[3] Peter J. Brockwell und Richard A. Davis. *Introduction to Time Series and Forecasting*. 2016.

[4] *cAdvisor*. URL: `https://github.com/google/cadvisor` (besucht am 10. 03. 2023).

[5] Carmen Carrión. „Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges“. In: *ACM Computing Surveys* 55 (2022).

[6] Cloud Native Computing Foundation. *CNCF Kubernetes Project Journey Report*. 2019. URL: `https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Kubernetes_Project_Journey_Report.pdf` (besucht am 31. 03. 2023).

[7] Yann Collet. *xxHash*. URL: `https://cyan4973.github.io/xxHash/` (besucht am 21. 06. 2023).

[8] Christina Delimitrou und Christos Kozyrakis. „Quasar: Resource-Efficient and QoS-Aware Cluster Management“. In: *ACM SIGARCH Computer Architecture News* 42 (2014), S. 127 –144.

[9] Tejun Heo. *Control Group v2*. 2015. URL: `https://docs.kernel.org/admin-guide/cgroup-v2.html` (besucht am 02. 03. 2023).

[10] Wei Huang, Xin Li und Zhuzhong Qian. *An Energy Efficient Virtual Machine Placement Algorithm with Balanced Resource Utilization*. 2013.

[11] Jörg A. Hölzing. *Die Kano-Theorie der Kundenzufriedenheitsmessung. Eine theoretische und empirische Überprüfung*. 2008.

[12] Alexandru Iosup, Radu Prodan und Dick Epema. „IaaS Cloud Benchmarking: Approaches, Challenges, and Experience“. In: *Cloud Computing for Data-Intensive Applications*. Hrsg. von Xiaolin Li und Judy Qiu. 2014, S. 83–104.

[13] Paul Johannesson und Erik Perjons. *An Introduction to Design Science*. 2014.

[14] David S. Johnson. „Bin Packing“. In: *Encyclopedia of Algorithms*. Hrsg. von Ming-Yang Kao. 2008, S. 94–97.

[15] George Kapetanios. „A note on an iterative least-squares estimation method for ARMA and VARMA models“. In: *Economics Letters* 79 (2003), S. 305–312.

[16] Gebhard Kirchgässner, Jürgen Wolters und Uwe Hassler. *Introduction to Modern Time Series Analysis*. 2013.

[17] Jitendra Kumar und Ashutosh Kumar Singh. „Cloud datacenter workload estimation using error preventive time series forecasting models“. In: *Cluster Computing* 23 (2020), S. 1363–1379.

[18] Krishan Kumar und Manish Kurhekar. „Economically Efficient Virtualization Over Cloud Using Docker Containers“. In: *IEEE International Conference on Cloud Computing in Emerging Markets* (2016), S. 95–100.

[19] William Leinberger, George Karypis und Vipin Kumar. „Multi-Capacity Bin Packing Algorithms with Applications to Job Scheduling under Multiple Constraints“. In: *Proceedings of the 1999 International Conference on Parallel Processing* (1999), S. 404–412.

[20] Marko Lukša. *Kubernetes in Action*. Shelter Island, 2018.

[21] *metrics-server*. URL: `https://github.com/kubernetes-sigs/metrics-server` (besucht am 10. 03. 2023).

[22] Mayank Mishra und Anirudha Sahoo. „On Theory of VM Placement. Anomalies in Existing Methodologies and Their Mitigation Using a Novel Vector Based Approach“. In: *2011 IEEE 4th International Conference on Cloud Computing* (2011), S. 275–282.

[23] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. 2014. URL: `https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf`.

[24] *OpenStack*. URL: `https://www.openstack.org/software/` (besucht am 12. 07. 2023).

[25] Rina Panigrahy u. a. *Heuristics for Vector Bin Packing*. 2011. URL: `https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/` (besucht am 17. 04. 2023).

[26] *Prometheus*. URL: `https://prometheus.io/` (besucht am 10. 03. 2023).

[27] Jan Recker. *Scientific Research In Information Systems. A Beginner's Guide*. 2013.

[28] Zeineb Rejiba und Javad Chamanara. „Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches“. In: *ACM Computing Surveys* 55 (2022).

[29] Khaldoun Senjab u. a. „A survey of Kubernetes scheduling algorithms“. In: *Journal of Cloud Computing: Advances, Systems and Applications* 87 (2023).

[30] Asser N. Tantawi und Malgorzata Steinder. „Autonomic cloud placement of mixed workload: An adaptive bin packing algorithm“. In: *IEEE International Conference on Autonomic Computing (ICAC)* (2019), S. 187 –193.

[31] *Thanos*. URL: `https://thanos.io/` (besucht am 10. 03. 2023).

[32] The kernel development community. *CFS Bandwidth Control*. 2023. URL: `https://docs.kernel.org/scheduler/sched-bwc.html` (besucht am 10. 03. 2023).

[33] The kernel development community. *CFS Scheduler*. 2023. URL: https://docs.kernel.org/scheduler/sched-design-CFS.html (besucht am 17. 03. 2023).

[34] The kernel development community. *Memory Management: Concepts overview*. 2023. URL: https://docs.kernel.org/admin-guide/mm/concepts.html (besucht am 02. 03. 2023).

[35] The Kubernetes Authors. *Control CPU Management Policies on the Node*. 2023. URL: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/ (besucht am 09. 06. 2023).

[36] The Kubernetes Authors. *Device Plugins*. 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/ (besucht am 03. 04. 2023).

[37] The Kubernetes Authors. *Kubernetes Source Code*. Version 1.27.0. 11. Apr. 2023. URL: https://github.com/kubernetes/kubernetes/tree/v1.27.0 (besucht am 12. 04. 2023).

[38] The Kubernetes Authors. *Node-pressure Eviction*. 2023. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/ (besucht am 03. 04. 2023).

[39] The Kubernetes Authors. *Pod Quality of Service Classes*. 2023. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/ (besucht am 03. 04. 2023).

[40] The Kubernetes Authors. *Resource Management for Pods and Containers*. 2023. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (besucht am 03. 04. 2023).

[41] The Kubernetes Authors. *Scheduler Configuration*. 2022. URL: https://kubernetes.io/docs/reference/scheduling/config/ (besucht am 04. 04. 2023).

[42] The Kubernetes Authors. *Scheduling Framework*. 2022. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (besucht am 04. 04. 2023).

[43] The Kubernetes Authors. *Vertical Pod Autoscaler Source Code*. Version 0.14.0. 20. Juni 2023. URL: https://github.com/kubernetes/autoscaler/tree/cd5ed0be530e2313f762e30042cb324b768507f6 (besucht am 09. 08. 2023).

[44] Odin Ugedal und Rakesh Kumar. „Mitigating Unnecessary Throttling in Linux CFS Bandwidth Control“. In: *IEEE 34th International Symposium on Computer Architecture and High Performance Computing* (2022).

[45] Nils Urbach und Frederik Ahlemann. *IT-Management im Zeitalter der Digitalisierung. Auf dem Weg zur IT-Organisation der Zukunft*. Heidelberg, 2016.

[46] Tim Xu und David Porter. *KEP-2570: Support Memory QoS with cgroups v2*. 2023. URL: https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/2570-memory-qos/README.md (besucht am 09. 08. 2023).

[47] Yongkang Zhang u. a. „Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly“. In: *SoCC '22: Proceedings of the 13th Symposium on Cloud Computing* (2022).

[48] Zhenyun Zhuang u. a. „Taming memory related performance pitfalls in linux Cgroups“. In: *2017 International Conference on Computing, Networking and Communications* (2017).

Anlagen

A.1 Fragen für die Experteninterviews

Cluster Nutzer

- Which applications do you run in which clusters of Converged Cloud?
- How is the performance characteristic of the application? Does resource consumption change fast or slow over time? Is it constant? Any specialties regarding the application startup?
- Are you satisfied with how your payload is scheduled? What could be improved in that regard?
- Do you know about incidents which were caused or prolonged by scheduling issues?
- Have you tried to use the vertical pod autoscaler? Which experience did you make? What holds you back?
- Would like you like to stay in control of scheduling or prefer more automatic approach to scheduling? Why?
- Which additional constraints would you like to be considered during scheduling (e.g. bandwidth, IOPS)?

Cluster Administratoren

- Are you satisfied how the workload is distributed within the clusters? What makes an optimal distribution for you?
- Which (kind of) application causes the most trouble caused by suboptimal placement?
- Do you know about incidents which were caused or prolonged by scheduling issues?
- Have you tried to use the vertical pod autoscaler? Which experience did you make?
- Which additional constraints would you like to be considered during scheduling (e.g. bandwidth, IOPS)?

A.2 Protokolle der Experteninterviews

Neutron

Applikationen und Dienste

OpenStack Dienst für Netzwerk: Neutron, verschiedene Komponenten: MariaDB, RabbitMQ, memcached, zentraler API-Server, mehrere Agents für die Interaktion mit der Netzwerk Hardware

Performance Charakteristika

- prinzipiell großer Bedarf an CPU und Arbeitsspeicher
- verhältnismäßig vorhersagbar, Requests und Limits überall gesetzt
- Ressourcenbedarf der API skaliert mit der Größe der Datenbank und Anzahl der Nutzer
- Außer bei der Datenbank, aufgrund des CPU Throttlings
- höherer Ressourcenbedarf beim Starten der Agents, Validieren und Aufräumen der Datenbank

Verbesserungsbedarf bezüglich Scheduling

Größere freie Kapazität auf einzelnen Knoten, sodass Pods mit großem Ressourcenbedarf planbar sind.

Ausgelöste oder verlängerte Störungen der Dienste durch suboptimales Scheduling

Im Rahmen von Wartungsarbeiten an Knoten werden Pods von diesen entfernt, welche dann auf den verbleibenden platziert werden müssen. Trotz insgesamt ausreichend freier Kapazität im Cluster gelingt dies teils nicht.

Erfahrungen bezüglich des Vertical-Pod-Autoscalers

Bisher nicht ausprobiert.

Automatisierung des Ressourcenmanagements bzw. Schedulings

Automatische Skalierung der Ressourcen wäre hilfreich, besonders bei der API.

Weitere für die Planung relevante Ressourcen

Keine direkten Anforderungen, möglicherweise Netzwerk Bandbreite.

Prometheus

Applikationen und Dienste

Applikationen im Bereich des Monitorings: verschiedene Prometheus Instanzen, Thanos Querier und Exporter für Metriken

Performance Charakteristika

Prometheus Instanzen sind sehr intensiv an Arbeitsspeicher, Verhältnis 1 CPU zu etwa 10 GiB RAM.

- tatsächlicher Bedarf hängt von der Nutzung der Cloud Region ab, benötigte Ressourcen wachsen in größeren Cloud Regionen schneller als in Kleinen
- Wachstum des Bedarfs wird reaktiv erfasst, eine Vorhersage wäre nützlich
- Arbeitsspeicher wird erst nach Laden der Daten vollständig genutzt, der Ladevorgang dauert ein bis zwei Stunden

Thanos Querier: Dynamischer Bedarf an Arbeitsspeicher, welcher von den gestellten Anfragen abhängt. Entsprechend komplexe Anfragen können zu OOM-Situationen führen.

Verbesserungsbedarf bezüglich Scheduling

Im Rahmen von Wartungsarbeiten an Knoten werden Pods von diesen entfernt, welche dann auf den verbleibenden platziert werden müssen. Trotz insgesamt ausreichend freier Kapazität im Cluster gelingt dies teils nicht.

Ausgelöste oder verlängerte Störungen der Dienste durch suboptimales Scheduling

Zumindest Verlängerung einer Störung mit Beeinträchtigung der Cloud Nutzer, aufgrund von Wartungsarbeiten.

Erfahrungen bezüglich des Vertical-Pod-Autoscalers

Viele Deployments, folglich hoher Aufwand für das Spezifizieren von Ressourcen. Automatisierung wäre hilfreich. VPA ausprobiert. Der VPA löscht Pods, um die Requests und Limits zu ändern. Beim Herunterskalieren besteht die Sorge vor Restart Loops aufgrund mangelnder Ressourcen.

Automatisierung des Ressourcenmanagements bzw. Schedulings

Automatisierung gewünscht, Verhalten der Container lernen und entsprechende Requests und Limits anwenden. Manuell werden die Ressourcenspezifikationen zu selten aktualisiert, folglich Ressourcen nicht genutzt.

Weitere für die Planung relevante Ressourcen

Netzwerkbandbreite könnte berücksichtigt werden. Weiterhin freie Netfilter-Conntrack-Table-Einträge. Kürzlich gab es eine Störung bei Überlauf jener Tabelle aufgrund von Noisy-Neighbours.

Cluster Administratoren

Verteilung der Auslastung

- In den Metal-Clustern ist die Auslastung der Knoten ungleichmäßig verteilt.
- CPU Requests der Pods liegen oft über der tatsächlichen Nutzung, folglich besteht ein Mangel an CPU aus der Perspektive des kube-schedulers.
- Eine breitere und akkuratere Verwendung von Requests und Limits ist gewünscht.
- Bezüglich Anschaffungskosten sind große Knoten lohnender als Kleinere. Für Wartungsarbeiten und Hochverfügbarkeit sind allerdings kleine Knoten vorteilhafter.

Problematische Applikationen

- Applikationen, die viel Arbeitsspeicher erfordern, werden oft vom OOM-Killer entfernt.
- Einige OpenStack-Komponenten erzeugen beim Starten viel Last. Die Kapazität dafür wird später nicht benötigt.
- Applikationen, die Multithreading fehlerhaft umsetzen und bezüglich CPU-Zeit nicht limitiert sind, beeinträchtigen weitere Anwendungen.

Ausgelöste oder verlängerte Störungen der Dienste durch suboptimales Scheduling

Im Rahmen von Wartungsarbeiten an Knoten werden Pods von diesen entfernt, welche dann auf den verbleibenden platziert werden müssen. Trotz insgesamt ausreichend freier Kapazität im Cluster gelingt dies teils nicht.

Erfahrungen bezüglich des Vertical-Pod-Autoscalers

Die hohe Last beim Starten einiger OpenStack-Komponenten wird von der Heuristik des VPA nicht korrekt abgebildet. Während der Laufzeit der Komponenten werden die abgeschätzten Ressourcen kleiner. Beim Neuerstellen des Pods sind diese dann unzureichend.

Weitere für die Planung relevante Ressourcen

Die Betrachtung von IOPS ist vorstellbar. Diese haben bisher aber mit CPU-Nutzung korreliert. Außerdem ist die Auslastung der Festplatten derzeit vernachlässigbar.

A.3 Antworten der Kano-Umfrage

Question/Person	1	2	3	4	5	6	7	8	9	10	11
If the scheduler could plan CPU time, It would be	5	4	5	4	4	4	4	4	5	5	5
If the scheduler could not plan CPU time, It would be	2	1	4	2	3	3	1	1	4	4	2
If the scheduler could plan memory usage, It would be	5	4	5	4	4	4	4	4	5	5	5
If the scheduler could not plan memory usage, It would be	1	1	4	2	2	1	1	1	4	4	4
If the scheduler could plan network bandwidth, It would be	5	5	5	5	5	5	4	5	5	5	5
If the scheduler could not plan network bandwidth, It would be	2	3	4	3	4	3	1	4	4	4	4
If the scheduler could plan I/O speed, It would be	5	5	5	4	3	4	5	5	5	5	4
If the scheduler could not plan I/O speed, It would be	1	3	4	3	3	3	4	4	4	4	2
If the scheduler would actively move pods around (recreating them), It would be	1	5	2	4	5	1	5	2	5	2	4
If the scheduler would not actively move pods around (recreating them), It would be	5	2	4	3	4	4	4	4	3	5	2
If the scheduler would prefer nodes where container images are already pulled, It would be	3	3	3	5	3	3	3	3	3	3	1
If the scheduler would not consider pulled images, It would be	3	4	4	3	4	4	3	4	3	4	4
If the scheduler would consider optional pod (anti-)affinities / topology spread constraints, It would be	4	4	4	5	5	5	5	4	5	4	5
If the scheduler would not consider optional pod (anti-)affinities / topology spread constraints, It would be	1	1	1	3	1	3	4	1	1	1	1
If the scheduler would consider optional node (anti-)affinities, It would be	3	4	3	5	5	3	5	4	5	4	4
If the scheduler would not consider optional node (anti-)affinities, It would be	3	3	3	3	2	3	4	1	1	1	3

Antworten der KANO-Umfrage: 1:„unakzeptabel“, 2:„gerade so akzeptierbar“, 3:„egal“, 4:„erwartet“, 5:„beeindruckend“

A.4 Vector-Bin-Packing-Simulation

Die folgenden Tabellen geben die durchschnittliche Anzahl an erforderlichen Knoten bei gegebenen Verfahren und gegebenen durchschnittlichen Bedarf der Pods an.

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	111.005333	115.368000	110.344000	110.368667	111.238667
0.333333	115.014667	121.508000	114.109333	114.261333	114.917333
0.250000	112.674000	119.304000	111.707333	111.943333	111.969333
0.200000	110.677333	116.514667	109.619333	109.874000	109.311333
0.166667	109.249333	114.057333	108.143333	108.311333	107.376667
0.142857	108.292667	112.275333	107.248000	107.373333	106.135333
0.125000	107.678667	111.029333	106.723333	106.814000	105.302667
0.111111	107.372000	110.180000	106.486000	106.528000	104.888667
0.100000	107.068000	109.556667	106.318667	106.311333	104.555333
0.083333	106.647333	108.638667	105.982667	105.970667	104.102000
0.062500	106.262000	107.812000	105.858000	105.729333	103.932667
0.050000	105.943333	107.411333	105.654000	105.552667	103.902667
0.040000	105.668000	107.012667	105.412000	105.311333	103.863333

Exponentialverteilte Pods bei 2 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	142.998667	145.970667	142.452667	142.980667	142.966000
0.333333	145.115333	151.546000	144.207333	145.142667	144.599333
0.250000	135.440000	143.229333	134.492000	135.220667	134.114667
0.200000	127.348667	134.850667	126.304000	126.732667	125.128000
0.166667	121.855333	128.574667	120.890667	121.006667	119.068000
0.142857	119.000667	124.856667	118.083333	117.968667	115.770000
0.125000	117.164667	122.269333	116.348667	116.029333	113.723333
0.111111	115.955333	120.418000	115.209333	114.727333	112.372000
0.100000	115.173333	119.204000	114.502667	113.952000	111.513333
0.083333	114.073333	117.596667	113.585333	112.906667	110.488667
0.062500	112.700000	115.658000	112.420667	111.728667	109.406667
0.050000	111.921333	114.537333	111.694667	111.046667	108.822667
0.040000	111.146000	113.645333	110.973333	110.380000	108.419333

Exponentialverteilte Pods bei 4 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	177.817333	178.750000	177.558000	177.809333	177.772000
0.333333	190.950000	196.358667	190.208667	191.483333	190.678667
0.250000	173.101333	181.715333	172.275333	173.973333	172.052667
0.200000	154.631333	164.009333	153.904667	155.328000	152.470667
0.166667	142.413333	151.358667	141.795333	142.924000	139.368667
0.142857	135.384667	143.370667	134.782667	135.490667	131.729333
0.125000	131.272000	138.444000	130.786000	131.118000	127.136667
0.111111	128.506667	134.988667	128.056000	128.324667	124.232667
0.100000	126.736000	132.690000	126.416667	126.445333	122.311333
0.083333	124.210667	129.454000	124.008000	123.900000	119.722667
0.062500	121.282667	125.668000	121.158667	120.953333	116.998000
0.050000	119.408667	123.304000	119.326000	119.132000	115.437333
0.040000	117.703333	121.336667	117.716000	117.497333	114.153333

Exponentialverteilte Pods bei 8 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	118.947333	126.540667	115.545333	115.618000	117.504000
0.333333	112.035333	121.679333	110.591333	110.874000	112.473333
0.250000	108.814667	116.938667	107.788667	107.948000	108.376667
0.200000	107.414667	113.933333	106.519333	106.610667	106.414000
0.166667	106.697333	112.012000	105.866000	105.946667	105.242000
0.142857	106.365333	110.778000	105.584667	105.649333	104.570000
0.125000	106.118000	109.906667	105.416667	105.446667	104.095333
0.111111	106.001333	109.259333	105.306667	105.308000	103.834000
0.100000	105.920000	108.807333	105.299333	105.268000	103.680667
0.083333	105.832000	108.164667	105.292667	105.248667	103.463333
0.062500	105.608000	107.450667	105.192000	105.131333	103.356667
0.050000	105.411333	107.067333	105.109333	105.025333	103.425333
0.040000	105.206000	106.694000	104.972667	104.888667	103.468667

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 2 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	121.810000	125.514000	116.841333	117.086667	117.250667
0.333333	127.654000	136.373333	126.040667	127.316667	127.011333
0.250000	121.390667	129.767333	120.242667	120.992000	119.942000
0.200000	118.096000	125.326000	117.174000	117.496667	116.056667
0.166667	116.218000	122.499333	115.482000	115.444667	113.780667
0.142857	115.019333	120.528000	114.390000	114.098667	112.268667
0.125000	114.158000	119.102000	113.610667	113.190667	111.249333
0.111111	113.507333	118.082667	113.044667	112.470667	110.468667
0.100000	113.076667	117.274000	112.662000	112.030000	109.981333
0.083333	112.367333	116.100000	112.026000	111.344667	109.224000
0.062500	111.539333	114.676000	111.272000	110.594667	108.398667
0.050000	110.949333	113.758000	110.710667	110.108667	108.026000
0.040000	110.371333	112.946000	110.250667	109.672667	107.745333

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 4 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	105.998000	107.215333	104.346000	104.336667	104.332667
0.333333	146.936667	155.300000	145.173333	147.193333	146.281333
0.250000	141.074667	150.233333	139.982667	141.797333	139.658000
0.200000	134.671333	143.184000	133.932667	135.117333	132.527333
0.166667	130.680667	138.298000	130.215333	130.884000	127.997333
0.142857	127.974000	135.020000	127.646000	128.072667	124.907333
0.125000	126.094667	132.579333	125.768000	125.982000	122.720000
0.111111	124.684667	130.682667	124.442667	124.514667	121.066000
0.100000	123.468000	129.038667	123.291333	123.267333	119.766667
0.083333	121.678000	126.738000	121.573333	121.472667	117.846667
0.062500	119.479333	123.839333	119.402000	119.185333	115.620667
0.050000	117.962000	121.992000	117.892667	117.702000	114.323333
0.040000	116.643333	120.273333	116.635333	116.458667	113.278667

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 8 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	124.522000	131.220000	122.604667	122.504000	125.038000
0.333333	113.000667	123.370000	112.030000	112.165333	113.645333
0.250000	108.649333	116.219333	108.130667	108.138667	108.826667
0.200000	106.827333	113.176000	106.486000	106.470667	106.691333
0.166667	105.833333	111.512000	105.600667	105.566667	105.550000
0.142857	105.094000	110.299333	104.926667	104.903333	104.688667
0.125000	104.636667	109.432000	104.472000	104.442000	104.122000
0.111111	104.314000	108.839333	104.175333	104.168000	103.766667
0.100000	104.092667	108.377333	103.983333	103.972667	103.506667
0.083333	103.730667	107.690000	103.678667	103.612667	103.075333
0.062500	103.350667	106.833333	103.294667	103.265333	102.669333
0.050000	103.122000	106.265333	103.077333	103.024000	102.459333
0.040000	102.967333	105.744667	102.914667	102.893333	102.363333

Gleichverteilte Pods bei 2 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	151.564000	154.653333	150.238667	150.849333	151.230667
0.333333	127.436000	135.416667	126.369333	127.342000	126.883333
0.250000	119.537333	126.306667	119.003333	119.396667	118.887333
0.200000	115.978000	122.030667	115.648667	115.748000	114.976000
0.166667	113.568667	119.155333	113.382667	113.327333	112.474000
0.142857	111.950667	117.168667	111.832667	111.695333	110.800000
0.125000	110.909333	115.823333	110.804667	110.608000	109.707333
0.111111	110.123333	114.793333	110.050000	109.812000	108.868667
0.100000	109.503333	113.982667	109.443333	109.234000	108.213333
0.083333	108.556000	112.716000	108.514000	108.256000	107.275333
0.062500	107.352667	111.095333	107.350667	107.092667	106.128000
0.050000	106.662667	110.090000	106.660667	106.390000	105.478000
0.040000	106.106000	109.212000	106.132667	105.865333	104.972667

Gleichverteilte Pods bei 4 Dimensionen

Weight	First-Fit	K8s-Least	K8s-Most	Perm-Pack	Vector-Dot
0.500000	186.082000	186.454667	185.803333	185.956667	185.937333
0.333333	146.708667	151.774000	145.984667	147.136667	146.233333
0.250000	134.394667	140.770000	134.245333	134.899333	134.139333
0.200000	128.632000	134.587333	128.519333	128.936667	127.766667
0.166667	124.294000	129.930667	124.230000	124.542000	123.209333
0.142857	121.684000	126.926667	121.653333	121.838000	120.460000
0.125000	119.676667	124.610000	119.636000	119.766667	118.322667
0.111111	118.064667	122.814000	118.093333	118.138000	116.698000
0.100000	116.923333	121.448667	116.938000	116.984000	115.502667
0.083333	115.042667	119.277333	115.040000	115.033333	113.558667
0.062500	112.768000	116.533333	112.756667	112.754000	111.284000
0.050000	111.330000	114.775333	111.337333	111.298667	109.895333
0.040000	110.140667	113.335333	110.154667	110.136667	108.818000

Gleichverteilte Pods bei 8 Dimensionen

Weight	First-Fit	KVD	KR	KM
0.500000	119.094000	116.410667	115.779333	115.655333
0.333333	111.967333	110.964000	110.585333	110.552000
0.250000	108.820667	107.680000	107.657333	107.788000
0.200000	107.379333	106.022000	106.290667	106.466667
0.166667	106.738667	105.165333	105.646000	105.857333
0.142857	106.378667	104.724667	105.285333	105.575333
0.125000	106.220000	104.419333	105.146000	105.477333
0.111111	106.000000	104.251333	105.015333	105.332000
0.100000	105.898667	104.100667	104.968667	105.283333
0.083333	105.793333	104.023333	104.944667	105.228000
0.062500	105.556000	104.042000	104.957333	105.191333
0.050000	105.427333	104.084667	104.966667	105.132000
0.040000	105.221333	104.078667	104.898000	104.990667

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 2 Dimensionen

Weight	First-Fit	KVD	KR	KM
0.500000	121.925333	117.454667	117.062667	117.103333
0.333333	127.646000	126.232667	125.943333	125.964000
0.250000	121.347333	119.700667	119.938667	120.175333
0.200000	118.090000	116.296667	116.918667	117.230667
0.166667	116.222000	114.331333	115.126000	115.505333
0.142857	115.043333	113.063333	114.041333	114.377333
0.125000	114.163333	112.216000	113.287333	113.638667
0.111111	113.559333	111.626667	112.760667	113.095333
0.100000	113.120667	111.224000	112.392667	112.680000
0.083333	112.422000	110.675333	111.793333	112.112667
0.062500	111.530667	109.973333	111.093333	111.264000
0.050000	110.930000	109.596667	110.629333	110.748667
0.040000	110.337333	109.274667	110.165333	110.238667

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 4 Dimensionen

Weight	First-Fit	KVD	KR	KM
0.500000	106.094000	104.498000	104.496000	104.510000
0.333333	147.127333	145.614000	145.340000	145.286667
0.250000	141.066000	139.471333	139.673333	139.885333
0.200000	134.692000	132.939333	133.681333	133.985333
0.166667	130.763333	128.894000	129.849333	130.206000
0.142857	128.007333	126.202667	127.316667	127.661333
0.125000	126.080667	124.360000	125.460667	125.804000
0.111111	124.574667	122.896667	124.114667	124.375333
0.100000	123.467333	121.851333	123.002667	123.237333
0.083333	121.709333	120.304000	121.497333	121.603333
0.062500	119.444000	118.314667	119.273333	119.364000
0.050000	117.950667	117.008667	117.823333	117.887333
0.040000	116.650667	115.937333	116.597333	116.646000

Pods bei gleichverteilter Aufteilung von 100 Knoten bei 8 Dimensionen

Selbständigkeitserklärung

Ich versichere, dass ich die Masterarbeit mit dem Titel „Analyse von Verfahren zur Optimierung des Container-Schedulings in Kubernetes“ selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Erik Schubert
Dresden, 31. August 2023