HAW HAMBURG

MASTERTHESIS
Christian Frank Bargmann

Entwicklung und Evaluation eines verteilten Autoscalers für den Einsatz in Hybrid-Cloud Umgebungen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Christian Frank Bargmann

Entwicklung und Evaluation eines verteilten Autoscalers für den Einsatz in Hybrid-Cloud Umgebungen

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

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachterin: Prof. Dr. Marina Tropmann-Frick

Eingereicht am: 16.06.2021

Christian Frank Bargmann

Thema der Arbeit

Entwicklung und Evaluation eines verteilten Autoscalers für den Einsatz in Hybrid-Cloud Umgebungen

Stichworte

Autoscaling, Elastizität, Cloud Computing, Container, Software-Architektur

Kurzzusammenfassung

Elastizität als wichtige Charakteristik des Cloud Computings, ist einer der wesentlichen Gründe weshalb zunehmend mehr Unternehmen ihre Anwendungen in die Cloud migrieren. Durch sie ist es möglich, von einer Cloud-Plattform bezogene Ressourcen anhand von vordefinierten Regeln dynamisch an den aktuellen Bedarf anzupassen. Somit können die mit Kunden vereinbarten Leistungen jederzeit verlässlich zugesichert werden. In dieser Arbeit wird eine verteilte Autoscaler-Architektur für den Einsatz in Hybrid-Clouds vorgestellt, die automatisiert Ressourcen auf Basis von Metriken als Container-as-a-Service bei einem Cloud-Provider provisionieren und horizontal skalieren kann. Die vorgestellte Architektur ist in der Lage, Anwendungen über mehrere Cloud-Plattformen hinweg zu skalieren und provisionierte Ressourcen von mehreren Providern parallel in die eigene lokale Infrastruktur einzubinden. Auf Grundlage der entworfenen Architektur wird ein Prototyp mit einem reaktiven Skalierungsalgorithmus, welcher den Lebenszyklus von Ressourcen berücksichtigt, implementiert und das Verhalten der Autoscaling-Architektur unter verschiedenen Lastszenarien evaluiert.

Christian Frank Bargmann

Title of Thesis

Development and implementation of a distributed autoscaling architecture for use in hybrid cloud environments

Keywords

Autoscaling, Elasticity, Cloud Computing, Container, Software Architecture

Abstract

The rapid elasticity characteristic of the cloud computing paradigm is an important reason why more and more companies are migrating their services to the cloud. It makes it possible to dynamically adjust resources provisioned at a cloud platform to current demand on the basis of predefined policies. In this way, SLAs contracted with customers can be reliably guaranteed at any time. This thesis presents a distributed autoscaling architecture for use in hybrid clouds that can automatically provision and horizontally scale resources based on metrics using container-as-a-service at a cloud provider. The architecture presented is capable of scaling applications across multiple cloud platforms and integrating provisioned resources from multiple providers into the local infrastructure stack in parallel. Based on the proposed architecture, a prototype that uses a reactive scaling algorithm which takes into account the lifecycle of resources is implemented and the behaviour of the autoscaler is evaluated under different workload scenarios.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

API Application Programming Interface.

CaaS Container as a Service.

CPU Central Processing Unit.

CRD Custom Resource Definition.

FFT Schnelle Fourier-Transformation.

GAE Google App Engine.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

MDP Markov-Entscheidungsprozesse.

NIST National Institute of Standards and Technology.

OS Operating System.

PaaS Platform as a Service.

PromQL Prometheus Query Language.

PTA Probalistische, zeitgesteuerte Automaten.

QoS Quality of Service.

REST Representational State Transfer.

RL Reinforcement Learning.

SaaS Software as a Service.

SLA Service Level Agreement.

SLAs Service Level Agreements.

SLO Service Level Objective.

SLOs Service Level Objectives.

SSE Server-Sent Events.

VM Virtuelle Maschine.

VMM Virtual-Machine-Monitor.

W3C World Wide Web Consortium.

YAML YAML Ain't Markup Language.

Listings

1 Einleitung

Die Charakteristik der schnellen Elastizität (Rapid Elasticity) des Cloud Computing Paradigmas ist ein wichtiger Grund, weshalb zunehmend mehr Unternehmen in die Cloud migrieren. Durch sie ist es möglich, von einer Cloud-Plattform bezogene Ressourcen anhand von vordefinierten Regeln dynamisch an den aktuellen Bedarf anzupassen. Somit können mit Kunden vereinbarte Leistungen jederzeit verlässlich erbracht werden.

Die Notwendigkeit, Ressourcen automatisch zu skalieren, hat zu einem breiten Angebot an Servicemodellen geführt, die über die Cloud-Plattformen unterschiedlicher Anbieter bezogen werden können (beispielsweise AWS EC2[1]). Die Auslagerung von Diensten zu einer Cloud-Plattform bietet viele Vorteile. So kann der Betrieb der ausgelagerten Dienste komplett an den Anbieter abgegeben werden, wobei der Nutzer einer Cloud-Plattform von der Expertise des Anbieters profitiert. Auch können Elastizitätsmechanismen der Cloud-Plattform genutzt werden, um dynamisch und bedarfsgerecht Ressourcen zu provisionieren und somit das Risiko einer Über- oder Unterprovisionierung zu minimieren.

Für viele Unternehmen ist es aus technischen, rechtlichen oder strategischen Gründen nicht möglich, ihre gesamte Infrastruktur in eine öffentliche Cloud zu verlagern. Deshalb ist eine Hybrid-Cloud als Bereitstellungsmodell eine interessante Alternative, beispielsweise wenn Lastspitzen temporär auf externe Cloud-Plattformen verlagert oder nur Teilanwendungen in die Kontrolle eines externen Cloud-Providers gegeben werden sollen. Trotz der wachsenden Anzahl an Cloud-Servicemodellen, ist die Umsetzung eines Hybrid-Cloud-Ansatzes für viele Unternehmen ein nicht zu unterschätzender Aufwand. Die Integration zwischen der eigenen Infrastruktur und den Schnittstellen externer Cloud-Provider sowie die Implementierung von plattformunabhängigen Skalierungsmethoden, stellen nach wie vor eine große Herausforderung dar.

[1] https://aws.amazon.com/de/ec2

1.1 Motivation

Die Menge an Ressourcen, die für eine Anwendung zu einem bestimmten Zeitpunkt benötigt werden, hängt von unterschiedlichen Faktoren ab. Einfluss haben zum Beispiel die Skalierungseigenschaft der Ressourcen, die angestrebten Quality of Service-Kriterien oder das Nutzungsprofil. Die Frage, wie viele Instanzen zu welcher Zeit zur Verfügung stehen müssen, ist daher nicht leicht zu beantworten.

Elastizitätsmechanismen müssen in einen Steuerungsprozess eingebettet werden und dieser sollte durch eine modulare und erweiterbare Software-Architektur abgebildet werden. Sowohl die Entwicklung von Elastizitätsmechanismen, als auch der Entwurf einer Architektur wird komplexer, wenn mehrere Cloud-Provider für die Skalierung von Ressourcen angebunden werden.

Oft ist Autoscaling eine Teilfunktionalität einer umfassenderen Plattform für die Skalierung und Verwaltung von Rechenressourcen. Diese Plattform kann entweder durch einen Cloud-Provider bereitgestellt oder selber in Form von Cloud-Infrastruktur betrieben werden. Die Teilfunktionalität „Autoscaling“ ist deshalb in der Regel stark an die Schnittstellen der jeweiligen Plattform gekoppelt. Um eine Unabhängigkeit von einzelnen Providern zu gewährleisten, ist es notwendig, Autoscaling als eigene Funktionalität anzubieten. Hierfür gibt es wenige Lösungen, insbesondere als Open-Source-Software.

Für den zuverlässigen Betrieb von Software auf einer Cloud-Plattform wird ein dediziertes Team mit Expertenwissen benötigt. Durch eine einfach zu betreibende Autoscaling-Lösung, die sich in bestehende interne Infrastrukturen integrieren und lokale Ressourcen bei Lastspitzen zu externen Cloud-Plattformen auslagern kann, wird Skalierung auch für Unternehmen zugänglich gemacht, die keine Cloud-Infrastruktur betreiben können oder wollen.

1.2 Zielsetzung

In dieser Arbeit wird eine verteilte Autoscaler-Architektur für den Einsatz in Hybrid-Clouds vorgestellt, die automatisiert Ressourcen auf Basis von Metriken als Container-as-a-Service (CaaS) bei Cloud-Providern provisionieren und horizontal skalieren kann. Die vorgestellte Architektur ist in der Lage, Anwendungen über mehrere Cloud-Plattformen

hinweg zu skalieren und provisionierte Ressourcen von mehreren Providern parallel in die eigene lokale Infrastruktur einzubinden.

Auf Grundlage der entworfenen Architektur wird ein Prototyp mit einem reaktiven Skalierungsalgorithmus, der den Lebenszyklus von Ressourcen berücksichtigt, implementiert und das Verhalten der Autoscaling-Architektur unter verschiedenen Lastszenarien wird evaluiert.

1.3 Abgrenzung

Für die Implementierung eines Autoscalers wird sowohl eine geeignete Software-Architektur mit Schnittstellen und Komponenten als auch ein Mechanismus zur Herstellung von Elastizität für eine zu skalierende Ressource benötigt. Im Fokus dieser Arbeit steht die technische Entwicklung der Software-Architektur für den Autoscaler zum Einsatz in Hybrid-Clouds. Es wird ein reaktiver Skalierungsalgorithmus vorgestellt, den die prototypische Implementierung verwendet.

In der aktuellen Forschung werden in einigen Arbeiten auch Elastizitätsmechnismen miteinander verglichen (siehe Kapitel 3). Ein solcher Vergleich ist nicht Bestandteil dieser Arbeit.

1.4 Aufbau der Arbeit

Diese Arbeit besteht aus sechs aufeinander aufbauenden Kapiteln. Nach dieser Einleitung (Kapitel 1) werden zunächst in Kapitel 2 die Grundlagen zu Cloud Computing und der Erzeugung von Elastizität in Softwaresystemen vorgestellt.

In Kapitel 3, der Analyse, werden die Problemstellung und die Anforderungen an die zu entwickelnde Autoscaler-Architektur formuliert. Hierzu werden Arbeiten vorgestellt, die sich mit der automatischen Skalierung von Ressourcen zu Cloud-Providern auseinandersetzen. Auf Basis dieser Arbeiten werden Eigenschaften eines Autoscalers für den Einsatz in Hybrid-Cloud Umgebungen herausgearbeitet und Anforderungen formuliert.

In Kapitel 4, der Umsetzung, wird die auf Grundlage der zuvor formulierten Anforderungen entwickelte Autoscaler-Architektur vorgestellt. Hierzu wird das Datenmodell sowie die Software-Architektur mit Komponenten und Schnittstellen beschrieben. Weiterhin

wird auf den Skalierungsprozess, den reaktiven Skalierungsalgorithmus und auf Details der Implementierung eingegangen. Abschließend wird die Umsetzung mit den Anforderungen abgeglichen.

In Kapitel 5 wird die Implementierung anhand einer zu skalierenden Beispielanwendung evaluiert. Hierzu wurde eine Beispielanwendung entwickelt, deren Service mithilfe des entwickelten Autoscalers zu mehreren Cloud-Providern skaliert wird. Dabei wird die Beispielanwendung verschiedenen Lastszenarien ausgesetzt und die Ergebnisse werden vorgestellt und diskutiert.

Abschließend werden in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

2 Grundlagen

Um die Funktionsweise eines Autoscalers in Hybrid-Cloud Umgebungen zu verstehen, werden in diesem Abschnitt zunächst die Grundlagen vorgestellt. In Abschnitt 2.1 wird eine allgemeine Definition von Cloud Computing gegeben. Aufbauend darauf wird auf gängige Bereitstellungs- und Servicemodelle des Cloud Computings eingegangen. In Abschnitt 2.2 wird die automatische Skalierung von Rechenressourcen beschrieben. Hierzu werden die Begriffe Elastizität und Skalierbarkeit voneinander abgegrenzt und Kategorisierungsmöglichkeiten aufgezeigt.

2.1 Cloud Computing

Cloud Computing ist ein Servicemodell, das es erlaubt, Rechenressourcen in Form eines Dienstes zu beziehen [3]. Benutzer einer Cloud-Plattform handeln mit den Cloud-Providern Bedingungen für den Betrieb der Dienste aus, die in Form von Service Level Objectives (SLOs) festgehalten werden. Die Cloud-Provider hingegen sind wiederum für die Bereitstellung und Verwaltung von Cloud-Diensten für Konsumenten verantwortlich. Um diese Aufgaben zu erfüllen, erwerben und betreiben Cloud-Provider Recheninfrastrukturen und organisieren den Netzzugang für die Benutzer der Plattform.

Der Begriff Cloud Computing bezieht sich sowohl auf Anwendungen, die als Dienste über das Internet bezogen werden, als auch auf Hardware und Systemsoftware in Rechenzentren, welche diese als Dienste bereitstellen. Um dieses Servicemodell anzubieten, bündelt der Cloud-Provider physikalische Hardware zu einem Rechenpool zusammen. Durch Virtualisierungstechnologien werden die Ressourcen dem Benutzer der Cloud zugänglich gemacht [22]. Durch diesen Rechenpool kann der Kunde dynamisch auf seinen individuellen Rechenbedarf reagieren und jederzeit flexibel seine Ressourcen dem Bedarf entsprechend anpassen [28]. Gleichzeitig entfällt die Notwendigkeit für den Betrieb eigener Hardware für Kunden, da Rechenressourcen auch vollständig beim Cloud-Provider betrieben werden können.

Das National Institute of Standards and Technology (NIST) beschreibt neben vier Bereitstellungsmodellen und drei Servicemodellen, welche in den Abschnitten 2.1.1 und 2.1.2 vorgestellt werden, insgesamt fünf wesentliche Charakteristiken, die eine Cloud erfüllen muss [22]:

On-demand self-service ermöglicht es Benutzern einer Cloud-Plattform, Ressourcen wie z.B. Speicherplatz oder Rechenzeit vollautomatisch zu beziehen, ohne dass hierfür ein menschliches Eingreifen notwendig ist. Auch lassen sich Ressourcen über die Cloud-Plattform, angepasst an den tatsächlich benötigten Bedarf, über einen bestimmten Zeitraum mieten.

Resource pooling als Eigenschaft betrachtet eine Cloud-Plattform als Bündelung von physikalischen Rechenressourcen, die unter den Benutzern der Plattform aufgeteilt werden (Mandantenfähigkeit). Die physikalischen Ressourcen werden virtuell unterteilt und dynamisch den Benutzern der Cloud-Plattform, abhängig von deren individuellen Bedarf, zugewiesen oder abgezogen. Der tatsächliche physikalische Standort von Ressourcen wird vom Benutzer abstrahiert. Viele Cloud-Provider bieten Benutzern jedoch die Möglichkeit auf höheren Abstraktionsebenen den Standort der virtuellen Ressourcen in Form von Ländern, Regionen oder Rechenzentrumsstandorten zu konfigurieren.

Broad network access ist die Eigenschaft einer Cloud-Plattform, heterogenen Client-Systemen durch standardisierte Mechanismen über ein Netzwerk den Zugriff auf Rechenressourcen zu gewähren. Die Auswahl dieser standardisierten Mechanismen ist abhängig vom gewählten Cloud-Provider.

Rapid elasticity erlaubt es Benutzern einer Cloud-Plattform, Rechenressourcen zu jedem beliebigen Zeitpunkt zu provisionieren oder freizugeben. Dem Benutzer gegenüber äußert sich diese Eigenschaft in Form eines scheinbar unbegrenzten Pools an Rechenressourcen, aus dem er sich nach Belieben bedienen kann.

Measured service setzt voraus, dass die Cloud-Plattform seinen Nutzern einen Monitoringdienst zur Verfügung stellt. Dieser Monitoringdienst macht die Nutzung der Cloud-Plattform transparent, sowohl für den Cloud-Provider als auch für ihre Benutzer. Durch die geschaffene Transparenz ist es der Cloud-Plattform möglich, abhängig von den überwachten Messgrößen (z.B. Auslastung von CPUs, Speicherbedarf, Anzahl der Benutzer) die Bereitstellung von Ressourcen zu optimieren.

2.1.1 Bereitstellungsmodelle

Insgesamt werden vier verschiedene Bereitstellungsmodelle, die ein Cloud-Provider anbieten kann, in dem NIST Standard genannt. Diese Bereitstellungsmodelle werden unterschieden nach Art der Benutzer, die Zugriff auf die Cloud-Plattform haben, und nach Art der Organisation, welche die Cloud-Plattform für die Benutzer bereitstellt [22].

Private Clouds sind Cloud-Plattformen, deren Infrastruktur exklusiv von einer einzigen Organisation betrieben wird und auf die nur Mitglieder dieser Organisation Zugriff haben. Die Infrastruktur der Plattform wird hierbei von der anbietenden Organisation selbst oder einer Drittpartei bereitgestellt. Die Bereitstellung erfolgt in einem eigenen Rechenzentrum unter der Kontrolle der anbietenden Organisation oder außerhalb in gemieteten Standorten.

Public Clouds sind Cloud-Plattformen, deren Infrastruktur der Öffentlichkeit zugänglich ist. Public Clouds können von jeder Art von Organisation angeboten werden. Oftmals wird die Infrastruktur an verschiedenen Standorten unter Kontrolle der Organisation bereitgestellt und betrieben.

Community Clouds werden ausschließlich für Organisationen bereitgestellt, die ein gemeinsames Anliegen oder Ziel verfolgen. Die Infrastruktur der Plattform wird von einer Untergruppe der Mitglieder der Community oder einer Drittpartei bereitgestellt und betrieben. Die Infrastruktur kann sich hierbei entweder innerhalb oder auch außerhalb der Standorte einer der beteiligten Organisationen befinden.

Hybrid Clouds sind Cloud-Plattformen, die mindestens zwei der oben genannten Bereitstellungsmodelle miteinander kombinieren. Jede Plattformen stellt jedoch eine selbständig funktionierende Einheit dar. Durch den Einsatz von Technologien, mit denen die Plattformen integriert werden, lassen sich Anwendungen und Daten zwischen den Plattformen portieren und austauschen. Durch die Kombination der Cloud-Plattformen lässt sich der Pool an verfügbaren Rechenressourcen erweitern. Auch kann im Falle von Ressourcenknappheit oder schwankenden Nutzungsverläufen eine Lastverteilung zwischen den verbundenen Plattformen stattfinden.

2.1.2 Servicemodelle

Im NIST Standard werden weiterhin vier verschiedene Servicemodelle unterschieden, welche im Folgenden erläutert werden [22].

Software as a Service (SaaS) bietet eine lauffähige Softwareanwendung als Dienst an, der auf der Infrastruktur einer Cloud-Plattform betrieben und über diese dem Benutzer zugänglich gemacht wird. Der Benutzer hat keinerlei Möglichkeit, die Software-Konfiguration zu beeinflussen und hat keine Kontrolle über die darunterliegende Infrastrukturschicht. Je nach Softwareanwendung kann der Benutzer wenige, benutzerspezifische Einstellungen vornehmen. Bekannte Beispiele für SaaS sind Google's Gmail[1] oder Microsoft's Office 365[2].

Platform as a Service (PaaS) bietet Benutzern einer Cloud-Plattform die Möglichkeit, eine eigene Softwareanwendung auf der Cloud-Infrastruktur bereitzustellen. Die Softwareanwendung muss allerdings das von der Cloud-Plattform bereitgestellte Tooling unterstützen. Der Benutzer muss sich bei PaaS selbst um die Konfiguration der Softwareanwendung und deren Laufzeitumgebung kümmern. Wie bei SaaS hat der Benutzer jedoch keinerlei Kontrolle über die zugrundeliegende Infrastruktur, auf der die Anwendung betrieben wird. Beispiele für moderne PaaS sind Heroku[3] für das Hosting von Webanwendungen oder Google App Engine (GAE)[4].

Infrastructure as a Service (IaaS) ermöglicht es Benutzern, komplette Infrastrukturkomponenten zu beziehen und zu verwalten. Der Benutzer einer Cloud-Plattform kann bei diesem Servicemodell Ressourcen wie zum Beispiel CPUs, Speicherplatz und Netzwerkregeln selbstständig konfigurieren. Auf den Infrastrukturkomponenten kann beliebige Software ausgeführt und das Operating System (OS) vom Benutzer festgelegt werden. Die provisionierten Infrastrukturkomponenten sind in der Regel virtualisiert. Auf die darunterliegende Infrastruktur, die für den Betrieb und die Aufteilung der Hardware auf die Benutzer der Cloud-Plattform notwendig ist und seitens des Cloud-Providers betrieben wird, hat der Benutzer auch keinen Zugriff.

Zusätzlich zu den vier beschriebenen Servicemodellen haben sich in den letzten Jahren viele weitere ***as a Service*** Angebote entwickelt. Im Rahmen dieser Arbeit ist ***Container as a Service (CaaS)*** als Servicemodell relevant, welches sich IaaS zuordnen lässt. Bei

[1] https://mail.google.com/
[2] https://www.office.com/
[3] https://www.heroku.com/
[4] https://cloud.google.com/appengine/

CaaS kann der Benutzer einer Cloud-Plattform Anwendungen in Form von Software-Containern auf der Infrastruktur des Cloud-Providers betreiben. Die Container, inklusive Softwareanwendung und Laufzeitumgebung, werden hierbei vom Benutzer erstellt. Der Cloud-Provider bietet dem Benutzer eine Schnittstelle an, über die ein Container auf der Cloud-Infrastruktur ausgeführt werden kann. Der Benutzer selbst hat keine Kontrolle über die zugrundeliegende Infrastruktur, auf welcher der Software-Container beim Cloud-Provider betrieben wird.

Zusammenfassend ist die Auswahl eines geeigneten Servicemodelles eine Abwägung zwischen der Verantwortung für den Betrieb und der Abgabe von Kontrolle über die bezogenen Ressourcen von der Cloud-Plattform. IaaS bietet im direkten Vergleich mit SaaS die größte Kontrolle über die provisionierten Ressourcen, wobei die Verantwortung für den Betrieb der Ressourcen beim Benutzer der Cloud-Plattform liegt. Andererseits ist SaaS eine einfache Möglichkeit, Ressourcen einer Cloud zu nutzen, ohne sich um darunterliegende Infrastruktur und die Wartung kümmern zu müssen. Die Verantwortung für den Betrieb der Software liegt hier beim Cloud-Provider auf Kosten der Kontrolle über die Softwareanwendung selbst.

2.2 Skalierung von Softwaresystemen

Die Elastizität von Rechenressourcen in einer Cloud ermöglicht eine dynamische Anpassung an den tatsächlich benötigten Rechenbedarf. Bei einer statischen Provisionierung von Rechenressourcen kann es zu einer Über- oder Unterprovisionierung (siehe Abbildung 2.1) von Rechenressourcen kommen. Beides wirkt sich negativ auf den Betrieb aus.

Bei einer ***Überprovisionierung*** ist die Kapazitätsgrenze an die erwarteten Lastspitzen angepasst. Bei geringerer Nachfrage werden Rechenressourcen bezahlt und vorgehalten, welche nicht ausgelastet werden. Anders formuliert übersteigt das Angebot A die Nachfrage N, d.h. $A > N$. Wird die Kapazitätsgrenze für Rechenressourcen niedriger gesetzt als die tatsächliche Nachfrage bei Lastspitzen, tritt der Fall einer ***Unterprovisionierung*** ein, d.h. $A < N$. Diese Unterprovisionierung hat zu Folge, dass bei Lastspitzen nicht genügend Rechenressourcen zur Verfügung stehen, um Anfragen ordnungsgemäß zu bearbeiten. Eine Unterprovisionierung kann sich beispielsweise durch längere Anfragezeiten oder durch Verbindungsabbrüche äußern, was zu einer Unzufriedenheit der Nutzer und somit langfristig zu einer sinkenden Gesamtnachfrage führt.

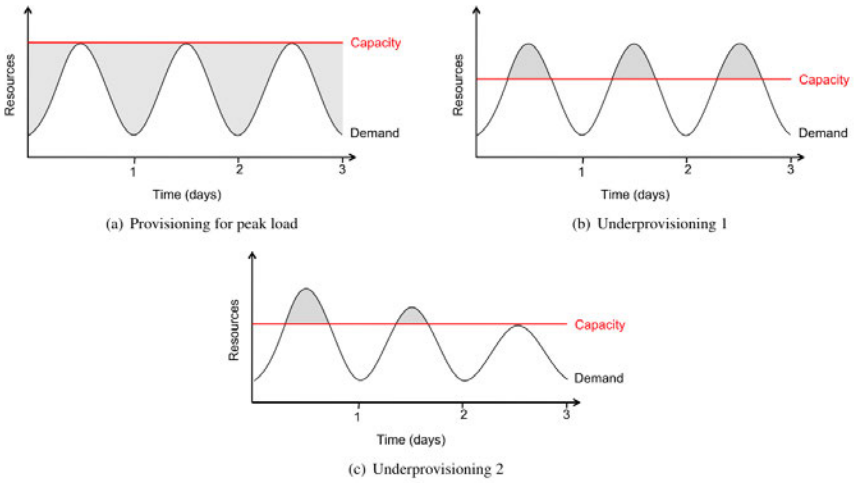


Abbildung 2.1: Über- und Unterprovisionierung bei statischer Ressourcenprovisionierung, entnommen aus [3]

Durch Cloud Computing lassen sich Kapazitätsgrenzen dynamisch an die Nachfrage anpassen. Für Unternehmen ist diese Eigenschaft wirtschaftlich interessant, da lediglich Ressourcen, die tatsächlich benötigt werden, auch bezahlt werden müssen. Hierdurch lassen sich Betriebskosten für Infrastruktur senken [30].

2.2.1 Elastizität und Skalierbarkeit

Die Begriffe Elastizität und Skalierbarkeit werden oft miteinander substituiert, haben jedoch eine unterschiedliche Bedeutung.

Elastizität beschreibt die Fähigkeit eines Softwaresystems, Ressourcen (wie CPUs, Speicherplatz, VM- und Container-Instanzen) dynamisch hinzuzufügen und zu entfernen, um sich in Echtzeit an Schwankungen einer gegebenen Lastkurve anzupassen [1]. Diese Anpassung geschieht in der Regel automatisch und setzt implizit das Vorhandensein eines Prozesses voraus, welcher diese Steuerung übernimmt. Um einen optimalen Elastizitätsgrad zu erreichen, muss dieser Prozess die Steuerung so vornehmen, dass zu jedem Zeitpunkt die verfügbaren Ressourcen dem aktuellen Bedarf so nahe wie möglich kommen. Anders formuliert muss im Idealfall zu jedem Zeitpunkt das Angebot exakt der Nachfrage entsprechen, d.h. $A = N$. Der Prozess, der diese Steuerung vollautomatisiert vornimmt, wird in dieser Arbeit als ***Autoscaler*** bezeichnet. Unterschieden wird zwischen horizontaler und vertikaler Elastizität. Horizontale Elastizität bezieht sich auf das Hinzufügen

und Entfernen von Instanzen einer Softwareanwendung. Vertikale Elastizität bezieht sich hingegen auf das Hinzufügen und Entfernen von Rechenressourcen einer Softwareanwendung, wie zum Beispiel CPU-Kerne, Speicherkapazitäten oder Bandbreite.

Skalierbarkeit beschreibt die Fähigkeit eines Softwaresystems, inklusive sämtlicher Hardware, Virtualisierungs- und Abstraktionsebenen, steigende Arbeitslasten durch die Nutzung zusätzlicher Rechenressourcen aufrechtzuerhalten [10]. Im Gegensatz zur Elastizität, ignoriert die Skalierbarkeit eines Softwaresystems zeitliche Aspekte, zum Beispiel wie oft oder wie feingranular Skalierungsoperationen vorgenommen werden. Skalierbarkeit ist somit eine statische Eigenschaft eines Softwaresystems und im Gegensatz zur Elastizität zeitlich unabhängig [19]. Vielmehr ist Skalierbarkeit eine Eigenschaft eines Softwaresystems, die notwendig aber nicht hinreichend für dessen Elastizität ist.

Ein weiterer Begriff, der oft im Kontext von Elastizität verwendet wird, ist die ***Effizienz***. Sie beschreibt die Menge an Ressourcen, die zur Bearbeitung einer bestimmten Arbeitslast notwendig ist [10]. In der Regel impliziert ein höherer Elastizitätsgrad eine bessere Effizienz (Implikation). Eine bessere Effizienz ist aber auch durch andere Maßnahmen beispielsweise durch performantere Implementierungen zu erreichen.

2.2.2 Skalierungsarten

Anhand der Art und Weise wie Skalierungsoperationen vorgenommen werden, können zwei verschiedene Modi unterschieden werden. Bei der ***manuellen Skalierung*** ist der Benutzer selbst für die Überwachung der Anwendung und deren Laufzeitumgebung sowie für die Durchführung aller Skalierungsmaßnahmen verantwortlich.

Bei der ***automatischen Skalierung*** hingegen übernimmt ein vollautomatisierter Prozess die Durchführung aller Skalierungsmaßnahmen. Damit kann die Elastizität eines Softwaresystems hergestellt werden. Die automatische Skalierung kann weiterhin unterteilt werden in ***reaktive*** und ***proaktive*** Skalierung.

Bei der ***reaktiven Skalierung*** werden bei Überschreitung von zuvor definierten Schwellenwerten oder Regeln Skalierungsoperationen durchgeführt. Grundlage für diese Skalierungsregeln sind Anforderungen aus Service Level Agreements (SLAs) oder Schwellenwerte, deren Bedingungen sich aus einer oder einer Menge an messbaren Metriken zusammensetzt (z.B. Auslastung von CPU, Speicher, Netzwerk, Antwortzeiten, Anfrageraten). Diese Schwellenwerte können entweder statisch oder dynamisch festgelegt werden.

Die Verwendung von Skalierungsregeln mit statischen Thresholds findet in der Praxis vergleichsweise oft Verwendung [1], so z.B. bei großen Cloud-Plattformen wie AWS EC2[5], VMware Tanzu[6] oder Kubernetes[7].

Bei der ***proaktiven Skalierung*** werden Techniken eingesetzt, um Vorhersagen über Veränderungen der Lastkurve zu treffen. Anschließend werden auf Basis dieser Vorhersage Skalierungsoperationen durchgeführt. Hierfür werden mathematische Ansätze verwendet, welche verschiedene Verhaltensweisen eines Systems untersuchen, um seine zukünftigen Zustände vorauszusagen (z.B. Markow-Entscheidungsprozesse (MDPs) oder probabilistische, zeitgesteuerte Automaten (PTAs)).

Einige Techniken wie Reinforcement Learning (RL), Steuerungstechnik oder Warteschlangentheorie finden sich sowohl in reaktiven als auch proaktiven Skalierungsansätzen wieder. In mehreren Arbeiten werden auch Kombinationen aus reaktiven und proaktiven Skalierungsansätzen vorgestellt, die unter dem Begriff der ***hybriden Skalierung*** zusammengefasst werden [29] [4].

[5]https://aws.amazon.com/de/ec2/
[6]https://tanzu.vmware.com/tanzu
[7]https://kubernetes.io/

3 Analyse

In diesem Kapitel werden die Herausforderungen beschrieben, die bei der Entwicklung eines Autoscalers für Hybrid-Cloud Umgebung zu beachten sind. Zunächst wird eine konkrete Problemstellung formuliert und Teilaspekte davon werden ausführlicher behandelt.

Aufbauend auf der Problemstellung (siehe 3.1) werden Arbeiten und Entwicklungsprojekte vorgestellt, die sich mit der automatischen Skalierung von Rechenressourcen durch die Verwendung von verschiedenen Servicemodellen von Cloud-Providern beschäftigen (siehe 3.2). Der Fokus dieser Arbeiten liegt hierbei nicht ausschließlich auf dem Einsatz in Hybrid-Clouds, da auch Skalierungsansätze vorgestellt werden, die sich für unterschiedliche Bereitstellungsmodelle einsetzen lassen.

Anschließend werden aus diesen Arbeiten Eigenschaften herausgearbeitet, die für einen Autoscaler mit dem Zweck der Skalierung von Ressourcen speziell in Hybrid-Cloud Umgebungen von Bedeutung sind (siehe 3.2.3).

Zuletzt werden in diesem Kapitel Anforderungen an den zu entwickelnden Autoscaler formuliert (siehe 3.3).

3.1 Problemstellung

Im Fokus dieser Arbeit steht die Entwicklung eines Autoscalers für den Einsatz in Hybrid-Cloud Umgebungen. Wie in Abschnitt 2.2.1 beschrieben, ist der Begriff *Autoscaler* die Bezeichnung für einen Steuerungsprozess, welcher vollautomatisiert die Skalierung von Ressourcen übernimmt. Dieser Prozess kann zentral oder verteilt ausgeführt werden. Ein Autoscaler hat die Aufgabe, für eine Ressource (z.B. eine Softwareanwendung oder eine VM), die auf einer Cloud-Plattform betrieben wird, Elastizität herzustellen [10]. Hierzu werden in zeitlichen Intervallen Rechenkapazitäten hinzugefügt oder entfernt. Die

verfügbaren Ressourcen sollen der aktuellen Nachfrage bestmöglich entsprechen, um eine in Abschnitt 2.2 angesprochene Über- oder Unterprovisionierung zu vermeiden [5]. Diese Tätigkeit wird im Rahmen dieser Arbeit als *Autoscaling* bezeichnet. Autoscaling muss nach [2] die folgenden Aufgaben umsetzen:

- *Scale Out*: Automatisches Hinzufügen zusätzlicher Ressourcen bei steigender Nachfrage.
- *Scale In*: Automatisches Entfernen ungenutzter Ressourcen bei sinkender Nachfrage.
- Konfiguration von Scale In/Out durch Regeln, entweder statisch festgelegt oder konfigurierbar.
- Erkennen und Ersetzen von fehlerhaften oder nicht erreichbaren Ressourcen.

Diese Aufgaben werden in eine kontinuierlich ausgeführte Prozessschleife eingebettet, die in Abbildung 3.1 dargestellt ist.

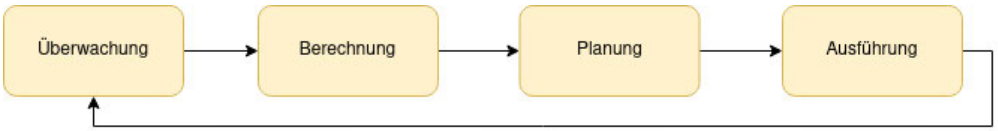


Abbildung 3.1: Prozessschleife für einen Autoscaling-Prozess

- *Überwachung*: Ein Autoscaler benötigt ein Monitoring-System, das Messgrößen über den Zustand der zu skalierenden Ressourcen und zur Einhaltung eines Quality of Service (QoS) liefert. In der Überwachungsphase werden Messgrößen gesammelt und aggregiert. Sie dienen als Grundlage für Skalierungsentscheidungen.
- *Berechnung*: In der Berechnungsphase werden die vom Monitoring-System gelieferten Daten verarbeitet und ermittelt, welche Skalierungsoperationen vorgenommen werden müssen.
- *Planung*: In der Planungsphase wird entschieden, wie die berechneten Skalierungsoperationen ausgeführt werden. Es wird festgelegt, welche konkreten Ressourcen hinzugefügt oder entfernt werden, gemeinsam mit einem Zeitpunkt, wann die Skalierungsoperation umgesetzt wird.

- *Ausführung*: In der Ausführungsphase werden die Skalierungsoperationen umgesetzt. Ressourcen werden gemäß des Skalierungsplans über Schnittstellen von Cloud-Infrastruktur hinzugefügt oder entfernt.

Die Implementierung von Autoscaling mit einer konkreten Autoscaler-Architektur ist Gegenstand aktueller Forschung - insbesondere, wenn Ressourcen über mehrere Cloud-Plattformen hinweg skaliert werden sollen. Auf diese Herausforderungen wird in den folgenden Abschnitten im Detail eingegangen.

3.1.1 Entwicklung von geeigneten Skalierungsalgorithmen

Skalierungsalgorithmen sollen sich möglichst breit unter Beachtung von unterschiedlichen Skalierungszielen einsetzen lassen. Diese Skalierungsziele ergeben sich meist, wie in Abschnitt 2.2 dargestellt, aus den Service Level Agreements (SLAs) zwischen Benutzern und Providern einer Cloud-Plattform. Ein SLA besteht aus mehreren Service Level Objectives (SLOs). Ein SLO kann zum Beispiel sein: „Eine Instanz muss weniger als 500 Anfragen pro Sekunde erhalten“. Anhand dieses Beispiels lassen sich drei Bestandteile einer SLO ableiten. Ein QoS Kriterium, in diesem Fall die Anzahl an Anfragen, eine Begrenzung, hier 500 Anfragen pro Sekunde, sowie ein Operator, „weniger als“. Weitere SLOs können technische Aspekte wie beispielsweise die Speicherauslastung oder Fehlerraten sowie wirtschaftliche oder strategische Aspekte beinhalten.

Skalierungsalgorithmen werden komplexer, wenn Ressourcen plattformübergreifend skaliert werden. Jede Cloud-Plattform stellt eine in sich geschlossene, unabhängige Einheit dar. Sie besitzt einen eigenen, gebündelten Pool an Rechenressourcen mit individuellen Eigenschaften. Plattformübergreifende Skalierungsalgorithmen sollten diese individuellen Eigenschaften, wie beispielsweise zeitliche Aspekte bei Scale In/Out, berücksichtigen. Auch gibt es mögliche strategische oder wirtschaftliche Gründe für eine Quotierung von Rechenressourcen. So kann es sinnvoll sein, bei der Skalierung von Rechenressourcen eine Cloud-Plattform zu priorisieren oder eine Gewichtung der Nutzung der Plattformen vorzunehmen.

Ob plattformspezifisch oder plattformübergreifend, in jedem Fall muss ein Skalierungsalgorithmus sicherstellen, dass die verfügbaren Ressourcen für eine Softwareanwendung bestmöglich der aktuellen Nachfrage entsprechen (vgl. 2.2.1). Eine Skalierung, sodass zu jedem Zeitpunkt gilt $A = N$, ist bei nichtdeterministischer Nachfrage schwer sicherzustellen. In den Abschnitten 3.2.1 und 3.2.2 werden Arbeiten vorgestellt, die durch die

Implementierung von Skalierungsalgorithmen eine Annäherung an den Idealzustand erzielen.

3.1.2 Technische Herausforderungen bei der Implementierung

Die Interoperabilität von unterschiedlichen Cloud-Infrastrukturen ist eine große Herausforderung. Oft ist Autoscaling eine Teilfunktionalität einer bestehenden Infrastrukturlösung und somit stark gekoppelt an die Schnittstellen und Konfiguration der jeweiligen Cloud-Plattform. Dadurch ist Autoscaling als Funktionalität einer spezifischen Cloud-Plattform grundsätzlich nicht für die Skalierung über mehrere Plattformen hinweg konzipiert. Für eine plattformübergreifende Steuerung von Ressourcen ist es deshalb notwendig, dass ein Autoscaler als eigenständige fachliche Komponente, unabhängig von den zu steuernden Cloud-Infrastrukturen, bereitgestellt und ausgeführt werden kann.

Aus der Forderung nach dem Entkoppeln von Autoscaler und Cloud-Plattformen resultieren technische Herausforderungen bei der Implementierung. Cloud-Plattformen besitzen individuelle Schnittstellen, über die sich Ressourcen allokieren und freigeben lassen. Unabhängig von der Art der zu skalierenden Ressource (ob auf Ebene einer virtuellen Maschine, eines Containers oder einer Anwendung) muss zur Entkopplung eine plattformunabhängige Abstraktion geschaffen werden. Auch für die Cloud-Plattformen selbst muss eine Abstraktion erfolgen, um eine plattformübergeifende Skalierung zu implementieren.

Eine weitere Herausforderung ist die Granularität von Cloud-Provider Angeboten, insbesondere mit Blick auf IaaS. Viele Public-Cloud-Provider bieten in ihrem Servicemodell VMs mit vordefinierten Ressourcen (z.B. CPU-Kerne, Arbeitsspeicher, Festplattenspeicher) an. Für Autoscaler mit einer horizontalen Skalierung sind diese vorgefertigten Ressourcen nicht vorteilhaft, da in der Regel die Leistung der Ressource durch die zu skalierende Softwareanwendung nicht vollständig ausgenutzt wird. Vertikale Skalierung wird oft nicht unterstützt (siehe AWS EC2[1]). Diese Limitation hat Auswirkungen auf die Effektivität des Skalierungsprozesses und auf die Elastizität der zu skalierenden Ressource. Diese Herausforderung lässt sich nicht durch Entwurfsentscheidungen innerhalb der Autoscaler-Architektur lösen, sondern resultiert vielmehr in einer Forderung an Cloud-Provider, die Granularität ihrer Servicemodelle zu verbessern.

[1] https://aws.amazon.com/de/ec2/

Die Überwachung von Messgrößen ist ein wichtiger Bestandteil des Skalierungsprozesses und Voraussetzung für die Funktionsweise eines Autoscalers. Weil die Anzahl an überwachten Messgrößen Auswirkungen auf den Skalierungsprozess hat, müssen die für die zu skalierende Ressource wichtigen Messgrößen identifiziert, klassifiziert und priorisiert werden. Auch die Intervalle, in denen Messgrößen kontrolliert werden, müssen je nach individueller Implementierung eines Autoscalers angepasst werden.

3.2 Verwandte Arbeiten

In diesem Abschnitt werden Arbeiten vorgestellt, welche die automatische Skalierung von Ressourcen auf Cloud-Plattformen zum Thema haben. Hierbei wurden Arbeiten ausgewählt, die neben Ansätzen zu Skalierungsalgorithmen auch einen konkreten Architekturentwurf für die Implementierung eines Autoscalers präsentieren.

Abschnitt 3.2.1 erläutert Arbeiten, die sich mit der Skalierung von Ressourcen auf einer Cloud-Plattform, d.h. mit einem Pool an Rechenressourcen, beschäftigen. Abschnitt 3.2.2 stellt Arbeiten vor, welche zur Skalierung von Ressourcen mindestens eine weitere Cloud-Plattform anbinden, beispielsweise durch Integration von externer Infrastruktur in die eigene Cloud-Plattform.

Die zusammenfassenden Bewertung (vgl. Abschnitt 3.2.3) vergleicht die vorgestellten Arbeiten anhand unterschiedlicher Kriterien und versucht, Eigenschaften zu identifizieren, die für die Entwicklung eines Autoscalers in Hybrid-Cloud Umgebungen relevant sind.

3.2.1 Autoscaling in Cloud Umgebungen

Shen et al. stellen in ihrer Arbeit [25] ein Autoscaling-System mit Namen CloudScale für mandantenfähige Cloud-Infrastrukturen vor. Das Autoscaling-System verwendet VMs als zu skalierende Ressource. Prozesse innerhalb der VMs können sowohl horizontal als auch vertikal skaliert werden. Das Autoscaling-System wird auf einer physikalischen Maschine installiert und ist in der Lage, über den Xen-Hypervisor virtuelle Ressourcen dynamisch zu verwalten. Um Elastizität für einen innerhalb der VMs ausgeführten Prozess herzustellen, wird eine proaktive Skalierungstechnik verwendet. Diese Skalierungstechnik basiert auf Schneller Fourier-Transformation (FFT), um Muster in einer Lastkurve zu identifizieren. Je nach Erfolg der Identifizierung eines Musters wird eine Vorhersage

des Ressourcenbedarfs durch ein Nachfragemodell [25] oder durch ein auf zeitdiskreten Markowketten basierendem Vorhersagemodell getroffen. Der Xen-Hypervisor selbst dient hier als Monitoring-System und liefert die zur Skalierung benötigten Daten zum Lastverhalten. CloudScale kann auf unterschiedliche Art und Weisen mit Skalierungskonflikten umgehen. Skalierungskonflikte treten auf, wenn mehrere Anwendungen gleichzeitig skaliert werden und nicht genügend Ressourcen für die Skalierung der Anwendungen zur Verfügung stehen. Das Autoscaling-System löst Skalierungskonflikte, indem es den Konflikt entweder lokal auflöst (innerhalb einer VM) oder Prozesse auf andere virtuelle Maschinen migriert. Migrationen werden auf Basis der Ressourcennutzung proaktiv durchgeführt, um einem Konflikt zuvorzukommen. Auch hier werden Zeitreihendaten des Xen-Hypervisors verwendet, um die virtuellen Maschinen zu konfigurieren.

Chieu et al. [7] präsentieren eine verteilte Autoscaler-Architektur, welche dynamisch Ressourcen in Form von virtuellen Maschinen innerhalb einer Cloud-Plattform skalieren kann. Das Besondere ist dabei, dass die Entscheidung über die Skalierung von Ressourcen in dieser Architektur verteilt vorgenommen wird. Hierzu wird ein sogenannter Capacity and Utility Agent (CUA) auf jedem physischen Host in einer dedizierten virtuellen Maschine ausgeführt. Dieser Agent versucht kontinuierlich die virtualisierten Ressourcen seines Hosts zu optimieren. Grundlage für die Skalierungsentscheidungen sind Zeitreihendaten zu aktuellem System- und Lastverhalten, welche über die API des auf dem physikalischen Host eingesetzten Hypervisors bezogen werden. Die Skalierung wird hierbei reaktiv durchgeführt. Die agentenbasierte Architektur wird durch einen sogenannten Distributed Capacity Agent Manager (DCAMgr) koordiniert und verwaltet. Diese zentrale Managerkomponente steuert die Kommunikation teilnehmender Agents und ist für die Ausführung der horizontalen Skalierungsoperationen über die Hypervisor API auf den einzelnen physikalischen Hosts zuständig.

Maximilien et al. [21] weisen in ihrer Arbeit auf die Bedeutung von plattformunabhängiger Middleware hin und präsentieren eine High-Level-Architektur für die Skalierung von Ressourcen auf einer Cloud-Plattform. Es wird eine Abstraktionsschicht für die Skalierung von virtuellen Maschinen vorgestellt. In ihrer Arbeit wird kein konkretes Konzept für die Verknüpfung mehrer Plattformen erläutert.

Weiterhin stellen Kukade et al. [18] einen Entwurf für eine monolithische Autoscaler-Architektur vor, die für die automatische Skalierung von containerisierten Microservices konzipiert wurde. Die Architektur besteht aus einer zentralen Masterkomponente, welche sowohl die Überwachung als auch die Skalierung von Containern auf Hosts übernimmt.

Die Autoscaler-Architektur kann auf physikalischen, auf virtualisierten oder einer Kombination aus physikalischen und virtualisierten Hosts betrieben werden. Die Masterkomponente besteht aus vier Modulen. Drei dieser Module sammeln über einen Polling-Ansatz Daten über die Anzahl von aktiven Container-Instanzen, die Speicherauslastung und die Anzahl an Zugriffen auf die jeweilige Instanz. Ein reaktives Skalierungsmodul trifft Skalierungsentscheidung anhand von zuvor definierten Thresholds und führt horizontale Skalierungsoperationen bei Verletzung der Skalierungsregeln durch.

Kubernetes ist ein Open-Source Cluster-Manager für das automatische Bereitstellen und Skalieren von containerisierten Softwareanwendungen [17]. Kubernetes wurde initial von Google entwickelt und ist seit seiner Veröffentlichung am 21. Juli 2015 [11] zum *de facto* Industriestandard für die Skalierung von containerbasierten Workloads geworden. Kubernetes ist in der Lage, nicht nur einzelne Container sondern ganze Containergruppen in Form von sogenannten *Pods* zu skalieren. Ein Replication-Controller sorgt dafür, dass eine bestimmte Anzahl von Replikationen eines Pods konstant im Einsatz ist. Für die Erreichung dieses Zustandes werden gelöschte oder beendete Pods neu gestartet, sodass der Ist-Zustand dem Soll-Zustand als Replikationen entspricht. Autoscaling in Kubernetes ist stark an Replication-Controller gekoppelt. Der Autoscaler teilt dem Replication-Controller mit, wie viele Instanzen eines Pods vorhanden sein sollen. Der Replication-Controller wiederum stellt sicher, dass diese Anzahl Instanzen aktiv ist. Der Autoscaler von Kubernetes arbeitet in einer Steuerungsschleife, welche in einem festgelegten Intervall Metriken zu überwachten Pods abfragt, um ihren aktuellen Ressourcenbedarf zu ermitteln und reaktiv auf Basis von Thresholds Skalierungsentscheidungen zu treffen. Mittlerweile unterstützt Kubernetes auch die Verwendung von benutzerspezifischen Metriken als Grundlage für die Skalierung von Ressourcen. Derzeit implementiert Kubernetes eine horizontale Pod-Autoskalierung. Da Kubernetes Ressourcen auf Basis von Containern skaliert, ist deshalb in Zukunft eine vertikale Autoskalierung denkbar, da es eine Vielzahl an Methoden gibt, um Ressourcen eines Containers zu erhöhen, ohne dabei seine Ausführung zu unterbrechen [12] [16]. Auch gibt es Ansätze, eine proaktive Skalierung in Kubernetes zu ermöglichen [27].

Ye et al. [29] stellen eine monolithische Autoscaler-Architektur für Container vor, die einen hybriden Skalierungsansatz verwendet, um Verletzungen von SLAs durch schwankende Lastverläufe entgegenzuwirken. Die Autoscaler-Architektur gliedert sich dabei in vier Komponenten auf. Eine Monitoring-Komponente sammelt Metriken innerhalb eines vordefinierten Zeitfensters, welche für die Skalierung von Container-Ressourcen benötigt werden. Die Abfrage wird intervallbasiert über bereitgestellte Schnittstellen einer Cloud-

Plattform durchgeführt. Diese Zeitreihen werden vom Autoscaler lokal gespeichert für die Vorhersage von Entwicklungsverläufen der Lastkurve. Eine Predictor-Komponente nimmt die Zeitreihendaten entgegen und erstellt ein Modell mithilfe von Zeitreihenanalyse-Techniken, um zukünftige Trends zu bestimmen. In jedem Skalierungsintervall des Autoscalers wird dieses Modell aktualisiert und eine Vorhersage für das nächste Zeitintervall durchgeführt. Eine Decider-Komponente nimmt den aktuellen und vorausgesagten Ressourcenbedarf als Eingangswerte entgegen und entscheidet auf dieser Grundlage, welche Strategie für die Skalierung verwendet werden soll. Anhand der Strategie werden Ressourcen hinzugefügt oder entfernt. Abschließend führt eine Executor-Komponente die Skalierungsentscheidungen über die Schnittstellen der Cloud-Plattform aus. Für die Evaluierung ihrer Implementierung nutzen Ye et al. Kubernetes als Cloud-Infrastruktur, um die Anzahl an Replikas für Pods horizontal zu skalieren.

3.2.2 Autoscaling in Multi-Cloud Umgebungen

Elastizität wird in Multi-Cloud Umgebungen hergestellt, indem die Kapazitäten von Rechenzentren und Cloudumgebungen mithilfe der Allokierung von Ressourcen aus externen Cloud-Plattformen erweitert werden.

Grid Computing ist ursprünglich motiviert durch die Bündelung von Rechenressourcen unter der Kontrolle von individuellen, oft wissenschaftlichen Institutionen [9]. Die Unterscheidung zwischen Grid- und Cloud-Computing ist oft unpräzise und hängt zumeist vom Verwendungszweck des Rechenpools ab. Die meisten Grid Computing Implementierungen werden für wissenschaftliche Berechnungen verwendet, während beim Cloud Computing die Bereitstellung von universell einsetzbarer (d.h. nicht zweckgebundener) Rechenleistung im Vordergrund steht. Ähnlich wie beim Cloud Computing mit dem Einsatz von Multi-Cloud Umgebungenm basiert das Grid Computing auf Middleware, die sowohl Identitätsmanagement- als auch Datenmanagement-Funktionen bereitstellt [26]. Bei Multi-Cloud-Umgebungen im Cloud Computing werden ähnliche Middleware und Abstraktionen benötigt, um unterschiedliche Cloud-Infrastrukturen miteinander zu verbinden. Im Unterschied zum Grid Computing hat der Provider einer Cloud-Plattform beim Cloud Computing die volle Kontrolle über die Skalierung seiner Ressourcen über mehrere Plattformen hinweg, wo hingegen beim beim Grid Computing die Kontrolle über die Ressourcen bei den jeweiligen Plattformen selbst liegt [13].

Nimbus[2] war eine der ersten Open-Source Implementierungen einer Cloud-Plattform mit einer cloudübergreifenden Ressourcenskalierung. Nimbus ist in der Lage, Ressourcen einer externen Cloudplattform in den eigenen Rechenpool zu integrieren, wenn lokale Ressourcen nicht ausreichend sind und so eine horizontale Skalierung des eigenen Rechenpools durchzuführen. Nimbus hat seine Ursprünge im Grid Computing und wurde daher für die Ausführung von verteilten Berechnungen entwickelt. Es ist jedoch nicht entwickelt worden, um unabhängige Cloud-Plattformen miteinander zu verknüpfen.

Keahey et al. [13] stellen in ihrer Arbeit Voraussetzungen für die Verbindung von Cloud-Plattformen vor, wie zum Beispiel die Forderung nach virtuellen Netzwerken, um Rechenressourcen in Form von virtuellen Maschinen, zugehörig zu einem Benutzer und verteilt auf unterschiedlichen Cloud-Plattformen, miteinander zu verbinden. Allerdings werden die Skalierungsansätze nicht näher beleuchtet.

Marshall et al. [20] stellen Elastic Site, eine Plattform zur Erweiterung der Rechenkapazität von einer internen Cloud-Infrastruktur mithilfe von externen Ressourcen, vor. Die Plattform wurde entwickelt, um die Rechenleistung einer auf Nimbus basierenden Private Cloud mit Amazon EC2-Instanzen[3] horizontal zu erweitern. Wenn zusätzliche Ressourcen benötigt werden, ist die Plattform in der Lage, entweder lokal oder extern virtuelle Maschinen zu provisionieren. Die Skalierungsentscheidungen werden reaktiv anhand von Messgrößen aus einer Job Queue im lokalen Cluster getroffen.

Biswas et al. [4] stellen in ihrer Arbeit eine monolithische Autoscaler-Architektur für einen Vermittler (Broker) zwischen Public- und Private-Cloud Umgebungen vor. Dieser Vermittler provisioniert bei Bedarf Rechenressourcen aus einer Public-Cloud Umgebung, um auf Lastschwankungen der Private-Cloud zu reagieren. In der Arbeit wird ein hybrider Skalierungsansatz für eine horizontale Ressourcenskalierung in Form von VMs erläutert, der jedoch nicht weiter formalisiert wird.

Keahey et al. [14] präsentieren in einer weiteren Arbeit eine Autoscaler-Architektur für Infrastruktur-Outsourcing in Multi-Cloud Umgebungen. Die Autoscaler-Architektur nutzt virtuelle Maschinen als zu skalierende Ressourcen. Weiterhin werden Domänen als eine Gruppe an homogenen VMs definiert, die gemeinsamen Regeln und Beschränkungen unterliegen. Eine Domäne kann sich über mehrere Cloud-Plattformen erstrecken. Ein sogenannter Domain Manager reguliert die Größe einer Domäne. Er ergänzt oder

[2] http://www.nimbusproject.org/about/
[3] https://aws.amazon.com/de/ec2/

entfernt Instanzen aus einer Domäne auf Basis von Informationen aus einer Sensor-Komponente und unter der Berücksichtigung von benutzerspezifischen Regeln (Policies). Eine Provisioner-Komponente ist für die horizontale Skalierung von Ressourcen über die Schnittstellen der Cloud-Plattformen verantwortlich. Zeitgleich liefert die Provisioner-Komponente Daten über den Zustand im Lebenszyklus einer provisionierten Ressource. Diese Zustandsinformationen werden gemeinsam mit sogenannten Heartbeats, Signale die Auskunft über die Gesundheit einer provisionierten virtuellen Maschine auf einer Cloud-Plattform geben, vom Domain-Manager verwendet um reaktive Skalierungsentscheidungen zu treffen.

Durch seine Flexibilität und Erweiterbarkeit lässt sich der Cluster-Manager Kubernetes auch in Multi-Cloud Szenarien einsetzen. Mit Kubernetes lassen sich auch Ressourcen in Form von virtuellen oder physischen Maschinen aus unterschiedlichen Cloud-Plattformen zu einem logischen Cluster bündeln. Kubernetes kann hierdurch containerisierte Anwendungen auf einzelne Knoten im Cluster schedulen. Durch den Cluster-Autoscaler kann die Größe des Kubernetes-Clusters automatisch angepasst werden, wenn es beispielsweise Pods gibt, die aufgrund unzureichenden Rechenressourcen nicht im Cluster ausgeführt werden konnten oder es Knoten im Cluster gibt, die über einen längeren Zeitraum nicht ausgelastet sind und deren Pods auf anderen vorhandenen Knoten platziert werden können. Der Cluster-Autoscaler besitzt Implementierungen, um für die Skalierung mit gängigen Cloud-Plattformen wie Microsoft Azure[4], AWS[5] oder Google Cloud Platform[6] zu kommunizieren und Knoten dynamisch zu allokieren.

Auch gibt es Projekte, die Kubernetes-Cluster um weitere Skalierungsmöglichkeiten ergänzen. Die Cluster-API[7] ist ein Projekt, dass es ermöglicht, Kubernetes um Abstraktionen für die Erstellung von Clustern zu erweitern. Dadurch können Benutzer neue Ressourcen wie ganze Kubernetes-Cluster oder Maschinen als Ressource in einem Management-Cluster verwalten. Eine Controller-Logik übernimmt anschließend die Provisionierung dieser Ressourcen bei einem externen Cloud-Provider. Hierfür existieren ebenfalls zahlreiche providerspezifische Implementierungen [15]. Durch ein Management Cluster können somit einzelne Workload-Cluster verwaltet werden, die wiederum einen eigenen Ressourcenpool skalieren und steuern können.

[4] https://azure.microsoft.com/
[5] https://aws.amazon.com/de/
[6] https://cloud.google.com/
[7] https://github.com/kubernetes-sigs/cluster-api

Chandra [6] stellt eine Möglichkeit für Cloud-Bursting mithilfe der Projekte Virtual Kubelet[8] und KIP[9] vor. Die vorgestellte Methode setzt den Betrieb eines eigenen Clusters in der privaten Cloud-Umgebung voraus und nutzt die in Abschnitt 3.2.1 beschriebenen Autoscaling-Funktionalitäten von Kubernetes für die horizontale Skalierung von Container-Instanzen. In diesem Ansatz wird Container-as-a-Service als Servicemodell verwendet.

Weiterhin beschreibt Mennig [23] einen Multicluster-Ansatz mit dem Einsatz von Service Meshes für die Skalierung von Rechenressourcen. Hier wird Cluster-as-a-Service als Servicemodell eines Public-Cloud-Providers genutzt, um Compute Cluster zu provisionieren und diese in die lokale Cluster-Infrastruktur einzubinden. Ein Service-Mesh wird für die Cross-Cluster-Kommunikation von Ressourcen verwendet, gemeinsam mit einem Multi-Cloud-Scheduler, um Workloads bei Lastspitzen über mehrere Cluster zu verteilen.

3.2.3 Zusammenfassende Bewertung

Arbeit	Ressource	Methode (H/V)	Modus	Plattform	Architektur
Biswas et al.	VMs	Horizontal	Hybrid	Mehrere	Monolithisch
Chieu et al.	VMs	Horizontal	Reaktiv	Eine Mehrere	Verteilt
Keahey et al.	VMs	Horizontal	Reaktiv	Mehrere	Verteilt
Kubernetes	Container VMs	Horizontal Vertikal	Reaktiv	Eine Mehrere	Verteilt
Kukade et al.	Container	Horizontal	Reaktiv	Eine	Monolithisch
Marshall et al.	VMs	Horizontal	Reaktiv	Mehrere	Monolithisch
Maximilien et al.	VMs	Horizontal Vertikal	Proaktiv Reaktiv	Mehrere	Monolithisch
Shen et al.	VMs	Horizontal Vertikal	Proaktiv	Eine	Monolithisch
Ye et al.	Container	Horizontal	Hybrid	Eine	Monolithisch

Tabelle 3.1: Übersicht und Vergleich von unterschiedlichen Autoscaler-Architekturen

[8] https://github.com/virtual-kubelet/virtual-kubelet
[9] https://github.com/elotl/kip

Es existieren viele Lösungen für die Implementierung von Autoscaler-Architekturen. Die Tabelle 3.1 fasst die vorgestellten Arbeiten zusammen. Jede der Arbeiten ermöglicht eine Elastizitätssteuerung auf Infrastrukturebene durch Container oder virtuelle Maschinen. Sowohl Container als auch virtuelle Maschinen eignen sich für die Skalierung von Client-Server Anwendungen. Es gibt weitere Autoscaler-Architekturen, die im Rahmen dieser Arbeit nicht vorgestellt werden, für andere Architekturstile, um Elastizität beispielsweise auf Anwendungs- oder Plattformebene zu implementieren.

Die Virtualisierung durch Hypervisor (VMMs) ist die am weitesten verbreitete Virtualisierungstechnik im Cloud Computing [24]. Durch ihre besonderen Eigenschaften hinsichtlich Flexibilität, Skalierbarkeit und Ressourceneffizienz werden Container jedoch kontinuierlich beliebter und werden mehr und mehr von Cloud-Providern als weitere Virtualisierungstechnik adaptiert. Viele Cloud-Provider wie Google oder AWS bieten ein eigenes Servicemodell für die Ausführung von Containern ohne Management von darunterliegender virtualisierter Cloud-Infrastruktur an (z.B. AWS Fargate[10] oder Google Cloud Run[11]).

Wie in Abschnitt 3.2.1 beschrieben, stellen Kukade et al. [18] einen Entwurf für eine monolithische Autoscaler-Architektur vor, die für die automatische Skalierung von containerisierten Microservices konzipiert wurde. Die Container-Ressourcen werden auf den Worker-Nodes über die Schnittstellen der Container-Runtime skaliert. Dadurch ist die Architektur stark gekoppelt an diese Schnittstellen, was die Erweiterbarkeit einschränkt. Gleichzeitig dient die Container Runtime als Monitoring-System um Daten für die Skalierung bereitzustellen, wodurch die Kopplung zwischen der Container-Runtime und der Autoscaler-Architektur weiter erhöht wird.

Biswas et al. [4] erläutern die Implementierung eines Brokers zwischen einer lokalen Cloud-Plattform und einer externen Cloud-Plattform. Ein Klient muss sich nicht um den Betrieb und die Wartung des Autoscalers kümmern, sondern lediglich Skalierungsregeln festlegen und Messgrößen für Skalierungsentscheidungen bereitstellen. Dies kann sowohl Vor- als auch Nachteil sein, da die Broker-Infrastruktur außerhalb der Kontrolle des Klienten liegt.

Die von Chieu et al. [7] vorgestellte Autoscaler-Architektur nutzt einen agentenbasierten Ansatz, um Skalierungsentscheidungen verteilt zu treffen. Der Ansatz ist interessant,

[10] https://aws.amazon.com/de/fargate
[11] https://cloud.google.com/run

da Skalierungsentscheidungen somit feingranularer getroffen werden und sich ressourcenspezifische Parameter unkompliziert in die Entscheidungslogik eines Agenten aufnehmen lassen. Diese feine Granularität birgt allerdings auch Risiken, da Skalierungsentscheidungen auf globaler Ebene schwerer nachvollziehbar werden. Auch hier sind die Agents stark an die Schnittstellen der zu skalierenden Ressource, in diesem Fall an die des Hypervisors, gekoppelt.

Die Multi-Cloud Architektur von Keahey et al. ist vielversprechend. Die Einführung von logischen Domänen, die sich über mehrere Cloud-Plattformen hinweg erstrecken können, ist ein wichtiger Abstraktionsschritt. Auch die fachliche Aufteilung der Komponenten gemäß des Skalierungsprozesses (vgl. Abb. 3.1) erscheint sinnvoll. Obwohl die Komponenten der Autoscaler-Architektur verteilt ausgeführt werden können, nutzt die Provisioner-Komponente konkrete Implementierungen für unterschiedliche, providerspezifische Schnittstellen. Hier kann es sinnvoll sein, die Logik für die Provisionierung von Ressourcen auf den externen Cloud-Plattformen wiederum in selbstständige Komponenten zu unterteilen, die wiederum von der Provisioner-Komponente gesteuert werden.

Kubernetes als Cluster-Manager ist ein mächtiges Werkzeug für die Skalierung von Containern, sowohl in Single- als auch mit entsprechenden Erweiterungen in Multi-Cloud Umgebungen. Mit seiner controllerbasierten Architektur und durch die Möglichkeit der Erweiterung um benutzerspezifische Abstraktionen (CRDs) ist Kubernetes in der Lage, eine Abstraktionsschicht für die Skalierung von Ressourcen beliebiger Art zu schaffen. Allerdings ist der Betrieb eines Kubernetes-Clustern ein nicht zu unterschätzender Aufwand, der oft ein dediziertes Team mit Expertenwissen voraussetzt. Die automatische Skalierung von Ressourcen ist eine Teilfunktionalität, die Kubernetes als Plattform für containerisierte Anwendungen bereitstellt. Die Skalierungfähigkeit als solche kann nicht dediziert bereitgestellt werden. Cluster-Betreiber müssen die gesamte Plattform mit all ihren Funktionalitäten betreiben. Die Auswahl von individuellen Features ist nicht möglich. Da viele Multi-Cloud Ansätze [6] [23] an Kubernetes-spezifische Schnittstellen gekoppelt sind und Kubernetes als Abstraktionsschicht nutzen, ist hier eine plattformunabhängige Skalierung von Ressourcen ohne den Einsatz von Kubernetes nicht möglich. Zusätzlich erhöht die Nutzung von Kubernetes-Erweiterungen für Multi-Cloud die ohnehin schon hohe Komplexität des Clusterbetriebs.

Grundlage für Skalierungsentscheidungen, egal ob in Single oder Multi-Cloud Umgebungen, sind Messgrößen über die zu skalierende Ressource. Die Arbeiten von [25] [7] nutzen Messgrößen über virtualisierte Ressourcen in Form von virtuellen Maschinen, die über

die API des Hypervisors bezogen werden. Kukade et al. [18] nutzt die API der Container-Runtime auf Knoten, um Messgrößen für die Skalierung zu beziehen. In beiden Fällen sind die eingesetzte Virtualisierungstechnik und die Monitoring-Komponente der jeweiligen Autoscaler-Architektur stark gekoppelt. Um weitere Metriken zu implementieren, muss bei [18] ein vollständiges neues Modul implementiert werden, nur um Messgrößen zu aggregieren und für die Autoscaler-Architektur verarbeitbar zu machen. Hier wird der Bedarf nach standardisierten Metrikformaten und darauf aufbauenden Aggregationmöglichkeiten sichtbar. Allgemein ist die Überwachung von Messgrößen für die Skalierung von Ressourcen ist ein wichtiger Bestandteil des Skalierungsprozesses. Sie ist ein Teil jeder Autoscaler-Architektur und besitzt eine eigene Komplexität.

3.3 Anforderungen

Aus den vorgestellten Herausforderungen (vgl. 3.1.1 und 3.1.2) und der zusammenfassenden Bewertung in Abschnitt 3.2.3 ergeben sich die Anforderungen an die zu entwickelnde Autoscaler-Architektur im Rahmen dieser Arbeit.

A1) Der Autoscaler soll die in 3.1 genannten Aufgaben und Prozessschritte umsetzen.

Damit die Autoscaler-Architektur Elastizität für Softwareanwendungen herstellen kann, implementiert die Architektur die in 3.1 beschriebenen Funktionalitäten. Die Architektur kann die Prozessschritte Überwachung, Berechnung, Planung und Ausführung durchführen.

A2) Der Autoscaler soll die Skalierung zu mehreren Cloud-Anbietern ermöglichen.

Die Autoscaler-Architektur kann auf Basis einer Ressourcenabstraktion automatisch eine Skalierung zu mehreren Cloud-Anbietern durchführen. Der Autoscaler kann containerisierte Anwendungen skalieren. Containerisierte Anwendungen werden plattformunabhängig beschrieben und lassen sich über individuelle Schnittstellen von Cloud-Plattformen allokieren.

A3) Der Autoscaler soll plattformunabhängig bereitstellbar und ausführbar sein.

Die Autoscaler-Architektur ist an keine plattformspezifischen Schnittstellen gebunden. Die Autoscaler-Architektur kann somit auf physischer oder virtualisierter Infrastruktur ausgeführt werden.

A4) Der zu entwickelnde Autoscaler soll externe Monitoring-Systeme unterstützen.

Die Autoscaler-Architektur kann standardisierte Metrikformate für Skalierungsentscheidungen nutzen und ist mit externen Monitoring-Systemen kompatibel.

A5) Der zu entwickelnde Autoscaler soll erweiterbar sein für die Anbindung weiterer Cloud-Plattformen.

Die Autoscaler-Architektur unterstützt die Anbindung von externen Cloud-Plattformen. Cloud-Plattformen können durch die Implementierung einer Interface-Spezifikation angebunden werden. Die Autoscaler-Architektur ist entkoppelt von providerspezifischen Schnittstellen.

A6) Der zu entwickelnde Autoscaler soll die Implementierung weiterer Skalierungsalgorithmen unterstützen.

Die Autoscaler-Architektur erlaubt die Implementierung individuell angepasster Skalierungsalgorithmen. Über eine Interface-Spezifikation können reaktive, proaktive oder hybride Skalierungsalgorithmen implementiert werden, ohne dass ein umfangreiches Refactoring von anderen Komponenten innerhalb der Architektur notwendig ist.

4 Umsetzung

In diesem Kapitel wird die Umsetzung des im Rahmen dieser Arbeit implementierten Autoscalers beschrieben. Aus den formulierten Anforderungen wurde zunächst ein Konzept für eine Autoscaler-Architektur entwickelt, welche in den folgenden Teilabschnitten ausführlich erläutert wird. Hierzu wird auf die Teilkomponenten der Architektur eingegangen und eine reaktive Skalierungstechnik vorgestellt. Anschließend werden technische Details aus der konkreten Implementierung erläutert und die Autoscaler-Architektur mit den zuvor definierten Anforderungen abgeglichen.

4.1 Datenmodell

In der Autoscaler-Architektur sind zwei Datentypen bedeutend, ***Services*** und ***Instanzen***. Eine ***Anwendung*** wird betrachtet als eine Komposition aus mehreren Services, wobei jeder Service einen eigenständigen Teil der Geschäftslogik der Gesamtanwendung kapselt. Einem Service zugeordnet sind eine Menge an Instanzen. Auf die Modellierung von Services und Instanzen wird in den folgenden Abschnitten eingegangen.

4.1.1 Modellierung von Services

Während eine Anwendung eine logische Gruppierung von mehreren Services darstellt, lässt sich ein Service als eine logische Gruppierung von mehreren Instanzen definieren. Ein Service definiert gemeinsame Regeln, die für eine Gruppe an Instanzen gelten. Die Datenstruktur eines Services ist in Abbildung 4.1 visualisiert.

Ein Service wird anhand eines eindeutigen Namens (*name*) identifiziert. Über einen Pfad (*path*) wird definiert, unter welche Route die Funktionalitäten des Services bereitgestellt werden. Innerhalb einer Anwendung dürfen keine unterschiedlichen Services mit

einem identischen Pfad existieren. Rechenoperationen für Anfragen über den Pfad werden von laufenden Instanzen aus der Menge an Instanzen eines Services durchgeführt. Jeder Service besitzt individuelle Skalierungsregeln, die durch den Benutzer in Form von Thresholds festgelegt werden können (*description*, *query*).

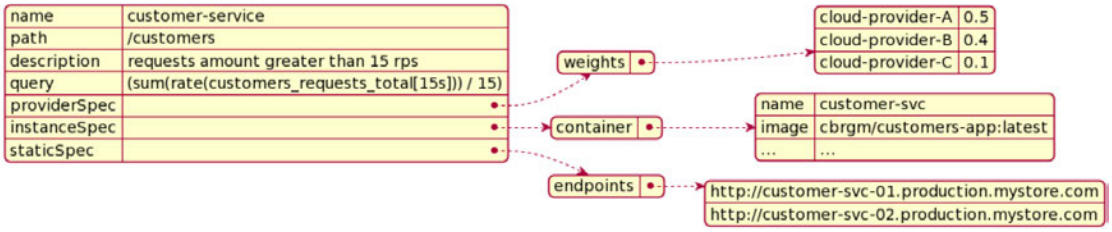


Abbildung 4.1: Datenstruktur eines Services als logische Abstraktion einer Menge an Instanzen

Über eine Provider-Spezifikation (*providerSpec*) lassen sich externe Cloud-Plattformen definieren, bei denen im Falle einer Skalierungsoperation Instanzen provisioniert werden. Jede Cloud-Plattform kann unterschiedlich gewichtet werden (weights). Hierdurch ist nicht nur eine automatische Skalierung von Instanzen zu mehreren Cloud-Plattformen möglich, sondern auch eine benutzerspezifische Priorisierung der Plattformen, die während des Skalierungsprozesses berücksichtigt wird.

Die Instanzen-Spezifikation (*instanceSpec*) ermöglicht die Parametrisierung von Containern, in denen die zu skalierende Anwendung, abstrahiert durch eine Instanz, ausgeführt wird. Abschließend können statische Endpunkte (*staticSpec*) von Instanzen festgelegt werden, die auf der lokalen Infrastruktur ausgeführt werden.

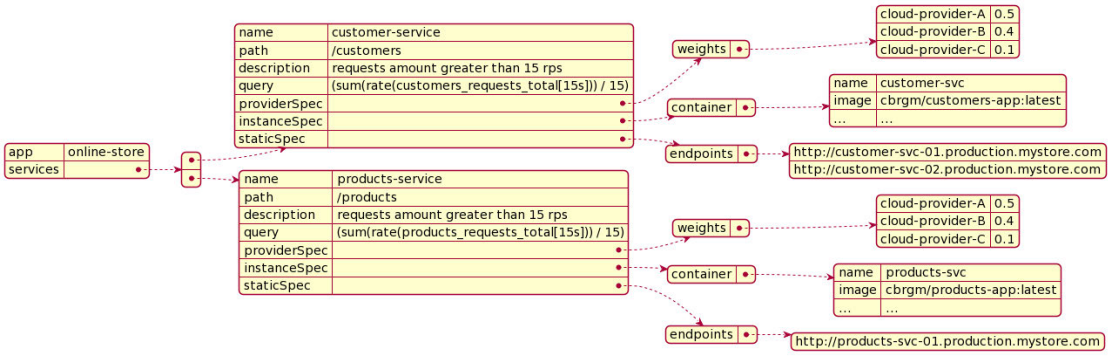


Abbildung 4.2: Datenstruktur einer Anwendung bestehend aus zwei Services

In Abbildung 4.2 ist eine Anwendung, bestehend aus zwei Services dargestellt. Je nach Anwendung kann die Anzahl an Services variieren. Für die Erzeugung von Elastizität muss eine Anwendung aus mindestens einem Service bestehen.

4.1.2 Modellierung von Instanzen

Als Instanz wird eine flüchtige und ersetzbare Rechenressource in Form einer containerisierten Anwendung bezeichnet. Eine Instanz kann *on-demand* provisioniert werden und kann eine beliebige zweckgebundene Rechenoperation ausführen. Die Flüchtigkeit einer Instanz ist eine wichtige Voraussetzung für die Skalierung mit der entwickelten Autoscaler-Architektur. Flüchtig bedeutet in diesem Kontext, dass eine Instanz jederzeit unerreichbar werden kann, beispielsweise durch Anwendungsfehler, einen Netzwerkausfall oder durch Speicherprobleme. Eine Instanz muss demnach jederzeit ersetzbar sein. Eine Instanz, die eine andere Instanz ersetzt, kann sich automatisch in die Ausführung der zweckgebundenen Rechenoperation eingliedern und einen Teil der Rechenlast übernehmen, ohne dass eine manuelle Konfiguration notwendig ist. Jede Instanz ist unabhängig von anderen Instanzen in der einem Service zugeordneten Gesamtmenge.

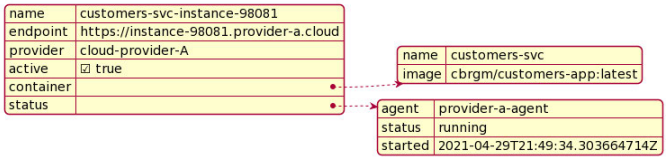


Abbildung 4.3: Datenstruktur einer Instanz

In Abbildung 4.3 ist die Datenstruktur einer Instanz dargestellt. Eine Instanz wird anhand eines Namens (*name*) identifiziert. Der Endpunkt gibt eine URL an, über die sich die Webschnittstelle der Instanz bedienen lässt. Eine Instanz wird auf einer Cloud-Plattform ausgeführt. Die Cloud-Plattform wird in Form einer Providerbezeichnung (*provider*) hinterlegt. Über einen booleschen Wert (*active*) wird definiert, ob die Instanz zur De-Provisionierung freigegeben wird.

Jede Instanz besitzt eine Container-Spezifikation (*container*). In dieser Spezifikation werden Eigenschaften für die Ausführung der containerisierten Anwendung, welche durch eine Instanz repräsentiert wird, angegeben. Diese Eigenschaften umfassen beispielsweise einen Container-Namen, ein Image, Umgebungsvariablen oder Portfreigaben.

Weiterhin besitzt jede Instanz einen Status (*status*). Der Status hält Eigenschaften die angeben, wie und wann die Instanz provisioniert wurde. Zusätzlich wird der Zustand einer Instanz vermerkt, beispielsweise ob sie derzeit gestartet oder gestoppt wird.

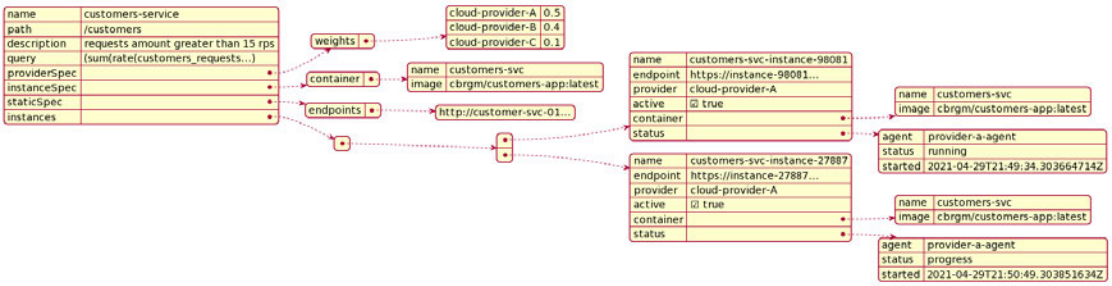


Abbildung 4.4: Datenstruktur eines Services mit zwei zugeordneten Instanzen

In Abbildung 4.4 ist ein Service mit zwei zugeordneten Instanzen dargestellt. Einem Service sind beliebig viele, homogene Instanzen zugeordnet. Die Container-Spezifikation (*instanceSpec*) aus einem Service dient als „Blaupause“ für die Ausführung einer Instanz in Form eines Containers auf einer externen Cloud-Plattform.

4.1.3 Lebenszyklus einer Instanz

Eine Instanz als flüchtige und ersetzbare Rechenressource besitzt einen Lebenszyklus (siehe Abb. 4.5). Wird eine neue Instanz als Ergebnis einer Skalierungsentscheidung erzeugt, befindet sich diese zunächst im *Pending*-Zustand. Der *Pending*-Zustand beschreibt eine Instanz die allokiert werden muss, allerdings vom Autoscaler noch nicht verarbeitet worden ist. Aus dem *Pending*-Zustand kann eine Instanz in den *Progress*-Zustand überführt werden. Der *Progress*-Zustand ist ein Zustand, in dem der Autoscaler die Allokierung einer Rechenressource auf einer Cloud-Plattform vornimmt, in dem diese jedoch noch nicht einsatzbereit ist. Aus dem *Progress*-Zustand können drei weitere Zustände erreicht werden. Der *Failure*-Zustand ist ein Endzustand, der beschreibt, dass eine Rechenressource aus unterschiedlichen Gründen (z.B. durch Netzwerkprobleme, Fehler bei der Provisionierung, ...) nicht funktionsfähig ist. Der *Terminated*-Zustand ist ebenfalls ein Endzustand. Dieser beschreibt, dass eine Rechenressource ordnungsgemäß beendet worden ist. Aus dem *Progress*-Zustand lässt sich eine Rechenressource in den *Running*-Zustand überführen. Der *Running*-Zustand beschreibt, dass eine Ressource lauffähig ist und ordnungsgemäß eine Rechenoperation ausführen kann. Aus dem *Running*-Zustand

kann eine Ressource wieder in den *Progress*-Zustand gelangen. Dies ist beispielsweise der Fall, wenn eine laufende Instanz nicht mehr benötigt wird. Der Prozess der De-Provisionierung einer Rechenressource durch den Autoscaler wird in diesem Fall durch den *Progress*-Zustand abgebildet. Eine Instanz, die sich im *Running*-Zustand befindet, kann weiterhin in den Endzustand *Failure* gelangen, beispielsweise durch Laufzeitfehler.

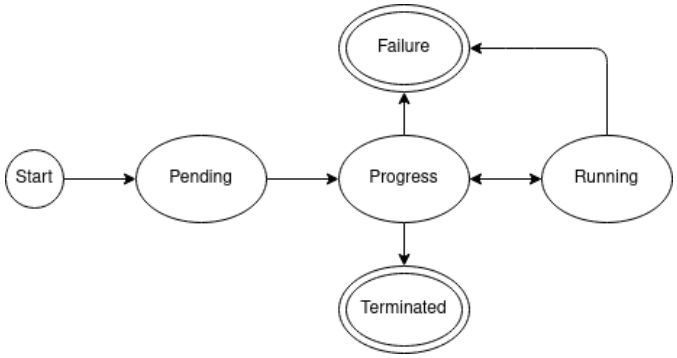


Abbildung 4.5: Zustandsdiagramm für den Lebenszyklus einer Instanz

4.2 Architektur

In diesem Abschnitt wird die Systemarchitektur des entwickelten Autoscalers vorgestellt. Hierzu werden die Komponenten der Architektur erläutert, um deren Abhängigkeiten und die benötigten Schnittstellen darzustellen.

Hybrid-Clouds kombinieren mindestens zwei der in Abschnitt 2.1.1 genannten Bereitstellungsmodelle miteinander. Die verbundenen Cloud-Infrastrukturen sind jedoch trotz Kombination der Plattformen jede für sich selbst einzigartige, selbständig funktionierende Einheiten. Unter Berücksichtigung dieser Eigenschaft wurde die im Rahmen dieser Arbeit entwickelte Autoscaler-Architektur als ein verteiltes System entworfen.

Der Begriff Komponente bezieht sich abhängig vom Kontext auf unterschiedliche Systemebenen. Die entwickelte Autoscaler-Architektur als ein verteiltes System ist eine Menge von unabhängigen Komponenten, die als nebenläufige Prozesse ausgeführt werden und Nachrichten miteinander austauschen, um gemeinsam den Skalierungsprozess (vgl. Abbildung 3.1) umzusetzen. Trotz der verteilten Ausführung präsentiert sich die Autoscaler-Architektur dem Benutzer als ein einziges System. Jede Komponente des Gesamtsystems ist wiederum selbst in einzelne Komponenten unterteilt.

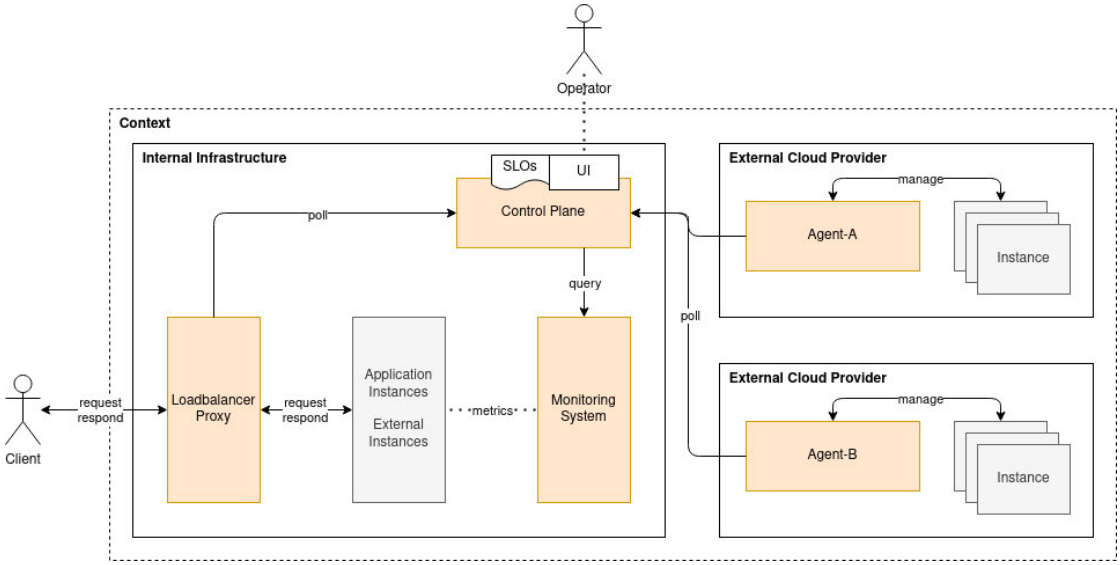


Abbildung 4.6: Systemarchitektur des verteilten Autoscalers

Die Abbildung 4.6 gibt einen Überblick über das Gesamtsystem. Dargestellt sind drei voneinander unabhängige Cloud-Infrastrukturen. Auf Grundlage von Regeln kann eine containerisierte Anwendung auf der lokalen Infrastruktur bei Bedarf zu externen Cloud-Plattformen skaliert werden. Die Komponenten der Autoscaling-Architektur orientieren sich fachlich an den in Abschnitt 3.1 vorgestellten Prozessschritten:

- *Überwachung*: Das Monitoring von Anwendungssystemen ist Voraussetzung für die automatische Skalierung von Ressourcen. Autoscaling ist jedoch nur ein Teilgebiet, in dem Monitoring angewendet wird. Das Monitoring von verteilten Anwendungen in Cloud-Umgebung besitzt ganz eigene Herausforderungen [8]. Deshalb wurden in den letzten Jahren viele Strategien und Tools entwickelt, die dabei helfen, Server zu überwachen, wichtige Daten zu sammeln und auf Vorfälle und veränderte Bedingungen in unterschiedlichen Laufzeitumgebungen zu reagieren. Aus diesem Grund nutzt die Autoscaler-Architektur keine eigene Implementierung für die Überwachung von zu skalierenden Ressourcen, sondern nutzt bestehende Monitoring-Lösungen um Daten zu sammeln und zu aggregieren.

- *Berechnung*: In der Control-Plane Komponente werden die gelieferten Daten verarbeitet und geprüft, ob Skalierungsoperationen vorgenommen werden müssen. Die Control-Plane ist eine zustandsbehaftete Komponente, in der die Services einer

Anwendung mit benutzerspezifischen Skalierungsregeln definiert wurden. Auf Basis dieser Informationen werden die für die Skalierung notwendigen Messgrößen regelmäßig aus dem Monitoring-System über einen Polling-Ansatz abgefragt.

- *Planung*: Sowohl die Berechnung als auch die Planung, wie die berechneten Skalierungsoperationen ausgeführt werden sollen, werden von der Control-Plane übernommen. Während der Planung wird der Zustand der Control-Plane verändert und Ressourcen in Form von Instanzen hinzugefügt oder zur Löschung freigegeben.
- *Ausführung*: Die Ausführung des Skalierungsplans über providerspezifische Schnittstellen wird von Agent-Komponenten übernommen. Ein Agent übernimmt die Steuerung einer Gruppe an Ressourcen auf einer Cloud-Plattform. Für jede angebundene Cloud-Plattform wird ein Agent gestartet, der in regelmäßigen Abständen den aktuellen Zustand der Instanzen von der Control-Plane bezieht. Gemäß des ermittelten Zustandes werden Instanzen auf der Cloud-Plattform provisioniert oder terminiert. Anders als die Control-Plane ist ein Agent zustandslos und führt lediglich in Intervallen eine festgelegte Steuerungsroutine durch.

Die Autoscaler-Architektur führt den Überwachungs- und Ausführungsschritt verteilt durch. Der Berechnungs- und Planungsschritt wird hingegen in einer gemeinsamen Komponente, der Control-Plane, umgesetzt. Eine weitere Komponente, die in Abbildung 4.6 dargestellt ist, ist der Loadbalancer (bzw. Proxy). Ebenso wie das Monitoring ist die Lastverteilung zwischen provisionierten Instanzen wichtig, um die Wirkung der durchgeführten Skalierungsoperation zu erzeugen. Die Lastverteilung ist allerdings nicht Bestandteil des Skalierungsprozesses. Über die Control-Plane Komponente können aktive Instanzen mitsamt erreichbaren Endpunkten abgefragt werden. Hierdurch ist es möglich, beliebige Loadbalancer-Implementierungen für die Lastverteilung zwischen lokaler und externer Infrastruktur anzubinden.

4.2.1 Control-Plane

Die Control-Plane ist die zentrale Komponente in der Autoscaler-Architektur. Ein Überblick über die Systemarchitektur der Control-Plane wird in Abbildung 4.7 gegeben.

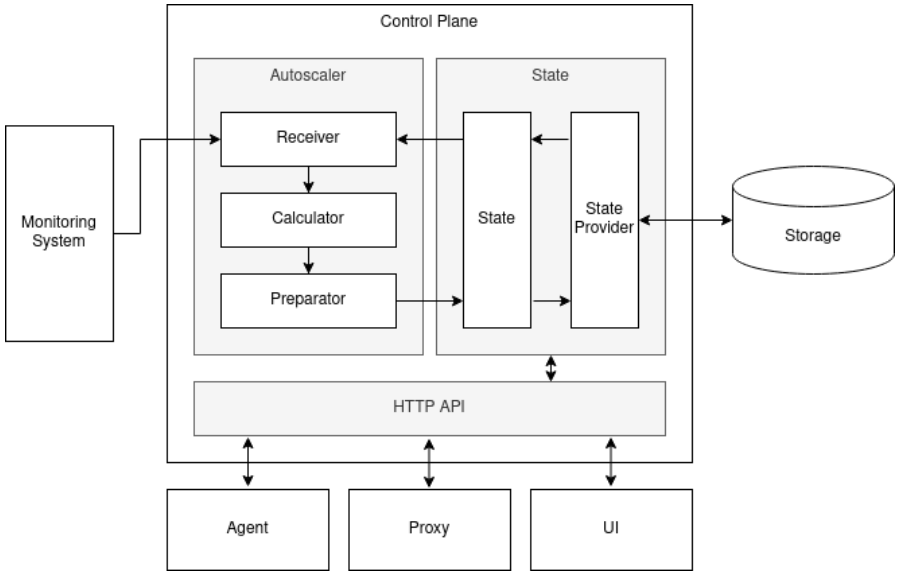


Abbildung 4.7: Architektur der Control-Plane

Die Control-Plane besteht aus drei Hauptkomponenten. Die Autoscaler-Komponente führt in festen Intervallen eine Berechnung des Skalierungsbedarfs durch. Die State-Komponente stellt eine Schnittstelle zu externen Speicherlösungen bereit über die der Zustand der Control-Plane persistiert werden kann und kapselt Implementierungsdetails. Die HTTP-API macht den Zustand der Control-Plane über eine Webschnittstelle verfügbar für die providerspezifischen Agents, den Proxy zur Lastverteilung und die grafische Benutzeroberfläche.

In Folgendem werden die Subkomponenten der Architektur erläutert.

Receiver

Die Receiver-Komponente bezieht Zeitreihendaten aus dem externen Monitoring-System, die als Grundlage für Skalierungsentscheidungen für einen Service dienen. Die Aggregation und Speicherung von Metriken für die Skalierung wird somit in der Autoscaler-Architektur komplett in ein externes Monitoring-System ausgelagert.

```
1 type MetricsReceiver interface {
2   // Poll performs a metrics query to an external monitoring tool and returns
      the result of the metrics query as a numerical value.
3   Poll(query string) (float64, error)
```

```
4   // PollFrom performs a metrics query to an external monitoring tool and
      returns the result of the metrics query as a numerical value. In addition
      , another URL can be specified to which the metrics query is sent.
5   PollFrom(url string, query string) (float64, error)
6 }
```

Listing 4.1: Spezifikation der Receiver-Subkomponente

Für die Receiver-Komponente ist ein Interface spezifiziert, welches in Listing 4.1 dargestellt ist. Die Receiver-Komponente sendet eine HTTP-Anfrage mit einer Metrikabfrage an die Webschnittstelle des Monitoring-Systems und dieses liefert den Ergebniswert des Ausdrucks als Rückgabewert zurück.

Calculator

Die Calculator-Komponente berechnet den Ressourcenbedarf für ein Skalierungsintervall. Als Grundlage für die Berechnung dient der zuvor durch die Receiver-Komponente bezogene Ergebniswert des Ausdrucks der Metrikabfrage und der aktuelle Zustand aus der State-Komponente.

```
1   type ScalingCalculator interface {
2     // Calculate calculates the resource demand for a scaling interval.
      Calculate is called regularly and takes a service, the current state of
      the control plane and the result of the metric query and returns a result
      .
3     Calculate(svc *Service, instances []*Instance, metricValue float64)
      Result
4   }
```

Listing 4.2: Spezifikation der Calculator-Subkomponente

Auch für die Calculator-Komponente ist ein Interface (vgl. Listing 4.2) spezifiziert, wodurch sich unterschiedliche Skalierungsansätze (proaktiv, reaktiv oder hybrid) implementieren lassen. Somit kann die Architektur flexibel um weitere Skalierungsalgorithmen erweitert werden, ohne dass andere Komponenten der Autoscaler-Architektur modifiziert werden müssen. Das Ergebnis der Calculator-Komponente ist eine Liste mit dem berechneten Bedarf pro zu skalierendem Cloud-Provider.

Preparator

Die Preparator-Komponente nimmt das Ergebnis der Berechnung von der Calculator-Komponente entgegen und modifiziert den Zustand der Control-Plane so, dass der aktuelle Ist-Zustand nach Bearbeitung durch die Agents dem berechneten Soll-Zustand an Instanzen für einen Cloud-Provider entspricht. Die Anpassung zwischen Soll- und Ist-Zustand wird unter Berücksichtigung des Instanzen-Lebenszyklus (siehe Kapitel 4.5) durchgeführt.

State

Die State-Komponente bietet den Komponenten der Control-Plane eine Schnittstelle an, mit welcher Daten persistiert und modifiziert werden können. Die Autoscaler-Komponente der Control-Plane greift zu Beginn eines Skalierungsintervalls auf den aktuellen Zustand zu und modifiziert diesen nach Berechnung des Bedarfs. Nebenläufige Komponenten der Autoscaler-Architektur, wie zum Beispiel der Proxy, die providerspezifischen Agents oder die Benutzeroberfläche, können indirekt über die API-Komponente in Form einer Webschnittstelle auf den Zustand der Control-Plane zugreifen. Für die State-Komponente ist ein Interface spezifiziert (vgl. Listing 4.3), durch das sich unterschiedliche Persistenzlösungen für die Autoscaler-Architektur implementieren lassen.

```
type State interface {
  // GetServices retrieves a list of all services observed by the autoscaler.
  GetServices() ([]*Service, error)
  // GetInstance retrieves an instance by service and name.
  GetInstance(service string, name string) (*Instance, error)
  // GetInstances retrieves a list of all instances associated with a service
    .
  GetInstances(service string) ([]*Instance, error)
  // RemoveInstances removes a list of instances associated with a service.
  RemoveInstances(service string, instances []*Instance) error
  // RemoveInstance removes an instance associated with a service.
  RemoveInstance(service string, instance *Instance) error
  // SaveInstances stores a list of instances associated with a service.
  SaveInstances(service string, instances []*Instance) ([]*Instance, error)
  // SaveInstance stores an instance associated with a service.
  SaveInstance(service string, instance *Instance) (*Instance, error)
}
```

Listing 4.3: Spezifikation der State-Subkomponente

Persistenzlösungen müssen hierbei Transaktionen unterstützen und bei der Ausführung von Transaktionen das ACID[1]-Prinzip garantieren. Die Änderungen am Zustand der Control-Plane müssen als eine zusammenhängende, logische Einheit entweder ganz und fehlerfrei oder gar nicht ausgeführt werden.

API

Die API-Komponente der Control-Plane stellt eine Webschnittstelle bereit, über die sich ihr Zustand abrufen und modifizieren lässt. Die Webschnittstelle berücksichtigt die REST-Prinzipien und bietet somit eine einheitliche Schnittstelle für die zustandslose Kommunikation zwischen den Komponenten der Autoscaler-Architektur.

Für die in Abschnitt 4.1 vorgestellten Datentypen werden hauptsächlich Lesezugriffe über die API-Komponente bereitgestellt. So können Informationen über zu skalierende Services nicht modifiziert und Instanzen können nur von der Control-Plane erstellt oder gelöscht werden. Lediglich für die Agents gibt es eine Schnittstelle, über die sich der Status einer Instanz (z.B. die Phase innerhalb des Lebenszyklus) aktualisieren lässt.

4.2.2 Agents

Die Agents sind verantwortlich für die Allokierung und die Freigabe von Ressourcen bei einem externen Cloud-Provider gemäß des Zustandes der Control-Plane. Die Agent-Komponente ist zustandslos und führt in festgelegten Intervallen eine Ausführungsroutine aus.

```
1 type InstanceLifecycleHandler interface {
2   // CreateInstance takes an instance and deploys it within the cloud
      provider.
3   CreateInstance(ctx context.Context, inst *Instance) error
4   // UpdateInstance takes an instance and updates it within the cloud
      provider.
5   UpdateInstance(ctx context.Context, inst *Instance) error
6   // DeleteInstance takes an instance and deletes it from the cloud provider.
7   DeleteInstance(ctx context.Context, inst *Instance) error
8   // GetInstance retrieves an instance by name from the cloud provider (can
      be cached). The instance returned is expected to be immutable, and may be
       accessed concurrently outside of the calling goroutine.
```

[1] Atomicity, Consistency, Isolation, Durability

```
9   GetInstance(ctx context.Context, namespace, name string) (*Instance, error)
10  // GetInstanceStatus retrieves the status of an instance by name from the
      cloud provider. The InstanceStatus returned is expected to be immutable,
      and may be accessed concurrently outside of the calling goroutine.
11  GetInstanceStatus(ctx context.Context, namespace, name string) (*
      InstanceStatus, error)
12  // GetInstances retrieves a list of all instances running on the cloud
      provider (can be cached). The Instances returned are expected to be
      immutable, and may be accessed concurrently outside of the calling
      goroutine.
13  GetInstances(context.Context) ([]*Instance, error)
14 }
```

Listing 4.4: Spezifikation des InstanceLifecycleHandlers zur Integration von Cloud-Plattformen

Agents kapseln providerspezifische Implementierungen in Form eines Interfaces (siehe 4.4), um Ressourcen in Form von Software-Containern über die Schnittstellen einer bestimmten Cloud-Plattform zu provisionieren. Ein Agent kann flexibel ausgeführt werden, entweder auf der eigenen, lokalen Infrastruktur oder direkt auf der Cloud-Plattform des Cloud-Providers.

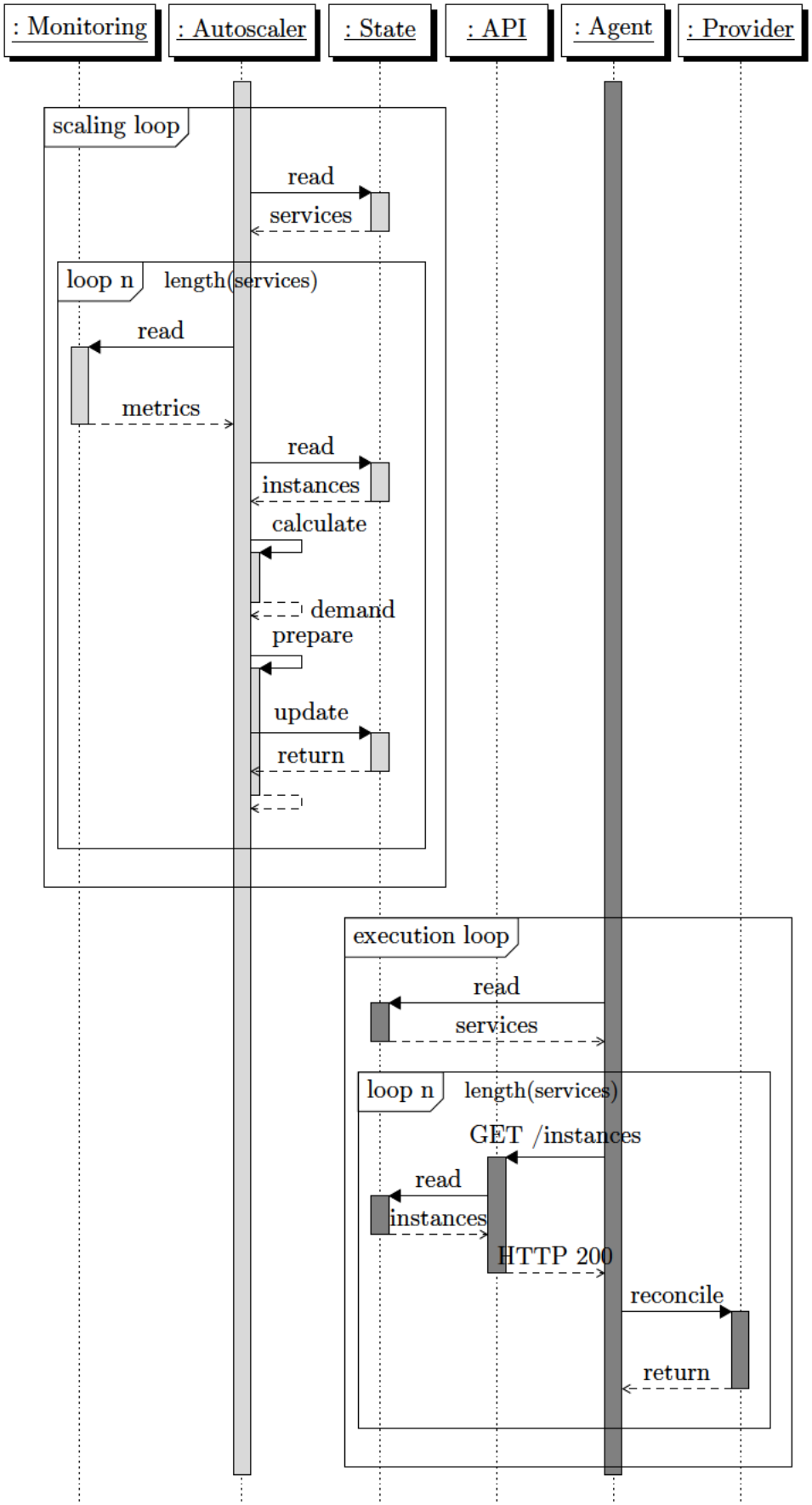
4.2.3 Proxy

Damit die Anfragen eines Klienten zwischen laufenden Instanzen auf der lokalen Infrastruktur und provisionierten Instanzen bei einem Cloud-Provider verteilt werden können, wird eine Proxy-Komponente in Form eines Loadbalancers zwischen der zu skalierenden Anwendung und den Klienten platziert.

Über die API-Komponente der Control-Plane können aktive Instanzen und ihre Endpunkte bezogen werden. Als *Single Source of Truth* lassen sich somit provisionierte Instanzen in bestehende Service-Discovery-Systeme integrieren. Gemäß der Entwurfsrichtlinie des Single-Responsibilty Prinzips wird die Lastverteilung nicht von der Autoscaler-Architektur selbst sondern von externen Lösungen umgesetzt.

4.3 Skalierungstechnik

In diesem Abschnitt wird auf den Skalierungsprozess der Autoscaler-Architektur eingegangen. Nachdem zuvor die Architektur des Autoscalers mit Komponenten und Schnittstellen im Vordergrund stand, wird in diesem Abschnitt der Fokus auf den konkreten Ablauf der Skalierung gelegt.


: Monitoring
: Autoscaler
: State
: API
: Agent
: Provider
scaling loop
read
services
loop n
length(services)
read
metrics
read
instances
calculate
demand
prepare
update
return
execution loop
read
services
loop n
length(services)
GET /instances
read
instances
HTTP 200
reconcile
return

In dem Sequenzdiagramm (siehe Abbildung 4.3) ist der Ablauf des Skalierungsprozesses visualisiert. Zu Beginn eines Skalierungsintervalls bezieht der Autoscaler die zu skalierenden Services einer Anwendung von der State-Komponente. Anschließend wird über jeden Service iteriert und Elastizität erzeugt.

Zunächst wird eine Metrikabfrage an das externe Monitoring-System gesendet und der Ergebniswert des Ausdrucks an den Autoscaler zurückgeliefert. Weiterhin wird der aktuelle Zustand über Instanzen eines Services von der State-Komponente bezogen. Das Ergebnis des Ausdrucks der Metrikabfrage und der aktuelle Zustand über die Instanzen eines Services dienen als Eingangsgrößen für die Berechnung des Bedarfs. Die Planung der Skalierungsoperation wird von der Preparator-Komponente übernommen. Hierzu modifiziert sie den Zustand der aktuellen Instanzen eines Services über die State-Komponente, gemäß des Ergebnisses der zuvor durchgeführten Berechnung des Bedarfs für das aktuelle Skalierungsintervall. Die in Abbildung 4.3 visualisierte Skalierungsroutine kann auch als Algorithmus (vgl. Algorithmus 1) dargestellt werden.

Algorithm 1: autoscaling loop

$t \leftarrow$ time duration in seconds
while *running* **do**
    $S \leftarrow$ list of services from state
    foreach $s \in \mathcal{S}$ **do**
        $V \leftarrow$ threshold percentage ratio calculated by metric query q
        $I \leftarrow$ list of instances for s from state
        $B \leftarrow calculate(s, I, V)$
        $prepare(B, I)$
    sleep t

Neben der Überwachung, Berechnung und Planung der automatischen Skalierung von Ressourcen wird die Ausführung der Skalierung asynchron durch die Agents durchgeführt. Ein providerspezifischer Agent fragt über die API-Komponente der Control-Plane regelmäßig die aktuellen Services ab. Anschließend wird über alle erhaltenen Services iteriert. Für einen Service wird der aktuelle Zustand der Instanzen, die auf der gewählten Cloud-Plattform provisioniert werden sollen, über die API-Komponente von der Control-Plane bezogen und über die Schnittstellen der Cloud-Plattform umgesetzt.

In den folgenden Teilabschnitten wird auf die einzelnen Abläufe gemäß der in Abschnitt 3.1 vorgestellten Prozessschritte eingegangen.

4.3.1 Berechnung (Calculator)

In der Berechnungsphase ermittelt die Calculator-Komponente für jeweils einen Service den Bedarf an Ressourcen. Bei jedem Evaluationsintervall i, wird der Zustand $State_i$ aus dem Zustand der Control-Plane zu einem Zeitpunkt t_i und dem Ergebnis des Ausdrucks der Metrikabfrage q_i ermittelt. Daraus wird ein Bedarf an Ressourcen B_i berechnet.

Der Zustand $State_i$ ist definiert als

$$State_i = \left(I_{(in,r)_i}, P, M_{(ex,pe)_i}, M_{(ex,r)_i}, M_{(ex,pr)_i}, M_{(ex,md)_i}, q_i\right)$$

wobei

$I_{(in,r)}$ = Anzahl Instanzen *intern* mit Zustand *Running*

P = Liste von Paaren (p, A_p) mit Provider p und Prozentsatz der bei dem Provider zu allokierenden externen Instanzen A_p

$M_{(ex,pe)}$ = Liste von Paaren $(p, I_{(ex,p,pe)})$ mit Provider p und Instanzen *extern* bei Provider p mit Zustand *Pending* $I_{(ex,p,pe)}$

$M_{(ex,r)}$ = Liste von Paaren$(p, I_{(ex,p,r)})$ mit Provider p und Instanzen *extern* bei Provider p mit Zustand *Running* $I_{(ex,p,r)}$

$M_{(ex,pr)}$ = Liste von Paaren $(p, I_{(ex,p,pr)})$ mit Provider p und Instanzen *extern* bei Provider p mit Zustand *Progress* $I_{(ex,p,pr)}$

$M_{(ex,md)}$ = Liste von Paaren $(p, I_{(ex,p,md)})$ mit Provider p und Instanzen *extern* bei Provider p als zu löschend markiert $I_{(ex,p,md)}$

q = Ergebnis des Ausdrucks der Metrikabfrage

Das Ergebnis der Berechnungsphase ist eine Bedarfsliste B_i von Paaren $(p, B_{i,p})$ mit Provider p und der Zahl der für den Provider zu allokierenden (positive Ganzzahl) oder zu terminierenden (negative Ganzzahl) Instanzen $B_{i,p}$. Ziel der Berechnung von B_i aus dem Zustand $State_i$ ist es, externe Ressourcen so zu skalieren, dass ein optimales $State_{i+1}$ herbeigeführt wird. Die Bewertung des $State_{i+1}$ ist abhängig von den benutzerspezifischen Performanz-, Kosten- oder Verfügbarkeitszielen, abgebildet durch SLOs.

Die Variablen des Zustandsvektors $State_i$ werden einerseits statisch und andererseits dynamisch ermittelt. Für interne Ressourcen und die Liste von Providern sowie deren Gewichtung werden die Variablen statisch aus den Angaben einer Systemkonfiguration bezogen. Die Metrikabfrage zur Ermittlung von q wird ebenfalls vor dem Start des Autoscalers durch den Benutzer festgelegt. Für externe Ressourcen werden die Variablen anhand des Status der Instanzen zur Laufzeit dynamisch ermittelt.

In dieser Arbeit wurde ein **reaktiver**, **thresholdbasierter Skalierungsalgorithmus** implementiert. Der Skalierungsalgorithmus misst durch eine anwendungsspezifische Metrik, in welchem Verhältnis V die benötigte Zahl der Instanzen (Soll-Zustand) zu der aktuellen Zahl der Instanzen steht (Ist-Zustand). Eine Annahme bei der Berechnung des Ressourcenbedarfs für ein Zeitintervall i ist, dass die Leistung von internen und extern allokierten Instanzen für alle Cloud-Plattformen vergleichbar ist.

Algorithm 2: reactive threshold-based demand calculation

Input: service s, list of instances I; threshold percentage ratio V

Output: demand B

$sum_{(in,r)} \leftarrow countInternalRunning(I)$
$sum_{(ex,r)} \leftarrow countExternalRunning(I)$
$b_{all} \leftarrow V * (sum_{(in,r)} + sum_{(ex,r)})$
$b_{ex} \leftarrow b_{all} - sum_{(in,r)}$
$result \leftarrow$ init list
foreach $p \in s.providers$ **do**
    $b_p \leftarrow round(b_{ex}/p.weight)$
    $b_p \leftarrow b_p - countRunningInstances(p, I)$
    $i \leftarrow (p, b_p)$
    $result.add(i)$
$rnd \leftarrow b_{ex} - result.sum$
if $rnd \neq 0$ **then**
    $p \leftarrow s.providers$ where $p.weight = max$
    add $rounding\_difference$ to d_p for p in $result$
return $result$

Für die Ermittlung des Gesamtbedarfs an Ressourcen für ein Zeitpunkt i, wird zunächst ein $b_{i,ex}$, die Zahl wie viele externe Instanzen in Summe über alle Provider benötigt werden, berechnet. Sie berechnet sich als Produkt aus V und der Summe der aktiv laufenden

Instanzen ($i_{in,r} + \sum_{p \in P} i_{ex,p,r}$), wobei anschließend die Zahl der internen Instanzen $I_{(in,r)_i}$ abgezogen wird.

$$b_{i,ex} = V \times (i_{in,r} + \sum_{p \in P} i_{ex,p,r}) - i_{in,r}$$

Die in der Planungsphase benötigte Bedarfsliste B_i ergibt sich aus der Verteilung der benötigten Ressourcen auf die externen Cloud-Provider p aus P unter Berücksichtigung ihrer Gewichtung A_p. Weil die Zahl der Instanzen für den Ressourcenbedarf ganzzahlig sein muss, wird der Wert gerundet. Nach der Berechnung von B_i müssen Rundungsdifferenzen ermittelt und die Differenz geeignet verteilt werden.

$$B_i = \left\{ (p, b_{i,p}) \,|p \in P \wedge b_{i,p} = round(\frac{b_{i,ex}}{A_p}) \right\}$$

Der beschriebene Ablauf ist in Algorithmus 2 dargestellt und ist eine Ausführung des Funktionsaufrufes $B \leftarrow calculate(s, I, V)$ aus Algorithmus 1.

4.3.2 Planung (Preparator)

In der Planungsphase wird der Zustand der Control-Plane gemäß des Bedarfs B_i für ein Zeitintervall i aus der Berechnungsphase angepasst. Für die Anpassung ist die Preparator-Komponente zuständig. Anhand des berechneten Bedarfs $B_{i,p}$ für einen Provider p werden Datenstrukturen von Instanzen erzeugt oder bestehende Datenstrukturen aktualisiert, sodass sie zur Bearbeitung durch die Agents vorbereitet sind.

Unter Berücksichtigung der bereits laufenden externen Instanzen $i_{ex,p,r}$, bereits in Erstellung befindlichen externen Instanzen $i_{ex,p,pr}$ und zur Terminierung vorgemerkten externen Instanzen $i_{ex,p,md}$ ergibt sich pro Provider die zusätzlich zu startende bzw. zu terminierende Anzahl $delta_{i,p}$ von Instanzen.

$$delta_{i,p} = B_{i,p} - i_{ex,p,r} - i_{ex,p,pr} + i_{ex,p,md}$$

Die Instanzen des Providers mit dem Status *Pending* sind angefordert, wurden allerdings noch nicht durch einen Agent bearbeitet und damit in den Status *Progress* versetzt. Bei $delta_{i,p} \leqq 0$ werden sie verworfen, bei $delta_{i,p} > 0$ bleiben $delta_{i,p}$ Instanzen mit Status *Pending* erhalten.

Um einem Flattern um eine Anzahl von Instanzen bei gleichbleibendem Lastverhalten entgegenzuwirken, kann ein Threshold von einem Benutzer festgelegt werden, innerhalb dessen keine Veränderungen an der Anzahl von Instanzen im Zustand vorgenommen wird. Alle Werte von zu startenden oder zu löschenden Instanzen, die innerhalb des Wertebereiches des Tresholds liegen, werden auf 0 gesetzt. Das Setzen eines Thresholds hat zur Folge, dass die Skalierungssteuerung weniger empfindlich gegenüber Schwankungen um einen Wert herum wird, gleichzeitig wird die Steuerung hierdurch träger.

Algorithm 3: demand preparation

Input: demand B, list of instances I

Data: threshold t

foreach $(p, b_p) \in \mathcal{B}$ **do**

- $sum_{(p,r)} \leftarrow countRunningInstances(p, I)$
- $sum_{(p,pr)} \leftarrow countProgressInstances(p, I)$
- $sum_{(p,t)} \leftarrow countTerminatingInstances(p, I)$
- $sum_{(p,pe)} \leftarrow countPendingInstances(p, I)$
- $delta_p \leftarrow b_p - sum_{(p,r)} - sum_{(p,pr)} + sum_{(p,t)}$
- **if** $(delta_p \geqq t.min \wedge delta_p \leqq t.max)$ **then**
  - $delta_p \leftarrow 0$
- **if** $delta_p == 0$ **then**
  - remove all *pending* instances for provider p from I
- **else if** $delta_p > 0$ **then**
  - $delta_p \leftarrow delta_p - sum_{(p,pe)}$
  - **if** $delta_p \geqq 0$ **then**
    - add $delta_p$ *pending* instances for provider p from I
  - **else**
    - remove $|delta_p|$ *pending* instances for provider p from I
- **else if** $delta_p < 0$ **then**
  - remove all *pending* instances for provider p from I
  - $i_{(p,r)} \leftarrow getRunningInstances(p, I)$
  - mark $|delta_p|$ instances from $i_{(p,r)}$ as *terminating*

Der dargestellte Algorithmus 3 entspricht dem Funktionsaufruf $prepare(B, I)$ aus Algorithmus 1.

Der angepasste Zustand in der Planungsphase wird im Anschluss durch die Agents umgesetzt. Die Ausführungsphase findet asynchron statt. Nach Beendigung der Planungsphase ist der von der Control-Plane durchgeführte Teilprozess der Skalierung abgeschlossen.

4.3.3 Ausführung (Agent)

Die Ausführungsphase im Skalierungsprozess wird asynchron durch die Agents durchgeführt. Für jeden genutzten Cloud-Provider wird genau ein Agent gestartet, welcher Instanzen über die providerspezifischen Schnittstellen auf einer Cloud-Plattform provisioniert. Agents fragen in festgelegten Intervallen den geplanten Zustand über die für sie relevanten Instanzen ab und setzen diesen um. Instanzen mit Status Pending werden in den Status Progress versetzt, wodurch signalisiert wird, dass der Agent die zu provisionierende Instanz verarbeitet. Jede Zustandstransition wird dabei von dem Agent zurück an die Control-Plane gemeldet und wird bei Skalierungsentscheidungen berücksichtigt (siehe Algorithmus 2). Anschließend werden die Instanzen auf der Cloud-Plattform gestartet und je nach Erfolg als *Running* oder *Failure* gekennzeichnet. Laufende Instanzen, die von der Control-Plane als zu terminierend gekennzeichnet wurden, werden auf der Cloud-Plattform beendet. Auch hier erfolgt eine Statusmeldung an die Control-Plane.

In Algorithmus 4 ist die Ausführungsroutine des Agents dargestellt.

Algorithm 4: agent execution routine

```
agent initialization for provider p
while running do
    S ← get list of services from control plane
    foreach s ∈ 𝒮 do
        I_p ← get list of instances for s with provider p from control plane
        I_(p,pe) ← get list of pending instances I_(p,pe) ⊆ I_p
        I_(p,md) ← get list of not active instances I_(p,md) ⊆ I_p
        foreach i ∈ I_(p,pe) ∪ I_(p,md) do
            set status of i as progress in control plane
        foreach i ∈ I_(p,pe) do
            provision instance at cloud provider p
            if success then
                set status of i as running in control plane
            else
                set status of i as failure in control plane
        foreach i ∈ I_(p,md) do
            terminate instance i at cloud provider p
            if success then
                set status of i as terminated in control plane
            else
                set status of i as failure in control plane
```

Neben dem kontrollierten Starten und Stoppen von Instanzen durch die Agents kann es zu Störungen kommen. Beispielsweise können Instanzen zur Laufzeit ihren Dienst verweigern, was zu Folge hat, dass keine Anfragen mehr bearbeitet werden können. Hierdurch kann der tatsächliche Zustand der Instanzen von dem in der Control-Plane erfassten Zustand abweichen.

Der Umgang mit diesen Störungen und Maßnahmen durch die Autoscaler-Architektur werden im folgenden Abschnitt thematisiert.

4.3.4 Umgang mit fehlerhaften Instanzen

In Kapitel 3.1 wurde Erkennen und Ersetzen von fehlerhaften oder nicht erreichbaren Ressourcen als eine Anforderungung an einen Autoscaler definiert. Beim asynchronen

Zusammenspiel des Autoscaler mit der Cloud-Plattform kann es in der Praxis zu Abweichungen zwischen den in der Control Plane registrierten Instanzen und den tatsächlich vorhandenen Instanzen kommen, wenn z.B. Statusmeldungen durch den Agent fehlschlagen oder Instanzen außerplanmäßig terminieren. Diese Abweichungen können durch eine regelmäßige Synchronisation behoben werden.

Neben den im Autoscaler persistierten Zustandsinformationen der Instanzen wird für einen Abgleich auch der tatsächliche Zustand der Instanzen benötigt. Die Agents als Schnittstelle zur jeweiligen Cloud-Plattform können in regelmäßigen Abständen diesen Abgleich vornehmen. Folgende Fälle für Abweichungen lassen sich unterscheiden:

- *Eine Instanz ist in der Control-Plane mit Status „Running“ vermerkt, ist allerdings nicht aktiv auf der externen Cloud-Plattform*

  Auswirkung: Die Control-Plane führt die Berechnungsphase mit fehlerhafter Anzahl laufender Instanzen durch und berechnet somit dem Bedarf fehlerhaft.

  Maßnahme: Der Agent setzt die Instanz in der Control-Plane auf *Terminated* und sie wird somit in dem nächsten Skalierungsinterval entfernt.

- *Eine Instanz läuft auf der externen Cloud-Plattform, ist aber im Zustand der Control-Plane nicht vermerkt*

  Auswirkung: Die Instanz auf der externen Cloud-Plattform erhält keine Anfragen und wird auch nicht mehr von der Control-Plane verwaltet.

  Maßnahme: Der Agent terminiert die Instanz auf der externen Cloud-Plattform. Es muss kein Status in der Control-Plane für die Instanz aktualisiert werden, da diese nicht bekannt ist.

- *Eine Instanz läuft auf der externen Cloud-Plattform und ist der Control-Plane bekannt. Health-Checks zeigen, dass die Instanz nicht reagiert.*

  Auswirkung: Die Control-Plane führt die Berechnungsphase mit fehlerhafter Anzahl laufender Instanzen durch und berechnet somit dem Bedarf fehlerhaft. Weiterhin erhält die nicht reagierende Instanz Anfragen.

  Maßnahme: Der Agent terminiert die Instanz auf der externen Cloud-Plattform und aktualisiert den Status der Instanz in der Control-Plane auf *Terminated*.

- *Eine Instanz läuft auf der externen Cloud-Plattform und ist der Control-Plane bekannt. Health-Checks zeigen, dass die Instanz ordnungsgemäß reagiert.*

  Auswirkung: Keine

  Maßnahme: Keine

Ein Umgang mit den Abweichungen ist in Algorithmus 5 dargestellt. Die Synchronisation kann zu Beginn oder zum Ende einer Ausführungsroutine eines Agents in regelmäßigen Abständen durchgeführt werden.

Algorithm 5: agent synchronisation

agent initialization for provider p
while *running* **do**
    [agent execution]
    if *agent_sync_trigger(p)* **then**
        $I_{(p)} \leftarrow$ query list of instances for all services for provider p from control plane
        $I_{(p,r)} \leftarrow$ *running* instances $I_{(p,r)} \subseteq I_{(p)}$
        $J_{(r)} \leftarrow$ get list of *running* instances from provider p
        foreach $i \in (I_{(p,r)} \setminus J_{(r)})$ **do**
            set status of i as *terminated* in control plane
        foreach $i \in (J_{(r)} \setminus I_{(p,r)})$ **do**
            terminate instance at provider p
        foreach $i \in (J_{(r)} \cap I_{(p,r)})$ **do**
            if $ping(i) \neq success$ **then**
                terminate instance at provider p
                set status of i as *terminated* in control plane

Bei einer gefundenen Abweichung kann ermittelt werden, in welchem Intervall zwischen zwei Synchronisationen sie aufgetreten ist. Der genaue Zeitpunkt der Entstehung der Abweichung kann allerdings nicht festgestellt werden. Somit können auch keine Rückschlüsse auf die Korrektheit des letzten Skalierungsprozesses gezogen werden.

4.4 Implementierung

Dieser Abschnitt beschreibt Details zur Implementierung der Autoscaler-Architektur und gibt einen Überblick über die verwendeten Technologien. Das Ergebnis der Umsetzung ist ein Autoscaler, der einzelne Services aus einer Anwendung durch einen reaktiven Skalierungsalgorithmus zu mehreren externen Cloud-Providern skalieren kann. Die Funktionsweise des Autoscalers ist in Abbildung 4.8 dargestellt.

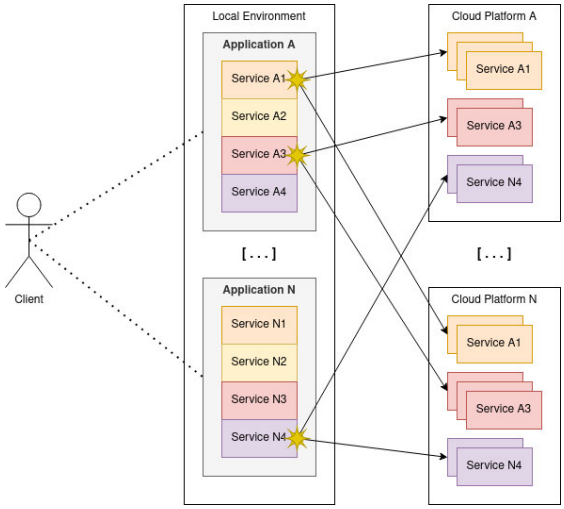


Abbildung 4.8: Darstellung einer Skalierungsoperation durch die Autoscaler-Architektur. Eine aus mehreren Services bestehende Applikation wird zu mehreren Cloud-Plattformen skaliert.

4.4.1 Technologien

Alle Komponenten der Autoscaler-Architektur sind in der Programmiersprache Go geschrieben worden. Die Control-Plane nutzt eine OpenAPI Spezifikation zur Beschreibung der Webschnittstelle und zur Generierung von Client-Bibliotheken. So verwenden beispielsweise die providerspezifischen Agents einen durch die OpenAPI Spezifikation generierten Go-Client zur Kommunikation mit der Control-Plane.

Wie in Abschnitt 4.2 erläutert, muss eine Lastverteilung zwischen provisionierten Ressourcen stattfinden. Der Autoscaler selbst führt selbst keine Lastverteilung durch, sondern nutzt externe Komponenten zur Verteilung von Anfragen auf aktive Instanzen. Die implementierte Autoscaler-Architektur nutzt als Loadbalancer das in Go geschriebene

Open-Source Projekt Skipper[2] von dem Unternehmen Zalando[3]. Skipper ist ein HTTP-Router und Reverse-Proxy für die Lastverteilung zwischen Services. Im Rahmen dieser Arbeit wurde Skipper durch eine eigene DataProvider Implementierung erweitert, um aktive Instanzen von der Control-Plane-Komponente abzufragen und in eine Routenkonfiguration für den Loadbalancer zu übersetzen. Skipper nutzt einen Polling-Ansatz um regelmäßig die Endpunkte von aktiven Instanzen von der Control-Plane abzufragen, Routen zu konfigurieren und Anfragen zu verteilen. Die erfolgreiche Anbindung von Skipper an die implementierte Autoscaler-Architektur zeigt, dass sich diese mit bestehenden Lösungen zur Lastverteilung integrieren lässt.

Weiterhin wird das Open-Source Projekt Prometheus[4] als externes Monitoring-Tool verwendet. Prometheus ist ein Open-Source-Toolkit zur Systemüberwachung und Alarmierung, welches ursprünglich von SoundCloud[5] entwickelt wurde und mittlerweile von einer aktiven Entwickler-Community als Teil der Linux Foundation[6] weiterentwickelt wird. Prometheus bietet eine funktionale Abfragesprache namens PromQL (Prometheus Query Language) an, mit der Clients Zeitreihendaten in Echtzeit selektieren und aggregieren können. Das Ergebnis eines Ausdrucks kann entweder als Diagramm angezeigt oder von externen Systemen über eine Webschnittstelle konsumiert werden. Der umgesetzte Autoscaler besitzt eine Implementierung des MetricReceiver Interfaces für Prometheus, um mit dieser Webschnittstelle zu kommunizieren und Metriken für die Skalierung zu beziehen. Skalierungsregeln können somit in PromQL formuliert werden und direkt in Prometheus aggregiert werden. Das Ergebnis der Aggregation wird für den Skalierungsprozess vom Autoscaler verwendet.

Im Rahmen dieser Arbeit wurden für alle Komponenten der Autoscaler-Architektur Container-Images gemäß des OCI-Standards[7] erzeugt. Sie können somit von jeder Container-Runtime, die den OCI-Standard unterstützt, als Software-Container ausgeführt werden. Durch die Container-Runtime findet eine Entkopplung von darunterliegender Infrastruktur statt.

[2] https://github.com/zalando/skipper
[3] https://zalando.de/
[4] https://prometheus.io/
[5] https://soundcloud.com/
[6] https://linuxfoundation.org/
[7] https://opencontainers.org/

4.4.2 Konfiguration

Die Autoscaler-Architektur soll Elastizität für containerisierte Anwendungen erzeugen. Auch benutzerspezifische Skalierungsregeln und für die automatische Skalierung genutzte Metriken sollen flexibel und zentral konfigurierbar sein. Die Architektur verwendet dafür eine externe Konfigurationsdatei. Ein Beispiel für eine scalingconfig.yaml Datei ist in Abbildung 4.5 dargestellt. Die .yaml Dateierweiterung nimmt bereits vorweg, dass es sich um eine in YAML strukturierte Textdatei handelt, die einem bestimmten Beschreibungsschema folgt.

```
prometheus_url: http://prometheus:9090
  services:
  - name: "bubblesort-service"
    path: "/bubblesort"
    description: "requests amount greater than 150 rps"
    query: |
      (sum(rate(example_sorting_requests_total[15s])) / 150)
    provider:
      weights:
        cloud-provider-alpha: 50
        cloud-provider-beta: 50
    spec:
      container:
        name: "bubblesort-svc"
        image: "cbrgm/example-app:latest"
        ...
      static:
        endpoints:
        - "http://example-app:9997"
```

Listing 4.5: Ein Beispiel für eine Konfigurationsdatei **scalingconfig.yaml**

In Listing 4.5 wird ein Service mit einer Abfrage in PromQL, eine Angabe der Parameter für zu startende Container-Instanzen (*spec*) und eine Liste von lokal erreichbaren Endpunkten (*static*) spezifiziert.

Global wird die Adresse der Prometheus-Instanz definiert, zu welcher Metrikabfragen, die für den Skalierungsprozess von Instanzen eines Services benötigt werden, gesendet werden. Darauf folgt eine Liste an Services. Ein Service wird definiert durch einen einzigartigen Namen, einem Pfad unter dem der Service durch den Proxy erreichbar ist, sowie einer in PromQL formulierten Metrikabfrage. Eine Gewichtung der Cloud-Provider, in

welchem Verhältnis Ressourcen skaliert werden sollen, kann angegeben werden. Weiterhin wird definiert, wie der Container einer neu gestarteten Instanz parametrisiert werden soll und welche Endpunkte lokal zur Verfügung stehen.

4.4.3 User Interface

Die Control-Plane der Autoscaler-Architektur besitzt eine grafische Benutzeroberfläche mit einer Echtzeitaktualisierung durch Server-Sent Events (SSE). Server-Sent Events sind eine durch das W3C standardisierte Server-Push-Technologie, die es einem Client ermöglicht, automatische Updates von einem Server über eine HTTP-Verbindung zu empfangen. Bei jeder Zustandsveränderung von Instanz-Datentypen der Control-Plane wird ein Event an das Frontend gesendet, welches anschließend die Benutzeroberfläche aktualisiert.

4.4.4 Stand der Umsetzung

A1) Der Autoscaler soll die in 3.1 genannten Aufgaben und Prozessschritte umsetzen.

Die Autoscaler-Architektur setzt die Prozessschritte Überwachung, Berechnung, Planung und Ausführung um. Die Umsetzung der Prozessschritte wurde in Abschnitt 4.3 dargestellt. Ebenso unterstützt der Autoscaler ein Scale-Out/In von Ressourcen bei Veränderungen der Nachfrage. Er ist durch den Benutzer konfigurierbar (siehe Abschnitt 4.4.2) und ist in der Lage, fehlerhafte Ressourcen zu erkennen und Maßnahmen zu ergreifen (siehe Abschnitt 4.3.4).

A2) Der Autoscaler soll die Skalierung zu mehreren Cloud-Anbietern ermöglichen.

Der Autoscaler kann Services einer Anwendung individuell zu mehreren Cloud-Plattformen skalieren. Cloud-Plattformen lassen sich durch den Benutzer statisch gewichten (siehe Abschnitt 4.4.2). Diese Gewichtung wird bei Skalierungsentscheidungen berücksichtigt (siehe Abschnitt 4.3.1).

A3) Der Autoscaler soll plattformunabhängig bereitstellbar und ausführbar sein.

Die Autoscaler-Architektur ist an keine plattformspezifischen Schnittstellen gebunden. Die Control-Plane und Agents sind als Software-Container ausführbar und nutzen eine Container-Runtime zur Abstraktion von darunterliegender Infrastruktur. Die Autoscaler-Architektur kann somit auf physischer oder virtualisierter Infrastruktur ausgeführt werden, sofern eine Container-Runtime installiert ist (siehe Abschnitt 4.4).

A4) Der zu entwickelnde Autoscaler soll externe Monitoring-Systeme unterstützen.

Die Autoscaler-Architektur nutzt externe Monitoring-Systeme für die Aggregation und Speicherung von Metriken für die Skalierung. Hierzu wurde ein Interface spezifiziert, durch welches sich weitere Monitoring-Systeme anbinden lassen (siehe Abschnitt 4.2.1). In der prototypischen Implementierung im Rahmen dieser Arbeit wurde Prometheus als Monitoring-Toolkit angebunden (siehe Abschnitt 4.4).

A5) Der zu entwickelnde Autoscaler soll erweiterbar sein für die Anbindung weiterer Cloud-Plattformen.

Die Autoscaler-Architektur unterstützt die Anbindung von externen Cloud-Plattformen. Durch die Agent-Komponente werden plattformspezifische Schnittstellen gekapselt und die Autoscaler-Architektur somit entkoppelt. Um andere Cloud-Plattformen zu integrieren, wird nur eine weitere Agent-Implementierung benötigt (siehe Abschnitt 4.2.2).

A6) Der zu entwickelnde Autoscaler soll die Implementierung weiterer Skalierungsalgorithmen unterstützen.

Die Autoscaler-Architektur erlaubt die Implementierung individuell angepasster Skalierungsalgorithmen. Über das in Abschnitt 4.2.1 vorgestellte Interface können reaktive, proaktive oder hybride Skalierungsalgorithmen implementiert werden, ohne dass ein umfangreiches Refactoring von anderen Komponenten innerhalb der Architektur notwendig ist.

5 Evaluierung

In diesem Kapitel wird für die implementierte Autoscaler-Architektur evaluiert, ob das Verhalten des Autoscalers den Anforderungen entspricht. Hierzu wurde ein Beispielservice implementiert, der repräsentativ als Teil einer größeren Anwendung durch die Autoscaler-Architektur zu mehreren Cloud-Providern skaliert werden soll. Für die Skalierung des Services werden zwei verschiedene SLOs definiert und der Service anschließend typischen Lastszenarien ausgesetzt. Während der Durchführung der Lastszenarien werden Metriken von dem Autoscaler und dem zu skalierenden Service der Beispielanwendung erhoben, um das Skalierungsverhalten beobachtbar machen.

Im Folgenden wird zunächst auf den Service der Beispielanwendung, die Testumgebung und die Durchführung der Evaluierung eingegangen. Anschließend werden die Metriken für die Skalierung und die Lastszenarien erläutert. Abschließend erfolgt die Auswertung der Ergebnisse und eine zusammenfassende Diskussion (vgl. 5.4).

5.1 Beispielanwendung

Der Autoscaler ist in der Lage, eine Anwendung als Komposition auf mehreren Services zu skalieren. Die kleinste Anwendung, die durch die Autoscaler-Architektur skaliert werden kann, ist somit eine Anwendung bestehend aus einem Service. Da der Autoscaler dieselben Prozessschritte für jeden zu skalierenden Service durchführt, kann das Skalierungsverhalten für eine beliebige Menge an Services abstrahiert werden.

Als Beispielanwendung für die automatische Skalierung wurde ein zustandsloser Microservice[1] implementiert, der über eine RESTful Webschnittstelle die Sortierung von Zahlenreihen durch eine Bubblesort-Implementierung anbietet. Da der Bubblesort-Algorithmus eine durchschnittliche Laufzeit von $\Theta\left(n^2\right)$ besitzt, kann durch die Erhöhung der Zahlenreihe in einer Anfrage der Aufwand beliebig konfiguriert werden. Gestartete Instanzen

[1] https://github.com/cbrgm/cloudburst/example

können sich ohne manuelle Konfiguration in die Ausführung der Sortierungsoperation eingliedern und können einen Teil der Rechenoperationen übernehmen. Ein Container-Image wurde für den Microservice erstellt und in der Container-Registry der Gitlab-Instanz der HAW Hamburg bereitgestellt.

Neben der Sortierungslogik, welche als Dienst über eine Webschnittstelle angeboten wird, exportiert der Microservice anwendungsspezifische Metriken, die durch Prometheus als externes Monitoring-System überwacht und aggregiert werden können. Über einen Metrik-Endpunkt können niedere Metriken, wie beispielsweise CPU-Auslastung oder Speicherallokation, und höhere Metriken, wie beispielsweise Anzahl und Bearbeitungsdauer von HTTP-Anfragen, abgerufen werden.

Der Microservice wurde in Go geschrieben. Die Wahl der Programmiersprache spielt jedoch keine Rolle bei der Skalierung durch die implementierte Autoscaler-Architektur.

5.2 Testumgebung

Um die Autoscaler-Architektur evaluieren zu können, wurden sämtliche Komponenten als Software-Container auf einer auf einer Testinstanz der Infrastruktur der HAW Hamburg ausgeführt. Die Instanz besitzt einen E5-2670-Prozessor und 32 GB RAM und ist über Intel Gigabit Ethernet angebunden. Als Betriebssystem wurde Ubuntu 18.04 verwendet mit der Linux-Kernel Version 5.3.

Der in 5.1 beschriebene Beispielservice soll repräsentativ als Teil einer größeren Anwendung durch die Autoscaler-Architektur zu mehreren Cloud-Providern skaliert werden. Um die Autoscaler-Architektur unabhängig von Provider-Schnittstellen testen zu können, wird die providerspezifische Implementierung der Agents gemockt. Der Agent provisioniert keine Ressourcen auf externen Cloud-Plattformen, sondern simuliert die Provisionierung von Ressourcen durch eine Wartezeit zwischen 3 bis 10 Sekunden pro Instanz. Die durch die Mocks gestarteten Instanzen referenzieren die Adresse einer separat gestarteten *Sink*-Anwendung. Diese Sink-Anwendung besitzt dieselbe Webschnittstelle wie der in Abschnitt 5.1 vorgestellte Beispielservice. Anstatt Anfragen zu verarbeiten, werden diese jedoch verworfen. Somit wird eine horizontale Skalierung durch das Hinzufügen und Entfernen von Instanzen simuliert, da sich die Last des Beispielservices analog der Verwendung echter externer Instanzen verändert.

Alle Services können in der Testumgebung durch eine *docker-compose.yaml* Datei gestartet und gestoppt werden. Somit lässt sich die Testumgebung einfach und reproduzierbar wiederherstellen. Folgende Services wurden in der Testumgebung gestartet:

- **Control-Plane** - Führt in regelmäßigen Intervallen eine Berechnung des Skalierungsbedarfs durch.
- **Prometheus** - Sammelt und aggregiert Metriken die sowohl für die Skalierung als auch für die Auswertung der Lastszenarien benötigt werden.
- **Proxy** - Verteilt eingehende Sortierungsanfragen in Form von HTTP-Aufrufen an die aktiven Service-Instanzen.
- **Agents** - Starten und Beenden Instanzen auf Grundlage des berechneten Skalierungsbedarfs durch die Control-Plane. Um die Skalierung des Services zu drei unterschiedlichen Cloud-Providern zu simulieren, wurden insgesamt drei Agents (*agent-a, agent-b, agent-c*) auf der Testinstanz gestartet.
- **Bubblesort-Service** - Der zu skalierender Service, welcher auf der eigenen lokalen Infrastruktur ausgeführt wird. Der Beispielservice ist Ziel der Skalierung durch die Autoscaler-Architektur.
- **Bubblesort-Sink** - Nimmt Sortierungsanfragen in Form von HTTP-Aufrufen entgegen, führt allerdings keine Sortierung durch, sondern meldet lediglich die erfolgreiche Bearbeitung (*HTTP 200 OK*) dem aufrufenden Klienten zurück.

5.3 Durchführung

Zur Simulation von Benutzeranfragen wurde das Tool *Artillery*[2] verwendet. Die Lastszenarien wurden in Artillerys eigenem Konfigurationsformat beschrieben, welches in der offiziellen Dokumentation[3] des Tools nachlesbar ist. Die Skriptingmöglichkeiten von Artillery in Form von Javascript Callbacks wurden genutzt, um für jede simulierte Benutzeranfrage an den Bubblesort-Service eine zufällige Zahlenreihe mit einer festen Kapazität von 5000 Elementen zu generieren. Jedes Lastszenario besitzt eine Laufzeit von insgesamt 10 Minuten mit variablen Anfrageverläufen. Die Lastszenarien werden in Abschnitt 5.3.2 genauer erläutert.

[2] https://artillery.io/
[3] https://artillery.io/docs/

Die vorgestellte Autoscaling-Architektur ist ein verteiltes System, welches die automatische Skalierung von Ressourcen asynchron umsetzt. Deshalb gibt es mehrere Faktoren, die Einfluss auf die Erzeugung der Elastizität haben. Folgende Zeitintervalle für die Abfragen zwischen den Komponenten der verteilten Autoscaling-Architektur wurden für die Experimente verwendet:

- (*Z1*) 5 Sek. Zeitintervall für das Abfragen von Metriken zwischen dem Bubblesort-Service und Prometheus
- (*Z2*) 25 Sek. Zeitintervall für das Abfragen von Metriken und Berechnung des Ressourcenbedarfs zwischen Prometheus und der Control-Plane
- (*Z3*) 5 Sek. Zeitintervall für das Abfragen des Zustands der Control-Plane durch die Agents

Darüber hinaus gibt es weitere Faktoren, die bei der Erzeugung von Elastizität relevant sind, beispielsweise die Dauer der Berechnungs- und Planungsphase der Control-Plane oder die Dauer des Aktualisierens des Zustandes. Weiterhin spielt die Bearbeitungszeit der Anfrage eines Agents an die Control-Plane eine Rolle sowie die Dauer der Ausführungsphase durch den Agent inklusive der Bearbeitungszeit von Anfragen an die Schnittstellen des externen Cloud-Providers. Eine effiziente Skalierung von Ressourcen ist somit abhängig von einer Vielzahl unterschiedlicher Variablen. Einige können durch den Benutzer festgelegt werden (wie beispielsweise die Abfrageintervalle). Andere wiederum sind abhängig von der verwendeten Implementierung und können nur indirekt oder gar nicht durch den Benutzer beeinflusst werden.

Z1 muss wesentlich kleiner sein als *Z2*, damit möglichst aktuelle Metriken für die Berechnungsphase zur Verfügung stehen. Das Zeitintervall *Z3*, inklusive der Dauer der Ausführungsphase, muss auch wesentlich kleiner sein als *Z2*, da die Skalierung davon ausgeht, dass das gesamte Zeitintervall *Z2* mit der zuvor geplanten Anzahl von Instanzen skaliert wurde. Die Wirkung der Skalierung muss deshalb innerhalb des größten Teils des Zeitintervalls *Z2* eintreten.

Wie in Abschnitt 5.2 beschrieben, soll der Beispielservice zu drei verschiedenen Cloud-Providern skaliert werden. Die Provider wurden wie folgt gewichtet: Provider-A (50), Provider-B (40), Provider-C (10).

Ein Threshold kann gesetzt werden, um die Steuerung der Skalierung zu stabilisieren. In Vorabtests hat sich herausgestellt, dass dies in manchen Szenarien notwendig ist, zum

Beispiel wenn der Bedarf um einen Wert wie 3,5 Instanzen schwankt und die Steuerung durch das Runden auf ganzzahlige Werte zwischen drei und vier Instanzen springt. Während der Durchführung der Lastszenarien wurde der Threshold auf 0 gesetzt.

5.3.1 Metriken

Für die Evaluierung des Skalierungsverhaltens wurden zwei SLOs festgelegt, mit denen der Service skaliert werden soll. Es wird gezeigt, dass eine Skalierung durch die Autoscaler-Architektur sowohl mit anwendungsspezifischen (*Requests*), als auch mit technischen (*Memory*) SLOs umgesetzt werden kann. Die verwendeten SLOs sind:

- (**Requests**) Die durchschnittliche Anzahl an HTTP-Anfragen pro Sekunde pro Instanz soll kleiner als 120 req/s sein.
- (**Memory**) Der durchschnittliche allokierte Speicher pro Instanz soll kleiner als 30 Megabyte sein.

Für *Requests* wurde die Metrik q = durchschnittliche HTTP-Anfragen pro Sekunde pro Instanz verwendet. Das in Abschnitt 4.3.1 beschriebene Verhältnis V wird durch den Ausdruck (q/th_{max}) mit $th_{max} = 120$ berechnet.

Für *Memory* wurde die Metrik q = durschnittlich allokierter Speicher in Megabyte pro Instanz verwendet. Das Verhältnis V wird durch den Ausdruck (q/th_{max}) mit $th_{max} = 30$ berechnet.

In beiden Fällen wird die Metrik von dem zu skalierenden Service bereitgestellt und verwendet. Bei *Requests* ist eine Einhaltung der SLO möglich, ohne die Gesamtanzahl der eingehenden HTTP-Anfragen zu kennen. Bei *Memory* ist eine Begrenzung des allokierten Speichers ohne Kenntnis der Anzahl der nebenläufig durch die Instanz verarbeiteten Sortierungsanfragen möglich.

5.3.2 Lastszenarien

Als Lastszenarien wurden vier typische Anfrageverläufe in Anwendungssystemen definiert. Diese Lastszenarien werden im Folgenden erläutert:

- **Cold to Spike** - Dieses Lastszenario simuliert einen Anfrageverlauf, bei dem Anfragen zwischen zwei Extremen schwanken. Zu einem Zeitpunkt werden fast keine Anfragen an die Anwendung gestellt, während wenig später die Anfragen ohne lange Wachstumsphase rapide auf ein Maximum ansteigen.
- **Periodic Spikes** - In diesem Lastszenario wird ein Anfrageverlauf definiert, der konstant verläuft, aber in festen, wiederkehrenden Intervallen einen Anstieg von Anfragen mit Wachstumsphase aufweist.
- **Random Spikes** - Der Anfrageverlauf dieses Lastszenarios ähnelt dem des *Periodic Spikes*, mit dem Unterschied, dass das Anfragewachstum unterschiedlich stark und die Wachstumsintervalle unterschiedlich lange ausfallen.
- **Rapid Growth** - Dieses Lastszenario simuliert einen Anfrageverlauf, der zunächst konstant wächst, eine Plateauphase erreicht und anschließend langsam wieder schrumpft.

5.4 Auswertung der Ergebnisse

Prometheus stellt die notwendigen Metriken für die Skalierung von Ressourcen für den Autoscaler bereit. Gleichzeitig zeichnet das Monitoring-Tool bereitgestellte Metriken von Komponenten der Autoscaler-Architektur bezüglich des Skalierungs- und Lastverhalten auf. Um das Verhalten der Autoscaling-Architektur zu überprüfen, wurden direkt nach der Durchführung eines Lastszenarios Zeitreihen mit Messwerten von verschiedenen Metriken exportiert und anschließend aggregiert.

Für jedes Lastszenario wurden jeweils vier Auswertungen für jede der zwei SLOs erstellt. In der ersten Auswertung wird das Verhalten des Autoscalers visualisiert, in dem der durch die Control-Plane berechnete Bedarf an Instanzen zu einem Zeitpunkt dem tatsächlichen Bedarf gegenübergestellt wird. Der tatsächliche benötigte Bedarf wird außerhalb des Autoscalers berechnet, beispielsweise durch bereitgestellte Metriken über die Gesamtanzahl von HTTP-Anfragen des Proxies. Weiterhin wird die Verteilung der Instanzen mit Status Running auf die einzelnen Cloud-Provider dargestellt und mit der Gesamtanzahl von Instanzen mit Status Running verglichen, um die Aufteilung der Instanzen auf die Cloud-Provider transparent zu machen. Um die Wirkung der Skalierung durch den Autoscaler zu beobachten und somit die Einhaltung der jeweiligen SLO an dem zu skalierenden Service festzustellen, wurden die eingehenden HTTP-Anfragen bzw. der allokierte Speicher an einer Service-Instanz gemessen. Eine weitere Auswertung zeigt

den Verlauf an gesendeten HTTP-Anfragen an eine Service-Instanz, um die Lastszenarien zu visualisieren.

5.4.1 Cold to Spike

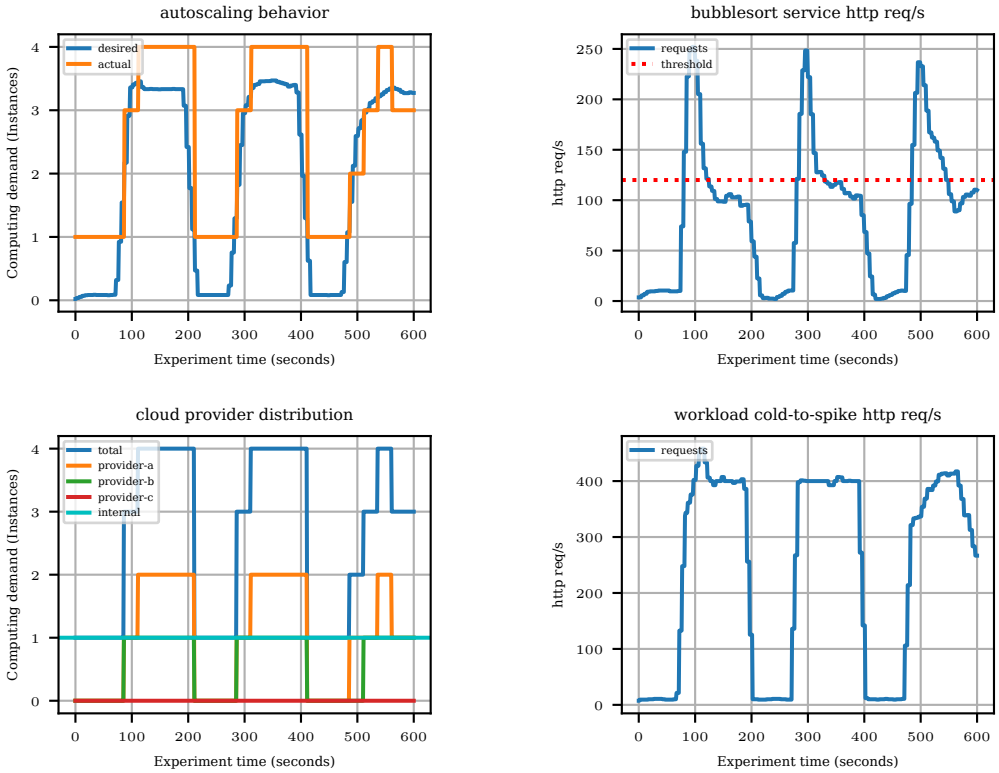


Abbildung 5.1: Benchmark HTTP Requests (Cold-To-Spike)

In Abbildung 5.1 sind die Ergebnisse des SLO *Requests* dargestellt. Zu beobachten ist, dass in der 80. Sekunde eine Verletzung des SLO stattfindet und kurz darauf ein Berechnungsintervall der Control-Plane durchgeführt wird. Wenig später werden erfolgreich zwei zusätzliche Instanzen durch die Agents provisioniert. Eine kurze Zeit später wird eine weitere Instanz gestartet (Sekunde 110), da die Lastspitze nach wie vor über dem Zielwert des Thresholds liegt. Die Wirkung der Lastverteilung durch den Proxy zeigt sich wenig später durch das rapide Sinken der Lastspitze unter den Zielwert des Thresholds mit den geforderten 120 durchschnittlichen HTTP-Anfragen pro Sekunde. Die Verteilung der Instanzen zeigt, dass die Gewichtung der Cloud-Provider bei der Provisionierung der

Instanzen berücksichtigt wurde. Bei Provider-A wurden zwei Instanzen gestartet, bei Provider-B Eine und bei Provider-C keine Instanz. Dies entspricht der geforderten Gewichtung von 50:40:10. Die abfallende Lastkurve hat Auswirkungen auf die Anzahl an Anfragen die an der Service-Instanz eingehen. Weil die Anzahl an Zugriffen weit unterhalb des Thresholds von 120 durchschnittlichen HTTP-Anfragen pro Sekunde liegt, werden die zuvor provisionierten Instanzen durch die Control-Plane als zu terminierend gekennzeichnet und wenig später durch die Agents entfernt (Sekunde 200). Das beschriebene Verhalten ist bei allen drei Lastspitzen ähnlich.

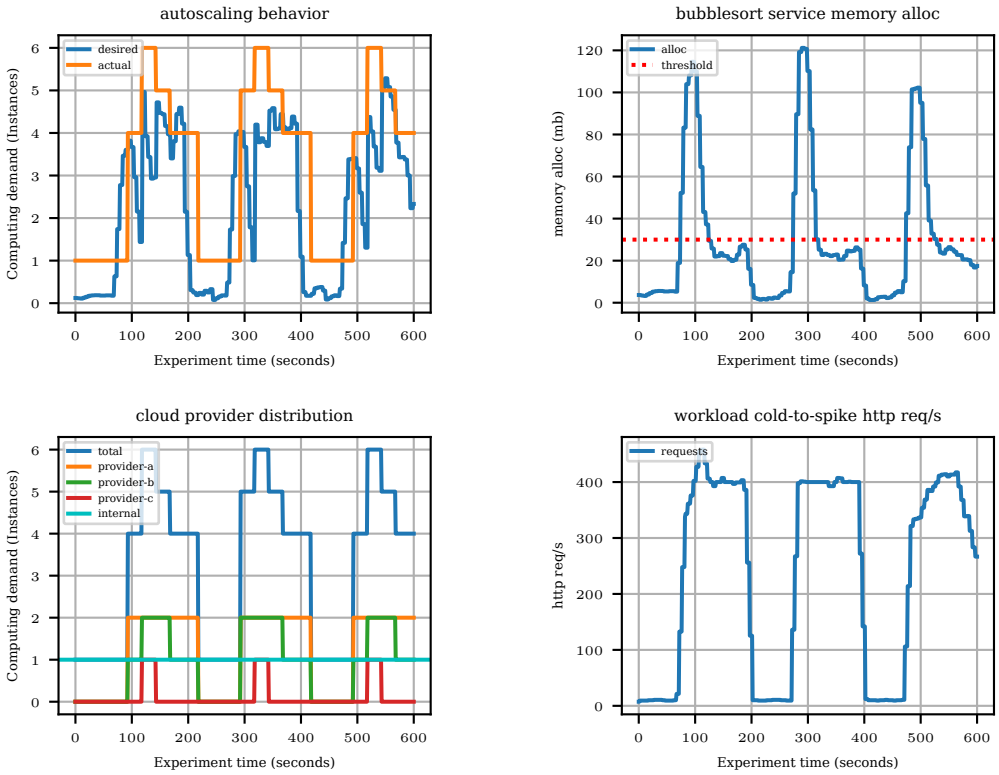


Abbildung 5.2: Benchmark Memory Usage (Cold-To-Spike)

Die in Abbildung 5.2 dargestellten Ergebnisse des SLO *Memory* sind vom Skalierungsverhalten her ähnlich zu den Ergebnissen des SLO *Requests* in Abbildung 5.1. Ab der 80. Sekunde wird der geforderte Sollwert von 30 MB als durchschnittlich allokierter Speicher pro Instanz deutlich überschritten. Die starke Überschießung ist dem Lastszenario geschuldet, da von keiner Anfrage sofort auf eine maximale Anzahl an Anfragen gesprungen wird. Ab der 90. Sekunde ist zu erkennen, dass daraufhin drei weitere Instanzen

durch die Control-Plane angefordert und wenig später durch die Agents in den Running Zustand versetzt worden sind. Durch das Starten der Instanzen treffen weniger Anfragen an der Service-Instanz ein, was sich in dem allokierten Speicher sichtbar niederschlägt (siehe 120. Sekunde). Zu erkennen ist, dass trotz sinkendem allokierten Speicher bei Sekunde 110 noch zwei weitere Instanzen angefordert und gestartet werden. Hier fand ein Berechnungsintervall durch die Control-Plane statt, während der Speicher nach wie vor trotz sinkender Speicherallokierung oberhalb des Thresholds von 30 MB lag. In Summe waren ab Sekunde 110 sechs Instanzen mit dem Status Running vorhanden. Zu erkennen ist, dass auch Provider-C eine Instanz gestartet hat. Wieder ist zu beobachten, dass die Gewichtung der Provider von 50:40:10 eingehalten wurde. Ab der 135. Sekunde pendelt sich der Bedarf unterhalb des geforderten Thresholds von 30 MB als durchschnittlich allokierter Speicher pro Instanz ein. Auch hier ist das beschriebene Verhalten bei allen drei Lastspitzen ähnlich.

5.4.2 Periodic Spikes

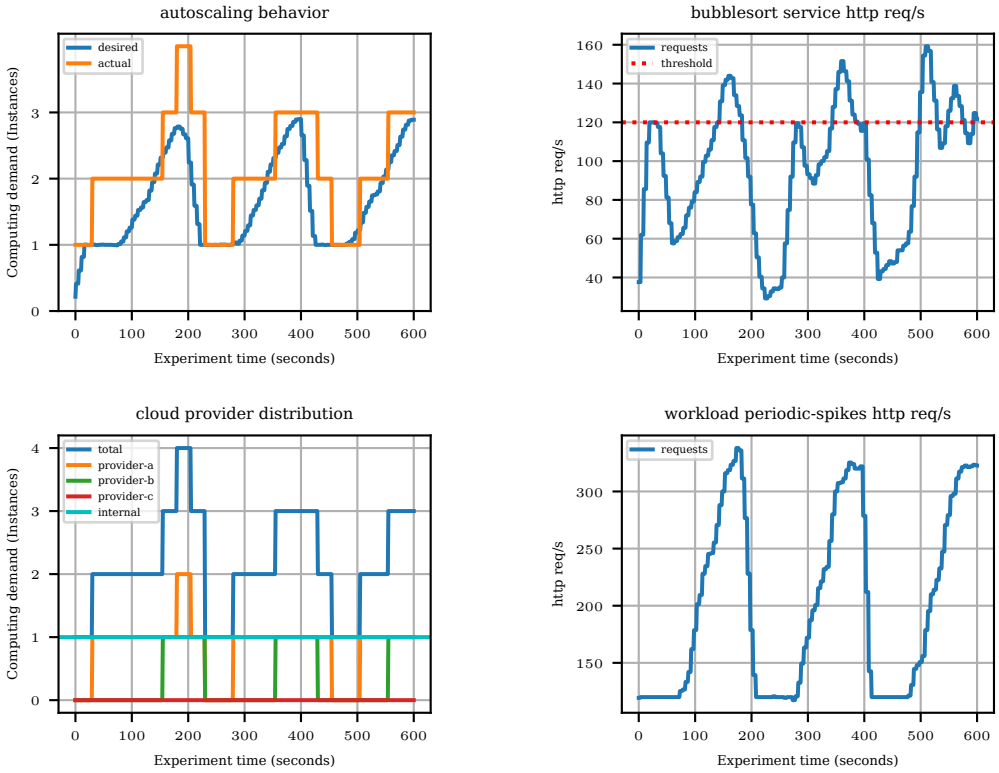


Abbildung 5.3: Benchmark HTTP Requests (Periodic-Spikes)

In Abbildung 5.3 sind die Ergebnisse des SLO *Requests* unter Durchführung des Lastszenarios *Periodic Spikes* dargestellt. Zu Beginn des Lastszenarios wird der Threshold von 120 durchschnittlichen HTTP-Anfragen pro Sekunde pro Instanz kurzzeitig überschritten. Ein Berechnungsintervall durch die Control-Plane fand statt und eine zusätzliche Instanz wurde daraufhin von dem Agent des Providers-A in den Running Zustand versetzt. Durch das Starten einer weiteren Instanz sinken die durchschnittlichen HTTP-Anfragen pro Sekunde an der überwachten Service-Instanz kurzzeitig. Ab Sekunde 65 steigen die HTTP-Anfragen bis auf ein Maximum von 340 req/s rapide an (150. Sekunde). Ab der 140. Sekunde wird der Threshold von 120 durchschnittlichen HTTP-Anfragen pro Sekunde pro Instanz überschritten, woraufhin eine weitere Instanz in den Running Zustand versetzt wird. Während des folgenden Berechnungsintervalls durch die Control-Plane liegt die Anzahl HTTP-Requests nach wie vor oberhalb des Thresholds, weshalb kurzzeitig bei Sekunde 160 eine weitere Instanz gestartet wird. Die gestartete Instanz

wird kurze Zeit später wieder terminiert. Ab Sekunde 180 fällt die Lastkurve rapide ab, woraufhin alle zusätzlich provisionierten Instanzen abgeräumt werden. Das Skalierungsverhalten ist bei allen drei Lastspitzen ähnlich. Jedoch kommt es bei Anstieg 2 und 3 zu keinen Ausreißern, da die Berechnungsintervalle durch die Control-Plane dann stattfanden, als der Threshold unterhalb den geforderten 120 HTTP-Anfragen pro Sekunde pro Instanz lag.

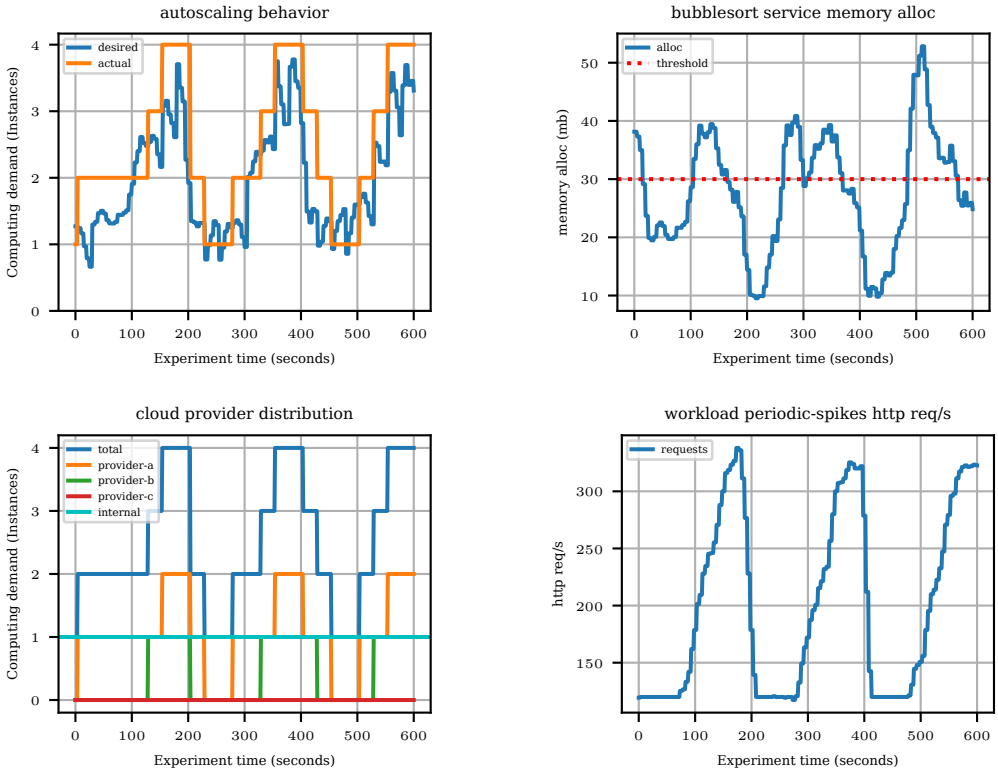


Abbildung 5.4: Benchmark Memory Usage (Periodic-Spikes)

In Abbildung 5.4 sind die Ergebnisse des SLO *Memory* visualisiert. In der 120. Sekunde wird der Threshold von 30 MB als durchschnittlich allokierter Speicher pro Instanz überschritten und sinkt erst wieder unterhalb des Zielwertes ab der 160. Sekunde. In Folge dessen wird schrittweise eine weitere Instanz durch die Agents in den Running Zustand versetzt. Ab der 200. Sekunde sinkt die Lastkurve rapide, weshalb auch der allokierte Speicher der überwachten Service-Instanz deutlich unterhalb des Thresholds liegt. Aus diesem Grund werden sämtliche, zusätzlich provisionierten Instanzen als zu terminierend gekennzeichnet und kurze Zeit später durch die Agents der Provider abgeräumt. Ab der

250. Sekunde wird der Threshold erneut überstiegen und weitere Instanzen bis zu einer Gesamtzahl von vier Instanzen in Summe provisioniert. Wieder ist zu erkennen, dass der allokierte Speicher der überwachten Service-Instanz mit dem Starten von zusätzlichen Instanzen kurzzeitig sinkt. Die stetig steigende Lastkurve zwischen Sekunde 280 bis Sekunde 400 sorgt allerdings dafür, dass der Threshold trotz eintretender Lastverteilung überschritten wird. Anhand der provisionierten Instanzen lässt sich erkennen, dass die Verteilung auf die Cloud-Provider gemäß der Gewichtung eingehalten wird (siehe Sekunde 150, 350 und 520).

5.4.3 Random Spikes

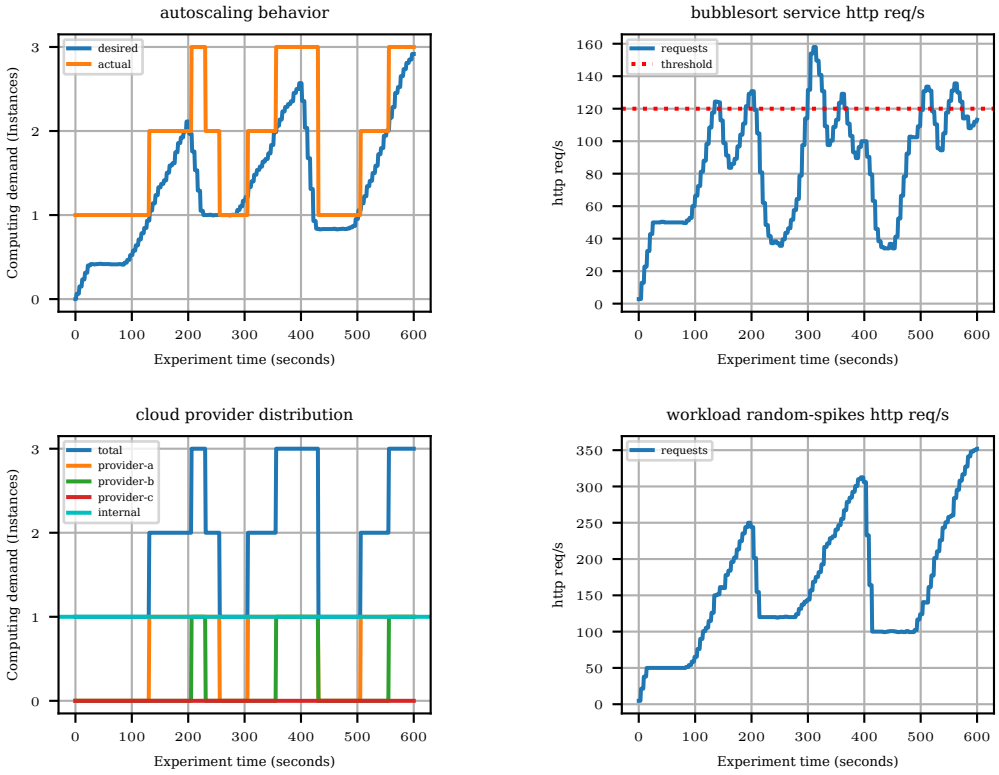


Abbildung 5.5: Benchmark HTTP Requests (Random-Spikes)

In Abbildung 5.5 sind die Ergebnisse des SLO *Requests* unter Durchführung des Lastszenarios *Random Spikes* dargestellt. Zu beobachten ist, dass ab der 100. Sekunde der Threshold von 120 durchschnittlichen HTTP-Anfragen pro Sekunde pro Instanz leicht

überschritten wird. Zu diesem Zeitpunkt fand ein Berechnungsintervall durch die Control-Plane statt, eine Instanz wurde zur Provisionierung bei Cloud-Provider A beantragt und kurze Zeit später durch den entsprechenden Agent in den Running Zustand versetzt. Eine Lastverteilung setzt ein, zu sehen an den abfallenden eingehenden HTTP-Anfragen der überwachten Service-Instanz. Die Gesamtlast steigt jedoch zwischen der 100. und 200. Sekunde rapide an, weshalb der Threshold bei Sekunde 190 wieder überschritten wird. Eine weitere Instanz wird provisioniert, diesmal bei Cloud-Provider B. Wenig später sinkt die Gesamtlast, weshalb die beiden Instanzen bei Provider-A und Provider-B von der Control-Plane als zu terminierend gekennzeichnet und wenig später durch die entsprechenden Agents beendet werden. Allgemein ist in den Ergebnissen sehr schön beobachtbar, wie der Autoscaler bei Überschreiten des Thresholds versucht, das SLO durchzusetzen.

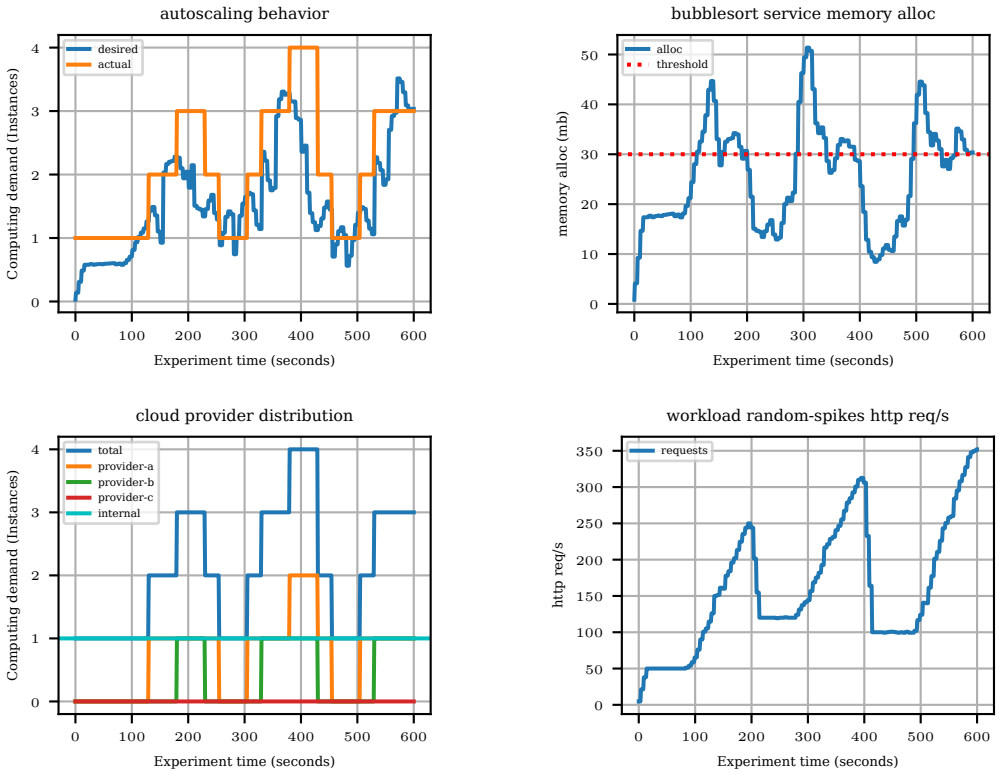


Abbildung 5.6: Benchmark Memory Usage (Random-Spikes)

Die Ergebnisse des SLO *Memory* in Abbildung 5.6 sind ähnlich den zuvor erläuterten Ergebnissen des SLO *Requests* aus Abbildung 5.5. Der Threshold von 30 MB als durch-

schnittlich allokierter Speicher pro Instanz wird bei jeder Lastspitze zunächst überschritten (siehe Sekunde 110, 300 und 500), anschließend jedoch durch die Provisionierung zusätzlicher Instanzen unter den Zielwert gedrückt.

5.4.4 Rapid Growth

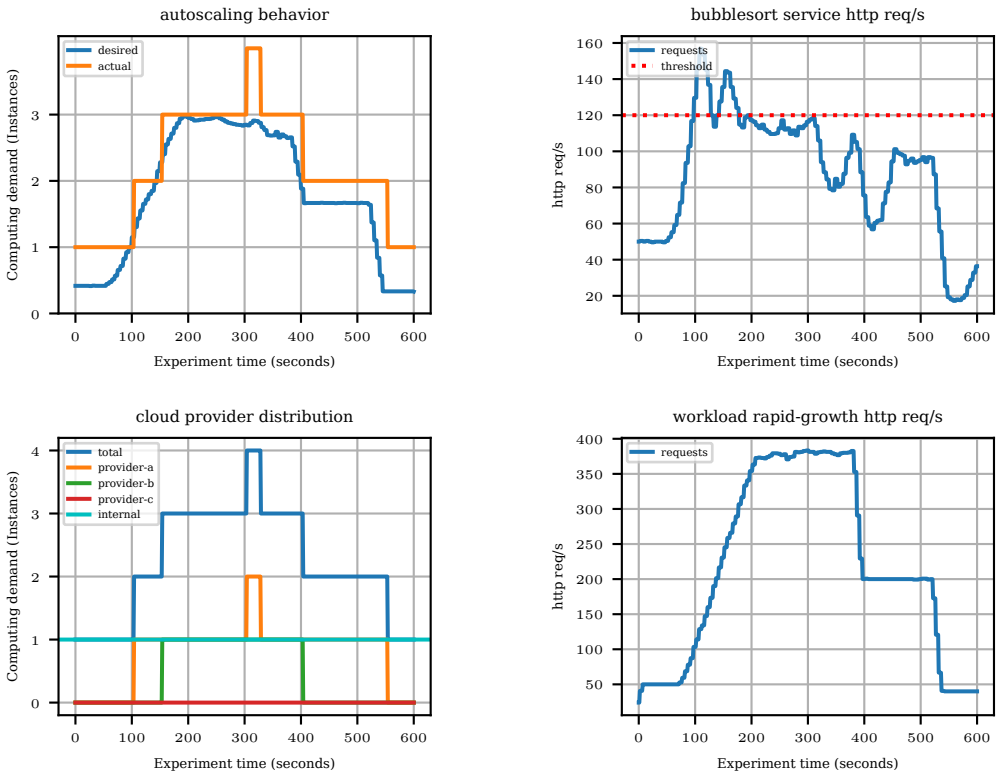


Abbildung 5.7: Benchmark HTTP Requests (Rapid-Growth)

Die Ergebnisse in Abbildung 5.7 zeigen das Skalierungsverhalten des Autoscalers bei der Durchführung des Lastszenarios *Rapid Growth*. Zu beobachten ist, dass die Lastkurve zwischen der 100. und 200. Sekunde zunächst stetig wächst. Stufenweise werden Instanzen durch die Agents provisioniert (siehe Sekunde 100 und 130), wodurch die Anzahl der eintreffenden HTTP-Anfragen an der lokalen, überwachten Service-Instanz unterhalb des Threshold von 120 durchschnittlichen HTTP-Anfragen pro Sekunde pro Instanz sinkt. In dem Zeitraum zwischen der 200. und 300. Sekunde ist zu erkennen, dass trotz anhaltendem Lastplateau das SLO erfüllt wird. In der 300. Sekunde kommt es zu einem

Ausreißer. Zu diesem Zeitpunkt wird der Threshold minimal überschritten, gleichzeitig fiel auf diesen Zeitpunkt ein Berechnungsintervall der Control-Plane. In Folge dessen wird eine weitere Instanz provisioniert, die allerdings kurze Zeit später wieder terminiert wird, da die eintreffenden HTTP-Anfragen an der lokalen Service-Instanz wesentlich unterhalb des Thresholds liegen. Auch ist wieder zu beobachten, dass mit dem Terminieren von Instanzen die Anzahl von eintreffenden HTTP-Anfragen an der lokalen Service-Instanz zunimmt, da die Menge an Instanzen die Anfragen entgegennimmt, verringert wird (siehe 340. und 420. Sekunde).

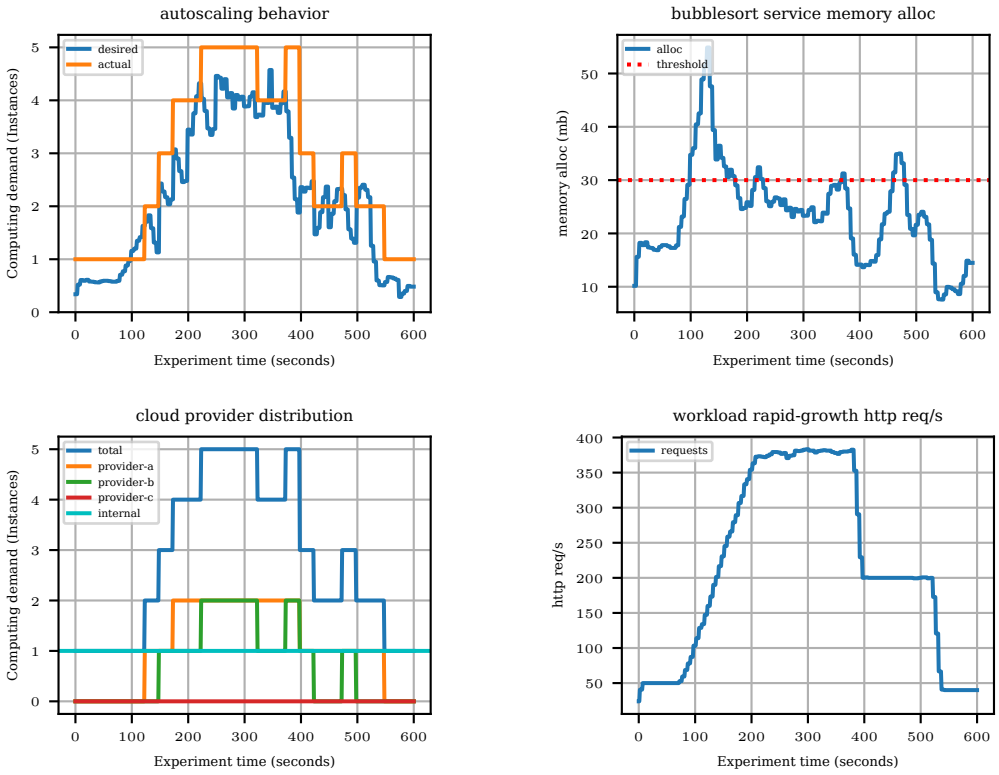


Abbildung 5.8: Benchmark Memory Usage (Rapid-Growth)

In Abbildung 5.8 sind die Ergebnisse des SLO *Memory* dargestellt. Auch hier werden stufenweise Instanzen gestartet, um den allokierten Speicher der lokalen, überwachten Service-Instanz unterhalb des geforderten Thresholds von 30 MB zu bringen. Zu erkennen ist, dass der tatsächlich berechnete Bedarf (siehe *autoscaling behavior*) nach dem Starten einer neuen Instanz kurzzeitig sinkt, als Folge von weniger eintreffenden Sortierungsanfragen und somit zu weniger allokierten Speicher führt. Zwischen der 100. und

200. Sekunde wächst die Anzahl an eingehenden Anfragen stetig. Trotz der Provisionierung einzelner Instanzen, wächst der Ressourcenbedarf weiter. Bei Sekunde 380 gibt es einen Ausreißer, da der allokierte Speicher kurzzeitig oberhalb des geforderten Thresholds liegt und ein Berechnungsintervall durch die Control-Plane exakt auf diesen Zeitpunkt gefallen ist. Hierdurch wurde eine weitere Instanz bei Cloud-Provider B angefordert, die kurze Zeit später durch den entsprechenden Agent gestartet wurde. In folgendem Berechnungsintervall wurde die gestartete Instanz jedoch umgehend wieder als zu terminierend gekennzeichnet und von Agent beendet (siehe 400. Sekunde). Das Beenden von zwei Instanzen gleichzeitig in der 400. Sekunde des Lastszenarios führt dazu, dass kurzzeitig wieder mehr Anfragen an der lokalen Service-Instanz eintreffen und somit mehr Speicher für Bearbeitung der Sortierungsanfragen allokiert wird. Zu sehen ist, dass ab der 440. Sekunde der Threshold kurzzeitig wieder überschritten wird und es kurzzeitig erneut zu einem Ausreißer kommt.

5.5 Diskussion

Die Evaluierung hat gezeigt, dass die implementierte Autoscaler-Architektur in der Lage ist, eine Anwendung als Komposition aus mehreren Services zu mehreren Cloud-Providern zu skalieren. Die in 5.3.1 vorgestellten SLOs wurden hierzu vier unterschiedlichen Lastszenarien ausgesetzt. Während der Durchführung wurde das Verhalten anhand exportierter Metriken des Autoscalers sichtbar gemacht.

Hierbei ist anzumerken, dass die beiden verwendeten SLOs Requests und Memory linear abhängig von der Veränderung der gemessenen Metrik sind. Wenn sich die eingehende Anzahl an HTTP-Anfragen an der überwachten Service-Instanz verdoppelt, dann werden zur Einhaltung des SLO doppelt so viele Instanzen benötigt. Ist diese lineare Abhängigkeit nicht gegeben, wird die in dieser Arbeit implementierte Berechnung des Autoscalers nicht funktionieren und keine zufriedenstellenden Ergebnisse liefern. Wie bereits dargestellt kann aber ohne strukturelle Änderung am Autoscaler eine andere Berechnungslogik abgebildet werden.

In jedem Lastszenario zeigen sich die Nachteile eines reaktiven Skalierungsalgorithmus. Die Skalierung von Ressourcen erfolgt nur als Reaktion auf Veränderungen der überwachten Metriken in Echtzeit. Zwar werden die Grenzwerte der definierten SLOs langfristig

eingehalten, bei steigendem Lastverhalten werden sie jedoch zunächst überschritten. Diesem Skalierungsverhalten lässt sich durch die Implementierung anderer Berechnungsansätze entgegenwirken. So kann eine proaktive oder hybride Berechnung für den Ressourcenbedarf implementiert werden, die Metriken über den aktuellen Lastverlauf mit historischen Verlaufsdaten kombiniert, um so zukünftige Schwankungen im Lastverlauf vorauszusagen. Einige Ansätze dafür wurden in Kapitel 3 vorgestellt.

Ein weiterer wichtiger Aspekt, der das Skalierungsverhalten des Autoscalers beeinflusst, ist die Wahl der Parameter für die Berechnungsintervalle. Wird der Abstand zwischen den einzelnen Berechnungsintervallen zu hoch gesetzt, wird die Steuerung für die Skalierung von Ressourcen träge und führt bei der Verwendung einer reaktiven Berechnung zu einer längeren und stärkeren Überschreitung der SLO-Grenzwerte. Andererseits kann das Skalierungsintervall nicht beliebig verkleinert werden, da sich sonst die Wirkung der Skalierungsoperation nicht in den überwachten Metriken widerspiegelt und somit eine Berechnung auf Grundlage einer falschen Annahme zur Ressourcenauslastung gemacht wird.

In den Ergebnissen ist außerdem zu beobachten, dass die Steuerung des Autoscalers sehr empfindlich auf minimale Überschreitungen der SLO-Grenzwerte reagiert. Dem kann durch das Einführen des Threshold wie in Abschnitt 4.3.1 erläutert entgegengewirkt werden. Jedoch führt dies dazu, dass die Steuerung erst bei einer stärkeren Verletzung der SLO-Grenzwerte eingreift.

6 Zusammenfassung und Ausblick

Diese Arbeit befasste sich zunächst mit den Grundlagen des Cloud Computings und der Erzeugung von Elastizität in Softwaresystemen. Anschließend wurde eine konkrete Problemstellung und ein Anforderungskatalog formuliert. Hierfür wurden Arbeiten analysiert, die sich mit der automatischen Skalierung von Ressourcen zu externen Cloud-Providern beschäftigen.

Auf Grundlage der Anforderungen wurde eine modulare Autoscaler-Architektur entworfen, die anhand von benutzerspezifischen Skalierungsregeln Ressourcen zu mehreren Cloud-Providern skalieren kann und die Ausführungsphase des Skalierungsprozesses verteilt durchführt. Die Phasen des Skalierungsprozesses wurden durch erweiterbare Komponenten innerhalb der Architektur abgebildet. Es wurde ein Datenmodell zur providerunabhängigen Spezifikation von zu skalierenden Services als Teil einer Gesamtanwendung entworfen. Weiterhin wurde ein threshold-basierter, reaktiver Skalierungsalgorithmus vorgestellt, welcher die Gewichtung von unterschiedlichen Cloud-Providern bei der Berechnung des Ressourcenbedarfs berücksichtigt.

Die entworfene Autoscaler-Architektur wurde prototypisch implementiert und das Skalierungsverhalten unter der Durchführung von vier unterschiedlichen Lastszenarien evaluiert.

6.1 Fazit

Der implementierte Prototyp der verteilten Autoscaler-Architektur hat gezeigt, dass sich der konzipierte Skalierungsprozess umsetzen lässt und eine Skalierung von Ressourcen zu mehreren Cloud-Providern möglich ist.

Die Aufteilung des Autoscalers in Komponenten stellt sicher, dass sich die Autoscaler-Architektur flexibel erweitern lässt. Über die in Abschnitt 4.2 vorgestellten Interface-Definition können weitere Skalierungsalgorithmen, Persistenzlösungen, Monitoring-Systeme

und Plattform-Anbindungen implementiert werden, ohne dass die Ausführungsreihenfolge der Phasen des Skalierungsprozesses verändert wird. Durch die verteilte Ausführung der Provisionierung von Instanzen, wird die Wartbarkeit der Autoscaler-Architektur erhöht, da plattformspezifische Implementierungen nicht Bestandteil der Steuerungslogik sind. Dies geht einher mit dem Nachteil, dass die Berechnung des Ressourcenbedarfs komplexer wird, da durch die Nebenläufigkeit zusätzliche Zustände der Instanzen berücksichtigt werden müssen.

6.2 Ausblick

In weiterführenden Arbeiten kann auf den Ergebnissen dieser Thesis aufgebaut werden. Wie in Abschnitt 4.3.1 erläutert, wurde ein threshold-basierter, reaktiver Skalierungsalgorithmus implementiert. Durch die Implementierung der Interface-Spezifikation aus Abschnitt 4.2.1 lassen sich auch andere Skalierungsalgorithmen mit der Autoscaler-Architektur nutzen. Hier wäre es interessant, sowohl proaktive als auch hybride Skalierungsalgorithmen aus einigen der vorgestellten Arbeiten in einem verteilten Kontext zu evaluieren.

In der Evaluierung wurden die Schnittstellen der externen Cloud-Plattformen gemockt, da die Implementierung der Provider-Schnittstellen gemäss des in 4.2.2 beschriebenen *InstanceLifecycleHandler* Interfaces für die Beurteilung des Skalierungsverhaltens nicht notwendig ist. Trotzdem wäre es interessant, zukünftig Agents mit Provider-Implementierungen umzusetzen und reale Instanzen auf externen Cloud-Plattformen von Cloud-Providern über Container-As-A-Service-Angebote wie AWS Fargate[1] oder Google Cloud Run[2] zu provisionieren.

Die implementierte Autoscaler-Architektur ist in der Lage, SLOs unter Nutzung anwendungsspezifischer Metriken einzuhalten. In der Evaluierung werden Metriken über ein externes Monitoring Tool bezogen, das lediglich Metriken von Instanzen auf der eigenen, lokalen Infrastruktur sammelt. Zukünftig wäre es sinnvoll, auch Metriken von extern provisionierten Instanzen in die Berechnung des Ressourcenbedarfs aufzunehmen. Hierfür muss eine Integration zwischen externen Monitoring-Systemen und dem lokalen Monitoring-System umgesetzt werden. Dies ist nicht direkt Bestandteil der vorgestellten Autoscaler-Architektur, da diese sich auf vorhandene Lösungen zur Überwachung

[1] https://aws.amazon.com/de/fargate
[2] https://cloud.google.com/run

und Aggregation von Metriken für die Skalierung stützt. Die Etablierung eines Standards für die Repräsentation und die Übertragung von Cloud-nativen Metriken über Cloud-Plattformen hinweg ist Gegenstand von Initiativen wie beispielsweise OpenMetrics[3], weshalb hier in Zukunft Fortschritte zu erwarten sind.

Die vorgestellte Autoscaler-Architektur lässt eine statische Gewichtung von externen Cloud-Plattformen zu, die bei der Berechnung des Skalierungsbedarfes berücksichtigt wird. Mit der Einbindung von externen Metriken wäre es denkbar, diese Gewichtungen dynamisch zu modifizieren, abhängig von Performanzindikatoren wie beispielsweise aktueller Verfügbarkeit oder Latenzen der jeweiligen Cloud-Plattform.

In der Arbeit wird ein deklaratives Beschreibungsformat für zu skalierende Services vorgestellt (siehe Abschnitt 4.4.2). Obwohl zu skalierende Services möglichst lose gekoppelt sein sollen und im optimalen Falle eine Fachlichkeit kapseln, ist dies in der Praxis nicht immer der Fall. So kann es beispielsweise durch einen falschen Komponentenschnitt dazu kommen, dass zwei Services doch stark voneinander abhängen. Ein Bestandteil weiterer Arbeiten könnte sein, das vorgestellte Beschreibungsformat zu erweitern, sodass sich Abhängigkeiten zwischen Services als Bestandteil der Konfiguration des Autoscalers definieren lassen. Diese Abhängigkeit zwischen den zu skalierenden Services kann anschließend bei der Berechnung des Ressourcenbedarfs berücksichtigt werden.

[3] https://openmetrics.io/

Literaturverzeichnis

[1] Al-Dhuraibi, Y. ; Paraiso, F. ; Djarallah, N. ; Merle, P.: Elasticity in Cloud Computing: State of the Art and Research Challenges. 11, Nr. 2, S. 430–447. – Conference Name: IEEE Transactions on Services Computing. – ISSN 1939-1374

[2] Alipour, Hanieh ; Liu, Yan ; Hamou-Lhadj, Abdelwahab: Analyzing auto-scaling issues in cloud environments. In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, IBM Corp. (CASCON '14), S. 75–89

[3] Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy H. ; Konwinski, Andrew ; Lee, Gunho ; Patterson, David A. ; Rabkin, Ariel ; Zaharia, Matei: *Above the Clouds: A Berkeley View of Cloud Computing*

[4] Biswas, A. ; Majumdar, S. ; Nandy, B. ; El-Haraki, A.: An Auto-Scaling Framework for Controlling Enterprise Resources on Clouds. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, S. 971–980

[5] Calcavecchia, Nicolò M. ; Caprarescu, Bogdan A. ; Di Nitto, Elisabetta ; Dubois, Daniel J. ; Petcu, Dana: DEPAS: a decentralized probabilistic algorithm for auto-scaling. 94, Nr. 8, S. 701–730. – URL https://doi.org/10.1007/s00607-012-0198-8. – Zugriffsdatum: 2021-04-24. – ISSN 1436-5057

[6] Chandra, Gokul: *Cloud Bursting with Virtual Kubelet and KIP (Kloud Instance Provider)*. – URL https://itnext.io/cloud-bursting-with-virtual-kubelet-and-kip-kloud-instance-provider-4b86a479ce38. – Zugriffsdatum: 2020-11-26

[7] Chieu, Trieu C. ; Chan, Hoi: Dynamic Resource Allocation via Distributed Decisions in Cloud Environment. In: *2011 IEEE 8th International Conference on e-Business Engineering*, S. 125–130

[8] Ellingwood, Justin: *Monitoring for Distributed and Microservices Deployments*. – URL https://www.digitalocean.com/community/tutorials/monitoring-for-distributed-and-microservices-deployments. – Zugriffsdatum: 2021-05-01

[9] Foster, Ian ; Kesselman, Carl: Globus: a Metacomputing Infrastructure Toolkit. 11, Nr. 2, S. 115–128. – URL https://doi.org/10.1177/109434209701100205. – Zugriffsdatum: 2021-04-28. – ISSN 1094-3420

[10] Herbst, Nikolas R. ; Kounev, Samuel ; Reussner, Ralf: Elasticity in Cloud Computing: What It Is, and What It Is Not. In: *10th International Conference on Autonomic Computing (ICAC 13)*, USENIX Association, S. 23–27. – URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst. – ISBN 978-1-931971-02-7

[11] Inc., Google: *Kubernetes V1 Released - Cloud Native Computing Foundation to Drive Container Innovation*. – URL https://cloudplatform.googleblog.com/2015/07/Kubernetes-V1-Released.html. – Zugriffsdatum: 2021-04-28

[12] Kane, Sean P. ; Matthias, Karl: *Docker: Up & Running: Shipping Reliable Containers in Production*. 2nd ed edition. O'Reilly UK Ltd.. – ISBN 978-1-4920-3673-9

[13] Keahey, Katarzyna ; Tsugawa, Mauricio ; Matsunaga, Andrea ; Fortes, Jose: Sky Computing. 13, Nr. 5, S. 43–51. – ISSN 1941-0131

[14] Keahey, Kate ; Armstrong, Patrick ; Bresnahan, John ; LaBissoniere, David ; Riteau, Pierre: Infrastructure outsourcing in multi-cloud environment. In: *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, Association for Computing Machinery (FederatedClouds '12), S. 33–38. – URL https://doi.org/10.1145/2378975.2378984. – Zugriffsdatum: 2021-04-24. – ISBN 978-1-4503-1754-2

[15] Kubernetes: *Provider List - The Cluster API Book*. – URL https://cluster-api.sigs.k8s.io/reference/providers.html. – Zugriffsdatum: 2021-04-28

[16] Kubernetes: *Vertical Pod Autoscaler for Kubernetes*. – URL https://github.com/kubernetes/autoscaler. – Zugriffsdatum: 2021-04-28

[17] Kubernetes: *Was ist Kubernetes?*. – URL https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/. – Zugriffsdatum: 2021-04-28

[18] Kukade, Priyanka P. ; Kale, Geetanjali: Auto-Scaling of Micro-Services Using Containerization. 4, Nr. 9, S. 1960 – 1963. – URL https://www.ijsr.net/search_index_results_paperid.php?id=SUB158576. – Zugriffsdatum: 2021-04-28

[19] Lehrig, Sebastian ; Eikerling, Hendrik ; Becker, Steffen: Scalability, Elasticity, and Efficiency in Cloud Computing: a Systematic Literature Review of Definitions and Metrics. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, Association for Computing Machinery (QoSA '15), S. 83–92. – URL https://doi.org/10.1145/2737182.2737185. – Zugriffsdatum: 2021-04-22. – ISBN 978-1-4503-3470-9

[20] Marshall, Paul ; Keahey, Kate ; Freeman, Tim: Elastic Site: Using Clouds to Elastically Extend Site Resources. In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, S. 43–52

[21] Maximilien, E. M. ; Ranabahu, Ajith ; Engehausen, Roy ; Anderson, Laura C.: Toward cloud-agnostic middlewares. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Association for Computing Machinery (OOPSLA '09), S. 619–626. – URL https://doi.org/10.1145/1639950.1639957. – Zugriffsdatum: 2021-04-25. – ISBN 978-1-60558-768-4

[22] Mell, Peter ; Grance, Tim: *The NIST Definition of Cloud Computing*. – URL https://csrc.nist.gov/publications/detail/sp/800-145/final. – Zugriffsdatum: 2021-04-22

[23] Mennig, Simon: *Cloud Bursting – platzt die Private Cloud, ist die Public Cloud zur Stelle*. – URL https://www.heise.de/hintergrund/Cloud-Bursting-platzt-die-Private-Cloud-ist-die-Public-Cloud-zur-Stelle-4968960.html. – Zugriffsdatum: 2020-11-26

[24] Metz, Cade: The Biggest Thing in Cloud Computing Has a New Competitor. . – URL https://www.wired.com/2014/12/google-others-eye-new-alternative-docker-cloud-computings-next-big-thing/. – Zugriffsdatum: 2021-04-29. – ISSN 1059-1028

[25] Shen, Zhiming ; Subbiah, Sethuraman ; Gu, Xiaohui ; Wilkes, John: CloudScale: elastic resource scaling for multi-tenant cloud systems. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Association for Computing Machinery (SOCC '11), S. 1–14. – URL https://doi.org/10.1145/2038916.2038921. – Zugriffsdatum: 2021-04-26. – ISBN 978-1-4503-0976-9

[26] Smith, M. ; Schmidt, M. ; Fallenbeck, N. ; Dörnemann, T. ; Schridde, C. ; Freisleben, B.: Secure On-Demand Grid Computing. 25, Nr. 3, S. 315–325. – URL https://doi.org/10.1016/j.future.2008.03.002. – ISSN 0167-739X

[27] Spazzoli, Raffaele: *How Full Is My Cluster, Part 6: Proactive Node Autoscaling*. – URL https://www.openshift.com/blog/how-full-is-my-cluster-part-6-proactive-node-autoscaling. – Zugriffsdatum: 2021-04-28

[28] Vogels, Werner: *A Head in the Cloud: The Power of Infrastructure as a Service*. – URL https://www.youtube.com/watch?v=9AS8zzUaO3Y. – Zugriffsdatum: 2021-04-22

[29] Ye, T. ; Guangtao, X. ; Shiyou, Q. ; Minglu, L.: An Auto-Scaling Framework for Containerized Elastic Applications. In: *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, S. 422–430

[30] Zhang, Qi ; Cheng, Lu ; Boutaba, Raouf: Cloud computing: state-of-the-art and research challenges. 1, Nr. 1, S. 7–18. – URL https://doi.org/10.1007/s13174-010-0007-6. – Zugriffsdatum: 2021-04-22. – ISSN 1869-0238

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

Ort	Datum	Unterschrift im Original