

Martin Sträßer

Benchmarking and Modeling of Container Orchestration Frameworks as a Basis of Modern Cloud Applications



Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2025

Gutachter:

Prof. Dr. Samuel Kounev, Julius-Maximilians-Universität Würzburg (GER)

Prof. Dr. Wilhelm Hasselbring, Christian-Albrechts-Universität zu Kiel (GER)

Datum der mündlichen Prüfung: 24.07.2025

Abstract

Cloud computing has transformed software applications and internet services by allowing customers to access scalable resources on demand, enabling them to deploy and run many modern services like video streaming and generative artificial intelligence at scale. Public cloud providers, like Amazon Web Services, Google Cloud, and Microsoft Azure, offer various cloud solutions serving different user needs. With public cloud spending growing from 175 billion US dollars in 2017 to a projected 675.4 billion US dollars in 2024 [Gar24], the technology's widespread adoption is underlined.

A key enabler for cloud computing is virtualization, which is the foundation for efficient resource sharing and security. Containers offer a lightweight approach to virtualization, enabling fast startups and good scalability. Containers are central to modern cloud applications and especially popular in microservice architectures and serverless functions, where applications are broken into smaller, independently deployable components. Trends like DevOps and Continuous Integration and Deployment (CI/CD) have fueled the adoption of container orchestration frameworks, which automate container management tasks like deployment, scaling, and health monitoring. By now, Kubernetes is the leading orchestration framework, regarded as a "core technology" [Clo24] of the cloud, with widespread usage across major public clouds and various industries.

Performance is a critical success factor for cloud applications, encompassing user-focused aspects like availability and provider-focused aspects like cost efficiency. Container orchestration frameworks determine the performance of modern cloud applications through features such as autoscaling and network traffic management. Cloud providers face increasing pressure to balance resource efficiency with application performance, especially in serverless computing, where users only pay for resources they actually use. This thesis addresses the role of container orchestration frameworks in the performance of modern cloud applications. We increase the understanding of the state of the art and propose approaches for improvement through four targeted contributions.

Contribution I - A Systematic Approach for Benchmarking of Container Orchestration Frameworks

Regardless of the environment or service context (serverless or traditional), managing the performance of container applications and clusters is non-trivial. The increasing number of features, algorithms, and control processes in modern container orchestration frameworks complicates their configuration. In addition, these elements all act in parallel and impact each other. A holistic approach to performance analysis and the joint assessment of control algorithms are necessary to pursue optimal performance. The fine-grained configurability of modern

frameworks expands the configuration space, posing significant challenges for practitioners.

Our first contribution addresses the outlined challenges by introducing a systematic benchmarking approach for container orchestration frameworks. We identify eight core requirements of container orchestration frameworks and link metrics to evaluate their performance. A benchmark architecture is then designed to reliably measure these metrics without imposing limiting constraints on the systems under test. The result is COFFEE, an open-source benchmarking framework that enables detailed test campaigns targeting various components of container orchestration frameworks. Four case studies demonstrate the usage and potential of COFFEE. These include comparisons of Kubernetes and HashiCorp Nomad running on public cloud and self-hosted machines. We cover scenarios focusing on container provisioning and networking, failure recovery, rolling updates, and persistent storage performance. Findings reveal, for example, differences in provisioning speeds and trade-offs in update strategies. COFFEE is the first comprehensive framework to evaluate performance-critical aspects of container orchestration frameworks across diverse technology stacks.

Contribution II - An Empirical Study on Factors Impacting Container Start Times

Container start times are a critical factor influencing the performance of containerized applications, affecting scaling, failure recovery, update processes, and more. While container start times have been compared to those of other virtualization technologies like virtual machines and unikernels, little is known about the factors influencing them. Addressing this gap is essential for answering questions like why one container starts faster than another. Cloud providers are already optimizing their infrastructure with container-specific operating systems, highlighting the importance of understanding the impact of host environments and hardware on start times. Identifying these influencing factors enables the development of guidelines for creating fast-starting containers and recommendations for optimal deployment strategies. However, due to a lack of data quantifying these factors, the potential to further reduce container start times is currently largely unexplored.

Our second contribution addresses the challenge of understanding factors influencing container start times through an extensive empirical study. Using a new dataset of approximately 200,000 open-source Docker Hub images, statistical analyses reveal key characteristics of container applications, such as image size and the number of file system layers. Extensive start time measurements on over 1000 images in both self-hosted and Google Cloud environments demonstrate that start times range from milliseconds to tens of seconds in the same environment, with no single dominant factor directly determining the performance. Instead, multiple factors, including image size and file system layers, jointly impact start times. Further experiments on the Google Cloud Platform reveal that host configurations, particularly disk types, significantly affect start times. Our study emphasizes that both image configuration and platform pa-

rameters must be evaluated together for an accurate assessment of container start times. This research represents the most comprehensive empirical study of container start times to date, providing important guidance for optimization potential.

Contribution III - A Continuous, Decentralized Approach to Autoscaling

Autoscaling is one of the most critical tasks of container orchestration frameworks, directly impacting both service quality and operating costs. Modern cloud applications, such as microservices, are typically deployed in multiple containers, enabling fine-grained scaling by managing application components independently. However, implementing autoscaling for the diverse and numerous services is a significant challenge. As cloud usage grows, workloads have become increasingly diverse, with varying request patterns, payloads, and response times. Traditional autoscalers struggle with these dynamic workloads due to fixed scaling periods and cooldown times, limiting their ability to adapt quickly. Furthermore, the growing number of services and frequent updates push autoscalers with preset models to their limits, highlighting the need for more adaptive and efficient autoscaling solutions.

Our third contribution addresses the challenge of autoscaling in dynamic cloud environments by introducing a novel decentralized and continuous autoscaling approach. Unlike most state-of-the-art autoscalers, our method allows individual service instances to trigger scaling actions. The actions are distributed over time so that a quasi-continuous scaling process evolves. We evaluate our approach on three abstraction levels. A discrete-time queueing model allows us to analyze decision speed and convergence. A discrete-event simulation assesses and evaluates the scaling behavior for different configurations and system scales. Experiments in cloud environments demonstrate superior scaling performance compared to state-of-the-art autoscalers, particularly under highly dynamic workloads. Our novel approach offers a robust and flexible autoscaling solution for modern cloud applications and questions some established paradigms for building autoscalers.

Contribution IV - Enriching Microservice Simulation Through Authentic Container Orchestration

Configuring container orchestration frameworks, such as for autoscaling and scheduling, is a complex and resource-intensive process. Performance simulations offer a cost-effective way to evaluate configuration changes without deploying costly test environments. Current performance simulations for modern cloud applications, like microservices, often do not support or oversimplify container orchestration processes. This gap makes evaluating specific configurations, like Kubernetes scheduling policies, challenging without extensive manual work. As a result, the simulations have limited applicability and expressiveness, preventing practitioners from efficiently optimizing orchestrator configurations for real-world applications.

Our fourth contribution enables the investigation of container orchestration configurations using simulations. We present an approach to integrate Kubernetes

components into a microservice performance simulation. It is based on analogies between discrete-event simulations and event-driven systems. An adapter enables seamless communication between the simulation and external Kubernetes components without needing orchestrator-specific logic in the simulation. This integration allows Kubernetes components and configurations to be tested in a simulated environment, mimicking real-world cloud conditions. Two case studies with the scheduler and cluster autoscaler of Kubernetes demonstrate the approach's ability to simulate cloud behavior accurately, expanding the simulation's use cases and improving its accuracy with respect to performance metrics of deployed applications. This contribution is the first to enable the operation of multiple Kubernetes components within a microservice simulation. It facilitates the evaluation of container orchestrator configurations without the need for expensive test environments.

Collectively, all contributions offer ways towards efficient and reliable operation of container clusters and modern cloud applications and significantly advance the research field. They also provide a basis and motivation for future work in area of performance benchmarking and modeling of container orchestration frameworks.

Zusammenfassung

Cloud Computing hat Software-Anwendungen und Internet-Dienste revolutioniert. Es erlaubt Nutzern bei Bedarf auf beliebig skalierbare Ressourcen zuzugreifen, sodass ihnen ermöglicht wird, moderne Dienstleistungen wie Video-Streaming oder generative künstliche Intelligenz im großen Stil anzubieten. Anbieter öffentlicher Cloud-Plattformen, wie Amazon Web Services, Google Cloud und Microsoft Azure, bieten zahlreiche Cloud-Lösungen für ganz unterschiedliche Nutzerinteressen an. Die Tatsache, dass die Ausgaben für öffentliche Cloud-Plattformen von 175 Mrd. US-Dollar in 2017 auf prognostizierte 675,4 Mrd. US-Dollar in 2024 gewachsen sind [Gar24], unterstreicht die Wichtigkeit dieser Technologien.

Eine der Grundlagen für Cloud Computing ist Virtualisierung, da diese ermöglicht, Ressourcen effizient und sicher zwischen verschiedenen Nutzern zu teilen. Container sind eine leichtgewichtige Art der Virtualisierung, die schnelle Startzeiten und gute Skalierbarkeit ermöglicht. Container sind essenzielle Bestandteile moderner Cloud-Anwendungen und vor allem bei der Bereitstellung von Microservice-Architekturen und Serverless Functions beliebt, da dort Anwendungen in viele kleine, unabhängig bereitstellbare Komponenten aufgeteilt werden. Trends wie DevOps oder Continuous Integration and Deployment (CI/CD) haben die Verwendung von Orchestrierungsplattformen für Container vorangetrieben. Diese automatisieren die Verwaltung von Containern durch Aktivitäten wie automatische Bereitstellung, Skalierung und Überwachung. Zur Zeit ist Kubernetes die meist verwendete Orchestrierungsplattform und wird auch als "Kerntechnologie" [Clo24] der Cloud bezeichnet.

Performance ist ein kritischer Erfolgsfaktor für Cloud-Anwendungen und umfasst nutzerorientierte Aspekte (z.B. Verfügbarkeit) sowie anbieterorientierte Aspekte (z.B. Kosteneffizienz). Container-Orchestrierungsplattformen bestimmen maßgeblich die Performance von Cloud-Anwendungen durch Aktivitäten wie Autoscaling oder der Steuerung des Netzwerkverkehrs. Anbieter von Cloud-Diensten haben ein steigendes Interesse sowohl Ressourceneffizienz als auch Anwendungsperformance zu optimieren. Dies ist besonders im Kontext von Serverless Computing wichtig, da dort Nutzer nur für Ressourcen zahlen, die tatsächlich genutzt wurden. Dies ist ein gravierender Unterschied zu früheren Geschäftsmodellen bei denen Nutzer Ressourcen reservieren mussten und selbst für die Optimierung von Kosten und Qualität zuständig waren. Diese Doktorarbeit befasst sich mit der Rolle von Container-Orchestrierungsplattformen und deren Einfluss auf die Performance moderner Cloud-Anwendungen. Durch gezielte Beiträge verbessern wir einerseits das Verständnis für den aktuellen Stand der Technik und schlagen andererseits Ansätze für Verbesserungen vor.

Beitrag I - Ein systematischer Ansatz für das Benchmarking von Container-Orchestrierungsplattformen

Die Steuerung der Performance von Container-Anwendungen und -Plattformen ist unabhängig vom Anwendungsgebiet (serverless oder konventionell) nicht trivial. Die stetig steigende Zahl an Features, Algorithmen und Steuerungsprozesse moderner Container-Orchestrierungsplattformen verkompliziert deren Konfiguration erheblich. Erschwerend hinzu kommt noch, dass all diese Elemente gleichzeitig agieren und sich gegenseitig beeinflussen. Ein gesamtheitlicher Ansatz für die Leistungsbewertung von Container-Orchestrierungsplattformen ist daher notwendig. Die umfassende Konfigurierbarkeit dieser Plattformen stellt Anwender vor Probleme, da die Menge an zu berücksichtigenden Einstellungen sehr groß ist.

Der erste Beitrag in dieser Doktorarbeit geht diese Herausforderungen durch die Einführung eines systematischen Ansatzes für das Benchmarking von Container-Orchestrierungsplattformen an. Wir identifizieren zunächst acht Anforderungen an Container-Orchestrierungsplattformen und ordnen diesen Leistungsmaße zu. Darauf basierend entwerfen wir eine Architektur, die in der Lage ist, diese Metriken zuverlässig zu erfassen, ohne dabei weitgehende Annahmen an das Testsystem zu stellen. Die Implementierung unseres Ansatzes ist die Open-Source-Software COFFEE, die die Ausführung detaillierter Testkampagnen ermöglicht. Wir demonstrieren den Einsatz und das Potential von COFFEE in vier Fallstudien. Diese vergleichen die Plattformen Kubernetes und HashiCorp Nomad auf Maschinen aus einer öffentlichen Cloud-Umgebung sowie aus einer lokalen, selbst verwalteten Umgebung. Die Fallstudien befassen sich mit der Bereitstellung und Netzwerkcommunication von Container-Anwendungen, Fehlererkennung und -behandlung, Update-Prozessen und Leistungsuntersuchungen von persistentem Speicher für Container. Unsere Untersuchungen zeigen zum Beispiel unterschiedliche Bereitstellungsgeschwindigkeiten und Trade-Offs bei Update-Prozessen auf. COFFEE ist die erste Software, die es erlaubt, leistungskritische Aspekte von verschiedenen Container-Orchestrierungsplattformen umfassend zu untersuchen.

Beitrag II - Eine empirische Studie zu Container-Startzeiten und deren Einflussfaktoren

Container-Startzeiten sind ein kritischer Leistungsfaktor für Container-Anwendungen und beeinflussen zahlreiche Prozesse wie Skalierung, Fehlerbehandlung und Updates. Während Container-Startzeiten bereits in früheren Arbeiten mit anderen Virtualisierungstechnologien wie virtuellen Maschinen und Unikernels verglichen wurden, ist derzeit wenig bekannt über Faktoren, die Container-Startzeiten beeinflussen. Wissen über solche Faktoren ist essenziell, um unterschiedliche Performance von Containern erklären zu können. Cloud-Anbieter optimieren ihre Infrastruktur für Container, z.B. durch spezialisierte Betriebssysteme. Dies unterstreicht die Wichtigkeit zu untersuchen, wie Hardware- und Software-Faktoren Container-Startzeiten beeinflussen. Basierend auf Ergebnissen dieser Untersuchungen können dann Richtlinien für das optimale Erstellen und Bereitstellen von Containern abgeleitet werden. Aktuell ist aufgrund fehlender Daten das Optimierungspotenzial von Container-Startzeiten weitgehend unerforscht.

Der zweite Beitrag in dieser Doktorarbeit ist eine umfassende empirische Studie, die eine Datengrundlage zum Identifizieren von Einflussfaktoren auf Container-Startzeiten bietet. Basierend auf einem neu erfassten Datenset von über 200.000 Open-Source-Containern von der Plattform Docker Hub, führen wir statistische Analysen zu den Eigenschaften von Container-Anwendungen, wie z.B. der Größe der Container und die Anzahl der Schichten des Dateisystems, durch. Auf einer repräsentativen Stichprobe von über 1000 Containern führen wir ausführliche Startzeit-Messungen in zwei Testumgebungen durch und zeigen, dass Startzeiten innerhalb einer Umgebung zwischen wenigen Millisekunden und über zehn Sekunden variieren können. Dabei stellen wir fest, dass mehrere Faktoren Container-Startzeiten beeinflussen und das Problem nicht auf die Untersuchung einzelner reduziert werden kann. Weitere Messungen in der Google Cloud zeigen, dass die Konfiguration der Maschine, auf der der Container ausgeführt wird, Startzeiten erheblich beeinflusst. Dabei sticht vor allem der Einfluss des Festplattentyps heraus. Unsere Studie zeigt, dass sowohl Eigenschaften des Containers als auch der ausführenden Maschine gemeinsam für eine genaue Analyse von Container-Startzeiten berücksichtigt werden müssen. Unsere Forschung stellt die bisher umfassendste Studie zu Container-Startzeiten dar und dient als wichtige Grundlage zur Identifizierung von Optimierungspotenzial.

Beitrag III - Ein Ansatz für kontinuierliches, dezentrales Autoscaling

Autoscaling ist eine der wichtigsten Aufgaben von Container-Orchestrierungsplattformen, da es direkt Servicequalität und Betriebskosten beeinflusst. Moderne Cloud-Anwendungen wie Microservice-Anwendungen bestehen typischerweise aus vielen Containern, die unabhängig skaliert werden können. Die Implementierung von Autoscaling für die Vielfalt und Diversität der Anwendungen ist eine große Herausforderung. Da immer mehr Nutzer in die Cloud wechseln, sind die Arbeitslasten deutlich diverser geworden, unter anderem mit wechselnden Zugriffsmustern und -parametern. Traditionelle Autoscaler haben Probleme mit diesen dynamischen Arbeitslasten umzugehen, da festgeschriebene Intervalle deren Anpassbarkeit hemmen. Die steigende Anzahl an Services und häufige Updates bringen Autoscaler mit trainierten Anwendungsmodellen an ihre Grenzen. Dies unterstreicht den Bedarf für neue anpassungsfähige und effiziente Autoscaling-Lösungen.

Der dritte Beitrag dieser Doktorarbeit adressiert die genannten Herausforderungen durch einen neuen Ansatz für kontinuierliches, dezentrales Autoscaling. Im Gegensatz zu etablierten Verfahren erlauben wir jeder Service-Instanz eine Skalierungsaktion zu veranlassen. Die Entscheidungen der Instanzen werden so über die Zeit verteilt, dass ein quasi-kontinuierlicher Skalierungsprozess entsteht. Wir evaluieren unseren Ansatz auf drei Abstraktionsebenen. Ein Warteschlangenmodell im diskreten Zeitspektrum erlaubt uns die Analyse der Reaktionsgeschwindigkeit und Konvergenz unseres Ansatzes. Eine ereignisorientierte Simulation beschreibt und evaluiert das Skalierungsverhalten für unterschiedliche Konfigurationen und Systemgrößen. Experimente in öffentlichen Cloud-Umgebungen zeigen, dass unser Ansatz besseres Skalierungsverhalten als etablierte Verfah-

ren ermöglicht, besonders bei dynamischen Arbeitslasten. Unser Ansatz ist eine robuste und flexible Autoscaling-Lösung für moderne Cloud-Anwendungen und hinterfragt einige etablierte Paradigmen zur Entwicklung von Autoscalern.

Beitrag IV - Erweiterung von Microservice-Simulation durch authentische Container-Orchestrierung

Die Konfiguration von Container-Orchestrierungsplattformen, z.B. für Autoscaling oder Verteilung (Scheduling) von Containern, ist ein komplexer und teurer Prozess. Performance-Simulationen bieten einen kosteneffizienten Weg, um Konfigurationsänderungen zu testen, ohne dabei auf eine teure Testumgebung angewiesen zu sein. Aktuelle Performance-Simulationen für moderne Cloud-Anwendungen wie Microservices unterstützen keine oder nur stark vereinfachte Container-Orchestrierungsprozesse. Diese fehlende Unterstützung hat zur Folge, dass spezielle Konfigurationen, wie z.B. Scheduling-Strategien, nur mit erheblichem manuellem Aufwand getestet werden können. Dies limitiert die Anwendbarkeit dieser Simulationen und verhindert, dass Anwender diese zum Optimieren von Konfigurationen nutzen können.

Der vierte Beitrag dieser Doktorarbeit ermöglicht die Untersuchung von Konfigurationen von Container-Orchestrierungsprozessen durch Simulationen. Wir präsentieren einen Ansatz zur Integration von Kubernetes-Komponenten in eine Microservice-Simulation. Dieser basiert auf Analogien zwischen ereignisorientierten Simulationen und ereignisgetriebenen Systemen. Ein Adapter ermöglicht nahtlose Kommunikation zwischen der Simulation und den Kubernetes-Komponenten, ohne dabei Kubernetes-spezifische Logik in der Simulation integrieren zu müssen. Dieses Zusammenspiel erlaubt es Kubernetes-Komponenten und deren Konfigurationen in einer simulierten Umgebung zu testen, die eine echte Cloud-Umgebung imitiert. In zwei Fallstudien mit dem Scheduler und Cluster-Autoscaler von Kubernetes demonstrieren wir, dass Ereignisse in realen Cloud-Systemen akkurat simuliert werden können. Dadurch erweitern wir die Anwendungsszenarien der Simulation und verbessern deren Genauigkeit bzgl. simulierter Leistungsmaße wie z.B. Antwortzeiten. Unser Ansatz ist der erste, der es erlaubt, mehrere Kubernetes-Komponenten gleichzeitig in einer Simulation zu evaluieren. Er ermöglicht es, Konfigurationen von Orchestrierungsprozessen, ohne die Notwendigkeit einer teuren Testumgebung, zu analysieren.

Gemeinsam eröffnen alle vier Beiträge dieser Doktorarbeit neue Wege für das effiziente und zuverlässige Betreiben von Container-Plattformen und modernen Cloud-Anwendungen und stellen Fortschritte auf diesem Forschungsgebiet dar. Unsere Beiträge stellen eine Basis und Motivation für zukünftige Arbeiten im Bereich der Performance-Analyse und -Modellierung von Container-Orchestrierungsplattformen bereit.

Danksagung

Ich möchte mich bei vielen Menschen bedanken, die mich während meiner Promotion privat und beruflich unterstützt haben. An erster Stelle stehen dabei meine Freundin Jenny, meine Freunde aus Würzburg und der Heimat, meine Eltern, Geschwister, Großeltern und der Rest meiner Familie. Ohne euch wäre das Projekt Promotion und die damit verbundene Lebensphase nicht erfolgreich gewesen.

Dankbar bin ich auch für die Unterstützung vieler Menschen mit denen ich zusammenarbeiten durfte. Allen voran danke ich meinem Betreuer Samuel Kounev, der mir die Möglichkeit gegeben hat zu promovieren, über die vergangenen Jahre frei zu forschen und mich stets gut beraten hat. Ich hatte das Vergnügen mit vielen Kollegen meinen beruflichen Alltag am Lehrstuhl zu verbringen. Darunter sind André Bauer, Lukas Beierlieb, Vanessa Borst, Timo Dittus, Simon Eismann, Simon Engel, Daniel Grillmeyer, Johannes Grohmann, Lorenz Gruber, Marius Hadry, Nikolas Herbst, Stefan Herrleben, Robert Leppich, Veronika Lesch, Yannik Lubas, Maximilian Meißner, Thomas Prantl, Florian Spieß, Ivo Rohwer, Norbert Schmitt, Maximilian Schwinger und Michael Stenger. Weiterhin bedanke ich mich für die Zusammenarbeit bei den Co-Autoren meiner Publikationen aus verschiedenen Universitäten in Deutschland und weltweit: Kyle Chard, Ian Foster, Sebastian Frank, Stefan Geißler, Alireza Hakamian, Pavithra Harsha, Tobias Hoffeld, Stanislav Lange, Chitra K. Subramanian, Jákím von Kistowski, André van Hoorn (in memoriam) und Lion Wagner. Während meiner Zeit am Lehrstuhl durfte ich auch mit vielen Studenten durch Praktika und Abschlussarbeiten zusammenarbeiten. Ich möchte mich bei Nicholas Erhard, Patrick Haas, Johannes Kohlmann, Jonas Mathiasch, Lukas Mönch, Lukas Kilian Schumann und Oliver Wizemann für deren Beiträge zu meinen Forschungsarbeiten bedanken. Abschließend möchte ich auch bei der Studienstiftung des deutschen Volkes für die finanzielle und ideelle Förderung meines Promotionsstudiums bedanken.

Ohne all diese Menschen wären die letzten Jahre sicherlich nicht dieselben gewesen und ich bin dankbar, dass ich diese Zeit mit euch verbringen und zusammen gestalten durfte.

Publication List

Peer Reviewed Journal Articles

- [SGL⁺25] Martin Straesser, Stefan Geißler, Stanislav Lange, Lukas Kilian Schumann, Tobias Hofffeld, and Samuel Kounev. Trust Your Local Scaler: A Continuous, Decentralized Approach to Autoscaling. Performance Evaluation, Volume 167, Elsevier, January 2025.
- [HLS⁺22] Elia Henrichs, Veronika Lesch, Martin Straesser, Samuel Kounev, and Christian Krupitzer. A Literature Review on Optimization Techniques for Adaptation Planning in Adaptive Systems: State of the Art and Research Directions. Information and Software Technology, Volume 149, Elsevier, September 2022.

Peer Reviewed Conference Papers

- [SHF⁺24] Martin Straesser, Patrick Haas, Sebastian Frank, Alireza Hakamian, André van Hoorn, and Samuel Kounev. Kubernetes-in-the-Loop: Enriching Microservice Simulation Through Authentic Container Orchestration. 16th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS), Springer, January 2024.
- [SBL⁺23] Martin Straesser, André Bauer, Robert Leppich, Nikolas Herbst, Kyle Chard, Ian Foster, and Samuel Kounev. An Empirical Study of Container Image Configurations and Their Impact on Start Times. 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, July 2023.
- [SMBK23] Martin Straesser, Jonas Mathiasch, André Bauer, and Samuel Kounev. A Systematic Approach for Benchmarking of Container Orchestration Frameworks. 14th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, April 2023.

- [SEvK⁺23] [Martin Straesser](#), Simon Eismann, Jóakim von Kistowski, André Bauer, and Samuel Kounev. Autoscaler Evaluation and Configuration: A Practitioner’s Guideline. 14th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, April 2023.
- [SGvK⁺22] [Martin Straesser](#), Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling. 13th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, April 2022.
- [LTB⁺25a] Christopher Lohse, Diego Tsutsumi, Amadou Ba, Pavithra Harsha, Chitra Subramanian, [Martin Straesser](#), and Marco Ruffini. Causal Latency Modelling for Cloud Microservices. 18th IEEE International Conference on Cloud Computing (CLOUD), IEEE, July 2025.
- [LSBK25] Yannik Lubas, [Martin Straesser](#), André Bauer, and Samuel Kounev. Generating Executable Microservice Applications for Performance Benchmarking. 16th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, May 2025.
- [BGP⁺24] André Bauer, Maxime Gonthier, Haochen Pan, Ryan Chard, Daniel Grzenda, [Martin Straesser](#), J. Gregory Pauloski, Alok Kamatar, Matt Baughman, Nathaniel Hudson, Ian Foster, and Kyle Chard. An Empirical Investigation of Container Building Strategies and Warm Times to Reduce Cold Starts in Scientific Computing Serverless Functions. 20th IEEE International Conference on e-Science (eScience), IEEE, September 2024.
- [BDS⁺24] André Bauer, Timo Dittus, [Martin Straesser](#), Alok Kamatar, Matt Baughman, Lukas Beierlieb, Marius Hadry, Daniel Grillmeyer, Yannik Lubas, and Samuel Kounev, Ian Foster, and Kyle Chard. Unveiling Temporal Performance Deviation: Leveraging Clustering in Microservices Performance Analysis. 15th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, May 2024.
- [BSL⁺23] André Bauer, [Martin Straesser](#), Mark Leznik, Lukas Beierlieb, Marius Hadry, Nathaniel Hudson, Kyle Chard, Samuel Kounev, and Ian Foster. Searching for the Ground Truth: Assessing the Similarity of Benchmarking Runs. 14th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, April 2023.
- [FWH⁺22] Sebastian Frank, Lion Wagner, Alireza Hakamian, [Martin Straesser](#), and André van Hoorn. MiSim: A Simulator for Resilience Assessment of Microservice-Based Architectures. 22nd IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, March 2023.

- [BSB⁺22] André Bauer, Martin Straesser, Lukas Beierlieb, Maximilian Meissner, and Samuel Kounev. Automated Triage of Performance Change Points Using Time Series Analysis and Machine Learning. 13th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, July 2022.
- [GSC⁺21] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. 12th ACM/SPEC International Conference on Performance Engineering (ICPE), Association for Computing Machinery, April 2021.

Peer Reviewed Workshop Papers

- [SEK24] Martin Straesser, Nicholas Erhard, and Samuel Kounev. An Empirical Study on the Impact of Selected Host Configuration Parameters on Container Start Times. 15th Symposium on Software Performance (SSP), Gesellschaft für Informatik e.V., November 2024.
- [LSR⁺25] Yannik Lubas, Martin Straesser, Ivo Rohwer, Samuel Kounev, and André Bauer. Microservice Applications and Their Workloads on GitHub. 8th Workshop on Hot Topics in Cloud Computing Performance (HotCloudPerf), Association for Computing Machinery, May 2025.
- [LTB⁺25b] Christopher Lohse, Diego Tsutsumi, Amadou Ba, Pavithra Harsha, Chitra K. Subramanian, Martin Straesser, and Marco Ruffini. Causal Discovery for Cloud Microservice Architectures. AAAI Workshop on Deployable AI (DAI), Association for the Advancement of Artificial Intelligence, March 2025.
- [FSW⁺24] Sebastian Frank, Martin Straesser, Lion Wagner, Patrick Haas, Alireza Hakamian, Samuel Kounev, and André van Hoorn. Simulating Microservice-based Architectures for Resilience Assessment Enriched by Authentic Container Orchestration. Software Engineering, Gesellschaft für Informatik e.V., February 2024.

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Shortcomings of Existing Approaches	4
1.3	Guiding Goals and Research Questions	6
1.4	Contribution Summary	9
1.5	Thesis Outline	13
I	Foundations and State of the Art	15
2	Foundations	17
2.1	Terms and Paradigms for Modern Cloud Applications and Platforms .	17
2.2	Container Virtualization	20
2.3	Container Orchestration Frameworks	22
2.4	Autoscaling	26
2.5	Software Performance Benchmarking	28
2.6	Basics of Probability and Random Variables	30
2.7	Queueing Networks and Stochastic State Processes	33
3	State of the Art	37
3.1	Comparison and Benchmarking of Container Orchestration Frameworks	37
3.2	Empirical Studies on Container Performance and Optimization	40
3.3	Autoscaling of Modern Cloud Applications	43
3.4	Simulation of Container Orchestration	49
II	Contributions	53
4	Benchmarking of Container Orchestration Frameworks	55
4.1	Identifying Benchmarking Use Cases and Scope	56
4.2	Requirements and Metrics	59
4.3	Design Considerations	61
4.4	The Benchmarking Framework COFFEE	63
4.5	Case Study I: Container Provisioning and Networking	69
4.6	Case Study II: Failure Recovery	73
4.7	Case Study III: Rolling Updates and Load Balancing	74
4.8	Case Study IV: Container Storage	77
4.9	Summary and Discussion	79

Contents

5	An Empirical Study on Factors Impacting Container Start Times	83
5.1	Container Starts and Their Performance Metrics	84
5.2	The Docker Hub Dataset	86
5.3	Study Design	91
5.4	Variance of Container Start Times	96
5.5	Influence of Image Configuration Parameters	98
5.6	Influence of Platform Parameters	101
5.7	Summary and Discussion	108
6	A Continuous, Decentralized Approach to Autoscaling	113
6.1	Experimental Insights Into Production-Grade Autoscaling	115
6.2	Challenges for Production-Ready Autoscaling	125
6.3	The Idea of Continuous Decentralized Autoscaling	129
6.4	Algorithm and Parameters of Continuous Decentralized Autoscaling	133
6.5	A Discrete-Time Queueing Model for CPU-Based Autoscaling	138
6.6	Simulation Study	147
6.7	Experimental Evaluation	156
6.8	Beyond CPU-Bound Container Applications	163
6.9	Summary and Discussion	167
7	Enriching Microservice Simulation Through Container Orchestration	173
7.1	The MiSim Microservice Simulator	175
7.2	Foundational Container Orchestration Entities and Models	176
7.3	Connecting Discrete-Event Simulation and Event-Driven Systems	177
7.4	Dealing with Incompatible Events and Data	179
7.5	Case Study I: Kubernetes Scheduling Policies in a Global Cluster	181
7.6	Case Study II: Evaluating Expansion Policies for Cluster Autoscaling	186
7.7	Summary and Discussion	190
III	Conclusion and Outlook	193
8	Summary	195
9	Open Challenges and Future Work	199
	List of Figures	203
	List of Tables	205
	Acronyms	207
	Bibliography	209

Chapter 1

Introduction

Cloud computing has revolutionized the world of software applications and internet services over the past decades. Cloud computing allows customers to use resources from a cloud provider on demand without significant manual overhead. The cloud provider owns a large pool of computing resources that allow customers to scale their applications at will. Different service models, such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), give users different levels of control and automation [Mel11]. An essential role in the broad adoption and success of cloud computing is played by public clouds operated by large tech companies, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. The service offerings in public clouds are very diverse. AWS, for example, offers its customers over 300 different services.¹ According to Gartner [Gar24], the end-user spending for public cloud services grew from 175 billion US dollars in 2017 to a forecast of 675.4 billion US dollars in 2024.

The basis for cloud computing and efficient and secure sharing of resources is virtualization. In public cloud services, there is usually a combination of two virtualization levels. The first level consists of virtual machines abstracting the underlying physical host. Virtual machines are the basic building blocks for most IaaS offerings. However, the business model of many companies is primarily software development, while infrastructure management (e.g., managing virtual machines and operating systems) is a rather undesirable task. This is the reason for the evolution of serverless computing, a cloud paradigm in which users develop and run cloud software without allocating and managing virtualized servers and resources or being concerned about other operational aspects [KHA⁺23]. The basis for the new, application-oriented generation of cloud services is the second level of virtualization: container virtualization. Several processes (containers) share a common operating system kernel in this concept. As a result, containers are more lightweight and can be started more quickly than conventional virtual machines [WRK⁺19]. According to the CNCF Annual Report 2023 [Clo24], 76% of surveyed organizations use containers for a few, most, or all of their production applications and business segments.

The success story of container virtualization is closely intertwined with newer trends in software development and architecture. For instance, in the microservice architecture, large applications are broken down into smaller, independently deployable components, the so-called microservices. These microservices are typically deployed in

¹<https://aws.amazon.com/products>

different containers. Microservice topologies can grow to be quite large. According to a report by Huye et al. [HSS23], Meta's microservice topology contained 18,500 active services and over 12 million service instances. This underscores the need for automated management and coordination of container applications, as manual management is not feasible even for significantly smaller topologies. In addition to web services, other workloads, such as batch jobs or AI training, are increasingly being executed in containers. The role of automated container management is taken over by container orchestration frameworks. These frameworks handle a variety of tasks, such as container deployment, autoscaling, or health checks. They have become an integral part of modern cloud environments. They form the basis for many cloud service offerings and serverless applications, further enhancing the efficiency and scalability of modern cloud applications.

Kubernetes is by far the best-known and most widely used container orchestration framework to date. The Cloud Native Computing Foundation recorded increasing user numbers for years and describes Kubernetes as a "core technology" [Clo24]. Other sources refer to Kubernetes as the "operating system" [The23] of the cloud. In addition to the original Kubernetes, numerous widespread commercial variants such as RedHat OpenShift or VMware Tanzu exist. All major public cloud providers also offer managed versions of Kubernetes, such as Amazon Elastic Kubernetes Service or Google Kubernetes Engine (GKE). Only a few alternatives to Kubernetes are relevant in practice, including HashiCorp Nomad, Docker Swarm, and CloudFoundry.

A key factor for the success of modern cloud applications and services is not only their functional properties but also their performance. Performance includes both user-oriented properties, such as availability and response speed, and provider-oriented properties, such as deployment costs, resource usage, and energy efficiency. Container orchestration frameworks significantly determine the performance of modern cloud applications through their capabilities, such as autoscaling and network traffic management [MTK22, PCB⁺19, TBVL⁺18, TVLLJ19, VBT⁺19]. This thesis discusses different performance aspects related to container orchestration frameworks. Performance aspects of container orchestration frameworks and connected tasks are becoming increasingly important, especially for cloud providers, as many performance management tasks need to be handled by cloud providers in the course of the increasing use of serverless applications. Since customers only pay for the resources they actually use, providers must achieve efficient utilization of their infrastructure without risking performance losses for user applications. This thesis addresses the role of container orchestration frameworks in the performance of modern cloud applications. We analyze and assess the state of the art and propose approaches for improvement through targeted contributions.

1.1 Problem Statement

Regardless of the environment (public, private, or even multi-cloud) or service context (serverless or traditional), managing the performance of container applications and clusters is non-trivial. As outlined before, container orchestration frameworks offer

numerous functionalities that can impact the performance of deployed applications. For example, autoscaling directly impacts both customer experience and operating costs, load balancing is crucial for efficient resource usage in the cluster, and scheduling controls which containers share resources of the same host. Practitioners face the problem of selecting and configuring container orchestration frameworks for their infrastructure. This problem is becoming more challenging as modern frameworks like Kubernetes offer an ever-growing set of functionalities, control processes, and algorithms that all act in parallel and potentially impact each other's behavior. For example, the speed of container deployment and scheduling could impact autoscaling periods and cooldowns; load balancing influences the processing of user requests during application updates. A holistic view is needed for performance assessment, and control algorithms must be optimized together. Modern container orchestration frameworks offer fine-grained options to define the behavior of the cluster control plane, enlarging the configuration space to explore for practitioners. In summary, we formulate the following challenge.

Challenge 1: The broad functionality of modern container orchestration frameworks complicates their configuration and performance assessment.

A crucial factor that directly impacts the performance of containerized applications is container start times. Start times influence the scaling behavior of an application, its failure recovery, update behavior, and more. While it has been shown that containers offer faster start times than conventional virtual machines [WRK⁺19], relatively little is known about the impacting factors on the start times of containers. More knowledge about these factors is necessary to answer questions like "Why does Container A start faster than Container B?". Cloud providers may also be interested in providing optimized infrastructure for containers. For example, Google Cloud offers a container-optimized operating system for their managed Kubernetes service. This opens the question of how the executing host and hardware impact container start times. Once we have pointed out influencing factors, we can derive guidelines for building fast-starting containers and recommendations on how to deploy containers, enhancing the scalability and failure recovery of applications. However, by now, the potential of lowering container start times is widely unexplored due to the lack of data quantifying influencing factors. This constitutes the second challenge.

Challenge 2: There is a lack of knowledge about the factors impacting container start times and differences between container images.

As mentioned earlier, modern cloud applications, like microservice applications, are typically deployed in multiple containers. This allows for fine-grained scaling, as different parts of the application can be handled independently. However, the question arises of how to implement autoscaling for the large variety of microservices and containers. Container orchestration frameworks typically employ simple threshold-based autoscaling algorithms that react to changing load intensities and resource usage. Determining resource usage characteristics and configuring autoscaling for the mass of applications

is challenging for practitioners [ADPDM18, ATGC20]. Moreover, most autoscalers rely on monitoring data that can be costly to collect in large-scale distributed applications and cloud environments. As more companies move their applications to the cloud and more users use cloud services, workload characteristics have also changed. As datasets from Microsoft Azure [ZGC+21] and Globus Compute [BPC+24] show, workloads for cloud applications can be very diverse. This holds for many metrics like arriving requests over time, inter-arrival times, request payloads, and response time distributions. Highly dynamic workloads are challenging for traditional autoscalers as their reaction time is limited by design due to fixed scaling periods and cooldown times. The high number of services and regular updates bring autoscalers with learned or preset application models to their limits. In summary, we formulate our third challenge.

<p>Challenge 3: Traditional autoscaling approaches are not designed for the diversity and scale of modern cloud applications and workloads.</p>

Whether autoscaling, scheduling, application updates, or other tasks of container orchestration frameworks, exploring the configuration space is very time-consuming and costly. Simulations are one way to evaluate the effect of configuration changes without provisioning costly test environments. Multiple performance simulations are available for microservices and other cloud applications [Bam20]. They allow for fine-grained specification of a microservice architecture, its control flow, and service resource demands. Based on these data, they estimate response times for different user operations. Therefore, they are well-suited to analyze the microservice architecture, pinpoint bottlenecks, and identify optimization potential related to the application code. However, the environment in which the application runs in production is typically either overly simplified or not modeled. Hence, evaluating configurations of container orchestration frameworks (e.g., a Kubernetes scheduling policy) in these simulations either requires significant manual implementations or is infeasible. This limits the use cases and expressiveness of the simulations and hinders practitioners from efficiently tailoring the orchestrator configuration to the application needs. This constitutes the fourth challenge.

<p>Challenge 4: The missing connection to container orchestration frameworks limits the applicability of microservice performance simulations.</p>
--

In this thesis, we develop approaches and provide contributions that address these challenges.

1.2 Shortcomings of Existing Approaches

Because container orchestration frameworks are integral parts of modern cloud environments and thus the basis of modern cloud applications, they are also of great research interest. Two main shortcomings of published work currently exist in the

area of performance analysis and benchmarking of container orchestration frameworks. First, many related works do not cover the full range of features of modern container orchestration frameworks and limit themselves to selected aspects. Investigated mechanisms of container orchestration frameworks include container provisioning and availability [JBB⁺19], failure discovery and recovery [VSTK21, BDSCDM24], scalability [DTR⁺18, LLJ⁺19], networking [ZWDZ17, BG18, OHS19], and persistent storage [MTA⁺21]. The limitation to isolated subtasks is insufficient for benchmarking container orchestration frameworks as there are complex interactions between the mechanisms, and all of them act simultaneously. The second shortcoming is that many approaches make limiting assumptions about the framework to be benchmarked. Most of the related works mentioned above are limited to the performance evaluation of Kubernetes. Even if Kubernetes currently has a dominant market position, it is important for a credible independent benchmark also to support alternatives and ensure fairness in the sense that it should not favor individual systems under test [KLvK20, vKAH⁺15].

Numerous studies have also looked at the performance of individual container instances. Start times play an essential role here. One group of studies examines container start times in comparison to other virtualization technologies, such as virtual machines or unikernels [TKT18, WRK⁺19, GSA⁺18, GSAN⁺22, SFSF19, XFJ16]. Other studies compare different technology stacks for containers, such as Docker and Podman [DTLD22, KTK23, MGD⁺23], or focus on the investigation of different low-level container runtimes [KT20, RT19, VKT20]. Many studies find a trade-off between the security and the performance of container applications. Even though the minimization of container start times is desirable in many areas, such as edge computing [SWK22] or serverless computing [GGA24], and individual optimizations have been proposed [AP18, CPB19, QWW⁺20], a systematic investigation of which factors influence container start times is currently missing. In particular, the influence of container image factors (such as image size or number of file system layers) is currently unexplored. Empirical studies are important here in order to explain differences between containers and thus recognize further optimization potential for container start times in the future.

In addition to container provisioning, autoscaling is one of the most important tasks of container orchestration frameworks. Autoscaling has been researched from the beginning of cloud computing and even before the widespread use of containers. Containers enable faster scaling and thus motivate a new generation of autoscalers. Numerous surveys summarize previous works on autoscaling and present taxonomies [SGJN19, ADPDM18, QCB18, CBY18]. Generally, there is a big difference between autoscaling in research and practice. Industrial autoscaling solutions offered in public clouds offer very simple threshold-based autoscaling methods. In contrast, more complex approaches are proposed in research (e.g., based on deep neural networks). Reasons for the lack of adoption of research autoscalers in practice include practitioners' distrust in complex models [Bau21] and significant configuration overheads [ADPDM18, ATGC20]. Furthermore, many related works try to achieve optimal scaling for a specific static application, which is impractical in times of fast release cycles and the multitude of deployed services. Last but not least, many autoscalers

are tuned to a specific workload and system scale and cannot respond adequately to unpredictable events (e.g., due to fixed scaling intervals). New autoscaling approaches are needed to support the modern cloud applications' dynamic characteristics and workloads while remaining explainable and configurable.

A cost-effective and resource-efficient alternative to measurement-driven performance analysis of container orchestration frameworks and individual mechanisms such as autoscaling is simulation. A core challenge here is the accurate mapping of container orchestration and the modeling of its influences on the performance of deployed cloud applications. Established simulators for cloud environments such as CloudSim [CRB⁺11] or OpenDC [MAJ⁺21] have a different focus, such as evaluating resource metrics and optimizing the cloud environment itself. Previous solutions for performance simulation of modern cloud applications, such as microservice applications, are unsuitable for analyzing container orchestration. Either container orchestration mechanisms are not supported at all, or only overly simplified models of individual mechanisms exist [BWB⁺19, FWH⁺22, JPG19, VDR⁺20]. In order to provide an alternative to cost-intensive measurements in real clusters and thus enable efficient optimization of container orchestration, accurate integration of container orchestration mechanisms, their configuration, and their impact on the performance of modern cloud applications into simulations is necessary.

In summary, we identify a clear need for action to address the challenges stated in Section 1.1. For a fair and comprehensive performance analysis and benchmarking of container orchestration frameworks, new approaches are needed to evaluate as many performance-relevant mechanisms of container orchestration frameworks as possible without being developed specifically for a concrete technology stack. Empirical studies on factors influencing container start times are necessary to identify differences between containers and derive further optimization potential. New autoscaling approaches are needed to handle the dynamics and diversity of modern cloud services. Performance simulations for modern cloud applications need to be extended to enable cost-efficient optimization of container orchestration frameworks and their configurations. In Chapter 3, we provide a detailed overview and summary of the current state of the art.

1.3 Guiding Goals and Research Questions

In the previous sections, we defined four challenges related to the performance aspects of modern container orchestration frameworks and the applications they manage. Furthermore, we analyzed that the current state of research has not sufficiently addressed these challenges. In the following, we therefore derive the four main goals of this thesis, which address the previously established challenges. For each goal, we pose research questions (RQs) that need to be answered in order to fulfill the goal.

<p>Goal A: Develop a systematic approach for benchmarking container orchestration frameworks.</p>
--

1.3 Guiding Goals and Research Questions

Goal A addresses Challenge 1, the performance assessment of container orchestration frameworks and their configurations. According to Kounev et al. [KLvK20], a benchmark is "a tool coupled with a methodology for the evaluation and comparison of systems or components with respect to specific characteristics, such as performance, reliability, or security." Therefore, developing a benchmarking approach addresses the need for practitioners to quantify the performance of container orchestration frameworks. This also allows different container orchestration frameworks or configurations of a specific framework to be compared in terms of their performance. Based on the results of benchmarks, operators can choose a container orchestration framework or configuration. We have to answer the following research questions to achieve this goal:

RQ A1: Which features of container orchestration frameworks are feasible to evaluate through benchmarking?

RQ A2: Which metrics can be used to quantify the performance, and how can they be measured?

RQ A3: How to define a benchmark for container orchestration frameworks and enable execution on diverse technology stacks?

The first research question aims to identify the benchmarking scope and use cases of the benchmark. This is important in our context, as container orchestration frameworks offer a broad set of features. We have to identify common tasks for container orchestration frameworks from academic literature and practice without being biased by the features of individual competitors. Once the benchmarking scope has been established, we have to find performance metrics that quantify how well container orchestration frameworks fulfill these tasks. Consequently, we also need to design ways to measure these metrics reliably. These concerns are addressed in RQ A2. Finally, we have to establish a methodology for defining and executing benchmarks for container orchestration frameworks. The main challenge here is to support the diverse technology stacks that different container orchestration frameworks and container clusters might have. The goal is not only to support the current market leader, Kubernetes, but also other competitors.

Goal B: Provide and analyze empirical data on container start times from a representative set of images and environments.

Goal B addresses Challenge 2, the lack of knowledge about the factors impacting container start times and differences between container images. To address this challenge, a measurement-based study is needed that includes as many diverse containers and their start times as possible. Diversity in terms of the characteristics of the containers and the executing hosts is important here, as this is the only way to identify influencing factors. The study can be divided into two parts, which are also reflected in the following research questions.

RQ B1: How to acquire a representative dataset of container images and their start times?

RQ B2: Which factors impact container start times?

The first research question targets the study design, including the search and selection of container images to test and the definition of the measurement methodology. The dataset has to mirror the diversity of container images in practice, reflecting a broad range of characteristics that allow us to derive which factors are influencing container start times and which are not. The second research question focuses on the analysis of the assembled dataset. Statistical analysis is necessary to separate important and neglectable factors influencing container start times. Fulfilling this research goal and answering the research questions will help practitioners identify and explain the start times of their containerized applications and prioritize subjects for improvements.

Goal C: Develop a broadly applicable autoscaling approach for modern cloud deployments capable of handling a large number of services with highly dynamic workloads.

Goal C addresses Challenge 3, the need for novel autoscaling solutions that are designed for the dynamics and scale of modern cloud applications and workloads. As discussed earlier, the number of cloud services and companies moving their applications to the cloud is rapidly growing. The faster development cycles of cloud applications fostered by distributed architectures, like the microservice architecture, as well as a closer connection between software development and operations (DevOps), challenge conventional autoscalers. Instead of solutions that provide optimal scaling for one application and workload type, an approach broadly applicable to different applications and dynamic workloads is needed. To achieve this goal, the following research questions have to be addressed:

RQ C1: Which conceptual weaknesses limit the applicability of conventional autoscalers to modern cloud applications and workloads?

RQ C2: How to design and implement an autoscaling approach for lightweight execution at scale?

RQ C3: How to enable adaptability to highly dynamic workloads?

RQ C4: How to enable adaptability to different characteristics of cloud applications while keeping the configuration manageable?

There are many autoscalers proposed in the literature; however, many of them do not find their way into practice. Autoscaling solutions in public clouds are often rather simple and threshold-based and require thorough configuration [ADPDM18, ATGC20]. The first research question aims to analyze the weaknesses of existing autoscaling solutions and pinpoint why autoscalers struggle with modern, fast-changing applications and workloads. The second research question addresses the concern that many conventional autoscalers rely on monitoring data that can be costly and unreliable to collect, especially in large-scale environments. Single points of failure emerge that

should be addressed by new, robust designs for autoscaling. The last two research questions directly address the requirements derived from Challenge 3, the adaptability to highly dynamic workloads and to different characteristics of cloud applications. To avoid retaining one core weakness of existing approaches and a showstopper for their practical adoption, the proposed concept should offer a configuration method that is manageable for practitioners.

Goal D: Develop an approach for integrating container orchestration frameworks into microservice performance simulations.

Goal D addresses Challenge 4, the missing connection between microservice performance simulations and container orchestration frameworks. There are two main purposes for using simulations in this context. First, practitioners want to understand the performance of their microservice applications in clusters with container orchestration frameworks. Second, practitioners want to evaluate different orchestrator configurations without having to provision test clusters and run costly and time-intensive experiments. The following research questions have to be addressed to realize the integration of container orchestration frameworks into microservice simulations.

RQ D1: How to integrate container orchestration frameworks into microservice performance simulations without requiring detailed models of their behavior?

RQ D2: How to enable seamless transfer of orchestrator configurations into simulations?

The first research question addresses the fundamental support for container orchestration frameworks and their mechanisms (such as scheduling and autoscaling) in microservice performance simulations. To enable the use cases mentioned above, it is crucial that this integration works without having to simplify or fully model the behavior of the container orchestration framework. This is not feasible because of the extensive feature set and concurrently operating control mechanisms of modern container orchestration frameworks (see also Challenge 1). The second research question deals with the large configuration space of container orchestration frameworks and their mechanisms. To allow for the meaningful evaluation of different configurations through simulation, a seamless transfer of the configuration between the real cluster and simulation has to be made possible.

1.4 Contribution Summary

In this section, we provide summaries of the four main contributions of this thesis. The contributions address the challenges stated in Section 1.1 and the goals declared in Section 1.3.

Contribution I - A Systematic Approach for Benchmarking of Container Orchestration Frameworks

The first contribution addresses Challenge 1 by presenting a systematic approach to benchmarking container orchestration frameworks. By doing so, we enable the performance assessment of container orchestration frameworks and their configurations. As preliminary work, we review common definitions of container orchestration from research and industry. We identify eight core requirements for container orchestration frameworks, that is, essential functionalities each container orchestration framework provides. We also associate metrics to quantify the performance of the container orchestration framework with respect to fulfilling these core requirements. In the next step, we design a benchmark architecture that allows us to reliably measure the established metrics without placing limiting assumptions on the system under test. The implementation of our approach is the open-source benchmarking framework COFFEE (short for Container Orchestration Frameworks' Full Experimental Evaluation). COFFEE supports the definition of complex test campaigns that stress different components of container orchestration frameworks.

We demonstrate the potential of our approach in a total of four case studies. The first case study covers container provisioning and networking tasks. Here, we compare container readiness and removal times as well as message round trip times in self-hosted and public cloud Kubernetes and Nomad clusters. In our second case study, we examine the ability of Kubernetes and Nomad to respond to sudden crashes of container applications. We show that Kubernetes can restore availability on the same physical hosts faster than Nomad. In a case study with rolling updates, we show trade-offs between update duration and failed user requests. Finally, we investigate different types of persistent storage for containers running in Kubernetes. We investigate both local storage and different storage types in the Google Cloud.

Our approach is the first to systematically investigate several performance-critical aspects of container orchestration frameworks, supporting and evaluating multiple technology stacks. This contribution thus addresses Goal A by developing a systematic, widely applicable, and comprehensive benchmarking framework for container orchestration frameworks. This contribution has been successfully published as a full research paper at the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE) [SMBK23].

Contribution II - An Empirical Study on Factors Impacting Container Start Times

Contribution II addresses Challenge 2 by presenting an empirical study on the factors influencing container start times. To this end, we present a new dataset of about 200,000 open-source Docker Hub images. We analyze the dataset with a number of features extracted from each container image. This gives us insights into statistical values about various characteristics of modern container applications, for example, the size of container images or the number of their file system layers. On a representative sample of over 1000 container images, we perform repeated start time measurements in a self-hosted environment as well as a Google Cloud cluster. Our results show that start times can vary between hundreds

of milliseconds and tens of seconds in one environment. We also show that no dominant feature determines the container start time; instead, several factors work together. The image size and the number of file system layers are among the most influential image configuration parameters. Our results also show clear differences between environments. Therefore, we conducted a second series of measurements to investigate container start times, specifically on different hosts. We used a Plackett-Burman experimental design [PB46] to investigate the influence of six configuration parameters that can be set when creating a virtual machine in the Google Cloud. Our results show that the storage type (hard disk vs. solid-state drive) has a decisive influence on the start time. Nevertheless, we conclude from our results that image configuration and platform parameters must be considered together to assess container start times accurately.

Our work represents the most comprehensive study of container start times to date. Our Docker Hub dataset is the most recent and one of the largest of its kind and can be used beyond this thesis. This contribution addresses Goal B by improving our understanding of container start times across different container images and environments. This contribution has been successfully published in a full research paper at the 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) [SBL⁺23] and in a workshop paper at the 15th Symposium on Software Performance (SSP) [SEK24].

Contribution III - A Continuous, Decentralized Approach to Autoscaling

Contribution III addresses Challenge 3 by introducing an approach for continuous decentralized autoscaling. Before presenting the actual approach, we conduct a comprehensive preliminary study that analyzes why many research autoscalers are not adopted in practice. We present a total of six key challenges, illustrate them with example experiments in a realistic environment, and analyze how they are addressed in the literature. The difference between our approach and most established autoscalers is that scaling decisions are not made by a centralized approach ("the autoscaler"). Instead, each service instance is given the right to trigger scaling actions. We achieve continuous scaling by distributing the decisions of instances over time using waiting time distributions. Probabilities also play a role in the decision-making process of individual service instances. So-called scaling functions provide probabilities of up- or downscaling actions.

We evaluate our approach on three different levels. Using a model, we obtain theoretical statements about our approach's decision speeds and convergence. With the help of a newly developed discrete-event simulation, we analyze our approach in different configurations and system scales. We show, for example, that our approach delivers almost identical performance for a few tens of requests per second as well as for thousands. Finally, we evaluate our approach in real cloud deployments with different serverless functions. We show that our solution delivers better scaling results than the established Kubernetes Horizontal Pod Autoscaler and the Knative Pod Autoscaler. We also show the applicability of

our approach beyond container applications in a scenario with virtual machines inspired by video streaming.

Our work presents a novel approach that differs conceptually from previous autoscalers. The evaluation of our approach employs, in contrast to most related works, modeling, simulation, and real experiments and provides good evidence for the potential of the approach in practical applications. This contribution thus addresses Goal C by proposing a flexible, broadly applicable autoscaling approach for modern cloud applications that achieves advantages over established methods, especially in highly dynamic workloads. This contribution has been successfully published as a journal paper in the Performance Evaluation journal [SGL⁺25]. The preliminary studies conducted together with an industry partner have been published as full industry papers at the 2022 and 2023 ACM/SPEC International Conference on Performance Engineering (ICPE) [SGvK⁺22, SEvK⁺23]. The former of those papers was selected as the runner-up for the Best Industry Paper Award.

Contribution IV - Enriching Microservice Simulation Through Authentic Container Orchestration

Contribution IV addresses Challenge 4 by taking an approach that allows Kubernetes components (in the form of their original artifacts and configurations) to be integrated into a microservice simulation. We use the state-of-the-art microservice simulation MiSim [FWH⁺22] as a basis. We integrate interfaces and standard algorithms for container orchestration tasks (such as autoscaling, scheduling, and load balancing) into the simulation. The core of the contribution consists of a generic approach to connect a discrete-event simulation (here: MiSim) and an event-driven system (here: Kubernetes). We use special event functions and event transformations that are implemented in an adapter. From the simulation's point of view, the communication with the adapter is an external function call. However, the adapter also provides communication interfaces to components from a real Kubernetes cluster. From the perspective of these components, it acts as a Kubernetes API server that manages a simulated cluster. Using a black-box information repository allows seamless integration of the Kubernetes components and their configurations without implementing orchestrator-specific logic in the simulation. In case studies with the scheduler and cluster autoscaler of Kubernetes, we show that these components and different configurations can be integrated into the simulation and act in the same way as in a real cloud environment. We show that this widens the use cases of the simulation and improves its accuracy.

Our approach is the first to allow multiple Kubernetes components to operate together in conjunction with a microservice simulation. It addresses Goal D by allowing us to investigate the relationship between container orchestration frameworks and microservice performance and to evaluate different container orchestrator configurations without setting up a costly test environment. This contribution has been successfully published as a full research paper at the 16th

EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS) [SHF⁺24].

All contributions offer ways towards efficient and reliable operation of container clusters and modern cloud applications. Together, they significantly advance the research field and could contribute to the success of cloud computing and cloud applications in the future.

1.5 Thesis Outline

This thesis is structured into three parts. Part I discusses the foundations of this work and examines related works and their shortcomings in detail. Within Part I, Chapter 2 focuses on the foundations and introduces essential terms and paradigms for modern cloud applications and platforms. A detailed look at container virtualization, as well as technical concepts and details on container orchestration frameworks, follows this. A separate section is spent on the basics of autoscaling, as it plays a major role in different contributions. Overviews of software performance benchmarking, probability and random variables, queuing networks, and stochastic state processes wrap up the foundations. Chapter 3 discusses the state of the art. The chapter is divided into four sections that align with the contributions of this thesis. Hence, the first section focuses on related work on performance analysis and benchmarking of container orchestration frameworks. The second section summarizes empirical studies addressing container performance and optimization efforts to reduce container start times. We also discuss related works from the areas of autoscaling of modern cloud applications and simulation of container orchestration to position our Contributions III and IV.

Part II presents the main contributions of the thesis. Each contribution is represented by one dedicated chapter. Chapter 4 focuses on benchmarking of container orchestration frameworks. It outlines our design process of the benchmarking framework COFFEE by first identifying the benchmarking scope and use cases and later explaining the requirements, metrics, and implementation details of the benchmarking approach. Our approach is used in four case studies concerning container provisioning and networking, failure recovery, rolling updates and load balancing, and container storage. Chapter 5 presents our empirical study on factors impacting container start times. It first introduces our container image dataset acquired from Docker Hub. Afterward, we explain how container start time measurements have been conducted and analyze our results in multiple steps. We perform a variance analysis for single container images and investigate the influence of image configuration and platform parameters.

Chapter 6 introduces our approach to continuous decentralized autoscaling. It first presents preliminary studies that highlight open challenges for production-ready autoscaling with measurements from representative environments. Before diving into the details of our approach, we introduce the idea on a high level and give a motivating example. The evaluation of our approach is split into several sections that address model-based, simulation-based, and measurement-based evaluation experiments. Chapter 7 presents Contribution IV, the connection between microservice simulation

and container orchestration. First, the MiSim microservice simulator, which has been used as a foundation for our implementation, is introduced in a dedicated section. Next, our approach is presented in multiple steps, including the integration of foundational container orchestration entities and models, a general concept for connecting discrete-event simulation and event-driven systems, and a methodology to deal with incompatible events and data. Finally, we present two case studies that show the capabilities of our approach in representative scenarios.

Part III wraps up this thesis by drawing conclusions and considering possible directions for future research. Chapter 8 provides a summary of all contributions and results of this thesis. Chapter 9 states challenges that remain open or emerge from this thesis. It also gives concrete proposals for future work that build on the findings of this thesis.

Part I

Foundations and State of the Art

Chapter 2

Foundations

This chapter briefly discusses the fundamental concepts that lay the foundation for our research. Section 2.1 discusses essential terms and current paradigms for developing and operating modern cloud applications and platforms. Section 2.2 explains the concepts of container virtualization and the differences to virtual machines, while Section 2.3 introduces state-of-the-art container orchestration frameworks. As it plays an important role in our research, the basics of autoscaling are introduced in Section 2.4. Section 2.5 outlines the field of software performance benchmarking with a focus on experimental design. Section 2.6 provides an overview of essential terms and relationships within the domain of probability and random variables. Section 2.7 concludes this chapter with a short overview of queueing networks and stochastic state processes.

2.1 Terms and Paradigms for Modern Cloud Applications and Platforms

The gigantic growth and demand for digital services and internet services in the 21st century have been made possible not only by advances in hardware but also by advances in software development. Earlier generations of user-facing software were designed and implemented as monolithic applications. Due to the increasing complexity and growing number of users, especially for internet services, the vision of discrete services that can be implemented and operated independently quickly emerged. The service-oriented architecture [PvdH07] was then developed as an architectural style with this focus. In the mid-2010s, the microservice architecture emerged as a realization of this style.

The microservice architecture divides the application into different microservices. The literature has different, partly overlapping, partly contradictory definitions of what exactly a microservice is [DFLM19]. In the following, we will limit ourselves to the properties and characteristics usually assigned to microservices without claiming completeness. Microservices are independently deployable software components with a well-defined context and functional scope. Complex application logic is realized through the interaction of different microservices. Several microservices are often involved in the processing of a user request. The relationships between microservices in an application are often represented as a directed, usually acyclic, graph [CAB⁺22]. A microservice offers other microservices and possibly uses an Application Programming

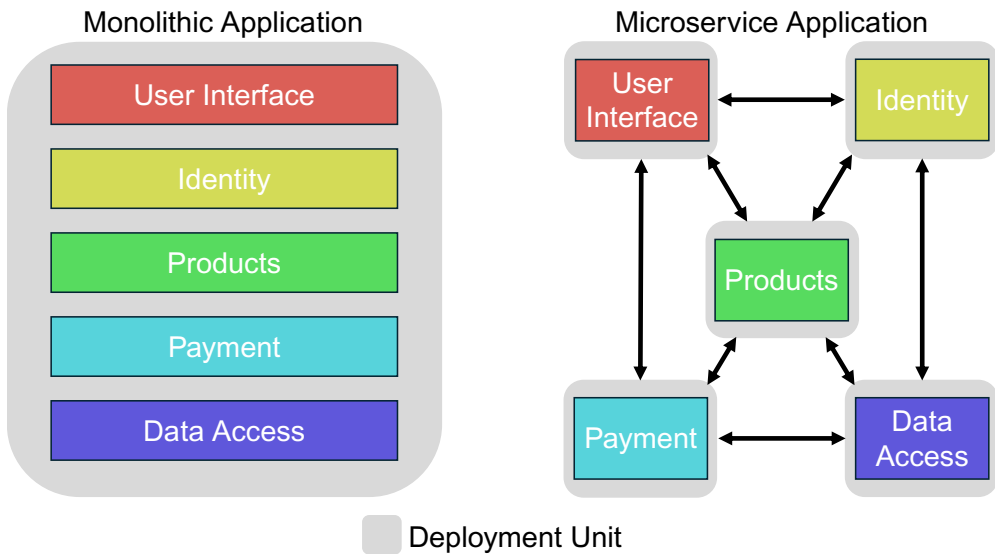


Figure 2.1: Monolithic application vs. microservice application.

Interface (API) through which dedicated functions can be called. Communication between microservices is network-bound and often involves HTTP requests or remote procedure calls (RPCs). HTTP-based microservices often offer several endpoints that can be called via different URL paths and possibly using different HTTP verbs. The totality of all endpoints of a microservice forms its API.

Figure 2.1 shows conceptual differences between monolithic applications and microservice applications. Large-scale monolithic applications can be characterized by high maintenance effort and limited scalability. Microservices, on the other hand, can be deployed and managed independently. For example, autoscaling and updates can be performed separately for different microservices. Due to the separation and well-defined interfaces of each microservice, its intrinsic logic can be viewed as a black box. This also enables the independent development of microservices, which facilitates the division of work between different development teams and thus increases the parallelism and speed of software development. Therefore, microservices are well suited to realizing concepts such as continuous integration and delivery (CI/CD), which aim to increase the frequency of software updates and releases [BHJ16]. Similarly, the division of an application into microservices and their loose coupling via network-based communication allows the use of different implementation technologies (programming languages, libraries, web frameworks). This allows companies to leverage the strengths of individual technologies or development teams in a more targeted manner.

The granularity of microservices is discussed conversely [HBK20]. A particularly fine-grained manifestation of the idea of distributed services is serverless functions. The serverless aspect refers to the fact that these functions are often deployed on serverless platforms, which we will introduce later in this section. Serverless functions

2.1 Terms and Paradigms for Modern Cloud Applications and Platforms

typically offer only one functionality and are invoked via a network call (e.g., with HTTP requests) or message queues. They provide the finest scalability and usually achieve fast start and response times due to the few dependencies that need to be loaded. With serverless functions, only the source code often has to be written, while the serverless platform supports packaging, releasing, and several operational tasks (such as scaling or updating).

Parallel to the spread of microservice architecture, the so-called DevOps philosophy has also been adopted in many companies. DevOps emphasizes a stronger connection between software development and operations. DevOps is realized through a high degree of automation (e.g., through CI/CD pipelines). In these pipelines, software updates quickly pass through various stages (such as unit and integration tests and security screenings) before being deployed in production. If errors occur in the operation of the software, the path to the developers is short, and updates can be delivered quickly due to the fast release cycles. DevOps is also aided by the fact that the computing infrastructure (e.g., servers, networks) is much easier to deploy and manage with a new generation of tools. Infrastructure as Code (IaC) tools, along with configuration management software, enable computing resources to be deployed and managed automatically. The basis for this is the easy availability of resources, which is made possible by the concept of cloud computing.

The International Organization for Standardization (ISO) defines cloud computing in ISO/IEC 19941:2017 as “a paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand.” A distinction is made between public, private, or hybrid clouds. Public clouds, in particular, are essential for the success of internet services in general and overcome the capacity and scaling problems of conventional on-premise computing environments. The largest public cloud providers currently include Amazon, Google, and Microsoft. At the beginning of 2025, Microsoft alone operated more than 300 data centers with more than 442.000 kilometers of network.¹ These large-scale environments offer a high degree of reliability and promote global availability and low latency. This motivates the paradigm of cloud-native computing, which describes the development of software that is explicitly designed to be executed in cloud environments. Distributed applications, such as microservices and serverless functions, are an important component of cloud-native computing.

Even if cloud computing considerably simplifies the operation and scalability of modern applications, the interaction with the infrastructure (e.g., servers, networks, load balancers), for example, during installation or updates, remains challenging and requires expert knowledge. Efficient use of the infrastructure, even under dynamic workloads, is particularly challenging. The serverless computing paradigm [KHA⁺23] aims to free customers from resource provisioning and management and places these tasks in the hands of the cloud provider. Users then only pay for the resources they actually use and do not have to reserve them in advance. Serverless computing supports the DevOps philosophy and simplifies deploying developed code into production. Current commercial offerings for serverless computing (such as AWS Lambda², Google

¹<https://datacenters.microsoft.com>

²<https://aws.amazon.com/lambda>

Cloud Run,³ and Azure Functions⁴) primarily support the deployment of functions or containers, which we introduce in more detail in the following section.

2.2 Container Virtualization

One of the main enablers of cloud computing and the efficient use of physical resources is the concept of virtualization. It creates an additional layer of abstraction between users and physical resources. The basic advantages of virtualization are the more efficient use of resources and increased security through limited access to physical machines. Virtual machines are a classic implementation for virtualization. They are managed by a hypervisor, which translates commands and then forwards them to physical resources (such as the CPU). Each virtual machine contains a complete operating system, which increases the size and makes startup times comparatively slow. With the use case of potentially deploying many (comparatively slim) microservices or even serverless functions, virtual machines are associated with too much overhead and little flexibility. Container virtualization is a more lightweight alternative to virtual machines and is optimized for hosting one or a few slim applications per instance.

Figure 2.2 shows the conceptual differences between virtual machines and containers. Several containers share a common operating system kernel. Individual containers are created using host operating system (OS) tools as isolated environments containing only language runtimes, libraries, and the application logic. In the area of microservices, this means that containers contain the application logic and all dependencies. This contrasts virtual machines, where each instance has an isolated operating system. At the same time, this also means that containers must always use the same kernel of the host OS, while virtual machines have fewer restrictions. Likewise, the isolation of physical resources is typically considered to be better for virtual machines than for containers [SCST16]. In public cloud environments, there is a combination of both types of virtualization. Here, containers are usually deployed on virtual machines.

In recent years, the Open Container Initiative (OCI), a project of the Linux Foundation, has developed open standards for container virtualization. In the following, we look closer at Docker containers under Linux operating systems. An alternative with a significant number of users is Podman,⁵ which this thesis will not examine in more detail. Both Docker and Podman use OCI standards. The Docker technology stack consists of different layers. The top layer consists of Docker itself, comprising a command-line interface and a daemon process. The Docker daemon interacts directly with `containerd`,⁶ a standalone container daemon responsible for container lifecycle management.

`containerd` is compatible with all OCI-compliant container runtimes. Container runtimes are the actual environments in which containers are executed. OCI-complaint

³<https://cloud.google.com/run>

⁴<https://azure.microsoft.com/en-us/products/functions>

⁵<https://podman.io>

⁶<https://containerd.io>

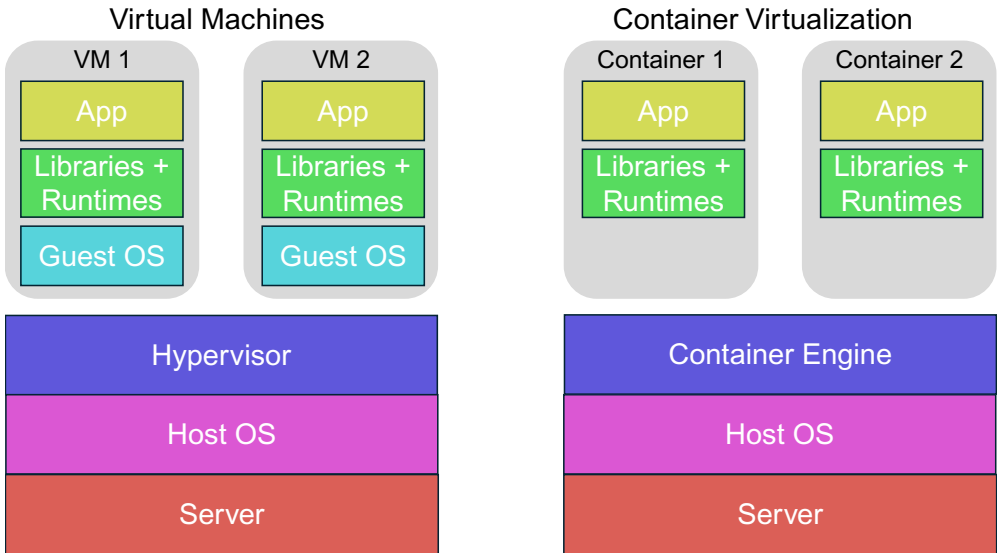


Figure 2.2: Virtual machines vs. containers.

runtimes include `runc`,⁷ `runcsc`,⁸ and `kata-runtime`.⁹ `runc` is the default runtime and the most commonly used, while the mentioned alternatives are more security-focused and promise increased isolation.^{8,9} One runtime process is started per container, which creates the actual process namespace and starts the container. The exact lifecycle of a container is defined by an OCI standard.¹⁰ Various Linux kernel features work together to execute a container, including namespaces and control groups. These features also allow resource limits (e.g., for CPU or RAM) to be defined for containers. CPU limits are specified as absolute floating-point numbers. A CPU limit of n means that within a CPU scheduling interval of length i (usually 100 ms), the container gets $n \cdot i$ computing time. Note that the container is normally not directly assigned to a core, meaning each container can theoretically use all of the machine's cores. The CPU limit only determines the computing time per CPU scheduling interval. A CPU limit of 1.5 with a scheduling interval of 100 ms means that a container may consume 150 ms of computing time per 100 ms on all cores together before throttling occurs. Resource constraints are useful to provide performance guarantees and avoid interference.

The question remains as to how exactly the content of a container is defined and which process is executed in the container. This is where container images come into play. Container images are templates for containers and organize their required files. As mentioned above, these include language runtimes, libraries, and application executables. Docker and other tools can be used to build images and make them

⁷<https://github.com/opencontainers/runc>

⁸<https://github.com/google/gvisor>

⁹<https://katacontainers.io>

¹⁰<https://github.com/opencontainers/runtime-spec>

available in so-called registries. When starting a container, the name of the image from which the container will be created is a required parameter. `containerd` checks whether the image from the host system is already stored and, if not, initiates a download from a registry (e.g., Docker Hub). To start a container, `containerd` passes a so-called OCI bundle created from the container image to the container runtime.

The OCI image specification¹¹ defines the structure of a container image. Essentially, an image consists of metadata, the file system, and a configuration. The file system is organized in layers, increasing efficiency in container development. Most application developers use pre-built images (e.g., with installed libraries and language runtimes) as a basis and then copy their application executables. Each build operation creates a new layer in the file system. The image configuration determines which command (often a shell command) is executed at the container start. When the execution of this command is completed, the entire container is terminated. The image configuration can contain further information, such as open network ports. Docker allows interaction with container images, for example, the image configuration can be inspected. When starting a container, the default configuration set in the container image, including the start command, can also be overwritten by the user.

2.3 Container Orchestration Frameworks

Containers are the dominant form of virtualization for modern cloud applications such as microservices and serverless functions. As already mentioned, they consist of many microservices that scale individually and typically require one container per instance. This results in a large number of containers that need to be managed at runtime. In addition, not all containers can usually be deployed on one host, which requires solutions for networking, load balancing, placement, and more. These and other tasks are handled by container orchestration (CO) frameworks. From a technical point of view, CO frameworks represent an abstraction layer for accessing a cluster capable of running containerized applications. Figure 2.3 schematically shows the main components of a container cluster. Usually, a control plane (e.g., a set of controller nodes) interacts with several other nodes (worker nodes) via a node agent. The worker nodes run a container engine (e.g., `containerd`). The container engine uses kernel functions like control groups or other low-level software to allocate resources.

Numerous orchestration frameworks were used in the mid-2010s. By the beginning of 2025, Kubernetes¹² clearly dominates. It was initially developed by Google and then donated to the Cloud-Native Computing Foundation, a Linux Foundation project. There are numerous variations of Kubernetes. For example, `k0s`,¹³ `k3s`,¹⁴ and `Microk8s`¹⁵ promise to be more lightweight and compatible with resource-constrained edge computing environments. Commercial distributions such as RedHat OpenShift¹⁶

¹¹<https://github.com/opencontainers/image-spec>

¹²<https://kubernetes.io>

¹³<https://k0sproject.io>

¹⁴<https://k3s.io>

¹⁵<https://microk8s.io>

¹⁶<https://www.redhat.com/en/technologies/cloud-computing/openshift>

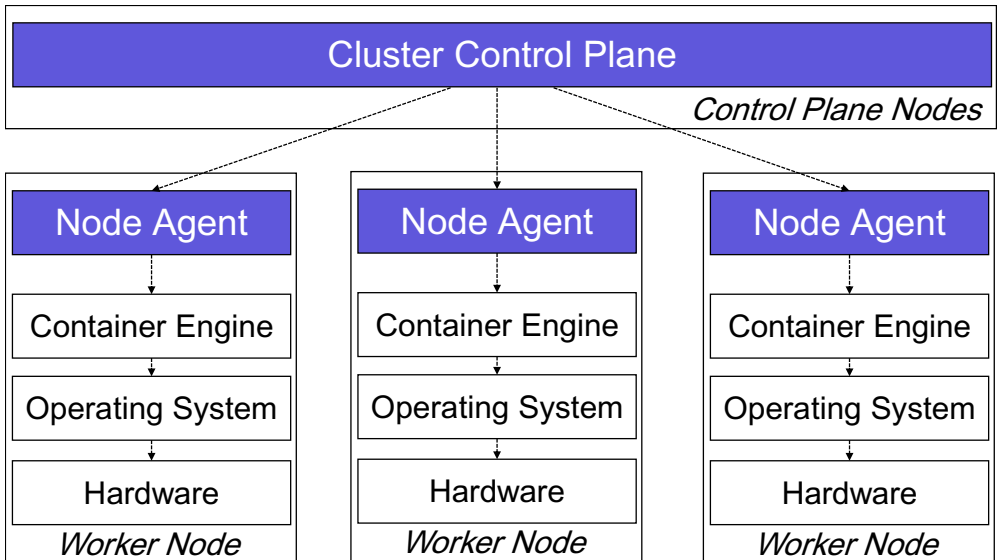


Figure 2.3: Components of a container cluster with container orchestration (blue).

or VMware Tanzu¹⁷ also exist. Kubernetes can also be deployed as a managed service in numerous public cloud environments such as AWS, Google Cloud, or Microsoft Azure. Some serverless platforms, such as Knative,¹⁸ are also based on Kubernetes.

Despite the dominant market position of Kubernetes, there are some alternatives that often offer reduced functionality and a significantly smaller user base. One example is Nomad.¹⁹ Nomad supports both containerized and non-containerized applications and is developed by HashiCorp, a company known for the infrastructure-as-code tool Terraform. Another orchestration framework is CloudFoundry,²⁰ which advertises itself as user-friendly and facilitates the container build process. CloudFoundry is used by various frameworks as a basis (e.g., VMware Tanzu) but has lost parts of its user base in recent years, for example, due to the deprecation in the IBM Cloud in 2022.²¹ Docker-native orchestration is possible with Swarm mode,²² which has replaced the previously developed standalone project, Docker Swarm. Compared to Kubernetes, Docker Swarm mode has significantly fewer functions. There are also some domain-specific alternatives. In the area of high-performance computing, Apptainer,²³

¹⁷<https://www.vmware.com/products/app-platform/tanzu>

¹⁸<https://knative.dev>

¹⁹<https://www.nomadproject.io>

²⁰<https://www.cloudfoundry.org>

²¹<https://www.ibm.com/blog/ibm-cloud-code-engine-migrate-from-cloud-foundry-using-a-toolchain>

²²<https://docs.docker.com/engine/swarm>

²³<https://apptainer.org>

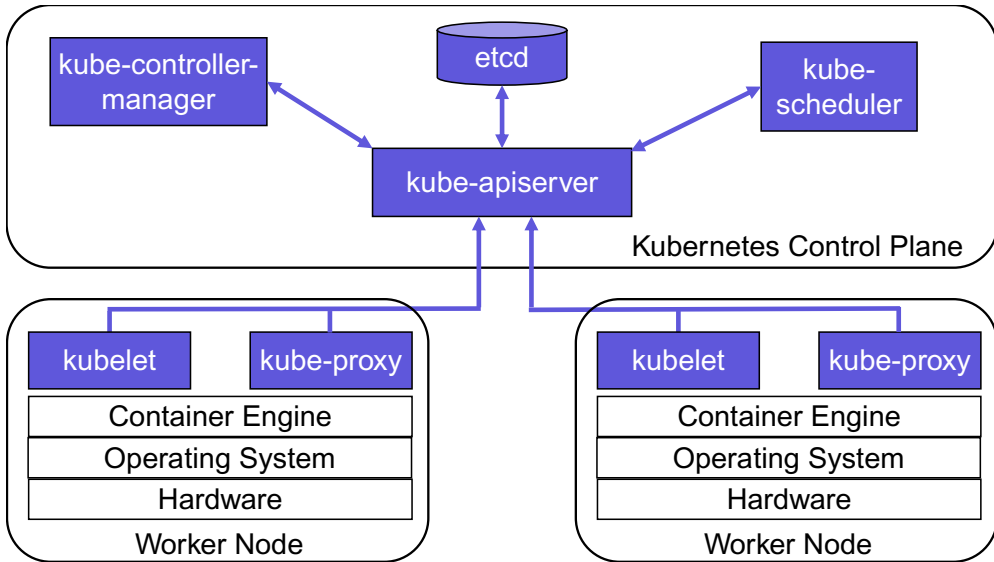


Figure 2.4: Schematic overview of Kubernetes control components.

Shifter,²⁴ Charliecloud,²⁵ and Slurm²⁶ offer possibilities to run container workloads. In the context of this work, we have examined Kubernetes and Nomad in more detail as they are the most important CO frameworks for cloud-native applications. In the following, we present the basic components and features of both frameworks in more detail.

Figure 2.4 shows an overview of the Kubernetes control plane and its components. The Kubernetes control plane consists of four main components: an **etcd** database, the **kube-controller-manager**, the **kube-scheduler** and the **kube-apiserver**. The **etcd** database records the states of the cluster resources. The controller manager executes processes that monitor cluster resources like nodes or workloads. The scheduler assigns containers, which are organized in pods, to worker nodes for execution. Each pod is assigned an IP for networking. Kubernetes uses a declarative interaction concept in which desired states (such as the number of running service instances) are regularly compared with observed states. Different resource types can manage groups of pods (e.g., **Deployments** or **StatefulSets**). Control processes such as health monitoring or autoscaling can be defined for groups of pods.

The central component is the **kube-apiserver**, which handles every communication between the control plane and worker nodes. All worker nodes host a node agent, the **kubelet**, and a networking proxy, the **kube-proxy**. The **kube-apiserver** is based on the Kubernetes API and serves endpoints for every resource type in the cluster (e.g.,

²⁴<https://github.com/NERSC/shifter>

²⁵<https://hpc.github.io/charliecloud>

²⁶<https://slurm.schedmd.com/overview.html>

nodes and pods). When a consumer (e.g., the `kube-scheduler`) requests information about a resource type in the cluster, it queries the corresponding endpoint.

Kubernetes uses a “list-then-watch” principle to distribute information to all interested consumers. After starting a Kubernetes component, it requests the latest list of selected resources with a *list request*. This list contains a resource version, which is later used as a reference to this list. The consumer stores this list in an internal cache. Then, the consumer sends a *watch request* with the resource version of its cached list as a query parameter. This request establishes an HTTP streaming connection between the `kube-apiserver` and the consumer. A watch event is emitted if a change to subscribed resources happens (e.g., a new node is added to the cluster). The watch event has a type (*added*, *modified*, or *deleted*) and a representation of the affected resource. The consumer receives this event, modifies its cache, and performs an action if necessary. We refer to the official documentation for more information about the Kubernetes API.²⁷

Kubernetes allows a high degree of customization for different environments. It uses generic interfaces for essential tasks. The container runtime interface (CRI) describes the communication protocol between the Kubernetes node agent and the container runtime. The standard solution for container runtimes is the technology stack introduced in Section 2.2 with `runc` as the low-level container runtime. However, the top layer (Docker) is not used; instead, Kubernetes interacts directly with `containerd`. Other central interfaces in Kubernetes are the Container Network Interface (CNI) and the Container Storage Interface (CSI). CNI plugins enable efficient communication between pods in the cluster. Widely used implementations include Flannel²⁸ and Calico.²⁹ The container storage interface controls how persistent storage is provided for containers. Widely used implementations here include Longhorn³⁰ and OpenEBS.³¹

Our second framework under test is Nomad. The equivalents to pods in Kubernetes are so-called tasks. A set of tasks can be managed with task groups. Analogously to Kubernetes, activities like health monitoring and readiness probes can be configured for task groups. In contrast to Kubernetes, Nomad’s core is a single binary that can be executed either in a server or client mode. Multiple nodes that run the Nomad binary in server mode form the cluster control plane and administer a production Nomad cluster. The cluster state is maintained with a distributed consensus protocol. Controller processes fulfill tasks that support the lifecycle of managed tasks like health checks. Similar to Kubernetes, Nomad servers offer an API to interact with the cluster. Nomad supports different technology stacks for containers, like `containerd` and Podman, but also non-containerized workloads. All runtimes are abstracted with the concept of task drivers. Nomad uses Consul³² to enable service networking. Nomad also supports CSI plugins for persistent volumes. Compared to Kubernetes, Nomad claims more simplicity and high scalability, supporting clusters of over 10,000 managed

²⁷<https://kubernetes.io/docs/reference>

²⁸<https://github.com/flannel-io/flannel>

²⁹<https://www.tigera.io/project-calico>

³⁰<https://longhorn.io>

³¹<https://openebs.io>

³²<https://www.consul.io>

nodes.³³ A comprehensive comparison between Kubernetes and Nomad is given in a comprehensive blog post.³⁴

2.4 Autoscaling

Autoscaling is one of the essential tasks of container orchestration frameworks, but also for cloud resource managers in a broader sense. Autoscaling refers to the process of being able to change provisioned resources according to "time-varying environmental conditions" [CBY18]. These time-varying conditions are primarily associated with changes in workload. Workloads can vary in intensity (e.g., measured by incoming requests per second) and/or structure over time. Intensity changes occur in many web-based applications, for example, due to different numbers of users during the day and night or on different days of the week. Structural changes can be changes in the types of requests or request parameters. Types of requests in the area of microservice applications can, for example, be requests to different endpoints of a microservice. An example could be the COVID-19 pandemic, in which a travel agency has to process a disproportionately high number of cancellations while bookings dominate at normal times. Request parameters can also influence an application's performance and thus potentially autoscaling [EWvKK18]. Consider a train booking platform as an example where a request for a long-distance journey with many transfers consumes more resources than a regional connection. A workload burst refers to an intensity or structural workload change that occurs suddenly and in a limited period of time. One example is a video platform that releases a new season of a series at a certain time, which many customers request instantly.

There are different dimensions of how to change resources when performing autoscaling. Horizontal scaling is the concept of replicating existing resources to increase the system's capacity or deleting replicas to reduce the capacity. In the area of microservices and containers, new container/service instances are started or terminated. One assumption of horizontal scaling is that the load can be adequately distributed between replicas and that each replica can process every user request. Even if statelessness is advantageous, horizontal scaling does not necessarily require strict statelessness of the replicas. For example, sessions can also be supported. Horizontal scaling is dominant in microservice applications as these often consume compute resources (e.g., CPU) [SGJN19]. The advantage of horizontal scaling here is that the limits of individual servers (e.g., in terms of CPU or network capacity) can be circumvented by distributing the workload among different physical or virtual servers.

Vertical scaling, on the other hand, attempts to allocate more resources to existing replicas to increase their throughput. For containers or virtual machines, this means increasing or decreasing their CPU, memory, or other resource limits. The assumption with vertical scaling is that this is technically possible and that the physical machines have sufficient resources, especially when scaling up. Vertical scaling is often used for stateful applications like databases [AEADE11]. Here, more parallel operations can be

³³<https://traefik.io/glossary/hashicorp-nomad-101>

³⁴<https://www.hashicorp.com/blog/a-kubernetes-user-s-guide-to-hashicorp-nomad>

carried out by increasing the CPU limit, or more data can be loaded/cached into the main memory by increasing its maximum size. Horizontal scaling is often complicated with such applications because the state has to be synchronized between replicas.

Another form of autoscaling has become increasingly popular in recent years, especially in the public cloud. With cluster autoscaling, the number of nodes in a container cluster can be changed. Depending on the configuration, nodes with different resources can be added or deleted. Therefore, cluster autoscaling can formally be seen as a mixture of horizontal and vertical scaling. Cluster autoscaling assumes that nodes can be integrated into the cluster and removed again. In public clouds, this is possible through calls to a management API that can start new virtual machines with pre-installed images and start scripts. Cluster autoscaling is worth mentioning because it provides an additional scaling dimension, especially with container orchestration frameworks. In summary, to implement scaling, replicas of an application (horizontal), their resource limits (vertical), or even the number of nodes in the cluster (cluster autoscaling) might be changed. Recent work proposes approaches that change all scaling dimensions simultaneously [SGJN19], resulting in multi-dimensional approaches for autoscaling.

Kubernetes and many derivatives, such as OpenShift, support all three introduced scaling dimensions. In Kubernetes, the Horizontal Pod Autoscaler (HPA), the Vertical Pod Autoscaler (VPA), and the `cluster-autoscaler` are actively developed components. A multidimensional pod autoscaler is currently in the development and testing phase as an enhancement proposal.³⁵ Nomad also supports all three introduced dimensions through the horizontal application autoscaler, horizontal cluster autoscaler, and dynamic application sizing. Other frameworks only support some dimensions (e.g., only horizontal and vertical in CloudFoundry) or have no integrated autoscaling at all (e.g., Docker Swarm mode) and therefore require third-party or custom plugins.

The fundamental assumption of autoscaling is that increasing the resources actually increases the application’s capacity. This requires an accurate configuration of the application. For example, increasing resource limits (such as CPU or memory) will not affect the capacity of a microservice application with a limited thread pool size or only to a limited extent. Furthermore, autoscaling cannot arbitrarily minimize the response time of an application. Only a request’s queueing/waiting time can be reduced by using more resources; autoscaling does not reduce the actual processing time. Autoscaling aims to maintain a defined quality of service (QoS) level under varying load while minimizing used resources (and thus real costs). There is much preliminary work on this topic, and different categories of autoscalers exist. For example, reactive autoscalers try to react to current changes in the environment, while proactive autoscalers try to predict future system states from historical data. A more detailed analysis of previous work can be found in Section 3.3.

³⁵<https://github.com/kubernetes/autoscaler/blob/master/multidimensional-pod-autoscaler/AEP.md>

2.5 Software Performance Benchmarking

This section introduces some of the main concepts in the field of benchmarking, with a special focus on software performance and experimental design. According to Kounev et al. [KLvK20], a benchmark is "a tool coupled with a methodology for the evaluation and comparison of systems or components with respect to certain properties, such as performance, reliability or safety." This work has a particular focus on performance, although we interpret this term very broadly. Most of the studies in this thesis evaluate the performance of a system using a time measure (e.g., container start time or request response time) and/or a measure of resource consumption (e.g., number of active servers, operating costs). Von Kistowski et al. [vKAH⁺15] define five quality criteria for benchmarks. According to these criteria, a benchmark should deliver results that are interesting for the users and correspond to the intended purpose of the benchmark (relevance). Benchmarks should provide consistent (reproducibility) and trustworthy (verifiability) results. Last but not least, benchmarks should not favor certain systems under test (fairness), and users should be able to run the benchmark in different test environments with reasonable effort (ease of use).

Measurements are essential in benchmarking to quantify the behavior of the system under test (e.g., concerning its performance). The mapping of specific events or objects to numerical values is called a measure, while a metric is a derived value from one or more measures [KLvK20]. When conducting experiments, different results usually occur for the same measurement setup when repeated. This variability is due to two types of measurement errors: systematic errors and random errors [Tay97]. Systematic errors come from avoidable or unavoidable errors in the measurement process. For example, when measuring timestamps from different servers, there may be a slight time shift between the servers' clocks. Ideally, these errors can be determined and compensated for, but this is not always the case. The same starting conditions must be established for each measurement repetition. Random errors are caused by random processes in the system under test (e.g., packet collisions in network traffic). Random errors can never be completely eliminated but can be statistically estimated with repeated measurements. It is usual to specify the mean value of experiment results together with the standard deviation or confidence intervals to quantify uncertainties.

Benchmarking and performance measurements often aim to compare different systems under test. For example, the configuration of a system or a component can be varied, or two completely independent systems that serve the same purpose can be compared. The central question is then how the manipulated factors (input variables) influence one or more metrics of interest (response variables). One technique to obtain statistical information about the influence of the input variables is Analysis of Variance (ANOVA) [Fis70]. ANOVA allows for detailed statements about influencing factors, but a large number of measurements are necessary, especially if several factors need to be analyzed. For example, we want to determine the throughput of an endpoint of a Java microservice. We decide on 100 measurement repetitions to consider the measurements' variability. If we would like to compare two different values for the heap size of the Java Virtual Machine (e.g., 2 GB and 8 GB), $2 \cdot 100 = 200$ measurements are necessary for full-factorial ANOVA. If we also want to compare two

Run	Factor 1	Factor 2	Factor 3	Response
1	+1	+1	+1	267
2	+1	-1	-1	135
3	-1	+1	-1	282
4	-1	-1	+1	156
Effect	-36	258	6	

Table 2.1: Example for a Plackett-Burman design with three factors.

different container CPU limits and two persistent storage configurations, the number of measurements needed increases to $2 \cdot 2 \cdot 2 \cdot 100 = 800$.

Choosing a full-factorial experiment design is not always possible due to time or cost constraints. Fractional factorial designs attempt to reduce the number of measurements required with as few restrictions as possible on the informative value of the results. One of the most widely used fractional factorial designs is the Plackett-Burman experiment design [PB46]. If the influence of $m \in \mathbb{N}$ factors is to be determined from a response variable, the Plackett-Burman design requires $n \in \mathbb{N}$ repetitions, where $n > m$ and $n \bmod 4 = 0$. For the above example with the Java microservice, only $4 \cdot 100 = 400$ measurements are required for analyzing three factors in the Plackett-Burman design instead of 800 measurements with full-factorial ANOVA.

The Plackett-Burman design is based on testing two alternatives of each factor under investigation. The alternatives are symbolically mapped to a low value (-1) and a high value (+1). In our example, we could code the heap size of the Java virtual machine as low for 2 GB and as high for 8 GB. If we have coded m factors with two alternatives in each case, we can determine the configurations to be tested. There are templates in the literature for the so-called design matrices. Table 2.1 shows the design matrix for $m = 3$ factors. Four runs need to be performed. The response variable is assigned to each run as the result. In Table 2.1, we have entered random values for the response variable. Using the so-called effect, we can now determine the influence of the individual factors. To calculate the effect, each value in a column is multiplied with the associated response and then summed up across all values in the column. This means the effect of Factor 1 from Table 2.1 results in: $(+1) \cdot 267 + (+1) \cdot 135 + (-1) \cdot 282 + (-1) \cdot 156 = -36$. The sign of the result does not play a role in the interpretation; only the absolute value is important. In our example, Factor 2 is the most important influencing factor, as the absolute value of its effect is the largest among all tested factors. In this work, we calculate the importance \bar{E}_i of a factor i based on the effect values of all considered factors. It normalizes E_i , the effect of factor i , by the sum of all effect values and thus gets a value between 0 and 1:

$$\bar{E}_i = \frac{|E_i|}{\sum_{j=1}^m |E_j|}. \quad (2.1)$$

In our example, the value for Factor 2 is $258/(36 + 258 + 6) = 0.86$. The Plackett-Burman design allows statements about the main effects of input variables but not about interactions between the input variables. Extensions like the Plackett-Burman design with foldover [KLvK20] exist but are not used in this work. When designing experiments, the Plackett-Burman design can provide good indications of the main influencing factors due to its lower costs. Depending on the use case, the most impactful factors can then be investigated in more detail in additional experiments.

2.6 Basics of Probability and Random Variables

In this section, we introduce some of the basic principles of probability theory and random variables. The goal of this section is to discuss the concepts that are used in this thesis briefly. For a comprehensive introduction to probability, we refer to established literature [WMMY93]. As described in the previous section, experiments play an important role in software performance benchmarking. Conceptually, an experiment is "a process of which the outcome is uncertain" [KLvK20]. The concrete outcome of an experiment is called an elementary experiment result or a sample ω ; the set of all possible experiment results is called the sample space Ω . The sample space can be finite or infinite. An example of a finite sample space is rolling a dice with the numbers 1 to 6. An example of an infinite sample set is throwing a football and measuring the length of the throw. Even if the length is potentially limited downwards (by 0) and upwards, the number of possible outcomes is not countable because it is an interval on the real numbers.

In the following, we denote a subset of the sample space (i.e., a set of elementary experiment results) as an event A . If we carry out n experiments and count how often an experiment result that belongs to A occurs, we obtain the counter n_A . The probability $P(A)$ is defined as the limit value of the relative frequency n_A/n for $n \rightarrow \infty$, that is, an infinite number of experiments. Some basic properties for probabilities can be derived. Based on the above definition, it follows that $0 \leq P(A) \leq 1$. Furthermore, for k mutually exclusive events A_1, A_2, \dots, A_k , which means that all events cover different elementary experiment results, it holds that $P(A_1 \cup A_2 \dots \cup A_k) = P(A_1) + P(A_2) + \dots + P(A_k)$. From the fact that Ω is the set of all elementary experiment results and hence also the set of all mutually exclusive events, it follows that $P(\Omega) = 1$.

In the following, we consider two events, A and B , that are not necessarily disjoint. Figure 2.5 shows a Venn diagram with two events A and B . The so-called joint probability $P(A \cap B)$ represents the probability that A and B occur simultaneously. Intuitively, it can be calculated in the general case as:

$$P(A \cap B) = P(A) + P(B) - P(A \cup B). \quad (2.2)$$

The conditional probability $P(A|B)$ denotes the probability that A occurs under the condition that B has occurred. Of course, it is only defined if B has a chance to occur ($P(B) > 0$). It can then be calculated by dividing the joint probability $P(A \cap B)$ by the probability of B :

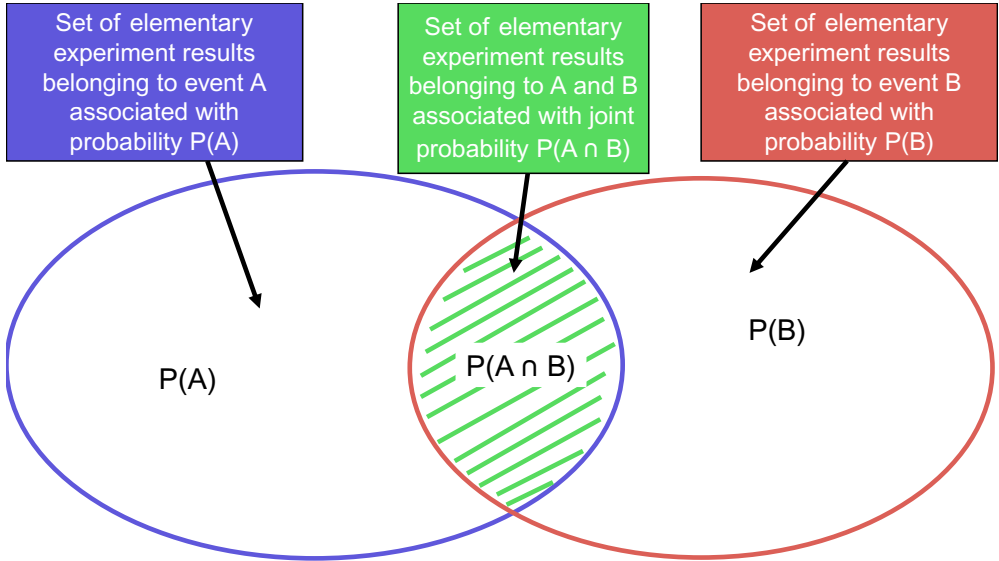


Figure 2.5: Illustration of joint probability.

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \quad (2.3)$$

In the case that $P(A|B) = P(A)$, the two events A and B are statistically independent. In the following, we consider N events A_1, A_2, \dots, A_N of which their union equals Ω . We can calculate the probability of an event A_k given the probabilities of all other events as follows:

$$P(A_k) = P(\Omega) - P(\Omega \setminus A_k) = 1 - \sum_{i=1, i \neq k}^N P(A_i). \quad (2.4)$$

The law of total probability says that the probability of an event B can be calculated given all the conditional probabilities $P(B|A_i)$ and the probabilities of all events A_i . The probability $P(B)$ can be interpreted as a sum of the conditional probabilities weighted by the probability of the event that forms the condition:

$$P(B) = \sum_{i=1}^N P(B|A_i) \cdot P(A_i). \quad (2.5)$$

Random variables build on these basic concepts of events and probabilities. A random variable is a function that maps an elementary experiment result to a real number. We distinguish between continuous and discrete random variables based on the domain of the random variable. Our previous example experiment with the dice roll can be described with a discrete random variable X that can take the values 1

to 6. In general, the function $x(i) = P(X = i)$, which represents the probability that the discrete random variable takes the value i , is known as the probability mass function (PMF). The function $X(i) = P(X \leq i)$, which indicates whether a discrete random variable takes at most the value i , is known as the cumulative distribution function (CDF).

Our example involving the length of a football throw can be described as a continuous random variable A . The CDF $A(t) = P(A \leq t)$ is defined analogously to discrete random variables as the probability that A assumes at most the value t . In contrast to discrete random variables, assigning a probability to a concrete value t does not make sense since A can take an infinite number of values. Nevertheless, the probability density function (PDF) $a(t)$ can be seen as an analogous concept to the PMF for discrete random variables. The PDF is defined as the derivative of the CDF $A(t)$:

$$a(t) = \frac{d}{dt}A(t).$$

In this work, to distinguish between random variables, PMFs/PDFs, and CDFs, we use the following convention: uppercase letters such as A denote random variables, their PMF is represented by $a(x) = P(A = x)$, and the corresponding CDF is defined as $A(x) = P(A \leq x) = \sum_{i=-\infty}^x a(i)$. Non-negative discrete random variables play a special role in this thesis. In the following, we consider selected functions from two non-negative statistically independent discrete random variables X_1 and X_2 . First, we want to determine the PMF of the random variable X , which is the sum of X_1 and X_2 . According to the law of total probability (Equation 2.5), we can express the PMF as:

$$x(i) = P(X_1 + X_2 = i) = \sum_{j=0}^i P(X_1 = i - j | X_2 = j) \cdot P(X_2 = j).$$

Since X_1 and X_2 are statistically independent, it follows that $P(X_1 = i - j | X_2 = j) = P(X_1 = i - j)$. Therefore, we can calculate the PMF of X as:

$$x(i) = \sum_{j=0}^i P(X_1 = i - j) \cdot P(X_2 = j) = \sum_{j=0}^i x_1(i - j) \cdot x_2(j) = x_1(i) * x_2(i). \quad (2.6)$$

The operation shown is also referred to as discrete convolution. In the context of this work, we use the operator $*$ as a shorthand notation. Analogous to the sum, we also want to consider the difference between two non-negative statistically independent discrete random variables. The derivation of the PMF of a random variable $X = X_1 - X_2$ follows the same idea as the sum:

$$\begin{aligned}
x(i) &= \sum_{j=0}^i P(X_1 = i + j | X_2 = j) \cdot P(X_2 = j) \\
&= \sum_{j=0}^i x_1(i + j) \cdot x_2(j) \\
&= x_1(i) * x_2(-i).
\end{aligned} \tag{2.7}$$

The notation $x_1(i) * x_2(-i)$ emphasizes the analogy to discrete convolution and is used analogously as a shorthand notation as in standard literature [TGH21]. Note that the difference between two non-negative random variables can also take negative values. Negative values are not meaningful for many random variables (such as the number of requests in a system). Therefore, we introduce the so-called left-sided sweep operator π_0 [TGH21], which accumulates the probability of negative values of a random variable X at 0:

$$\pi_0(x(i)) = \begin{cases} x(i) & i > 0 \\ \sum_{j=-\infty}^0 x(j) & i = 0 \\ 0 & i < 0 \end{cases} \tag{2.8}$$

Various distributions for random variables occur in the context of this thesis. Table 2.2 introduces the CDFs, the PMFs (for discrete distributions), and PDFs (for continuous distributions) as well as possible abbreviations of the distributions used in this thesis.

2.7 Queueing Networks and Stochastic State Processes

Queueing networks are a common model type in the area of computer systems [Rob00]. In the context of modern cloud applications, metrics such as response time or CPU utilization can be approximated analytically. Figure 2.6 shows the schematic representation of a simple queueing network. Consider this model as a simple representation of a microservice instance. The first important component is the arrival process. In our example, this process would describe the arriving user requests at the microservice instance. A parameter of the arrival process is the rate λ , which describes how many arrivals arrive on average within a time interval. These arrivals are placed in a queue/buffer at the microservice instance (e.g., implemented by the web server framework). The third component of the queueing model is the service process. One parameter of the service process is the service rate μ , which describes how many requests can be processed on average per time unit. For example, it can model a CPU that processes the workloads associated with user requests. The requests get served one after the other under the assumption that the buffer works based on the principle of "first come, first serve." After a request gets served, it leaves the system. There are

Name	Discrete Distributions		
	PMF	CDF	Parameters/Conditions
Deterministic	$x_k(i) = \delta_k(i) = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$	$X_k(i) = \begin{cases} 1 & i \geq k \\ 0 & i < k \end{cases}$	k - Only value that X can take
Uniform	$x(i) = \text{unif}(a, b) = \frac{1}{n}$	$X(i) = \{j : j \leq i\} \cdot \frac{1}{n}$	$a \in \mathbb{N}$ - lower bound $b \in \mathbb{N}, b > a$ - upper bound $n = b - a + 1$
Geometric	$x(i) = (1 - p)^{i-1} p$	$X(i) = 1 - (1 - p)^{[i]}$	$0 < p \leq 1$ - success probability $i \in \mathbb{N}, i \geq 1$
Poisson	$x(i) = \frac{\lambda^i e^{-\lambda}}{i!}$	$X(i) = e^{-\lambda} \sum_{j=0}^{[i]} \frac{\lambda^j}{j!}$	λ - Mean $i \in \mathbb{N}$
Continuous Distributions			
	PDF	CDF	Parameters/Conditions
Normal	$a(t) = \frac{1}{\sqrt{2\pi}\sigma^2} \exp\left(-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2\right)$	$A(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^t e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} dx$	μ - Mean σ - Standard deviation
Negative Exponential	$a(t) = \text{Exp}(\lambda) = \lambda e^{-\lambda t}$	$A(t) = 1 - e^{-\lambda t}$	λ - Rate

Table 2.2: Selected discrete and continuous distributions.

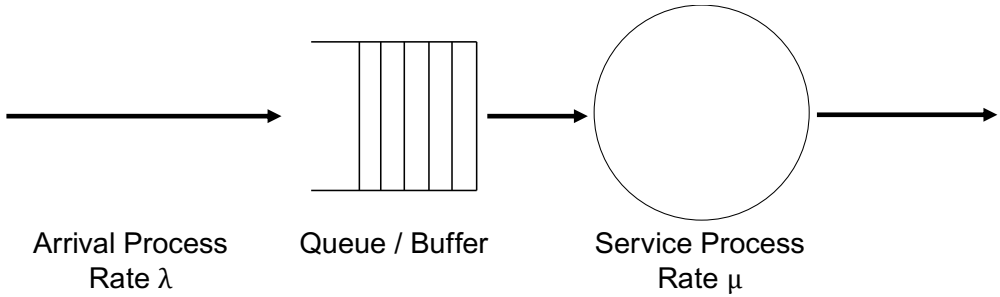


Figure 2.6: Visualization of a simple queueing network.

several more advanced queueing models (e.g., with branches or limited queue lengths) discussed in common literature [TGH21].

The concept of stochastic state processes is important for analyzing queueing networks. In technical systems, we often consider measured variables that change over time. These metrics usually describe the state of the analyzed system. Our measurements occur at specific points in time t_i at which the system's state becomes known to us. If we describe these measured variables as random variables X and consider tuples of the form $\{X(t_i), t_i\}$ for different points in time t_i , we obtain a stochastic state process. Like the random variables, we can describe both the state X and the time t as continuous or discrete. A group of stochastic state processes is particularly often used for the performance analysis of computer systems: the so-called Markov processes. A stochastic state process is called a Markov process if it holds that the probability that the system has the state x_{n+1} at time t_{n+1} depends only on the state x_n of the system at time t_n :

$$\begin{aligned}
 P(X(t_{n+1}) = x_{n+1} | X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \dots, X(t_0) = x_0) \\
 = P(X(t_{n+1}) = x_{n+1} | X(t_n) = x_n) \quad (2.9)
 \end{aligned}$$

This characteristic greatly facilitates the analysis of technical systems and allows future system states to be calculated iteratively. Given the system state x_0 at a time t_0 and the so-called transition probabilities $p_{ij} = P(X(t_{n+1}) = j | X(t_n) = i)$, we can iteratively compute future system states. In the case that $x_{n+1} = x_n$, we determine that the system has reached a stationary state. A form of modeling that makes use of this property and iterative solution method is discrete-time analysis. Here, a stochastic state process with discrete (not necessarily equidistant) time points is considered. By skillfully selecting the so-called embedding times, the process can then be treated as a Markov process, and formulas for the transition probabilities can be found. This constitutes a discrete-time Markov chain.

Markov chains play an important role in modeling an entire queueing network and calculating metrics such as queue length over time. Subprocesses (e.g., the arrival process) are modeled as point processes. Point processes are sequences of random points in time on the real time axis (see Figure 2.7). In the context of arrival processes,

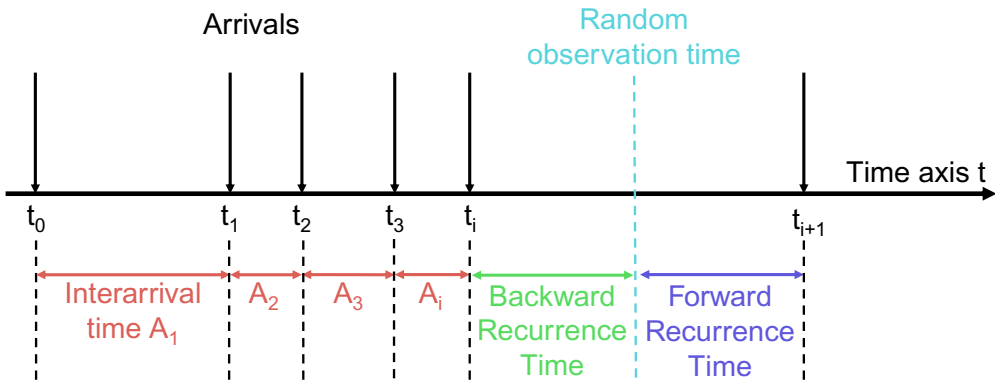


Figure 2.7: Point process and describing random variables (adopted from [TGH21]).

these can be the arrival times of user requests. The time between two user requests is called inter-arrival time and is described by the random variable A . If an observer looks at the system at a random point in time, the time to the next arrival is the forward recurrence time, and the time elapsed since the last arrival is the backward recurrence time.

Point processes in which the inter-arrival times are independently and identically distributed (iid) are called renewal processes. Poisson processes are a group of renewal processes in which the inter-arrival times follow a negative exponential distribution ($A \sim \text{Exp}(\lambda)$). The defining feature of a Poisson process is that the interarrival times and forward recurrence times follow the same distribution. Finally, we consider an arrival process where requests are sent by a very large number of users. Inter-arrival times of one user's requests represent a renewal process (not necessarily a Poisson process). The arrival process at the system is then the superposition of all user requests, that is, the superposition of many renewal processes. The Palm-Khinchine theorem [HS04] is applicable in the case that the total load on the system is finite and no component process (user) dominates the entire process (i.e., its rate is significantly larger than the rates of the other users). It states that the superimposed process behaves (if the number of subprocesses runs towards infinity) asymptotically like a Poisson process whose rate is the sum of all rates of the subprocesses. Numerically, in many scenarios, this is a good approximation already for a double-digit number of subprocesses [TGH21].

Chapter 3

State of the Art

This chapter discusses research works and tools related to our contributions. We summarize prior achievements, highlight differences in our work, and underline its novelty. We split the related works into four parts that align with the contributions of this thesis. First, we take a close look at comparative studies and benchmarking approaches for container orchestration frameworks in Section 3.1. Section 3.2 summarizes empirical studies on the performance of container virtualization with a focus on comparing different technology stacks and container runtimes. Section 3.3 gives an overview of the state of the art in autoscaling of modern cloud applications. It also discusses approaches following a decentralized design paradigm similar to this work. Last, Section 3.4 summarizes existing simulation and modeling approaches for container orchestration frameworks.

3.1 Comparison and Benchmarking of Container Orchestration Frameworks

Container orchestration frameworks are the engines of modern cloud environments. They act as an abstraction layer for access to a computing cluster and can have different roles. In cloud-native computing, developers interact directly with container orchestration frameworks and use them for container deployment, maintenance, storage, and more. In serverless computing, they are the backbone of serverless platforms, responsible for critical tasks such as container provisioning, placement, and scaling. In either use case, they fulfill multiple performance-critical tasks. From a performance engineering perspective, it is therefore essential to evaluate frameworks like Kubernetes because they significantly impact the performance of cloud applications, as shown in several studies [MTK22, PCB⁺19, TBVL⁺18, TVLLJ19, VBT⁺19]. This section summarizes prior works in the area of performance analysis and benchmarking of container orchestration frameworks. It is split into three parts: feature-based comparisons, empirical studies, and benchmarking approaches.

Feature-based comparisons of container orchestration frameworks. Rodriguez and Buyya [RB19] present a taxonomy of cluster orchestration systems. They introduce a systematization of the main characteristics of a cluster orchestration system, including the application model, scheduling management, infrastructure, resource management, system objective, security, network, and performance isolation. They compare several

orchestration frameworks that are able to run container workloads, such as Kubernetes and Docker Swarm, and classify their objectives and capabilities. Truyen et al. [TVLP⁺19] conduct a comprehensive feature comparison study between Docker Swarm, Mesos, Kubernetes, and other orchestration frameworks. Alamoush and Eichelberger [AE24] conduct a feature analysis for different orchestration frameworks, including several Kubernetes distributions, Docker Swarm, Mesos, Marathon, DC/OS, and Nomad. The goal of this work is to evaluate the applicability of the analyzed frameworks in an Industry 4.0/Internet of Things environment. All of the mentioned works aim to provide guidance for practitioners to select an appropriate framework for a given use case based on the frameworks' features. In contrast, our contribution defines a set of essential features for container orchestration frameworks and benchmarks the performance using associated metrics for every feature.

Empirical studies on container orchestration performance. Several studies have made empirical comparisons between individual orchestration frameworks with the emerging trend of containers and the release of Docker and Kubernetes. Pan et al. [PCB⁺19] compare the performance properties of Kubernetes and determine its influence on the execution time of cloud applications. Baresi and Quattrocchi [BQ20] measure execution times for different operations, such as cluster scaling actions and container start and removal times in different public cloud providers and Kubernetes environments. Mao et al. [MFG⁺20] conduct several experiments to compare performance metrics for different workload types, including deep learning jobs, in Docker Swarm and Kubernetes environments. The reported performance metrics include CPU usage, job completion time, and placement metrics.

Many researchers addressed performance comparisons of container orchestration frameworks in resource-constrained environments. Fayos-Jordan et al. [FJFCSG⁺20] compare Kubernetes and Docker Swarm for Internet of Things use cases. Usman et al. [UFB24] present benchmarking studies comparing the lightweight Kubernetes distributions Microk8s, k0s, k3s, and MicroShift. They investigate different metrics for the systems under test, including resource usage, control plane, and data plane performance. The authors conclude that every tested distribution has its unique strengths and weaknesses. The variety of edge computing scenarios requires a thorough consideration of the factors influencing the decision to choose a container orchestration platform. Similar studies have been performed by Koziolok and Eskandani [KE23] and Kudla et al. [KFW⁺22]. The latter focuses on a military scenario.

Benchmarking and performance metrics for container orchestration frameworks. There are a number of related works that combine dedicated benchmarking approaches with empirical case studies on the performance of container orchestration frameworks. Multiple works [BW21, TSZN21] investigate performance and resource consumption metrics for different Kubernetes-like frameworks. **k-bench**¹ is a framework for assessing the performance of Kubernetes' control and data plane with support for pod start, networking, and I/O metrics. **kube-burner**² is an open-source tool for Kubernetes performance and scale testing. Performance metrics that can be measured include pod readiness times and network latencies.

¹<https://github.com/vmware-tanzu/k-bench>

²<https://github.com/kube-burner/kube-burner>

3.1 Comparison and Benchmarking of Container Orchestration Frameworks

Frisbee [NCMB21] is a benchmarking tool for assessing the performance of cloud-native applications running on Kubernetes. The architecture of Frisbee is similar to the architecture of our framework COFFEE, with a controller as the main component that parses user inputs, communicates with the Kubernetes control plane, and collects metrics from deployed containers. However, while COFFEE focuses more on metrics related to container orchestration frameworks with some support for application metrics, Frisbee focuses on the deployed applications. One presented case study focuses on Redis and its performance in different workload scenarios. Another difference to our approach is that COFFEE supports testing other container orchestrators beyond Kubernetes. Singh et al. [SSJ24] use a mix of state-of-the-art resource benchmarks for CPU, storage, and network measurements in Kubernetes clusters. While the focus is on resource usage metrics of the host machines, they also compare the network performance of managed and self-hosted Kubernetes environments. We conduct similar measurements in one of our case studies with COFFEE (see Section 4.5). Nomad has not been investigated much in the scientific literature. Rare examples include the deployment of network services [VMZK21, VMK22].

Another group of prior works provides approaches for assessing the performance of container orchestration frameworks with respect to selected aspects. Al Jawarneh et al. [JBB⁺19] focus on provisioning and availability metrics for Kubernetes, Docker Swarm, Mesos, and Cattle. In the pre-container era, performance analysis approaches for cloud environments addressed, inter alia, provisioning times and request throughputs [CCVK13, IOY⁺11, SHG⁺13]. Bozoki et al. [BSP⁺20] focus on the resilience testing of Kubernetes and especially examine pod restart times. Similar to our work, fault injection is used to force restarts. Kjorveziroski and Filiposka [KF22] compare different deployment options of Kubernetes for serverless applications based on start and response time metrics.

Vayghan et al. [VSTK21] define several metrics for quantifying Kubernetes' performance with regard to failure discovery and recovery. Barletta et al. [BDSCDM24] present a detailed breakdown and performance analysis of Kubernetes' failover mechanism. The failover process is divided into phases, and the length of these phases is analyzed in different scenarios. COFFEE also supports the analysis of failover mechanisms, and we conduct a case study for demonstration in Section 4.6.

Lei et al. [LLJ⁺19] focus on the scalability of Kubernetes clusters. Pod startup times, latencies of API callbacks, and other metrics are analyzed in clusters of varying sizes. Henning and Hasselbring [HH22] focus on scalability benchmarking of cloud-native applications. Therefore, they also provide tooling for Kubernetes and conduct experiments on the Google Cloud Platform. Delnat et al. [DTR⁺18] present K8-scalar, a tool for comparing different autoscaling mechanisms for databases hosted in Kubernetes clusters. Similar to our work, custom, time-changing workloads can be defined. The focus of this tool is on collecting metrics directly related to autoscaling, like request latencies and CPU usage. COFFEE is able to provide more holistic insights into many container orchestration mechanisms and related metrics.

Yu et al. [YKXY19] evaluate load balancing algorithms in the Internet of Things sector for interconnected microservices. They explicitly consider the time to decision, the maximum runtime of the load balancing algorithms, as a performance metric. Amaral

et al. [APC⁺15] evaluate the performance of container-based microservices and target network performance, throughput, and latency. The works of Zeng et al. [ZWDZ17] and Bankston and Guo [BG18] focus on the empirical evaluation of different container networking solutions. SeCoNetBench [OHS19] is a framework for benchmarking secure inter-container networking frameworks. Case studies demonstrate the usage of the tool in experiments involving the Kubernetes CNI plugins Canal, Weave, Cilium, and Contiv. `kube-bench`³ is a framework concerned with the security evaluation of Kubernetes clusters.

CNSBench [MTA⁺21] is a benchmarking tool for cloud-native storage. Case studies include the evaluation of different configurations for OpenEBS and Ceph as persistent storage providers for Kubernetes clusters. Similar to our work, the authors not only evaluate performance metrics for storage operations (like I/O throughput) but also measure provisioning times for pods that require persistent storage. `kubestr`⁴ is another tool that aims to evaluate the performance of different cluster storage options.

In summary, the uniqueness of our contribution compared to related works lies in the combination of two essential characteristics. First, we support collecting numerous performance metrics related to different container orchestration tasks (e.g., failure recovery and container networking). This stands in contrast to many of the mentioned papers, which focus on single or few tasks. Second, we support the evaluation of different container orchestration frameworks and technology stacks. This is enabled by posing minimal assumptions on the system under test.

3.2 Empirical Studies on Container Performance and Optimization

While the previous section focused on performance studies in the area of container orchestration frameworks, we now move one level deeper into the technology stack. We analyze studies on the performance of container virtualization and comparisons with other virtualization technologies. Also, we review studies that compare different container runtimes. A special focus in this work is also on container start times as one particular performance metric. Fast start times are a primary motivation for the use of containers. We review scientific works concerning influencing factors of container start times and the optimization of the start process. Last, we take a wider look at empirical studies concerned with containers. We summarize works that provide large container datasets but are not primarily performance-focused.

Performance comparison of virtualization techniques. Many studies discuss trade-offs between the security and performance aspects of virtualization. Performance is usually quantified by start times and runtime metrics (like throughput). Faster start times have always been a goal for developers of container technologies. Several studies compare container start times and further performance metrics with other virtualization technologies. Tesfatsion et al. [TKT18] examine virtual machine (VM)

³<https://github.com/aquasecurity/kube-bench>

⁴<https://github.com/kastenhq/kubestr>

3.2 Empirical Studies on Container Performance and Optimization

startup times compared to Docker and Linux containers. Many studies conclude that containers have faster start times than traditional VMs [WRK⁺19]. To preserve the conceptual advantages of VMs with respect to isolation and security, researchers also aim to make virtual machines more lightweight [MLS⁺17]. Unikernels are also often compared to container virtualization [GSA⁺18, GSAN⁺22, SFSF19, XFJ16]. The differences in performance here are less clear, but the use cases of unikernels are usually limited to single-process applications.

Another research area looks at start times concerning different container runtimes, often in combination with security and isolation properties. Kumar et al. [KT20] compare `runc`, the default Docker runtime, with `kata-runtime`, a runtime with increased isolation and security mechanisms. Randazzo and Tinirello [RT19] conduct a qualitative comparison between `runc` and `kata-runtime`. The authors see the main advantages of `kata-runtime` in the advanced security model and increased isolation between multiple containers running on the same host. Dordevic et al. [DTLD22] compare performance metrics for Docker and Podman container technology stacks, also in comparison with executing the workloads on the bare-metal host. Their results show that both Docker and Podman have negligible runtime performance overheads compared to execution on the host. Also, they state that Docker has a slight, not significant, performance advantage over Podman. Kaiser et al. [KTK23] compare Docker, Podman, and Singularity as container engines running on ARM-based edge devices. The authors conclude that the tested frameworks have strengths and weaknesses on different workloads. Melo et al. [MGD⁺23] compare different performance metrics of Docker and Podman and conclude that Podman has lower pull and instantiation times, especially for tiny containers.

Espe et al. [EJPG20] compare technology stacks composed of `containerd/cri-o` and `runc/runsc`. They conclude that `cri-o` and `runc` is a good combination for many use cases, while `containerd` and `runc` perform best for I/O-heavy workloads. Viktorsson et al. [VKT20] also analyze `runc`, `runsc`, and `kata-runtime` and conclude that `runsc` and `kata-runtime` while having better isolation capabilities, have longer start times and decreased throughput compared to `runc`. All works find a clear trade-off between performance and increased security. Volpert et al. [VWWD24] present a framework for investigating the isolation capabilities of different container runtimes.

Optimization of container start times. Another group of papers deals with certain characteristics of the nodes used for container deployment and their impact on start times. In general, container start times in different technology stacks can differ significantly. De Velp et al. [dVRS21] find that the used storage driver significantly impacts the start times in particular. This result is also confirmed by Harter et al. [HSL⁺16a], who investigate I/O patterns during startups and propose a specialized storage driver. Tarasov et al. [TRS⁺19] conduct several experiments regarding Docker's storage performance with different storage drivers, devices, and file system layers. They conclude that many dimensions can impact Docker's storage performance and that there is no single configuration that should be favored in all use cases and environments. Lingayat et al. [LBKG18] and Mavridis and Karatza [MK17] quantify overheads introduced by starting containers on virtual machines instead of bare-metal servers. Wei et al. [WMG⁺17] investigate the start times of Kubernetes pods and

show the dependency on the overall system load.

Due to the observed start time variabilities, researchers worked early on identifying optimization potential for start times. Thereby, many optimization approaches focus on specific domains and workloads. One major use case is serverless computing, where so-called cold starts can impact the application performance. Ghorbian and Ghobaei-Arani [GGA24] review optimization techniques for cold start latencies in serverless computing. They distinguish between approaches that aim to reduce the load times of serverless functions and approaches that aim to optimize resource usage. The work of Du et al. [DYX⁺20] contains an overview of startup optimization in serverless computing and ranks different solutions concerning their isolation capabilities and startup speed. Optimization approaches include caching techniques, faster installation of dependencies, and container reuse [OYZ⁺18, QWW⁺20]. Bauer et al. [BGP⁺24] investigate how different container build strategies affect cold starts of serverless scientific applications. Their approach includes different strategies for pre-installing Python packages on containers. Empirical results show significant improvements in terms of cold start latency and storage requirements. They also show that the build strategy does not affect the container start time.

Edge computing is another domain in which researchers are very interested in optimizing container start times. Here, resource-constrained nodes or time-sensitive workloads pose defining challenges. Stahlbock et al. [SWK22] analyze the process of container starts in detail and propose an improvement, especially for resource-constrained embedded devices. They distinguish between two operating modes: normal operation and update operation. Several steps in the normal mode can be skipped, assuming all container instances use the exact same configured images. Civolani et al. [CPB19] modify the container start process of Docker so that the container is started before all image layers have been downloaded. The assumption is that the container is startable from a base image, and other dependent files can be included later in the container. While this shortens the start time, it comes with the prices of increased image size (once all layers have been downloaded) and performance overheads. The latter is caused by the need to wrap system calls to prevent failures for file accesses when the container images are not fully downloaded yet. Ahmed and Pierre [AP18] find that pull times significantly impact edge computing use cases and propose optimization through better download schemes and layer decompressing. Littlely et al. [LAF⁺19] find optimization potential in container registries intending to reduce pull times.

Empirical studies on containers. There are prior empirical studies analyzing large datasets of container images. Zhao et al. [ZTA⁺19] use a Docker Hub dataset to explore the properties of layers, image popularity, compressed and uncompressed image sizes, and what type of files the analyzed containers include. Cito et al. [CSW⁺17] analyze over 70,000 Dockerfiles from GitHub and identify common base images, primary programming languages, and more. Baresi et al. [BQT22] similarly analyze a large number of Docker and Docker Compose files crawled from GitHub to identify common technology stacks for microservice applications. The authors provide empirical data on the use of different database technologies, programming languages, and common base images for microservice applications.

Lin et al. [LNK20] analyze over 3.3 million Docker images gathered from Docker

Hub, GitHub, and BitBucket. Similar to our work, they report and analyze the metadata of the images, like the file system size. Moreover, they lay their focus on smells and antipatterns in Dockerfiles. By analyzing Git repositories linked to the images, they also investigate how Dockerfiles evolve with the application’s source code. Henkel et al. [HBLR20] analyze the quality of around 178,000 Dockerfiles from software repositories. They state several bad practices and outline their consequences. Consequences include short- or long-term problems with the image build, increased attack surfaces, and increased image size. Durieux [Dur24] builds on this work and analyzes more than 10,000 Dockerfiles for bad practices that increase the image size of Docker containers. These contribute to an average increase in the image size of about 48 MB. An increased image size generally leads to an increased pull time. In addition, as we show in this work, image size also contributes to the container start time. Shu et al. [SGE17] analyze more than 350,000 Docker Hub images for security vulnerabilities. The analysis is based on installed packages in the image and linked known vulnerabilities. The ImageJockey framework of Yoshimura et al. [YNC20] enables performance testing of multiple container images.

The uniqueness and novelty of our study lie in the interconnection of a large container dataset with start time measurements. Furthermore, in contrast to related work, we maintain a black-box view of container execution and content, relying only on data extractable from the OCI image manifest of the container. This information is available before starting the container, enabling prediction models that estimate container start times of previously unseen images.

3.3 Autoscaling of Modern Cloud Applications

Autoscaling is an essential functionality of modern container orchestration frameworks and plays an important role in the context of this thesis. In Chapter 6, we identify open challenges for the autoscaling of modern cloud applications. Based on our findings, we present a new decentralized approach to autoscaling that treats autoscaling as a continuous process. In this section, we position our work to existing autoscaling works. In the first part, we examine a broad range of related works from the autoscaling field in general, which means that we also include literature from the pre-container and microservice era. We structure this section into paragraphs that outline the aspects of autoscaling relevant to our approach and the challenges we state later in this thesis. In the second part of this section, we separate our contribution from its closest related works that specifically target decentralized and probabilistic autoscaling. Finally, we analyze the use of Markov chains in modeling and validating autoscaling approaches.

Surveys and taxonomies of autoscaling. Numerous surveys [SGJN19, ADPDM18, QCB18, CBY18] provide an overview and taxonomies of previous works in autoscaling research. There are several categories in which autoscalers can be divided. One of the characteristic points of autoscalers is the kind of infrastructure that should be scaled. Al-Dhuraiibi et al. [ADPDM18] distinguish between solutions for containers and virtual machines. The reason why different autoscaling approaches are necessary here is that containers are more lightweight. Scaling decisions can be realized faster and are less

costly. This opens new possibilities for autoscaling. Another common category is reactive and proactive autoscaling. Reactive autoscalers consider only the current values of scaling metrics, whereas proactive approaches aim to predict future system states. Approaches combining both elements are called hybrid autoscalers [SGJN19].

Autoscalers are also categorized by the scaling dimensions they cover (see Section 2.4). All initially mentioned surveys list approaches that support horizontal, vertical, or multidimensional scaling. Scaling metrics determine the input data for autoscalers. Platform metrics (e.g., CPU or memory utilization) and application metrics (e.g., queue length or response times) are commonly distinguished. The survey of Singh et al. [SGJN19] also includes an overview of approaches evaluated by simulation only or by real experiments. Many techniques have been proposed to answer the question of how autoscalers make scaling decisions based on their input data. The analyzed surveys cover approaches based on thresholds, queueing theory, control theory, reinforcement learning, time series analysis, and optimization models. In these taxonomies, our autoscaler can be categorized as a reactive, horizontal autoscaler. Our reasoning approach and scaling metrics are flexible and depend on the configuration of our approach. In contrast to many reviewed works, we evaluate our autoscaler using modeling, simulation, and experiments. In the following, we focus on selected aspects of autoscaling that are particularly related to our contribution.

Reactive and proactive autoscaling. Proactive autoscaling generally aims to closely match the resource demands by acting ahead of load changes. To correct wrong decisions caused by inaccurate predictions, proactive and reactive scalers are combined. Less than 15%, in total, 15 out of 104 of the papers analyzed by Singh et al. [SGJN19] combine reactive and proactive scaling. Hybrid scaling is beneficial in production systems to combine predictive power and stabilizing actions in case of unseen load spikes. Ali-Eldin et al. [AETE12] analyze different combinations of reactive and proactive scalers and conclude that reactive scalers should be involved in upscaling decisions while downscaling should be initiated by proactive components only. This principle has also been adapted by other approaches [JD18, IDCJ11]. These approaches require a proactive scaler that is not too conservative, meaning it should regularly trigger downscaling actions. Otherwise, the goal of cost-efficiency is not reached. The decision of whether to use proactive or reactive scaling logic is often solved by using user-defined thresholds [IDCJ11, USC⁺08, SKG⁺21] or other user-specified parameters [BHS⁺19]. Most hybrid scalers rely on the reactive component only in case of service-level objective (SLO) violations [AEKTE12]. Bauer et al. [BLV⁺19] use so-called trust thresholds that take the accuracy of the predictive model into account and possibly omit reactive decisions to resolve scaling conflicts. In the approach of Zou et al. [ZLZ⁺24], combining scaling recommendations from both reactive and proactive modules is the task of an optimization component that employs a model predictive control mechanism. They refer to their algorithm not only as hybrid but as collaborative, which means that optimization and uncertainty handling are first-class features of their algorithm.

The Kubernetes HPA has been targeted by many researchers that included proactiveness in the component. For example, Zhou et al. [ZZM⁺23] present an extension of the HPA used in the Alibaba Cloud. Here, a combination of reactive and proactive

scaling is used. It is one of many works utilizing time series forecasting to predict future workloads and analyze trends. In addition, the authors use a queueing model to predict the performance of managed applications. The authors show that their approach performs better than the HPA baseline for various scenarios. Our approach pursues the same goal as predictive autoscaling, to match the resource demand best. In contrast to many proactive approaches, we do not use forecasting or modeling to predict future system states. Our approach is characterized by a high scaling frequency that allows near-instant reactions to load changes. The lower scaling frequency of interval-based autoscalers makes wrong decisions costly. Many state-of-the-art hybrid autoscalers rely on additional configuration parameters or models to balance reactive and proactive scaling. This generally increases the configuration effort and introduces another source of error.

Autoscaling metrics. As stated earlier, autoscalers can use platform and/or application metrics as inputs for their scaling algorithm. Many papers use platform metrics such as CPU utilization and memory metrics as their only input for scaling [ADTK⁺20, GNI⁺19, SSGW11, TTT⁺18]; some are even purely CPU-focused [GGW10, KCH09, SGLI11]. The main advantage of these approaches is their broad applicability, as platform metrics are virtually always available and captured automatically in many environments. The number of incoming requests is the most used application-level metric used by more than half of the approaches analyzed by Singh et al. [SGJN19]. Other custom scaling metrics used include the number of active connections [CHL⁺08] or the number of active sessions [CMKS09]. Casalicchio [Cas19b] proposes a modification of the Kubernetes HPA for CPU-intensive applications where an absolute CPU metric is used instead of the default relative CPU utilization metric. The work shows that changing the scaling metric can improve the convergence of the HPA and save costs. Zhu et al. [ZHZ22] present a modification of the HPA that uses, in addition to the default CPU utilization metric, also the utilization of a thread pool. The authors show the advantages of the approach for microservices on Tomcat and Httpd servers. Generally, we identify potential in scaling based on application-level metrics and the combination of platform and application metrics for autoscaling.

Monitoring and autoscaling. Tari et al. [TGAPG24] identify monitoring as a main challenge for autoscaling serverless functions due to their distributed and transient nature. In the use case of serverless functions and large-scale microservice applications, sending monitoring data can be costly and error-prone due to the vast number of deployed service instances. A related research area is decentralized monitoring systems. Recent research from this area originates in the edge computing community aiming to increase fault tolerance [IFDB22] and workload balance [CSFN20]. To the best of our knowledge, there is currently no autoscaling approach that explicitly concerns the problem of inaccurate or delayed metrics, although it has been reported in production systems [GAAC24]. A major point of criticism for research autoscalers is that many of them are evaluated only in simulation environments. According to the survey by Singh et al. [SGJN19], 30 autoscaling approaches are evaluated using simulation only, and an even greater subset is using synthetic workloads. In general, this leaves the question open of how well these autoscalers perform in production environments and leads to the fact that delayed, inaccurate, or incomplete measurements cannot be considered.

Many autoscaling approaches use response time measurements for an internal model evaluation or, when reinforcement learning is used, to calculate the reward. For example, Aslanpour et al. [AGAT17] require low and high response time thresholds for down- and upscaling, respectively. We argue that there are two problems when relying heavily on response time measurements. First, as we show in Section 6.1, response time measurements can be erroneous or delayed in critical scenarios. As a special case, if errors occur in the application, the response time alone might not be suited to characterize an overloaded service. In these cases, other metrics might depict the application state better. Second, the response time has, by design, lower and upper bounds. The lower bound is given by the minimal execution time of business requests, which cannot be further reduced by provisioning more resources for the respective service. Request timeouts give the upper bound in interactive applications. This is why the response time cannot be used to derive scaling decisions without limitations.

Configuration of autoscalers. The aspect of configurability is rarely addressed in the autoscaling domain. Kalyvianaki et al. [KCH09] provide a resource provisioning scheme based on Kalman filters while explicitly claiming low configuration overhead. The type and meaning of configuration parameters needed by various approaches are manifold. Threshold- or rule-based autoscalers require many critical manual settings that influence the performance of the autoscaler massively [AHSS13]. Most autoscalers require at least up- and downscaling thresholds. Some require other inputs like manually created models [BLV⁺19] or costs of reconfiguration [RDG11]. As stated above and by Jiang et al. [JLZL13], especially hybrid scalers often require manual parameter or offline tuning. Many of these settings cannot be determined in advance by application developers or require extensive load testing to be set appropriately. An additional challenge is keeping configurations up to date, especially with respect to frequent application updates, which are likely to happen in DevOps contexts. Reinforcement learning, as used by various recent approaches [JD19, RFS⁺20, WKL⁺19], offers one way to reduce configuration overhead. However, it comes with the difficulties of defining a suitable reward function and needing lots of training data. Moreover, most reinforcement learners assume a static application and have problems with changes introduced by updates [DMM⁺10].

Explainability in autoscaling. Explainability is a desired property of autoscaling because of its criticality in influencing the service quality and operating costs of a cloud application. Klinaku et al. [KSZB23] outline expectations and requirements on explainable autoscaling frameworks. Ghanbari et al. [GSLI11] state that model-based autoscalers (e.g., based on queueing networks) are hard to understand, while rule-based scaling is, in general, better in terms of explainability. However, the authors state that complexity can also be high for rule-based scaling when a large set of rules is used. Many research autoscalers achieve some explainability by reducing the complexity of the scaling problem through the use of many configuration parameters. Zhao and Uta [ZU22] argue that serverless functions are fine-grained, tiny workloads, and hence, it is worth investigating simple, tiny autoscalers for them. Their experiments show that simple, explainable algorithms like moving averages can deliver appropriate scaling results. Explainability is a concern, especially for AI-based or black-box autoscalers. New research investigates the use of explainable AI for autoscaling [MBKV23].

Autoscaling and dependent orchestration tasks. Only a few existing autoscaling approaches explicitly target the dependency between autoscaling and other orchestration tasks, like load balancing and health monitoring. Chen et al. [CHL⁺08] propose an autoscaling mechanism and also evaluate load dispatching algorithms in parallel. Dezhabad and Sharifan [DS18] connect a scaling and a load balancing unit to provision firewall applications. Gandhi et al. [GHRK12] connect the question of server provisioning and traffic routing in a multi-tier data center. These studies indicate that load balancing and other orchestration mechanisms have to work together to achieve good quality of service. Selected papers from the cloud orchestration domain, where autoscaling is supported but not in focus, are reviewed later in this section.

In the following, we summarize works closely related to our contribution of continuous decentralized autoscaling and state the main points of differentiation.

Decentralized autoscaling. DEPAS [CCDN⁺12, CKP12, CP12] is an interval-based decentralized autoscaler built for P2P architectures. Similar to our work, scaling decisions are made at the service instance level. In contrast to our work, DEPAS assumes that each instance is aware of a set of other instances and can approximate the total system load. DESA [PBCK23] is a decentralized autoscaler for fog computing environments. In contrast to our approach, the work is limited to threshold-based scaling and simulative evaluation. Fractal [KOM⁺19] embeds orchestration logic into service replicas, including scaling, failure recovery, and more. In contrast to our work, the approach is strongly coupled to a concrete orchestrator and uses threshold-based scaling only. In the Swayam approach [GEH⁺17], frontend services can autonomously trigger scaling for backend machine learning inference services. One key difference to our work is that the authors focus on machine learning backend services with high setup times and explicitly keep idle services deployed for some time.

Herrera and Molto [HM20] introduce a bio-inspired autoscaling algorithm where the autoscaling process is interpreted as an evolutionary optimization. Different scaling metrics (e.g., CPU load, RAM usage, and I/O operations) are merged into one number by using so-called normalized extended resource metrics functions. The authors distinguish between two alternative concepts. In the self-sufficient cell model, one container has only its own metrics available. In the interactive cell model, one container also receives information from its neighbor containers. One container at a decision cycle can decide between vertical and horizontal scaling actions. The scaling logic is threshold-based and has a so-called dead range between the thresholds where no scaling actions are performed. Probabilistic corrections stop cells from making the same decision, preventing overreactive behavior. The evaluation of the different models is based only on simulation. In contrast to this work, our approach is more flexible, allowing us to define arbitrary scaling logic. Moreover, we explicitly design a continuous algorithm where container decisions are distributed over time, and the speed and convergence of the algorithm can be analytically assessed. Also, we proved the compatibility of our approach with modern container orchestration frameworks with actual implementations and performed evaluations in representative cloud environments.

An autoscaling approach where decisions are made at the last instance of a call chain is proposed by Desmonceaux et al. [DEC21]. In contrast to our work, an explicit model of the load balancer and only threshold-based scaling is used. Two works [NFT20,

NLV⁺19] propose decentralized scaling for microservices where decisions are made at the node level, making them not easily applicable in the serverless function use case where servers are hosting many different service instances. A hybrid threshold-based autoscaling approach, evolving both local decisions and global monitoring data, is proposed by Merkouche and Bouanaka [MB22]. In general, several decentralized scaling approaches [XSY⁺23, BT19, Rus18] specialized in stream processing tasks have been proposed. In summary, the uniqueness of our work is that we propose a pure-local continuous scaling approach with flexible configuration.

Decentralized cloud orchestration. Several papers dealing with the decentralization of orchestration tasks in cloud and edge computing are reviewed by Ullah et al. [UKK⁺23]. For example, Swarmchestrator [KUT⁺24] and CODECO [SSFR⁺24] envision decentralized orchestration frameworks for the cloud-edge continuum. Tomarchio et al. [TCM20] motivate using autonomic, self-organizing approaches to cope with decentralized multi-cloud setups. MiCADO [KKK⁺19] is an application-level cloud orchestration framework. MiCADO supports scaling both at the virtual machine and application level and allows for creating flexible scaling rules that can also be optimized with a machine learning model [Kov19]. In contrast to our work, scaling is based on metrics from Prometheus, a metric collector that combines application- and VM-based metrics. ENORM [WVMN20] is a resource management framework for edge servers with support for autoscaling, resource provisioning, and more. Edge nodes can make scaling decisions here, but only in periodic intervals. In the HYDRA approach [JS22], orchestration tasks are performed collectively by a set of nodes, and each node in the system serves both as a computation resource and controller.

Use of probability in autoscaling. DEPAS [CCDN⁺12] uses explicit probabilities for scaling actions as an outcome of their decision policies. Mazidi et al. [MGYT20] propose a model that outputs probabilities for resource over- and underprovisioning. Similar to our work, comparisons with random numbers are used to ultimately decide which action to take. Our work adopts this idea and extends it with an execution layer that has the right to deny the execution of a decision, for example, if it violates a global constraint. Other works use probabilities as part of intermediate steps to determine the scaling action, primarily to handle uncertainties. For example, MagicScaler [PWZ⁺23] uses probabilistic demand prediction and a sampling method to approximate an optimal scaling policy.

Autoscaler modeling and validation. A review by Agos Jawaddi et al. [JJI22] finds that discrete- and continuous-time Markov chains or Markov decision processes are the most popular model specifications for autoscaler verification. Comparative studies outline the strengths and weaknesses of different modeling formalisms for cloud autoscaling [AJIMHK23, BDvZ⁺21]. Toka et al. [TDFS20] present a discrete-time queueing model of the Kubernetes HPA. It is used within a simulation that estimates the HPA's decisions for different workload traces. Tournaire et al. [TCTH23] study VM autoscaling approaches based on hysteresis policies (i.e., multi-threshold rules) using Markov chains. These hysteresis policies can be interpreted as special cases of our configuration space. An alternative modeling formalism for microservice autoscalers is presented by Merkouche et al. [MBB23], where the authors use hierarchical parallel Petri Nets for verifying the scaling behavior. PEAS [PAEr⁺16] allows

for computing probabilistic bounds for the autoscaler behavior by evaluating many workload scenarios.

In summary, autoscaling of modern cloud applications is a large research field, and although many prior works exist, there are still open challenges that motivate novel works. We will systematically investigate these challenges and illustrate them with experiments in Chapter 6. Our contribution touches on many of the autoscaling aspects and challenges outlined in this section. In the following, we briefly discuss the main characteristics of our contribution. We bridge the gap between reactive and proactive scaling by increasing the scaling frequency to better match actual resource demands. Thereby, we still rely on monitoring data and circumvent potential errors by predictive models. Chapter 6 will show that we offer a flexible approach supporting both application and platform metrics while keeping the configuration overhead manageable. By relying on the concept of decentralization, we eliminate the dependence on central monitoring systems, making our autoscaler more failure-resistant and well-suited for large-scale environments with many service instances. By reducing the action space of single autoscaler decisions, we can apply explainable, tiny reasoning mechanisms. We will also show that our autoscaler can be meaningfully evaluated analytically, by simulation, and in real experiments. The unique combination of continuity and decentralism is the standalone characteristic of our contribution compared to existing works.

3.4 Simulation of Container Orchestration

This section covers related works in the area of modeling and simulation of cloud applications and platforms. The section is split into multiple parts that examine general-purpose cloud simulation frameworks, approaches focusing on Kubernetes clusters, and performance models for microservice applications. As our approach for accurate and authentic simulation of microservices in Kubernetes clusters presented in Chapter 7 is based on the interaction between a simulation and actual components of Kubernetes clusters, we also take a broader look at literature that focuses on the interaction of simulations and external artifacts.

Cloud simulators. Bamrik [Bam20] reviews cloud simulation approaches. In this survey, simulators are categorized by their primary purposes. The stated categories are general cloud modeling, data center processing, economical modeling, and application modeling. The CloudSim ecosystem [CRB⁺11] is one of the largest simulation frameworks for cloud computing. It has a history of over 10 years of development and supports cloud, edge, and fog computing environments. Several single-purpose simulators have been proposed as extensions. One example is iFogSim [GVDGB17], which focuses on the modeling and simulation of resource management methods in Internet of Things, edge, and fog computing environments. Another extension is ContainerCloudSim [PDCB17], which simulates containers running on top of virtual machines. Demonstrated use cases in the paper include the investigation of container and VM start delays and container placement strategies.

Mastenbroek et al. [MAJ⁺21] introduce OpenDC, a comprehensive simulator for data center operations. They propose a generic model that enables the simulation of various types of workloads, including virtual machines, containers, stream processing frameworks, machine learning frameworks, and high-performance computing batch tasks. The authors claim to implement the most recent operational models for modern workloads, such as serverless functions, as well as more convenience functions that allow users to run simulations without deep knowledge. In contrast to CloudSim, OpenDC follows an integrative approach, which makes it easier to combine results from different submodules. Nikdel et al. [NGN17] propose DockerSim, a simulation engine focusing on modeling the behavior of containers accurately. One focus here is modeling a container’s network and resource usage behavior. Differences between traditional cloud simulators and their models for virtual machines are explicitly highlighted. Evaluation results focus on single container software-as-a-service applications.

Kubernetes modeling and simulation. Medel et al. [MRBnA16] present a formal model of one of the earliest versions of Kubernetes. They combine it with a container lifecycle model to define metrics like pod creation, restart, and termination times. Ghirardini et al. [GSFP20] propose a model-driven technique to predict platform metrics like CPU and disk processing rates based on the Palladio component model. They also outline the usage of the developed simulator as a tool for comparing different settings for Kubernetes components, like the HPA. Turin et al. [TBD⁺20] present a formal model of Kubernetes based on the real-time abstract behavioral specification language. Their approach includes models of pods, services, autoscalers, nodes, and schedulers. It supports the evaluation of different client workloads and microservices. Their evaluation compares simulation outputs with real experiments and shows that although deviations exist, the model can provide useful estimations for metrics like CPU usage and deployed pods over time.

Bhardwaj et al. [BB23] present KubeKlone, a simulator for microservices running in Kubernetes cloud or edge clusters. The approach uses queueing networks for microservices and common sidecar containers in Kubernetes environments like API gateways and service meshes. Their evaluation shows that the 99th percentile latency can be accurately estimated with the framework. The framework has interfaces to export simulated metrics to train machine learning models. Borsatti et al. [BCF⁺24] present KubeTwin, a simulator for Kubernetes clusters that comprises its own implementations of network services, load balancers, DNS resolution, autoscaling, and scheduling. The main use case evaluated is motivated by edge computing and multi-regional clusters. Therefore, enhanced networking models for inter-cluster and intra-cluster communication are proposed. Wen et al. [WHQ⁺23] propose K8ssim, a framework for simulating real cluster traces. Similar to our work, the evaluation of scheduling algorithms is a particular focus. However, in contrast to our work, their approach requires re-implementation of scheduling algorithms while our approach directly interacts with the `kube-scheduler` and executes its original source code.

Microservice performance simulation. Heinrich et al. [HvHK⁺17] define the learning of infrastructure behavior and integration into performance models as one of the research challenges for performance engineering of microservice applications. Barna et al. [BKFL17] propose a layered queueing network for modeling the performance

of containerized applications. Interconnected microservices within an application are modeled as sequential queues, while each microservice has software, container, and hardware queues. The model is connected with a Kalman filter that is used for parameter tuning based on real measurement data. Courageux-Sudan et al. [CSOQ21] present a microservice performance model requiring fewer calibration values than many other state-of-the-art works. The model accuracy was evaluated in benchmarking studies with microservice reference applications. The performance model has been used by other simulators as their core model, for example, by MiSim [FWH⁺22], which is introduced in Chapter 7 and a foundation of our contribution. As an alternative to queueing networks for modeling microservice performance, Da Silva Pinheiro et al. [dSPPSM23] present an alternative approach using Stochastic Petri Nets.

A particular focus of our work is the connection between microservice performance and the effects of container orchestration. Previous work in microservice performance modeling and simulation integrates either no or only self-implemented, simplified runtime orchestration mechanisms [BWB⁺19, FWH⁺22, JPG19, VDR⁺20]. There are several simulations for component-based systems that cover models of selected container orchestration mechanisms, (e.g., load balancing [BBM13, ZGD19], autoscaling [ATTG21] or networking [VDR⁺20]). In contrast to these works, we aim to combine microservice simulation with multiple container orchestration mechanisms using their original implementation (i.e., without hand-crafted models).

One potential use case of our contribution is the evaluation and optimization of container orchestration policies. Khan et al. [KTADK23] propose Perfsim, a framework specialized in simulating large-scale service chains in cloud-native computing. Similar to our work, one aim is to analyze resource management and placement policies in Kubernetes environments. The authors state that the resource management algorithms of Kubernetes are only "partially imitated," which stands in contrast to our work. ConfAdvisor [CNH⁺19] is a performance tuning framework for Kubernetes applications. Based on crawlers, information about container images and application configurations is extracted from deployed applications. Also, runtime metrics queried by Prometheus serve as inputs to the framework. The core of the approach is a rule-based configuration advisor. The assumption is that developers can define custom rules from their expert knowledge or best practices. The evaluation shows improvements in database applications like Cassandra and MongoDB. Raith et al. [RND24] propose SimuScale, a framework for optimizing autoscaling parameters in serverless environments. Here, the simulation is used at runtime and receives the state of a running Kubernetes cluster as input. Several simulation scenarios are executed to find optimized configuration parameters for autoscaling.

Interaction between simulation and external artifacts. Previous works have dealt with the connection between simulations and real code. *Software-in-the-Loop* [DGK07] is a term in this area that describes this connection, inspired by *Hardware-in-the-Loop* simulations. Such approaches focus primarily on testing control software in autonomous systems [BAZA17, HSL16b]. Hildebrandt et al. [HBH07] propose the idea of simulation-driven development of peer-to-peer networks. Here, components in the development process are first represented by models and then iteratively replaced by real system components. Erb and Kargl [EK14] note general analogies

between discrete-event simulations and event-driven architectures. In the area of software performance simulations, a similar concept has been used by Von Massow et al. [vMvHH11], combining a simulation and an adaptation controller. In this work, we apply a similar concept to microservice simulation and Kubernetes using scheduling and cluster autoscaling as exemplary container orchestration mechanisms. In general, both Kubernetes scheduling and cluster autoscaling are active research areas, as confirmed by recent articles [Car22, TTEP20, WZW20]. For Kubernetes scheduling, there is a community project called kube-scheduler-simulator⁵ where different scheduling policies can be tested, similar to our work. In addition, our work allows us to evaluate the impact of different scheduling strategies on the performance of deployed services, as we will show in a case study.

In summary, the uniqueness of our contribution to an authentic simulation of microservice performance and container orchestration lies in the combination of a state-of-the-art microservice simulator and performance model with original Kubernetes artifacts. With this combination, we are to explore novel use cases of the simulation (such as optimization of Kubernetes orchestration policies) and increase the accuracy of the simulation. Integrating Kubernetes components using their original artifacts allows us to efficiently integrate authentic container orchestration mechanisms without the need for complex modeling, re-implementation, or simplification.

⁵<https://github.com/kubernetes-sigs/kube-scheduler-simulator>

Part II

Contributions

Chapter 4

A Systematic Approach for Benchmarking of Container Orchestration Frameworks

According to Gartner,¹ 80% of custom software running at the physical edge will be deployed in containers by 2028. Container orchestration frameworks play a critical role in managing container applications and in modern cloud computing paradigms such as cloud-native or serverless computing. They significantly impact the quality and cost of service deployment as they manage many performance-critical tasks like container provisioning, scheduling, scaling, and networking. Consequently, a comprehensive performance assessment of container orchestration frameworks is essential.

Both conceptual and technical challenges arise for comprehensive performance evaluation and benchmarking of container orchestration frameworks. The term container orchestration is not used uniformly. A wide range of tasks is usually associated with CO frameworks, for example, load balancing, networking, horizontal and/or vertical scaling, scheduling, and availability. As a basis for developing a benchmarking methodology for these frameworks, we have to review the essential roles and functionalities of CO frameworks in modern cloud environments. All the mentioned tasks influence each other, and good metrics that quantify the performance of the CO framework have to be defined. According to Kounev et al. [KLvK20], good metrics for benchmarking should be reliable, repeatable, consistent, and independent of particular systems under test. Another challenge is the interference of applications and orchestrator performance. On the one hand, we want to look at the performance of CO frameworks as isolated as possible. On the other hand, we do not want to lose sight of the application, which, in practice, strongly influences end-to-end metrics like response times that are crucial for users. From a technical point of view, one problem is that different CO frameworks have very different technology stacks and interfaces. A benchmarking framework has to either support all technology stacks or define abstractions that enable capturing all relevant benchmark metrics without making limiting assumptions. As discussed in Section 3.1, state-of-the-art benchmarking approaches are limited to analyzing single orchestration tasks or only considering one specific CO framework and technology stack.

Until now, there is no benchmarking approach that covers the variety of tasks implemented in such frameworks and supports evaluating different technology stacks. This chapter presents a systematic approach that enables benchmarking of CO frame-

¹<https://cloud.google.com/resources/gartner-magic-quadrant-containers>

works. First, we review and summarize various definitions of container orchestration and derive the benchmarking scope and use cases for such frameworks. After eliciting requirements for a benchmark, we propose a design capable of capturing relevant metrics. The implementation of our concept is the benchmarking framework COFFEE. COFFEE allows for defining complex benchmarking campaigns through a user-friendly, script-like, and framework-agnostic language. We demonstrate the potential of our approach with case studies of the frameworks Kubernetes and Nomad in a self-hosted environment and on the Google Cloud Platform.

This chapter addresses Challenge 1 from the overall context of this thesis. Our contribution allows a comprehensive performance assessment of container orchestration frameworks, considering modern frameworks' broad functionality. The results of benchmarking runs can provide important data for selecting and appropriately configuring a container orchestration framework in practice. To achieve Goal A of this thesis, we address the following guiding research questions in this chapter:

- RQ A1: Which features of container orchestration frameworks are feasible to evaluate through benchmarking?
- RQ A2: Which metrics can be used to quantify the performance, and how can they be measured?
- RQ A3: How to define a benchmark for container orchestration frameworks and enable execution on diverse technology stacks?

The remainder of this chapter is structured as follows: In Section 4.1, we define our benchmarking use cases and scope. In Section 4.2, we derive requirements for container orchestration frameworks and associate metrics with every requirement. In Section 4.3, we discuss a high-level design that enables capturing all previously derived metrics without making vast assumptions about the system under test. Section 4.4 presents our benchmarking framework COFFEE, which implements the proposed architecture. Sections 4.5, 4.6, 4.7, and 4.8 feature several case studies, including Kubernetes and Nomad as frameworks under test. Section 4.9 summarizes the chapter and discusses our findings. We published parts of this chapter, including text paragraphs, figures, and tables, as a full research paper at the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE) [SMBK23].

4.1 Identifying Benchmarking Use Cases and Scope

This section focuses on the term container orchestration and derives use cases and scope for benchmarking of CO frameworks. To define a fair benchmarking methodology that is not biased towards one particular CO framework, we need to investigate container orchestration conceptually. In order to get a better understanding of this term, what it means, and how it is commonly used, we looked into scientific and non-scientific sources, including technical documentation and blog posts. We used Google Scholar, ACM Digital Library Search, IEEE Xplore Search, and Google Search to find

4.1 Identifying Benchmarking Use Cases and Scope

the relevant references. The aim of this pre-study is to identify essential performance-critical tasks of CO frameworks that we need to address in our benchmarking methodology.

A main finding in this pre-study is that there is not one commonly accepted and highly cited definition of the term container orchestration. Casalicchio [Cas19a] states that container orchestration is concerned with managing containers at runtime and supporting deployment, execution, and maintenance. Common features are resource limit control, scheduling, load balancing, health checks, fault tolerance, and autoscaling. Khan [Kha17] names seven capabilities of container orchestration frameworks: cluster state management and scheduling, high availability and fault tolerance, security, networking, service discovery, continuous deployment, as well as monitoring and governance. Rodriguez and Buyya [RB19] propose a container orchestration reference architecture with the key tasks of container provisioning, monitoring, scheduling, as well as accounting and admission control. Siddiqui et al. [SSK19] define container orchestration as lifecycle management of containers, including "regular" activities, such as scaling, load balancing, and upgrades. Struhar et al. [SCA⁺21] see the placement of containers as a key functionality of CO frameworks and name container removal, load balancing, scaling, and health monitoring as further tasks.

Looking at non-scientific sources, we analyze the definitions from technology leaders like IBM,² VMware,³ Google Cloud,⁴ CircleCI,⁵ AWS,⁶ Datadog,⁷ and more. We found that those definitions are broader and cover all functionalities in modern CO frameworks. The definition of Red Hat⁸ is one of the most comprehensive that we reviewed. It states that container orchestration automates the deployment, management, scaling, and networking of containers. The following typical tasks are named: provisioning and deployment, configuration and scheduling, resource allocation, container availability, scaling, load balancing and traffic routing, health monitoring, application configuration, and securing container interaction. This definition overlaps also with the ones from the scientific literature but adds additional capabilities of state-of-the-art tools. It introduces container orchestration as a generic and broad term but also names concrete responsibilities. We use this definition as a basis for deriving the scope of benchmarking that we target in this contribution.

Various implications and use cases for benchmarking are created from the technical scheme of container orchestration frameworks presented in Figure 2.3. The figure shows that CO frameworks sit on top of the technology stack, abstracting lower-level layers, like the operating system and server hardware. When we evaluate the performance of a CO framework, we automatically examine the entire technology stack, meaning that the software and hardware of the cluster nodes also play a role. This has to be considered when interpreting benchmarking results. Based on these findings, we identify three use cases for the benchmarking of container clusters:

²<https://www.ibm.com/topics/container-orchestration>

³<https://www.vmware.com/topics/glossary/content/container-orchestration.html>

⁴<https://cloud.google.com/discover/what-is-container-orchestration>

⁵<https://circleci.com/blog/what-is-container-orchestration>

⁶<https://aws.amazon.com/what-is/container-orchestration>

⁷<https://www.datadoghq.com/knowledge-center/container-orchestration>

⁸<https://www.redhat.com/en/topics/containers/what-is-container-orchestration>

1. *Comparing different CO frameworks*: In this case, a fixed resource landscape is given, and different CO frameworks are evaluated in this context. The use case in practice is to choose a CO framework for a specific environment.
2. *Comparing configuration options of one CO framework*: Similar to the first use case, the environment is not changed. Instead, a different configuration of the CO framework is used (e.g., a different scheduling algorithm or networking solution). The use case in practice is fine-tuning a specific framework (e.g., comparing different networking plugins for Kubernetes).
3. *Comparing different cluster environments*: In this case, the focus is not on the CO framework but on the nodes. There can be different configurations of the nodes (e.g., container engines, operating systems, or hardware resources). One use case in practice is selecting the size of VM instances in public clouds.

This chapter demonstrates all three use cases in our case studies in Sections 4.5, 4.6, 4.7 (Use Cases 1 and 3), and 4.8 (Use Cases 2 and 3).

We have to define a benchmarking scope to enable benchmarking of CO frameworks. Earlier in this section, we discussed that container orchestration is usually interpreted as a set of tasks that automate the management of container applications at runtime. In this work, we select those CO tasks as subjects to evaluation through benchmarking that are expected to impact either the performance of the managed container applications or the costs of their operation (e.g., quantified by resource utilization or energy consumption). Based on these criteria and our reviewed definitions, we derive the following performance-critical tasks of CO frameworks:

- T1: Container provisioning and deployment
- T2: Container scheduling
- T3: Resource allocation
- T4: Container availability
- T5: Health monitoring
- T6: Scaling
- T7: Load balancing and traffic routing
- T8: Inter-container networking

This list correlates most with Red Hat’s definition, which we found to be clear and comprehensive. However, we omit security-related activities of CO frameworks (e.g., access and identity management), as they are hard to quantify by performance metrics and to generalize across different frameworks. We will explore the meaning and facets of these tasks further in the next section.

Given this list of assumed tasks of CO frameworks and the environment setting shown in Figure 2.3, three necessary properties for our benchmarking approach emerge.

First, it must provide a level playing field for evaluating different CO frameworks. This is implied from Use Case 1 and goes along with the best practice of designing a fair benchmark that is not tailored to a specific system under test [KLvK20]. Second, since we deal with different nodes and technology stacks, the measured performance metrics must be generic, with minimal assumptions on the orchestrator and cluster nodes. Third, the variety of tasks of a CO framework necessitates broad coverage and metrics for every task. One should also consider that all of these tasks are concurrently executed; therefore, interdependencies exist between individual tasks. Consequently, a benchmarking approach should target as many tasks as possible in parallel instead of evaluating them one after the other.

4.2 Requirements and Metrics

In this section, we decompose the abstract list of CO tasks defined in the previous section into concrete fine-granular requirements, which can then be evaluated and quantified by specific metrics. The list of requirements has been deduced from three sources: (1) Scientific studies that analyze or discuss performance aspects of container orchestration (see Section 3.1), (2) use case descriptions for CO frameworks from industry leaders and blog posts (see references in the previous section), and (3) technical capabilities of state-of-the-art CO frameworks retrieved from their documentation. In the following, we define and explain these requirements. In addition, we map metrics to each requirement that can quantify how well a CO framework fulfills each requirement. As mentioned before, the metrics should be repeatedly and reliably measurable and not impose limiting assumptions on the nodes or the orchestration framework.

Requirement R1. *Containers can be started.*

Related tasks from Section 4.1: T1, T2, T3, T5

One of the core requirements of any CO framework is the ability to start containers. If a new container has to be launched, the scheduling algorithm decides first on which node the container should be placed. Then, node resources are reserved, and if necessary, the image is downloaded from a registry before the container is started. The primary metric to quantify the start performance is the readiness time, the time it takes from issuing the start command until the container context is initialized and the container is running and able to execute business logic. The readiness time consists of different phases: scheduling, image pull, start time, and setup time. The start time is the time it takes for the container engine to process the image manifest and set up the controller environment (e.g., control groups and namespaces). The setup time is the application-specific part of the readiness time and could involve activities like initializing a language runtime or connecting to a database.

Requirement R2. *Containers can be removed.*

Related tasks from Section 4.1: T1

Every container that was started should also be terminated and removed. The associated performance metric is the removal time. The removal time is expected to

be significantly lower than the readiness time since no placement and image pull will occur. However, depending on the container, there may also be pre-destroy statements that allow a graceful exit (e.g., a clean disconnect from a database).

Requirement R3. *Containers can access persistent storage.*

Related tasks from Section 4.1: T1, T3

Although most containers are stateless, as best practice guidelines suggest, there are use cases when containers need access to persistent storage. A prominent example is a database, where the database engine can run inside a container, but the data is in persistent storage that also exists after the container's lifetime. Usually, the persistent storage of a container is called a *volume*. Volumes can be provided by classical storage on the host's file system or via network storage. Related performance metrics are read/write times and throughput. In addition, an increased readiness time of the container could be observed, as a link to the persistent storage has to be established.

Requirement R4. *Containers can be restarted manually or in case of failures.*

Related tasks from Section 4.1: T1, T2, T3, T4, T5, T7

Automated restarts are usually out of scope for container engines but a core task of CO frameworks. The reasons for restarts can be various: developers can manually trigger restarts, or unexpected errors in the container process might cause them. Another reason could be that the container runs out of resources (e.g., in cases when it is overloaded). In the latter case, health monitoring should track the state of the container. One performance metric for this requirement is the restart time. It comprises the removal time of the old container and the readiness time of the new one. In case of an error-induced restart, the failure discovery time can be considered, that is, the time difference between the occurrence of an error and the time when the restart is initiated. If the container receives external requests, the load balancing algorithm must also react and not allocate any additional load to the container. In this case, the number of failed requests can also be considered as a performance metric.

Requirement R5. *Containers can be updated to a new version.*

Related tasks from Section 4.1: T1, T2, T3, T4, T5

In continuous deployment, it is essential to be able to perform a rolling update, that is, to update a set of containers to a new version without interrupting the service. One common use case is to change the container image. The task of the CO framework is to perform and coordinate this update as fast as possible but also to maintain availability and avoid violating SLOs. Consequently, we use the total update time, response time, and number of failed user requests as performance metrics.

Requirement R6. *Provisioned resources can be varied depending on workload.*

Related tasks from Section 4.1: T1, T2, T3, T5, T6

Many of the definitions we reviewed in Section 4.1 state that scaling is one of the main tasks of CO frameworks. Dynamic resources can be the number of container instances (horizontal scaling), the computing resources allocated to a container (vertical

scaling), or the number of nodes in the cluster (cluster scaling). Typically, autoscalers are evaluated using a mixture of cost metrics (e.g., instance runtimes or allocated resources) and QoS metrics, such as response times (see Chapter 6).

Requirement R7. *Requests from external sources are balanced across running containers.*

Related tasks from Section 4.1: T7

In the context of microservice applications, multiple application instances are usually deployed simultaneously. When a user requests a service, the request must be assigned to a chosen instance. Load balancing is used to avoid overloading and performance degradation. Metrics for the evaluation of load balancers are the end-to-end response times of user requests and the load balancer’s generated overhead [YKXY19]. In this work, we also consider the distribution of requests over service instances to determine how the load balancer distributes requests over a set of instances.

Requirement R8. *Containers are able to communicate with other containers in the same cluster.*

Related tasks from Section 4.1: T8

Communication between containers is essential for multi-service applications. However, container-to-container communication can also be essential for containers of the same kind (e.g., when states have to be synchronized). Usually, this kind of networking is realized using overlay networks. Classical network metrics like throughput, round-trip time, and latency can be used to evaluate in-cluster networking.

These eight requirements and associated metrics serve as the basis for our benchmarking approach. In the next section, we introduce the high-level design of our benchmarking approach, while Section 4.4 introduces the implementation of our approach, the benchmarking framework COFFEE, in more detail.

4.3 Design Considerations

In this section, we propose an architecture that can quantify a CO framework’s performance based on the requirements and associated metrics defined in the previous section. Our design goals are that the architecture should be as generic as possible and not depend on the specifics of individual orchestration frameworks and underlying technology stacks.

Figure 4.1 shows our proposed architecture. The core component is the benchmark controller, which receives the benchmark specification from the user. The benchmark specification defines starting conditions, instructions for the benchmark run, and its termination. The benchmark controller ensures that the instructions from the benchmark specification are correctly executed. Individual instructions within the benchmark specification can either generate requests to the cluster control plane (e.g., for container starts) or selected test containers (e.g., for failure injection). The test containers are custom containers that run inside the test cluster and collect metrics relevant to the benchmark. A proxy enables communication between the benchmark

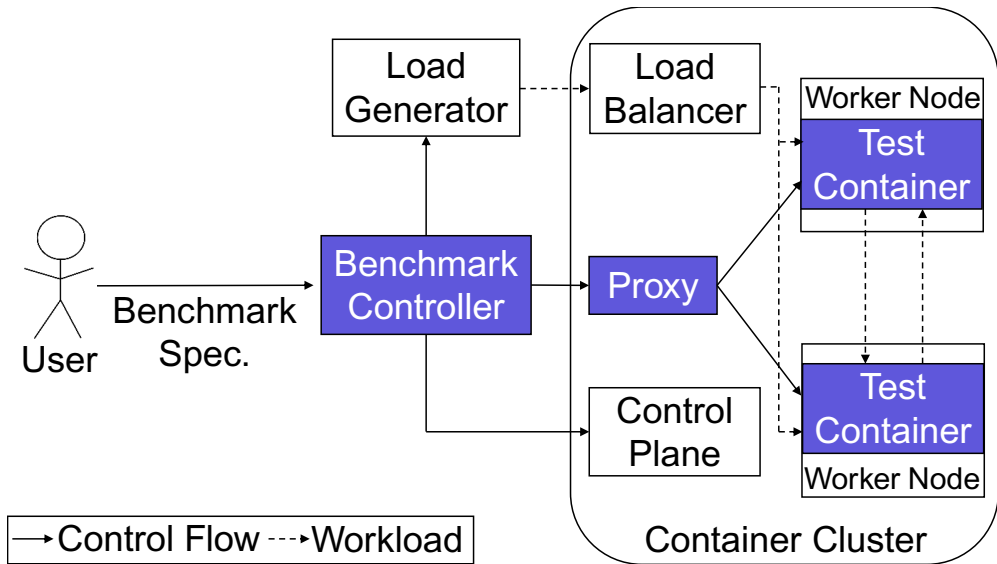


Figure 4.1: Benchmark architecture and components (blue).

controller and the test containers. This is because individual containers are usually not directly addressable from outside the cluster as a network barrier exists between the cluster and external users. A load generator is used to imitate user requests and enables capturing end-to-end metrics like request response times and failed requests. The benchmark controller and load generator should be deployed outside the cluster to avoid interferences. Several test container instances communicate with each other via in-cluster networking.

This architecture satisfies all the properties we defined earlier. The only assumptions made about the CO framework are that a proxy can be deployed, the benchmark controller has access to the control plane, and a load generator can send requests to the cluster. Modern CO frameworks fulfill all these requirements. Furthermore, the proposed architecture allows us to measure many of the performance metrics from Section 4.2. The controller generates a timestamp when the command is sent to measure readiness, removal, update, and outage times. The created, removed, or updated test container instance also reports a timestamp, and the controller can calculate the time difference between the two events. Using a load generator enables measuring metrics like response times, the number of failed user requests, and more. Additional fine-granular metrics might be requested via the cluster control plane. The built-in user-level metrics might be complemented with system-level metrics, like the CPU utilization of nodes. However, note that no extended monitoring capabilities are necessary to use the proposed benchmark architecture.

4.4 The Benchmarking Framework COFFEE

This section introduces COFFEE, a benchmarking framework for container orchestrators. COFFEE follows the benchmarking concept and reference architecture presented in the previous section. Accordingly, it consists of three main components: controller, test container, and proxy. All components use Java and Spring as implementation technologies. The source code of COFFEE and all examples used in this chapter can be found on GitHub⁹ and Zenodo.¹⁰

A core part of COFFEE is the test containers deployed in the cluster. Once the test container's context is initialized, a timestamp is taken and sent to the controller so that it can confirm the instance start. Once the test container is started, seven endpoints can be accessed. The `load` endpoint is accessed by the load generator and performs some dummy operations that can be freely implemented. In our case, it creates CPU load by checking prime numbers and doing factorial calculations with Java's `BigInteger` class. The `load` endpoint also contains a counter tracking the number of received requests from the load generator. The second endpoint `crash` causes the process to end with a non-zero exit code. The third endpoint is used by the container orchestrator for HTTP-based health monitoring. The fourth endpoint causes the HTTP-based health monitoring to fail. Two endpoints are used for network tests. One is called when the instance should send requests to other containers, while the other is used for receiving requests from other instances. Finally, there is an endpoint for storage tests. It writes and reads a random number of Universally Unique Identifier (UUID) strings to and from the container's file system. The file system can be either backed by ephemeral or persistent storage. Before the container is terminated, it sends a timestamp to the controller to confirm the instance removal. In this step, the number of received load requests is also submitted to the controller.

Container orchestration frameworks usually establish an internal network for container-to-container communication. A standard for implementing this is given by the CNI. While internal IP addresses or DNS names of containers can be extracted through APIs from outside the cluster, it is generally not possible to address specific containers in the cluster from outside. This motivates the usage of a proxy that runs inside the container cluster. The COFFEE proxy application acts as a reverse proxy and forwards commands to the test containers deployed in the cluster. It receives the target addresses (internal IP addresses) from the controller that extracts these addresses through APIs exposed by the cluster control plane.

The controller is the most complex component of COFFEE, as Figure 4.2 shows. The controller requires two inputs. First, the test campaign must be specified. The textual input is translated into a set of operations. All supported operations can be found in Table 4.1. In general, there are three types of operations. The first group interacts with the load generator (`LOAD`, `ENDLOAD`). COFFEE provides an interface for the HTTP Load Generator,¹¹ but other load generators can be integrated with low effort. The second group consists of orchestrator-agnostic operations forwarded to the

⁹<https://github.com/DescartesResearch/COFFEE>

¹⁰<https://doi.org/10.5281/zenodo.7603961>

¹¹<https://github.com/joakimkistowski/HTTP-Load-Generator>

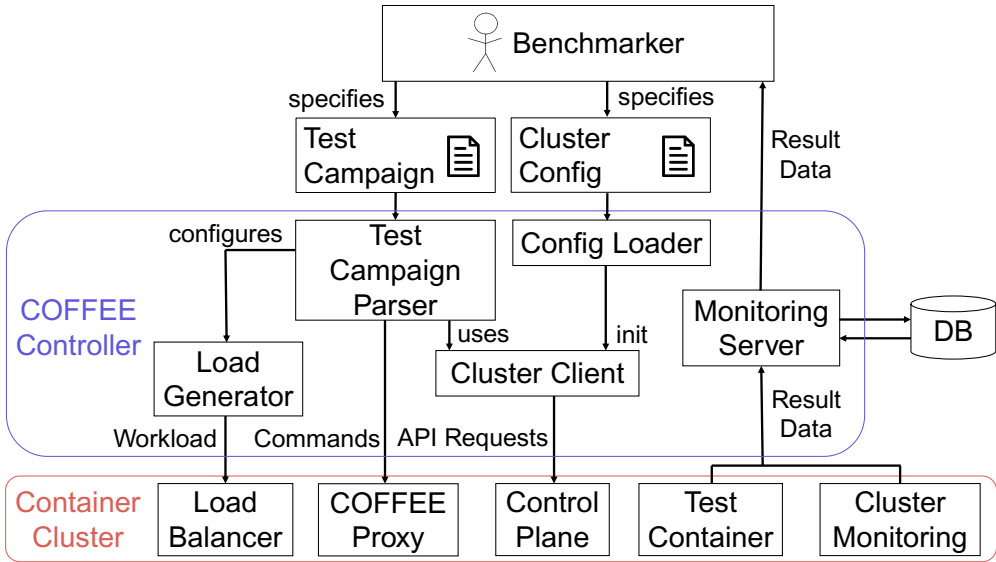


Figure 4.2: The COFFEE controller in detail.

test containers via the proxy (CRASH, HEALTH, NETWORK, STORAGE). The test containers process these requests, so they do not need to be re-implemented for different orchestrators. The third group of operations is orchestrator-specific and needs to be implemented for each framework. It includes container starts (START), removals (REMOVE), updates (UPDATE), and manual restarts (RESTART). In the following, we describe how each operation is implemented and which COFFEE components are involved.

START <n>. This command is used to start n test container instances. The controller initiates the start process by sending the request to the cluster control plane. The implementation of the request is specific for each CO framework. In our implementations for Nomad and Kubernetes, only one request to the cluster control plane has to be sent even if $n > 1$. A timestamp is saved by the controller when the request is sent to the cluster. In parallel, n start command items, including their issue timestamps, are inserted into a first-in-first-out queue that represents unfinished container starts. Once a test container has been started, it immediately sends a request containing the current timestamp to the controller’s monitoring server, the component responsible for processing result data. The monitoring server polls the next item from the queue and calculates the readiness time by subtracting the timestamp from the request from the timestamp saved in the queue item. The readiness time is stored in the result database. A START <n> command is considered completed if all n items in the queue have been polled, or, in other words, n test containers have reported a successful start.

REMOVE <n>. This command is used to shut down n test container instances. The implementation of the required request is specific for each CO framework. Similar to the START command, in our implementations for Nomad and Kubernetes, only one

Command	Description
START <n>	Starts n test container instances
REMOVE <n>	Shuts down n instances
RESTART <n>	Restarts n instances
HEALTH <n>	Sets unhealthy flag in n instances
CRASH <n>	Causes crash of n instances
UPDATE <n>	Updates instances (changes image), considered complete if n instances updated
NETWORK	Measures round-trip time between running instances
STORAGE	Measures disk I/O read and write throughput at one random instance
LOAD / ENLOAD	Starts/Ends load generation
SEQ / ENDSEQ	Starts/Ends a sequence
LOOP <n> / ENLOOP	Starts/Ends a loop with n iterations
OFFSET <t>	Invokes next command after t sec.
DELAY <t>	Pauses a sequence/loop for t sec.

Table 4.1: Commands for test campaign definition.

request to the cluster control plane has to be sent even if $n > 1$. In general, the `REMOVE` command works very similar to the `START` command. Once a request has been sent to the control plane, n items, including their current timestamps, are inserted into a queue. When a test container receives a stop signal from the CO framework, it sends a request containing the current timestamp to the monitoring server, which polls an item from the queue and calculates the removal time. A `REMOVE <n>` command is considered completed if all n items in the remove queue have been polled or, in other words, n test containers have reported a shutdown.

`RESTART <n>`. This command is used to trigger manual restarts of n randomly selected instances. The concrete implementation of the command depends on the CO framework. In our cases, the controller first retrieves a list of identifiers for the currently active test container instances (e.g., the pod names in Kubernetes). In the next step, n items are selected from this list. As both Kubernetes and Nomad have automatic restarts enabled for deployments and task groups, respectively, we have to send a stop signal to the n selected instances. The instances shut down and are directly restarted. Consequently, manual restarts can be interpreted as a `REMOVE` and `START` command. Hence, the controller’s monitoring server expects n shutdown and n start messages from the test containers. The command is considered completed if all these messages have been registered.

`HEALTH <n>`. This command causes the health monitoring of n instances to fail. Health monitoring for containers can be implemented differently, which is why the

implementation of the command could also vary. In our case, we use HTTP-based health monitoring, where regular heartbeat requests are sent from the control plane to the test containers, which have to answer with a healthy response code within a specified time interval. The COFFEE test containers log every timestamp when they receive a health check request. This type of health monitoring allows us to choose an orchestrator-agnostic implementation for the `HEALTH` command. The controller sends requests to n randomly selected instances via the COFFEE proxy. These requests switch a boolean value in the test containers. This value determines how the test container answers requests to the health check endpoint. By default, test containers send healthy responses. When a `HEALTH` command is executed, the response switches to an unhealthy response with a 400 response code. The cluster control plane is expected to recognize the unexpected response and trigger a restart of this instance. Like the manual `RESTART` command, the monitoring server expects n shutdown and n start messages from test containers.

`CRASH <n>`. The `CRASH` command causes a sudden exit of the application inside the n randomly selected test containers. This mimics an unexpected application error. We use an orchestrator-agnostic implementation here. When a test container receives the `CRASH` request from the proxy, it sends a timestamp to the controller and then uses a `System.exit` call to shut down the application. The CO framework has the task of noticing the exit and then restarting the test container. Similar to the `RESTART` and `HEALTH` commands, n crash and n start messages from test containers are expected by the monitoring server.

`UPDATE <n>`. The `UPDATE` command can be used to test (rolling) update procedures (like canary updates) in clusters managed by CO frameworks. The implementation is orchestrator-specific. The core idea is to change the container image to a new version by changing its tag. A request to the cluster control plane is sent to implement this change. COFFEE offers several options to configure the expected update behavior (e.g., the maximum number of concurrently updated containers) before a benchmarking run. After the image version has been changed, the CO framework terminates the existing test container instances and starts instances with the new version. Note that the `UPDATE` command ultimately updates every test instance. In some test campaigns, it might be unclear how many instances are running. In these cases, the parameter n can be used to indicate how many updated instances must be noted before the command should be considered complete. That means, from the controller's viewpoint, n shutdown and n start messages are expected. In addition to the total update time, the update's progress can also be tracked by reviewing when containers are shut down and when new ones are started. If a rolling update is configured, the parameters of the update (e.g., maximum number of unavailable containers) can be verified.

`NETWORK`. The `NETWORK` command allows assessing the performance of in-cluster container-to-container communication. The controller first retrieves the identifiers of all k running test containers and then generates all $k^2 - k$ pairs of those. The first item of a pair represents the sending instance, and the second one is the receiver. All $k^2 - k$ pairs are evaluated one after each other. The controller sends the instruction, including the address of the receiver, via the proxy to the sending instance. The sending instance captures the current time and sends a request to the receiver. The

receiver puts the current timestamp in the response. The sending instance captures another timestamp when the response arrives. With this procedure, both one-way network delays and round-trip times can be measured. The sending instance submits the metrics to the controller, which proceeds with the next pair. The command is considered completed if the controller has received all $k^2 - k$ results.

STORAGE. The **STORAGE** command exercises the disk I/O of one randomly selected test container. The controller sends the request to the test container via the COFFEE proxy. The test instance generates between 500,000 and 100,000,000 random UUID strings separated by line endings. This corresponds to a data size between 8.5 MB and 1.7 GB. The generated data is then used for writing and reading to and from the container's file system. By tracking the read and write times, the read and write throughputs can be measured. The procedure is, by default, repeated 30 times, always with different data chunks. The measured read and write times are submitted together to the monitoring server that stores these results in the database. As we measure the I/O performance only at one randomly selected container, the command is considered completed if the results from this instance are received. While the implementation of the **STORAGE** command is rather simple, the measured performance metrics heavily depend on how the container's file system is backed. Possibilities include ephemeral or persistent storage and the used storage technology (e.g., network storage, hard disk drive (HDD), or solid state drive (SSD)). In the case that persistent storage should be used, COFFEE executes orchestrator-specific instructions to prepare and mount persistent volumes to the test containers (e.g., creating `PersistentVolumeClaims`¹² in Kubernetes).

LOAD/ENDLOAD. To evaluate load balancing and to measure end-to-end metrics like response times, it is necessary to be able to send user-like requests to the test containers. An established method to do this in benchmarking studies is to use a load generator. In this work, we use the HTTP Load Generator,¹¹ which allows for setting variable load intensities over time. The user has to specify the properties for the load generator before the benchmarking run (e.g., number of requests, request timeouts, maximum runtime). When a **LOAD** command is used in a script, the load generator is started as a process. The load generation stops when an **ENDLOAD** command is called or when the load generator's maximum runtime has been reached. The load generator automatically generates response time statistics. The COFFEE test containers maintain a counter for how many requests they receive from the load generator. Before shutdown, the counter's value is submitted to the monitoring server. With this, the load distribution, a metric to quantify load balancing, can be captured.

Furthermore, auxiliary commands enable the definition of complex test campaigns. They do not produce additional measurements but allow the specification of the control flow within a test campaign. By default, all commands are executed asynchronously and in parallel; by specifying an **OFFSET** `<t>`, the user can specify that a command should be sent t seconds after the start of the experiment. If a sequential execution of a series of commands is desired, one can wrap this sequence by **SEQ/ENDSEQ**. A command is considered completed if all expected responses/metrics of the command

¹²<https://kubernetes.io/docs/concepts/storage/persistent-volumes>

have been reported to the controller. `LOOP <n>/ENDLOOP` can be used for n repeated executions of a sequence. `SEQ` is a syntactic sugar for `LOOP 1`. The keyword `DELAY <t>` can be used within a sequence to set a pause of t seconds between operations.

```
LOOP 100
  START 10
  LOAD
    DELAY 120
    CRASH 10
    DELAY 120
  ENDLLOAD
  REMOVE 10
  DELAY 30
ENDLOOP
```

Listing 4.1: Example sequential test campaign for failure recovery.

Listing 4.1 shows an example sequential test campaign, which is also used in Section 4.6. Here, we use `LOOP 100` to indicate that this experiment should be repeated 100 times. First, we start ten test containers. After starting the load generation, the test script is paused for 2 minutes. This is the warmup phase for the test containers, as the load generator sends requests in parallel. A `CRASH` command is issued in the next step, which terminates all ten running containers. Again, we wait two minutes while the load generator continues to record the response times during that recovery phase. The load generator is shut down, and all ten containers are removed at the end of one experiment run. The 30-second delay at the end provides a short pause before the next loop iteration. Note that using a loop always implies that the loop content is executed sequentially.

```
START 5
OFFSET 10 UPDATE 5
OFFSET 20 LOAD
OFFSET 60 CRASH 1
OFFSET 60 HEALTH 1
OFFSET 140 ENDLLOAD
```

Listing 4.2: Example non-sequential test campaign involving updates and restarts.

Listing 4.2 shows a non-sequential test campaign. Trace-like campaigns can be realized using the `OFFSET` command. In this campaign, five test instances are started at the start of the experiment, and ten seconds after that, these instances are updated. Ten seconds later, the load generator is started and active for a period of two minutes. One minute after the start of the experiment, one test container crashes, and another gets into an unhealthy condition. The difference between the sequential campaign and this scenario is that the execution of the commands at a specific time (relative to the experiment start) is guaranteed. This means that test commands do not wait for predecessors to complete. Here, for example, the `LOAD` command is executed after 20

4.5 Case Study I: Container Provisioning and Networking

seconds, and this happens regardless of whether the previous `UPDATE` command has been completed or not. If this was a sequence, the load generator would be started after the `UPDATE` command has been completed. By supporting both sequential and non-sequential test campaigns, COFFEE allows for defining both interpretable step-by-step workloads and complex trace-like workloads with potentially many actions in parallel.

The specified test scripts can be reused for different frameworks. However, the controller also needs an orchestrator-specific configuration containing at least the cluster control plane address, the orchestrator type, and, if necessary, authentication data. Currently, COFFEE supports Kubernetes and Nomad as frameworks under test. COFFEE can be extended to work with other orchestrators as well. Therefore, the orchestrator-specific configuration, the four orchestrator-specific commands (`START`, `REMOVE`, `RESTART`, and `UPDATE`) explained above, and some query operations, like the retrieval of node names, must be implemented. All result data are processed by the monitoring server component of the controller and stored in a MySQL database. A summary of the most important results can be printed on the console at the end of the experiment.

Overall, COFFEE offers a simple design and is easy to use. It enables automatized benchmarking experiments with different CO frameworks. By design, COFFEE considers benchmark scalability by reducing the number of requests to the controller to a minimum. As COFFEE makes no critical assumptions on the test cluster, it can be used with test clusters of different sizes and resources. The proxy and test containers can be scaled both horizontally and vertically if necessary. The scalability of the load generation depends on the used generator. The HTTP Load Generator used in this work supports distributed execution on various nodes to achieve high request rates. In the following, we show the usefulness of COFFEE in case studies concerning typical use cases for container orchestration frameworks.

4.5 Case Study I: Container Provisioning and Networking

The following sections are concerned with case studies highlighting the features of COFFEE. In the course of this and the next three sections, we cover each of the benchmarking use cases proposed in Section 4.1. We compare the CO frameworks Nomad and Kubernetes in a self-hosted setup and on virtual machines running on the Google Cloud Platform. The test scenarios in this section include container provisioning and networking. All measurement results and evaluation scripts used are available in our replication package.¹³

First, we describe the technical setup used in our case studies. Both our Kubernetes and Nomad clusters consist of three worker nodes and one node acting as the control plane. In the case of our self-hosted environment, the CO frameworks run on bare-metal servers of type HPE ProLiant DL360 Gen9 with Intel Xeon CPU E5-2650 v4 and 16 GiB DDR4 RAM and Ubuntu 18.04.6 LTS as the OS. In the Google Cloud, we

¹³<https://doi.org/10.24433/CO.8875394.v3>

use VM instances of type `e2-standard-8` with Ubuntu 22.04 LTS placed in one common compute zone. The COFFEE controller, result databases, and load generators run on independent machines outside the test cluster to avoid interferences. We use Kubernetes version v1.24.4 in the self-hosted setup and the equivalent v1.24.4-gke.800 in GKE, both with containerd v1.6.6 as the container engine. For container networking, we use Flannel v0.19.2 as the networking plugin in our self-hosted environment, while GKE provides a built-in networking solution. For Nomad, we use version v1.4.1 with Docker v20.10.18 and Consul v1.13.2. Nomad also requires the manual deployment of a load balancer. For this, we use HAProxy v2.6.2. Note that by the date of publication, Google Cloud does not provide a managed Nomad service. Therefore, it is necessary to deploy it manually on a set of Google Cloud virtual machines. We provide the scripts used to configure our test nodes in our GitHub repository.⁹

This section focuses on three core requirements for CO frameworks: container starts, container removals, and inter-container networking. First, we examine readiness and removal times in Kubernetes and Nomad running in our self-hosted environment and on GCP. We consider the number of containers n that should be started simultaneously as an additional degree of freedom. In both Kubernetes and Nomad, declarative methods are used for deployments (Kubernetes) and tasks (Nomad) to specify the desired number of running replicas. A warmup run ensures that the test image is already present on all cluster nodes; therefore, the image pull time does not matter in the experiments. The experiment is relatively simple: n containers are deployed, and after 30 seconds, n containers are removed again. Between two repetitions, there is also a delay of 30 seconds. We repeat the experiment 100 times in all test environments.

Figure 4.3 shows the average readiness time per container for the four test systems. The number of containers to be started is plotted on the horizontal axis. The error bars indicate the 95 percent confidence intervals in this and all the following figures. Figure 4.3 shows that regardless of the CO framework, the self-hosted environment with the older technology stack performs worse than the virtual machines hosted on GCP. This can be seen in the pairwise comparison between the red and green, as well as blue and purple bars. Furthermore, the average readiness time per container increases with the number of desired containers. This is due to queuing effects during the sequential scheduling process. The only anomalies are the two Nomad test systems, which show high readiness times with high variability for low replica numbers. Since this effect repeatedly occurs in both test environments, it is specific to the CO framework, not the environment. When looking at logging messages in the Nomad test clusters, we noticed that the pure container start time (the time the container engine took to start the test container, see Chapter 5) was comparable to Kubernetes. Therefore, the root cause for the increased readiness time lies before the container starts, which is most likely an increased processing time at the cluster control plane. The investigation of the readiness times shows that COFFEE can investigate differences between CO frameworks (Use Case 1 from Section 4.1) as well as differences between two test clusters (Use Case 3).

In contrast to the readiness times, no clear trends can be seen in the average removal times per container shown in Figure 4.4. Looking at all four test systems, there is no consistent correlation between the number of containers to remove and the removal

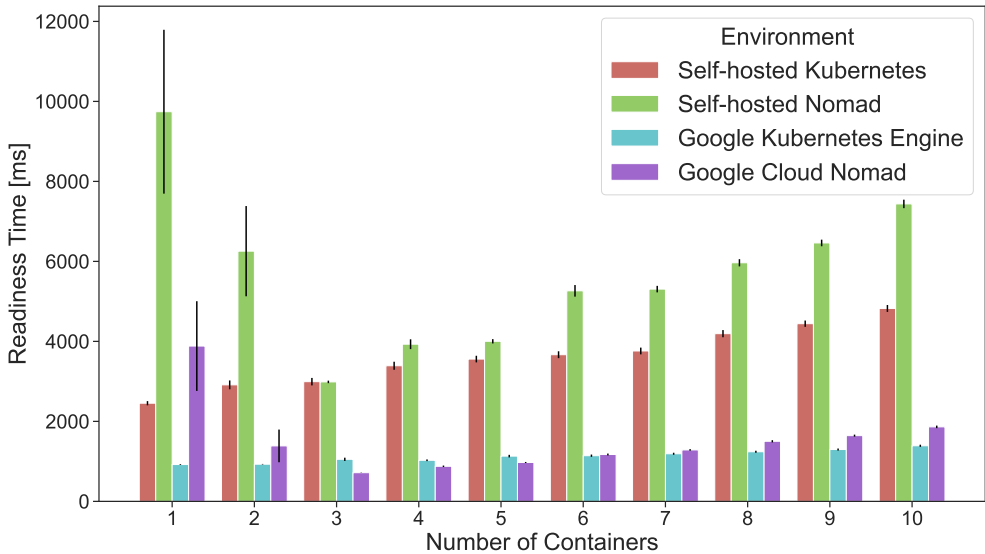


Figure 4.3: Container readiness time for different CO frameworks and test clusters.

time. However, there are still two takeaways. First, the removal time is significantly lower than the readiness times, which is in line with our expectations. Furthermore, we see that self-hosted Nomad performs worst among the test systems.

As a second baseline test for our clusters, we look at the performance of the in-cluster networking. We deploy one test container per worker node (3 in total). After a 30-second pause, requests are exchanged between all containers. Thereby, each instance acts as both sender and receiver. This creates six evaluated routes in total. As a performance metric, we consider the round-trip time of the messages. After the end of the network tests, the three test instances are shut down again. The experiment is repeated 100 times, with a 30-second pause between the runs. As an additional test system, we created a multi-region Kubernetes cluster in the Google Cloud. One node is located in the compute region us-west1 (Oregon, USA), one in europe-west2 (London), and one in asia-southeast1 (Singapore). The node specification and software stack are the same as described for the other Google Cloud virtual machines. We expect to measure a significantly higher round-trip time than in the self-hosted environment and the single-region GKE and Google Cloud Nomad clusters.

Table 4.2 shows all test systems' average round-trip times and confidence intervals. Since the round-trip time does not vary significantly between different routes for our four standard test systems, all routes are included in the average calculation, 600 measured values overall. It can be seen that the round-trip time in the self-hosted and the Google Cloud Nomad clusters are nearly equal. We observe slightly higher values in the case of the GKE cluster. However, it should be noted that the networking metrics are subject to measurement errors caused by interfering network traffic and imperfect time synchronization between the hosts, as discussed in Section 4.9.

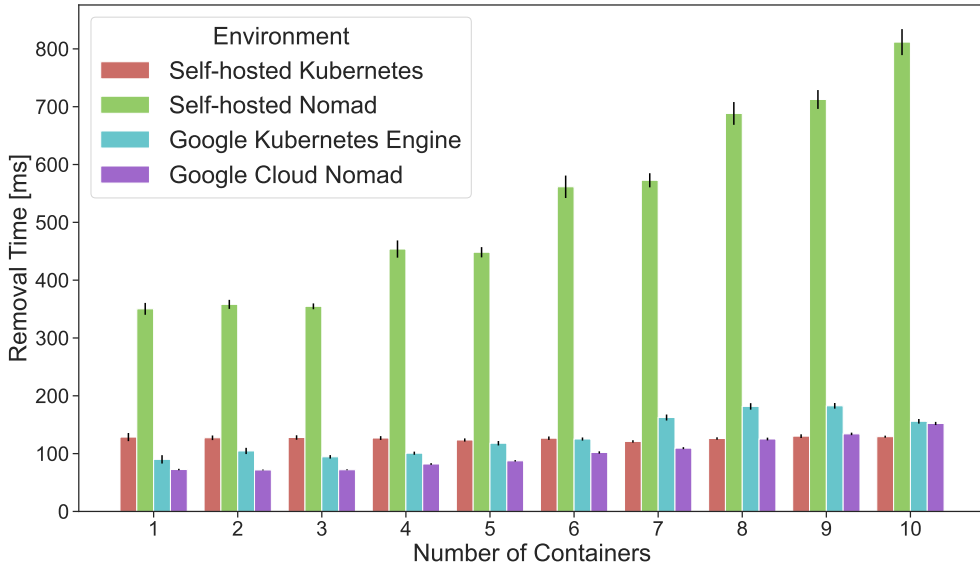


Figure 4.4: Container removal time for different CO frameworks and test clusters.

Test system	Average round-trip time [ms]
Self-hosted Kubernetes	15.07 ± 0.60
Google Kubernetes Engine	24.63 ± 3.46
Self-hosted Nomad	15.17 ± 0.52
Google Cloud Nomad	14.95 ± 0.46
Multi-Region Cluster	396.52 ± 11.47

Table 4.2: Average round-trip times between two nodes.

Furthermore, our hypothesis that the round-trip times in the multi-regional cluster are significantly higher than in the other systems is confirmed. Table 4.2 reports the average value over all routes, but we also measure significant differences between the routes in this case. For example, the Oregon-London route was the fastest, with an average round-trip time of 267.88 ms, followed by the Oregon-Singapore route with 326.62 ms. The slowest route was London-Singapore, with a 595.07 ms average round-trip time. In summary, we demonstrated that our approach is able to detect differences in network performance. Further interesting investigations could be the differences between different container networking solutions, similar to the works of Bankston and Guo [BG18] and Zeng et al. [ZWDZ17].

4.6 Case Study II: Failure Recovery

After the baseline tests from the previous section, we shift our focus to more complex experiments. In this section, we examine how fast our test systems can detect and react to the failure of several containers. We have already introduced the test procedure as an example in Listing 4.1. In summary, ten containers are started and stressed with 50 requests per second using the HTTP Load generator. After a warmup period of two minutes, all containers are abruptly terminated using the `CRASH` command. Consequently, the container process exits with a non-zero code. In parallel, load requests are sent at the same rate. We repeated the experiment 100 times. In all test environments, HTTP-based container health checks are configured in intervals of ten seconds. The expectation is that after detecting the container crashes, all ten containers will be restarted. Requests should fail for a short time as all containers crash almost simultaneously.

We consider the crash recovery time, that is, the time from the crash of the application until all ten instances are restarted and ready, as a performance metric. Furthermore, we look at the total number of failed user requests reported by the load generator. Figure 4.5 shows our measurement results. We see that the number of failed requests correlates strongly with the crash recovery time in all four test systems. This is expected due to the constant request rate. Furthermore, we see that Kubernetes has lower crash recovery times than the Nomad systems in pairwise comparison within the same environment and in absolute comparison.

The order of magnitude of the measured values aligns with our expectations, considering a simplified model. Let us assume that the time of the crash and the time of the next HTTP-based health check are independent. With a deterministic health check interval of ten seconds, the expected time until the next health check is five seconds. Suppose we add the removal and readiness time from Section 4.5 for ten parallel starting containers and a little overhead to this value. In that case, the result corresponds approximately to the observed values here. Assuming that the health checks work reliably, the readiness time of the containers is again one of the most critical performance factors. Our methodology, as seen from the confidence intervals, also ensures good repeatability of the experiments containing container crashes.

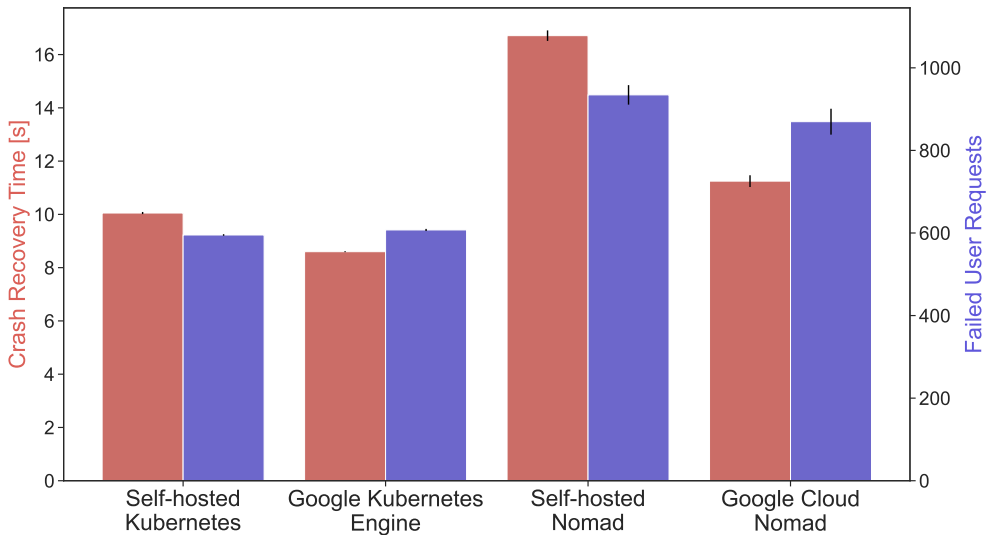


Figure 4.5: Failure recovery metrics for different CO frameworks and test clusters.

4.7 Case Study III: Rolling Updates and Load Balancing

This section examines performance metrics related to rolling updates and load balancing. The experiment setup is similar to the previous experiment. Ten containers are started and stressed with a request load of 50 requests per second (rps). After a 2-minute warmup phase, all containers are updated to a new version; that is, the image is changed. To prevent downtimes by terminating all containers at the same time, both Nomad and Kubernetes offer settings for rolling updates. Kubernetes provides two key parameters. First, `maxUnavailable` specifies the percentage of running replicas that can be down at the same time during the update process. Second, `maxSurge` specifies how many additional containers can be started during an update relative to the number of desired replicas. We set `maxSurge` and `maxUnavailable` to 25%, meaning between 8 and 12 pods are deployed during an update. In Nomad, one can set the number of containers that can be updated simultaneously (absolute number). In our experiments, we set this parameter named `maxParallel` to 2, similar to the `maxUnavailable` setting in Kubernetes. In our experiments, a warmup run ensures that both the original and updated images are already present on all nodes; that is, no image pull takes place.

We consider two performance metrics for rolling updates. First is the update time, which is the time it takes until all ten test containers are updated and ready. Second, we look at the number of failed user requests, similar to Section 4.6. Figure 4.6 shows our measurement results for the four test systems. Similar to the previous experiments, the Google Cloud virtual machines perform significantly better considering pairwise comparison with the self-hosted environment. We see an excellent update performance

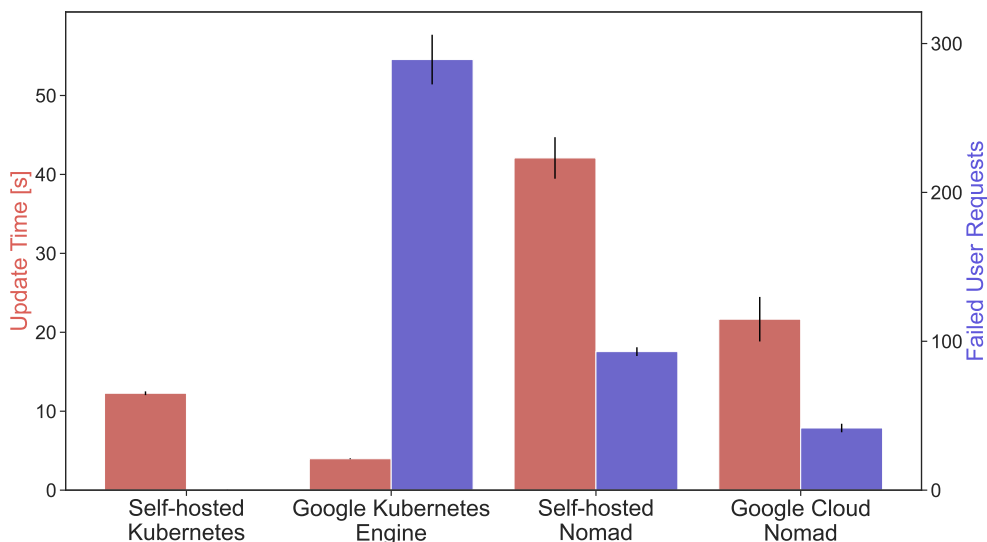


Figure 4.6: Rolling update metrics for different CO frameworks and test clusters.

in the self-hosted Kubernetes environment with an update time of 10 to 15 seconds and almost no failed requests (average below 1). Google Kubernetes Engine offers the fastest update with an update time well below 10 seconds, but up to 300 user requests fail, corresponding to a downtime of about 6 seconds. The total update times for Nomad are significantly higher than for Kubernetes. Furthermore, a small but measurable number of requests fail for those test systems.

The results obtained require a more detailed analysis. Figure 4.7 shows two exemplary runs for Google Kubernetes Engine and Google Cloud Nomad. The number of failed requests over time is shown in blue. In addition, we depict the number of ready instances, that is, instances registered in the COFFEE controller, in red. If the number of ready instances decreases, a container of the old version is shut down. If the number of ready instances increases, a container with the new image is deployed. In Figure 4.7a, we see that the update for the Google Kubernetes Engine is very fast. The time between the first event (the removal of the first container) and the last event (the removal of the last container) is about four seconds. Furthermore, we see that requests fail mainly towards or after the end of the update process, that is, when all old containers have been terminated. This can be explained by the fact that some requests are still routed to IP addresses that are no longer occupied. We further see that there are always at least 8, but at some points, more than the expected 12 containers deployed and ready. There are two possible causes for this unexpected behavior: First, the Kubernetes documentation states that terminating pods do not belong to the available quantity and, therefore, more pods than expected can be deployed during the update.¹⁴ Second, it can also be a matter of measurement errors, as the events

¹⁴<https://kubernetes.io/docs/concepts/workloads/controllers/deployment>

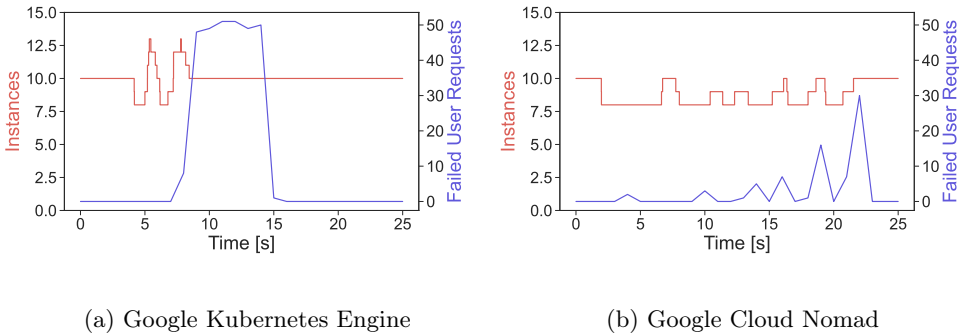


Figure 4.7: Detailed view on rolling updates for exemplary runs.

Test system	Avg. load rate [rps]	Std. Dev.
Self-hosted Kubernetes	4.567	1.777
Google Kubernetes Engine	4.565	1.775
Self-hosted Nomad	4.624	0.352
Google Cloud Nomad	4.431	0.618

Table 4.3: Received user requests per second and instance.

are separated by only a few milliseconds.

In Figure 4.7b, we see that the update process in the Nomad system takes significantly longer than in the Kubernetes system. According to our setting of the *maxParallel* parameter, between 8 and 10 containers are deployed at any time. Similar to the Kubernetes system, most request failures occur toward the end of the update process. However, the total number of failed requests is significantly lower. Our investigations motivate operators to set minimum healthy times/cooldown times for rolling updates to ensure that sufficient containers are always accessible and routing tables can be updated.

In addition, we look at the empirical load distribution over individual instances to quantify the load balancing behavior in our systems under test. Table 4.3 shows the average number of requests an instance receives per second and its standard deviation. The numbers have been measured in the rolling update experiments. However, they have also been confirmed in the context of the failure recovery experiments from Section 4.6. The average request rate is calculated as follows: Every test container counts the number of user requests it receives. In addition, it captures both its start and termination time. The difference between termination and start time is the total instance runtime. To calculate the request rate, we divide the total received requests by the instance runtime.

We see significant differences between Kubernetes and Nomad in terms of load balancing. Kubernetes distributes requests much more inequally among instances than

Nomad. This indicates a different load-balancing algorithm. Nomad is much closer to an equal distribution with a significantly lower standard deviation. There is no significant difference between different cluster environments. This shows that COFFEE can capture empirical results for load balancing metrics. This enables investigations of load balancers that are not open-source, such as the Google Cloud Load Balancer. Furthermore, one can compare different load balancers, for example, as offered by Nomad.¹⁵ As mentioned in Section 4.5, we use HAProxy load balancing for our Nomad test systems. Other open-source load balancers might be future benchmarking targets.

4.8 Case Study IV: Container Storage

Our fourth and final case study shows how COFFEE can be used to evaluate different configurations of a single CO framework. This corresponds to Use Case 2 from Section 4.1. Specifically, we evaluate different persistent volume storage classes in Kubernetes. We again use both a GKE cluster and our self-hosted environment and thus also address Use Case 3 from Section 4.1. We have not carried out a test with the orchestration framework Nomad, as the provision of persistent volumes must occur before the cluster is started. Hence, COFFEE cannot create these volumes dynamically, and the comparability to Kubernetes is no longer given. In our test scenario, a test container is deployed in the cluster. This instance then executes the `STORAGE` command, which writes randomly generated UUID strings to a file in the `/var/log` directory. This path is mounted to a persistent volume. We use different storage classes (i.e., technologies that provide persistent volumes). To do this, COFFEE creates a `PersistentVolumeClaim` with the corresponding storage class when creating the test container deployment. The persistent volume is then provided dynamically. At the end of the test campaign, the test container is terminated again. The experiment is repeated 30 times for each storage class, each repetition comprising 30 read-and-write tests, so a total of 900 measured values per storage class are recorded for the I/O metrics.

We are interested in two types of metrics. First, we want to measure I/O throughput values and are interested in possible differences between the storage classes and the test environments. In the GKE cluster, virtual machines are used as nodes, while the nodes in our self-hosted environment are bare-metal servers. This means that our self-hosted environment has one less virtualization layer. Secondly, we want to evaluate the effect on the container readiness time. We expect a higher readiness time due to the provision of the persistent volume.

We test six storage configurations in GKE and two in our self-hosted environment. We consider the *no persistence* case as a baseline. In this case, no persistent volume is attached to the test container, meaning that the `STORAGE` command writes data directly to the temporary file system of the container, which is lost when the container is deleted. In the storage class *local-storage*, a host path on the node where the container is currently deployed is used as a persistent volume. In GKE, there are a

¹⁵<https://developer.hashicorp.com/nomad/tutorials/load-balancing/load-balancing>

Storage Class	Read Throughput [kB/s]	Write Throughput [kB/s]	Initial Read [kB/s]	Initial Write [kB/s]
Google Kubernetes Engine				
No persistence	1372.83 ± 263.03	443.55 ± 86.65	545.97 ± 109.19	127.31 ± 48.52
local-storage	1324.72 ± 237.65	454.15 ± 64.08	553.85 ± 101.31	120.01 ± 44.01
default	1393.80 ± 276.52	463.21 ± 91.19	570.59 ± 117.85	131.50 ± 58.49
pd-csi-ssd	1396.51 ± 264.59	468.83 ± 95.68	552.73 ± 100.53	114.51 ± 52.61
pd-csi-standard	1374.84 ± 264.68	444.02 ± 98.03	562.08 ± 111.72	110.33 ± 39.56
filestore-standard	1243.52 ± 291.99	442.88 ± 108.78	488.39 ± 125.29	91.27 ± 27.19
Self-hosted environment				
No persistence	2166.82 ± 507.56	704.45 ± 114.83	852.93 ± 146.05	197.08 ± 56.81
local-storage	2078.86 ± 488.84	641.62 ± 109.90	871.07 ± 124.10	174.12 ± 61.94

Table 4.4: I/O throughput for different storage classes and environments.

number of alternatives available for storage classes.¹⁶ With the *default* storage class, a balanced persistent disk type (ext4) is created. Furthermore, we consider different configurations of the Compute Engine persistent disk container storage interface driver. We define the storage classes *pd-csi-ssd* (SSD-backed) and *pd-csi-standard* (HDD-backed) according to the documentation.¹⁷ As the last storage class we consider Filestore,¹⁸ a provisioner of network file system (NFS) servers for the Google Cloud. Specifically, we look at the standard read-write variant where a basic HDD tier is used as storage.

Table 4.4 shows the I/O throughput measurements with their standard deviations for the different storage classes and cluster environments. During the measurements, we found that a test container’s first read and write operations were significantly slower than the others. Therefore, we report these values separately as *initial read* and *initial write*. It can be seen that the values within a cluster environment barely differ. This applies to both the average value and the order of magnitude of the variance. It can also be seen that the self-hosted environment generally achieves higher read throughput and slightly better write throughput. The lack of the additional virtualization layer in the Google Cloud can explain this. It should be noted that COFFEE does not measure the maximum I/O throughput values; it only performs sequential reads and writes. We opted for this variant because sequential reads and writes are more common in most microservices and function container use cases due to the event-driven triggers. In addition, the CPU is usually the bottleneck in containers with limited resources rather than I/O. Established tools such as `hdparm` or advanced benchmarks such as SPECstorage¹⁹ can be used to measure storage performance for parallel reads and writes.

¹⁶<https://cloud.google.com/kubernetes-engine/docs/concepts/persistent-volumes>

¹⁷<https://cloud.google.com/kubernetes-engine/docs/how-to/persistent-volumes/gce-pd-csi-driver>

¹⁸<https://cloud.google.com/filestore/docs>

¹⁹<https://www.spec.org/storage2020>

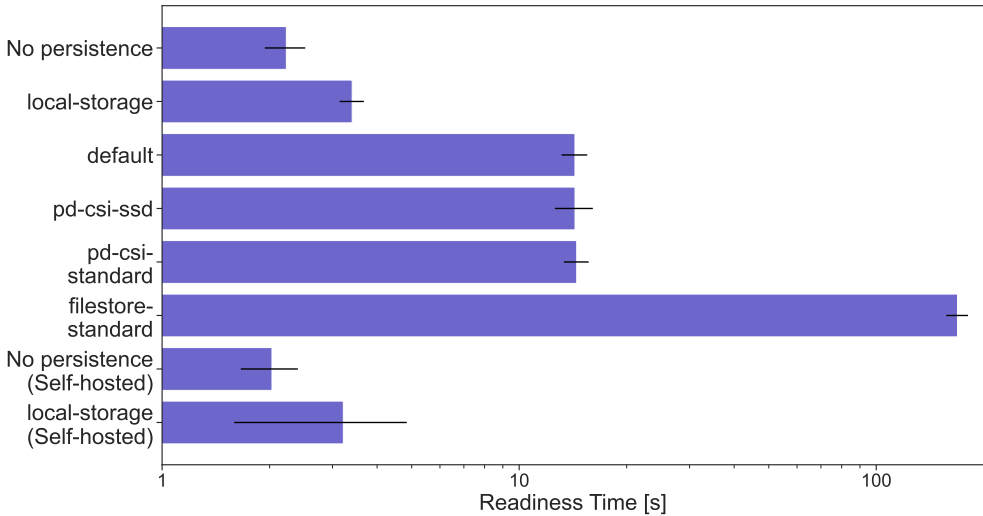


Figure 4.8: Readiness times for different storage classes and environments.

Another question is how the storage classes influence the readiness time of the test container. Figure 4.8 shows the readiness time in seconds of the respective storage classes on a logarithmic scale. We can see the expected result that the baseline without persistent volumes has the lowest readiness time. The second fastest is the local storage, where a volume is created on the host node of the container. In the Google Cloud, Compute Engine persistent disks and the default storage class are about the same, but the readiness time increases by about 10 seconds when there is a persistent volume. The NFS server Filestore is an order of magnitude slower, with a readiness time of over 100 seconds. From this, it can be concluded that Filestore should be used for long-running containers without horizontal autoscaling. Local storage outperforms the classic persistent volumes in the Google Cloud, but it should be noted that in the event of a node crash, data may be lost, and Kubernetes cannot recover the volume.

This case study illustrates the interdependencies of the individual core tasks of CO frameworks well. COFFEE's design is intended to collect as many generic metrics as possible in order to cover the variety of container orchestration tasks. In this case study, we were able to show that COFFEE is also suitable for evaluating different configurations of a single CO framework (in this case, storage solutions for Kubernetes). Due to the currently dominant market position of Kubernetes, this use case could be the most interesting in the near future.

4.9 Summary and Discussion

In this section, we give a summary of our main findings from this chapter, point out strengths, weaknesses, and limitations, and discuss the contribution to the overall

goals of this thesis. In this chapter, we have presented an approach to benchmarking of CO frameworks that allows us to support three benchmarking use cases: the comparison of different CO frameworks, the comparison of different configurations of one CO framework, and the comparison of different cluster environments. To develop our approach, we first assembled a list of tasks associated with the term container orchestration by analyzing scientific literature and non-scientific sources. Based on these tasks, we derived core requirements and associated performance metrics for CO frameworks. In the next step, we created a high-level design for a benchmarking framework that is able to measure the proposed performance metrics without making limiting assumptions or being constrained to a specific CO framework. COFFEE is our implementation of this design and allows us to create simple sequential as well as complex trace-like test campaigns with a set of pre-defined commands.

We then demonstrated in a total of four case studies that COFFEE is able to capture (i) relevant performance metrics for CO frameworks, (ii) address the large number of CO tasks operating in parallel and (iii) cover the defined benchmarking use cases. We could detect performance differences for the same orchestration framework running in different environments (self-hosted vs. Google Cloud), as well as for different orchestration frameworks running in the same environment (Kubernetes vs. Nomad). Our results indicate that Kubernetes outperforms Nomad in many scenarios, but more experiments in different environments are needed to solidify this statement. Furthermore, using the example of container storage provisioners for Kubernetes, we were able to show that COFFEE can also evaluate different configurations of a single orchestration framework.

This contribution presents the first benchmarking approach for container orchestration frameworks that covers the multitude of CO tasks and has been tested with multiple CO frameworks. This stands in contrast to state-of-the-art approaches that either focus on single CO tasks or on the analysis of one specific CO framework and technology stack. We showed in our case studies that these limitations are severe because clear interdependencies exist between CO tasks. This is shown, for example, between failure recovery and container readiness times in Section 4.6. Consequently, we argue that a benchmarking framework for container orchestration should support a holistic evaluation instead of isolated tests of selected CO tasks. In this regard, our approach provides a valuable advancement in the research area of container orchestration frameworks. Another strength of our approach, and especially of our developed framework COFFEE, is the ease of use. The user-friendly format for specifying test campaigns allows users to define and execute their own campaigns quickly. The configuration of COFFEE follows the approach that only a few necessary parameters have to be set (e.g., the IP address of the cluster control plane), but many more fine-grained parameters can be set (e.g., detailed parameters for rolling updates). This makes it possible to use COFFEE in different ways in practice. Simple test campaigns can be evaluated with the default cluster settings (e.g., to initially evaluate a CO framework on some specific infrastructure). More complex test campaigns can be used to fine-tune the environment for specific use cases and workloads.

Another strength of our approach is its extensibility. Users can easily add new commands to COFFEE's test campaigns. If the desired operation can be performed

orchestrator-agnostically, then only an extension of the test container (e.g., by adding a new endpoint) is necessary. The established proxy-based communication of COFFEE can then be used to submit the new command. New orchestrator-specific commands can also be introduced. COFFEE uses maintained client libraries for Kubernetes and Nomad, which offer numerous functions for communication with the cluster control plane. The extension of COFFEE to additional CO frameworks is more complex. Here, orchestrator-specific commands (`START`, `REMOVE`, `RESTART`, and `UPDATE`) and some auxiliary functions, such as authentication or the retrieval of container or node names, must first be implemented. The effort for this cannot be determined across the board and depends, for example, on whether a suitable client library exists for the targeted CO framework.

This leads to the limitations and open challenges of this work. Our approach and the COFFEE framework provide a basis for future work and numerous case studies in the area of benchmarking of CO frameworks. However, our work does not yet represent a complete benchmark as defined by the latest empirical standards [Has21]. The main missing points are representative workloads for container orchestration frameworks. In the context of COFFEE, this means we need realistic test campaigns. Our case studies cover the essential functions of CO frameworks. More enhanced studies would be enabled if more data from production workloads of CO frameworks were available.

One aspect we have not yet evaluated in this chapter is the size of the test cluster and the scalability of our approach. The influence of the number of nodes on metrics like the container readiness time or load distribution is still to be investigated. We reduced the number and size of requests to the COFFEE controller as far as possible in the design phase. However, proof is still needed that COFFEE can handle complex test scripts in large test clusters. Further optimization or redundancy of the COFFEE controller might be necessary. Production-sized cluster environments were not available for experimentation in this thesis. Another current limitation is that only one type of test container can be deployed in the cluster. In future work, one could introduce different types of testing containers (e.g., with different resource requirements) to evaluate the impact on readiness times. The results of our case studies should not be interpreted as official benchmarking results that give usage guidelines for production use cases. While COFFEE is a tool that could be used for such investigations, we tested only a few configurations of Kubernetes and Nomad in this thesis and left a large portion of their configuration space unexplored. Changing the configuration of the CO frameworks might change the benchmarking results presented in this study, which is why our results from the case studies cannot be generalized. The case studies aimed to show the usage of COFFEE with different orchestration frameworks, configurations, and test environments to underline our approach's broad use cases.

Our case studies have not yet covered all the core tasks of CO frameworks, especially autoscaling and placement. COFFEE can track placement decisions and also supports autoscaling experiments by supporting dynamic loads or reconfiguring the load endpoint. These two tasks, in particular, have the characteristic that they are commonly evaluated with application-specific metrics like response times and do not offer directly interpretable and application-agnostic metrics. For these purposes, specialized

frameworks such as Theodolite [HH22] for scaling might be preferred. Moreover, we address autoscaling and placement in Kubernetes later in this thesis in Chapters 6 and 7. COFFEE’s functionalities could be tuned towards specific environments or CO frameworks by including system-level metrics and orchestrator-specific data, such as log data. This allows the tracking of fine-grained metrics, such as the runtime of load balancing algorithms [YKXY19]. However, including more metrics raises the question of how to aggregate these metrics (e.g., to a total score for the whole CO framework). More measurements with more frameworks and in different environments must be conducted to propose such an overall score. In this chapter, we show the usage of COFFEE with its minimum requirements to make clear that our approach does not rely on advanced monitoring capabilities and the presence of system-level or orchestrator-specific metrics.

We reduce statistical errors by repeating all measurements at least 30 and up to 900 times. However, as known from previous studies [PFS19, LC16], measurements in public clouds are always subject to variability, and results in other settings, for instance, other compute regions, might be different. More empirical results must be collected to allow further claims on the performance of different cloud environments. Note that we selected nodes with more computing resources than required to avoid hardware bottlenecks. In test runs of our workloads, CPU and memory utilization of all nodes stayed far below 50%. In our case, the most significant source of uncertainty is the time synchronization between the nodes. Some operations rely on timestamps from different machines. This influence is especially significant for network measurements. We use network time protocol-based (NTP) synchronization between hosts in all test clusters. Better synchronization methods like precision time protocol (PTP) are not supported on Google Cloud virtual machines as they do not fulfill specific network interface requirements.

All in all, this chapter presented a systematic approach for benchmarking container orchestration frameworks. Our approach allows a comprehensive performance assessment of container orchestration frameworks. The results of benchmarking runs generate data that can serve as a basis for decisions on the selection and configuration of container orchestration frameworks. We have thus made an important contribution that addresses Challenge 1 of this thesis.

We answered RQ A1 ("Which features of container orchestration frameworks are feasible to evaluate through benchmarking?") by identifying eight key features of container orchestration frameworks based on a review of common definitions. Based on these features, we derived testable requirements, which were annotated with performance metrics. We defined a design consisting of a controller, load generator, proxy, and test containers that can capture the defined metrics. This addresses RQ A2 ("Which metrics can be used to quantify the performance, and how can they be measured?"). Based on this preliminary work, we implemented our benchmarking framework COFFEE, our answer to RQ A3 ("How to define a benchmark for container orchestration frameworks and enable execution on diverse technology stacks?"). COFFEE places minimal requirements on the system under test and uses a user-friendly scripting language to define complex benchmarks.

Chapter 5

An Empirical Study on Factors Impacting Container Start Times

Containers have become the predominant deployment method for modern cloud applications and a key enabler of the wide adoption of microservice architectures. They are now used in all areas of distributed computing, from traditional cloud to edge and serverless computing. Docker containers are a leading solution for containerization; the popular image repository Docker Hub, with over nine million accounts, has over 42 billion image downloads every quarter.¹ The container ecosystem gets even bigger if we include other registries like GitHub Packages² or further container managers like Podman.

One regularly mentioned advantage of containers over other virtualization technologies, like virtual machines, and an important driver of adoption is their faster start times. Understanding start times is essential for many application areas. For example, the fact that containers permit start times of a few seconds enabled the broad usage of serverless computing. However, start times remain an active research field in this domain as they are critical factors for desired rapid scale-ups [LWC⁺23, LGC⁺22]. Likewise, the optimization of container deployment processes also plays an important role in edge computing [HSL22, AP18]. Start times also play a significant role throughout this thesis. In the previous chapter, we analyzed readiness times for different container orchestration frameworks. Start time is a part of the readiness time, as discussed later in this chapter. Also, in the context of continuous decentralized autoscaling, which is introduced in Chapter 6, start times heavily influence the autoscaler’s behavior. Overall, container starts are crucial in modern cloud environments and a critical performance factor for cloud, serverless, and edge applications.

While the fact that containers have faster start times than virtual machines is well-established in the scientific literature [XFJ16, TKT18, SFSF19], relatively little is known about variations in start times between different containers. Understanding the factors that influence container start times may help developers build containers with fast start times and define rules and best practices for creating such lightweight containers. No prior work has examined start time variations across a wide variety of containers to make statements about influencing factors based on a large number of measurements.

¹<https://www.docker.com/blog/docker-index-shows-surging-momentum-in-developer-community-activity-again>

²<https://github.com/features/packages>

This chapter presents an analysis of how various container image and host characteristics influence a container’s start time. For this, we use about 200,000 open-source images retrieved from Docker Hub by a web crawler. This dataset mixes the most popular and recent Linux images queried in April 2022. We extract various properties of these images from their configuration, determine the empirical distributions of those properties, and analyze their variance using principal component analysis. We then measure start times for a subset of these images in two test environments: the Google Cloud Platform and a local testbed. We apply regression analysis techniques to determine the relevance of each image configuration parameter in predicting start times. We find that start times can vary between hundreds of milliseconds and tens of seconds in the same environment. However, we also show that these variations cannot be explained by looking at single configuration parameters and that hardware and software parameters must be considered together for an accurate assessment. We further underline this statement with selected experiments using different host configurations.

This chapter addresses Challenge 2 from the overall context of this thesis. Our contribution provides quantitative data on container images and their start times. Our analysis increases the understanding of the factors impacting container start times and thus explains differences between container images. To achieve Goal B of this thesis, we address the following guiding research questions in this chapter:

- RQ B1: How to acquire a representative dataset of container images and their start times?
- RQ B2: Which factors impact container start times?

The remainder of this chapter is structured as follows: In Section 5.1, we provide background information on container start times and clarify the scope of this study. Section 5.2 focuses on our container image dataset, its characteristics, and how it has been acquired. Section 5.3 features our methodology for the start time analysis. Section 5.4 analyzes the variability of container start times, while Section 5.5 focuses on the influence of image configuration parameters on start times. Section 5.6 focuses on the influence of different platform parameters (i.e., host configurations) on container start times. In Section 5.7, we discuss the overall findings and limitations of this contribution and discuss its role in the overall vision of this thesis. We published parts of this chapter, including text paragraphs, figures, and tables, as a full research paper at the 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) [SBL⁺23] and in a workshop paper at the 15th Symposium on Software Performance (SSP) [SEK24].

5.1 Container Starts and Their Performance Metrics

Different components in a layered architecture play a role in the execution of containers (see also Section 2.2). We use Docker, one of the most common technology stacks, in the study presented in this chapter. Docker is based on `containerd`. As the actual container runtime, we use the OCI reference implementation `runc`. Section 3.2

discusses related work with other container managers and runtimes. In the following, we look in detail at the start process of a container according to the OCI runtime specification.³

Containers are a bundle of a root file system and a configuration. The container start process begins with the container creation. The OCI runtime receives a pointer to the bundle and a (within the host) unique identifier that the upper-layer container manager has set. The container is created based on the settings defined by its configuration. The configuration contains permissions, volume mounts, resource limits, devices, and more. After this step, a number of so-called hooks allow for customization of the creation process and can manipulate the runtime and container namespaces. As an example, network namespaces could be set up for Linux containers. The OCI runtime specification does not define more examples or best practices for these hooks. Once all hooks are executed, the container is created. This can be interpreted as the state when all initialization work of the low-level Linux namespaces is done. After the container creation, the actual start begins. Again, several pre-start hooks can be specified. The container runtime then executes the user-specified command in the container set in the `process` field of the image configuration. Once this command is invoked, the container start is finished, and the container itself transitions to a running state. In the following, we do not distinguish between the creation and start sub-steps and refer to the whole process as the start process.

In the following, we link performance metrics to the process of initializing and running a container. The end-to-end metric for quantifying the duration of this process is the *readiness time*, as shown in Figure 5.1. The readiness time is the time taken from when the container is downloaded from a registry until it can execute production workload; for web applications, this usually means processing user requests. This state of readiness is checked on container orchestration frameworks like Kubernetes by so-called readiness probes, a type of health check. These readiness probes can be test requests to the container or custom commands executed inside the container. The total readiness time consists of three main components. *Pull time*, the time required to transfer the image from a registry to the host, depends mainly on network parameters. Note that no image transfer is necessary if the required image is already present on the host. *Start time*, the focus of this chapter, is the duration of the start process defined by the OCI runtime specification that we introduced earlier in this section. It also includes the container creation. Potential factors influencing start time are, on the one hand, the underlying hardware and software stack and, on the other hand, image configuration parameters.

The start time is the timespan from receiving the start command to when the container starts running the user-specified process (also known as entrypoint). According to the OCI runtime specification, containers are content-agnostic, meaning that operations such as container starts and stops are executed in the same way regardless of the container's contents. Consequently, the start time can be evaluated without knowing the container's content and function. It can be determined even for containers that cannot actually perform their intended function. Consider an example container that

³<https://github.com/opencontainers/runtime-spec>

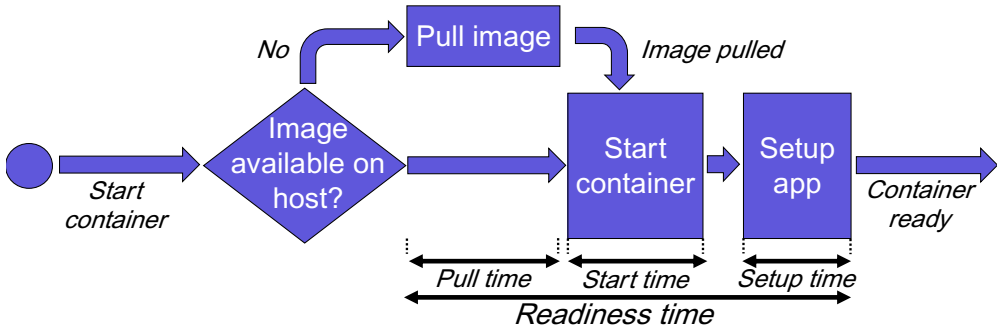


Figure 5.1: Container readiness process and performance metrics.

tries to connect to a database at an unreachable IP address. This container would never reach the ready state; instead, it would probably crash after a certain timeout. However, its start time can still be determined because the start process ends immediately when the entrypoint is executed. Therefore, the only condition for examining the start time is that the container is startable. We discuss exceptional cases for when a container is not startable in Section 5.2.

The start time is followed by the *setup time*, the time taken for a started container to reach the ready state. This timespan depends mainly on the user-specified process in the container, which means on the actual use case of the container. In the example of the preceding paragraph, a database connection must be established. In many application areas (e.g., serverless computing), a programming language runtime (e.g., Python) has to be initialized. Analysis of setup time, and thus the total readiness time, requires knowledge of container internals and external dependencies. Consequently, a generalization of start times to readiness times is not possible. An automated analysis of setup or readiness times is not possible, as these are, in contrast to the start time, not content-agnostic. A readiness probe must be executed to determine setup and readiness times. However, readiness probe or other health check templates are not included in the OCI image specification.⁴

5.2 The Docker Hub Dataset

This section presents the dataset used in the remainder of this chapter, describing first how we acquired the data and then various characteristics of the data. The dataset is available on Zenodo⁵ and in our replication package.⁶

To investigate variation in image configuration parameters and the influence of those parameters on start times, we first created a dataset containing a broad selection of container images. We choose Docker Hub as a widely used container image repository

⁴<https://github.com/opencontainers/image-spec>

⁵<https://doi.org/10.5281/zenodo.7602500>

⁶<https://doi.org/10.24433/CO.4595026.v2>

and use a web crawler (based on the Docker Hub Explore function⁷) to extract a large number of image names from this repository. Specifically, we first did a substring search on the Docker Hub "most popular" and "recently updated" categories with search strings up to a length of three characters with all letter combinations in the range a–z (i.e., from **a** to **zzz**). Then, we included the images displayed on the start page. In this way, we obtained a selection of both the most popular images (the most downloaded) and an unbiased set of less popular images. We included only Linux **amd64** architecture images in our dataset to ensure compatibility with our test machines. Both the Linux operating system and **amd64** architecture are widely used in modern public clouds. We queried the image data in April 2022.

After removing duplicates, we were left with 286,294 image names. The number of images we could crawl is limited by the fact that the Docker Hub Explore function limits the number of results per search request to 500. Before we go into more detail about further filtering steps and the characteristics of the dataset, we review the general structure of a container image. According to the OCI image specification,⁴ a container image consists of a manifest, the file system layers, and a configuration. In this work, the configuration plays a major role, which contains information about exposed ports, volumes, the start command, and more. The default configuration of an image is the configuration that the creator of the image has set. Docker can display the image configuration by the `inspect` command.

We attempted to download the associated image for each of the 286,294 crawled names, capture its configuration data (including unique identifier hash), and run it once. We eliminated images that could not be downloaded or were not startable from further consideration. These filtering steps left us with 200,986 valid and pullable images. The detailed reasons why images were dropped from the dataset were:

- The image's root file system was missing a dependency to invoke the configured user-specified process
- The image's root file system was invalid⁸
- The image's manifest was invalid⁸
- The image did not define a process to start⁹
- The image could not be downloaded as further authentication was required
- The size of the image's root file system was larger than 100 GB¹⁰
- The image has been renamed or removed in the time from when the web crawler ran and the time of the test downloads

⁷<https://hub.docker.com/explore>

⁸In most cases, the image is an OCI-compliant image, but some contents or configurations are not supported by Docker.

⁹The OCI runtime specification does not require image builders to set a start command in the image configuration. This is primarily used for helper images, such as build templates. Start commands then have to be specified manually before execution.

¹⁰This is a technical limitation of the test machines used in this study.

We characterize each image in the dataset with 20 features extracted from its default configuration. We downloaded all images and analyzed their OCI configuration properties to decide which features to include in our analysis. The OCI image configuration specification¹¹ defines required and optional configuration properties. The 20 resulting dataset features include required properties (e.g., `fs_root_fs_type`) and optional properties (e.g., `net_ports`) where a value has been set for all 200,986 images. Other optional properties were only set for a very small number of images and thus are not considered in this study. We provide a complete list and description of the extracted features in Table 5.1. The features' meaning was taken from the OCI image specification and the Docker documentation.¹² We consider five feature categories: metadata, I/O streams, networking, file system, and information about the start command. Selected features are further described in later sections if they were found to affect start time.

In the following, we present more details about selected features and their distributions. First, we focus on the Docker version (`meta_docker_version`), which indicates when the images were built/published most likely. Table 5.2 shows the number of times different Docker versions appear in our dataset. We also labeled each version with its release date.¹³ Images for which no Docker version is specified or cannot be unambiguously assigned are marked as N/A. We see that while most images are likely from the last five years, some older images are also included in the dataset. Note that version 20.10 was the latest version available at the time when this study was conducted. The next Docker version, 23.0, was released in 2023.

Figure 5.2 shows, with a logarithmic scale, the distributions of 10 other features in the dataset. The first four features in the upper row are the boolean settings mentioned in Table 5.1, considering the I/O settings of the container. We see that these features are all similarly distributed and that, in most cases, no streams are attached by default. In the upper right corner, we see the exposed ports (`net_ports`) as a networking feature. As expected, many containers expose only a few ports by default. However, the dataset also contains two containers that expose the maximum number of 65,536 ports. In the bottom row of Figure 5.2, we see on the left side the number of environment variables (`cmd_envvars`) and the size of the list of arguments (`cmd_args`) to use as the command to execute when the container starts. Like the exposed ports, we see the highest frequency at the value of zero but again with a relatively large value range in total.

The three features shown in the bottom right give quantitative information about the container file system. For the number of volumes (`fs_volumes`), we see a distribution with a maximum at zero and a sharply decreasing frequency. However, the dataset also contains two outliers with 21 and 32 volumes. We observe more complex distributions regarding the number of file system layers (`fs_layers`) and the image size (`fs_size`). The containers in our dataset have a minimum of 1, an average of 10.84, a median of 9, and a maximum of 125 root file system layers. The observed maximum is equal to

¹¹<https://github.com/opencontainers/image-spec/blob/main/config.md>

¹²<https://docs.docker.com/engine/reference/run>

¹³<https://docs.docker.com/engine/release-notes>

Feature	Type	Description
Metadata		
<code>meta_repo_digest</code>	String	A SHA-256 hash which is used to uniquely identify and download the image from Docker Hub
<code>meta_architecture</code>	String	The CPU architecture which the binaries in the image are built to run on
<code>meta_os</code>	String	The name of the operating system which the image is built to run on
<code>meta_docker_version</code>	String	The Docker version used to built this image
I/O Streams		
<code>io_attach_stdin</code>	Boolean	Determines whether the console should be attached to the container process' <code>stdin</code> stream
<code>io_attach_stdout</code>	Boolean	Determines whether the console should be attached to the container process' <code>stdout</code> stream
<code>io_attach_stderr</code>	Boolean	Determines whether the console should be attached to the container process' <code>stderr</code> stream
<code>io_tty</code>	Boolean	Determines whether the console should pretend to be a terminal (TTY) when attached
<code>io_open_std_in</code>	Boolean	Determines whether the container process' <code>stdin</code> stream should be open even if the console is not attached
<code>io_std_in_once</code>	Boolean	Determines whether the container process received input from <code>stdin</code> stream at least once
Start Command		
<code>cmd_args</code>	Integer	Length of the list of arguments to use as the command to execute when the container starts
<code>cmd_envvars</code>	Integer	Number of environment variables set per default when the container starts
<code>cmd_additional_args</code>	Integer	Length of the list for additional arguments to the container's entrypoint
File System		
<code>fs_volumes</code>	Integer	Number of volumes to create/use by default
<code>fs_size</code>	Integer	Size of the image in bytes
<code>fs_virtual_size</code>	Integer	Virtual size of the image in bytes
<code>fs_graph_driver_name</code>	String	Name of the image's graph driver
<code>fs_root_fs_type</code>	String	Name of the file system type used in the image
<code>fs_layers</code>	Integer	Number of root file system layers
Networking		
<code>net_ports</code>	Integer	Number of ports to expose by default

Table 5.1: Overview of features in the extracted dataset.

Version (M/Y)	No. of Images	Version (M/Y)	No. of Images
20.10 (12/2020)	19,647	1.7.x (06/2015)	78
19.03 (07/2019)	42,479	1.6.x (04/2015)	158
18.x (01/2018)	53,697	1.5.x (02/2015)	51
17.x (03/2017)	34,886	1.4.x (12/2014)	18
1.13.x (01/2017)	4821	1.3.x (10/2014)	31
1.12.x (07/2016)	14,089	1.2.x (08/2014)	11
1.11.x (04/2016)	3440	1.1.x (07/2014)	10
1.10.x (02/2016)	1196	1.0.x (06/2014)	19
1.9.x (11/2015)	340	0.x (03/2013)	7
1.8.x (08/2015)	205	N/A	25,803

Table 5.2: Frequencies of different Docker versions in dataset.

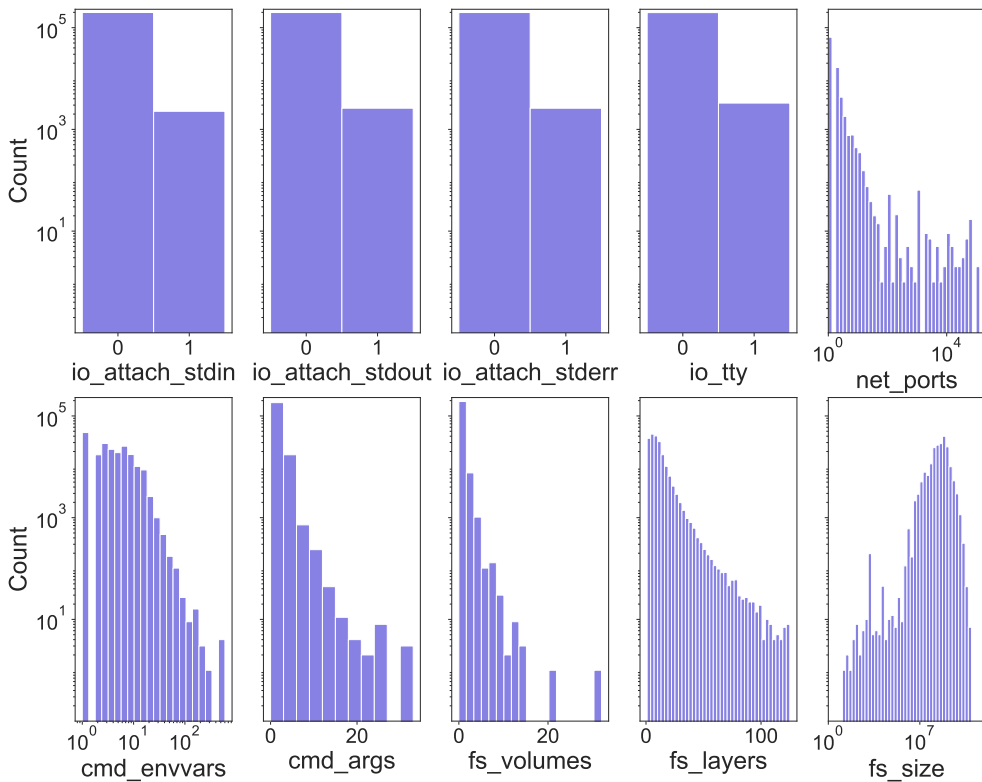


Figure 5.2: Empirical distributions of selected features.

the technical maximum,¹⁴ which means that our dataset represents the largest range possible. The image sizes show an even higher scatter. The minimum image size in our dataset is 45 bytes, the mean is 837.3 MB, and the median is 398.6 MB. The largest image has a size of about 90.4 GB. As mentioned earlier, we did not consider images with sizes larger than 100 GB due to restrictions on our test machines.

In summary, the dataset shows a good diversity across many features. In the following, we examine the total variance of the dataset in more detail using principal component analysis (PCA) [AW10]. PCA can be used to reduce the number of features in a dataset, for example, to apply machine learning algorithms more efficiently. In particular, constant and (nearly) linearly dependent features are identified. PCA calculates the eigenvectors of the covariance matrix, the so-called principal components. The principal components are linear combinations of the original features. By looking at the proportion of the variance of each principal component in the sum of the variances of all principal components (often referred to as *explained variance ratio*), one gets an idea of how many principal components are needed to represent a certain share of the total dataset variance. If only a few components are needed to capture a large proportion of the variance, the dataset contains many linearly dependent or constant features. In our context, we use PCA to characterize the diversity of our dataset further and to identify correlations between different features.

Figure 5.3 shows a cumulative view of the explained variance ratio. As preprocessing steps, we removed the metadata features and scaled the remaining data using z-score normalization. We see that the first principal component covers about 31% of the total variance and that, with ten components, about 99.9% of the total variance can be described. These numbers show that our dataset contains about six features with a negligible variance or that are linearly dependent on other features. All other features make some non-negligible contribution to the total variance, and their influence on the start time can be investigated meaningfully.

To summarize, the dataset presented in this section is the latest and one of the largest container image datasets. Comparable datasets (e.g., by Zhao et al. [ZTA⁺19]) were released before the broad adoption of containerized applications. Our dataset contains images with diverse characteristics, such as image size and number of exposed ports. Consequently, the dataset can be valuable in many use cases that require diverse images, such as optimizing image layers or image pulls [LLT⁺23, ZTA⁺21]. In the following, we use the dataset to analyze container start times.

5.3 Study Design

This section explains our methodology for using data from the presented dataset to determine the factors influencing container start times. As explained in Section 5.2, we analyzed 200,986 distinct images in this study. Our goal is to measure the container start time and then investigate the impact of the image configuration parameters as well as selected platform parameters. To this end, we studied how start times vary for different containers as well as for repeated starts of a given container image. We

¹⁴https://github.com/moby/moby/blob/master/layer/layer_store.go

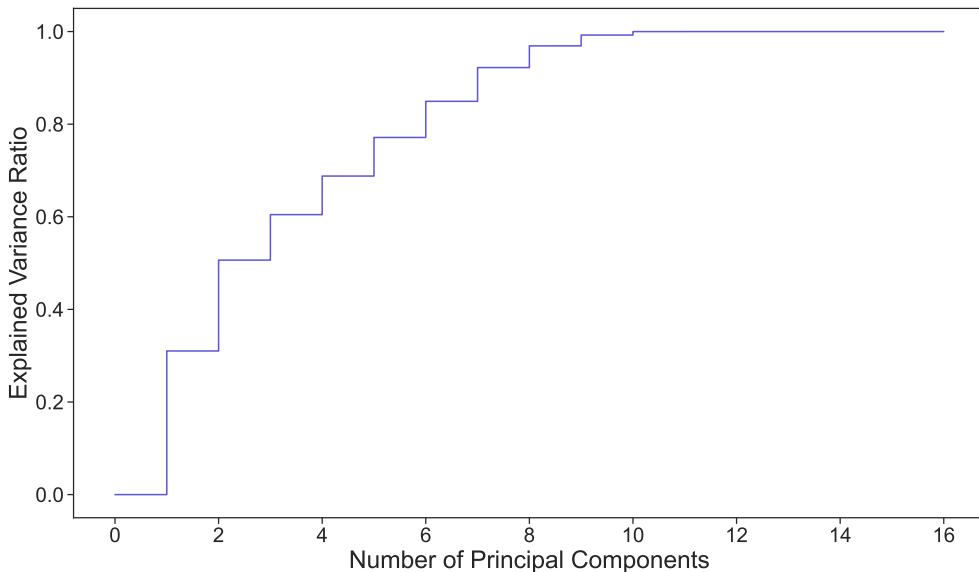


Figure 5.3: Cumulative explained variance ratio by principal components.

assume that the image configuration parameters, the hardware, and the software of the deployment environment potentially impact container start times. To account for this, we measure the start times in two test environments: on bare-metal servers in a self-hosted testbed and on VM instances on the Google Cloud Platform. Time, cost, and technical limitations prevented us from testing all 200,986 images multiple times. The main limitations are image download limits enforced by Docker Hub and the induced costs for the virtual machines. Consequently, we tested a sample of images from the entire dataset. In total, our measurement campaign spanned about three months. In the following, we describe our sampling and measurement processes.

Different sampling techniques can be applied to draw a sample from our dataset. We aim to select a sample that, on the one hand, reflects well the typical configurations (i.e., the most frequent image parameter combinations) but, on the other hand, also takes outliers into account (e.g., images of enormous size). As shown in Section 5.2, our dataset contains many image and parameter combinations. To cope with this heterogeneity, we applied stratified sampling [SSW03] instead of simple random sampling because the latter may fail to capture outliers and extreme values properly.

Figure 5.4 shows our sampling process. In stratified sampling, data is divided into so-called strata. The idea is to divide a dataset into subpopulations that are as homogeneous as possible. Random sampling is then used to draw a sample from each stratum. The combined strata samples form the total dataset sample. Accordingly, the first step in our sampling process is to divide the dataset into strata. For this, we evaluate different clustering algorithms and perform a hyperparameter study. Table 5.3 shows the evaluated clustering algorithms and the parameter search space. The

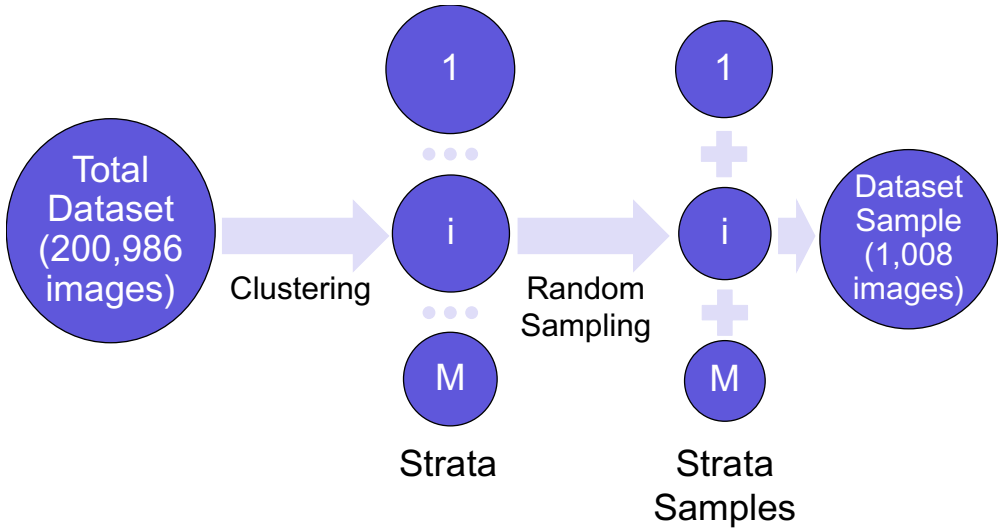


Figure 5.4: Illustration of our sampling process.

Algorithm ¹⁵	Hyperparameters and Values
KMeans	<code>n_clusters</code> \in {3..15}
AgglomerativeClustering	<code>n_clusters</code> \in {3..15}
AffinityPropagation	<code>damping</code> \in {0.5, 0.7, 0.9}
MeanShift	<code>max_iter</code> \in {300, 400}
SpectralClustering	<code>n_clusters</code> \in {3..10}, <code>n_init</code> = 10
DBSCAN	<code>eps</code> \in {0.01, 0.05, 0.07, 0.1}

Table 5.3: Clustering algorithms and tested parameters for identifying dataset strata.

clustering algorithms can be applied to the dataset directly; the only feature modified for clustering is the Docker version number, which we reduced to the major version and then applied one-hot encoding. The best clustering algorithm in our hyperparameter study was k-means clustering with 13 resulting strata. We used the silhouette coefficient [Rou87] to evaluate the resulting clusters, obtaining a value of 0.93 for the final clustering. After clustering, we trained a decision tree to predict the final clusters and capture its feature importance values. These values quantify the influence of the individual image parameters on the clustering. Table 5.4 shows a quantitative breakdown of the feature importance. As expected, the image size is the most important parameter in the clustering (because it has the highest value range), followed by the number of file system layers. The largest stratum contains 52,574 images, while the smallest contains 351 images.

In the next step, we need to determine how many images to take from each stratum.

Feature	Clustering Importance
<code>fs_size</code>	0.520
<code>fs_layers</code>	0.116
<code>meta_docker_version</code>	0.112
<code>cmd_envvars</code>	0.102
<code>cmd_args</code>	0.064
<code>net_ports</code>	0.052
<code>fs_volumes</code>	0.028
<code>io_tty</code>	0.003
<code>io_attach_stdout</code>	0.002
<code>io_attach_stdin</code>	0.001

Table 5.4: Importance of dataset features for final clustering.

For this, we use disproportionate allocation [SSW03], meaning that the stratum’s size and variance play a role in deciding how many images to take. The number of images n_i taken from a stratum i is given by:

$$n_i = \left\lceil N \cdot \frac{d_i/D \cdot s_i}{\sum_{j=0}^{M-1} d_j/D \cdot s_j} \right\rceil, \quad (5.1)$$

where N is the target sample size from the dataset, M is the number of strata, d_i is the number of images in stratum i , D is the total number of images, and s_i is the standard deviation in stratum i . We use the square root of the total variance to measure the standard deviation of a stratum. The total variance is a generalization of the variance for multidimensional datasets and is defined as the trace of the covariance matrix [RS97]. We set the target sample size N to 1000; the number of strata M is 13. As a result, the number of images selected from each stratum varies between 3 and 238, and the total number of images in the dataset sample is 1008; the ceil operation explains the difference of eight to the parameter N in the formula. In summary, through our sampling process, we reflect the diversity of the entire dataset in our sample. A visualization of the resulting sample composition is shown in Figure 5.5.

We use two test environments for the start time measurements presented in the next sections. The first environment consists of three bare-metal servers with identical hardware. The servers are HP ProLiant DL360 Gen9 with Intel Xeon CPU E5-2640, 2x 16 GiB HP 752369-081 DDR4 RAM, and a 500 GB HDD disk of type HP MB0500GCEHE. The second test environment consists of three Google Cloud virtual machines of type `e2-medium` running in the `us-central1-c` zone with a 100 GB zonal balanced persistent disk. All test machines use Ubuntu 22.04 LTS as the operating system and Docker v20.10.21, `containerd` v1.6.10, and `runc` v1.1.4.

In addition to the three test machines, we deployed a master VM in each environment to host the image and result databases. Figure 5.6 shows the measurement process

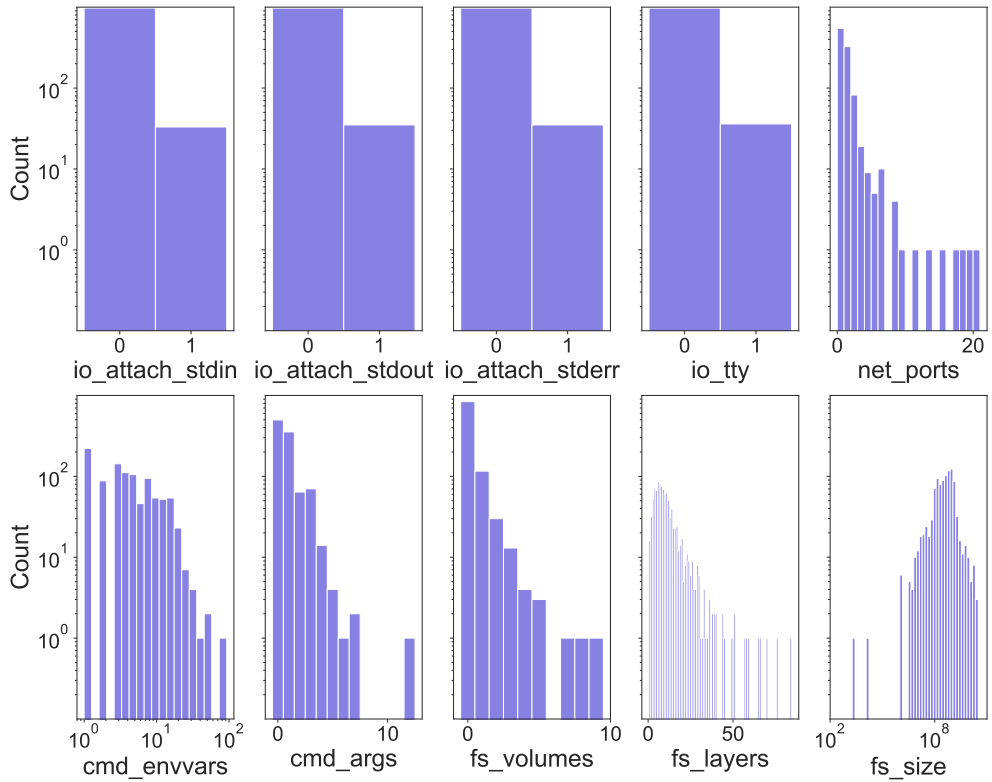


Figure 5.5: Feature distributions for the final sample.

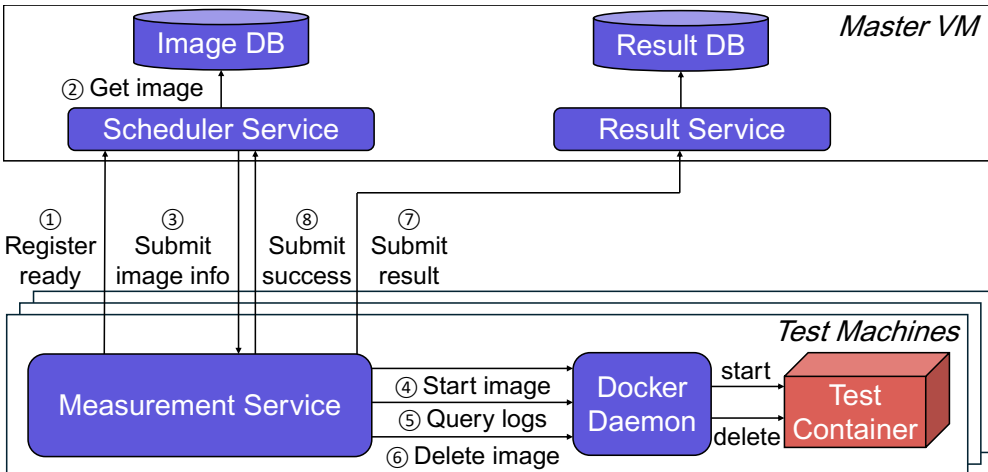


Figure 5.6: Measurement process for container start times.

and involved components. The master VM hosts a scheduler service that sends jobs, each defined by a specific image hash and a desired number of repetitions, to the test machine. The test machine runs the measurement service, a tiny web service, as an ordinary process (not in a container). Besides that process, only the Docker daemon is running on the test machines during the measurements. The execution of a job on a test machine consists of three steps. First, the required image is downloaded from Docker Hub using the `docker pull` command. Then, the container is started with its default configuration using the `docker run` command. We extract the time when the Docker daemon receives the `run` command from its logs and the time when the container finished the start process from the output of the `docker inspect` command. The start time is the difference between these two timestamps. Note that we only use data that Docker collects and do not manually generate any timestamps. The pull time is not recorded and does not play a role in the tests. Once the timestamps have been recorded and the job has finished, the container and image are removed from the test machine. The results are submitted to a result service on the master VM and stored in a database. For the 1008 sampled images, we run 30 repetitions for each image, resulting in 30,240 container starts per environment. We use the randomized multiple interleaved trials methodology [AB17] to account for the variability of the cloud environment and reduce caching effects. This results in the order of repetitions being random.

5.4 Variance of Container Start Times

We use a three-step approach for the description of our results. In the first step, we investigate how the start time of one particular container varies between repeated starts. In this step, we assess the start time variance for one fixed feature combination

Environment	Coefficient of Variation				
	Min	Median	Mean	95th Percentile	Max
Google Cloud	0.023	0.153	0.179	0.389	0.966
Self-hosted	0.051	0.177	0.270	0.788	3.069

Table 5.5: Coefficients of variation for start times of one image configuration.

in our dataset. In the second step, we analyze the impact of the image configuration on the start time. Here, we consider the measurements from the Google Cloud test environment and our local testbed. This means we look at the start times of different image configurations in two specific environments. The last step deals with the influence of selected platform parameters. We investigate whether the results from the Google Cloud environment and our self-hosted environment can be transferred to other machine types in the Google Cloud. Here, our focus lies on the impact of the host configuration, including hardware and software. In other words, we evaluate the start times of different image configurations in different environments.

Before we present a detailed analysis of the start times, it is worth looking at the overall statistics to understand the magnitude of the start times. On Google Cloud’s `e2-medium` virtual machines, the minimum start time is 277 ms, the mean is 1886 ms, the median is 1689 ms, and the maximum is 17,605 ms. In the self-hosted testbed, the minimum is 1241 ms, the mean is 8417 ms, the median is 6470 ms, and the maximum is 426,687 ms.

As described in Section 5.3, we start each image configuration 30 times. We consider the coefficient of variation (CoV) as a metric to quantify the variability of the start time. Consequently, we obtain a CoV for each of our 1008 sampled images. Table 5.5 shows the minimum, mean, median, 95th percentile, and maximum CoV for the two test environments.

We see that the self-hosted environment exhibits a larger variation than the Google Cloud, as evidenced by all statistical values, although the median for both environments is comparable at 15.3% and 17.7%, respectively. The cause of the higher variation in the self-hosted environment can be explained by the storage technology used. While SSDs back the Google Cloud VMs, the self-hosted servers have conventional HDDs. We conducted I/O benchmarking runs according to the methodology described by Google.¹⁶ The results indicate that the CoV of submission and completion latency in read-intensive scenarios is about 2-11 times higher in the self-hosted environment than in the Google Cloud. We further investigate the influence of the disk type and other platform parameters in Section 5.6.

In our analysis, we had a particular focus on the outliers that have a high CoV. The 95th percentile shows that there are only a few outliers in both environments. We analyzed the 5% images with the highest CoV from both environments, totaling

¹⁶<https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance-linux>

51 per environment. Only four images were outliers in both test environments. We further analyzed the configurations of these images but found no clear relationship to any single dataset feature. This is also in line with the results of the next section.

5.5 Influence of Image Configuration Parameters

In the following, we perform a regression analysis to determine factors from the OCI image configuration that influence the container start time. First, we focus on the results in the Google Cloud environment, as this environment is representative and widely used in practice. Out of our 20 initial features, we eliminated the six textual features from Table 5.1 along with two features that are linearly dependent on other features (`fs_virtual_size` and `io_attach_stderr`). Another set of three features has constant values for all sample elements (`io_std_in_once`, `io_open_std_in`, `cmd_additional_args`). From the definition of these features (see Table 5.1), it is clear that their values are expected to be zero/false by default and only change if the container starts in a non-default configuration. After these exclusion steps, we have a set of nine features left for further evaluation.

Our analysis methodology follows guidelines proposed by Glantz and Slinker [GS01]. We start our analysis with linear regression techniques, as these give us explainable results about which image factors influence the start times. Note that we do not aim to find the best start time prediction model here; we instead investigate which parameters impact the start time. We first consider the influence of our image configuration parameters on the start time individually; that is, we consider a univariate regression problem. Therefore, we train a linear regression model in the form

$$y = \beta_0 + \beta_1 x. \quad (5.2)$$

Here, y denotes the start time, our dependent variable, and x the considered feature, as the independent variable, while β_0 and β_1 are the linear regression coefficients. In addition, we consider the p -value, which indicates the probability of observing similar results, under the assumption that the null hypothesis is correct. Table 5.6 shows the β_1 and p -values of different features from our dataset. In the table, p -values smaller than $1e-100$ were rounded to zero.

The values in Table 5.6 for the univariate regression show that all factors considered, except `io_attach_stdout` and `io_tty`, yield results with a low p -value in the linear regression. The values' units play a role in the interpretation of β_1 . The start time in our dataset is given in milliseconds, and the size is in bytes. Consequently, the small absolute value of β_1 for the image size is explained. However, a deeper interpretation of the coefficients is unnecessary because the error for the univariate regression is very high for all features. The lowest mean absolute error (MAE) is 777 ms for the feature `fs_layers`. To calculate the MAE, we used a five-fold cross-validation. For comparison, the MAE of a baseline model that predicts the mean of all measurements is 806 ms. This shows that the data do not have a simple linear dependence. This is also confirmed by Figure 5.7, which shows the scatter plots for the two features with the lowest p -values, `fs_size` and `fs_layers`. Due to the high amount of data points

Feature	Univariate Regression		Multivariate Regression	
	β_1	p	β_1	p
io_attach_stdin	-2.388e+02	5.422e-08	-3.360e+03	~ 0
io_attach_stdout	6.461e-01	9.879e-01	6.049e+02	4.399e-05
io_tty	8.930	8.321e-01	2.250e+03	1.038e-39
cmd_envvars	2.471e+01	1.561e-84	7.063	3.284e-07
cmd_args	-7.068e+01	4.092e-23	-7.686e+01	4.904e-28
fs_volumes	8.365e+01	6.614e-17	7.892e+01	1.886e-15
fs_size	7.012e-08	~ 0	3.571e-08	2.261e-27
fs_layers	3.109e+01	~ 0	2.576e+01	~ 0
net_ports	1.915e+01	2.027e-05	-1.843e+01	4.323e-05

Table 5.6: Univariate and multivariate linear regression coefficients and p -values.

and for better visibility, density is encoded as brightness. Darker points indicate many overlapping data samples. Intuitively, one could assume a clear relationship between those features and the start time. However, the figure clearly shows that none of the two factors can explain the variance in the start time individually.

We deduce that no single dominant image configuration feature determines the start time; it is instead a multivariate problem. For this reason, we extend our analysis to multivariate linear regression. Table 5.6 also shows the β_1 coefficients and the p -values of multivariate linear regression. The p -values show that all features individually contribute to the start time prediction in the multivariate regression. Further, we performed an F -test to evaluate whether the features taken together contribute to the start time prediction [GS01]. The result is a p -value of $2.2e-16$, indicating that feature interactions also contribute to the start time prediction and that the model cannot be reduced to a naive intercept-only model. However, the MAE of 772 ms also shows that the multivariate linear model cannot describe the data well, performing just slightly better than the baseline. We conclude that there is no linear relationship between our features and start time.

Due to the complex relationships in the data, we next apply a random forest model as an example of an explainable non-linear regression approach. We build a random forest model with 500 trees, each having three randomly sampled features and allowing unlimited depth. Table 5.7 shows the feature importance values for the Google Cloud environment retrieved from the random forest in the second column. We see that size is the most important feature, followed by the number of root file system layers. The MAE of the model based on a five-fold cross-validation is 329 ms, which is less than half the error of the linear models and the baseline. This shows that non-linear models can better describe the data.

In the following, we analyze to what extent our findings apply in the self-hosted

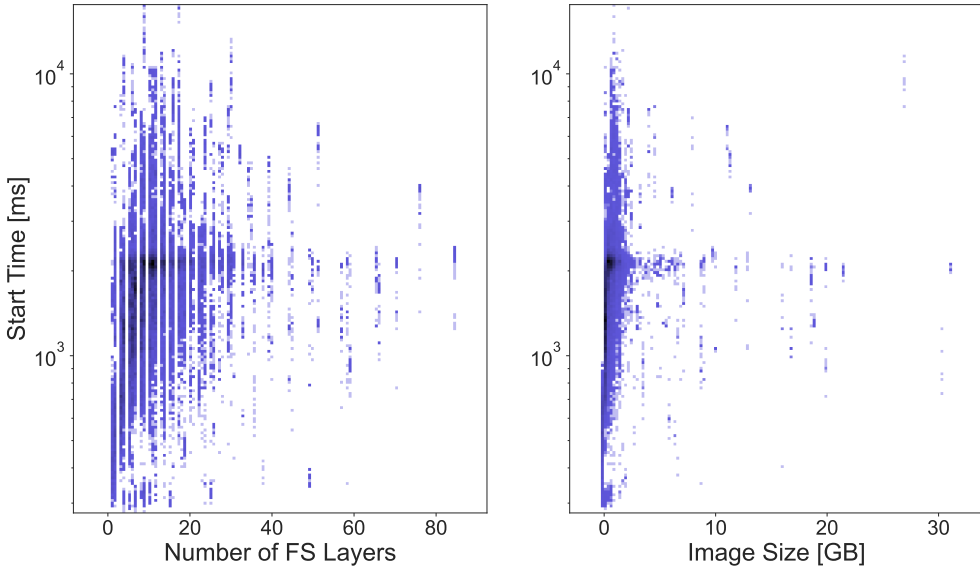


Figure 5.7: Dependence between two selected features and the start time.

Feature	Feature Importance	
	Google Cloud	Self-hosted
io_attach_stdin	0.007	0.008
io_attach_stdout	0.009	0.011
io_tty	0.008	0.010
cmd_envvars	0.123	0.155
cmd_args	0.048	0.032
fs_volumes	0.040	0.032
fs_size	0.527	0.554
fs_layers	0.176	0.165
net_ports	0.062	0.033

Table 5.7: Random forest feature importance values for both test environments.

environment. As stated in Section 5.4, the start times measured in our self-hosted environment have a higher variance than those measured on Google Cloud VMs. Furthermore, we note that of the 1008 images tested, 1007 start faster in the Google Cloud environment, as measured by median start times, than in the self-hosted environment. On average, an image starts 6.1 seconds faster in the Google Cloud than in the self-hosted environment. From this, we conclude that the environment significantly influences the absolute start time values.

In the following, we investigate whether our statements about the impacts of different features on start time are confirmed in our second environment. Figure 5.8 shows, for the same nine features, the start times for all sample images in both environments as a function of feature value. Similar to Figure 5.7, we use brightness-based density encoding. We see that the absolute numbers of start times differ. However, the distribution of data points is clearly similar in both test environments, indicating that each of the nine features has a similar impact in both environments. Apart from a few outliers, similar trends are visible for nearly all of the selected features.

To compare the results quantitatively, Table 5.8 shows different error measures for all evaluated models in our two test environments. All error measures have been derived using five-fold cross-validation with a fixed random seed. The table reports the mean values calculated from the five folds. The baseline model is a model that always predicts the mean of the training data. For the univariate linear models, the error depends on the feature selected for the regression; we report the best score achieved. As the MAE is an absolute measure and differs significantly for the two environments, we also present the mean absolute percentage error (MAPE) as a relative measure to eliminate the scale of the start time. As the MAPE shows, the prediction quality is roughly the same considering the pairwise comparison in both environments. We conclude that the environment determines the absolute values of the start time. This is why we conduct a deeper analysis of platform parameters in the next section. However, from Figure 5.8 and the comparable prediction quality of the models, we conclude that the influence of the image configuration parameters is comparable in both environments. This is also confirmed by the feature importance of the random forest model shown in the third column of Table 5.7. We see that size, layers, and the number of environment variables were the most relevant features in both environments, while all other features were significantly less important for the prediction.

5.6 Influence of Platform Parameters

The results from the previous section indicate that, besides image configuration parameters, platform parameters of the test machines also influence the container start time. We can argue that our large Docker Hub dataset is a good sample for image configuration parameters, which allows for a representative study. Due to the many possible and diverse host configurations, such a broad investigation is impossible for the platform parameters (including hardware, OS, and software parameters). However, an investigation of the influence of host configurations on container start times is needed. Although the results might not be universally applicable, we can still point

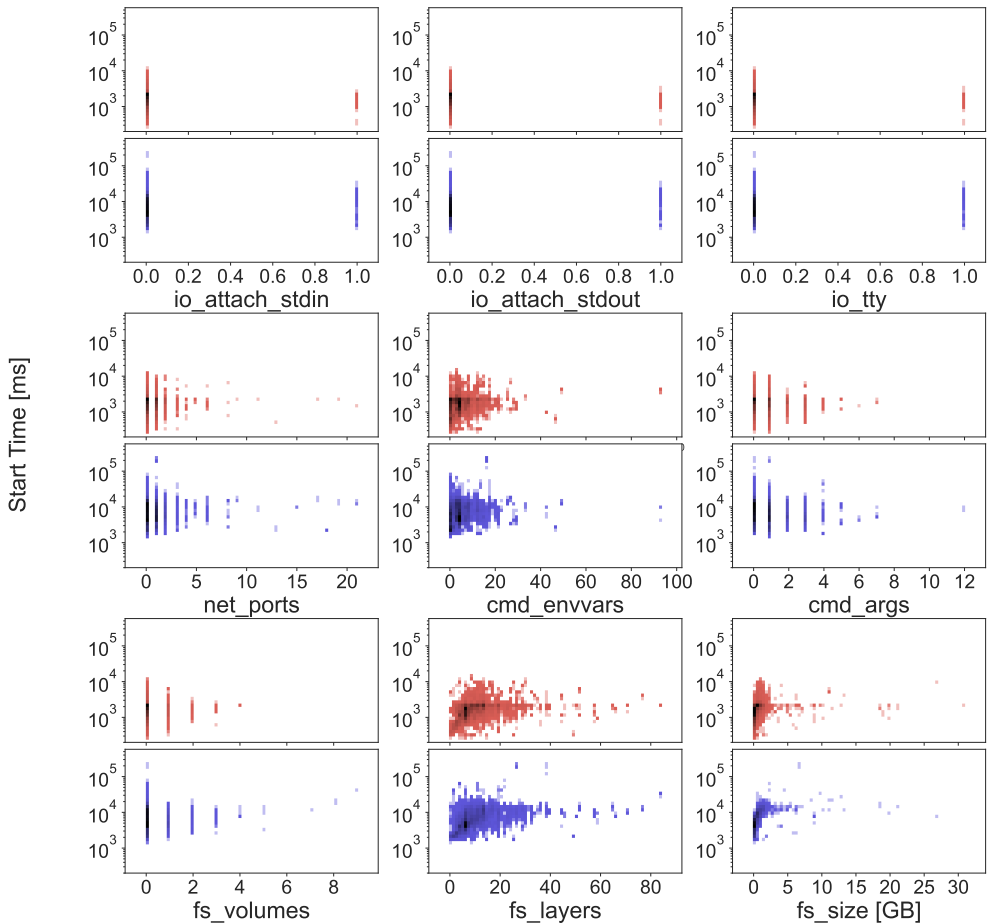


Figure 5.8: Start times and image features in GCP (red) and local (blue) environments.

Model	Google Cloud		Self-hosted	
	MAE [ms]	MAPE	MAE [ms]	MAPE
Baseline	806	0.607	4513	0.748
Univariate LinReg	≥ 777	≥ 0.565	≥ 3751	≥ 0.569
Multivariate LinReg	772	0.559	3661	0.541
Random Forest	327	0.215	1816	0.205

Table 5.8: Error measures for all models and test environments.

Parameter	Low (-1)	High (+1)
Disk	Standard	SSD
RAM	4 GB	16 GB
OS	Ubuntu 22.04 LTS	Fedora Cloud Base 36
Control Groups	v1	v2
CPU Vendor	AMD Milan	Intel Cascade Lake
CPU Cores	1 (2 vCPUs)	4 (8 vCPUs)

Table 5.9: Encoding platform parameters to Plackett-Burman factors.

out tendencies and test our presumptions from previous results, for example, that the disk type (HDD or SSD) has a significant impact. For this purpose, we use different configurations (regarding CPU, RAM, OS, and disk) of Google Cloud `n2-custom` and `n2d-custom` VMs and start the same container images on different machine types. Our methodology is based on a Plackett-Burman experimental design (see Section 2.5), which reduces the number of experiments to conduct compared to full factorial analysis. Full factorial analysis is not feasible in this study due to the high cost of start time measurements. The evaluation is analogous to the previous sections. First, we analyze the variance of the container start time for different machine types. Then, we try to determine which platform parameters have a particular influence on the start time.

The Plackett-Burman experimental design is used to investigate the impact of m controllable parameters on a response measure (in our case, the container start time). Each parameter has two alternatives, which are encoded as -1 (*low*) and 1 (*high*). In the following, we explain the parameters considered in this section in more detail and which values we have chosen for *low* and *high*. Table 5.9 shows an overview of the tested parameters. The platform parameters are selected based on the settings that can be set when creating a virtual machine on GCP. An essential parameter is the *disk type*. GCP offers standard (HDD-backed), balanced, and SSD (both SSD-backed) disks for virtual machines. SSD disks offer significantly higher read and write throughput than standard disks. In our experiment design, we assign the value of -1 (*low*) to standard disks and 1 (*high*) to SSD disks. As the second parameter, we consider the *RAM size*. We choose 4 GB for *low* and 16 GB for *high* as they are commonly used RAM sizes for virtual machines in the cloud. For the CPUs, we consider two parameters. We distinguish Intel Cascade Lake and AMD Milan processors as *CPU vendors*. Intel Xeon Gold 6268CL and AMD EPYC 7B13 processors were available at the time of the measurements. We code the Intel processor as 1 (*high*) because it is better rated in benchmarks¹⁷ and has a higher base frequency than the AMD processor. The AMD processor is assigned the -1 (*low*) value.

¹⁷See <https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Gold+6268CL+%40+2.80GHz> and <https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7B13>

Config No.	Disk	RAM	OS	cgroups	CPU Cores	CPU Vendor
1	+1	+1	+1	-1	+1	-1
2	-1	+1	+1	+1	-1	+1
3	-1	-1	+1	+1	+1	-1
4	+1	-1	-1	+1	+1	+1
5	-1	+1	-1	-1	+1	+1
6	+1	-1	+1	-1	-1	+1
7	+1	+1	-1	+1	-1	-1
8	-1	-1	-1	-1	-1	-1

Table 5.10: Plackett-Burman design for eight configurations to measure.

In addition to the hardware configuration of the VMs, we also consider two software-related settings. As *operating systems*, we choose Fedora Cloud Base 36 and Ubuntu 22.04 LTS. Fedora tends to use the latest software, while Ubuntu focuses more on stability, offering long-term versions. The Fedora OS uses a newer Linux kernel (version 5.17) than Ubuntu (version 5.15). The operating system as a feature covers the whole OS, including the kernel and all dependencies. Due to the newer software stacks, we encode Fedora as *high* and Ubuntu as *low*. While it is out of scope to examine impacts from all software dependencies present on the host, we take a closer look at *cgroup versions*. As described in Section 2.2, Linux containers are based on so-called control groups. The Linux kernel offers two major versions of control groups. `cgroups v2` is the newer version and promises more functionalities and increased performance compared to `cgroups v1`. Docker allows to configure the control group version. In our experimental design, we encode `cgroups v2` as *high* and `v1` as *low*.

We already know that image configuration parameters, such as the size of the container image and the number of file system layers, influence the start time of a container. Therefore, our experiments again include testing containers with different configuration parameters. As in the previous sections, we use a sample of 1008 images created with the same methodology described in Section 5.3. The Plackett-Burman design specifies which parameter combinations are to be tested, as Table 5.10 shows. Each of our 1008 sample images is run 20 times on each of the eight host configurations. This results in a total of 161,280 container start measurements considered in this section.

Before proceeding with the detailed analysis of the platform parameters, we report a comprehensible summary of our measurement results. Table 5.11 shows statistical measures of the start time measurements for each tested host configuration. Similar to the previous section, we observe a wide value range of start times. The lowest start time is below 250 ms for all configurations. Significant differences between host configurations are visible in all other reported measures. For example, the configurations differ by up to two orders of magnitude in the mean and median start time. Based on

Config No.	Start time statistics [ms]				
	Min	Median	Mean	P95	Max
1	234.0	453.0	1370.4	4881.4	20221.1
2	244.6	2388.8	18865.1	98072.7	363794.0
3	226.5	9364.0	12366.2	28751.0	121088.0
4	231.5	1097.6	991.4	1367.4	5554.6
5	233.2	25775.7	40851.4	124614.1	354270.0
6	238.8	520.6	787.5	1525.2	5763.5
7	240.3	1877.8	2390.4	5993.2	14569.7
8	226.5	14168.9	15527.9	31403.2	217957.0

Table 5.11: Start time statistics for different host configurations.

this data, Configurations 1, 4, and 6 tend towards faster start times. Configurations 5 and 8, on the other hand, stand out with remarkably high start times and median start times of well over 10 seconds. These results are also confirmed when taking a different look at the data. For each image, we analyze on which host configuration the container start completed fastest and slowest. Considering a single run, 481 images start fastest on Configuration 1. Another 397 images start fastest on Configuration 6. These values change only slightly when we consider the median start time. In this case, 512 images start fastest on Configuration 1 and 367 images on Configuration 6. Configurations 5 and 3 stand out for the slowest start times. A total of 501 images start slowest on Configuration 5 (439 when considering the median run). The second slowest machine type is Configuration 3, with 219 images that start slowest on this kind of test machine (233 in the median).

In addition to the absolute values of the start time, we also look at the variability of the start time measurements for individual images. As mentioned above, we measure a total of 1008 images 20 times on each host configuration. This results in 1008 coefficients of variations for each host configuration. Table 5.12 shows statistical measures of the coefficients of variation. The results are similar to the values for the start time measurements in Table 5.11. We see that the configurations with faster start times also have a low variability. Configurations 1, 4, and 6 have a relatively low variability, while Configurations 5 and 8 have an above-average variation. Considering a per-image evaluation, 440 images have the smallest CoV on Configuration 7, while 301 images have their largest CoV on Configuration 8. Overall, the results shown above confirm our assumption that there are significant differences in the start times for different host configurations. In the following, we will examine in more detail which platform parameters influence start times most. We also analyze whether image configuration or platform parameters are more important for the start time.

One option to measure the importance of the platform parameters on the start time is the Plackett-Burman feature importance (see Section 2.5). Each configuration

Config No.	CoV statistics				
	Min	Median	Mean	P95	Max
1	0.002	0.035	0.087	0.354	3.447
2	0.009	0.176	0.282	0.774	3.662
3	0.009	0.304	0.389	0.981	2.826
4	0.008	0.059	0.102	0.292	0.835
5	0.005	0.219	0.328	0.861	3.411
6	0.007	0.049	0.096	0.309	1.683
7	0.002	0.022	0.064	0.278	1.936
8	0.007	0.354	0.393	0.835	2.181

Table 5.12: Start time variability per image on different host configurations.

must be provided with a scalar response measure to calculate these values. We use the median start time of all test containers as the response measure. The feature importance is the dot product of the corresponding column of the Plackett-Burman design matrix and the vector of the response measures. We consider the absolute value of the dot product, as the sign is not important for our analysis. We normalize the values so that their sum equals 1, similar to the importance of our random forest models. Figure 5.9 shows the resulting importance values sorted in descending order. According to this measure, disk type is the most important parameter, followed by the operating system and `cgroups` version.

The Plackett-Burman feature importance has, in our case, the disadvantage that the differences between the individual images are lost due to the reduction to a scalar response measure per host configuration. To take a per-image view into account, we look at the question of which setting (*low* or *high*) was faster for each of the 1008 images tested. We do this for every platform parameter. Therefore, we distinguish the configurations that have different settings for one platform parameter. For example, we select Configurations 2, 3, 5, and 8 for the *low* setting of the disk type parameter. We then calculate the median start time from this quantity per image and then over all selected configurations. Similarly, we summarize the values for the opposite feature setting. This way, we can compare median start times for 1008 images for *low* and *high* values of every parameter. Figure 5.10 shows for each platform feature how many images started faster for which feature value. Dominant features tend to have a clear split, while less important features tend to be near a 50:50 split. All but one image has lower median start times when we use SSD disks rather than standard HDD disks. Likewise, Fedora tends to have lower start times than Ubuntu, and `cgroups` v2 tends to be faster than v1. The amount of RAM and CPU vendor features seem less important as they are closer to a 50:50 split.

Overall, disk type is the most important platform parameter for start times in our study. Hence, we can confirm our presumption from the previous section. The

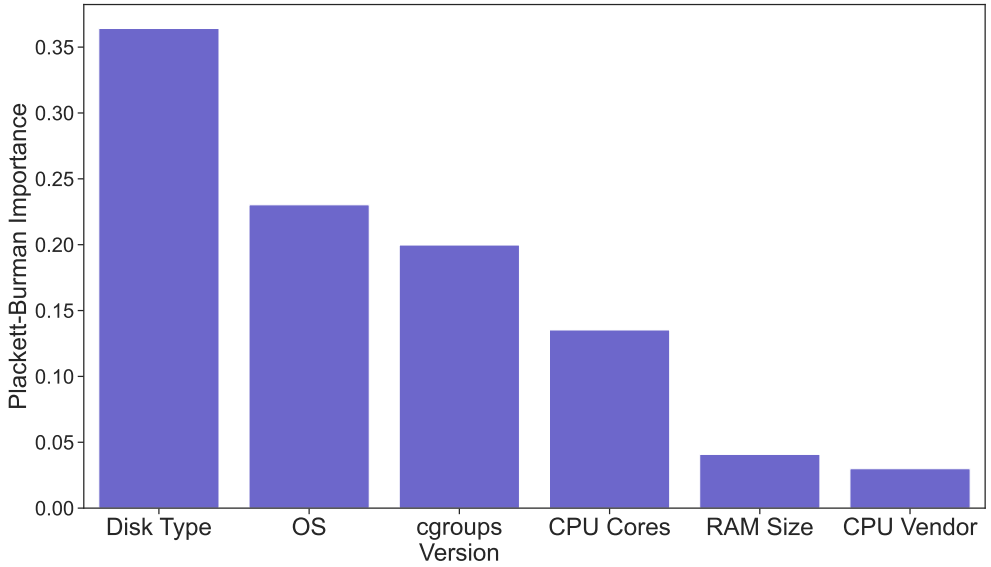


Figure 5.9: Plackett-Burman feature importance for all platform parameters.

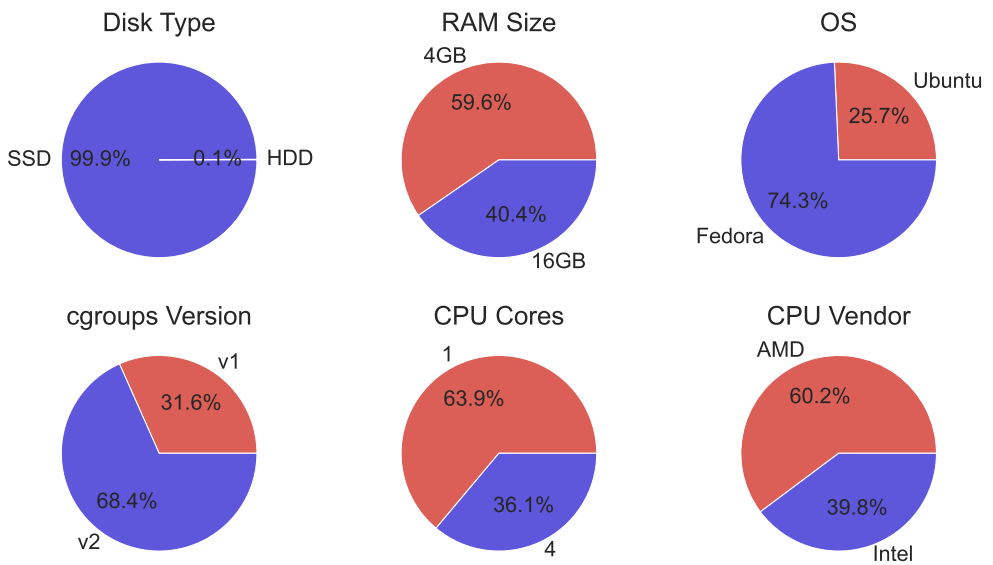


Figure 5.10: Analysis of which platform parameter setting results in faster start times.

disk-intensive workload of a container start is the unpacking and provisioning of the container’s root file system. Hence, increasing disk throughput should be the first consideration when trying to lower start times using host configuration changes. It should be noted that we cannot analyze interactions between features. This would require a Plackett-Burman design with foldover and thus double the measurements. This was not feasible in this thesis due to the high costs. The investigation can be further refined according to other decisive hardware parameters. We opted to analyze only the features we can easily set in the Google Cloud. We did not analyze the influence of individual software dependencies beyond `cgroups`. Instead, we used the operating system parameter, which acts as a cover for different software dependencies (e.g., different Linux kernel versions).

Finally, we consider platform and image configuration parameters together. To do this, we add the image configuration parameters to the measurements from this section and train a random forest model to predict the start time. Table 5.13 shows the feature importance of all features used. As in Section 5.5, the size of the image is the most important factor for the prediction. The second most important feature is the disk type. Both features are the dominant ones in our dataset and together determine about 60% of the total importance. Our statement that both platform and image configuration parameters are essential impact factors for start times is confirmed. In this dataset, platform parameters account for 43.8% of the feature importance and image configuration parameters for 56.2%. Hence, both types of parameters are essential for an accurate assessment of the start time.

5.7 Summary and Discussion

In the following, we summarize our main contributions and findings from this chapter. We first introduced the essential performance metrics for container starts and explained the difference between container start and readiness times. We analyzed over 200,000 images and their metadata from the public container registry Docker Hub, which we obtained from a web crawler-based search. Our results show that container image parameters can vary greatly. Particularly significant are the image size and number of file system layers. We have seen that the average image is several hundreds of megabytes in size and has about 9 to 10 file system layers. After analyzing the dataset, we analyzed container start times and contrasts between different images and environments. To do this, we created a sample of 1008 images from the described dataset using stratified sampling.

As a first result, we found that container start times vary significantly in the same environment. For example, on Google Cloud’s `e2-medium` virtual machines, the start time between different image configurations varies between 277 ms and 17.6 s. Furthermore, we could quantify the variance when starting a container image. The median coefficient of variation was 15.3% in Google Cloud’s `e2-medium` virtual machines and 17.7% in the self-hosted environment. Only a few outliers had significantly larger CoVs. We showed that modeling of start times cannot be reduced to the examination of one single image configuration feature and that a multivariate approach is necessary.

Feature	Feature Importance
<code>io_attach_stdin</code>	0.001
<code>io_attach_stdout</code>	0.003
<code>io_tty</code>	0.003
<code>cmd_envvars</code>	0.043
<code>cmd_args</code>	0.020
<code>fs_volumes</code>	0.008
<code>fs_size</code>	0.397
<code>fs_layers</code>	0.072
<code>net_ports</code>	0.015
<code>platform_disk</code>	0.201
<code>platform_ram</code>	0.062
<code>platform_os</code>	0.029
<code>platform_cgroups</code>	0.025
<code>platform_cores</code>	0.036
<code>platform_vendor</code>	0.085

Table 5.13: Random forest feature importance for platform and image parameters.

We also showed that the relationship between image configuration parameters and the start time is not linear. Using a random forest model as a non-linear regressor, we achieved significantly better prediction results than linear models and have shown that the image size and the number of file system layers have the highest impact on start times among the analyzed image configuration parameters. We also confirmed all of these statements in our self-hosted environment. From the similar feature importance of the random forest models and the comparison of the start times mapped to the features, we infer that image configuration parameters have a comparable influence on start times in both environments. However, we also saw that the host configuration and hardware strongly influence absolute start time values.

Hence, we also looked deeper into the influence of selected platform parameters. While a representative study, similar to the image configuration parameters, was not possible, we focused on selected parameters regarding the host CPU, RAM, OS, and disk. We found that the disk type is the most impactful parameter, with SSD disks offering significantly higher throughput than HDD disks. However, when combining image configuration and platform parameters, we have shown that both feature classes must be considered to assess container start times accurately.

Modern containerized cloud, serverless, and edge applications require high performance, scalability, and resilience. All these characteristics are inevitably tied to low container start times. We have reported the most extensive empirical study to date on how container image configurations and selected platform parameters impact start

times. Our publicly available Docker Hub dataset is one of the largest and most recent container image datasets. The dataset itself contains interesting data regarding modern containerized applications and can be used beyond this work. Our investigations show potential for possible improvements to minimize container start times that application developers and system operators can tackle. Recommended optimizations include reducing the container file system size and the number of file system layers. Moreover, SSD disks are necessary for guaranteeing fast and stable container start times.

We next discuss the limitations and open challenges of this study. Our dataset reflects only a subset of all container images and has been collected from one registry. Furthermore, we limit our study to Linux images with `amd64` target architecture and start them using one specific version of Docker; we did not investigate whether these results can be generalized to other architectures and container engines. Nevertheless, we argue that our results remain valuable, as the selected technologies are common in modern cloud environments. For technical, time, and cost reasons, we tested only a subset of the more than 200,000 images in our dataset and used their default configuration only. Through our stratified sampling strategy, we aim to reflect the diversity of the dataset in the evaluated sample, but further measurements of other image configurations may provide additional insights.

Previous work has shown that measurements in cloud environments can exhibit considerable variability [LC16, LSL19]. To reduce the impact of such variability on our results, we repeated each measurement up to 30 times for each image, using the randomized multiple interleaved trials methodology, as suggested by state-of-the-art benchmarking guidelines [AB17, KLvK20]. However, further measurements in the Google Cloud (e.g., in other compute regions) may lead to different results. Furthermore, our measurements were designed in a way that only one container was active on one machine at a time. In practice, several containers usually run in parallel on a VM or server. This and other factors are out of the scope of this work.

As introduced in Section 5.1, the start time, as considered in this chapter, cannot be equated with the readiness time. A generalization is impossible due to the application-specific setup and network-specific pull times. In particular, analyzing setup time requires knowledge about the content and purpose of the container image. In this work, we maintained a black-box view of the container’s file system. Nonetheless, we showed that the start time is important, as it varies significantly for different container images. Moreover, one has to regard that some image features, especially size, might also influence the pull and setup times. In summary, this chapter presents a solid foundation for further measurement-driven investigations of container start and readiness times. Some of the limitations in the measurement setup can be eliminated in future work if more resources are available.

All in all, this chapter presented an empirical study on container start times, including data from numerous container images. Our dataset extracted from Docker Hub gives valuable insights into the characteristics of container images in the wild. We used this dataset as a basis for comprehensive start time measurements in different representative environments. The collected data provide a basis for identifying factors

impacting container start times. We have thus made an important contribution that addresses Challenge 2 of this thesis.

In the first part of this chapter, we addressed RQ B1 ("How to acquire a representative dataset of container images and their start times?"). We extracted a representative, unbiased container image dataset from Docker Hub that includes both the most popular and most recent images. This balance is crucial as the most popular images provided by large companies are, on the one hand, the most relevant ones in practice, but on the other hand, might be highly optimized. The most recent images enrich the dataset with images from random persons and thus create diversity in the dataset, which is essential for determining the factors impacting container start times. The second part of this chapter focused on the analysis of our measurement results, thus addressing RQ B2 ("Which factors impact container start times?"). Our main insights include that both image and platform parameters impact container start times and that the most important features are disk-related (such as the disk type and container image size).

Chapter 6

A Continuous, Decentralized Approach to Autoscaling

Cloud computing has rapidly advanced digital technologies and will be pivotal for future applications, such as widespread AI and virtual reality. In this context, autoscaling is vital in modern elastic cloud systems, ensuring optimal resource utilization and quality of service under varying loads. The shift to serverless computing underscores the need for efficient autoscaling, as this task falls into the hands of providers. The evolution from monolithic applications to microservices and fine-grained functions has transformed autoscaling mechanisms and the workload patterns to be handled [XSY⁺23, XYW⁺23]. Containers with short lifecycles, replacing long-lived virtual machines, present both challenges and opportunities [ADPDM18, BQ20, QCB18]. A core problem, for example, is that it is difficult to create explicit (white-box) models of applications due to the vast number of deployed services and the limited availability of source code. Promising opportunities arise from fast container start times (see Chapter 5) and the fine-grained and comprehensive control of applications enabled by modern container orchestration frameworks. These developments motivate a new generation of autoscalers specialized in fine-grained, fast-starting applications rather than traditional heavy applications deployed on slow-starting virtual machines.

Autoscalers must essentially solve the problem of *which scaling action* to take *at which time*. As discussed in Section 3.3, existing approaches range from threshold-based to control-theoretic and reinforcement learning mechanisms. Most of these works use the paradigm that a central instance ("the autoscaler") makes scaling decisions at fixed intervals based on preprocessed monitoring data from multiple service instances. For example, the Kubernetes HPA scales a service based on average values for resource utilization in adjustable periods (e.g., 60 seconds). This paradigm is also used in established, often simple, autoscaling solutions offered by public cloud providers, such as AWS or Azure [QIP⁺24].

The described conventional autoscaling paradigm has conceptual weaknesses. (1) The autoscaler acts as a single point of failure, impacting resource adjustment if it malfunctions or encounters incorrect or missing monitoring data. (2) The scale-dependent nature of the autoscaling process poses challenges, making it uncertain whether it works equally well for varying system scales. Bottlenecks arise in the acquisition and processing of monitoring data, and scaling policies may need to be revised when relying on absolute instance or load numbers. (3) Centralized autoscalers face the challenge of determining optimal decisions from a large action space, often requiring signifi-

cant training data for learning-based approaches. Many approaches use configuration parameters like cooldown times, posing challenges for practitioners and hindering practical adoption [ADPDM18, ATGC20]. (4) Reaction times to critical, unforeseen events are limited and suitable only for stable and predictable workloads. In summary, the autoscaler being a single point of failure, coupled with the need for optimal actions at fixed intervals, demands complex solutions lacking critical explainability for practical use.

In this chapter, we first thoroughly analyze open challenges in autoscaling and then present a novel approach to autoscaling that diverges from the conventional paradigm by granting each service instance the authority to trigger autoscaling. This decentralized, scale-invariant process relies on self-organizing instances rather than centralized monitoring. We view autoscaling as a continuous process where numerous minor decisions collectively contribute to global scaling behavior. Service instances, processing local monitoring data, are pivotal, feeding into scaling functions that determine probabilities for actions like adding an instance, removing one, or taking no action. The decision is communicated to the cluster control plane via the execution layer, with scaling decisions occurring randomly over time. In the case of sufficiently large deployments, an asymptotically continuous autoscaling process ensues. Overloaded instances trigger the creation of new ones, and as the number of instances increases, overall utilization decreases, halting upscaling. Conversely, decreased load leads to underutilized instances, prompting the removal of replicas. In contrast to existing decentralized autoscaling approaches, our concept poses minimal assumptions on applications and runtime environments and features maximum flexibility through simple yet powerful configuration.

We formally introduce and evaluate the outlined approach in three steps. First, we analyze formal properties using a discrete-time queuing model. Through simulations, we evaluate different configurations, scenarios, and key influencing factors. Using a proof-of-concept implementation, we show the applicability of our approach for scaling serverless functions in a realistic environment and compare it to established baseline autoscalers. Our results show the great adaptability of our approach to different workload profiles. Compared to the standard autoscaling mechanisms of Kubernetes and Knative, we improve various QoS metrics while simultaneously reducing costs. In addition, we showcase the applicability of our approach beyond CPU-bound container applications in a use case inspired by video streaming.

This chapter addresses Challenge 3 from the overall context of this thesis. Our contribution provides an autoscaling mechanism that is applicable to a broad range of modern cloud applications and offers great adaptability to different workloads and resource usage characteristics. To achieve Goal C of this thesis, we address the following guiding research questions in this chapter:

- RQ C1: Which conceptual weaknesses limit the applicability of conventional autoscalers to modern cloud applications and workloads?
- RQ C2: How to design and implement an autoscaling approach for lightweight execution at scale?

- RQ C3: How to enable adaptability to highly dynamic workloads?
- RQ C4: How to enable adaptability to different characteristics of cloud applications while keeping the configuration manageable?

The remainder of this chapter is structured as follows: Section 6.1 provides experimental insights into production-grade autoscaling in an environment mirroring the production setup of an industry partner. Based on these insights, Section 6.2 states six essential open challenges for production-ready autoscalers. Section 6.3 outlines the idea of continuous decentralized autoscaling, explains how it addresses the aforementioned challenges, and includes a motivating example showing the potential of our approach. Section 6.4 formally introduces our approach and provides a deep dive into its algorithms, parameters, and properties. Sections 6.5 and 6.6 further analyze the properties of our approach and its performance in different scenarios using a discrete-time queueing model and a discrete-event simulation. Section 6.7 introduces the implementation of our approach for containerized applications and includes experiments with different serverless functions that validate the model and simulation and compare our approach to established competitors. Section 6.8 discusses and shows the applicability of our approach beyond container applications. A summary and comprehensive discussion of this chapter is given in Section 6.9. We published parts of this chapter, including text paragraphs, algorithms, figures, and tables, as a journal paper in the Performance Evaluation journal [SGL⁺25] and in two full industry papers at the 2022 and 2023 ACM/SPEC International Conference on Performance Engineering (ICPE) [SGvK⁺22, SEvK⁺23]. The former of the industry papers was selected as the runner-up for the Best Industry Paper Award.

6.1 Experimental Insights Into Production-Grade Autoscaling

Autoscaling has been a vital research topic since the beginning of the cloud computing era and has high relevance in several subdomains, such as serverless computing or fog computing [TTT⁺18]. In general, scaling refers to the task of dynamically provisioning computing resources under varying loads. Scaling has to be automated in modern cloud environments with highly dynamic and complex workloads. Moreover, scaling has a major impact on the business value of cloud software as it affects both operating costs and customer experience. An optimal autoscaler is able to minimize costs as well as SLO violations.

Autoscaling solutions in the industry, for example, offered in public clouds, like GCP,¹ Microsoft Azure,² or AWS,³ are often relatively simple and rely on user-defined scaling rules (e.g., CPU utilization thresholds). The default autoscaling behavior for Kubernetes is also based on a simple scaling rule assuming a linear relationship between the supplied resources and the target metric. Singh et al. [SGJN19] review over 100 autoscalers proposed in the scientific literature. These autoscalers are usually

¹<https://cloud.google.com/compute/docs/autoscaler>

²<https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview>

³<https://aws.amazon.com/ec2/autoscaling>

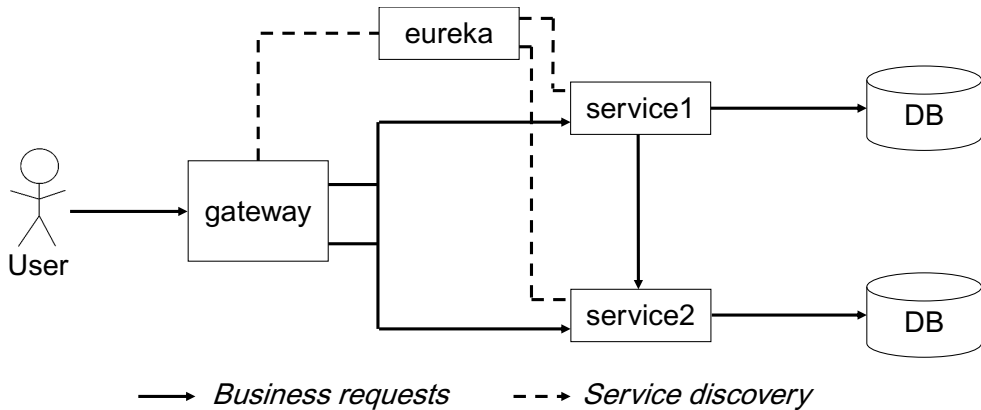


Figure 6.1: Application under test consisting of services from our industry partner.

more complex, employing mechanisms based on concepts from queueing theory, fuzzy methods, control theory, reinforcement learning, and more. We generally observe a big difference between state-of-the-art autoscaling in research papers and production systems. Consequently, the question of why the majority of research autoscalers are not deployed in practice arises.

To address this question, we aim to identify core challenges that autoscalers face in modern production systems covering conceptual, technical, and non-functional requirements. As a foundation, we conduct experiments with a real-world multi-service business application (based on Java and Spring microservices) running different realistic workloads. A cluster with a representative hardware and software technology stack is used for the deployment. We evaluate the performance of different autoscaling strategies, including reactive, proactive, and hybrid scalers and different scaling metrics. Moreover, we report on the performance behavior of overloaded services and, based on our findings, outline the problems that arise for autoscalers. This section focuses on the experiments, while Section 6.2 focuses on the derived challenges for production-ready autoscaling.

For the experiments in this section, we use a test application that comprises a representative subset of some business services of an industry partner in production. An overview of the application, which consists of a `gateway` service, an `eureka` instance, and two services with their own databases, is shown in Figure 6.1. Every user request is first processed by the `gateway` service, which verifies whether the request is valid. A request is considered valid if special HTTP headers are present and the call refers to a registered Uniform Resource Identifier (URI). The `gateway` service checks if these conditions are met. It then either rewrites some headers and forwards the request to the business services `service1` and `service2` or rejects the request.

Microservice `service1` offers five HTTP endpoints overall: three generate `SELECT` or `INSERT` commands to a connected PostgreSQL database, one generates a request to `service2`, and one retrieves data from a local information cache. In contrast,

`service2` offers only one endpoint, which causes a `SELECT` call to another PostgreSQL database. Every service registers itself at the `eureka` service instance at startup. All services are implemented using Java and Spring,⁴ a widely used framework for backend development.

The test application is deployed in a `kubecf` cluster. `kubecf` is a Kubernetes-based open-source distribution of the platform-as-a-service environment CloudFoundry. The `kubecf` components run on top of a Kubernetes cluster and are deployed using Kubernetes pods. Business applications to be hosted in a `kubecf` cluster are built and deployed in so-called Diego cells, which offer an isolated execution environment. For each application, a memory limit has to be set. The maximum CPU usage of an application is then derived from the memory setting. We use a memory limit of 1024 MB for each service of our test application. We deploy nine Diego cells in total, each with a capacity of 40 GB. This technology stack mirrors the production setup of our industry partner.

Our cluster consists of one physical controller node and six physical worker nodes. Five worker nodes are HPE ProLiant DL360 Gen9 servers with Intel Xeon E5-2650 CPU and 16 GiB DDR4 RAM, and one worker node is an HPE ProLiant DL20 Gen9 server with Intel Xeon E3-1230 CPU and 16 GiB DDR4 RAM. We use a Prometheus v2.27.1 server for monitoring, which scrapes both metrics from the `kubecf` platform (such as the number of currently deployed instances and their CPU, memory, and disk usage) and the application instances once every 30 seconds. The Spring services expose their metrics using Spring Boot Actuator.⁵ For load generation, we use three Apache JMeter⁶ v5.4.1 instances, which generate load for the six different endpoints of the test application. Request parameters are sampled from a uniform distribution. The implemented autoscalers query metrics from the Prometheus server and send scaling requests to the controller node. The autoscalers, Prometheus server, load generators, and PostgreSQL databases run on dedicated servers with sufficient resources outside the `kubecf` cluster and are not scaled within the experiments.

Although we evaluate different scaling metrics, we use one generic scheme for scaling. Independently from the scaling metric, we use the generic and widely used default scaling rule of the HPA to calculate the instances of service s to be deployed in the next scaling interval:

$$n_{t+1}(s) = \left\lceil n_t(s) \cdot \frac{m_t(s)}{m^*(s)} \right\rceil. \quad (6.1)$$

Hereby, $n_{t+1}(s)$ is the number of deployed instances of service s in the next interval, while $n_t(s)$ is the number of currently deployed instances of service s . The measured value of the scaling metric averaged over all instances of service s is denoted as $m_t(s)$, and the desired metric value of service s is denoted as $m^*(s)$. We limit the upscaling to 5 instances per minute for stability reasons. The downscaling is limited to 2 instances per 5 minutes. These rules represent company policies deduced from the production

⁴<https://spring.io/projects/spring-framework>

⁵<https://docs.spring.io/spring-boot/reference/actuator/metrics.html>

⁶<https://jmeter.apache.org>

environment of our industry partner. The scaling mechanism is triggered once every 30 seconds, which means every time when new measurement values are available.

To rate the quality of the different scaling strategies, we consider both costs and SLO violations. We define T as the set of all measurement intervals in the experiment and S as the set of all services that need to be scaled. For our experiments, we do not include the `eureka` service for scaling. Hence, S consists of `service1`, `service2`, and `gateway`. Five instances of each service are deployed at the beginning of the experiments. The total costs C are defined as the sum of all deployed instances n_t of all services in all measurement intervals:

$$C = \sum_{s \in S} \sum_{t \in T} n_t(s). \quad (6.2)$$

For the SLO violation metric V , we consider the ratio of failed requests r_t and total requests R_t for each measurement interval t . A request is considered failed if its response time is above 20 seconds or if an unhealthy response code is returned. We sum this ratio up for all intervals t and then divide the sum by $|T|$, which is the number of measurement intervals in the experiment:

$$V = \frac{1}{|T|} \cdot \sum_{t \in T} \frac{r_t}{R_t}. \quad (6.3)$$

As stated above, the performance of an autoscaler should take both costs and SLO violations into account. Depending on the use case, they may be weighted differently. This is why we introduce a scaling performance metric P_w , which includes an adjustable weight w for the desired costs and SLO violation ratio:

$$P_w = w \cdot V + (1 - w) \cdot \frac{C}{C_{max}}. \quad (6.4)$$

Thereby, $1/C_{max}$ is a normalization factor that maps the costs to a scale between 0 and 1. We choose:

$$C_{max} = |S| \cdot |T| \cdot n_{max},$$

where n_{max} is the highest number of deployed instances for one service. Hence, C_{max} would be the costs if n_{max} instances of all services would be deployed the whole time. In our experiments, the observed maximum number of instances of one service was 20, and we set n_{max} accordingly. All C , V , and P_w can be considered as lower-is-better metrics.

In the first experiment series, we evaluate threshold-based CPU autoscaling, as it can be configured in public cloud environments. We use the Kubernetes default scaling rule from Equation 6.1 to calculate the number of instances. As the desired CPU usage, we use a value of 0.8. We use a simple reactive scaling strategy first, which is later compared to proactive and hybrid strategies. We use a typical workload from the production system of our industry partner and evaluate the scaling behavior over 24 hours. We repeat the measurement three times for each scaling strategy to validate our results.

6.1 Experimental Insights Into Production-Grade Autoscaling

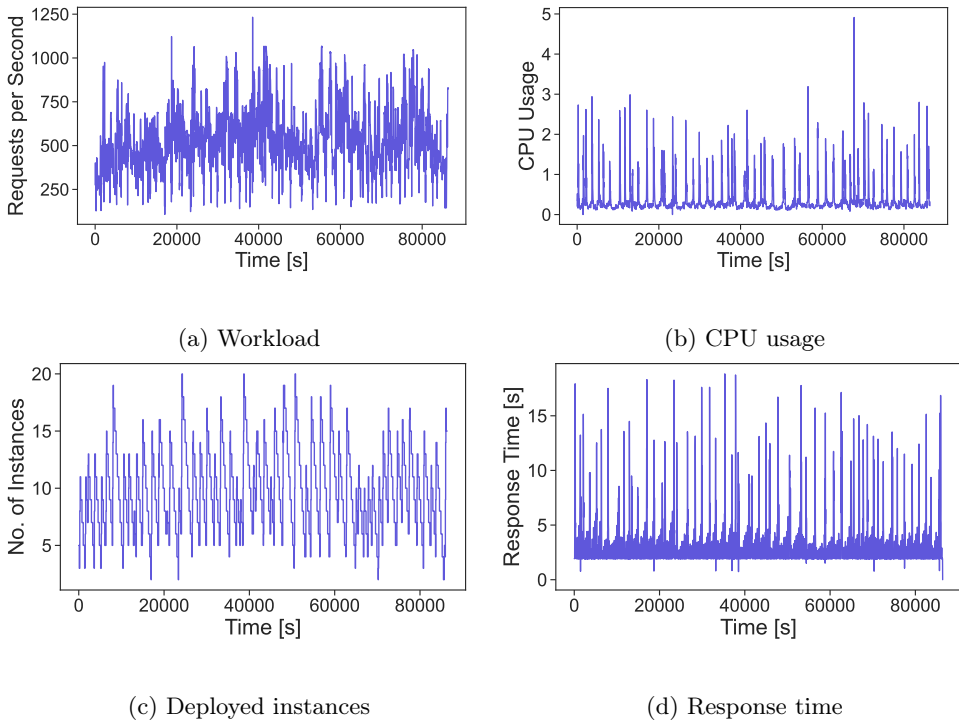


Figure 6.2: Workload and measurement results for reactive CPU scaling.

Figure 6.2a shows the workload used in this experiment. It varies between 300 and 700 rps, with significant outliers in both directions. In our setup, mostly `service1` and `gateway` are bottlenecks, while `service2` shows a consistent performance with few instances. Figure 6.2b shows the average CPU usage, while Figure 6.2c shows the number of deployed instances of `service1` over time measured in one run. We see that the number of instances deployed by the reactive autoscaler rises sharply during high load. The upscaling period ends with a short time when the peak number of instances is deployed. After that, a long and consistent downscaling period follows. Figure 6.2d further shows the average response times in each measurement interval. We see that, congruent with the CPU usage, the response times rise from their normal value range of 2.5 to 4 seconds to a peak value of about 19 seconds. This congruence shows that CPU usage can be used to detect overloaded services. The reactive scaler responds only to high CPU usage and has limited capabilities to prevent SLO violations, as the response time rises exponentially in these cases.

We compare a proactive and a hybrid autoscaler to this baseline. The proactive autoscaler uses the time series of the total CPU usage, that is, the sum of CPU usage of all instances of one service, and it predicts the value one minute in the future. This forecast horizon is long enough to start new instances, as the average readiness time of the evaluated microservices is about 35 seconds. The forecast value is then divided

Strategy	C	V	$P_{1/2}$	$P_{2/3}$
Reactive	46,626	0.205	0.237	0.226
Proactive	41,465	0.213	0.226	0.222
Hybrid	44,818	0.203	0.231	0.222

Table 6.1: Quality metrics for CPU autoscaling.

by our desired metric value of 80 percent and yields the number of instances to be deployed. As a time series forecaster, we use a non-seasonal ARIMA model [New83] as it is commonly used in many forecasting scenarios. The hyperparameters p , d , and q are optimized based on a grid search and the time series conducted in the experiments with the reactive autoscaler. Our proactive scaler is designed similarly to the AWS EC2 predictive autoscaler,³ which also combines metric forecasting and threshold-based scaling. The hybrid autoscaler uses both the reactive and proactive approach and deploys the rounded-up mean number of instances calculated by both strategies. The problem of weighting reactive and proactive scaling is further discussed in Section 6.2.

Table 6.1 shows the average costs and SLO violation scores of all three scaling strategies. The proactive autoscaler incurs about 11.1% lower costs than the reactive autoscaler while causing 3.9% more SLO violations. Compared to the reactive approach, the hybrid autoscaler lowers both the costs (-3.9%) and SLO violations (-0.9%). Which scaling strategy performs best depends on the desired costs-to-SLO-violation ratio as shown by the $P_{1/2}$ and $P_{2/3}$ scores. If we weigh both goals equally, the proactive autoscaler would perform better. The hybrid autoscaler is the better choice if we put more weight on reducing SLO violations, which is likely to be desired for production-grade customer-oriented business applications.

In the second experiment series, we use application-level metrics for scaling in addition to platform-level metrics like CPU usage. Overall, we collect 73 metrics per service, which can be divided into three groups. The first group is the platform metrics queried from `kubecf`, for example, the CPU, memory, and disk usage. The second group is application metrics exported by the Spring microservices, such as different Java Virtual Machine and logging metrics. The last group contains automatically created metrics from the Prometheus monitoring server, such as the scrape duration, which is the time Prometheus needs to query application metrics from a service instance. In the following, we pick metrics from all three categories and evaluate how a simple reactive autoscaler performs with these input values.

A suitable scaling metric should preferably have a simple relationship to the application state and performance and needs to depict overloads as a minimum requirement. We know from the first experiment series that CPU usage fulfills this requirement in our case. In search of alternative scaling metrics, we analyzed the results from the previous experiment series and selected the metrics from each category that best fulfill the stated requirements. The first metric selected for further evaluation is the

thread ratio θ . We define θ as the ratio of the total number of threads of a service instance divided by the number of running threads. A value of 1 would mean that all created threads are running. The higher this value, the more threads are in a waiting or blocked state in proportion to the number of running threads. This means that requests are potentially queuing and waiting for processing time. For the autoscaler, we use the average value of θ of all instances of one service and use Equation 6.1 to calculate the number of instances. The desired metric value is set to 7 based on the experiment data from the first experiment series.

This metric works only for the business services `service1` and `service2`. The `gateway` service works with a nearly constant number of threads. For this service, we choose the application metric `system_load_average_1m`, which is exposed by Spring Boot Actuator. This metric shows the sum of the number of runnable entities queued to available processors and the number of runnable entities running on the available processors averaged over the last minute.⁷ Similar to the thread ratio, this metric depicts some kind of queue length. We use 4.5 as the desired metric value for scaling.

From the group of automatically created metrics, we use the scrape duration exported by the Prometheus server. This metric mainly captures the response time of a service instance at its endpoint for metric exposition. Hence, in contrast to conventional response time measurements, we only use the response time of one request per 30 seconds to an endpoint, which is irrelevant to users. Consequently, this metric is available without any additional overhead and does not require an external tracing engine or similar. It can be compared to a health check, whose response time is interpreted as an independent metric. The desired value for scaling is set to 3 seconds.

To show the relationship of these metrics, we performed a preliminary test. We deployed a single instance of `service1` and stressed it with an increasing load intensity. The test ended when the first request timed out. Figure 6.3 shows the temporal courses of the different scaling metrics with increasing load. As all metrics have different value ranges, we normalized the values by the observed maximum. In general, we see that all metrics are sensitive to increasing load while having their own characteristics. The CPU usage follows the load course best, while `system_load_average_1m` (labeled as system load) has higher variations. The thread ratio reaches its maximum value prior to most other metrics, which means that, at this time, the maximum number of total threads and running threads has been reached. Consequently, it could be interpreted as an early warning of potentially evolving overload. In contrast, the scrape duration stays mostly the same for different load levels and rises exponentially when the service reaches its maximum throughput.

In the following, we evaluate how reactive autoscalers with these different input metrics perform in our environment. We use the 3-hour workload from the production system of our industry partner shown in Figure 6.4 for evaluation. This workload consists of a peak (morning), followed by a phase of a rather low workload (lunch break) and an increasing load at the end (afternoon). We compare three different scaling strategies. The first autoscaler is the same CPU-based autoscaler used in the first experiment series. The second autoscaler, the application-specific autoscaler, uses

⁷<https://github.com/micrometer-metrics/micrometer>

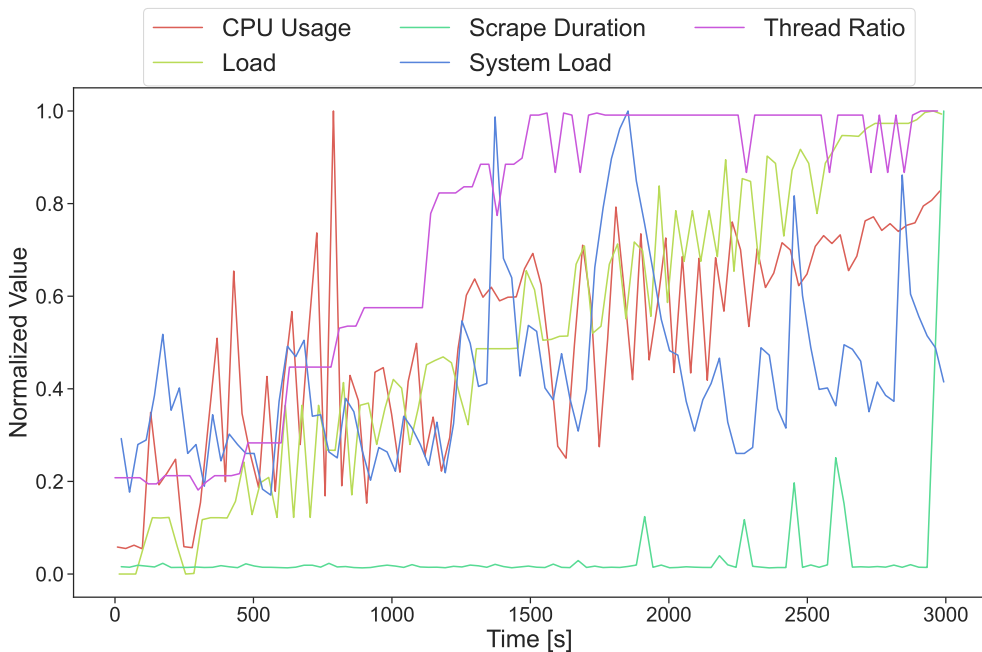


Figure 6.3: Behavior of scaling metrics with increasing load.

the thread ratio for scaling `service1` and `service2` and the system load for scaling the `gateway` service. In exceptional cases when application metrics are not reported for one minute or longer, the application-specific autoscaler temporarily falls back to CPU usage for scaling. We will discuss this issue further later in this section. The third autoscaler uses the scrape duration as the scaling metric. We performed three repetitions per scaling strategy similar to the previous experiment to validate our results.

Table 6.2 shows the average costs and SLO violation metrics for all scaling strategies. The application-specific autoscaler caused the fewest SLO violations and the highest costs. The CPU-based autoscaler has lower costs (-27%) but a higher number of SLO violations (+15.1%) compared to the application-specific strategy. The autoscaler with the scrape duration as its scaling metric performs best in this case, having the lowest costs and only a few more SLO violations compared to the application-specific autoscaler. This is also underlined by the $P_{1/2}$ and $P_{2/3}$ scores. In general, all strategies have pros and cons, and the unconventional strategies achieve results comparable to those of the CPU-based autoscaler.

In the following, we discuss four phenomena observed during the described experiments and how they affect autoscalers. First, we observed that scaling metrics might be delayed, invalid, or unavailable. We used the scrape duration as a scaling metric in the second experiment series. In general, varying scrape duration is rather a problem than an opportunity. In our experiments, the scrape duration varies between 0.2 and

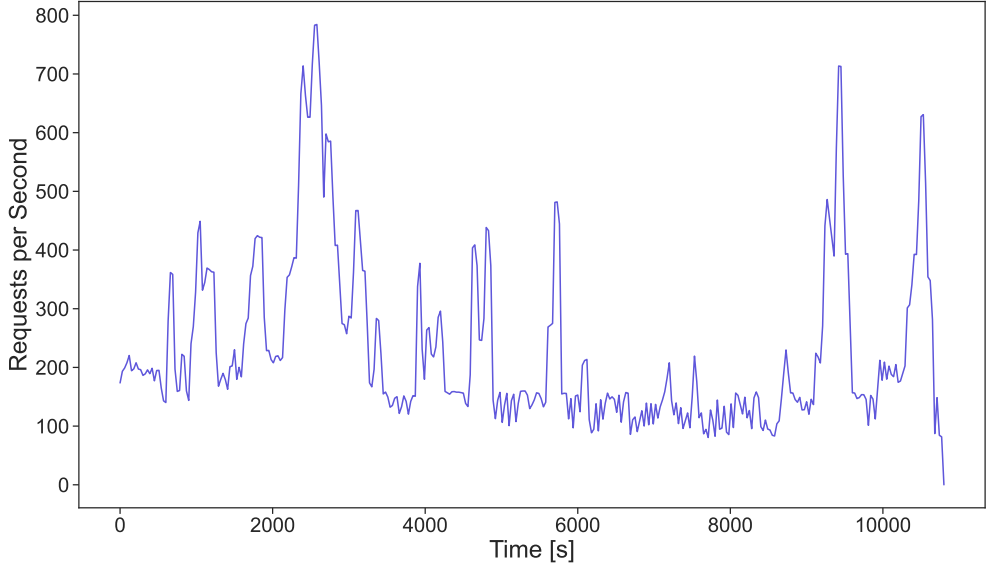


Figure 6.4: Workload for the second experiment series.

Strategy	C	V	$P_{1/2}$	$P_{2/3}$
CPU-Based	3833	0.206	0.236	0.226
App-Specific	5250	0.179	0.272	0.241
Scrape Duration	3822	0.187	0.226	0.213

Table 6.2: Quality measures for different scaling metrics.

10 seconds. The highest value is thereby a third of the monitoring interval. Such delays can be problematic for autoscalers for various reasons. First, these effects mainly appear when a service is overloaded, that is, exactly when a scaling action is necessary. The delays can increase the reaction time of reactive autoscalers. Nevertheless, also for proactive autoscalers, delayed metrics can be problematic, especially in cases when time series forecasting is used, as many time series forecasters rely on or work best with equidistant time series [BZH⁺20]. In addition to delayed values, we observed that scaling metrics can be unavailable or invalid, for example, when the CPU usage is zero. These failures appear more often in our experiments with application metrics than with platform metrics. This is why our application-specific autoscaler from the second experiment series temporally falls back to platform metrics for scaling if application metrics are not reported for one minute or longer.

Second, we noted that response time measurements can be misleading. As a special case for misleading measurements, we analyze response time measurements because they play an essential role in many state-of-the-art autoscalers (see Section 3.3). Our experiments show that the response time cannot always be used to characterize the application state. In Figure 6.2d, we see that the response time rises exponentially when service instances are overloaded and the phases of slightly increasing response times are short. Moreover, we notice a clear lower bound near 2.5 seconds, which equals the minimum response time of business applications in our setting. This time is independent of the provisioned resources and hence cannot be lowered further by supplying more resources. However, a few measurements report response times below this boundary. This occurs when `service1` is overloaded and returns an unhealthy response code. It shows that low response times can be misleading and might be misinterpreted by autoscalers. We conclude that the response time cannot be used to quantify the application state without limitations as it has lower and upper boundaries.

Third, we observed that dependencies between services affect scaling metrics. As shown in Figure 6.1, our test application consists of several services that depend on each other. We especially see that every user request has to pass the `gateway` service first. As a result, the informative value of scaling metrics varies. We observe that whenever the `gateway` service is overloaded, the processing time of a request increases. This results in the fact that the arrival rate and consequently also the resource demand of `service1` and `service2` is lowered. In particular, the simple reactive CPU autoscaler sometimes performs a downscaling action in these cases. This evolves as problematic when the `gateway` service returns to normal operation and processing time, as the number of forwarded requests to `service1` and `service2` increases significantly in a short time interval. Hence, the bottleneck moves from the frontend to the backend. Similar observations for the response time have been made in the literature [GSC⁺21]. These dependencies and scaling metrics' potentially limited informative value are problematic for many autoscalers as services are mostly treated as independent entities.

Finally, we observed that health monitoring causes restarts of overloaded services. Health monitoring is used in environments with orchestration frameworks, like Kubernetes clusters or our test environment based on `kubecf`. For many microservices, it is common to check the application's health by sending HTTP requests to dedi-

cated endpoints. The respective service instance might be restarted if these requests fail or the response time is too high. This phenomenon occurs mainly for overloaded services. In our experiments, these restarts occur up to 486 times in one run of the first experiment series and up to 15 times in one run of the second experiment series. Most autoscalers are unaware of such restarts, although they significantly influence the application performance. During the restart, fewer instances process the workload than assumed by the autoscaler. In addition, the performance of recently started instances differs from the performance of instances that are longer in operation. These and more effects are discussed further and captured in challenges for production-ready autoscalers in the next section.

6.2 Challenges for Production-Ready Autoscaling

This section summarizes our findings from the conducted experiments and states six challenges likely to arise in production systems and, therefore, should be addressed by production-ready autoscalers.

Challenge 1: Balancing proactive and reactive scaling and ensuring the handling of structural or sudden workload changes. In an optimal setting, we would always provide resources proactively, which means that we would know the workload and resource demand in advance and know how to match this demand in a cost-efficient manner. In many customer-oriented business applications, the workload and associated resource demands contain seasonal patterns (e.g., daily, weekly, or monthly cycles). For seasonal time series, a bunch of forecasting approaches is available [MSR16]. However, this is only one side of the medal. Real-world workloads are often more complex, contain bursts, batch jobs, or anomalies, and differ from service to service. Hence, workloads cannot always be reduced to strict seasonality and trend components.

Structural changes in workload patterns are problematic for many autoscalers, especially those with explicit workload models. Proactive approaches are suitable for regular workloads but are often unfit to handle unexpected events [ADPDM18]. Autoscalers based on machine learning experience a shift in the distribution of their input data (e.g., arrival rate patterns or resource usage profiles) compared to their training phase. This is why many learning-based approaches use periodic retraining by default [LXY⁺22, RAH⁺20]. Sudden workload changes that might outpace the autoscaler response and configuration issues have been identified as key challenges for modern autoscalers in industrial contexts [Dat24]. In many cases, the reaction time to sudden workload changes is limited by design because a fixed scaling interval aligned with the scraping period of the used monitoring tool (e.g., one minute [NYK⁺20]) is employed. This scaling interval is often also a hard-to-configure parameter for many practitioners (see Challenge 4).

Another problem for predictive scaling approaches is choosing the right metric to forecast. In research and industry, both CPU forecasting [Ser21, KCH09, NSG⁺13] and workload forecasting [AETE12, BHS⁺19, USC⁺08] is used. For further investigation, we analyze the load intensity and total CPU usage time series from the experiments in the previous section and calculate their approximate entropy (ApEn) [PGE91]. ApEn

is an algorithm for determining the regularity of a time series based on the existence of patterns [DBM19]. As a measure of entropy, ApEn quantifies the information content of a time series [Sha48]. The lower the result value, the less information the time series captures. We calculate ApEn⁸ for the time series measured in the experiments from the previous section. For the first experiment series, the load time series has an ApEn value of 1.066, and the CPU time series has a value of 0.453. For the second series, the ApEn value of the load is 0.741, and 0.307 for the CPU.⁹ We see that the CPU usage has a lower information content compared to the number of arriving requests. Concerning the numeric entropy values and the visual impression from Figure 6.2b, we see that pure forecasting of CPU usage is not sufficient to predict the future resource demand accurately, and workload forecasting may model usage patterns better.

In general, we see that forecasting realistic workloads is challenging, and it can only achieve limited accuracy in the presence of unexpected anomalies. Consequently, a reactive scaling component, able to act in case of unexpected SLO violations, should be part of every production-ready autoscaler. This motivates hybrid scaling, which also showed promising results in the first experiment series from the previous section. However, when having both a reactive and proactive component in operation, conflict situations occur where the outputs of both components are different and have to be aggregated into a single decision [AETE12]. This raises several other research questions, like how to aggregate proactive and reactive scaling decisions, especially in conflict situations.

Challenge 2: Combining application-specific and -agnostic scaling strategies and metrics. Application-specific solutions strive to achieve the optimal scaling behavior for a single application. For example, novel approaches use graph neural networks [MST⁺23, WZL⁺22] that take inter-service dependencies into account and thus achieve coordinated autoscaling for entire microservice architectures. Such approaches might not always be feasible, for example, in the presence of continuous application updates, while application-agnostic and out-of-the-box usable autoscaling solutions are particularly suited for real-world applications [WKKG19]. In many use cases, for example, in the context of short-lived serverless functions [MLKL19], only limited information is available for building application-specific models for autoscaling, as providers usually cannot access the source code and application-level metrics.

In environments with advanced observability, we can make use of the many metrics cloud services expose that describe the application state and health. In our setup with Spring Boot Actuator, we retrieve 62 different application metrics from every service. Not all of them are indeed meaningful for scaling tasks. However, the second experiment series showed that simple reactive scalers based on unconventional metrics like the scrape duration can achieve competitive results compared to traditional CPU-based scaling.

⁸The algorithm needs input values for the parameters m (template length) and r (noise filtering). For our calculations, we choose $m = 2$ and $r = 0.25\sigma$, where σ is the standard deviation of the time series, according to the recommendations in the literature [DBM19].

⁹As a reference, the ApEn for a constant time series is 0, while the ApEn for a time series with random values is around 1.92 for the first experiment series and around 1.44 for the second experiment series. The latter values depend on the length of the time series.

In general, the question of the best metric for scaling is congruent with the question of what metric correlates most with the application performance. It has to be stated for nearly every single service. While CPU, memory, and disk usage are easily interpretable and mostly available metrics, custom metrics could be advantageous in cases when the performance profile of a service is not clearly CPU- or memory-dominated. Moreover, hardware metrics are limited by design, and the resource demand of a service cannot be derived in all cases [BGHK18]. The expressiveness of scaling metrics can be limited, for example, by their value range. Metrics like CPU or memory utilization are defined between 0 and 100 percent. In an overload (100% utilization) situation, we cannot deduce how many additional resources are needed. In this case, stepwise adjustments find the optimal supply, and overall reaction time is limited by the configured fixed scaling intervals (see Challenge 1). Overall, it is not trivial for practitioners to determine the best scaling metrics for individual services.

Challenge 3: Handling the interdependencies between monitoring and autoscaling. Trusting the input metrics is crucial for any autoscaler. Many advances have been achieved in the field of continuously observing cloud systems. However, capturing the correct monitoring data reliably and efficiently remains a key challenge in production systems [Ten24]. Especially during high loads, several effects can influence the validity of measurements in cloud environments. Application metrics are particularly prone to delay or failed measurements, as they must be queried directly from the application instances. Our experiments showed that the scrape duration during high loads increased by more than nine seconds, and some instances do not even report valid values when overloaded. Platform metrics are less prone to failures; however, measurements could be erroneous, especially in large clusters, because metrics have to be collected from many nodes distributed all over the cluster. The monitoring data source is always a single point of failure; if monitoring data is unavailable, no reliable autoscaling can occur. The impact of inaccurate data is significant, given that autoscaling plays a crucial role in alleviating the effects of heavy loads.

Response time measurements play a special role in the design and operation of many state-of-the-art autoscalers (see Section 3.3). However, the reliable acquisition of response times, especially in high-load scenarios, is non-trivial. These rely either on measurements by the application itself, have to be sampled using external tracing frameworks, or are gathered through service meshes. The monitoring overhead required to obtain the autoscaler’s input data can be a crucial factor here. It has been shown that this kind of monitoring introduces considerable overhead [ZSX⁺23], and multiple guidelines for practitioners recommend thoroughly weighing pros and cons before adoption [IP23, Tig24]. Moreover, by design, the response time always reflects the past state of the application, as it is measured shortly before or after the request leaves the system. Therefore, a delay of, for example, 10 seconds in response time measurement is only observable after these 10 seconds have passed. Moreover, one has to consider that the response time should not be the only criterion for evaluating the application’s health and quality. In the presence of some errors, the response time might even be lowered, for example, when a service responds faster because of an internal error, as discussed in the previous section.

Another critical problem when interpreting scaling metrics is the dependency be-

tween different cloud services. A typical property of microservice architectures is that multiple services are involved in processing a single user request. Consequently, errors and high response times might propagate across several services, although they have enough resources assigned to them [GSC⁺21]. In our experiments, we saw similar effects. Whenever the `gateway` service was overloaded, not all requests have been forwarded to the backend services `service1` and `service2`. This resulted in a temporary decline in CPU utilization, and the simple reactive CPU autoscaler reduced the number of instances of the backend services. The backend services then experienced degradations when the `gateway` pursued working. All in all, we see that several factors can influence the reliability of scaling metrics. Another challenge is that the compatibility between the deployed monitoring tool and the autoscaler must always be ensured. Changes in the system monitoring might require reconfiguration of the autoscaler [ADPDM18].

Challenge 4: Keeping configuration overhead manageable. One of the major points of criticism for many approaches in the autoscaling domain is the aspect of configurability, for example, determining suitable scaling intervals, thresholds, cooldown times, model parameters, or similar. The scaling interval is often a hard-to-configure parameter for many practitioners [ADPDM18, ATGC20]. A trade-off between sensitivity and stability of the autoscaling behavior has to be made [CBY18]. In Section 6.1, we used fixed CPU utilization thresholds of 80 percent. The thresholds for the custom metrics have been chosen based on the results of the first experiment series. As stated earlier, the configuration of an autoscaler is hard, and the complexity is increased further when the configuration differs from one service to another. Often, many configuration values depend on the desired costs-to-SLO-violations ratio, that is, how conservative the autoscaler should act. However, other factors influence the configuration, such as expected application start and shutdown times [ATGC20, BQ20, Dat24] or the anticipated kind of workload. Although the DevOps principle stands for stronger coupling of the development and the operation of cloud software, application developers are often not concerned with autoscaling or performance of the application [BEF⁺19]. Furthermore, autoscaling might be outsourced to the cloud provider (e.g., in the context of serverless computing). In these cases, there is only limited insight into the functionalities and performance characteristics of the services to be scaled. Frequent code updates aggravate the problem. This might lead to changed performance properties and the need to adapt models or thresholds. In such situations, it is not feasible to rely on manual reconfiguration. Therefore, production-ready autoscalers should minimize configuration overhead or rely on self-optimization instead.

Challenge 5: Ensuring trust and explainability in autoscaling. As stated earlier, scaling plays an outstanding role in the operation of cloud services. It directly influences both customer experience and operating costs. In modern complex cloud environments, scaling tasks must be handled automatically. An additional requirement for autoscalers in production systems is the transparency of their decisions. This is not only useful for debugging purposes but also necessary to increase trust and potentially propose further enhancements. This explainability does not mean that all calculations must be comprehensible for everyone; it is rather the requirement for an autoscaler to state reasons for its actions. Potentially, log entries like *scaling up*

6.3 The Idea of Continuous Decentralized Autoscaling

because metric x has/will have value y which is considered too high may provide real benefits. Explainability also includes the intuitive adjustability of configuration parameters so that the effect of a configuration change is foreseeable and explainable. While simple, threshold-based autoscalers fulfill those criteria, they are especially a showstopper for the practical adoption of autoscalers based on deep learning. The issue of trust is characterized as one of the core challenges for using AI for autoscaling [Unt24]. Recent research explores the use of explainable AI for autoscaling to mitigate these issues [MBKV23]. We argue that explainability, together with configurability (see Challenge 4), are key requirements for production-ready autoscalers in order to increase trust in the resulting decisions.

Challenge 6: Managing autoscaling in the context of modern orchestration frameworks. As discussed in the previous section and Chapter 4, the autoscaler is not the only mechanism that controls service instances in modern cloud environments. Two conceptual challenges arise. First, in production systems, there are technical or implicit SLOs present, which are enforced by third parties and the autoscaler might not be aware of, like the restarts caused by failed heartbeats discussed in Section 6.1. Second, autoscaling, load balancing, other orchestration tasks, and application-level resilience mechanisms (e.g., circuit breakers) are strongly interdependent. All these activities must work together to fulfill their goal of keeping the application quality as high as possible. Consequently, autoscalers need to interoperate with other orchestration mechanisms. As a concrete example, we showed that the health monitoring unit might restart overloaded services as they fail to send heartbeats. An autoscaler can profit from knowledge about such restarts, as they influence the application performance. First, the number of available instances is reduced during the restart, and some requests might be dropped. Second, after the restart, the performance of the newly deployed instance often differs from those running for a longer time.

In the following, we present continuous decentralized autoscaling, a novel approach that addresses many of the mentioned challenges and exploits the high elasticity potential of modern cloud applications and orchestration frameworks.

6.3 The Idea of Continuous Decentralized Autoscaling

To address the above challenges of production-ready autoscalers, we propose an approach with two important properties: (i) *Decentrality*: In our approach, individual service instances make independent scaling decisions. (ii) *Continuity*: Instead of viewing autoscaling as a problem where the optimal number of instances must be decided at discrete points in time, we treat autoscaling as a continuous process that converges against an optimal system state. In the following, we provide a rough description of the method while introducing details and formal notation in Section 6.4.

In the following, we use the term *service instance* to refer to a deployable unit of a software application that operates within its own computing environment, such as a container, virtual machine, or similar execution environment. It is responsible for directly processing workloads or requests. A service instance may be an instance of a microservice, a serverless function, a virtual machine, or other independent software

artifacts that execute specific tasks within a distributed application. Our approach assumes that service instances can collect and process their own monitoring data, which is also the first step in a scaling cycle. In every iteration, an instance chooses one of the three action alternatives: add a replica (UP), remove a replica (DOWN), or no action (HOLD). The decision is made based on the measured monitoring values. We generally distinguish three ranges: a downscaling range, an upscaling range, and a tolerance zone. For example, for CPU utilization limited between 0 and 100%, we could define the ranges as follows: upscaling range 80-100%, downscaling range 0-50%, and tolerance zone 50-80%. The instance always decides to HOLD if the measured value is in the tolerance zone. If the measured value is in the upscaling (downscaling) range, the system evaluates whether UP (DOWN) should be executed as an action. In the following, we limit our explanation to the upscaling range, but the statements apply analogously to downscaling.

One way would be always to scale up if the measured value is in the upscaling range. This corresponds to a classical threshold-based approach. However, this approach has disadvantages in our decentralized decision-making, as we explore further in Section 6.6. Hence, we consider a more flexible approach based on *scaling functions*. The upscaling function receives a measured monitoring value and returns the probability for a UP action. An intuitive choice would be a monotonically increasing function, returning greater values for inputs farther from the tolerance area. In our CPU utilization example, we could configure that at 81% utilization, the UP probability should be 10%, while at 100% utilization, an UP action should always be performed. The choice of the scaling function and the size of the tolerance, up-, and downscaling ranges allow a more flexible and powerful configuration of the scaling behavior. In the implementation of the approach, the execution of the decision is outsourced to an external component, the execution layer, which provides an interface to a control plane (e.g., the Kubernetes control plane).

Before starting a new cycle, an instance waits some time. Again, one option would be to wait for a fixed period, which equals a classical interval-based strategy. In contrast, we draw this waiting time from a previously defined distribution. This leads to the fact that all currently available instances make decisions at different times and, hence, a high frequency of scaling decisions. If many service instances are active, our approach converges into a quasi-continuous process [HMH18, WTGGH19]. Deterministic waiting times can result in many up- or downscaling decisions within short intervals, resulting in bad scaling behavior.

Our idea addresses the challenges identified in Section 6.2 as follows: By using a quasi-continuous scaling process, we achieve low reaction times to (sudden) workload changes and good adaptability to structural workload changes (Challenge 1), especially compared to autoscalers with fixed scaling intervals. Depending on the scaling functions, our approach could be proactive, reactive, or hybrid. However, within the scope of this thesis, we evaluate only reactive strategies. We will show that reactive strategies, together with the high scaling frequency, are sufficient to match changing demands in the tested scenarios. Nonetheless, we discuss extension options for future work, including simple proactive strategies, in Section 6.9. Our approach provides a good balance between application-specific and -agnostic autoscaling (Challenge 2). On

the one hand, application-specific characteristics can be considered by implementing special scaling functions and using special scaling metrics. On the other hand, our approach can be used similarly to conventional threshold-based autoscalers by choosing simple scaling functions and established platform-level metrics. Our approach does not depend on centralized monitoring and thus eliminates single points of failure (Challenge 3). As service instances process their own monitoring data, no transmission to a server is necessary, and no delay is added.

Our approach has only three key tuning parameters: the upscaling function, the downscaling function, and the waiting time distribution. We will show that concrete configurations of these parameters work equally well for different system sizes, and hence, we argue that managing these parameters is also manageable for practitioners (Challenge 4). The scaling functions provide a flexible, robust, and explainable configuration of the autoscaler (Challenge 5). We will show that all parameters intuitively impact the scaling behavior and that simple scaling functions executed on every service instance are sufficient to handle complex workloads. As we use instance-local monitoring data, we can consider the condition of each service instance directly. For example, the decisions of long-running and newly started instances are mixed, contributing to a robust overall scaling behavior. We will show that our approach is deployable and suitable to act in the context of modern orchestration frameworks (Challenge 6). With our decentralized design paradigm, we differ conceptually from many state-of-the-art autoscaling solutions. Our realization of a continuous autoscaling process and its flexible configuration through scaling functions separate our work from existing decentralized autoscalers.

In the following, we demonstrate the potential of our idea in a motivating experiment. Our experiment setup is a GCP cluster with five worker nodes of type e2-medium and Kubernetes v1.26 with `containerd` as the container runtime. An HTTP Load Generator¹⁰ instance is deployed on an external VM in the same compute zone. Our test app is a Python function with an integrated Nginx gateway that repeatedly computes SHA256 hashes. The average service time is about 2 ms. The container resources are limited to 0.1 CPU cores. The test workload has a total length of six hours. It combines the six workload patterns Gentle, Rise, Decline, Burst, Diurnal, and Seasonal, proposed in a recent autoscaler evaluation methodology [XZPY21], overlaid with uniformly distributed noise.

We evaluate a decentralized autoscaler (DAS) and the Kubernetes Horizontal Pod Autoscaler with different configurations. All scalers make their decisions based on CPU utilization. The HPA receives the metrics from the Kubernetes metrics server. DAS is placed as a binary in the container, runs as a background process, and calculates the CPU utilization directly in the container. In preliminary experiments, we tested 27 HPA configurations, two of which we selected for display here. HPA1 acts in intervals of 60 seconds and uses the default values of all other HPA parameters. HPA2 is the best configuration (best QoS-to-cost ratio) for this workload in our parameter study. It is an aggressive configuration acting in intervals of 15 seconds with no downscaling stabilization. The waiting time for our decentralized autoscaler is drawn

¹⁰<https://github.com/joakimkistowski/HTTP-Load-Generator>

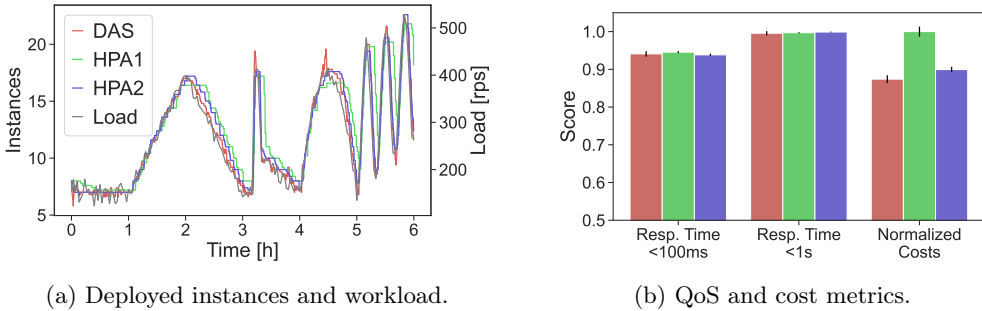


Figure 6.5: Motivating example comparing DAS and two HPA configurations.

from a uniform distribution between 30 and 90 seconds so that the expected value corresponds to the sync interval of HPA1. A target CPU utilization of 75% was chosen for HPA1 and HPA2. For the decentralized autoscaler, we choose a tolerance zone between 70% and 80%. The upscaling probability increases exponentially between 80% and 95% utilization; upscaling always takes place if the utilization exceeds 95%. The downscaling probability increases linearly between 70% and 60%, limited to a maximum of 30%. We chose these scaling functions as they were recommended in our parameter study (see Section 6.6). Refer to Section 6.4 for formal definitions of the scaling functions.

Figure 6.5a shows the workload and the number of deployed instances over time for all autoscalers averaged over five repeated runs. All autoscalers generally follow the workload curve and adapt to the changing arrival rate. Figure 6.5b shows autoscaler evaluation metrics. To assess QoS, we use the proportion of requests with a maximum response time of 100 ms or 1 s (corresponding to different SLO targets). We use a generic cost metric C , which is the area under the curves shown in Figure 6.5a. Hence, it is defined as the definite integral of the running instances over time $I(t)$ limited by the experiment start $t = 0$ and the end of the experiment t_e :

$$C = \int_0^{t_e} I(t) dt.$$

It is, therefore, a generalization of the cost metric introduced in Section 6.1 that applied only to interval-based autoscalers. For better display, we normalized the cost values in Figure 6.5b by dividing all values by the mean costs of HPA1. The results clearly show that the DAS performs better than HPA1 and HPA2. It achieves the same QoS level with costs reduced by 12.7% compared to HPA1 and by 2.8% compared to the optimized configuration HPA2. This is mainly due to the increased scaling speed and elasticity. Figure 6.5a shows the lower reaction times to load changes of DAS compared to the Kubernetes autoscalers. The experiment shows the potential of continuous decentralized autoscaling. Especially in complex scenarios with dynamically increasing or decreasing arrivals, the strengths of our concepts are revealed, as faster reactions are possible through the higher frequency of scaling decisions.

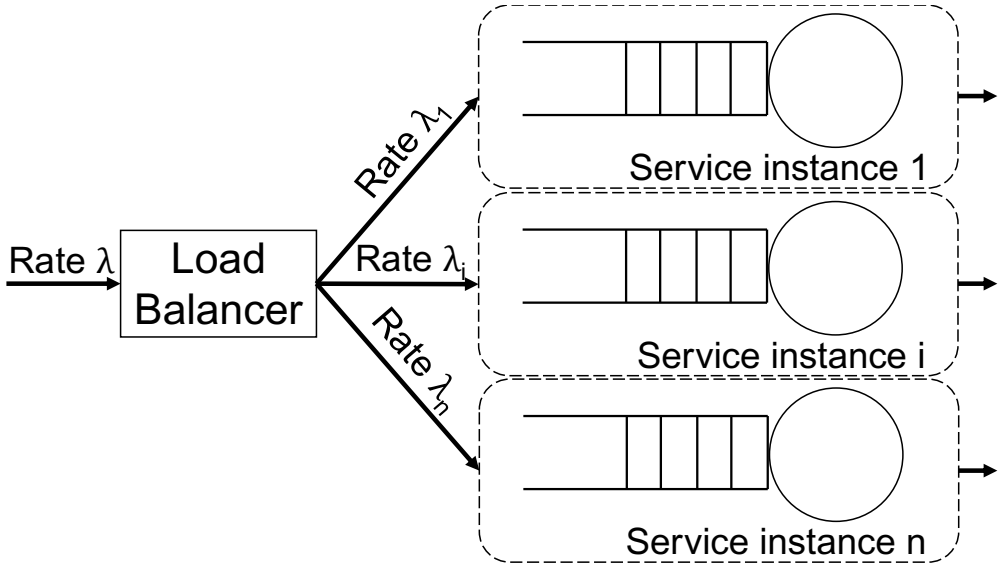


Figure 6.6: Overview of our system model.

6.4 Algorithm and Parameters of Continuous Decentralized Autoscaling

This section first describes our system model and then formally introduces continuous decentralized autoscaling. In this thesis, we consider horizontal scaling only, meaning that scaling takes place by adjusting the number of service instances. We assume that scaling actions can be executed with a reasonable delay and high frequency. This assumption holds especially for modern containerized applications but not for environments where the addition or removal of instances is costly.

Figure 6.6 gives an overview of our system model. Service instances process requests from external sources. A *system arrival process* with an arbitrary inter-arrival time distribution with arrival rate λ dictates the mean number of incoming requests per time unit. In this work, we make no further assumptions or restrictions regarding the sources of the requests and assume that each request originates from one of an infinite number of sources. In practice, requests can be issued by users or external services. The requests first hit a *load balancer*, which distributes the arrival stream to k ($k \in \mathbb{N}$) service instances. Similar to previous works, we assume that the load balancer assigns incoming requests immediately to a service instance and has no centralized queue [MDBvL17]. We do not further model the behavior of the load balancer, as commonly used commercial load balancers in cloud environments, for example, Google Cloud Load Balancers (also used in this work), are closed-source.

The load balancing process splits the system arrival process into k *instance arrival processes* with arrival rates λ_i ($i \in [1; k]$). Service instances have queues with unlim-

ited maximum length. In this work, we assume that all instances are replicas; that is, they are stateless and that their properties, such as service time distribution, are identical. This is justified by real-world deployments of modern cloud applications, such as microservices and serverless functions. Furthermore, we assume there is no interference between the service instances. An infinite number of instances can be deployed without interfering with each other. In public cloud deployments, this is justified as (i) instances are usually resource-constrained, or mechanisms are used to ensure performance isolation (see Section 2.3) and (ii) the amount of available computing resources is virtually infinite. We do not use an explicit model for the cloud environment (e.g., physical servers, resources, and networks). We do not consider performance overheads introduced by these entities (e.g., network delays or the processing time at the load balancer). We do this since we cannot control these factors in our public cloud experiment setup.

As a result, the key performance indicator of our system, the response time, consists of two components: the service time and the waiting time of a request in the queue of a service instance. We assume that once a request starts being processed, this request does not join a queue again. This includes the assumptions that there is no parallel processing of requests and that a resource is always fully loaded or idle. We assume that a request has no external dependencies, that is, synchronous calls to other services for whose response the request must wait. Along with making no assumptions about request sources, we can limit ourselves in the analysis to single instance types, not networks of service instances. External dependencies may be considered indirectly via the choice of the service time distribution.

Autoscaling aims to find the optimal number of instances k^* for a concrete system arrival process with rate λ . k^* is the smallest number of instances for which all SLOs are fulfilled. In our work, we formulate SLOs based on response times. If the instance arrival rates λ_i increase in our system, the waiting times and response times r increase. If r exceeds critical values, SLOs are violated. Adding new instances makes the instance arrival rates λ_i smaller, meaning SLOs can be met again.

In conventional centralized autoscaling systems, an autoscaler processes data collected from all service instances by a monitoring system and calculates the optimal number of instances. For this calculation, the autoscaler uses values averaged over the service instances. In contrast, in our decentralized approach, decisions are made at the instance level based on instance-level monitoring data. This assumes that instances can collect and process their own monitoring data. This is a valid assumption for many runtimes of modern cloud applications, such as containers, unikernels, VMs, and microVMs. Resource metrics such as CPU or memory utilization can be determined directly (e.g., via `cgroup` features in containers). Application-specific metrics (e.g., thread counts) are usually exposed to the outside via dedicated metric endpoints. Instead of querying these by a monitoring tool like Prometheus, we can also process these values locally. By making the decisions at the instance level, we do not rely on a central monitoring system.

In our approach, each service instance executes the algorithm shown in Algorithm 1 during its runtime. A scaling cycle starts with the collection of the current monitoring data m_t . We consider $m_t \in M$ as a vector containing the current values of the *scaling*

metrics, which are the metrics that should influence the autoscaling behavior. M is the set of all possible monitoring data. In each cycle, an instance must decide between three scaling decision alternatives. *UP* causes a new instance to be started. *DOWN* causes an instance to be switched off. *HOLD* does not change the number of service instances. Note that the service instances do not know the total number of instances deployed in the system. This would require global knowledge and would be against our decentralized design paradigm.

Algorithm 1 Algorithm for Decentralized Autoscaling

Input: $D : M \rightarrow [0; 1]$, $U : M \rightarrow [0; 1]$,

W : Probability distribution in interval $[w_{min}; w_{max}]$

```

1: while instance is running do
2:   Collect recent monitoring data into  $m_t$ 
3:   Draw sample  $r$  from a uniform distribution in  $[0; 1]$ 
4:    $P(UP) = U(m_t)$ 
5:   if  $P(UP) > 0$  and  $r < P(UP)$  then
6:     decision = UP
7:   else
8:      $P(DOWN) = D(m_t)$ 
9:     if  $P(DOWN) > 0$  and  $r < P(DOWN)$  then decision = DOWN
10:    else decision = HOLD
11:  end if
12:  Submit decision to execution layer
13:  Draw sample  $w$  from  $W$  and wait for time  $w$ 
14: end while

```

Which action is executed is determined by the *scaling functions* U and D . In the following, we use the term *scaling policy* S to refer to a combination of a concrete U and D . Both functions are defined to return a probability that a scaling decision is made. U determines the probability for UP, D the probability for DOWN. Operators can flexibly define the desired autoscaling behavior by adjusting these scaling functions. By using probabilistic behavior, we enlarge the design space for scaling policies. As a special case of U and D , conventional threshold-based policies that always return either the probability 0 or 1 can also be implemented. In our work, we also assume that the decision is always based on the current measured values m_t . As a conceivable alternative, the functions could also be implemented to use the monitoring values of past cycles (see Section 6.9 for further extensions). What action should be executed is ultimately determined by a random number $r \in [0; 1]$ drawn from a uniform distribution. We assume that for all possible monitoring values, only either the upscaling or downscaling probability is greater than 0:

$$\forall m_t \in M : U(m_t) > 0 \Rightarrow D(m_t) = 0 \wedge D(m_t) > 0 \Rightarrow U(m_t) = 0. \quad (6.5)$$

This means that no vector of monitoring data can cause both a UP and a DOWN action. Lines 4 to 11 of Algorithm 1 show the decision-making of the instance based on

the evaluation of the functions D and U as well as the comparison with the random number r . As a consequence of constraint 6.5, it does not matter whether up- or downscaling is evaluated first. Dependent on the choice of U and D , a subset of M may be created where both the upscaling and downscaling probability is 0. This corresponds to a desired state where no autoscaling should be performed. In the following, we refer to this subset as the tolerance area τ , formally defined as:

$$\tau = \{m_t \in M : U(m_t) = 0 \wedge D(m_t) = 0\}. \quad (6.6)$$

The instance submits the decision to the *execution layer*. The execution layer is an external component that realizes the scaling decision. In real-world systems, cloud applications could be using container orchestration frameworks, which start new or terminate running instances. The execution layer is an interface that passes scaling commands to the cluster control plane. We outsource the execution of the scaling decision to an external component for three reasons. First, we avoid dependencies of the instance scaling logic on particular deployment frameworks. Second, we can also use logging at the execution layer, increasing the explainability of our approach. Third, we can enforce global rules required in production systems, such as maximum or minimum instance numbers. The only rule used in this work is that at least one service instance must be deployed at any time, meaning that every decision made by the service instances is guaranteed to be executed as long as the resulting replica number is greater than zero. However, additional logic might be implemented at the execution layer in future work. Further details on the execution layer are discussed in Section 6.7, and its extensions in Section 6.9.

The last step is determining the amount of time to wait until a new scaling cycle begins. We use a random number w , drawn from a configured distribution W . It is assumed that W yields a minimum $w_{min} > 0$ and a maximum waiting time w_{max} . The advantage of using a probabilistic approach here is that the decisions of the individual scaling instances are distributed over time. The more instances are deployed, the higher the scaling frequency. This means that instead of interval-based autoscaling, we convert autoscaling into a continuous process. As a special case of W , a deterministic distribution can also be used, which, however, can lead to the fact that if several instances start simultaneously, they also make their scaling decisions simultaneously.

This concept can be applied to arbitrary scaling metrics and scaling functions. In the following, we introduce three classes of scaling functions, which we look at in more detail in this work. Figure 6.7 shows a schematic representation of the scaling functions and their parameters. The scaling functions are defined for metrics normalized between 0 and 1. As a concrete example, we focus on CPU-based scaling and choose the utilization $\rho \in [0; 1]$ as our scaling metric. For U and D , monotonic functions are the natural choice in this case. The downscaling function D decreases with increasing utilization ρ . The upscaling function U increases with increasing utilization. This means that:

$$\forall \rho_1, \rho_2 \in [0; 1] : \rho_1 \leq \rho_2 \iff U(\rho_1) \leq U(\rho_2) \wedge D(\rho_1) \geq D(\rho_2).$$

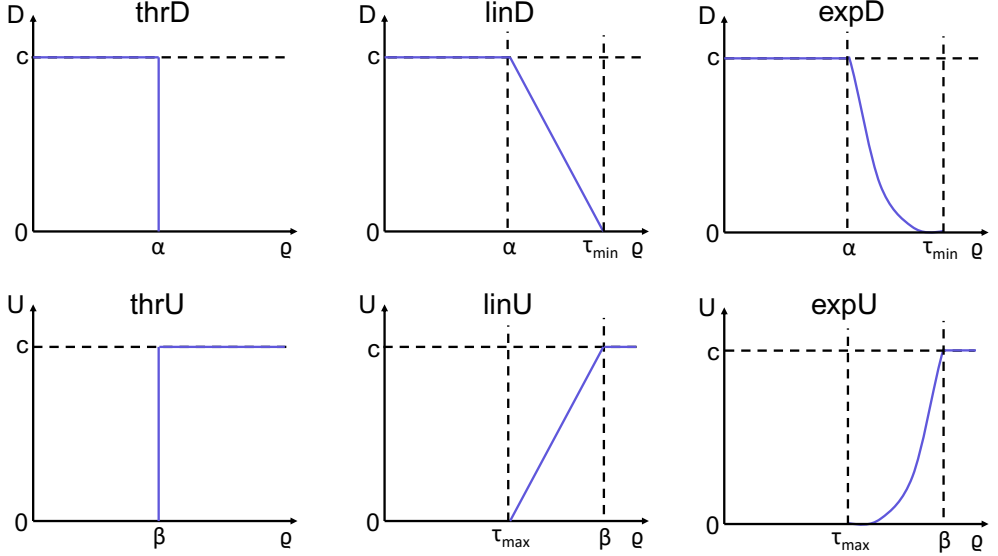


Figure 6.7: Visualization of the used scaling functions and their parameters.

Assuming that neither U nor D are zero functions and in order to satisfy constraint 6.5, a tolerance area τ bounded by τ_{min} and τ_{max} (inclusive or exclusive) is created. In the case that $\tau_{min} < \tau_{max}$, the tolerance area τ is the interval $[\tau_{min}; \tau_{max}]$. This interval could be open, half-open, or closed depending on whether τ_{min} and τ_{max} are inclusive or exclusive bounds.

For all function classes, we introduce the parameter $c \leq 1$. c represents the maximum of the scaling function and, therefore, the maximum probability for a scaling action. This means it is not set that there are input values for which a scaling action is guaranteed. This allows for defining policies that implement a hysteresis-like behavior. As the first class of scaling functions, we consider threshold functions. We denote threshold functions used for downscaling as $thrD_{\alpha,c}$ and define them as follows:

$$thrD_{\alpha,c}(\rho) = \begin{cases} c & 0 \leq \rho < \alpha \\ 0 & \text{else} \end{cases}$$

If a threshold function is used for upscaling, we call this function $thrU_{\beta,c}$ and define it analogously:

$$thrU_{\beta,c}(\rho) = \begin{cases} c & \beta < \rho \leq 1 \\ 0 & \text{else} \end{cases}$$

In addition to threshold functions, we also consider functions with linear and exponential slopes. In the downscaling case, these functions are constructed as follows. Consider utilization values τ_{min} and α with $D(\tau_{min}) = 0$ and $D(\alpha) = c$. As ex-

plained earlier, τ_{min} denotes the lower bound of the tolerance range. linD -functions are linearly decreasing between α and τ_{min} and defined as:

$$\text{linD}_{\alpha, \tau_{min}, c}(\rho) = \begin{cases} \min(c, \frac{\tau_{min} - \rho}{\tau_{min} - \alpha} \cdot c) & \rho < \tau_{min} \\ 0 & \text{else} \end{cases}$$

expD -functions have an exponential slope in the interval $[\alpha; \tau_{min}]$ and are defined as:

$$\text{expD}_{\alpha, \tau_{min}, c}(\rho) = \begin{cases} \min(c, e^{k \cdot (\rho - \alpha)} - 1) & \rho < \tau_{min} \\ 0 & \text{else} \end{cases} \quad \text{with } k = \frac{\log(1 + c)}{\alpha - \tau_{min}}.$$

In the upscaling case, the definitions are similar. We first define two points τ_{max} and β with $U(\tau_{max}) = 0$ and $U(\beta) = c$. linU -functions have a linear gradient between these two points and are defined as:

$$\text{linU}_{\beta, \tau_{max}, c}(\rho) = \begin{cases} \min(c, \frac{\rho - \tau_{max}}{\beta - \tau_{max}} \cdot c) & \rho > \tau_{max} \\ 0 & \text{else} \end{cases}$$

Last but not least, expU -functions have an exponential slope between τ_{max} and β and are defined as:

$$\text{expU}_{\beta, \tau_{max}, c}(\rho) = \begin{cases} \min(c, (1 + c) \cdot e^{k \cdot (\rho - \beta)} - 1) & \rho > \tau_{max} \\ 0 & \text{else} \end{cases} \quad \text{with } k = \frac{-\log(1 + c)}{\tau_{max} - \beta}.$$

In the next section, we will further discuss the properties of our approach, such as convergence and reaction speed, using a discrete-time queueing model. Our focus remains on CPU utilization as the scaling metric, while further scaling metrics are tested in Section 6.8.

6.5 A Discrete-Time Queueing Model for CPU-Based Autoscaling

We evaluate our approach to continuous decentralized autoscaling in three steps. In this section, we present a discrete-time queueing model, which analytically considers the essential properties of our approach. Later in this chapter, we complete our evaluation with a simulation and real-world experiments that also validate the accuracy of the analytical model and the simulation. In this section, we consider the following guiding questions that relate to key performance indicators of our decentralized autoscaler:

1. What is the distribution and expected time to the next scaling decision? (Reaction time)

2. What is the distribution of the time between a change in load until the autoscaler has restored stable operation? (Convergence speed)
3. What is the distribution of the number of instances once the system has reached stability? (Steady state)

In the scope of this section, we distinguish between random variables, distributions, and distribution functions and use the notation convention introduced in Section 2.6. In the first step, we want to calculate the reaction time of our approach. Since the service instances in our approach decide independently at random times, the reaction time is the time until the next decision of any service instance in the system. We consider the decision times of different service instances to be independent of each other. This is particularly the case if a sufficient number of service instances are deployed or the service instances have already been deployed for a sufficient time. The edge case in which only one instance is deployed and then another instance is created whose decision times in the first iterations correlate with those of the older instance is not considered separately. Since all service instances have the identical waiting time distribution W , the time between two scaling decisions can be modeled as the superposition of several identical stochastic processes.

In the following, we perform the general calculation for discrete distributions and show numerical results for a concrete example. Solutions for continuous distributions are also possible but are not considered in detail in this work due to the increased computational effort. In addition, digital computers only use discrete calculations anyway due to limited floating point precision. In order to employ the terms commonly used in the field of queueing networks, we consider the decisions of individual instances as events. The time between events is then the inter-arrival time. We consider the inter-arrival times of an instance i as a random variable X_i . X_i follows the discrete distribution W with maximum w_{max} described by the PMF $w(t) = x_i(t)$ and CDF $W(t) = X_i(t)$ and with expected value $E[W] = E[X_i]$. The sequence of decision times as a stochastic process is a renewal process since the inter-arrival intervals are all independent and identically distributed. In the case of multiple deployed service instances, the events of n independent (non-synchronized) processes described by random variables X_1, X_2, \dots, X_n are merged into a superimposed process with inter-arrival times \bar{X}_n . We are interested in two properties of this superimposed process. First, we derive the forward recurrence time described by the random variable \bar{R}_n , which in practice describes the time to the next scaling decision (reaction time). In a second step, we then consider the inter-arrival times \bar{X}_n of the superimposed process, giving us information about the frequency of scaling decisions. Our derivations are based on a work by Hasslinger and Rieger [HR96].

Since the n independent processes are all renewal processes, the following relationship exists between forward recurrence times and inter-arrival times for one instance, assuming that our observation is just after discretized time instants [TGH21]:

$$r_i(t) = \frac{1 - P(X_i < t)}{E[X_i]} = \frac{P(X_i \geq t)}{E[X_i]} = \frac{1}{E[W]} \cdot \sum_{i=t}^{w_{max}} w(i). \quad (6.7)$$

The forward recurrence time of the superimposed process \bar{R}_n is the minimum of the forward recurrence times of the n active instances R_1, R_2, \dots, R_n :

$$\bar{R}_n = \min(R_1, R_2, \dots, R_n). \quad (6.8)$$

To calculate the minimum of n random variables, the following relation can be used in general [TGH21]:

$$\bar{R}_n(t) = 1 - \prod_{i=1}^n (1 - R_i(t)). \quad (6.9)$$

Since all instances have the same recurrence time distribution in our case, the product can be replaced by a power. By using Equation 6.7, we obtain the following relationship between the forward recurrence time of the superimposed process and the waiting time distribution:

$$\bar{R}_n(t) = 1 - (1 - R(t))^n = 1 - \left(1 - \sum_{i=1}^t \left(\frac{1}{E[W]} \cdot \sum_{j=i}^{w_{max}} w(j) \right) \right)^n. \quad (6.10)$$

We can then calculate the PMF of the forward recurrence time iteratively from its CDF:

$$\bar{r}_n(t) = \begin{cases} \bar{R}_n(1) & t = 1 \\ \bar{R}_n(t) - \bar{R}_n(t-1) & t > 1 \end{cases} \quad (6.11)$$

This superimposed process is generally non-renewal. Especially for a low number of component processes, the inter-arrival times can be correlated. However, the superposition is often approximated using a renewal representation [HR96], as these correlations weaken/effectively cancel each other out also for a few overlaid processes. Making this approximation, we can calculate the PMF of the inter-arrival times from the PMF of the forward recurrence time. Starting with the general relationship between forward recurrence time and inter-arrival time in a renewal process shown in Equation 6.7, we first rearrange the equation to:

$$\sum_{i=t}^{w_{max}} \bar{x}_n(i) = E[\bar{X}_n] \cdot \bar{r}_n(t). \quad (6.12)$$

To get the desired formula for $\bar{x}_n(t)$, we express $\bar{x}_n(t)$ as the difference between two successive sums:

$$\bar{x}_n(t) = \sum_{i=t}^{w_{max}} \bar{x}_n(i) - \sum_{i=t+1}^{w_{max}} \bar{x}_n(i). \quad (6.13)$$

Inserting Equation 6.12 gives us:

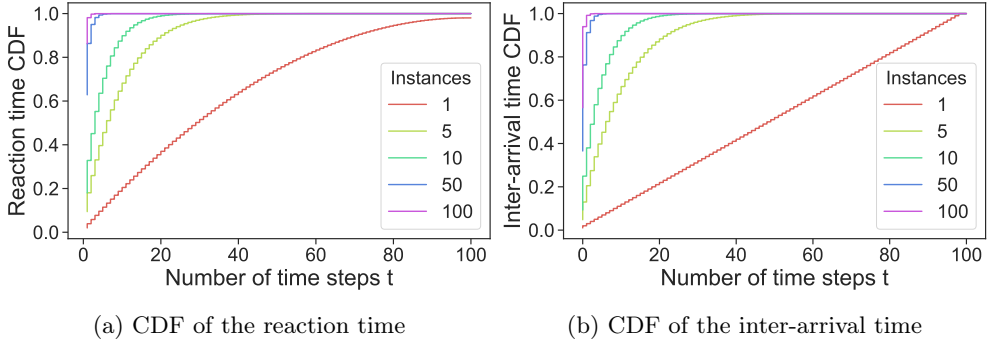


Figure 6.8: Statistical properties of the superimposed process.

$$\bar{x}_n(t) = E[\bar{X}_n] \cdot \bar{r}_n(t) - E[\bar{X}_n] \cdot \bar{r}_n(t+1) = E[\bar{X}_n] \cdot (\bar{r}_n(t) - \bar{r}_n(t+1)). \quad (6.14)$$

Considering the extreme case that $\bar{r}_n(w_{max} + 1) = 0$ and calculating $\bar{x}_n(0)$ as the complementary probability from the sum of all other cases, gives us the final desired relationship:

$$\bar{x}_n(t) = \begin{cases} 1 - \bar{r}_n(1)/\lambda & t = 0 \\ (\bar{r}_n(t) - \bar{r}_n(t+1))/\lambda & 0 < t < w_{max} \\ \bar{r}_n(w_{max})/\lambda & t = w_{max} \end{cases} \quad \text{with } \lambda = \sum_{i=1}^n \frac{1}{E[W]} = \frac{n}{E[W]}. \quad (6.15)$$

Figure 6.8 shows the CDF of the reaction time $\bar{R}_n(t)$ and the CDF of the inter-arrival times $\bar{X}_n(t)$ for the superimposed process. In this case, a discrete uniform distribution between 1 and 100 was chosen for the waiting time distribution W of the individual instances. The different lines represent the distribution for different numbers of instances n . We see that as n increases, both the reaction times and the inter-arrival times of the superimposed process decrease considerably. As an example, we consider the probability that the reaction time is at most 5 time steps. In the case that only one instance is deployed, this is approximately 9.6%. For $n = 5$ it is 39.6%, for 10 instances 63.6%. For 50 or more instances, the probability is more than 99%. Overall, we can see that our approach converges to a quasi-continuous process and that low reaction times are likely even for small numbers of instances.

In the next step, we develop a discrete-time queuing model that enables the calculation of both the duration until the system converges towards a stable state and the distribution of the expected number of instances based on easily obtainable input parameters. Figure 6.9 shows the development of this model over time and highlights critical aspects and parameters of the model. We abstract the whole system as a single entity and a fluctuating number of instances is modeled as a changing service rate. We describe the system via an embedded Markov chain with embedding times right before

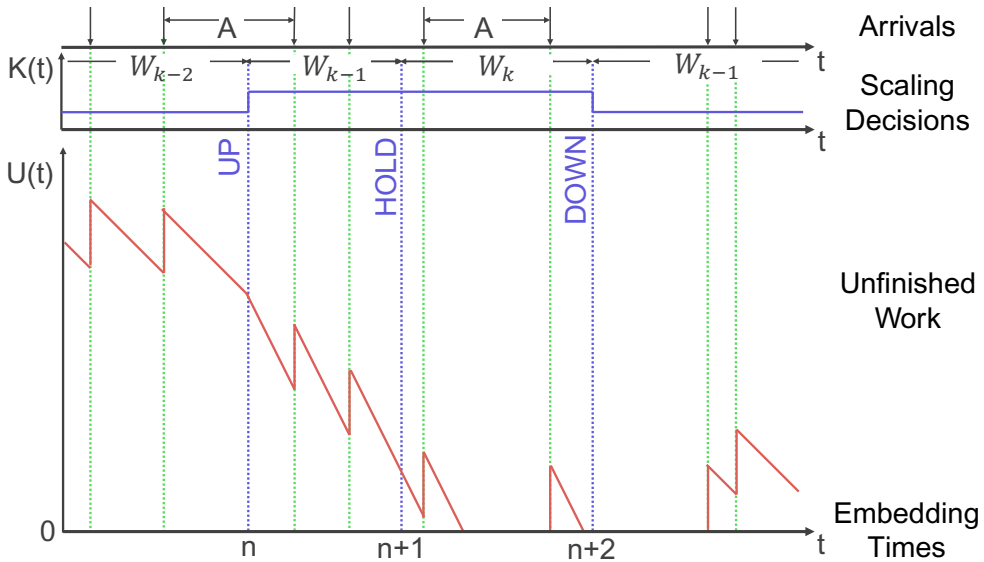


Figure 6.9: Evolution of the discrete-time queuing model.

each scaling decision. Each time step i is characterized by two state variables. First, the current number of active instances K . K changes over time as a result of up- and downscaling decisions. The second state variable is the amount of unfinished work U in the system, corresponding to the number of unprocessed requests. In general, the arriving requests are characterized through their inter-arrival time A and arrival rate $\lambda = 1/E[A]$. Arrivals increase the amount of work in the system. All available instances are continuously processing the unfinished work in the system. Hence, the rate (slope) with which the unfinished work ($U(t)$ in Figure 6.9) declines depends on the current scale of the system and may change at every decision time, depending on whether the number of instances K increases, decreases, or remains unchanged. The service rate of a single instance is denoted as μ . The time between two scaling decisions of a single instance is given the waiting time distribution W . We can subsequently solve the model by means of fixed-point iteration by tracking U_i and K_i iteratively at each subsequent embedding time i . Table 6.3 summarizes all input and output parameters for the model.

To realize the aforementioned fixed-point iteration, we define the following transition functions to compute the system state (U_i, K_i) at embedding time i based on their values at embedding time $i - 1$. Using these transition functions, it is then possible to iteratively compute the system state at the next embedding time based on the initial system state at embedding time 0, described by K_0 and U_0 .

First, we focus on the computation of the unfinished work U . Within one time step, we can describe the change of unfinished work ΔU by the difference of the arriving work U_A and the processed work U_B . As discussed in Section 2.6, the difference between two non-negative independent discrete random variables can be described

Parameter	Description
λ	Arrival rate of requests at the system
μ	Service rate of a single instance
$W, w(x)$	Inter-decision time of a single instance
$U_i, u_i(x)$	Unfinished work at embedding time i
$K_i, k_i(x)$	Number of instances at embedding time i
$\bar{\phi}_i$	Average system load in interval between embedding times $i - 1$ and i
ϕ_i	60 sec average system load at embedding time i
ρ_i	60 sec average utilization at embedding time i

Table 6.3: Parameters of the queuing model.

using a convolution-like operation. The arriving and processed work depend on the inter-arrival time of the superimposed process \bar{X}_k and hence on the number of active instances K as they are proportional to the inter-decision time. Hence, to calculate the PMF of ΔU , the result of the convolution is weighted with every possible value of K :

$$\Delta u_i(x) = \sum_{k=-\infty}^{+\infty} (u_{A,k}(x) * u_{B,k}(-x)) \cdot k_{i-1}(k). \quad (6.16)$$

Thereby, $k_{i-1}(k)$ is the probability that k instances were deployed at time step $i - 1$. The amount of arriving and processed work is equal to the number of arrivals and departures occurring within each inter-decision interval. The distributions of these quantities are computed according to Gebert et al. [GZL⁺16]:

$$\begin{aligned} u_{A,k}(x) &= \bar{x}_k(0) \cdot \delta_0(x) + \sum_{i=1}^{\infty} \bar{x}_k(i) \cdot \sum_{j=0}^{i-1} \left(f^{(x)}(j) - f^{(x+1)}(j) \right) \\ u_{B,k}(x) &= \bar{x}_k(0) \cdot \delta_0(x) + \sum_{i=1}^{\infty} \bar{x}_k(i) \cdot \sum_{j=0}^{i-1} \left(g^{(x)}(j) - g^{(x+1)}(j) \right) \end{aligned} \quad (6.17)$$

Thereby, $\delta_0(x)$ is a special case of the deterministic distribution with PMF $\delta_y(x)$ whose entire probability mass is at y . Hence, $\delta_y(x) = 1$ if $x = y$. In any other case, $\delta_y(x)$ takes the value of 0. $F^{(x)}$ with PMF $f^{(x)}(j)$ is a random variable describing the time from the start of the observation interval to the x -th arrival. It is the sum of the forward recurrence time and $(x - 1)$ -times the inter-arrival time. As the sum of random variables can be computed using discrete convolutions, x convolutions are necessary for the computation. Refer to the work of Gebert et al. [GZL⁺16] for a step-by-step derivation. Analogously, random variable $G^{(x)}$ with PMF $g^{(x)}(j)$ describes the time from the start of the observation interval to the x -th departure. The departure

process can be computed analogously to the arrival process and is the superposition of k independent instance departure processes, each following a service time distribution with rate μ .

Now that we know the change of unfinished work within one time step, we can calculate the unfinished work at time i by adding ΔU_i to U_{i-1} . The sum of discrete random variables is again calculated using a convolution. Further, note that while the change in unfinished work within a time step might be negative (if more work can be processed than arrives), the unfinished work after a time step can not be negative. Therefore, we apply the left-side sweep operator π_0 introduced in Section 2.6 to the result of the convolution:

$$u_i(x) = \pi_0(\Delta u_i(x) * u_{i-1}(x)). \quad (6.18)$$

Based on the amount of unfinished work in the system, we can compute the system load in the interval since the previous embedding time for specific realizations of K_i and U_i . Contributing to the system load are, on the one hand, the arriving requests described by their rate λ . On the other hand, the unfinished work in the system also contributes to the load. The rate of unfinished work is defined as the amount of unfinished work u divided by the expected length of the interval $E[\bar{X}_k]$. All requests are processed with rate μ . Note that the system load represents a demand metric and is not equal to the utilization, which we will approximate later. Notably, the system load $\bar{\phi}_i(k, u)$ could grow beyond one and even beyond k . We get the following relationship for the computation of $\bar{\phi}_i(k, u)$:

$$\bar{\phi}_i(k, u) = \frac{1}{\mu} \cdot \left(\frac{u}{E[\bar{X}_k]} + \lambda \right). \quad (6.19)$$

Subsequently, the overall observed load at embedding time i can be computed as:

$$\bar{\phi}_i = \sum_{k=-\infty}^{+\infty} \sum_{u=-\infty}^{+\infty} \bar{\phi}_i(k, u) \cdot k_{i-1}(k) \cdot u_{i-1}(u). \quad (6.20)$$

In accordance with UNIX systems [Gun10], we further model an exponential moving average approach with exponentially decaying weights to compute the mean weighted load ϕ_i over a specified historical interval. Similar to the technical system, we use the last 60 seconds to model the average load of the system and compute the average load for specific realizations of K_i and U_i :

$$\phi_i(k, u) = \frac{\bar{\phi}_i(k, u) + \sum_{j=1}^t \bar{\phi}_{i-j} \cdot e^{-j}}{\sum_{j=0}^t e^{-j}} \quad (6.21)$$

with $t : \sum_{j=0}^{t-1} E[W^{i-j}] \leq 60 < \sum_{j=0}^t E[W^{i-j}]$.

Thereby, W^i represents the inter-decision time at embedding time i and has the following PMF:

6.5 A Discrete-Time Queueing Model for CPU-Based Autoscaling

$$w^i(x) = \sum_{k=-\infty}^{+\infty} \bar{x}_k(x) \cdot k_i(k). \quad (6.22)$$

Analogously to the non-weighted load above, we can now calculate the overall weighted load by:

$$\phi_i = \sum_{k=-\infty}^{+\infty} \sum_{u=-\infty}^{+\infty} \phi_i(k, u) \cdot k_{i-1}(k) \cdot u_{i-1}(u). \quad (6.23)$$

Although the system load is an established metric to assess CPU demand and performance, autoscalers normally use the utilization ρ as the scaling metric. Under the assumption of perfect load balancing, we can approximate the utilization by dividing the system load ϕ by the number of instances k and applying a minimum function, as the utilization by design cannot grow beyond one:

$$\rho = \min\left(1, \frac{\phi}{k}\right). \quad (6.24)$$

Inserting this relationship into Equation 6.23 gives us the formula for computing the overall utilization at embedding time i :

$$\begin{aligned} \rho_i(k, u) &= \min\left(1, \frac{\phi_i(k, u)}{k}\right) \\ \rho_i &= \sum_{k=-\infty}^{+\infty} \sum_{u=-\infty}^{+\infty} k_{i-1}(k) \cdot u_{i-1}(u) \cdot \min\left(1, \frac{\phi_i(k, u)}{k}\right). \end{aligned} \quad (6.25)$$

Using the system utilization as its input, we can now apply the previously introduced scaling policy to determine the change in the number of running instances for a specific realization of K_i :

$$(S_i(k))(x) = \sum_{u=-\infty}^{+\infty} (S(\rho_i(k, u)))(k, x) \cdot u_{i-1}(u). \quad (6.26)$$

Here, $S(x)$ represents the system-specific scaling policy that combines the upscaling function U and downscaling function D from Section 6.4. Given a concrete number of instances k and system utilization ρ , we can calculate the probabilities to scale down (instance number decreases by 1), hold (instance number remains unchanged), or scale up (instance number increases by 1), respectively.

$$(S(\rho))(k, x) = \begin{cases} D(\rho) & x = k - 1 \\ 1 - \max(D(\rho), U(\rho)) & x = k \\ U(\rho) & x = k + 1 \end{cases} \quad (6.27)$$

Hence, the result $(S_i(k))(x)$ of Equation 6.26 represents the probabilities for scaling actions given a concrete realization of K_i . In the final step, we can compute the

complete distribution of running instances at embedding time i based on the system state at the previous embedding time and the scaling decisions resulting from this system state. We use the π_0 -operator to ensure that instance numbers are always greater or equal to 0:

$$k_i(x) = \sum_{k=-\infty}^{+\infty} \pi_0((S_i(k))(x)) \cdot k_{i-1}(k). \quad (6.28)$$

Using all the above equations, we can iteratively compute the system state until the model converges at embedding time j and $k_j(x) = k_{j+1}(x)$ holds for all x and embedding times $i > j$ to answer the previously posed questions. This model-based approach can track and output entire distributions for the state variables. This stands in contrast to the simulation model that is introduced in the next section. One simulation run always considers only one system state per iteration. In contrast to benchmarks using a real cloud application setup, the model makes it easy to evaluate different parameter combinations and their impact on the stability and convergence of the autoscaling approach.

Figure 6.10 shows the evolution of essential state variables of the model with expected value and standard deviation in an experiment with a load increase and decrease. The system is initialized with eight instances. Initially, the system is stressed with a load of 20 rps; after 900 seconds, the load jumps to 80 rps before dropping back to 20 rps after 2700 seconds. The scaling policy has a tolerance area between 60% and 80% utilization; the probabilities for up- and downscaling increase exponentially with increasing distance to this tolerance area. Our model's embedding times are always right before the next scaling decision. Consequently, the time interval between two embedding times is determined by the distribution of the inter-decision times. The projection of the results of the fixed-point iteration onto the time axis is done by summing up the distribution of the inter-decision time and taking the expected value in each step. We have opted for this type of presentation as it is easier to interpret.

We see that when the load increases, the unfinished work in the system rises to over 1000 unanswered requests on average. This leads to increased instance utilization, and our scaling policy triggers the addition of new instances. As the number of instances increases, the unfinished work decreases, and the system settles into a steady state in which the distribution of instance utilization is entirely within the tolerance range. The steady state is also reached after downscaling. Here, the unfinished work drops significantly in average and variance during the load decline, which leads to a drop in instance utilization and, thus, to downscaling. For this configuration, the model estimates a low probability that the system overshoots downwards, meaning it removes too many instances. This leads to a counter-reaction explaining the slight increase in the number of instances on average. The inter-decision times behave inversely to the number of instances; if more instances are deployed, the inter-decision time is shorter. In this example, with about 22 instances, a scaling decision occurs about every 2 seconds, while the waiting time of each individual instance is equally distributed between 30 and 60 seconds. In Section 6.7, we compare these model results to both

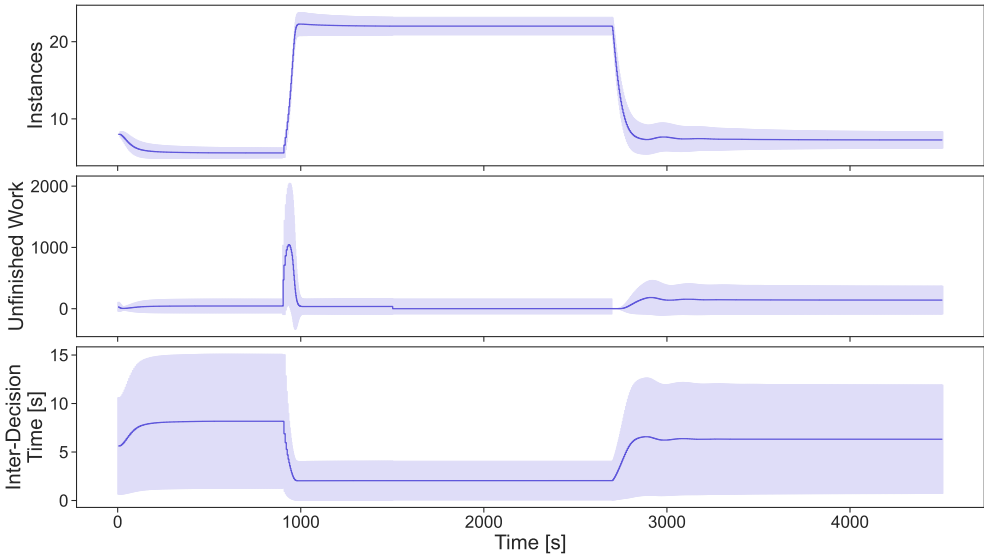


Figure 6.10: Evolution of the model’s state variables in a scenario with changing load.

measurements in the real system and our simulation model and provide a detailed description of other experiment parameters.

6.6 Simulation Study

We have developed a discrete event simulation using the *simmer*¹¹ framework to evaluate our approach further. The simulation implements the system model described in Section 6.4. Our simulation supports both trace-based and model-based arrival streams. In addition to our algorithm parameters W , U , and D , the simulation allows us to test different service time distributions, start time distributions, and load balancer policies. The simulation applies the same instance selection policy as Kubernetes to determine which instance should be stopped in case of downscaling (by default: last in, first out). As output, the tool displays various metrics such as response times, instances over time, instance utilization, and more. In the following, we state and answer three research questions that are related to the configuration, scalability, and robustness of our approach with the help of simulation:

1. What role does the system scale play, that is, the number of instances and the total number of incoming requests? (Scalability)
2. How do typical influencing factors from real environments affect our approach? (Robustness)

¹¹<https://r-simmer.org>

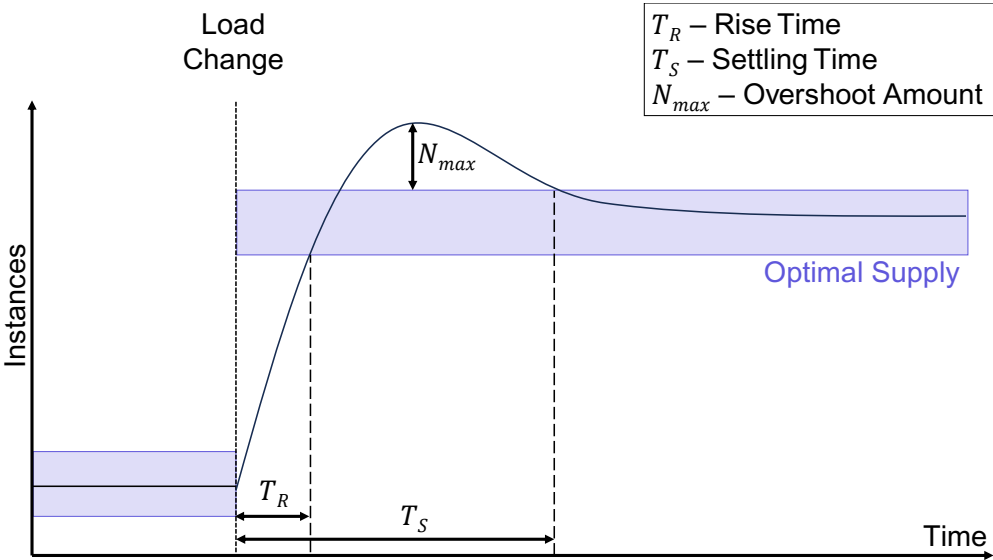


Figure 6.11: Illustration of step response metrics.

3. What is the influence of the waiting time distribution W and the scaling functions U and D on the scaling behavior? (Configuration)

Table 6.4 shows an overview of our simulation study and the parameter settings we used. We conducted a series of experiments where every experiment investigates different aspects and modifies categories of parameters. The columns represent experiments described in this section. The column name indicates the focus of the experiment. Cells marked with * indicate that different values were chosen for this parameter in the experiment. These values are stated explicitly in the corresponding experiment description. We give the parameter overview here to keep the description of the experiment results compact.

First, we analyze the role of the system scale, that is, the number of incoming requests and active instances in the system. We aim to analyze whether our approach can be used with a fixed configuration (consisting of W , U , and D) for different system scales. For this, we use a well-known methodology from control theory. As input, we use a step function that increases the load by a factor of 10 after 1350 seconds. Our evaluation metrics (rise time, overshoot amount, and settling time) follow definitions from control theory and are visualized in Figure 6.11.

In the following, t_0 denotes the time the load on the system is changed. λ is the arrival rate at the system after the load increase. μ denotes the service rate. In order to calculate the step response metrics, the system's desired target state G after the load jump must be defined. In our case, it is an interval in which the number of deployed instances should be. G results directly from the scaling functions U and D and the derived tolerance range for the utilization $\tau = [\tau_{min}; \tau_{max}]$. G includes all

Parameters	Experiment Focus						
	Scalability	Timeouts	Readiness Times	CPU Throttling	Load Balancer	Non-Markov Service Time	Parameter Study
Repetitions	5	30	30	30	30	30	30
Parameters of the decentralized autoscaler							
Waiting Dist. W	unif(30s, 90s)	unif(30s, 90s)	unif(30s, 90s)	unif(30s, 90s)	unif(30s, 90s)	unif(30s, 90s)	*
Upscaling Fct. U	exp $U_{0.8,0.95,1}$	exp $U_{0.8,0.95,1}$	exp $U_{0.8,0.95,1}$	exp $U_{0.8,0.95,1}$	exp $U_{0.8,0.95,1}$	exp $U_{0.8,0.95,1}$	*
Downscaling Fct. D	exp $D_{0.2,0.6,1}$	exp $D_{0.2,0.6,1}$	exp $D_{0.2,0.6,1}$	exp $D_{0.2,0.6,1}$	exp $D_{0.2,0.6,1}$	exp $D_{0.2,0.6,1}$	*
Workload Parameters							
Shape	Step	Step	Step	Step	Step	Step	Combined
Min Rate [rps]	*	50	50	50	50	50	120
Max Rate [rps]	*	500	500	500	500	500	548
Other Parameters							
Service Time Dist.	Exponential	Exponential	Exponential	Exponential	Exponential	*	Exponential
\emptyset Service Time [ms]	200	200	200	*	200	*	200
Request Timeout [s]	10	*	10	10	10	10	10
Readiness Time [s]	0	0	*	0	0	0	Measured
CPU Throttling	off	off	off	*	off	off	off
Load Balancer	Random	Random	Random	Random	*	Random	Random

Table 6.4: Experiment and parameter overview of our simulation study.

instance numbers k for which the mean instance utilization lies within the tolerance range τ . Using the general definition of the utilization $\rho = \lambda/\mu$, G results as:

$$G = \left[\frac{\lambda}{\tau_{max} \cdot \mu}; \frac{\lambda}{\tau_{min} \cdot \mu} \right]. \quad (6.29)$$

In the following, let $I(t)$ be the time series of deployed instances. The rise time T_R is defined as the time difference between the time of the load increase t_0 and the first time the target state G is reached:

$$T_R = \min(\{t : I(t) \in G\}) - t_0.$$

The settling time T_S is defined as the time difference between t_0 and the point in time at which $I(t)$ has a value in G and after which all further values of $I(t)$ lie in G :

$$T_S = \min(\{t : (I(t) \in G) \wedge (\exists \bar{t} < t : I(\bar{t}) \notin G)\}) - t_0.$$

Last but not least, the overshoot amount N_{max} is defined as the difference between the maximum of $I(t)$ and the maximum of G :

$$N_{max} = \max(I(t)) - \max(G) = \max(I(t)) - \frac{\lambda}{\tau_{min} \cdot \mu}.$$

Using these step response metrics, we evaluate the behavior of our approach with a fixed configuration for different system scales. In the reference case (scale $s = 1$), the load increases from 3 rps to 30 rps. In all other cases, the arrival rate before and after the jump is multiplied by the scale factor. This means that in the case $s = 100$, the load increases from 300 rps to 3000 rps. We use a uniform distribution between 30 and 90 seconds for W . The desired range for CPU utilization is between 60% and 80%. We use exponential up- and downscaling functions. A DOWN decision is guaranteed if the utilization is smaller than 20%. Upscaling is guaranteed if the utilization is at least 95%. The service time follows an exponential distribution with an expected value of 200 ms. The scenario was repeated with five different random seeds.

Table 6.5 shows our evaluation metrics and their standard deviations for different system scales. Note that the rise and settling times are nearly equal for any scale. This is because a higher system scale increases the load jump in absolute numbers but concurrently increases the potential elasticity and number of scaling decisions due to the higher number of deployed instances. In the case $s = 100$, inter-decision times are lower than 100 ms on average. The normalized overshoot gets less with increasing system scale because of the increased absolute size of the tolerance area. The results show that the policy provides consistent and predictable performance for our approach at different system scales. We achieve this even though there is no coordination among instances. Especially in large-scale systems, our advantage is that we do not rely on a scalable monitoring system, where data has to be collected and processed from many instances. In our approach, the monitoring overhead is stable at every scale because the instances monitor themselves.

Next, we analyze how our approach is influenced by other factors that occur in production cloud environments but that we have not considered in the model-based

Scale Factor s	T_R [s]	T_S [s]	N_{max}/s
1	241.6 ± 24.8	268.3 ± 68.2	1.00 ± 1.26
5	246.5 ± 9.9	322.9 ± 26.8	0.64 ± 0.23
10	238.9 ± 10.1	300.0 ± 61.7	0.36 ± 0.42
50	237.8 ± 7.5	311.4 ± 10.2	0.27 ± 0.03
100	240.1 ± 6.1	310.2 ± 7.9	0.21 ± 0.05

Table 6.5: Step response metrics for different system scales.

analysis from Section 6.5. These factors are external factors that cannot be controlled by the autoscaler but still influence its performance.

Request timeouts are a fixed part of many network protocols, such as TCP or HTTP. Clients abort requests if they are not acknowledged or answered within a specific time. Different solutions exist on the server side to detect stale requests (e.g., by checking the underlying TCP connection). In the following, we assume that the processing time of a timed-out request at an instance is 0. In this experiment, we analyze our system’s response to a sudden increase in load from 50 to 500 rps. Like in our system model from Section 6.4, the simulation generates the arrival stream with a constant rate from an infinite number of clients. Hence, we do not model single clients’ behavior and reactions to request timeouts.

Figure 6.12 shows the mean of 30 runs of the number of deployed instances converging into the tolerance area indicated by the gray box. The tolerance area is computed using Equation 6.29. We use the same configuration of the decentralized autoscaler and service time distribution as in the scalability experiment described above. We see that request timeouts accelerate the convergence of our scaling process after an overload situation. Immediately after the load increases, the queues of the deployed instances fill up very quickly. If new instances are deployed, they never reach the queue size of the older instances due to the lower instance arrival rate. This means that the unfinished work in the system is unevenly distributed. Request timeouts lead to the system’s queue sizes equalizing more quickly, as unfinished work on the old instances is eliminated. As Table 6.6 shows, timeout values of 10, 20, 30, and 60 seconds significantly reduce the overshoot amount and the settling time compared to the case with no timeouts. Lower timeout values lead to better convergence while more requests are dropped.

Our model-based analysis assumes that new instances are spawned immediately after an upscaling decision. In practice, container readiness times typically range from a few milliseconds to tens of seconds (see Chapter 5), while other runtimes, such as VMs, might have even higher initialization times. Figure 6.13 shows that non-zero readiness times degrade the performance of our approach in the step response experiment. We evaluate constant readiness times from 0 to 60 seconds in 5-second steps. We see a clear increase in settling time and overshoot amount. The result of a linear regression shows a high likelihood of a linear dependence both for setting time (R^2 score: 0.898,

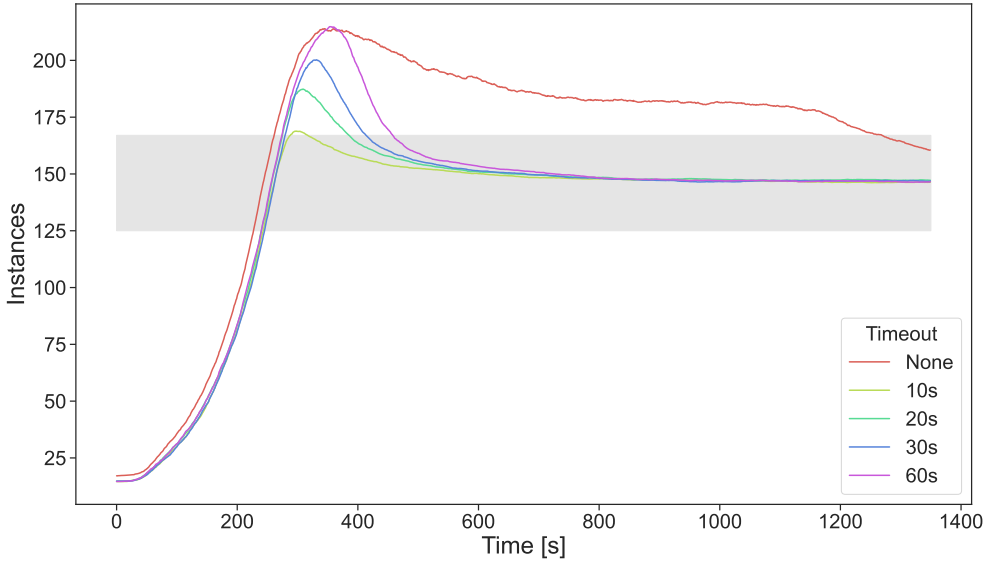


Figure 6.12: Deployed instances over time for different timeout values.

Timeout	T_S [s]	N_{max}
10 s	295.6 ± 41.1	3.7 ± 4.1
20 s	381.2 ± 16.5	23.3 ± 4.8
30 s	411.9 ± 17.4	37.2 ± 4.5
60 s	455.3 ± 15.8	51.6 ± 5.1
None	1261.5 ± 66.0	52.9 ± 4.9

Table 6.6: Step response metrics for different timeout values.

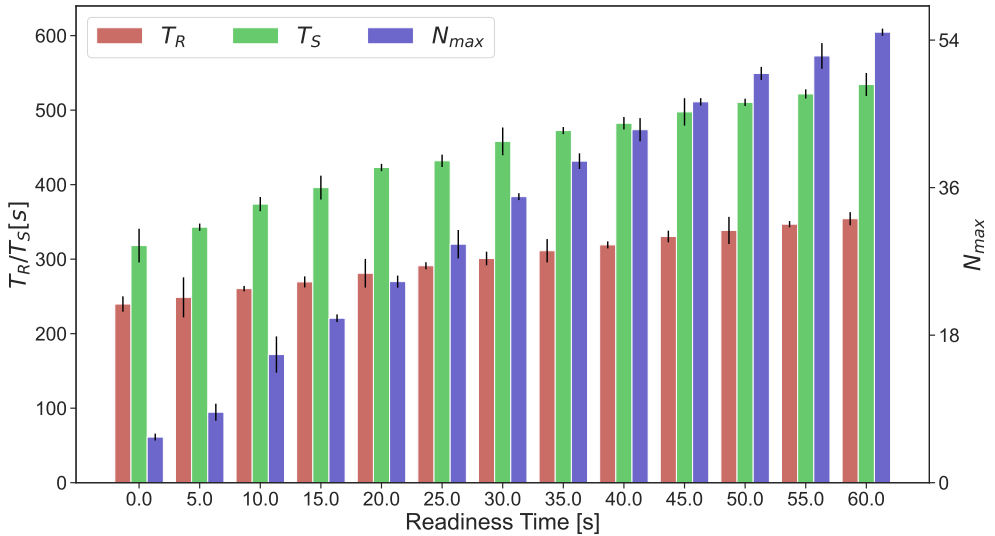


Figure 6.13: Step response metrics for different readiness times.

p -value: $8e-195$) and overshoot amount (R^2 score: 0.911, p -value: $2e-205$). The results are as expected, as readiness times delay the implementation of upscaling decisions. This prolongs the transient phase in which the unfinished work is unevenly distributed.

As part of our simulation study, we also investigated other factors potentially influencing our scaling behavior. However, none of these factors significantly impacted our autoscaler evaluation metrics. In practice, the resources of containers can be limited to create better performance isolation. Container runtimes and VM hypervisors support CPU throttling to enforce such limits. We implemented the default throttling procedure of Docker and Kubernetes (see Section 2.2). We found that the scaling behavior quantified by settling time and overshoot amount is identical within the standard deviation when we compared a throttled system with CPU share $1/s$ and an unthrottled system with an s -times higher service time. This is also validated through the visual impression, as Figure 6.14 shows. The primary motivation was that simulating CPU throttling makes the simulation runs very computationally intense and slow. As we focus on evaluating the scaling behavior here, we conclude that it is sufficient to evaluate unthrottled systems in our simulative analysis.

In addition, we tested different simple load balancing policies: round-robin, shortest queue, random, and power of d . In the power-of- d load balancing scheme, the instance with the lowest utilization is selected from d randomly selected instances. Also, we tested non-Markovian service time distributions with different variation coefficients and auto-correlation. This emulates more complex processing patterns where effects like I/O-waits or external dependencies can be modeled. Like CPU throttling, all these factors mainly affect the response time distribution but not the scaling behavior, especially not the convergence of our method.

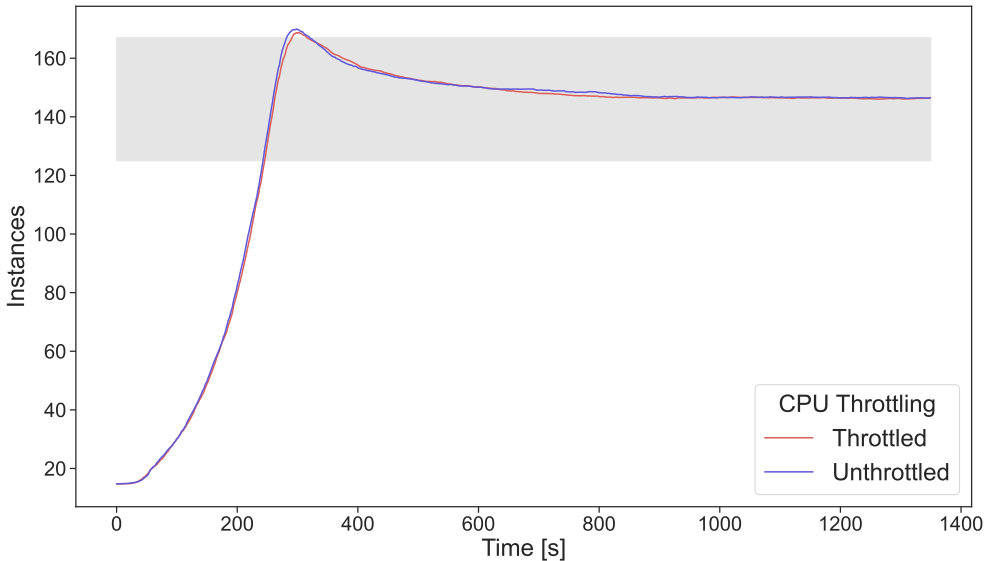


Figure 6.14: Comparing CPU-throttled application and its unthrottled equivalent.

In the following, we use our simulation to analyze the autoscaler performance with respect to our algorithm parameters U , D , and W . First, we show the intuitive effect of different configurations in an illustrative example. In the second step, we present the results of an extensive parameter study. Figure 6.15 shows the scaling behavior of eight configurations of our algorithm. The workload consists of one sudden load increase and decrease. The diagrams show the deployed instances over time. The optimal supply over time calculated using Equation 6.29 is marked as a blue area in the background. The top left is the reference configuration with a moderate up- and downscaling behavior. The other seven figures each represent configurations where one parameter has been changed. Here, we have deliberately chosen to visualize the results of only one run in Figure 6.15 to avoid slanting effects due to averaging.

In the first example, we reduced the mean of W from 60 seconds to 15 seconds compared to the reference configuration. The reaction time to load changes is reduced, but continuous scaling occurs even under constant load as the scaling frequency is high. With a more aggressive U , the probability that a new instance is spawned is high, even for values close to the tolerance range. With a conservative U , only values close to 100% utilization cause scaling. An aggressive U accelerates the upscaling period, but we observe a high overshoot and an increased fluctuation of the instance numbers even in the tolerance range. A conservative U prevents overshoot and provides more stability at constant load. These results can also be transferred to the D parameter. Finally, we consider the influence of the tolerance area. In the reference case, this lies between 60% and 80% utilization. Figure 6.15 shows the results with a larger range between 30% and 80% and a configuration where only exactly 80% utilization

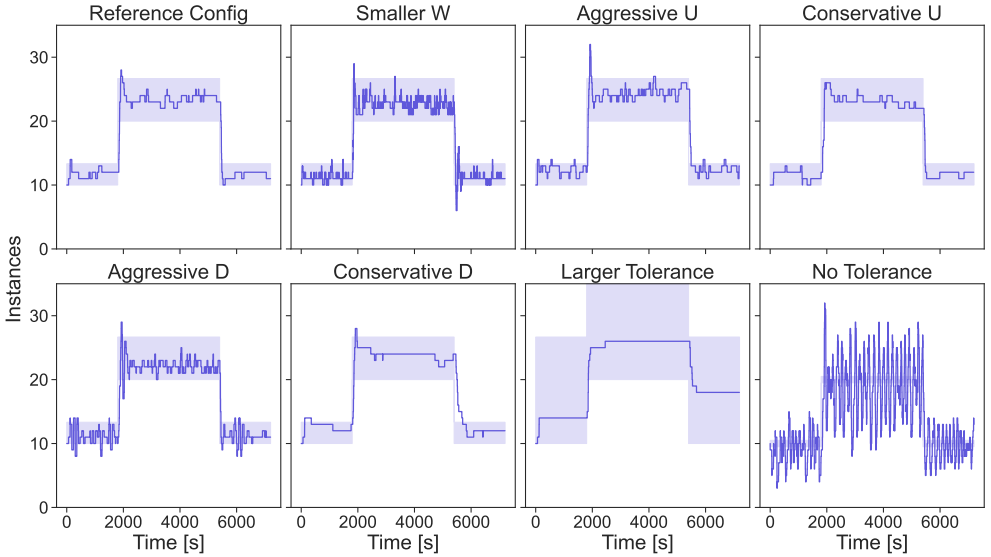


Figure 6.15: Examples illustrating the influence of various configuration parameters.

guarantees no scaling. Choosing a larger tolerance reduces the number of scaling decisions significantly. In contrast, permanent up-and-down scaling occurs in the scenario with no tolerance. In summary, we can see in this intuitive example that our algorithm parameters W , U , and D have an explainable and measurable influence on the scaling behavior.

We see that configuring the scaling behavior is not trivial and that trade-offs exist, so there is no single optimal configuration. In the following, we present the results of a parameter study examining the trade-off between deployment cost and quality of service in more detail. We emulate the setup from our motivation example in Section 6.3 in terms of test trace, a request timeout of 10 s, and a measured readiness time distribution with a mean of 2 s. For W , we choose deterministic, normal, and uniform distributions with mean values between 15 and 90 seconds. For U and D , we consider monotone functions of three groups. For *lin-* (*exp-*) functions, the probability of up- and downscaling increases linearly (exponentially) with the distance from the tolerance range. Furthermore, we consider *thr*-functions that always scale up or down from a certain threshold. Table 6.7 describes our full parameter search space. In our grid search, we test a total of 3390 configurations, each of which is repeated with five random seeds. We always consider the mean values from these five repetitions in the following.

Figure 6.16 shows the key insights of our parameter study. The upper left shows the trade-off between the fraction of requests with a response time of less than 2 seconds and deployment costs. The Pareto front shows a linear cost increase for increasing quality of service up to about 90%. Beyond that, the costs increase exponentially for higher performance.

Parameter	Tested Values
W	Deterministic distributions with values of 15s, 30s, and 60s Normal distributions $N(\mu, \sigma)$: $N(30s, 5s)$, $N(60s, 10s)$, $N(90s, 15s)$ Uniform distributions $U(a, b)$: $U(15s, 90s)$, $U(30s, 90s)$, $U(15s, 120s)$, $U(30s, 120s)$
U	Threshold-based thrU functions with $\beta \in \{0.7, 0.8, 0.9\}$ and $c = 1$ Linear Upscaling linU with $\tau_{max} = 0.8$, $\beta = 0.95$, and $c \in \{0.3, 0.5, 1\}$ Exponential Upscaling expU with $\tau_{max} = 0.8$, $\beta = 0.95$, and $c \in \{0.3, 0.5, 1\}$
D	Threshold-based thrD functions with $\alpha \in \{0.2, 0.3, 0.4\}$ and $c = 1$ Linear Downscaling linD with $\alpha \in \{0, 0.3\}$, $\tau_{min} \in \{0.6, 0.7, 0.8\}$, and $c \in \{0.3, 0.5, 1\}$ Exponential Downscaling expD with $\alpha \in \{0, 0.3\}$, $\tau_{min} \in \{0.6, 0.7, 0.8\}$, and $c \in \{0.3, 0.5, 1\}$

Table 6.7: Parameter search space for configuration grid search.

The colors of the data points represent different function classes for the parameter U . The tested threshold-based configurations result in a high quality of service. The main reason for this behavior is that threshold-based scaling creates a large tolerance range, so fewer scaling actions happen overall. This ensures stable behavior but tends to generate high costs. Linear and exponential scaling are more balanced and largely cover the Pareto front. Here, the tolerance range can be kept small with low probabilities for scaling actions close to the tolerance range, enabling cost-reduced supply if the workload remains stable.

The lower right figure shows the results for different mean values of W . While a low mean of 15 s leads to unstable outcomes, the results for higher mean values of W are almost identical. In our simulations, more than 150 instances are active at the same time, making the scaling frequency small for any tested W . Overall, we conclude that our approach can represent a variety of configurations. The influence of U , D , and W can be explained. Users must determine their system's trade-off parameters and choose an adequate configuration. We see huge optimization potential in future works, as all algorithm parameters can be arbitrary functions.

6.7 Experimental Evaluation

In this section, we analyze our approach in experiments within a cloud environment. We focus on validating the simulation and the model by comparing them with a real system. We also compare our approach with established baseline autoscalers using

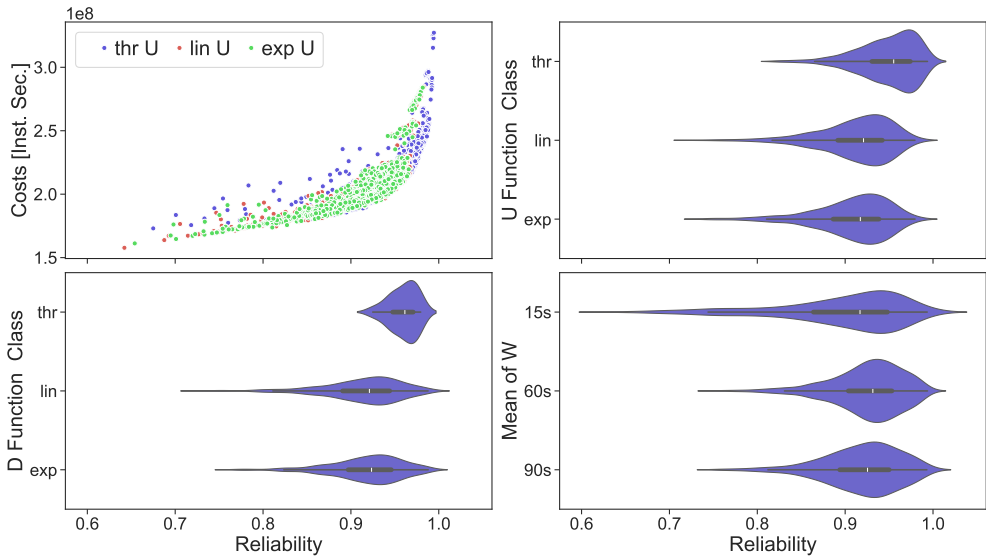


Figure 6.16: Configuration trade-offs and influence of the algorithm parameters.

realistic workloads and explore a use case with network-limited virtual machines. Our guiding questions are:

1. How can the approach be implemented with the low overhead for container-based cloud applications?
2. Are the model and the simulation suitable for approximating the scaling behavior of a real system?
3. How does our approach perform compared to established baseline autoscalers for different serverless test functions?

First, we describe the implementation of our approach for containerized applications and introduce all the components involved. Figure 6.17 shows an overview of our system. We add a layer to the container image of the service instances to perform the logic of decentralized autoscaling (DAS). The logic is encapsulated in a tiny C program that implements the algorithm introduced in Section 6.4 and reads the CPU utilization from the container’s file system. In Linux containers, this value is either available within the `cgroups` or `proc` directory. The gathering process of the monitoring data must be extended if more or different scaling metrics are to be used. The core parameters of the algorithm D , U , and W can be set either by implementing functions in the source code or via environment variables. The operator can set different configurations for different services.

This type of implementation offers the following advantages: (1) It minimizes the assumptions on the application container, (2) it can be easily integrated into existing

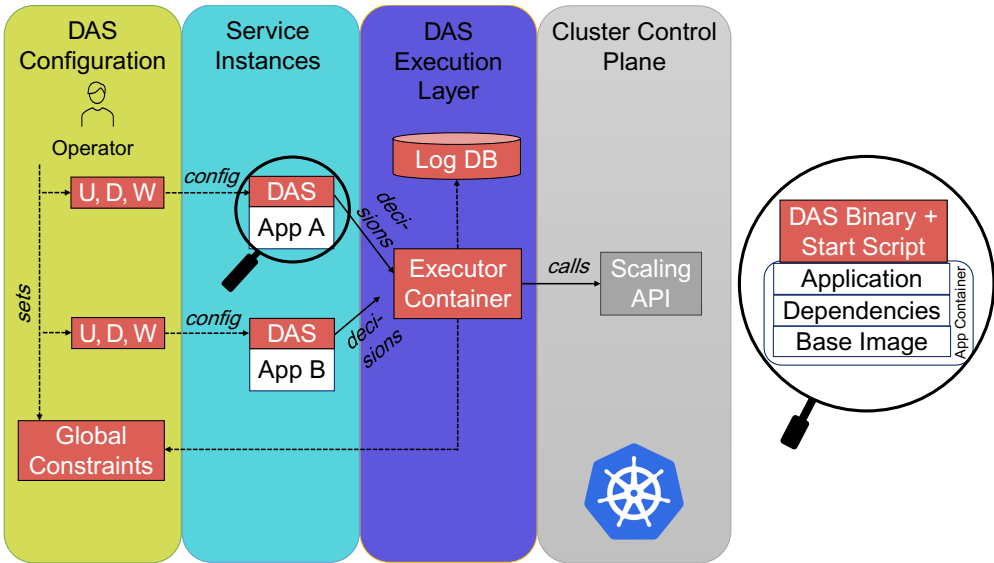


Figure 6.17: Implementation of decentralized autoscaling for containers.

build pipelines, and (3) it promises low overhead as no additional (sidecar) containers need to be started, and the autoscaling process idles most of the time. The latter statement depends on the choice of W , U , and D and assumes that one iteration of the algorithm has a low runtime and uses a negligible amount of resources. We could not detect any measurable runtime overhead for the test applications we used in this work (e.g., significant change in throughput, response time, or resource usage). By injecting the DAS binary, the image size of our test containers increased by about 1 MB to 20 MB. This value depends on whether dynamic linking is used and which libraries are present in the container. This overhead is feasible for most environments; optimizations for constrained environments are left for future work.

The service instances receive a URL as an environment variable to reach the execution layer, which then forwards the decision to the control plane. By using this abstraction layer, no orchestrator-specific logic must be integrated into the DAS binary. We implemented an execution layer for Kubernetes clusters. The execution layer can enforce global constraints defined by the user. In our case, we set the constraint that at least one instance of each service should always be deployed. In addition, we log every scaling decision and up- and downscaling probabilities and persist them in a database. The degree of logging may be customized and may not require a database. The communication between the instances and the execution layer is unidirectional, which means that the instances do not receive feedback on their decision. The executor containers are stateless, making them horizontally scalable at will. We emphasize that, except for constraint checking, the execution layer does not execute any other logic that might influence the scaling behavior. However, it might be worth exploring this option in future work, shifting from a purely decentralized to a hybrid approach.

The first step needed to run our autoscaler is the deployment of the execution layer. Operators might implement global constraints there. In the second step, our algorithm parameters D , U , and W have to be set in our C program that contains the scaling logic. The current version contains realizations for the waiting time distributions and scaling function stereotypes used in this work that take parameters via environment variables. Operators might implement the retrieval of further scaling metrics, custom scaling functions, and waiting time distributions in dedicated functions in the source code. In the last step, the source code needs to be compiled and packaged into a binary, which must then be added to the application container image (e.g., via adding a step in the build process). The container's start command needs to change so that the DAS binary is started as a background process first, and then the original entrypoint is executed. In contrast to conventional autoscalers, no additional setup or connection to monitoring tools is required, and the whole scaling behavior can be controlled by our three parameters D , U , and W .

If certain requirements are met, our approach can be implemented for other types of instances beyond containerized applications, such as unikernels or microVMs. First, an execution layer must be deployable, able to trigger scaling, and enforce user-defined global constraints. Second, the instances must be able to capture the scaling metrics and run the algorithm from Section 6.4 concurrently with their business logic (e.g., as a background process). Moreover, instances must be able to submit their decisions to the control plane.

In the following, we use our prototype to conduct an experiment comparing the results of our model, simulation, and an actual run of our approach in a cloud environment. The SHA256 hash function from Section 6.3 in the identical Google Cloud setup serves as the test application. Eight instances are deployed in the beginning. The workload is 20 rps for the first 900 seconds. Then, the load suddenly jumps to 80 rps, while after 2700 seconds, the rate drops back to the initial state. We test two DAS configurations. Config 1 is threshold-based with a tolerance range between 30% and 90% utilization. The waiting time is drawn from a uniform distribution between 30 and 90 seconds. Config 2 has a smaller tolerance range between 60% and 80%; the probabilities for up- and downscaling increase exponentially with increasing distance to the tolerance range. The waiting time distribution is uniform between 30 and 60 seconds. The measurements and simulations are repeated 30 times. After carrying out the prototype measurements, we used the results to estimate the average service rate, which is then used in the model and simulation. A complete list of parameters for this experiment can be found in the second column of Table 6.8. Similar to Table 6.4, cells with * indicate that different parameter values have been used in the experiment.

Figure 6.18 shows the deployed instances over time expressed as mean and standard deviations. Those values can be obtained directly from the prototype and the simulation. For the model, we extracted these values from the distributions resulting from Equations 6.15 and 6.28. We see that simulation and model can qualitatively duplicate the behavior of the real system. All three systems converge to a value within the tolerance range. Simulation and model can also predict the behavioral change due to a different configuration, especially during downscaling.

It is noticeable that both the simulation and model overestimate the system's reac-

Parameters	Experiment	
	Comparison of model, simulation, and prototype	Comparison of DAS, HPA, and KPA
Repetitions	30	5
	DAS Parameters	
Waiting time distribution W	*	Uniform(30s, 90s)
Upscaling function U	*	$\exp U_{0.8,0.95,1}$
Downscaling function D	*	$\exp D_{0.3,0.6,1}$
	HPA Parameters	
Sync Interval [s]	N/A	15 (HPA) 15 (HPA2)
Target Utilization [%]	N/A	75
Downscale Stabilization [s]	N/A	300 (HPA) 0 (HPA2)
	KPA Parameters	
Scaling Metric	N/A	rps
Target	N/A	9 (Matrix) 3 (Image)
	Other Parameters	
Request Timeout [s]	10	10
CPU Throttling [mCores]	100	100 (Matrix) 200 (Image)
Load Balancer	Google Cloud	Google Cloud

Table 6.8: Parameters of the experiments with our prototype.

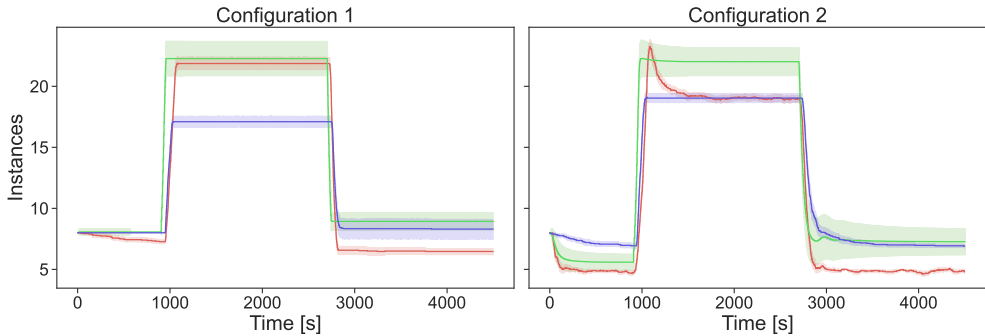


Figure 6.18: Comparison of real system (blue), simulation (red), and model (green).

tion to load increase. In Config 1, this results in an offset that is not corrected due to the large tolerance range. In the real system, about 17 instances are deployed on average, while model and simulation estimate 22. In Config 2, the simulation correctly predicts the system’s steady state of about 19 instances. Three factors can explain the deviations from the prototype. First, the analytical model does not consider request timeouts (here: 10 seconds). As shown in Section 6.6, these reduce the overshoot notably. Second, in the prototype, some instances report utilizations up to 103%. This happens because the containers’ CPU limits are not strictly enforced¹² and leads to a temporarily increased service rate, further reducing the overshoot. Last, we found that the phase in which a significant difference exists in the instances’ utilization is shorter than in the simulation. One reason for this could be a utilization-aware load balancer. In the simulation, the requests are always distributed randomly. An intelligent load balancer could also be a root cause for deviations in the downscaling case. In Config 2, the prototype and model converge to 7, while the simulation predicts 5 deployed instances. Another explanation is the inaccurate system state right before the load drops, leading to more downscaling decisions, especially in the simulation. These deviations are then not corrected as a steady state is hit.

A general uncertainty factor is the service time estimation. In the model and simulation, we assume an exponentially distributed service time. The rate of this distribution has been derived from the measured mean service time. Hence, it is only an approximation and contributes to an incorrect prediction of the steady state. However, only a few incorrectly predicted instance decisions are the root causes of the deviations. In the case of downscaling for Config 2, the simulation predicts two instances less than the prototype actually deploys. This means that only two instance decisions were wrongly predicted. Besides the systematic errors discussed above, this could also be a consequence of the randomly drawn inter-decision times. That is why the deviations are smaller than the visual impression conveys. The random errors (e.g., caused by inter-decision intervals) are reduced the more service instances are deployed. Overall,

¹²<https://kubernetes.io/docs/concepts/configuration/manage-resources-containers>

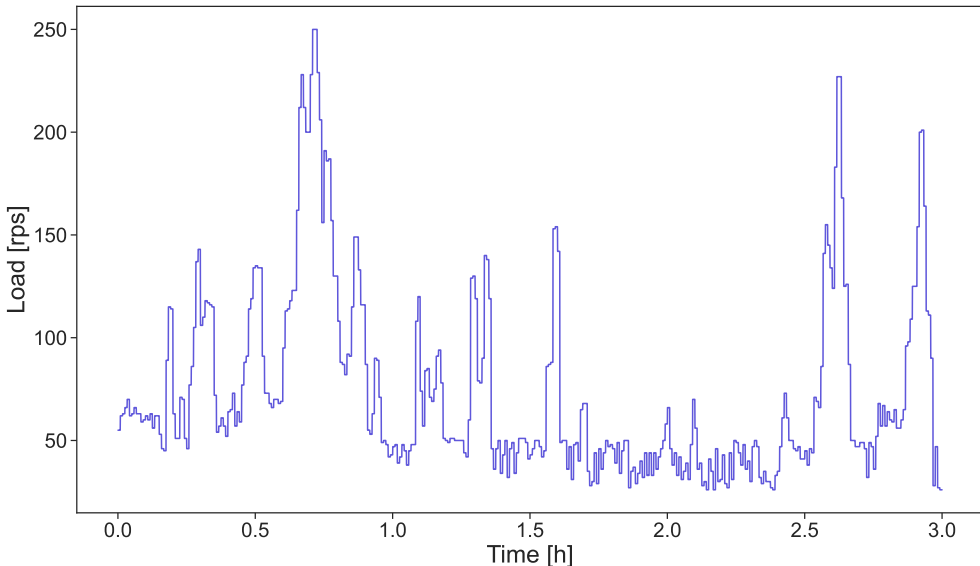


Figure 6.19: Test trace for comparing the decentralized, HPA, and KPA autoscalers.

we conclude that both the simulation and the model provide valuable approximations for the real system despite some systematic deviations.

In the following, we compare our approach with two established autoscalers and evaluate its performance for two different stateless containerized applications, in this case, serverless functions. In addition to the HPA and its refined configuration HPA2, introduced in Section 6.3, we also consider the Knative Pod Autoscaler (KPA), which is often used in the Function-as-a-Service context. We chose these three baselines as they are de-facto industry standards [QIP⁺24] and can, similar to our approach, be used out of the box. A comparison to other state-of-the-art autoscalers that require a more extensive setup (e.g., pretraining) is out of the scope of this work. The test workload shown in Figure 6.19 is the trace from our industry partner already used in Section 6.1.

We evaluate two test functions from a serverless test dataset [EBG⁺21]. The *Matrix* function (Python) inverts a random matrix of variable size while *Image* (Node.js) generates a random image and then resizes it. Matrix is much faster and more stable in its performance than Image. Consequently, different SLO values of 100 ms and 2 seconds have been chosen. All autoscalers aim to maintain a target utilization of 75%. The decentralized autoscaler uses a tolerance range between 70% and 80% with exponentially increasing probabilities for up- and downscaling outside this range. The HPAs run with a sync interval of 15 seconds. HPA uses the default configuration, while HPA2 allows faster downscaling. The KPA does not scale based on CPU utilization but either on the average concurrency (queue size) or the arrival rate. To achieve comparability, we tested the KPA in different configurations and chose an arrival rate-

based policy that keeps the utilization around 75%. A detailed list of all parameters can be found in the third column of Table 6.8. All parameters for the Horizontal Pod Autoscaler and Knative Pod Autoscaler that are not stated in this table have their default values.

Table 6.9 shows the QoS and cost metrics of all autoscalers in the average and standard deviation of five repetitions. The costs are normalized to the mean costs of HPA. The decentralized autoscaler causes the lowest costs for both test functions. For Matrix, it achieves this with minimally reduced QoS values compared to HPA. In the case of the more unstable Image function, the QoS metric decreases by about 10%, with a cost saving of about 60%. Our approach beats HPA2 in costs and QoS for both test functions. The KPA ranks third in costs but last in QoS for Matrix and Image. However, it should be noted that the KPA uses a different scaling metric and processes the requests via a queue proxy. Therefore, the comparison is not fair. Nevertheless, we decided to use it as a reference and to compare the achieved QoS at a cost level similar to the other scalers. In addition to the primary cost and SLO metrics, Table 6.9 also shows the total number of container starts and shutdowns. This metric helps to further understand the behavior of the decentralized autoscaler since the missing coordination between the instances could raise concerns over many up- and downscaling decisions that cancel each other out. As expected, the numbers for DAS are significantly higher than for HPA, which has longer scaling and stabilization periods. However, the numbers of DAS are smaller than those of HPA2 (Matrix and Image) and KPA (Matrix). This shows that the scaling behavior of DAS can be as stable as that of centralized solutions. As our parameter study in Section 6.6 shows, the algorithm parameters U , D , and W can effectively control the scaling frequency and stability.

In general, the experiments shown here are to be understood as a first step. Due to the extensive measurement effort, we only show selected configurations of all autoscalers here. Other configurations and other test applications may produce different results. A detailed experimental study, especially at a larger scale, could not be conducted due to the lack of resources and is therefore left for future work. However, we conclude that our decentralized approach has great potential, creating at least comparable results to state-of-the-art autoscalers without the need for centralized monitoring. In the case of rapid load changes, we adapt faster than the HPA due to the higher scaling frequency.

6.8 Beyond CPU-Bound Container Applications

In the previous sections, we showed that our approach can scale fine-grained serverless functions deployed in containers with the CPU as their bottleneck. While this was the primary motivation of this work, we introduced our approach on an abstract level and referred generically to *instances* as the entities to scale. In this section, we demonstrate the applicability of our approach in a scenario involving virtual machines, and the limiting resource is the egress network bandwidth.

Scaler	Matrix			
	RT < 100 ms [%]	Costs	Starts	Shutdowns
DAS	98.52 ± 0.08	0.381 ± 0.012	171.6 ± 5.7	166.2 ± 5.6
HPA	98.85 ± 0.05	1.000 ± 0.019	92.4 ± 4.8	68.2 ± 4.2
HPA2	98.47 ± 0.02	0.502 ± 0.014	177.2 ± 5.8	174.6 ± 5.7
KPA	94.36 ± 4.65	0.838 ± 0.000	233.0 ± 4.6	209.8 ± 4.0
	Image			
	RT < 2 s [%]	Costs	Starts	Shutdowns
DAS	80.72 ± 2.16	0.393 ± 0.048	125.8 ± 12.2	134.2 ± 12.6
HPA	89.23 ± 0.26	1.000 ± 0.025	65.4 ± 2.9	60.0 ± 2.3
HPA2	77.72 ± 0.87	0.442 ± 0.006	174.0 ± 9.9	182.4 ± 9.9
KPA	63.81 ± 2.24	0.807 ± 0.001	90.6 ± 1.5	81.6 ± 1.5

Table 6.9: Scaling metrics for different autoscalers.

Our scenario is inspired by a video streaming use case where a variable number of customers c request streams from relay servers. A minimum bitrate b_{min} is necessary to maintain the desired video quality. In our scenario, customers have enough bandwidth while a single relay server has a limited egress network bandwidth n_{max} . When more customers request streams from a single relay server, the fraction of n_{max} and c could fall below b_{min} , which means the customers cannot view their videos with the desired quality. New relay servers must be deployed (in a different network location) to serve the growing number of customers. The video streaming provider is interested in maintaining the quality of service and minimizing costs, meaning that when fewer customers are streaming, fewer relay servers should be deployed.

We implement this scenario using virtual machines in the Google Cloud. The relay servers are VMs of type `e2-highcpu-4`. The per-VM egress bandwidth in GCP is generally limited to 2 Gbps per vCPU.¹³ To create bottlenecks with a reasonable amount of workload (customers), we limit the egress bandwidth further to 600 Mbps using Linux’s `tc` command. The customers are deployed on an `e2-medium` VM with 30 Gbps ingress bandwidth. Another `e2-medium` VM acts as the control plane. The control plane takes care of deploying and removing relay servers. The control plane balances the load between relay servers using a least active connections strategy. We use `iperf3`¹⁴ to emulate video streaming, which establishes a constant bitrate UDP stream between the relay server and the client. We set the desired bitrate per stream to 15 Mbps, the recommendation for 4K streams on Netflix.¹⁵

¹³<https://cloud.google.com/compute/docs/network-bandwidth>

¹⁴<https://iperf.fr>

¹⁵<https://help.netflix.com/en/node/306>

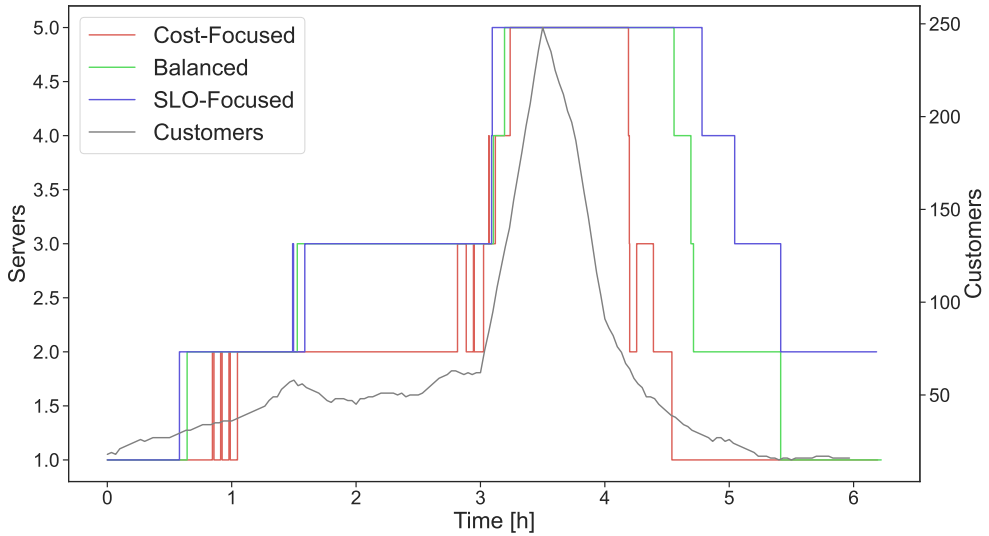


Figure 6.20: Deployed servers and workload in the video streaming experiment.

We deploy our decentralized autoscaler on every relay server. It tracks the number of sent bytes using Linux’s built-in network interface stats. It initiates upscaling when the sent rate in the last scaling interval goes close to the limit of 600 Mbps and downscaling when the used bandwidth is low. Scaling decisions are submitted to the control plane, which acts as the execution layer. In the case of downscaling, it checks that at least one relay server remains active. If so, the control plane removes the server with the least active connections from the load balancer. When all streams end on that server, it is shut down. When a new relay server is started, it is added to the load balancer once it passes a health check. Customers do not switch the relay server during a stream once assigned to one; there is no rebalancing when the number of servers changes.

Figure 6.20 shows our workload as the number of active customers over time. It represents a 24-hour cycle that starts and ends at 6 a.m. and has its peak at primetime in the evening hours [Com17]. Customers request videos of different lengths. In our scenario, the stream duration follows a normal distribution with a mean of 8 minutes and a standard deviation of 2 minutes. The deployment of new virtual machines took 15 seconds on average. These values differ significantly from the ones in earlier sections, where a client request was processed within milliseconds or a few seconds, and the readiness time of the containers was negligibly small. Hence, we must use significantly higher waiting times for the decentralized autoscaler; we choose a uniform distribution between 3 and 5 minutes here.

To reduce the experiment time and allow for multiple repetitions, we decided to speed up the described scenario by a factor of 4. This means that the actual experiment duration was six instead of 24 hours. The stream durations and waiting times are

Parameters	DAS Configuration		
	Cost-Optimized	Balanced	QoS-Optimized
Repetitions	5	5	5
DAS Parameters			
Waiting dist. W	Uniform(45s, 75s)	Uniform(45s, 75s)	Uniform(45s, 75s)
Upscaling fct. U	expU _{0.8,0.95,1}	expU _{0.7,0.95,1}	expU _{0.7,0.8,1}
Downscaling fct. D	expD _{0,0.6,1}	expD _{0,0.3,0.5}	expD _{0,0.2,0.3}
Other Parameters			
Load Balancer	Least Active Connections		

Table 6.10: Parameters of the video streaming experiment.

scaled accordingly. To gain control of VM readiness times, we do not actually delete the VMs when scaling down. Instead, we remove them from the load balancer and keep them running. During upscaling, the health check of the inactive VMs succeeds after the virtual readiness time. To determine an empirical distribution for this virtual readiness time, we measured 50 VM readiness times in GCP and divided their value by our scale factor 4. Adjusting the experiment duration was necessary to save costs. However, the results are still valid as the dynamic of the original setting is preserved, which is most important for autoscaler evaluation. In addition, we only consider normalized metrics in our evaluation. We test three DAS configurations and repeat every setting five times, yielding 90 hours of measurement data overall. The three DAS configurations (*cost-focused*, *balanced*, and *SLO-focused*) all use different exponential scaling functions U and D that represent different weights for the opposing goals of QoS maintenance and cost minimization. All parameters are reported in Table 6.10. The parameters for the expU and expD functions represent the fraction of the measured egress bandwidth divided by the per-server bandwidth limit of 600 Mbps.

Figure 6.21 shows the costs and SLO violations of the three tested DAS configurations. The values have been normalized to the costs and SLO violations of the *balanced* scaler. The costs are defined as the integral of the active relay servers over time. For the SLO violations, we consider the number of customers that experience a decreased bitrate and, therefore, potentially a video quality drop. We consider a bitrate decreased when it is less than 98% of the desired value of 15 Mbps; `iperf3` reports this metric once every second. Figure 6.21 confirms the expected costs and QoS tradeoff and, therefore, underlines the configurability of our approach. In addition, Figure 6.20 shows exemplary runs of every DAS configuration. All settings generally react adequately to changing workloads in this scenario, while the *cost-focused* setting scales up last and initiates downscaling first.

This scenario shows the applicability of our approach to a horizontal autoscaling scenario beyond CPU-bound container applications. We showed that the configuration

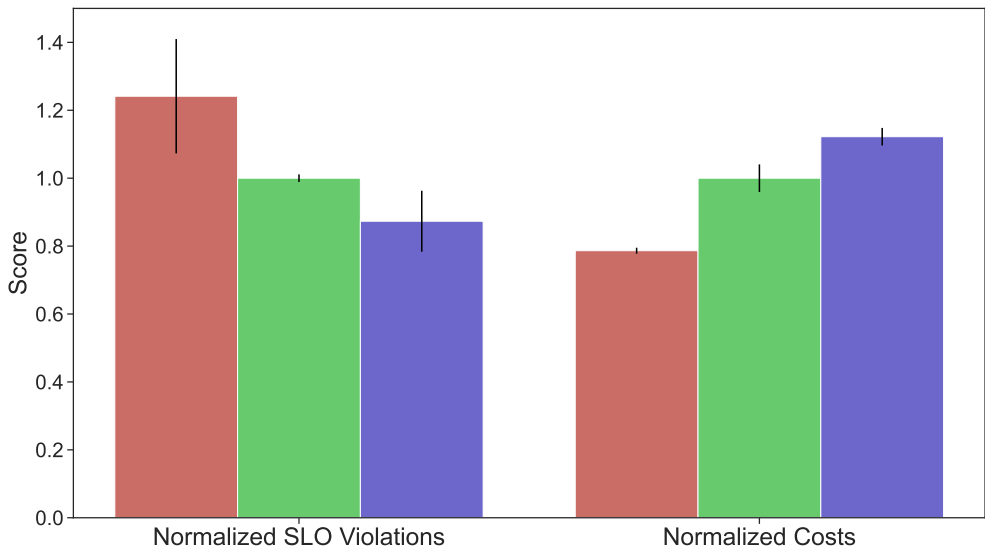


Figure 6.21: Quality of service and cost metrics in the video streaming experiment.

parameters remain powerful and explainable. The waiting time distribution W had to be adjusted to match the changed workload dynamics and instance readiness times. The scaling functions U and D retained their exponential shape and were fitted to the new scaling metric. Overall, we tested our approach in (1) a serverless function-as-a-service environment with different test applications, workloads, and baselines and (2) a use case inspired by video streaming based on virtual machines with different algorithm configurations. However, we can only cover some potential scenarios and cannot guarantee that our approach performs better than conventional autoscalers in all cases. A comprehensive discussion follows in the next section.

6.9 Summary and Discussion

Although autoscaling is an established area of research in the performance engineering and cloud computing community, many research autoscalers do not find their way into production deployments. In this chapter, we presented experimental insights into production-grade autoscaling and derived six core challenges for autoscalers in production systems. Those include handling structural and sudden workload changes, the balance between application-specific and -agnostic autoscaling, and the interdependence between monitoring and autoscaling. To address these challenges, we presented a new, decentralized, and continuous approach to autoscaling applicable to a wide range of modern cloud applications. In our approach, service instances make autoscaling decisions autonomously based on local monitoring data. By distributing the decisions over time, we significantly increase the scaling frequency compared to interval-based

approaches and achieve a quasi-continuous process when many instances are deployed. We conducted a model-, simulation-, and experiment-driven evaluation to prove the convergence, robustness, configurability, and performance of our approach in scenarios with different workloads, environments, and test applications.

A key finding is that although we do not coordinate between instances, our approach can achieve a scaling performance comparable to established autoscalers. This has been shown for different system sizes, test applications, and workload patterns and was evident from the queueing model, simulation, and prototype. Furthermore, our parameter study has shown that our autoscaling approach is configurable; that is, by changing parameters that adjust the instances' scaling behavior, the overall scaling behavior is significantly and explainable influenced (see Challenge 5 from Section 6.2). Also, our scaling functions and waiting time distribution offer a flexible yet lightweight configuration. We have shown that a large configuration space can be covered by adjusting three parameters, meaning that the configuration is manageable for practitioners (Challenge 4). In scenarios with highly dynamic loads, we outperform baselines with our approach's ability to react quickly to sudden workload changes (Challenge 1). This is enabled by high-frequency decisions made from a small action space. This high frequency also minimizes slowing effects during upscaling, created by constrained metrics like CPU utilization and their limited expressiveness in high-load situations.

In addition to pure performance, we differ conceptually from established autoscalers. In particular, there is no dependency on central monitoring systems; instead, arbitrary scaling policies can be defined (Challenge 3). A core characteristic of our approach is the flexibility an operator has in defining scaling functions without being bound to a given configuration space. However, we could also show that specific configurations, such as exponential up- and downscaling, yield good results for different test apps. This enables a good balance between application-specific and -agnostic solutions (Challenge 2). Our results indicate that our model and simulation can be tools to estimate the scaling behavior in different scenarios and configurations. Our prototype is deployable within the context of modern orchestration frameworks. As scaling decisions are made by service instances, the conditions of individual instances are considered by design. This means we could identify freshly started instances (e.g., by saving a timestamp at the instance start) and tune scaling functions to handle transient phases separately. Similar modifications could be made in the presence of application-level resilience patterns, like circuit breakers. As we focused on fast-starting, stateless serverless functions in our evaluation, such modifications were unnecessary but might be investigated for other use cases. Overall, we conclude that our approach is interoperable with modern cloud applications and orchestration frameworks (Challenge 6).

In Section 3.3, we analyzed state-of-the-art research autoscalers and summarized how they address the challenges for autoscalers in production systems. We came up with the fact that most autoscalers focus on either reactive or proactive scaling. Few also addressed hybrid scaling, which poses the challenge of weighing the recommendations of reactive and proactive strategies. Our approach of continuous decentralized autoscaling is, in the current implementation, reactive. However, due to the high scaling frequency, we overcome the limitations of many reactive and proactive autoscalers

that mostly act in predefined intervals. Both in the simulation and experiments, we were able to show that we can better match the actual resource demand, which leads to fewer SLO violations in the upscaling case and less resource wastage in the down-scaling case. This increase in scaling frequency also means we take many small scaling decisions instead of a few critical ones. This makes it possible to use comparatively simple, explainable scaling policies.

Many state-of-the-art autoscalers rely on various application-specific configuration parameters, which heavily influence their performance and scaling behavior. We argue that these parameters are hard to choose or must be tuned offline. This is especially difficult in DevOps contexts with frequently updated applications. As stated before, our approach has only three parameters that allow for powerful and flexible configuration. Operators can choose to apply custom, application-specific scaling functions or simple, general-purpose strategies similar to established autoscalers, like the HPA. Our scaling functions impact the scaling behavior intuitively and provide a compact configuration interface. This stands in contrast to other autoscalers that require extensive manual configuration through multiple parameters or AI-based approaches that cannot be configured intuitively. Moreover, they offer a significantly higher configuration space than traditional threshold-based approaches, so our approach can be very well customized for specific use cases. Our model and simulation offer great support for testing different configurations before deployment.

Through our decentralized decision-making, we eliminate the dependency on a specific centralized monitoring tool, which is a single point of failure, making us more robust against missing, delayed, and inaccurate monitoring data. Also, in our approach, there is no need to submit measurement values to a central instance, which could save network bandwidth in constrained environments and avoid delays in large, potentially globally distributed clusters. Our scaling decisions can be represented with a constant message size, essentially 1 bit (UP or DOWN), while sending HOLD decisions or logging information is optional. In Section 6.6, we showed that our approach works for different system scales, even without modifying its configuration. We refer to this as scale-invariance, meaning that as the system and load grow, the autoscaling potential also grows. This separates us from many related works that rely on training data or configurations linked to some specific system size. Another weakness of related works is that many approaches focus much on algorithmic details and are evaluated with synthetic workloads or in simulation environments only, making the applicability in practice questionable. This work covered evaluations based on a queueing model, discrete event simulation, and experiments in different environments and with different test applications. In summary, the abovementioned conceptual advantages could make our approach beneficial in various usage scenarios and provide a clear difference from the state of the art.

The large configuration space of our approach can also be intimidating at first. More experimentation is needed here, and adequate scaling policies must be found. This is a problem for any autoscaler that allows for more sophisticated configurations beyond simple thresholds. Other works use automated approaches like (reinforcement) learning to find suitable settings, lowering the manual experimentation effort. We have decided against this for now because of the issues of trust and explainability. In this

study and several preliminary experiments, we found some solid default strategies, but further exploration and optimization are out of the scope of this work.

Overall, a certain amount of expertise is required to use our approach; in addition to the parameter selection for up- and downscaling and the waiting behavior, the instrumentation of the test application currently also requires some manual work, although this can certainly be automated in the future. For smaller use cases with low load intensity and few deployed replicas, our approach might not have huge advantages over established autoscalers, as the monitoring overhead here is not large, and the deterministic behavior of traditional autoscalers achieves greater consistency. Our autoscaler, with its probabilistic nature, is especially advantageous for larger system sizes. As outlined in Section 6.7, the steps to run our autoscaler are manageable and are not dependent on the system size, just like the scaling policies. The efforts to set up our approach are slightly higher than for integrated solutions such as the Kubernetes HPA but lower than for many other research autoscalers that require manual or trained models as inputs. A core assumption of our approach is fast instance readiness times and generally high elasticity of cloud environments, enabling scaling decisions to be executed at high frequency. Different instance start times and workload dynamics might be addressed by adjusting the waiting time distribution as demonstrated in Section 6.8. We limited ourselves to horizontal scaling problems and assumed the usage of homogeneous instances (i.e., all replicas of one service have the same resource limitations). This is a common primary focus for autoscaling in the microservice and serverless domain [SGJN19], and we argue that most of our observations and results can also be transferred to vertical scaling.

Our measurement results from the cloud systems are subject to variability [LC16, LSL19]. We addressed this issue by repeating all measurements at least five times. In general, there are open questions regarding the generalizability of our (experimental evaluation) results. We chose representative test applications and environments used in prior works but could only evaluate some possible use cases of our approach. It is not guaranteed that our autoscaler outperforms established solutions in every use case. Our results cannot be generally transferred to other environments. The focus of this chapter is to introduce a novel autoscaling approach that differs conceptually from the state-of-the-art and to perform formal, simulative, and experimental analyses. Especially in the experimental analysis, there is more to explore as many factors play a role here (e.g., test workloads, test applications, baseline autoscalers, and parameter settings of our approach). We further discuss extensions and future work in Chapter 9.

We have limited ourselves to comparisons with the standard scalers of Kubernetes and Knative. These are very well suited as baselines because they are frequently used in industry and can be used universally and without training, just like our approach. A comparison with specialized solutions, such as reinforcement learning autoscalers, remains for future work. By design, we are limited to capturing metrics per instance only; we have yet to evaluate how services with external dependencies, for example, synchronous calls to databases, scale. Moreover, the role of the load balancer has yet to be fully explored. We can not make provable statements about its impact, as we used closed-source Google Cloud load balancers in our experiments. We decided to choose the default and maintained load balancer for our environment instead of a

custom solution to preserve the representativeness of our test environment. Lastly, our prototype evaluation was conducted in relatively small, homogeneous clusters as we lacked the resources to conduct large-scale experiments. However, Section 6.6 indicates that our approach performs well in larger systems.

All in all, this chapter presented a novel approach to autoscaling of modern cloud applications. Our approach follows a decentralized design paradigm and transforms autoscaling into a continuous process. It exploits the capability of modern cloud applications and environments to scale often and fast. We have shown that our approach is applicable to containerized applications and virtual machines and is capable of handling highly dynamic workloads. Thus, We have made an important contribution that addresses Challenge 3 of this thesis.

We answered RQ C1 (“Which conceptual weaknesses limit the applicability of conventional autoscalers to modern cloud applications and workloads?”) by conducting experiments with different autoscalers in a setup mirroring the setup of our industry partner. Based on our insights, we derived six challenges in production systems. These challenges were the basis of the design of our approach for continuous, decentralized autoscaling. The decentralized design and near overhead-less integration of the scaling logic into service instances allow running our approach at scale, thus answering RQ C2 (“How to design and implement an autoscaling approach for lightweight execution at scale?”). Our answer to RQ C3 (“How to enable adaptability to highly dynamic workloads?”) is the distribution of instance scaling decisions randomly over time. By reducing inter-decision times with every upscaling decision, we achieve fast reaction times to load changes and match time-changing resource demands accurately. Our approach requires three input parameters, and we have shown that cost-performance trade-offs can be realized efficiently using these. Our scaling functions can be kept simple but also allow for adaptability to specific characteristics of applications. Therefore, our approach and its configuration answer RQ C4 (“How to enable adaptability to different characteristics of cloud applications while keeping the configuration manageable?”). Overall, we conclude that our approach has great potential and motivates interesting future works.

Chapter 7

Enriching Microservice Simulation Through Authentic Container Orchestration

In the previous chapters, we have looked at CO frameworks and their performance metrics, as well as some selected aspects of container orchestration, such as container provisioning and autoscaling. As discussed in Chapter 4, end users are not only interested in the orchestration framework but also in the performance of the actual cloud application. Microservices are a modern architectural style for developing complex cloud software with a steadily increasing adoption in practice. Containers are the most popular deployment technology for microservices. As manual management of hundreds to thousands of containers is impractical, container orchestration frameworks are widely used, with Kubernetes being the most popular platform. As previous studies have shown, CO frameworks can significantly influence performance metrics of deployed applications, like response times or CPU usage [MTK22, PCB⁺19, TBVL⁺18, TVLLJ19, VBT⁺19]. This motivates users to find optimal CO framework configurations for their application, maximizing its performance or resilience. A full-blown configuration parameter study is often impossible in practice, as accurate measurements are costly and time-consuming. Simulations offer a good opportunity to test a large number of scenarios and configurations in a cost-efficient manner.

Many other researchers have studied microservice application simulation before, as Frank et al. [FWH⁺22] point out. To achieve our goal of evaluating different CO framework configurations and their impact on the application performance, microservice performance simulations have to consider the CO frameworks' behavior. Several challenges arise here. First, the orchestration mechanisms to be simulated are highly complex. For example, the Kubernetes scheduling algorithm has nine extension points that can be individually configured for specific use cases.¹ Second, there are dependencies between single orchestration mechanisms or application-level patterns. Figure 7.1 shows how Kubernetes' Horizontal Pod Autoscaler, `kube-scheduler`, and `cluster-autoscaler` work together when new service instances need to be deployed reacting to a performance degradation. In addition, we discussed dependencies between load balancing and health monitoring, as well as autoscaling and load balancing in the previous chapter. Third, CO frameworks are regularly updated, leading to a change in behavior and causing performance models to be outdated quickly. Previ-

¹<https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework>

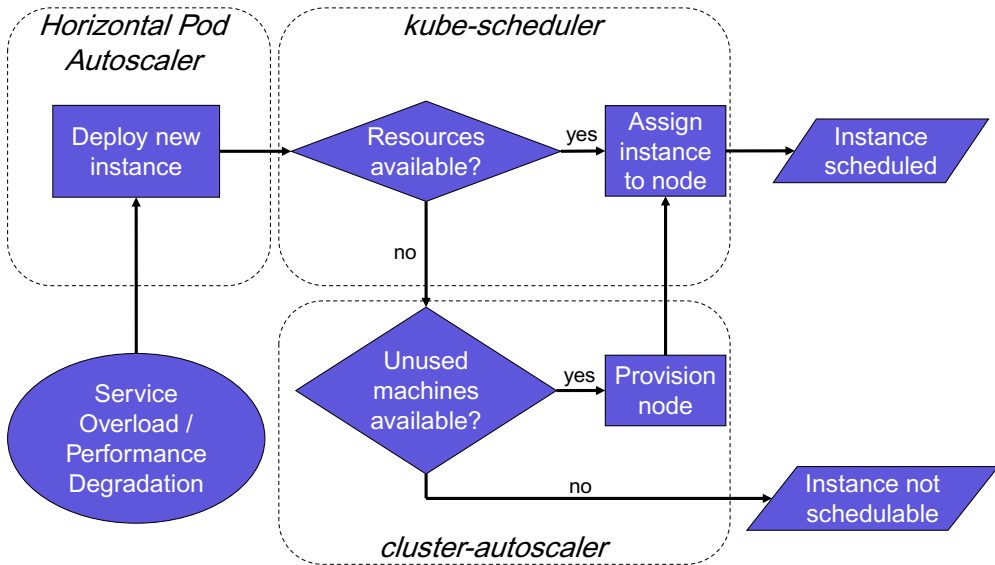


Figure 7.1: Interactions between Kubernetes components.

ous work in microservice performance modeling and simulation integrates either no or only self-implemented, simplified runtime orchestration mechanisms, as discussed in Section 3.4.

This contribution aims to establish a link between microservice performance simulation and modern CO frameworks. We present an approach that connects discrete event simulation and event-based systems. We use this approach to extend the microservice simulator MiSim [FWH⁺22] with Kubernetes orchestration mechanisms using their original code artifacts. By this, orchestration mechanisms can be integrated with simulation in their full complexity without requiring abstract models, making the simulation more authentic. Our implementation uses an adapter that translates events from the simulation to Kubernetes events and vice versa. Actions of the Kubernetes components directly modify simulated entities. We validate our approach in experiments with Kubernetes' `kube-scheduler` and `cluster-autoscaler`. We show that different complex scheduling and autoscaling configurations can be tested in the simulation with low overhead.

Microservice application designers can use our framework to evaluate realistic scenarios with impact factors present at runtime early in the development process. Furthermore, performance engineers could use the developed framework to study the performance of a microservice application in what-if scenarios in conjunction with different workloads and Kubernetes component configurations. Container orchestration researchers can use the framework to design new orchestration mechanisms and evaluate their behavior with different applications. Combining a microservice simulation with original Kubernetes components allows the use of the full complexity of the orchestration mechanisms in each of these use cases, leading to a more expressive and

realistic simulation.

This chapter addresses Challenge 4 from the overall context of this thesis. Our contribution provides a powerful connection between container orchestration frameworks and microservice performance simulation. Thereby, we do not require comprehensive behavioral models of container orchestration frameworks or parts of them and allow for the direct integration and evaluation of orchestrator configurations. To achieve Goal D of this thesis, we address the following guiding research questions in this chapter:

- RQ D1: How to integrate container orchestration frameworks into microservice performance simulations without requiring detailed models of their behavior?
- RQ D2: How to enable seamless transfer of orchestrator configurations into simulations?

The remainder of this chapter is structured as follows: In the next section, we introduce the microservice simulator MiSim, which serves as a foundation of this work. Sections 7.2, 7.3, and 7.4 provide details on our approach of enriching microservice simulation through authentic container orchestration and how we integrate Kubernetes orchestration mechanisms in the MiSim simulator. Sections 7.5 and 7.6 present our case studies with the `kube-scheduler` and `cluster-autoscaler` as examples of integrated Kubernetes components. Section 7.7 summarizes this chapter and discusses the advantages, disadvantages, and limitations of our approach. We published parts of this chapter, including text paragraphs, figures, and tables, as a full research paper at the 16th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS) [SHF⁺24].

7.1 The MiSim Microservice Simulator

This section introduces MiSim, a microservice simulator that serves as a foundation of this work and is the implementation basis for our approach. Frank et al. [FWH⁺22] propose MiSim as a microservice performance and resilience simulator. Technically, MiSim utilizes discrete-event simulation (DES) realized through the DES framework DESMO-J [LP99]. MiSim simulations require two essential inputs. First, a static description of the microservice architecture to be simulated is required. The architecture is a list of microservices; each microservice offers a set of operations (endpoints). For every operation, the computational demand and potential dependencies (other endpoints) must be specified. MiSim simulates only CPU resources. Four CPU scheduling policies are implemented. By default, MiSim uses multi-level feedback queues and the SARR algorithm proposed by Matarneh [Mat09] that schedules computation time based on the median of the remaining burst time of all scheduled processes.

The computational demand of a microservice operation is a number that quantifies the resource usage for one invocation of the operation. Each microservice is assigned a capacity that quantifies the maximum resource usage of a microservice within a simulation time unit (STU). For example, a microservice with a capacity of 100 could handle up to 20 invocations of an operation with a demand of 5 within 1 STU. The

utilization of a microservice is calculated as the fraction of actual resource usage and capacity. A queue forms in case the arrival rate exceeds the service rate. The operation demands and microservice capacity must be set correctly for an accurate simulation. Especially the performance characteristics of microservices within a common architecture must be set accurately relative to each other to identify bottlenecks correctly. MiSim further allows for the specification of resilience mechanisms per microservice, such as circuit breakers or retry patterns.

As a second input, MiSim requires an experiment model, which specifies the dynamic aspects of a simulation run. This work uses the experiment model to specify workloads, that is, user request arrival rates over time for different microservice operations. In addition, MiSim allows the definition of special timed events that influence the microservice simulation, such as unexpected service failures. MiSim outputs several performance metrics, such as microservice utilization and response times for every operation. We choose MiSim as an implementation basis for two reasons. First, MiSim is conceptually and technically developed to be extensible. In particular, it uses the strategy design pattern for many components, allowing us to substitute their behavior easily. Second, in contrast to many related works discussed in Section 3.4, MiSim is lightweight. The architecture and experiment models are simple JSON files; the source code is available and has few external dependencies. Frank et al. [FWH⁺22] showed that MiSim allows for qualitative and quantitative insights into microservice-based systems' complex behavior.

7.2 Foundational Container Orchestration Entities and Models

In the next sections, we explain our approach to enrich microservice simulation with authentic container orchestration. In all sections, we discuss our generic approach first and then illustrate the implementation with Kubernetes orchestration mechanisms and the microservice simulation MiSim. Our implementation² is to be seen as a wrapper that is 100% compatible with the MiSim core and builds on its extension points. This section explains the integration of some entities and basic CO models into MiSim. This is a necessary preparation for the more complex extensions presented in the upcoming sections.

The MiSim core does not have any deployment models; it only has models for microservice architectures and resilience patterns. CPU resources are considered independently from nodes. For this reason, we added some basic entities and associated events relevant at deployment time in the simulation. A *node* is modeled as a bunch of resources. A *cluster* is a graph with a set of nodes connected with edges indicating network latencies between individual nodes. A *container* has a 1:1 relationship to a *microservice instance* from the MiSim core and is deployed on exactly one node. Next, we need to define which aspects of container orchestration should be considered in our simulation. In Chapter 4, we identified eight performance-relevant tasks of

²<https://github.com/DescartesResearch/misim-orchestration>

container orchestration frameworks: container deployment, scheduling, resource allocation, availability, health monitoring, scaling, load balancing, and networking. We add support for all of these tasks in the simulation and implement simple orchestration mechanisms directly.

Performance overheads due to the deployment of containers are taken into account in the simulation by readiness times. Each container has a certain resource requirement, which has to be considered during scheduling. When the scheduler assigns a container to a node, a certain amount of resources are reserved and unavailable for other containers on the same node. The scheduler has to check if enough resources are available on the node. We implement two standard scheduling algorithms: random and round-robin. Health monitoring is represented in the simulation by periodic events that check the status of all deployed containers. Restarts can be simulated if a container has been crashed (e.g., as a result of MiSim’s chaos monkey function). Networking is determined by the cluster network graph indicating the mean latency and standard deviation between nodes. Furthermore, we have adapted one autoscaling and several load balancing strategies already available in MiSim to work with the new container and node models.

7.3 Connecting Discrete-Event Simulation and Event-Driven Systems

This section captures our core concept for connecting a discrete-event simulation and an event-driven system (EDS). While the first part introduces the theoretical concepts and notation, the second part outlines our implementation with MiSim (the DES) and Kubernetes (the EDS).

General Concept. Our goal in combining simulation and event-driven systems is to extend the authenticity and use cases of the DES by including mechanisms/algorithms used in the EDS. The EDS could be interpreted as an extension plugin that provides additional external functionality to the simulation. As discussed earlier in this work, a discrete-event simulation [LK07] is a sequence of events where each event $e \in E$ is associated with a time point t in the simulation time. Here, E denotes the set of all possible types of events. Each event e is associated with a function $F(e)$, which is executed as soon as e occurs. This function can, for example, change modeled entities in the simulation. An event-driven system [Mic06] is a popular architectural style for developing component-based software. Two components communicate with each other by the sender (producer) emitting an event e' and sending it via a proxy and middleware (channel) to the receiver (consumer). In the following, we denote the set of defined event types in the EDS as E' .

Consider a set of special event types $\Theta \subseteq E$ in the DES. We modify all functions $F(e)$ for all events $e \in \Theta$ such that these events are passed to the EDS. If such an event e occurs at simulation time t in the simulation, two more steps are needed to make the event processable by the EDS. First, we need a transformation τ that maps the event e to an event or a sequence of events $e' \in E'$, as usually not all events in the DES are represented in the EDS and vice versa. Second, passing a representation

of the simulation state $S(t)$ at time t may be necessary. This is especially necessary if not all events $e \in E$ are passed to the EDS. $S(t)$ should contain the states of all simulated entities relevant to the EDS. If the interface of the EDS allows only event-based communication, the simulation state $S(t)$ must be transformed into a sequence of events from E' . When sending e' and $S(t)$, the DES acts as an event producer from the event-driven system's point of view. The EDS forwards all received events to consuming components. The consumers now execute a black-box logic and generate responses $r \in E'$ as reactions to received events. Note that the DES does not have to model or make assumptions about the logic executed by the consumers in the EDS. The responses are passed back to the DES and processed by $F(e)$. As a consequence, entities or states in the simulation might be modified.

Application to MiSim and Kubernetes. In our use case, we connect the discrete-event simulation MiSim with some event-based orchestration mechanisms of Kubernetes. Figure 7.2 shows a part of our developed framework. To bridge the gap between MiSim and Kubernetes, we decided to use an adapter.³ This has the advantage that the simulation remains slim, and if no Kubernetes components are to be used, no unnecessary code must be loaded and executed. We select a set of events Θ from MiSim, that are relevant for Kubernetes components. Θ includes all events affecting containers and nodes, while several other MiSim events (e.g., related to CPUs or logging) are excluded. The simulation sends selected events and the state of the simulated entities to the adapter. The adapter is responsible for the event transformation τ and the processing and transformation of the components' responses r . It implements relevant parts of the Kubernetes API, especially endpoints for pods and nodes. Hence, it acts like the `kube-apiserver` from the Kubernetes components' point of view. Because the `kube-apiserver` handles all communication in the Kubernetes control plane, we only need this one adapter for different components. The adapter and Kubernetes components are started prior to the simulation. We initialize the component caches with empty lists for all resource types. When the simulation is started, events for modeled resource types (like pods or nodes) are forwarded to the Kubernetes components.

As an example, we consider a simulation scenario where one microservice instance should be deployed at the start of the experiment. In this case, the simulation schedules a `StartPodEvent` at time $t = 0$. This event type is linked to an event function that forwards information (e.g., pod name and resource requirements) to the adapter. The adapter transforms this information into a Kubernetes watch event of type `ADD` for the resource type `Pod`. Every consumer connected to the adapter and its pod event stream will receive this event. This event will then trigger some logic in the consumers and potentially a response. In our example, an instance of the `kube-scheduler` assigns the newly created pod to a node. In this case, the event function waits for a response from the adapter and modifies the simulation-internal representations of the pod and the assigned node afterward. We further discuss examples involving the `kube-scheduler` in the next section.

³<https://github.com/DescartesResearch/misim-k8s-adapter>

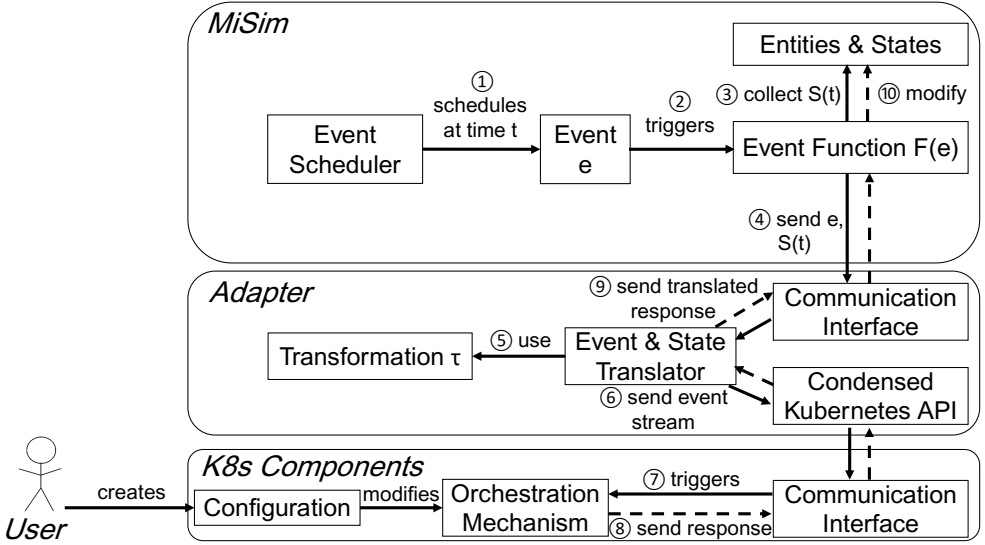


Figure 7.2: Overview of our approach and component interactions.

7.4 Dealing with Incompatible Events and Data

In this section, we expand our concept by handling incompatible events and data. This is motivated by the fact that usually not all events in the DES have equivalent events in the EDS and vice versa. Similar to the previous section, we start by introducing our concept on an abstract level before describing our implementation.

General Concept. In the following, we consider cases where the DES and the EDS use different models or entities that cannot be transformed into each other. In the following, let M_D be the set of models and entities that appear in the DES but have no equivalent in the EDS. Similarly, let M_E be the set of models and entities that exist in EDS but have no equivalent in the DES. Let M_{D+E} be the set of models and entities with equivalences in DES and EDS. First, we consider elements from M_D . Since, in our case, we are only interested in the results of the DES, these do not pose a problem. We simply exclude all events concerning entities M_D from the set Θ . We just have to ensure that we correctly model interactions between entities from M_D and entities from M_{D+E} .

We can divide elements from M_E into two groups. We ignore the group of models or entities from M_E that have no interaction with elements from M_{D+E} or whose interactions should not be considered in the DES. The second group of elements from M_E that have relevant interactions with elements from M_{D+E} has to be considered in our approach. These can, for example, influence the response r to an event e from the DES and thus influence the simulation run. To solve this problem, we propose to use a black-box information repository (BIR). The DES receives a set of additional EDS-specific information B before starting a run, as well as information about which

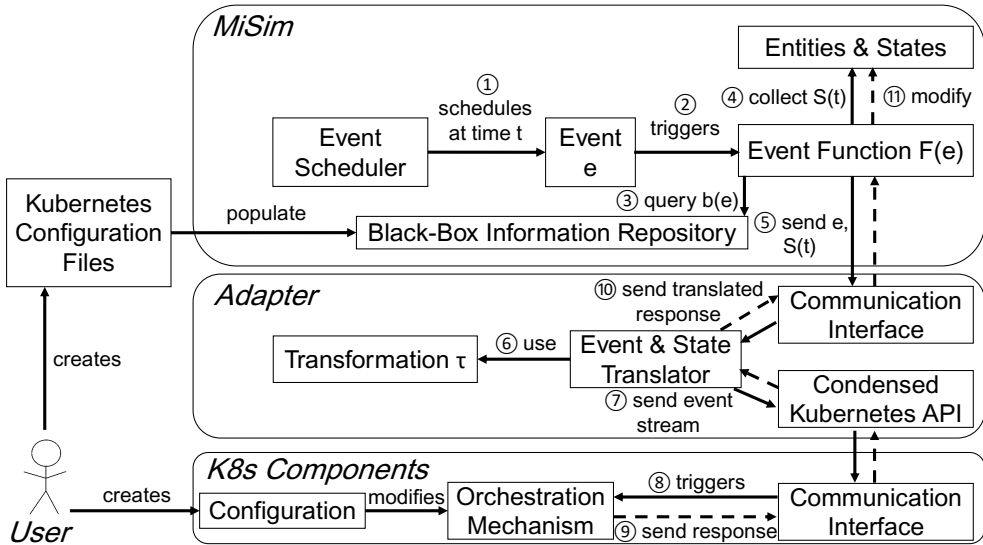


Figure 7.3: Extended framework version supporting incompatible events and data.

events $e \in \Theta$ should use which information $b \in B$. The simulation does not interpret or change elements in B but stores them in the repository. The function $F(e)$ passes them to the EDS for specific events e together with the simulation state $S(t)$. This way, entities not modeled in the DES can still be considered in simulation runs and in the black-box logic of the EDS components.

Application to MiSim and Kubernetes. In this work, we choose scheduling and cluster autoscaling as two Kubernetes orchestration mechanisms to integrate into MiSim. The scheduling algorithm comprises nine extension points where user-defined plugins can be attached.¹ As these plugins contain arbitrary logic, it is impossible to integrate this configuration space in the simulation using traditional models. Similarly, the `cluster-autoscaler` has more than 30 configuration options.⁴ Simulative evaluation is especially beneficial since cluster autoscaling can only be performed with special infrastructure where nodes can be added or deleted on demand. Both mechanisms use entities that are not present in MiSim (like node affinities, labels, or machine set definitions). Kubernetes uses YAML files to define these entities. Our framework stores these user-created files in the BIR and forwards the contents on specific events. In the following, we look deeper into how the `kube-scheduler` and `cluster-autoscaler` are integrated into MiSim using the aforementioned concepts. Figure 7.3 shows the extended version of our framework, including the black-box information repository.

The `kube-scheduler` receives a request from the adapter whenever an event in MiSim is triggered that creates a new container. This request contains information about the container to be deployed (e.g., its resource requirements) and all other

⁴<https://gist.github.com/neerfri/4bd7477920cb33a2a229807ed10c29c2>

7.5 Case Study I: Kubernetes Scheduling Policies in a Global Cluster

currently deployed containers (the simulation state). The selected node or an error (e.g., if all resources in the cluster are reserved) is expected as a response. The adapter converts this request into a series of Kubernetes watch events and sends them to the scheduler. The `kube-scheduler` determines nodes with enough resources and selects a node for the container according to its scheduling policy. This policy is set as a configuration¹ at the start, like in a real cluster. If no node is available, the `kube-scheduler` returns an error to the adapter, which passes it on to the simulation.

Scheduling policies in Kubernetes can also be influenced by other factors (e.g., affinities⁵). These are examples of critical elements from the set M_E , that is, entities that exist in Kubernetes but do not exist in the simulation and yet affect relevant tasks in the simulation. For the example mentioned above, we create node definitions with labels and pod definitions with affinities in the form of Kubernetes YAML files that populate the BIR at the simulation start. The simulation and adapter forward them on events concerning nodes or pods, respectively. The `kube-scheduler` uses this information for scheduling. With this approach, we can cover the scheduling policies' maximum complexity without implementing new logic ourselves. We demonstrate the usage of different scheduling policies in Section 7.5.

As a second Kubernetes component, we integrate the `cluster-autoscaler` (CA) into the simulation. It becomes active whenever the `kube-scheduler` cannot deploy a pod (upscaling, see Figure 7.1) or when the utilization of a node falls below a certain threshold (downscaling). Hence, the CA subscribes events for pods and nodes; our adapter forwards all decisions of the `kube-scheduler` directly to the CA. Furthermore, the CA needs endpoints to manage node groups. Here, different implementations for cloud providers exist.⁶ We use the generic open-source Kubernetes Cluster API.⁷ Note that the chosen cloud provider implementation does not influence the autoscaling logic. By now, we support the integration of `MachineSets` from the Cluster API. A `MachineSet` is a set of machines with equal resources that can be scaled from a specified minimum value (≥ 0) to a maximum value. The user can specify definitions of `MachineSets` as part of the BIR at the simulation start. We compare two different upscaling policies for the CA in Section 7.6.

7.5 Case Study I: Kubernetes Scheduling Policies in a Global Cluster

In the following, we provide empirical evidence on the usefulness of our approach for evaluating orchestration policies and validate the behavior of the included Kubernetes components by conducting two experiments. In this section, we consider a microservice application whose services are deployed in a global cluster. We examine the interactions between different scheduling policies, the application, and the cluster architecture. In the next section, we deploy an increasing number of service instances in

⁵<https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity>

⁶<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/README.md>

⁷<https://cluster-api.sigs.k8s.io>

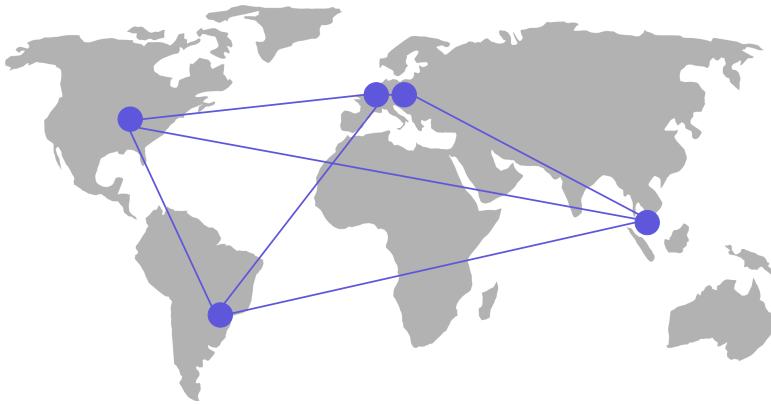


Figure 7.4: Modeled cluster environment and node locations.

a heterogeneous cluster with two machine types. We look at different expansion policies of the `cluster-autoscaler` and show that our framework can correctly capture the interactions between the CA and the `kube-scheduler`. We provide a CodeOcean capsule⁸ where all experiments can be reproduced.

In this experiment, we model a cluster with nodes in different geographic regions. A microservice reference application is deployed in this cluster. We simulate a constant load and analyze the response time of different user operations. Three different scheduling policies are tested, causing services to be deployed on different nodes and experiencing different network latencies. As surveyed by Carrion [Car22], optimization of Kubernetes scheduling policies is an active field of research. Scheduling policies can significantly impact service response times, resource utilization, quality of service, and deployment costs. We show that our framework can use the `kube-scheduler` in a way that it behaves the same as in the real cluster and that the simulation can reflect the effects of different scheduling policies on the simulated test application. Hence, we show that our approach reduces the overhead for operators searching for an improved scheduling policy for their applications.

We use a cluster with five workers and one master in the Google Cloud, as shown in Figure 7.4. All machines are of type `e2-standard-4` with four CPU cores. Two workers (`eu1` and `eu2`) and the master are deployed in Germany. The remaining workers run in Singapore, Brazil, and Iowa (USA). Every node has its compute zone as a Kubernetes node label. Before the experiment, we measure the network latency between all nodes using 100 ping (ICMP) messages. The network latencies are given to the simulation as means and standard deviations. We use Kubernetes YAML files to specify the five workers for the simulation. At the simulation start, this information is forwarded to the `kube-scheduler`, which extracts available resource capacities, node labels, and more. The control plane node is not considered in the simulation.

⁸<https://doi.org/10.24433/CO.4913288.v1>

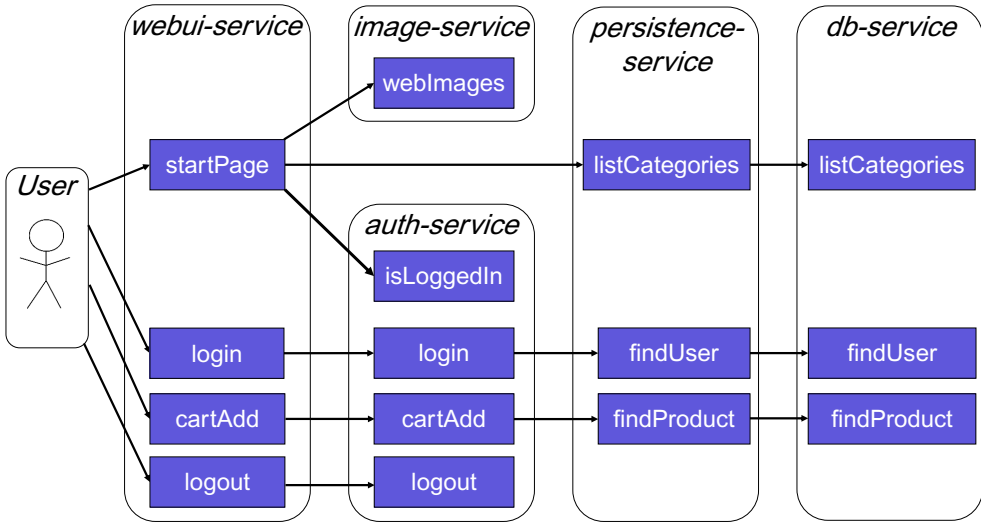


Figure 7.5: Dependency graph of the test application workload.

We use the popular benchmarking application TeaStore [vKES⁺18] with seven microservices in our experiments. All microservices request 0.8 CPU cores per instance. In the real cluster, we use the HTTP Load Generator⁹ to generate a constant load of 20 requests per second for five minutes. A user behavior consisting of four steps is simulated. First, a user accesses the start page and then logs in with his account data. Afterward, the user adds a random product from the store to his cart and then logs out again. In the simulation, we build on the architecture model of the TeaStore used by Frank et al. [FWH⁺22] and extend it with the database service. The defined workload stimulates a total of five services and 15 endpoints. The dependency graph of the operations for our workload is shown in Figure 7.5.

We deploy one instance of each TeaStore service in our cluster. Hence, the total CPU core requirement is 5.6 cores, meaning that at least two nodes are needed for the deployment. We evaluate three scheduling policies: *default*, *most-allocated*, and *europe-only*. The *default* scheduling policy deploys the pod to the node where the least resources have been reserved so far. This results in the services being distributed to all nodes. Kubernetes offers two general options to influence scheduling. First, different configurations of the `kube-scheduler` can be used, resulting in a different algorithm being used for all pods. The *most-allocated* policy is an example of this. We use a profile¹ of the `kube-scheduler`, which causes the next pod to be deployed to the node with the most resources reserved but still enough resources to run the pod. This policy requires two nodes to run the services. The scheduling policy is set at the start of the `kube-scheduler`.

⁹<https://github.com/joakimkistowski/HTTP-Load-Generator>

The second option to influence the scheduling in Kubernetes is to set special pod properties. These will influence the scheduling only for selected pods. For our *europa-only* policy, we use node affinities. Precisely, we specify that our services can only be deployed on nodes with a label indicating that they belong to the compute zone **Europe**. This causes the services to be distributed to the nodes **eu1** and **eu2**. We use the same Kubernetes deployment files to specify the affinities both in the simulation and the real cluster. This means that no changes are necessary, and the effort to set up the simulation is low.

Before deploying the services, all nodes have no reserved resources. Therefore, the nodes are equally suitable for selection. Furthermore, the order in which the containers are to be deployed plays a role in the scheduling since the `kube-scheduler` processes the pods sequentially (i.e., one after the other). In initial tests, both factors led to different placements occurring during repeated simulation and real cluster runs. We made two adjustments to ensure the experiment’s comparability and repeatability. First, we fixed the order in which the containers should be deployed to the recommended deployment order of the TeaStore.¹⁰ Furthermore, we deployed other containers on four of the five worker nodes that occupy different resources at the nodes: **Singapore** (0.05 cores), **Brazil** (0.1 cores), **eu2** (0.15 cores), and **USA** (0.2 cores). This setup leads to the scheduling decisions not being random and the experiment being repeatable. In the simulation, these helper containers are part of the black-box information repository. They are loaded at the start of the simulation run and forwarded to the `kube-scheduler`. The simulation, meanwhile, does not hold any model of these containers. This example shows our approach’s flexibility and its support for information mismatches between the simulation and integrated Kubernetes components.

Table 7.1 gives an overview of the scheduling decisions for different policies. These scheduling decisions are the same in the real cluster and the simulation. This shows that our approach can correctly represent different scheduling policies in the simulation without implementing them. The second question to be answered is whether the effects of different scheduling policies on the performance of the modeled microservice application can be simulated. Figure 7.6 shows the simulated and measured response times with their standard deviation for all four considered request types. We see that the scheduling policies significantly impact the response times. The *default* policy that distributes the services worldwide has the highest response times, while the *europa-only* policy has the lowest. This is visible in both the measured and simulated results.

Furthermore, using the `logout` operation as an example, we see an interaction between application architecture and scheduling. As shown in Figure 7.5, the `logout` operation depends only on one operation of the `auth` service. According to Table 7.1, the *default* policy causes `webui` and `auth` to be deployed in Europe. In contrast, with the *most-allocated* policy, the `auth` service is deployed in the USA, and the `webui` service is in Europe. This explains the higher response time for this operation in the *most-allocated* policy compared to the *default* policy. Figure 7.6 also shows the predicted response times of the original MiSim version (labeled as *baseline*) without

¹⁰<https://github.com/DescartesResearch/TeaStore>

7.5 Case Study I: Kubernetes Scheduling Policies in a Global Cluster

Service	Scheduling Policies		
	default	most-allocated	europa-only
auth	eu2	usa	eu2
db	eu1	usa	eu1
image	usa	eu2	eu1
persistence	brazil	usa	eu1
recommender	eu1	eu2	eu2
registry	singapore	usa	eu2
webui	eu2	eu2	eu1

Table 7.1: Nodes for services selected by different scheduling policies.

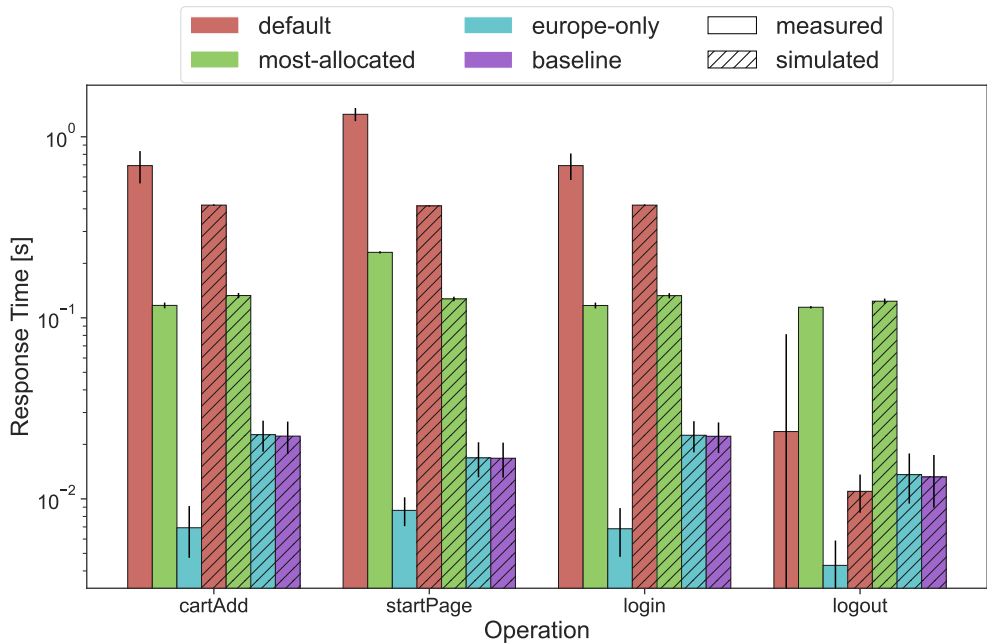


Figure 7.6: Measured and simulated response times for different scheduling policies.

an integrated scheduler. As mentioned earlier, the original version of MiSim considers resources independently of nodes, and therefore, inter-node latencies cannot be modeled. Consequently, the predicted response times are slightly smaller than in the *europa-only* policy, which has the smallest network latencies. These results show that our approach not only allows for the integration of authentic container orchestration policies but also can increase the accuracy of the simulation.

Overall, the effects of different scheduling policies on microservice performance can be qualitatively modeled. However, some operations also have significant errors in the response time prediction. These are caused by inaccuracies in the MiSim performance model. For example, the large deviation for the `startPage` operation in the *default* policy is caused by large payloads with variable sizes (here: images) sent over the network. MiSim lacks support for modeling different payload sizes and parametric dependencies. Hence, the real network overhead cannot be predicted accurately in this case. However, this is a limitation of MiSim’s performance model rather than a weakness of our approach for including orchestration mechanisms, as further discussed in Section 7.7. We also analyzed the overhead of our adapter and the integration in total. In this experiment, we measure only a tiny overhead. The simulation runs take, on average, about 9.2 seconds, of which only about ten milliseconds are spent on the communication with the adapter and `kube-scheduler`. This shows that our approach has a reasonable overhead and can enrich MiSim with authentic Kubernetes container orchestration mechanisms.

7.6 Case Study II: Evaluating Expansion Policies for Cluster Autoscaling

In this section, we demonstrate our simulation with the `cluster-autoscaler` and show its ability to capture the interactions between the CA and the `kube-scheduler`. Therefore, we dive deeper into the simulation’s configuration and detail the interconnections between the components outlined in Figure 7.1. We model a heterogeneous cluster with two different node groups. The CA offers the possibility to define *expansion policies*, that is, options to configure which node groups should be prioritized during upscaling. We compare two expansion policies and prove they work as expected in our framework.

We define a cluster with two machine sets. Machines in the `small-set` have 4 CPU cores and in the `large-set` 8 cores. Both machine sets can be scaled from one to ten nodes. One node from each machine set is deployed at the start. The analogy in a real cluster would be to deploy different virtual machine instance groups with different resources and pricing models. Unfortunately, we cannot directly compare this experiment to a Google Kubernetes Engine cluster where a proprietary, not freely configurable version of the `cluster-autoscaler` is used.

The expansion policies of the `cluster-autoscaler` can be adjusted via a command line flag. The CA triggers upscaling whenever the `kube-scheduler` cannot assign a pod to a node due to a lack of resources. We consider two expansion policies. The *random* policy randomly selects a node group for upscaling. In contrast, the *least-*

7.6 Case Study II: Evaluating Expansion Policies for Cluster Autoscaling

waste policy prefers the node group, where as few resources as possible (here: CPU cores) are wasted after the deployment of the new pods. In this case, the machine set with four cores is always preferred. We use a Kubernetes deployment file of the TeaStore registry service as a test application, with each replica requesting 0.5 cores. For demonstration purposes, we use an autoscaler that requests a new instance every 15 seconds.

```
{
  "microservices": [
    {
      "name": "teastore-registry",
      "instances": 1,
      "capacity": 1000000,
      "operations": [
        {
          "name": "register",
          "demand": 30,
          "dependencies": []
        },
        ...
      ]
    }
  ]
}
```

Listing 7.1: Architecture model for cluster autoscaling experiment.

In the following, we describe in detail how this scenario is realized in the simulation. First, we define the two MiSim core models. The first is the architecture model describing the microservices, their operations, performance characteristics, and dependencies. The architecture model is relatively simple in this case study, as only the TeaStore registry service is considered. Listing 7.1 shows the model used here. The second model introduced in MiSim is the experiment model that contains settings about generated load, the experiment duration (here: 3601 STU), and logging. Also, it allows the specification of a random seed to enable the reproducibility of the experiment run. We specify the generated load using CSV files similar to the output of the LIMBO tool [vKHK14]. Essentially, the number of requests sent to a microservice operation for every STU is specified there. Listing 7.2 shows an excerpt of the experiment model used for the simulations in this section.

Once the essential models for the MiSim core have been defined, the orchestration mechanisms can be configured. We start by filling the black-box information repository with Kubernetes YAML files that represent objects present in the cluster at the start of the experiment. Users of our approach could query these settings directly from their cluster environment. Alternatively, freely modified or re-designed files can also be used for the simulation. In this case, we provide a `Deployment` file for the TeaStore registry service, two `MachineSet`, two `Machine`, and two `Node` objects. The `Deployment` object specifies the scheduler responsible for deploying instances of the service and specifies the resource requests.

```

{
  "simulation_meta_data": {
    "experiment_name": "CAExperiment",
    "model_name": "architecture_model",
    "duration": 3601,
    "seed": 979,
    ...
  },
  "request_generators": [
    {
      "type": "limbo",
      "config": {
        "operation": "teastore-registry.register",
        "limbo_model": "./load.csv",
        ...
      }
    }
  ]
}

```

Listing 7.2: Experiment model for cluster autoscaling experiment.

Listing 7.3 shows an excerpt of the `Deployment` file used in this case study. The `MachineSet` objects specify a template for virtual machines that can be included in the cluster. Essential properties of the machines, such as their CPU cores and memory, as well as scaling properties of the `MachineSet`, are specified as annotations. In our case, both machine sets can scale between 1 and 10 machines. The `Machine` objects are instances from the `MachineSet`. They hold references to the `MachineSet` they were created from and to the `Node` they represent in the Kubernetes cluster. If a `Machine` had no reference to a `Node`, then the `Machine` would not be integrated into the cluster.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: teastore-registry
spec:
  replicas: 1
  template:
    spec:
      schedulerName: custom-scheduler
      containers:
        - name: teastore-registry
          image:
            descartesresearch/teastore-registry
          resources:
            requests:
              cpu: "0.5"
          ...

```

Listing 7.3: Deployment file for the TeaStore registry service.

7.6 Case Study II: Evaluating Expansion Policies for Cluster Autoscaling

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: MachineSet
metadata:
  name: small-set
  annotations:
    cluster.x-k8s.io/cluster-api-autoscaler-node-group-min-size: "1"
    cluster.x-k8s.io/cluster-api-autoscaler-node-group-max-size: "10"
    capacity.cluster-autoscaler.kubernetes.io/memory: "128G"
    capacity.cluster-autoscaler.kubernetes.io/cpu: "4"
    capacity.cluster-autoscaler.kubernetes.io/maxPods: "120"
  ...
```

Listing 7.4: MachineSet specification.

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Machine
metadata:
  name: small-machine
  ownerReferences:
    - apiVersion: cluster.x-k8s.io/v1beta1
      kind: MachineSet
      name: small-set
  ...
status:
  phase: Running
  nodeRef:
    apiVersion: v1
    kind: Node
    name: small-node
  ...
```

Listing 7.5: Machine specification.

```
apiVersion: v1
kind: Node
metadata:
  name: small-node
status:
  phase: Running
  capacity:
    cpu: "4"
    memory: "128G"
    pods: 120
  ...
```

Listing 7.6: Node specification.

Listings 7.4, 7.5, and 7.6 show excerpts of the Kubernetes YAML files for the machine set `small-set`, one of its machines, and the associated node. These files populate

the black-box information repository for this simulation run. Note that the simulation itself has no implementations of the concepts described above. The simulation reads these files and forwards the objects to the adapter without further processing. The adapter is able to parse this information using Kubernetes' Go libraries. The adapter creates an addition event for each object in the black-box information repository. It sends this event to all connected Kubernetes orchestration mechanisms that listen to the specific object types. In our case study, the `kube-scheduler` and `cluster-autoscaler` are active. The `kube-scheduler` listens to events affecting `Node` and `Deployment` objects. The `cluster-autoscaler` additionally subscribes to events affecting `Machine` and `MachineSet` objects.

The last preparation step for the simulation is to configure the Kubernetes components and their orchestration mechanisms. The steps needed for each individual component differ. In our case, the `kube-scheduler` is configured using a YAML file containing an object of type `KubeSchedulerConfiguration`. This object specifies the weights of implemented scheduling strategies. In this section, we use the *default* scheduling policy introduced in the previous case study. In contrast, the `cluster-autoscaler` uses command-line arguments for configuration. We use the `cloud-provider` flag to specify that we use the Cluster API to provision machines. In addition, the `expander` flag is used with the values *random* and *least-waste* to specify which machine types are preferred during upscaling.

Figure 7.7 shows the two expansion policies in comparison. Since the *random* policy returns different results with repeated runs, Figure 7.7 shows only a selected test run. Both expansion policies behave as expected: The *random* expander deploys alternating instances with 4 and 8 CPU cores on average (both red lines increase early). The *least-waste* expander first scales the machine set with 4 CPU cores up to the maximum number of 10 (only the solid red line rises first). This leads to a slow increase in the number of provided CPU cores (blue line) and many upscaling decisions being necessary initially. The *random* policy makes capacity increase more stable, making it a policy that can be used when nothing is known about workload changes. The *least-waste* policy is a cost-optimized policy that can be used when slowly increasing loads and few scale-ups are expected. Both policies stop the expansion when the maximum size of the machine sets is reached. Overall, we show that different `cluster-autoscaler` configurations can be correctly executed in our simulation. The basis for this is the interaction of the CA with the `kube-scheduler`, as shown in Figure 7.1. Hence, we show that multiple Kubernetes components can be used in parallel in our framework.

7.7 Summary and Discussion

In this chapter, we presented a new approach for connecting a microservice simulation with Kubernetes components based on a generic concept for combining discrete-event simulation and event-driven systems. We implemented the approach with the microservice simulation MiSim and Kubernetes. A core novelty of our approach is that container orchestration mechanisms can be included using their original code artifacts,

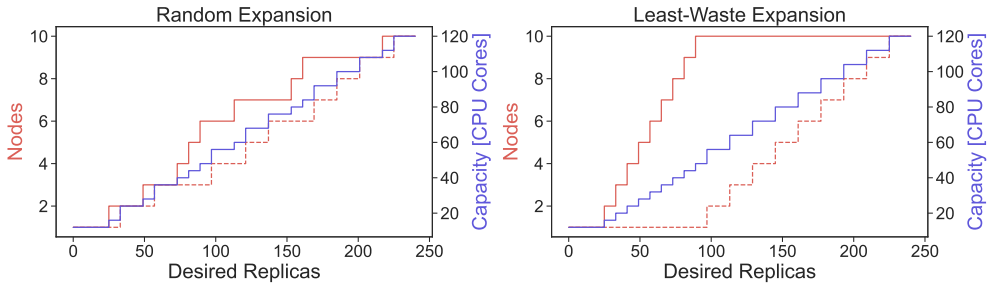


Figure 7.7: Cluster scaling with nodes from small-set (solid) and large-set (dashed).

meaning that their behavior does not have to be modeled in the simulation. A basis for this is the handling of incompatible events and data. We use a black-box information repository to store and forward data to the orchestration mechanisms without any need to process them further in the simulation. We validated our approach in experiments with Kubernetes’ `kube-scheduler` and `cluster-autoscaler`. We have shown that different orchestration mechanism configurations can be represented in the simulation without manual re-implementation using Kubernetes components. Our results show that our approach enables new use cases of the simulation without heavily modifying its core. In the case of the scheduling policies, we also showed that our approach can help increase the simulation’s accuracy.

Different user groups could be interested in our framework. We enable designers of microservice architectures to simulate realistic scenarios with authentic container orchestration. We provide a new lightweight test platform for Kubernetes or container orchestration developers, where real code and new configurations can be tested without requiring a cluster. A significant advantage is that orchestration mechanisms can be included in the simulation in their full complexity without having to model them. Our approach improves and differentiates from related work discussed in Section 3.4 in mainly two points. First, related simulators for microservice performance often consider none or only single orchestration mechanisms. As discussed in Section 7.2, we included all performance-relevant container orchestration tasks that we introduced in Chapter 4 in the simulation. Our case studies and the interactions shown in Figure 7.1 underline the necessity that multiple CO tasks have to be considered and simulated in parallel. Second, our approach allows for the integration of Kubernetes orchestration mechanisms and different configurations of them in their full complexity without any behavior models in the simulation.

One drawback of our approach is that expert knowledge of the orchestrator interfaces is required to integrate its code. This could be an obstacle for microservice designers, less for developers of CO frameworks. To include new orchestration mechanisms in our framework, the interfaces required by the mechanism must be identified and implemented in the adapter. In a concrete use case, the question of whether integration in this form is worthwhile or whether a simple model of the mechanism is sufficient must be answered. The integration is worthwhile if the mechanism has a complex configuration and logic. A simple model should be preferred if it is a relatively

simple algorithm without regular updates. Generally, we assume that the integrated components are independently deployable and work event-based. A current limitation of our approach is that component responses are always executed immediately in the simulation. While this is a good approximation for the `kube-scheduler`, cluster autoscaling includes non-negligible node start times. Extensions in our framework for these purposes are left for future work. Since we do not change the performance models in MiSim, the challenge for accurate predictions for response times and similar metrics remains the correct calibration of the performance models as well as a correct modeling of parametric dependencies [EWvKK18]. However, these weaknesses concern the MiSim core rather than the extension presented in this chapter, which, in theory, would also support other performance models.

All in all, this chapter presented our approach for enriching microservice performance simulation through authentic container orchestration. The established connection widens the use cases and expressiveness of the simulation significantly. The actions of the container orchestration framework are represented more accurately, which also improves the overall accuracy of the simulation (e.g., quantified by predicted response times). Moreover, with our approach, the effect of different orchestrator configurations can be analyzed in the simulation. We have thus made an important contribution that addresses Challenge 4 of this thesis.

We answered RQ D1 ("How to integrate container orchestration frameworks into microservice performance simulations without requiring detailed models of their behavior?") by exploiting the similarities between event-driven architectures and discrete-event simulations. Essentially, relevant events of the container orchestration framework are identified and translated into actions within the simulation. By doing this, we enable the components of the CO framework to run with their original source code and thus eliminate the need for (simplified) behavioral models of them. This concept is also the basis for answering RQ D2 ("How to enable seamless transfer of orchestrator configurations into simulations?"). As components of container orchestration frameworks, such as the `kube-scheduler`, run unmodified source code, all of their configuration options can be supported. This allows users to test configurations from a real cluster in the simulation without major modifications.

Part III

Conclusion and Outlook

Chapter 8

Summary

Cloud computing services have become an integral part of today's world. From a technical point of view, container virtualization plays an important role in provisioning modern cloud applications. Container orchestration frameworks are used to manage the large number and diversity of containerized applications. Thus, they form the basis of modern cloud applications and are largely responsible for their efficient and reliable operation. Modern frameworks offer a multitude of features and configurations, and it is by no means trivial to adapt them to a specific use case. In this thesis, we made four contributions on the topic of benchmarking and modeling of container orchestration frameworks. All contributions offer ways towards efficient and reliable operation of container clusters and modern cloud applications. In the following, we summarize the individual contributions and revisit the goals and research questions stated in Section 1.3.

Contribution I: A Systematic Approach for Benchmarking of Container Orchestration Frameworks. Chapter 4 introduced our benchmarking approach for container orchestration frameworks. In order to answer research question A1 ("Which features of container orchestration frameworks are feasible for evaluation through benchmarking?"), we investigated the question of how to define container orchestration through an analysis of research papers and industrial sources. We derived eight essential tasks of container orchestration frameworks based on which we stated eight testable requirements. Each requirement has been linked to associated performance metrics that should be acquired during benchmarking. We presented a benchmark architecture, consisting of a controller, proxy, load generator, and test containers, that poses minimal assumptions on the system under test. It allows us to acquire the previously stated performance metrics of interest and thus answers research question A2 ("Which metrics can be used to quantify the performance, and how can they be measured?").

Our approach is implemented through our benchmarking framework COFFEE. COFFEE uses abstractions for essential container orchestration tasks and allows for testing different container orchestration frameworks and technology stacks. In the current version, test campaigns can be executed on any Kubernetes or Nomad cluster. Through a human-readable and powerful script language, users can define simple or complex test campaigns that stress various components of container orchestration frameworks. Therefore, COFFEE is our answer to research question A3 ("How to define a benchmark for container orchestration frameworks and enable execution on diverse technology stacks?"). We demonstrated the usage of COFFEE in four case studies focusing on container provisioning and networking, failure recovery, rolling

updates and load balancing, and container storage. We compared different configurations of Kubernetes and Nomad in a self-hosted cloud environment and on the Google Cloud Platform. This contribution enables other researchers and practitioners to execute complex benchmarks for container orchestration frameworks. It supports decisions on which container orchestration framework to choose and how to configure it for a specific use case.

Contribution II: An Empirical Study on Factors Impacting Container Start Times. Chapter 5 presented our empirical study on container images and factors influencing their start times. We first provided a systematic description of the container start process and associated performance metrics. We acquired a large dataset of around 200,000 open-source container images from the popular container registry Docker Hub. The dataset combines popular, frequently used, and less used recent images from many different developers. We thus acquired a diverse dataset that captures container images and their characteristics in the wild well. We reported statistics derived from the manifest parameters of the collected images (such as image size and exposed ports). We developed a measurement process for container start times for Docker containers that only relies on automatically logged timestamps from the Docker daemon and thus avoids additional manual instrumentation. The collected dataset and our measurement process answer research question B1 ("How to acquire a representative dataset of container images and their start times?").

We then conducted numerous start time measurements in order to address research question B2 ("Which factors impact container start times?"). We started by focusing on image configuration parameters and analyzed how they influence the variance and magnitude of container start times. Insights included that the image size and the number of file system layers were the most important image configuration parameters in our case study. We validated our results on the Google Cloud Platform and a self-hosted environment and noted indications that platform parameters significantly influence start times, too. As platform parameters cannot be exhaustively tested due to costs and the virtually infinite number of configurations, we focused on six parameters that can be set when creating virtual machines in the Google Cloud. We applied a Plackett-Burman experimental design and evaluated the effects of different CPU, memory, OS, and disk settings. We observed that the disk type (SSD vs. HDD) is the most important factor among all tested platform parameters. However, by combining image and platform parameters, we saw that both need to be considered for accurate assessment and prediction of container start times. Our contribution provides valuable insights into container start times and gives indications for potential optimizations.

Contribution III: A Continuous, Decentralized Approach to Autoscaling. Chapter 6 described a novel approach to autoscaling that differs conceptually from many state-of-the-art autoscalers. We conducted an extensive experimental study in cooperation with an industry partner that reveals conceptual weaknesses of conventional autoscalers and gives indications for potential improvements that make use of the characteristics of modern cloud applications and workloads. We state six core challenges for autoscalers that are present in production systems. This extensive preliminary study addressed research question C1 ("Which conceptual weaknesses limit the applicability of conventional autoscalers to modern cloud applications and workloads?"). Our

approach to addressing the stated challenges relies on two essential design paradigms: decentralism and continuity. In our approach, service instances make their own independent scaling decisions based on local monitoring data. This decentralized design eliminates the need for complex monitoring systems and improves failure resistance at scale, answering research question C2 ("How to design and implement an autoscaling approach for lightweight execution at scale?").

Instead of viewing autoscaling as a process where the optimal resource provisioning has to be estimated at discrete points in time, we view autoscaling as a continuous process where many small resource changes happen at a high frequency. Due to this high frequency, we achieve great adaptability to different workloads. Our experimental analysis shows especially advantages over state-of-the-art autoscalers in highly dynamic workloads, answering research question C3 ("How to enable adaptability to highly dynamic workloads?"). While many established autoscalers rely on numerous configuration parameters that are hard to optimize for practitioners, we provide an intuitive, flexible, and powerful configuration based on only three parameters. We show that these parameters can significantly and explainably influence autoscaling behavior, delivering our answer to research question C4 ("How can we enable adaptability to different characteristics of cloud applications while keeping the configuration manageable?"). We extensively evaluate our approach, covering an analytical model, a discrete-event simulation, and prototypes for container and VM applications. Our contribution provides a powerful alternative to conventional autoscalers and is able to handle diverse applications and workloads with minimal overheads.

Contribution IV: Enriching Microservice Simulation Through Authentic Container Orchestration. Chapter 7 presented our approach for combining microservice performance simulation and container orchestration. Concretely, we extended the state-of-the-art microservice simulation engine MiSim with container orchestration mechanisms from Kubernetes. As these mechanisms are complex and highly customizable, simplified models or a re-implementation would decrease the accuracy of the simulation and limit its use cases. Our contribution includes container orchestration by using Kubernetes components with their original source code. We exploit analogies between discrete-event simulations and event-driven architectures to realize the integration. From the view of integrated components such as the `kube-scheduler` and `cluster-autoscaler`, the simulation acts as an event producer implementing the Kubernetes API. Consequently, these components behave the same as in an actual cluster. This way of integration provides an answer to research question D1 ("How to integrate container orchestration frameworks into microservice performance simulations without requiring detailed models of their behavior?").

A core challenge for the inter-operability of the simulation and Kubernetes components is the support of Kubernetes concepts and configuration options that cannot be interpreted in the simulation (e.g., affinities for scheduling or expansion policies for cluster autoscaling). To address this challenge, we used a black-box information repository consisting of Kubernetes configuration files that are not interpreted by the simulation but forwarded to the Kubernetes components. We demonstrate the effectiveness of our approach in two case studies focusing on evaluating scheduling policies on a global cluster and expansion policies of cluster autoscaling. We show that differ-

ent configurations of integrated components, specified in the same ways as in a real cluster, can be accurately tested with our simulation. Our approach thus answers research question D2 ("How to enable seamless transfer of orchestrator configurations into simulations?"). Our contribution bridges the gap between microservice performance simulation and container orchestration, enables new use cases for simulations, and increases their accuracy.

Overall Summary. In this thesis, we present four contributions targeting benchmarking and modeling of container orchestration frameworks. Our benchmarking framework COFFEE and our simulation approach allow for a holistic performance assessment of container orchestration frameworks. In particular, different configurations of container orchestration frameworks can be compared. As container orchestration frameworks act as a basis of modern cloud applications, this is a topic of increasing interest in practice, especially in serverless computing, where service providers are responsible for the efficient operation of container clusters and applications. Our contributions in the area of container start times and autoscaling focus on two of the most important tasks of container orchestration frameworks. Our empirical study provides useful data on the characteristics of container applications, increases the understanding of container start times, and gives indications for future optimizations. Our autoscaling approach questions the established way of building autoscaling solutions and addresses prevailing conceptual weaknesses. All contributions offer ways towards efficient and reliable operation of container clusters and modern cloud applications. Together, they significantly advance the research field.

Chapter 9

Open Challenges and Future Work

Benchmarking and modeling of container orchestration frameworks is a very complex and currently highly relevant topic for both research and industry. In the following, we discuss some further thoughts and open challenges that we have not addressed in this thesis. They should also serve as an inspiration for future work in this field.

Connecting the individual contributions. In this thesis, we presented four contributions in the field of performance engineering for container orchestration frameworks. We treated the contributions individually and differentiated them from state-of-the-art. In general, the individual contributions are closely related, and it is worth investigating how they can be combined. For example, we could examine our decentralized scaling approach with the COFFEE benchmarking framework and further compare it to other autoscaling approaches. We could also analyze the potential side effects of decentralized autoscaling (like savings of network bandwidth and resources due to less monitoring overheads). Also, COFFEE naturally has many synergies with our simulation approach. For example, a container orchestration configuration (e.g., a new scheduling approach) can be simulated before it is tested in a real cluster using COFFEE. Our empirical study on the container start times can provide important input for simulations (as model parameters) and decentralized autoscaling (e.g., for the configuration of the waiting time distribution). In the same way, our empirical study motivates us to provide more than one kind of test container in COFFEE (e.g., different image sizes) to investigate the performance of the container cluster concerning these factors as well.

Optimizing container orchestration mechanisms. In addition to the performance evaluation of container orchestration frameworks and their configurations primarily addressed in this thesis, automated optimization methods could be researched in the future. Initial work on this has already delivered promising results (e.g., [WHQ⁺23] and [RND24]). A cost-efficient approach would certainly be simulation-driven optimization. The basis for this can be our approach, which enables testing original configurations in their entire complexity. For example, scheduling processes in Kubernetes could be tuned by testing realistic workloads in the simulation. This requires procedures that not only test random configurations but can also draw conclusions from the simulation results as to which parameters need to be changed. Again, COFFEE can be used to test the behavior of the new configuration or to parameterize the simulation before new configurations go into production. Figure 9.1 illustrates a continuous process for optimizing container orchestration mechanisms. Both COFFEE and our simulation approach can play an essential role here.



Figure 9.1: Continuous process for optimization of container orchestration frameworks.

Developing a holistic benchmark for container orchestration frameworks. This thesis presented a systematic approach for benchmarking container orchestration frameworks and the benchmarking framework COFFEE. The next necessary step is the development of a container orchestration framework. One of the open challenges here is to define relevant and realistic workloads. Currently, there is not a sufficiently large amount of data on the operations of container orchestration frameworks available, such as the number of container starts, updates, failure recoveries, or network traffic. Such a dataset could be used to derive realistic test campaigns that provide substantial guidance for practitioners on the performance of their container clusters. Such a benchmark would also allow for rating configurations and environments and orchestration frameworks in general. This also requires scores, and it should be explored whether smaller subscores are helpful and whether it makes sense to define a unifying common score for orchestration frameworks. Related works indicate that each container orchestration framework has individual strengths and weaknesses [AE24, UFB24].

Analyzing and predicting container readiness times. In this work, we have performed a large-scale analysis of container start times. The restriction to start times allowed us to perform measurements independent of the containers' contents. The associated end-to-end metric is the container readiness time, which indicates when a container is ready to process business workloads. This requires knowledge about the internals of the container since its purpose must be known. A suitable procedure (readiness probe) must be defined to determine the readiness time. A similar systematic, large-scale investigation would give end users good guidance on how container applications should optimally be built and which frameworks and languages influence container readiness times. This is particularly interesting for fine-grained serverless functions

and further addresses the problem of cold start latencies.

Integrating runtime tuning in continuous decentralized autoscaling. As mentioned in Chapter 6, there are numerous possibilities to extend our approach for continuous decentralized autoscaling. One of the most promising approaches is to integrate learning or parameter auto-tuning. In the current version, our prototype allows parameters for scaling functions to be set via environment variables. This could be used by the execution layer or another external component to adjust the algorithm parameters at runtime. As a basis for this, more types of scaling functions and waiting time distributions should first be investigated and evaluated in large-scale environments. Another extension, especially in the context of microservice applications, would be dependency-aware scaling, which can be achieved by informing services which other services they depend on. In a case where Service A calls Service B and Service B is overloaded, Service A could also trigger the scaling of Service B or proactively scale itself. This could reduce possible bottleneck shifting in service chains and match the actual resource demand even more precisely.

Exploring use cases in resource-constrained environments. As discussed in Chapter 3, many related works originate in the area of edge or fog computing. In these environments, computing nodes are resource-limited, and networks only offer limited performance. Our contributions can also be used profitably in these areas. COFFEE could be used analogously to evaluate container orchestration frameworks for resource-constrained nodes and to measure latencies between distributed containers. Decentralized autoscaling could be a resource-saving alternative that does not require a central monitoring system and thus saves data bandwidth. Our simulation approach could also include new models for resource-constrained environments and be applied there. In general, our contributions might further find applications in the emerging concept of the edge-to-cloud or computing continuum.

List of Figures

2.1	Monolithic application vs. microservice application.	18
2.2	Virtual machines vs. containers.	21
2.3	Components of a container cluster with container orchestration.	23
2.4	Schematic overview of Kubernetes control components.	24
2.5	Illustration of joint probability.	31
2.6	Visualization of a simple queueing network.	35
2.7	Point process and describing random variables.	36
4.1	Benchmark architecture and components.	62
4.2	The COFFEE controller in detail.	64
4.3	Container readiness time for different CO frameworks and test clusters.	71
4.4	Container removal time for different CO frameworks and test clusters.	72
4.5	Failure recovery metrics for different CO frameworks and test clusters.	74
4.6	Rolling update metrics for different CO frameworks and test clusters.	75
4.7	Detailed view on rolling updates for exemplary runs.	76
4.8	Readiness times for different storage classes and environments.	79
5.1	Container readiness process and performance metrics.	86
5.2	Empirical distributions of selected features.	90
5.3	Cumulative explained variance ratio by principal components.	92
5.4	Illustration of our sampling process.	93
5.5	Feature distributions for the final sample.	95
5.6	Measurement process for container start times.	96
5.7	Dependence between two selected features and the start time.	100
5.8	Start times and image features in GCP and local environments.	102
5.9	Plackett-Burman feature importance for all platform parameters.	107
5.10	Analysis of which platform parameter setting results in faster start times.	107
6.1	Application under test consisting of services from our industry partner.	116
6.2	Workload and measurement results for reactive CPU scaling.	119
6.3	Behavior of scaling metrics with increasing load.	122
6.4	Workload for the second experiment series.	123
6.5	Motivating example comparing DAS and two HPA configurations.	132
6.6	Overview of our system model.	133
6.7	Visualization of the used scaling functions and their parameters.	137
6.8	Statistical properties of the superimposed process.	141
6.9	Evolution of the discrete-time queueing model.	142
6.10	Evolution of the model's state variables in a scenario with changing load.	147

List of Figures

6.11	Illustration of step response metrics.	148
6.12	Deployed instances over time for different timeout values.	152
6.13	Step response metrics for different readiness times.	153
6.14	Comparing CPU-throttled application and its unthrottled equivalent.	154
6.15	Examples illustrating the influence of various configuration parameters.	155
6.16	Configuration trade-offs and influence of the algorithm parameters.	157
6.17	Implementation of decentralized autoscaling for containers.	158
6.18	Comparison of real system, simulation, and model.	161
6.19	Test trace for comparing the decentralized, HPA, and KPA autoscalers.	162
6.20	Deployed servers and workload in the video streaming experiment.	165
6.21	Quality of service and cost metrics in the video streaming experiment.	167
7.1	Interactions between Kubernetes components.	174
7.2	Overview of our approach and component interactions.	179
7.3	Extended framework version supporting incompatible events and data.	180
7.4	Modeled cluster environment and node locations.	182
7.5	Dependency graph of the test application workload.	183
7.6	Measured and simulated response times for different scheduling policies.	185
7.7	Cluster scaling with nodes from small-set and large-set.	191
9.1	Continuous process for optimization of CO frameworks.	200

List of Tables

2.1	Example for a Plackett-Burman design with three factors.	29
2.2	Selected discrete and continuous distributions.	34
4.1	Commands for test campaign definition.	65
4.2	Average round-trip times between two nodes.	72
4.3	Received user requests per second and instance.	76
4.4	I/O throughput for different storage classes and environments.	78
5.1	Overview of features in the extracted dataset.	89
5.2	Frequencies of different Docker versions in dataset.	90
5.3	Clustering algorithms and tested parameters for identifying dataset strata.	93
5.4	Importance of dataset features for final clustering.	94
5.5	Coefficients of variation for start times of one image configuration.	97
5.6	Univariate and multivariate linear regression coefficients and p -values.	99
5.7	Random forest feature importance values for both test environments.	100
5.8	Error measures for all models and test environments.	102
5.9	Encoding platform parameters to Plackett-Burman factors.	103
5.10	Plackett-Burman design for eight configurations to measure.	104
5.11	Start time statistics for different host configurations.	105
5.12	Start time variability per image on different host configurations.	106
5.13	Random forest feature importance for platform and image parameters.	109
6.1	Quality metrics for CPU autoscaling.	120
6.2	Quality measures for different scaling metrics.	123
6.3	Parameters of the queueing model.	143
6.4	Experiment and parameter overview of our simulation study.	149
6.5	Step response metrics for different system scales.	151
6.6	Step response metrics for different timeout values.	152
6.7	Parameter search space for configuration grid search.	156
6.8	Parameters of the experiments with our prototype.	160
6.9	Scaling metrics for different autoscalers.	164
6.10	Parameters of the video streaming experiment.	166
7.1	Nodes for services selected by different scheduling policies.	185

Acronyms

API Application Programming Interface

AWS Amazon Web Services

CDF cumulative distribution function

CNI Container Network Interface

CO container orchestration

CoV coefficient of variation

DES discrete-event simulation

EDS event-driven system

GCP Google Cloud Platform

GKE Google Kubernetes Engine

HDD hard disk drive

HPA Horizontal Pod Autoscaler

MAE mean absolute error

OCI Open Container Initiative

OS operating system

PMF probability mass function

QoS quality of service

rps requests per second

SLO service-level objective

SSD solid state drive

VM virtual machine

Bibliography

- [AB17] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *8th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2017. [see pages 96 and 110]
- [ADPDM18] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing*, 11(2), 2018. [see pages 4, 5, 8, 43, 113, 114, 125, and 128]
- [ADTK⁺20] Auday Al-Dulaimy, Javid Taheri, Andreas Kassler, M. Reza Hoseiny Farahabady, Shuiguang Deng, and Albert Zomaya. Multiscaler: A multi-loop auto-scaling approach for cloud-based applications. *IEEE Transactions on Cloud Computing*, 10(4), 2020. [see page 45]
- [AE24] Ahmad Alamoush and Holger Eichelberger. Open source container orchestration for industry 4.0—requirements and systematic feature analysis. *International Journal on Software Tools for Technology Transfer*, 26(5), 2024. [see pages 38 and 200]
- [AEADE11] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. In *International conference on database systems for advanced applications*. Springer, 2011. [see page 26]
- [AEKTE12] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*. Association for Computing Machinery, 2012. [see page 44]
- [AETE12] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012. [see pages 44, 125, and 126]
- [AGAT17] Mohammad Sadegh Aslanpour, Mostafa Ghobaei-Arani, and Adel Nadjaran Toosi. Auto-scaling web applications in clouds: A cost-aware approach. *Journal of Network and Computer Applications*, 95, 2017. [see page 46]

Bibliography

- [AHSS13] Fahd Al-Haidari, Mohammed H. Sqalli, and Khaled Salah. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. IEEE, 2013. [see page 46]
- [AJIMHK23] Siti Nuraishah Agos Jawaddi, Azlan Ismail, Muhammad Nur Haziq Mohammad Hatta, and Anis Faqihah Kamarulzaman. Insights into cloud autoscaling: a unique perspective through MDP and DTMC formal models. *The Journal of Supercomputing*, 80(4), 2023. [see page 48]
- [AP18] Arif Ahmed and Guillaume Pierre. Docker container deployment in fog computing infrastructures. In *IEEE International Conference on Edge Computing*. IEEE, 2018. [see pages 5, 42, and 83]
- [APC⁺15] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohamed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE, 2015. [see page 40]
- [ATGC20] Mohammad Sadegh Aslanpour, Adel Nadjaran Toosi, Raj Gaire, and Muhammad Aamir Cheema. Auto-scaling of Web Applications in Clouds: A Tail Latency Evaluation. In *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020. [see pages 4, 5, 8, 114, and 128]
- [ATTG21] Mohammad Sadegh Aslanpour, Adel Nadjaran Toosi, Javid Taheri, and Raj Gaire. Autoscalesim: A simulation toolkit for auto-scaling web applications in clouds. *Simulation Modelling Practice and Theory*, 108, 2021. [see page 51]
- [AW10] Hervé Abdi and Lynne J. Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4), 2010. [see page 91]
- [Bam20] Ilyas Bambrik. A survey on cloud computing simulation and modeling. *SN Computer Science*, 1(5), 2020. [see pages 4 and 49]
- [Bau21] André Bauer. *Automated Hybrid Time Series Forecasting: Design, Benchmarking, and Use Cases*. Universität Würzburg, 2021. [see page 5]
- [BAZA17] Mossaad Ben Ayed, Lilia Zouari, and Mohamed Abid. Software in the loop simulation for robot manipulators. *Engineering, Technology & Applied Science Research*, 7(5), 2017. [see page 51]
- [BB23] Ayush Bhardwaj and Theophilus A. Benson. Kubeklone: A digital twin for simulating edge and cloud microservices. In *Proceedings of the*

- 6th Asia-Pacific Workshop on Networking*. Association for Computing Machinery, 2023. [see page 50]
- [BBM13] Matthias Becker, Steffen Becker, and Joachim Meyer. Simulizar: Design-time modeling and performance analysis of self-adaptive systems. In *Software Engineering*. Gesellschaft für Informatik eV, 2013. [see page 51]
- [BCF⁺24] Davide Borsatti, Walter Cerroni, Luca Foschini, Genady Ya Grabarnik, Lorenzo Manca, Filippo Poltronieri, Domenico Scotece, Larisa Shwartz, Cesare Stefanelli, Mauro Tortonesi, and Mattia Zaccarini. Kubetwin: A digital twin framework for kubernetes deployments at scale. *IEEE Transactions on Network and Service Management*, 21(4), 2024. [see page 50]
- [BDS⁺24] André Bauer, Timo Dittus, Martin Straesser, Alok Kamatar, Matt Baughman, Lukas Beierlieb, Marius Hadry, Daniel Grillmeyer, Yannik Lubas, Samuel Kounev, Ian Foster, and Kyle Chard. Unveiling temporal performance deviation: Leveraging clustering in microservices performance analysis. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2024. [see page xiv]
- [BDSCDM24] Marco Barletta, Luigi De Simone, Raffaele Della Corte, and Catello Di Martino. Failover timing analysis in orchestrating container-based critical applications. In *2024 19th European Dependable Computing Conference (EDCC)*. IEEE, 2024. [see pages 5 and 39]
- [BDvZ⁺21] Stephen Burroughs, Helge Dickel, Martin van Zijl, Vladimir Podolskiy, Michael Gerndt, Robi Malik, and Panos Patros. Towards Autoscaling with Guarantees on Kubernetes Clusters. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2021. [see page 48]
- [BEF⁺19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, and Felix Willnecker. How is performance addressed in devops? In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2019. [see page 128]
- [BG18] Ryan Bankston and Jinhua Guo. Performance of container network technologies in cloud environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018. [see pages 5, 40, and 73]
- [BGHK18] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. On the Value of Service Demand Estimation for Auto-Scaling.

- In *Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018)*. Springer, 2018. [see page 127]
- [BGP⁺24] André Bauer, Maxime Gonthier, Haochen Pan, Ryan Chard, Daniel Grzenda, Martin Straesser, J. Gregory Pauloski, Alok Kamatar, Matt Baughman, Nathaniel Hudson, Ian Foster, and Kyle Chard. An empirical investigation of container building strategies and warm times to reduce cold starts in scientific computing serverless functions. In *2024 IEEE 20th International Conference on e-Science (e-Science)*. IEEE, 2024. [see pages xiv and 42]
- [BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3), 2016. [see page 18]
- [BHS⁺19] André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4), 2019. [see pages 44 and 125]
- [BKFL17] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. Delivering elastic containerized cloud applications to enable devops. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017. [see page 50]
- [BLV⁺19] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. Chamulleon: Coordinated auto-scaling of micro-services. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019. [see pages 44 and 46]
- [BPC⁺24] André Bauer, Haochen Pan, Ryan Chard, Yadu Babuji, Josh Bryan, Devesh Tiwari, Ian Foster, and Kyle Chard. The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems*, 153, 2024. [see page 4]
- [BQ20] Luciano Baresi and Giovanni Quattrocchi. A Simulation-based Comparison between Industrial Autoscaling Solutions and COCOS for Cloud Applications. In *IEEE International Conference on Web Services (ICWS)*. IEEE, 2020. [see pages 38, 113, and 128]
- [BQT22] Luciano Baresi, Giovanni Quattrocchi, and Damian Andrew Tamburri. Microservice architecture practices and experience: a focused look on docker configuration files. *arXiv preprint arXiv:2212.03107*, 2022. [see page 42]

- [BSB⁺22] André Bauer, Martin Straesser, Lukas Beierlieb, Maximilian Meissner, and Samuel Kounev. Automated triage of performance change points using time series analysis and machine learning. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2022. [see page xv]
- [BSL⁺23] André Bauer, Martin Straesser, Mark Leznik, Lukas Beierlieb, Marius Hadry, Nathaniel Hudson, Kyle Chard, Samuel Kounev, and Ian Foster. Searching for the ground truth: Assessing the similarity of benchmarking runs. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2023. [see page xiv]
- [BSP⁺20] Szilárd Bozóki, Jenő Szalontai, Dániel Pethő, Imre Kocsis, András Pataricza, Péter Suskovics, and Benedek Kovács. Application of extreme value analysis for characterizing the execution time of resilience supporting mechanisms in kubernetes. In *Dependable Computing - EDCC 2020 Workshops*. Springer International Publishing, 2020. [see page 39]
- [BT19] Mehdi Mokhtar Belkhiria and Cédric Tedeschi. Design and Evaluation of Decentralized Scaling Mechanisms for Stream Processing. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019. [see page 48]
- [BW21] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*. CEUR Workshop Proceedings, 2021. [see page 38]
- [BWB⁺19] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9), 2019. [see pages 6 and 51]
- [BZH⁺20] André Bauer, Marwin Züfle, Nikolas Herbst, Samuel Kounev, and Valentin Curtef. Telescope: An automatic feature extraction and transformation approach for time series forecasting on a level-playing field. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020. [see page 124]
- [CAB⁺22] Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microservice architecture reconstruction and visualization techniques: A review. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022. [see page 17]
- [Car22] Carmen Carrión. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Computing Surveys*, 55(7), 2022. [see pages 52 and 182]

Bibliography

- [Cas19a] Emiliano Casalicchio. Container orchestration: A survey. In *Systems Modeling: Methodologies and Tools*. Springer International Publishing, 2019. [see page 57]
- [Cas19b] Emiliano Casalicchio. A study on performance measures for auto-scaling cpu-intensive containerized applications. *Cluster Computing*, 22(3), 2019. [see page 45]
- [CBY18] Tao Chen, Rami Bahsoon, and Xin Yao. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Computing Surveys*, 51(3), 2018. [see pages 5, 26, 43, and 128]
- [CCDN⁺12] Nicolo Maria Calcavecchia, Bogdan Alexandru Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. DEPAS: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94, 2012. [see pages 47 and 48]
- [CCVK13] Mohan Baruwal Chhetri, Sergei Chichin, Quoc Bao Vo, and Ryszard Kowalczyk. Smart cloudbench – automated performance benchmarking of the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013. [see page 39]
- [CHL⁺08] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008. [see pages 45 and 47]
- [CKP12] Bogdan Alexandru Caprarescu, Eva Kaslik, and Dana Petcu. Theoretical analysis and tuning of decentralized probabilistic auto-scaling. *arXiv preprint arXiv:1202.2981*, 2012. [see page 47]
- [Clo24] Cloud Native Computing Foundation. Cloud native computing foundation annual survey 2023, 2024. <https://www.cncf.io/reports/cncf-annual-survey-2023/>. [see pages iii, vii, 1, and 2]
- [CMKS09] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *2009 IEEE International Conference on e-Business Engineering*. IEEE, 2009. [see page 45]
- [CNH⁺19] Tatsuhiro Chiba, Rina Nakazawa, Hiroshi Horii, Sahil Suneja, and Seetharami Seelam. Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019. [see page 51]

- [Com17] Comscore Inc. 2017 U.S. Cross-Platform Future in Focus, 2017. <https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/2017-US-Cross-Platform-Future-in-Focus>. [see page 165]
- [CP12] Bogdan Alexandru Caprarescu and Dana Petcu. Decentralized probabilistic auto-scaling for heterogeneous systems. *arXiv preprint arXiv:1203.3885*, 2012. [see page 47]
- [CPB19] Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. Fogdocker: Start container now, fetch image later. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2019. [see pages 5 and 42]
- [CRB⁺11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1), 2011. [see pages 6 and 49]
- [CSFN20] Roger Pueyo Centelles, Mennan Selimi, Felix Freitag, and Leandro Navarro. Redemon: Resilient decentralized monitoring system for edge infrastructures. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020. [see page 45]
- [CSOQ21] Clément Courageux-Sudan, Anne-Cécile Orgerie, and Martin Quinson. Automated performance prediction of microservice applications using simulation. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021. [see page 51]
- [CSW⁺17] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the Docker container ecosystem on GitHub. In *IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 2017. [see page 42]
- [Dat24] Datadog Knowledge Center. What is auto-scaling?, 2024. <https://www.datadoghq.com/knowledge-center/auto-scaling/>. [see pages 125 and 128]
- [DBM19] Alfonso Delgado-Bonal and Alexander Marshak. Approximate entropy and sample entropy: A comprehensive tutorial. *Entropy*, 21(6), 2019. [see page 126]
- [DEC21] Yoann Desmouceaux, Marcel Enguehard, and Thomas H. Clausen. Joint Monitorless Load-Balancing and Autoscaling for Zero-Wait-Time in Data Centers. *IEEE Transactions on Network and Service Management*, 18(1), 2021. [see page 47]

Bibliography

- [DFLM19] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150, 2019. [see page 17]
- [DGK07] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. A generic solution to software-in-the-loop. In *MILCOM 2007 - IEEE Military Communications Conference*. IEEE, 2007. [see page 51]
- [DMM⁺10] Xavier Dutreilh, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. From data center resource allocation to control theory and back. In *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 2010. [see page 46]
- [DS18] Naghme Dezhabad and Saeed Sharifian. Learning-based dynamic scalable load-balanced firewall as a service in network function-virtualized cloud computing environments. *The Journal of Supercomputing*, 74(7), 2018. [see page 47]
- [dSPPSM23] Thiago Felipe da Silva Pinheiro, Paulo Pereira, Bruno Silva, and Paulo Maciel. A performance modeling framework for microservices-based cloud infrastructures. *The Journal of Supercomputing*, 79(7), 2023. [see page 51]
- [DTLD22] Borislav Dordevic, Valentina Timcenko, Milovan Lazic, and Nikola Davidovic. Performance comparison of docker and podman container-based virtualization. In *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 2022. [see pages 5 and 41]
- [DTR⁺18] Wito Delnat, Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. Association for Computing Machinery, 2018. [see pages 5 and 39]
- [Dur24] Thomas Durieux. Empirical study of the docker smells impact on the image size. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024. [see page 43]
- [dVRS21] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. Understanding the performance of container execution environments. In *6th International Workshop on Container Technologies and Container Clouds*. Association for Computing Machinery, 2021. [see page 41]
- [DYX⁺20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings*

of the *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2020. [see page 42]

- [EBG⁺21] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*. Association for Computing Machinery, 2021. [see page 162]
- [EJPG20] Lennart Espe, Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance evaluation of container runtimes. In *10th International Conference on Cloud Computing and Services Science*. SciTePress, 2020. [see page 41]
- [EK14] Benjamin Erb and Frank Kargl. Combining discrete event simulations and event sourcing. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*. Association for Computing Machinery, 2014. [see page 51]
- [EWvKK18] Simon Eismann, Jürgen Walter, Jóakim von Kistowski, and Samuel Kounev. Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018. [see pages 26 and 192]
- [Fis70] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 1970. [see page 28]
- [FJFCSG⁺20] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Jesus Lopez-Ballester, and Maximo Cobos. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting internet of things applications. *Journal of Network and Computer Applications*, 169, 2020. [see page 38]
- [FSW⁺24] Sebastian Frank, Martin Straesser, Lion Wagner, Patrick Haas, Alireza Hakamian, Samuel Kounev, and André van Hoorn. Simulating microservice-based architectures for resilience assessment enriched by authentic container orchestration. In *Software Engineering 2024 (SE 2024)*. Gesellschaft für Informatik e.V., 2024. [see page xv]
- [FWH⁺22] Sebastian Frank, Lion Wagner, Alireza Hakamian, Martin Straesser, and André van Hoorn. Misim: A simulator for resilience assessment of microservice-based architectures. In *IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022. [see pages xiv, 6, 12, 51, 173, 174, 175, 176, and 183]

Bibliography

- [GAAC24] Darek Gajewski, Muhammad Ashfakur Rahman Arju, Amr S. Abdelfattah, and Tomas Cerny. Case study: Applying automated optimization tooling to microservice environments that scale safely at ancestry.com and the learnings. In *European Conference on Software Architecture*. Springer, 2024. [see page 45]
- [Gar24] Gartner. Gartner forecasts worldwide public cloud end-user spending to surpass \$675 billion in 2024, 2024. <https://www.gartner.com/en/newsroom/press-releases/2024-05-20-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-surpass-675-billion-in-2024>. [see pages iii, vii, and 1]
- [GEH⁺17] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Association for Computing Machinery, 2017. [see page 47]
- [GGA24] Mohsen Ghorbian and Mostafa Ghobaei-Arani. A survey on the cold start latency approaches in serverless computing: an optimization-based perspective. *Computing*, 106(11), 2024. [see pages 5 and 42]
- [GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*. IEEE, 2010. [see page 45]
- [GHBRK12] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4), 2012. [see page 47]
- [GNI⁺19] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. Monitorless: Predicting performance degradation in cloud applications with machine learning. In *Proceedings of the 20th International Middleware Conference*. Association for Computing Machinery, 2019. [see page 45]
- [GS01] Stanton A. Glantz and Bryan K. Slinker. *Primer of Applied Regression & Analysis of Variance*. McGraw-Hill, Inc., New York, 2001. [see pages 98 and 99]
- [GSA⁺18] Tom Goethals, Merlijn Sebrechts, Ankita Atrey, Bruno Volckaert, and Filip De Turck. Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications. In *IEEE 8th International Symposium on Cloud and Service Computing*. IEEE, 2018. [see pages 5 and 41]

- [GSAN⁺22] Tom Goethals, Merlijn Sebrechts, Mays Al-Naday, Bruno Volckaert, and Filip De Turck. A functional and performance benchmark of lightweight virtualization platforms for edge computing. In *IEEE International Conference on Edge Computing and Communications*. IEEE, 2022. [see pages 5 and 41]
- [GSC⁺21] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2021. [see pages xv, 124, and 128]
- [GSFP20] Federico Ghirardini, Areeg Samir, Ilenia Fronza, and Claus Pahl. Model-driven simulation for performance engineering of kubernetes-style cloud cluster architectures. In *Advances in Service-Oriented and Cloud Computing: Workshops of ESOC 2018*. Springer, 2020. [see page 50]
- [GSLI11] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011. [see page 46]
- [Gun10] Neil J. Gunther. UNIX load average part 1: How it works. Technical report, TeamQuest Corporation, 2010. [see page 144]
- [GVDGB17] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9), 2017. [see page 49]
- [GZL⁺16] Steffen Gebert, Thomas Zinner, Stanislav Lange, Christian Schwartz, and Phuoc Tran-Gia. Discrete-Time Analysis: Deriving the Distribution of the Number of Events in an Arbitrarily Distributed Interval. Technical report, University of Würzburg, 2016. [see page 143]
- [Has21] Wilhelm Hasselbring. Benchmarking as empirical standard in software engineering research. In *Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, 2021. [see page 81]
- [HBH07] Dieter Hildebrandt, Ludger Bischofs, and Wilhelm Hasselbring. Realpeer—a framework for simulation-based development of peer-to-peer systems. In *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*. IEEE, 2007. [see page 51]

Bibliography

- [HBK20] Sara Hassan, Rami Bahsoon, and Rick Kazman. Microservice transition and its granularity problem: A systematic mapping study. *Software: Practice and Experience*, 50(9), 2020. [see page 18]
- [HBLR20] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, 2020. [see page 43]
- [HH22] Sören Henning and Wilhelm Hasselbring. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering*, 27(6), 2022. [see pages 39 and 82]
- [HLS⁺22] Elia Henrichs, Veronika Lesch, Martin Straesser, Samuel Kounev, and Christian Krupitzer. A literature review on optimization techniques for adaptation planning in adaptive systems: State of the art and research directions. *Information and Software Technology*, 149, 2022. [see page xiii]
- [HM20] José Herrera and Germán Moltó. Toward bio-inspired auto-scaling algorithms: An elasticity approach for container orchestration platforms. *IEEE Access*, 8, 2020. [see page 47]
- [HMH18] Tobias Hoßfeld, Florian Metzger, and Poul E. Heegaard. Traffic modeling for aggregated periodic IoT data. In *21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2018. [see page 130]
- [HR96] Gerhard Hasslinger and Erik S. Rieger. Analysis of open discrete time queueing networks: A refined decomposition approach. *Journal of the Operational Research Society*, 47(5), 1996. [see pages 139 and 140]
- [HS04] Daniel P. Heyman and Matthew J. Sobel. *Stochastic models in operations research: stochastic optimization*. Courier Corporation, 2004. [see page 36]
- [HSL⁺16a] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy Docker containers. In *14th USENIX Conference on File and Storage Technologies*. USENIX Association, 2016. [see page 41]
- [HSL16b] Matthias Hellerer, Martin J. Schuster, and Roy Lichtenheldt. Software-in-the-loop simulation of a planetary rover. In *The International Symposium on Artificial Intelligence, Robotics and Automation in Space*. DLR, 2016. [see page 51]

- [HSL22] Shihong Hu, Weisong Shi, and Guanghui Li. CEC: A containerized edge computing framework for dynamic resource provisioning. *IEEE Transactions on Mobile Computing*, 22(7), 2022. [see page 83]
- [HSS23] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on {Meta’s} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023. [see page 2]
- [HvHK⁺17] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. Association for Computing Machinery, 2017. [see page 50]
- [IDCJ11] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6), 2011. [see page 44]
- [IFDB22] Shashikant Ilager, Jakob Fahringer, Samuel Carlos de Lima Dias, and Ivona Brandic. Demon: Decentralized monitoring for highly volatile edge environments. In *IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022. [see page 45]
- [IOY⁺11] Alexandru Iosup, Simon Ostermann, M. Nezhil Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6), 2011. [see page 39]
- [IP23] InfraCloud and Dhout Pooja. Should You Always Use a Service Mesh?, 2023. <https://www.infracloud.io/blogs/should-you-always-use-service-mesh/>. [see page 127]
- [JBB⁺19] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019. [see pages 5 and 39]
- [JD18] Bibal Benifa J.V. and Dejeý Dharma. Has: Hybrid auto-scaler for resource scaling in cloud environment. *Journal of Parallel and Distributed Computing*, 120, 2018. [see page 44]
- [JD19] Bibal Benifa J.V. and Dejeý Dharma. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications*, 24(4), 2019. [see page 46]

Bibliography

- [JJI22] Siti Nuraishah Agos Jawaddi, Muhammad Hamizan Johari, and Azlan Ismail. A review of microservices autoscaling with formal verification perspective. *Software: Practice and Experience*, 52(11), 2022. [see page 48]
- [JLZL13] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013. [see page 46]
- [JPG19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2019. [see pages 6 and 51]
- [JS22] Lara Lorna Jiménez and Olov Schelén. Hydra: Decentralized location-aware orchestration of containerized applications. *IEEE Transactions on Cloud Computing*, 10(4), 2022. [see page 48]
- [KCH09] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing*. Association for Computing Machinery, 2009. [see pages 45, 46, and 125]
- [KE23] Heiko Koziolk and Nafise Eskandani. Lightweight kubernetes distributions: A performance comparison of microk8s, k3s, k0s, and microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2023. [see page 38]
- [KF22] Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *The Journal of Supercomputing*, 78(11), 2022. [see page 39]
- [KFW⁺22] Thomas Kudla, Mattia Fogli, Sean Webb, Geert Pingen, Niranjan Suri, and Harrie Bastiaansen. Quantifying the performance of cloud-oriented container orchestrators on emulated tactical networks. *IEEE Communications Magazine*, 60(5), 2022. [see page 38]
- [Kha17] Asif Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, 4(5), 2017. [see page 57]
- [KHA⁺23] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. Serverless computing: What it is, and what it is not? *Communications of the ACM*, 66(9), 2023. [see pages 1 and 19]

- [KKK⁺19] Tamas Kiss, Peter Kacsuk, Jozsef Kovacs, Botond Rakoczi, Akos Hajnal, Attila Farkas, Gregoire Gesmier, and Gabor Terstyanszky. Micado—microservice-based cloud application-level dynamic orchestrator. *Future Generation Computer Systems*, 94, 2019. [see page 48]
- [KLvK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Springer, 2020. [see pages 5, 7, 28, 30, 55, 59, and 110]
- [KOM⁺19] Masoud Koleini, Carlos Oviedo, Derek McAuley, Charalampos Rotso, Anil Madhavapeddy, Thomas Gazagnaire, Magnus Skejgstad, and Richard Mortier. Fractal: Automated application scaling. *arXiv preprint arXiv:1902.09636*, 2019. [see page 47]
- [Kov19] József Kovács. Supporting programmable autoscaling rules for containers and virtual machines on clouds. *Journal of Grid Computing*, 17(4), 2019. [see page 48]
- [KSZB23] Floriment Klinaku, Sandro Speth, Markus Zilch, and Steffen Becker. Hitchhiker’s guide for explainability in autoscaling. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2023. [see page 46]
- [KT20] Rakesh Kumar and Balasubramanian Thangaraju. Performance analysis between RunC and Kata container runtime. In *IEEE International Conference on Electronics, Computing and Communication Technologies*. IEEE, 2020. [see pages 5 and 41]
- [KTADK23] Michel Gokan Khan, Javid Taheri, Auday Al-Dulaimy, and Andreas Kassler. Perfsim: A performance simulator for cloud native microservice chains. *IEEE Transactions on Cloud Computing*, 11(2), 2023. [see page 51]
- [KTK23] Shahidullah Kaiser, Ali Şaman Tosun, and Turgay Korkmaz. Benchmarking container technologies on arm-based edge devices. *IEEE Access*, 11, 2023. [see pages 5 and 41]
- [KUT⁺24] Tamas Kiss, Amjad Ullah, Gabor Terstyanszky, Odej Kao, Soren Becker, Yiannis Verginadis, Antonis Michalas, Vlado Stankovski, Attila Kertesz, Elisa Ricci, Jörn Altmann, Bernhard Egger, Francesco Tusa, Jozsef Kovacs, and Robert Lovas. Swarmchestrator: Towards a fully decentralised framework for orchestrating applications in the cloud-to-edge continuum. In *International Conference on Advanced Information Networking and Applications*. Springer, 2024. [see page 48]
- [LAF⁺19] Michael Littlely, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitrios Skourtis, Mohamed Mohamed,

- Heiko Ludwig, Yue Cheng, and Ali R. Butt. Bolt: Towards a scalable Docker registry via hyperconvergence. In *IEEE 12th International Conference on Cloud Computing*. IEEE, 2019. [see page 42]
- [LBKG18] Ashish Lingayat, Ranjana R. Badre, and Anil Kumar Gupta. Performance evaluation for deploying Docker containers on baremetal and virtual machine. In *3rd International Conference on Communication and Electronics Systems*. IEEE, 2018. [see page 41]
- [LC16] Philipp Leitner and Jürgen Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology*, 16(3), 2016. [see pages 82, 110, and 170]
- [LGC⁺22] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *USENIX Annual Technical Conference*. USENIX Association, 2022. [see page 83]
- [LK07] Averill M. Law and W. David Kelton. *Simulation modeling and analysis*. McGraw-hill New York, 2007. [see page 177]
- [LLJ⁺19] Qing Lei, Weidong Liao, Yingtao Jiang, Mei Yang, and Haifeng Li. Performance and scalability testing strategy based on kubemark. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE, 2019. [see pages 5 and 39]
- [LLT⁺23] Jiong Lou, Hao Luo, Zhiqing Tang, Weijia Jia, and Wei Zhao. Efficient container assignment and layer sequencing in edge computing. *IEEE Transactions on Services Computing*, 16(2), 2023. [see page 91]
- [LNK20] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020. [see page 42]
- [LP99] Tim Lechler and Bernd Page. Desmo-j: An object oriented discrete simulation framework in java. In *Proceedings of the 11th European Simulation Symposium (ESS)*. SCS Europe, 1999. [see page 175]
- [LSBK25] Yannik Lubas, Martin Straesser, André Bauer, and Samuel Kounev. Generating executable microservice applications for performance benchmarking. In *Proceedings of the 2025 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2025. [see page xiv]

- [LSL19] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering*, 24(4), 2019. [see pages 110 and 170]
- [LSR⁺25] Yannik Lubas, Martin Straesser, Ivo Rohwer, Samuel Kounev, and André Bauer. Microservice applications and their workloads on github. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2025. [see page xv]
- [LTB⁺25a] Christopher Lohse, Diego Tsutsumi, Amadou Ba, Pavithra Harsha, Chitra Subramanian, Martin Straesser, and Marco Ruffini. Causal latency modelling for cloud microservices. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*. IEEE, 2025. [see page xiv]
- [LTB⁺25b] Christopher Lohse, Diego Tsutsumi, Amadou Ba, Pavithra Harsha, Chitra K. Subramanian, Martin Straesser, and Marco Ruffini. Causal discovery for cloud microservice architectures. In *AAAI 2025 Workshop on Deployable AI*. Association for the Advancement of Artificial Intelligence, 2025. [see page xv]
- [LWC⁺23] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology*, 32(5), 2023. [see page 83]
- [LXY⁺22] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the 13th Symposium on Cloud Computing*. Association for Computing Machinery, 2022. [see page 125]
- [MAJ⁺21] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin van Eyk, Laurens Versluis, Vincent van Beek, and Alexandru Iosup. Opende 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021. [see pages 6 and 50]
- [Mat09] Rami J. Matarneh. Self-adjustment time quantum in round robin algorithm depending on burst time of the now running processes. *American Journal of Applied Sciences*, 6(10), 2009. [see page 175]
- [MB22] Souheir Merkouche and Chafia Bouanaka. A Hybrid approach for containerized Microservices auto-scaling. In *IEEE/ACS 19th International*

Bibliography

- Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2022. [see page 48]
- [MBB23] Souheir Merkouche, Chafia Bouanaka, and Elhadj Benkhelifa. A petri net-based formal modeling for microservices auto-scaling. In *20th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2023. [see page 48]
- [MBKV23] Mohamed Mekki, Bouziane Brik, Adlen Ksentini, and Christos Verikoukis. XAI-Enabled Fine Granular Vertical Resources Autoscaler. In *IEEE 9th International Conference on Network Softwarization (Net-Soft)*. IEEE, 2023. [see pages 46 and 129]
- [MDBvL17] Debankur Mukherjee, Souvik Dhara, Sem C. Borst, and Johan S. H. van Leeuwen. Optimal Service Elasticity in Large-Scale Distributed Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1), 2017. [see page 133]
- [Mel11] Peter Mell. The nist definition of cloud computing. *NIST Special Publication*, 800(2011), 2011. [see page 1]
- [MFG⁺20] Ying Mao, Yuqi Fu, Suwen Gu, Sudip Vhaduri, Long Cheng, and Qingzhi Liu. Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes. *arXiv preprint arXiv:2010.10350*, 2020. [see page 38]
- [MGD⁺23] Pedro Melo, Lucas Gama, Jamilson Dantas, David Beserra, and Jean Araujo. Performance evaluation of container management tasks in os-level virtualization platforms. In *2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2023. [see pages 5 and 41]
- [MGYT20] Arash Mazidi, Mehdi Golsorkhtabaramiri, and Meisam Yadollahzadeh Tabari. An autonomic risk- and penalty-aware resource allocation with probabilistic resource scaling mechanism for multilayer cloud resource provisioning. *International Journal of Communication Systems*, 33(7), 2020. [see page 48]
- [Mic06] Brenda M. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12), 2006. [see page 177]
- [MK17] Ilias Mavridis and Helen Karatza. Performance and overhead study of containers running on top of virtual machines. In *IEEE 19th Conference on Business Informatics*. IEEE, 2017. [see page 41]
- [MLKL19] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. In *Proceedings of the 29th Annual International*

- Conference on Computer Science and Software Engineering*. Association for Computing Machinery, 2019. [see page 126]
- [MLS⁺17] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2017. [see page 41]
- [MRBnA16] Víctor Medel, Omer Rana, José ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2016. [see page 50]
- [MSR16] Ganapathy Mahalakshmi, Subbiah Sridevi, and Shyamsundar Rajaram. A survey on forecasting of time series data. In *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*. IEEE, 2016. [see page 125]
- [MST⁺23] Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023. [see page 126]
- [MTA⁺21] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupperecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. CNSBench: A cloud native storage benchmark. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 2021. [see pages 5 and 40]
- [MTK22] Mohamed Mekki, Nassima Toumi, and Adlen Ksentini. Microservices configurations and the impact on the performance in cloud native environments. In *IEEE 47th Conference on Local Computer Networks (LCN)*. IEEE, 2022. [see pages 2, 37, and 173]
- [NCMB21] Fotis Nikolaidis, Antony Chazapis, Manolis Marazakis, and Angelos Bilas. Frisbee: automated testing of cloud-native applications in kubernetes. *arXiv preprint arXiv:2109.10727*, 2021. [see page 39]
- [New83] Paul Newbold. Arima model building and the time series analysis approach to forecasting. *Journal of forecasting*, 2(1), 1983. [see page 120]
- [NFT20] Elisabetta Di Nitto, Luca Florio, and Damian A. Tamburri. Autonomic Decentralized Microservices: The Gru Approach and Its Evaluation. In *Microservices: Science and Engineering*. Springer International Publishing, 2020. [see page 48]

Bibliography

- [NGN17] Zahra Nikdel, Bing Gao, and Stephen W. Neville. Dockersim: Full-stack simulation of container-based software-as-a-service (saas) cloud deployments and environments. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 2017. [see page 50]
- [NLV⁺19] Seyed Mohammad Reza Nouri, Han Li, Srikumar Venugopal, Wenxia Guo, MingYun He, and Wenhong Tian. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. *Future Generation Computer Systems*, 94, 2019. [see page 48]
- [NSG⁺13] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, 2013. [see page 125]
- [NYK⁺20] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors*, 20(16), 2020. [see page 125]
- [OHS19] Amr Osman, Simon Hanisch, and Thorsten Strufe. Seconetbench: A modular framework for secure container networking benchmarks. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019. [see pages 5 and 40]
- [OYZ⁺18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference*. USENIX Association, 2018. [see page 42]
- [PAEr⁺16] Alessandro Vittorio Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. Peas: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(4), 2016. [see page 48]
- [PB46] Robin L. Plackett and J. Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4), 1946. [see pages 11 and 29]
- [PBCK23] EunChan Park, KyeongDeok Baek, Eunho Cho, and In-Young Ko. Fully Decentralized Horizontal Autoscaling for Burst of Load in Fog Computing. *Journal of Web Engineering*, 22(6), 2023. [see page 47]
- [PCB⁺19] Yao Pan, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, and Richard Sinnott. A performance comparison of cloud-based container orchestration tools. In *IEEE International Conference on Big Knowledge (ICBK)*. IEEE, 2019. [see pages 2, 37, 38, and 173]

- [PDCB17] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience*, 47(4), 2017. [see page 49]
- [PFS19] Arnaldo Pereira Ferreira and Richard Sinnott. A performance evaluation of containers running on managed kubernetes services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019. [see page 82]
- [PGE91] Steven M. Pincus, Igor M. Gladstone, and Richard A. Ehrenkranz. A regularity statistic for medical data analysis. *Journal of clinical monitoring*, 7(4), 1991. [see page 125]
- [PvdH07] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16, 2007. [see page 17]
- [PWZ⁺23] Zhicheng Pan, Yihang Wang, Yingying Zhang, Sean Bin Yang, Yunyao Cheng, Peng Chen, Chenjuan Guo, Qingsong Wen, Xiduo Tian, Yunliang Dou, Zhiqiang Zhou, Chengcheng Yang, Aoying Zhou, and Bin Yang. Magicscaler: Uncertainty-aware, predictive autoscaling. *Proceedings of the VLDB Endowment*, 16(12), 2023. [see page 48]
- [QCB18] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Computing Surveys*, 51(4), 2018. [see pages 5, 43, and 113]
- [QIP⁺24] Giovanni Quattrocchi, Emilio Incerto, Riccardo Pincioli, Catia Trubiani, and Luciano Baresi. Autoscaling solutions for cloud applications under dynamic workloads. *IEEE Transactions on Services Computing*, 17(3), 2024. [see pages 113 and 162]
- [QWW⁺20] Shijun Qin, Heng Wu, Yuewen Wu, Bowen Yan, Yuanjia Xu, and Wenbo Zhang. Nuka: A generic engine with millisecond initialization for serverless computing. In *IEEE International Conference on Joint Cloud Computing*. IEEE, 2020. [see pages 5 and 42]
- [RAH⁺20] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, and Biswanath Mukherjee. Auto-scaling network service chains using machine learning and negotiation game. *IEEE Transactions on Network and Service Management*, 17(3), 2020. [see page 125]
- [RB19] Maria A. Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5), 2019. [see pages 37 and 57]

Bibliography

- [RDG11] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011. [see page 46]
- [RFS⁺20] Krzysztof Rządca, Paweł Findeisen, Jacek Swiderski, Przemysław Zych, Przemysław Broniek, Jarek Kusmieriek, Paweł Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery, 2020. [see page 46]
- [RND24] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Simuscale: Optimizing parameters for autoscaling of serverless edge functions through co-simulation. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, 2024. [see pages 51 and 199]
- [Rob00] Thomas G. Robertazzi. *Computer networks and systems: queueing theory and performance evaluation*. Springer Science & Business Media, 2000. [see page 33]
- [Rou87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20, 1987. [see page 93]
- [RS97] Alvin C. Rencher and Michael G. Schimek. Methods of multivariate analysis. *Computational Statistics*, 12(4), 1997. [see page 94]
- [RT19] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019. [see pages 5 and 41]
- [Rus18] Gabriele Russo. Towards Decentralized Auto-Scaling Policies for Data Stream Processing Applications. In *ZEUS*. CEUR Workshop Proceedings, 2018. [see page 48]
- [SBL⁺23] Martin Straesser, André Bauer, Robert Leppich, Nikolas Herbst, Kyle Chard, Ian Foster, and Samuel Kounev. An empirical study of container image configurations and their impact on start times. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2023. [see pages xiii, 11, and 84]
- [SCA⁺21] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. React: Enabling real-time container orchestration. In *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021. [see page 57]

- [SCST16] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Yong Chiang Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*. Association for Computing Machinery, 2016. [see page 20]
- [SEK24] Martin Straesser, Nicholas Erhard, and Samuel Kounev. An empirical study on the impact of selected host configuration parameters on container start times. *Softwaretechnik-Trends*, 44(4), 2024. [see pages xv, 11, and 84]
- [Ser21] Amazon Web Services. Predictive scaling for ec2, 2021. <https://aws.amazon.com/en/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/>. [see page 125]
- [SEvK⁺23] Martin Straesser, Simon Eismann, Jóakim von Kistowski, André Bauer, and Samuel Kounev. Autoscaler evaluation and configuration: A practitioner’s guideline. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2023. [see pages xiv, 12, and 115]
- [SFSF19] Robert-Steve Schmoll, Tobias Fischer, Hani Salah, and Frank H. P. Fitzek. Comparing and evaluating application-specific boot times of virtualized instances. In *2nd IEEE 5G World Forum*. IEEE, 2019. [see pages 5, 41, and 83]
- [SGE17] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. Association for Computing Machinery, 2017. [see page 43]
- [SGJN19] Parminder Singh, Pooja Gupta, Kiran Jyoti, and Anand Nayyar. Research on auto-scaling of web applications in cloud: survey, trends and future directions. *Scalable Computing: Practice and Experience*, 20(2), 2019. [see pages 5, 26, 27, 43, 44, 45, 115, and 170]
- [SGL⁺25] Martin Straesser, Stefan Geißler, Stanislav Lange, Lukas Kilian Schumann, Tobias Hoffeld, and Samuel Kounev. Trust your local scaler: A continuous, decentralized approach to autoscaling. *Performance Evaluation*, 167, 2025. [see pages xiii, 12, and 115]
- [SGLI11] Bradley Simmons, Hamoun Ghanbari, Marin Litoiu, and Gabriel Iszlai. Managing a saas application in the cloud using paas policy sets and a strategy-tree. In *2011 7th International Conference on Network and Service Management*. IEEE, 2011. [see page 45]
- [SGvK⁺22] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. Why is it not solved yet? challenges for production-ready autoscaling. In *Proceedings of the 2022*

Bibliography

- ACM/SPEC on International Conference on Performance Engineering*. Association for Computing Machinery, 2022. [see pages xiv, 12, and 115]
- [Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3), 1948. [see page 126]
- [SHF⁺24] Martin Straesser, Patrick Haas, Sebastian Frank, Alireza Hakamian, André van Hoorn, and Samuel Kounev. Kubernetes-in-the-loop: Enriching microservice simulation through authentic container orchestration. In *Performance Evaluation Methodologies and Tools*. Springer Nature Switzerland, 2024. [see pages xiii, 13, and 175]
- [SHG⁺13] Marcio Silva, Michael R. Hines, Diego Gallo, Qi Liu, Kyung Dong Ryu, and Dilma da Silva. Cloudbench: Experiment automation for cloud environments. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2013. [see page 39]
- [SKG⁺21] Parminder Singh, Avinash Kaur, Pooja Gupta, Sukhpal Singh Gill, and Kiran Jyoti. Rhas: robust hybrid auto-scaling for web applications in cloud computing. *Cluster Computing*, 24(2), 2021. [see page 44]
- [SMBK23] Martin Straesser, Jonas Mathiasch, André Bauer, and Samuel Kounev. A systematic approach for benchmarking of container orchestration frameworks. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2023. [see pages xiii, 10, and 56]
- [SSFR⁺24] Rute C. Sofia, Josh Salomon, Simone Ferlin-Reiter, Luis Garcés-Erice, Peter Urbanetz, Harald Mueller, Rizkallah Touma, Alejandro Espinosa, Luis M. Contreras, Vasileios Theodorou, Nikos Psaromanolakis, Lefteris Mamatas, Vassilis Tsaoussidis, Xiaoming Fu, Tingting Yuan, Alberto del Rio, David Jiménez, Andries Stam, Efterpi Paraskevoulakou, Panagiotis Karamolegkos, Vitor Vieira, Josep Marrat, Ignacio Mariscal Prusiel, Dorine Matzakou, John Soldatos, David Remon, and Marco Jahn. A framework for cognitive, decentralized container orchestration. *IEEE Access*, 12, 2024. [see page 48]
- [SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2011. [see page 45]
- [SSJ24] Siddhartha Singh, FNU Shivanshi, and Rachit Jain. A framework for evaluating container performance across diverse kubernetes environments. In *2024 IEEE 13th International Conference on Cloud Networking (CloudNet)*. IEEE, 2024. [see page 39]

- [SSK19] Tamanna Siddiqui, Shadab Alam Siddiqui, and Najeeb Ahmad Khan. Comprehensive analysis of container technology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*. IEEE, 2019. [see page 57]
- [SSW03] Carl-Erik Särndal, Bengt Swensson, and Jan Wretman. *Model Assisted Survey Sampling*. Springer Science & Business Media, 2003. [see pages 92 and 94]
- [SWK22] Lukas Stahlbock, Jan Weber, and Frank Köster. An optimization approach of container startup times for time-sensitive embedded systems. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2022. [see pages 5 and 42]
- [Tay97] John R. Taylor. *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, 1997. [see page 28]
- [TBD⁺20] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, and Ferruccio Damiani. A formal model of the kubernetes container framework. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2020. [see page 50]
- [TBVL⁺18] Eddy Truyen, Matt Bruzek, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Evaluation of container orchestration systems for deploying and managing nosql database clusters. In *IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018. [see pages 2, 37, and 173]
- [TCM20] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, 9(1), 2020. [see page 48]
- [TCTH23] Thomas Tournaire, Hind Castel-Taleb, and Emmanuel Hyon. Efficient Computation of Optimal Thresholds in Cloud Auto-Scaling Systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 8(4), 2023. [see page 48]
- [TDFS20] Laszlo Toka, Gergely Dobreff, Balazs Fodor, and Balazs Sonkoly. Adaptive ai-based auto-scaling for kubernetes. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020. [see page 48]

Bibliography

- [Ten24] Tenesys. Cloud Monitoring: Benefits, Challenges, And Best Practices for Your Infrastructure, 2024. <https://tenesys.io/en/cloud-monitoring-benefits-challenges-and-best-practices-for-your-infrastructure/>. [see page 127]
- [TGAPG24] Mohammad Tari, Mostafa Ghobaei-Arani, Jafar Pouramini, and Mohsen Ghorbian. Auto-scaling mechanisms in serverless computing: A comprehensive review. *Computer Science Review*, 53, 2024. [see page 45]
- [TGH21] Phuoc Tran-Gia and Tobias Hoßfeld. *Performance Modeling and Analysis of Communication Networks: A Lecture Note*. BoD–Books on Demand, 2021. [see pages 33, 35, 36, 139, and 140]
- [The23] The New Stack. Why kubernetes has emerged as the ‘os’ of the cloud, 2023. <https://thenewstack.io/why-kubernetes-has-emerged-as-the-os-of-the-cloud/>. [see page 2]
- [Tig24] Tigera. Service Mesh: Benefits, Challenges, and 7 Key Concepts, 2024. <https://www.tigera.io/learn/guides/service-mesh/>. [see page 127]
- [TKT18] Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. Virtualization techniques compared: Performance, resource, and power usage overheads in clouds. In *ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2018. [see pages 5, 40, and 83]
- [TRS⁺19] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating docker storage performance: from workloads to graph drivers. *Cluster Computing*, 22, 2019. [see page 41]
- [TSZN21] Sergii Telenyk, Oleksii Sopov, Eduard Zharikov, and Grzegorz Nowakowski. A comparison of kubernetes and kubernetes-compatible platforms. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. IEEE, 2021. [see page 38]
- [TTEP20] Mulugeta Ayalew Tamiru, Johan Tordsson, Erik Elmroth, and Guillaume Pierre. An experimental evaluation of the kubernetes cluster autoscaler in the cloud. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2020. [see page 52]
- [TTT⁺18] Fan-Hsun Tseng, Ming-Shiun Tsai, Chia-Wei Tseng, Yao-Tsung Yang, Chien-Chang Liu, and Li-Der Chou. A lightweight autoscaling mechanism for fog computing in industrial applications. *IEEE Transactions on Industrial Informatics*, 14(10), 2018. [see pages 45 and 115]

- [TVLLJ19] Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Association for Computing Machinery, 2019. [see pages 2, 37, and 173]
- [TVLP⁺19] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences*, 9(5), 2019. [see page 38]
- [UFB24] Muhammad Usman, Simone Ferlin, and Anna Brunstrom. Performance analysis of lightweight container orchestration platforms for edge-based iot applications. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2024. [see pages 38 and 200]
- [UKK⁺23] Amjad Ullah, Tamas Kiss, József Kovács, Francesco Tusa, James Deslauriers, Huseyin Dagdeviren, Resmi Arjun, and Hamed Hamzeh. Orchestration in the cloud-to-things compute continuum: taxonomy, survey and future directions. *Journal of Cloud Computing*, 12(1), 2023. [see page 48]
- [Unt24] Unthinkable Solutions. Applications of AI in autoscaling of cloud infrastructure, 2024. <https://www.unthinkable.co/blog/applications-of-ai-in-autoscaling-of-cloud-infrastructure/>. [see page 129]
- [USC⁺08] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1), 2008. [see pages 44 and 125]
- [VBT⁺19] Stef Verreydt, Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, and Wouter Joosen. Leveraging kubernetes for adaptive and cost-efficient resource management. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*. Association for Computing Machinery, 2019. [see pages 2, 37, and 173]
- [VDR⁺20] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Jorge Larracochea, and Christina Herzog. Draceo: A smart simulator to deploy energy saving methods in microservices based networks. In *IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2020. [see pages 6 and 51]
- [vKAH⁺15] Jóakim von Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to build a benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2015. [see pages 5 and 28]

Bibliography

- [vKES⁺18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018. [see page 183]
- [vKHK14] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. Limbo: a tool for modeling variable load intensities. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2014. [see page 187]
- [VKT20] William Viktorsson, Cristian Klein, and Johan Tordsson. Security-performance trade-offs of Kubernetes container runtimes. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2020. [see pages 5 and 41]
- [VMK22] Alexandros Valantasis, Nikos Makris, and Thanasis Korakis. Orchestration software for resource constrained datacenters: an experimental evaluation. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE, 2022. [see page 39]
- [vMvHH11] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. In *5th European Conference on Software Architecture (ECSA)*. Springer, 2011. [see page 52]
- [VMZK21] Alexandros Valantasis, Nikos Makris, Christos Zarafetas, and Thanasis Korakis. Experimental evaluation of orchestration software for virtual network functions. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2021. [see page 39]
- [VSTK21] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. A kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software*, 175, 2021. [see pages 5 and 39]
- [VWWD24] Simon Volpert, Sascha Winkelhofer, Stefan Wesner, and Jörg Domaschka. An empirical analysis of common oci runtimes' performance isolation capabilities. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2024. [see page 41]
- [WHQ⁺23] Shilin Wen, Rui Han, Ke Qiu, Xiaoxin Ma, Zeqing Li, Hongjie Deng, and Chi Harold Liu. K8ssim: A simulation tool for kubernetes schedulers and its applications in scheduling algorithm optimization. *Micro-machines*, 14(3), 2023. [see pages 50 and 199]

- [WKKG19] Muhammad Wajahat, Alexei Karve, Andrzej Kochut, and Anshul Gandhi. MLscale: A machine learning based application-agnostic autoscaler. *Sustainable Computing: Informatics and Systems*, 22, 2019. [see page 126]
- [WKL⁺19] Yi Wei, Daniel Kudenko, Shijun Liu, Li Pan, Lei Wu, and Xiangxu Meng. A reinforcement learning based auto-scaling approach for saas providers in dynamic cloud environment. *Mathematical Problems in Engineering*, 2019(1), 2019. [see page 46]
- [WMG⁺17] Tianming Wei, Madhav Malhotra, Bing Gao, Tomas Bednar, Derek Jacoby, and Yvonne Coady. No such thing as a “free launch”? Systematic benchmarking of containers. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. IEEE, 2017. [see page 41]
- [WMMY93] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability and statistics for engineers and scientists*. Macmillan New York, 1993. [see page 30]
- [WRK⁺19] Junzo Watada, Arunava Roy, Ruturaj Kadikar, Hoang Pham, and Bing Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7, 2019. [see pages 1, 3, 5, and 41]
- [WTGGH19] Florian Wamser, Phuoc Tran-Gia, Stefan Geißler, and Tobias Hoffeld. Modeling of Traffic Flows in Internet of Things Using Renewal Approximation. In *Advances in Optimization and Decision Science for Society, Services and Enterprises*. Springer International Publishing, 2019. [see page 130]
- [WVMN20] Nan Wang, Blesson Varghese, Michail Matthaiou, and Dimitrios S. Nikolopoulos. Enorm: A framework for edge node resource management. *IEEE Transactions on Services Computing*, 13(6), 2020. [see page 48]
- [WZL⁺22] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, Kadangode K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. DeepScaling: Microservices Autoscaling for Stable CPU Utilization in Large Scale Cloud Systems. In *Proceedings of the 13th Symposium on Cloud Computing*. Association for Computing Machinery, 2022. [see page 126]
- [WZW20] Mingming Wang, Dongmei Zhang, and Bin Wu. A cluster autoscaler based on multiple node types in kubernetes. In *IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 2020. [see page 52]

Bibliography

- [XFJ16] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of KVM, Docker and unikernels in a cloud platform. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2016. [see pages 5, 41, and 83]
- [XSY⁺23] Le Xu, Divyanshu Saxena, Neeraja J. Yadwadkar, Aditya Akella, and Indranil Gupta. Dirigo: Self-scaling Stateful Actors For Serverless Real-time Data Processing. *arXiv preprint arXiv:2308.03615*, 2023. [see pages 48 and 113]
- [XYW⁺23] Minxian Xu, Lei Yang, Yang Wang, Chengxi Gao, Linfeng Wen, Guoyao Xu, Liping Zhang, Kejiang Ye, and Chengzhong Xu. Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster. *Software: Practice and Experience*, 54(1), 2023. [see page 113]
- [XZPY21] Jingxuan Xie, Shubo Zhang, Maolin Pan, and Yang Yu. A Comprehensive Evaluation Method for Container Auto-Scaling Algorithms on Cloud. In *Computer Supported Cooperative Work and Social Computing*. Springer Singapore, 2021. [see page 131]
- [YKXY19] Ruozhou Yu, Vishnu Teja Kilari, Guoliang Xue, and Dejun Yang. Load balancing for interdependent iot microservices. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE, 2019. [see pages 39, 61, and 82]
- [YNC20] Takeshi Yoshimura, Rina Nakazawa, and Tatsuhiro Chiba. Image-Jockey: A framework for container performance engineering. In *IEEE 13th International Conference on Cloud Computing*. IEEE, 2020. [see page 43]
- [ZGC⁺21] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, 2021. [see page 4]
- [ZGD19] Yanqi Zhang, Yu Gan, and Christina Delimitrou. μ qsim: Enabling accurate and scalable simulation for interactive microservices. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019. [see page 51]
- [ZHZ22] Changpeng Zhu, Bo Han, and Yinliang Zhao. A bi-metric autoscaling approach for n-tier web applications on kubernetes. *Frontiers of Computer Science*, 16, 2022. [see page 45]
- [ZLZ⁺24] Ding Zou, Wei Lu, Zhibo Zhu, Xingyu Lu, Jun Zhou, Xiaojin Wang, Kangyu Liu, Kefan Wang, Renen Sun, and Haiqing Wang. Optscaler: A

- collaborative framework for robust autoscaling in the cloud. *Proceedings of the VLDB Endowment*, 17(12), 2024. [see page 44]
- [ZSX⁺23] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2023. [see page 127]
- [ZTA⁺19] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. Large-scale analysis of the Docker Hub dataset. In *IEEE International Conference on Cluster Computing*. IEEE, 2019. [see pages 42 and 91]
- [ZTA⁺21] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K. Paul, Keren Chen, and Ali R. Butt. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(4), 2021. [see page 91]
- [ZU22] Yuxuan Zhao and Alexandru Uta. Tiny autoscalers for tiny workloads: Dynamic cpu allocation for serverless functions. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022. [see page 46]
- [ZWDZ17] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, 2017. [see pages 5, 40, and 73]
- [ZZM⁺23] Zhiqiang Zhou, Chaoli Zhang, Lingna Ma, Jing Gu, Huajie Qian, Qingsong Wen, Liang Sun, Peng Li, and Zhimin Tang. Ahpa: adaptive horizontal pod autoscaling systems on alibaba cloud container service for kubernetes. In *Proceedings of the AAAI Conference on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence, 2023. [see page 44]