Simon Eismann

Performance Engineering of Serverless Applications and Platforms

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2022

Gutachter: Prof. Dr. Samuel Kounev, Julius-Maximilians-Universität Würzburg, Germany

Datum der mündlichen Prüfung:

Abstract

Serverless computing is an emerging cloud computing paradigm that offers a high-level application programming model with utilization-based billing. It enables the deployment of cloud applications without managing the underlying resources or worrying about other operational aspects. Function-as-a-Service (FaaS) platforms implement serverless computing by allowing developers to execute code on-demand in response to events with continuous scaling while having to pay only for the time used with sub-second metering. Cloud providers have further introduced many fully managed services for databases, messaging buses, and storage that also implement a serverless computing model. Applications composed of these fully managed services and FaaS functions are quickly gaining popularity in both industry and in academia.

However, due to this rapid adoption, much information surrounding serverless computing is inconsistent and often outdated as the serverless paradigm evolves. This makes the performance engineering of serverless applications and platforms challenging, as there are many open questions, such as: What types of applications is serverless computing well suited for, and what are its limitations? How should serverless applications be designed, configured, and implemented? Which design decisions impact the performance properties of serverless platforms and how can they be optimized? These and many other open questions can be traced back to an inconsistent understanding of serverless applications and platforms, which could present a major roadblock in the adoption of serverless computing.

In this thesis, we address the lack of performance knowledge surrounding serverless applications and platforms from multiple angles: we conduct empirical studies to further the understanding of serverless applications and platforms, we introduce automated optimization methods that simplify the operation of serverless applications, and we enable the analysis of design tradeoffs of serverless platforms by extending white-box performance modeling. More precisely, we make the following contributions in this thesis:

- *Evaluation of the Characteristics and Performance of Serverless Applications*. Our first contribution improves the understanding of serverless applications on the basis of two empirical studies. The first study systematically collects close to ninety serverless applications from different sources and characterizes them through a comprehensive pair-reviewing process with regard to sixteen characteristics. We complement this with a meta-analysis that compares the results of existing studies to our results to analyze community consensus. The second study

is an exploratory case study on the stability of performance measurements of serverless applications, in which we investigate the baseline performance variability and the stability of performance tests over time.

In these studies, we identify a community consensus on eight characteristics of serverless applications and provide the first quantitative data on eight further characteristics. Additionally, we detect short-term performance fluctuations and observe multiple long-term performance changes where undisclosed, provider-side changes permanently alter the performance of serverless applications.

- *Automating Operational Tasks of Serverless Applications*. Our second contribution introduces two approaches to automate the operational tasks associated with serverless applications. We introduce an approach to predict the optimal resource size of serverless functions that first implements a serverless function generator capable of generating a large number of synthetic serverless functions. Then, we measure the execution time and resource consumption metrics of many synthetic functions for different resource sizes and construct a multi-target regression model to predict the optimal size of previously unseen serverless functions. Further, to enable the cost optimization of serverless workflows, we apply mixture density networks to predict the response time and output parameter distributions for individual serverless functions. Based on these individual function models, a Monte-Carlo simulation derives cost predictions for entire serverless workflows.

  In our evaluation, our resource size optimization model selects the optimal resource size for 79.0% of the serverless functions in four realistic applications, which results in an average speedup of 39.7% while simultaneously reducing average costs by 2.6%. For two audio transcription workflows, our cost prediction approach achieves a mean workflow cost prediction accuracy of 96.2%.

- *Enabling White-Box Performance Modeling and Simulation of Serverless Platforms*. Our third contribution introduces two approaches that extend white-box performance models to make them applicable for the analysis of serverless platforms. The first approach is directed at speeding up the simulation time required to solve white-box performance models. We introduce a generic modeling approach that enables a parallel description of subsystems as both fast-to-solve black-box performance models and as traditional queueing models. Further, we extend an existing discrete event simulation solver to support these hybrid models. The second approach addresses the issue that white-box performance modeling does not support the integration of empirically observed relationships between model parameters. We propose a novel approach to modeling empirical parametric dependencies in architectural performance

models that derives a fully parameterized performance model by transforming the empirical information into a directed graph.

In our evaluation on a distributed, component-based system of medium size, our approach maintains sufficient prediction accuracy and achieves speedups of up to 94.8%. In two case studies in the context of a media store, our approach for the integration of empirical parametric dependencies achieves a mean prediction error for utilization and response time of less than 5% and 10%, respectively.

Together, these contributions address the lack of performance engineering knowledge and techniques for serverless applications and platforms. The empirical data collected in this thesis provides quantitative evidence for many commonly debated questions surrounding serverless applications and provides the tools to analyze and optimize the efficiency and performance of both serverless applications and platforms. Further, this work provides a foundation for further research on the performance engineering of serverless applications and platforms.

Zusammenfassung

Serverless Computing ist ein neues Cloud-Computing-Paradigma, das ein High-Level-Anwendungsprogrammiermodell mit nutzungsbasierter Abrechnung bietet. Es ermöglicht die Bereitstellung von Cloud-Anwendungen, ohne dass die zugrunde liegenden Ressourcen verwaltet werden müssen oder man sich um andere betriebliche Aspekte kümmern muss. Function-as-a-Service (FaaS)-Plattformen implementieren Serverless Computing, indem sie Entwicklern die Möglichkeit geben, Code nach Bedarf als Reaktion auf Ereignisse mit kontinuierlicher Skalierung auszuführen, während sie nur für die genutzte Zeit mit sekundengenauer Abrechnung zahlen müssen. Cloud-Anbieter haben darüber hinaus viele vollständig verwaltete Dienste für Datenbanken, Messaging-Busse und Orchestrierung eingeführt, die ebenfalls ein Serverless Computing-Modell implementieren. Anwendungen, die aus diesen vollständig verwalteten Diensten und FaaS-Funktionen bestehen, werden sowohl in der Industrie als auch in der Wissenschaft immer beliebter.

Aufgrund dieser schnellen Verbreitung sind jedoch viele Informationen zum Serverless Computing inkonsistent und oft veraltet, da sich das Serverless Paradigma weiterentwickelt. Dies macht das Performanz-Engineering von Serverless Anwendungen und Plattformen zu einer Herausforderung, da es viele offene Fragen gibt, wie zum Beispiel: Für welche Arten von Anwendungen ist Serverless Computing gut geeignet und wo liegen seine Grenzen? Wie sollten Serverless Anwendungen konzipiert, konfiguriert und implementiert werden? Welche Designentscheidungen wirken sich auf die Performanzeigenschaften von Serverless Plattformen aus und wie können sie optimiert werden? Diese und viele andere offene Fragen lassen sich auf ein uneinheitliches Verständnis von Serverless Anwendungen und Plattformen zurückführen, was ein großes Hindernis für die Einführung von serverlosem Computing darstellen könnte.

In dieser Arbeit adressieren wir den Mangel an Performanzwissen zu Serverless Anwendungen und Plattformen aus mehreren Blickwinkeln: Wir führen empirische Studien durch, um das Verständnis von Serverless Anwendungen und Plattformen zu fördern, wir stellen automatisierte Optimierungsmethoden vor, die das benötigte Wissen für den Betrieb von Serverless Anwendungen reduzieren, und wir erweitern die White-Box-Performanzmodellierungerung für die Analyse von Designkompromissen von Serverless Plattformen. Genauer gesagt, leisten wir in dieser Arbeit die folgenden Beiträge:

- *Bewertung der Eigenschaften und der Performanz von Serverless Anwendungen*. Unser erster Beitrag verbessert das Verständnis von Serverless Anwendungen auf

der Grundlage von zwei empirischen Studien. Die erste Studie sammelt systematisch knapp neunzig Serverless Anwendungen aus verschiedenen Quellen und charakterisiert sie durch ein umfassendes Pair-Reviewing-Verfahren im Hinblick auf sechzehn Merkmale. Wir ergänzen dies durch eine Meta-Analyse, die die Ergebnisse bestehender Studien mit unseren Ergebnissen vergleicht, um die Übereinstimmung zu analysieren. Bei der zweiten Studie handelt es sich um eine explorative Fallstudie zur Stabilität von Performanzmessungen bei Serverless Anwendungen, in der wir die grundlegende Performanzvariabilität und die Stabilität von Performanztests im Laufe der Zeit untersuchen.

In diesen Studien ermitteln wir eine Übereinstimmung zu acht Merkmalen von Serverless Anwendungen und liefern die ersten quantitativen Daten zu acht weiteren Merkmalen. Darüber hinaus stellen wir kurzfristige Performanzschwankungen fest und beobachten langfristige Performanzänderungen, bei denen anbieterseitige Änderungen die Performanz von serverlosen Anwendungen dauerhaft verändern.

- *Automatisierung der Betriebsaufgaben von Serverless Anwendungen*. Unser zweiter Beitrag enthält zwei Ansätze zur Automatisierung der mit dem Betrieb von Serverless Anwendungen verbundenen Aufgaben. Wir stellen einen Ansatz zur Vorhersage der optimalen Ressourcengröße von Serverless Funktionen vor. Dabei wird zunächst ein Generator für Serverless Funktionen implementiert, der eine große Anzahl synthetischer Serverless Funktionen erzeugen kann. Anschließend messen wir die Ausführungszeit und den Ressourcenverbrauch vieler synthetischer Funktionen für verschiedene Resourcengrößen und erstellen ein Multitarget-Regressionsmodell, um die optimale Größe von Serverless Funktionen vorherzusagen. Um die Kostenoptimierung von Serverless Workflows zu ermöglichen, wenden wir außerdem Mixture Density Networks an, um die Verteilung der Ausführungszeit und der Ausgabeparameter für einzelne Serverless Funktionen vorherzusagen. Basierend auf diesen individuellen Funktionsmodellen leitet eine Monte-Carlo-Simulation Kostenvorhersagen für komplette Serverless Workflows ab.

  In unserer Evaluierung wählt unser Modell zur Optimierung der Ressourcengröße die optimale Ressourcengröße für 79,0% der Serverless Funktionen von vier realistischen Anwendungen aus, was zu einer durchschnittlichen Beschleunigung von 39,7% bei gleichzeitiger Reduzierung der durchschnittlichen Kosten um 2,6% führt. Für zwei Audiotranskriptions-Workflows erreicht unser Ansatz zur Kostenvorhersage eine durchschnittliche Genauigkeit der Workflow-Kostenvorhersage von 96,2%.

- *Ermöglichung der White-Box-Performanzmodellierung und Simulation von Serverless Plattformen*. Unser dritter Beitrag stellt zwei Ansätze vor, die White-Box-Performanzmodelle erweitern, um sie für die Analyse von Serverless Plattfor-

men anwendbar zu machen. Der erste Ansatz zielt darauf ab, die für die Lösung von White-Box-Performance-Modellen erforderliche Simulationszeit zu beschleunigen. Wir führen einen generischen Modellierungsansatz ein, der eine parallele Beschreibung von Subsystemen sowohl als schnell zu lösende Black-Box-Performanzmodelle als auch als traditionelle Warteschlangen-Modelle ermöglicht, und erweitern einen bestehenden Solver für die diskrete Ereignissimulation, um diese Hybridmodelle zu unterstützen. Der zweite Ansatz befasst sich mit dem Problem, dass die White-Box-Performanzmodellierung die Integration von empirisch beobachteten Beziehungen zwischen Modellparametern nicht unterstützt. Wir schlagen einen neuen Ansatz zur Modellierung von empirischen Zusammenhängen in White-Box-Performanzmodellen vor, der ein parametrisiertes Performanzmodell durch Umwandlung der empirischen Informationen in einen gerichteten Graphen erzeugt.

In unserer Evaluierung auf einem verteilten, komponentenbasierten System mittlerer Größe erreicht unser Ansatz eine ausreichende Vorhersagegenauigkeit und eine Beschleunigung von bis zu 94,8%. In zwei Fallstudien im Kontext eines Medienshops erreicht unser Ansatz zur Integration empirischer parametrischer Abhängigkeiten einen mittleren Vorhersagefehler weniger als 5% für Auslastungen und weniger als 10% für Antwortzeiten.

Gemeinsam adressieren diese Beiträge den Mangel an Performanzmanagement-Wissen und -Techniken für Serverless Anwendungen und Plattformen. Die in dieser Arbeit gesammelten empirischen Daten liefern quantitative Ergebnisse für viele häufig diskutierte Fragen im Zusammenhang mit Serverless Anwendungen und bieten die Werkzeuge zur Analyse und Optimierung der Effizienz und Leistung sowohl von Serverless Anwendungen als auch von Serverless Plattformen. Darüber hinaus bildet diese Arbeit die Grundlage für weitere Forschungen zum Performancemanagement von Serverless Anwendungen und Plattformen.

Contents

Peer-Reviewed Publications

Journal and Magazine Articles

[Eis+22] Simon Eismann, Diego Elias Costa, Lizhi Liao, Cor-Paul Bezemer, Weiyi Shang, Andre Hoorn, Samuel Kounev. "A Case Study on the Stability of Performance Tests for Serverless Applications". In: *Journal of Systems and Software* (2022).

[Eis+21a] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, Alexandru Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* (2021).

[Eis+21b] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (2021), pp. 32–39.

[Gro+21a] Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, Samuel Kounev. "SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31.

[Eyk+19] Erwin Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18.

[Spi+19] Simon Spinner, Johannes Grohmann, Simon Eismann, Samuel Kounev. "Online model learning for self-aware computing infrastructures". In: *Journal of Systems and Software* 147 (2019), pp. 1–16.

Full Conference Papers

[Eis+21c] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, Samuel Kounev. "Sizeless: Predicting the Optimal Size of Serverless Functions". In: *Proceedings of the 22nd International Middleware Conference (MIDDLEWARE)*. 2021, pp. 248–259.

[Eis+20a] Simon Eismann, Johannes Grohmann, Erwin Eyk, Nikolas Herbst, Samuel Kounev. "Predicting the Costs of Serverless Workflows". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2020, pp. 265–276.

[Eis+20b] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, André Hoorn. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2020, pp. 265–276.

[Eis+19] Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim Kistowski, Samuel Kounev. "Integrating Statistical Response Time Models in Architectural Performance Models". In: *Proceedings of the 27th IEEE International Conference on Software Architecture* (*ICSA*). 2019, pp. 71–80.

[Eis+18] Simon Eismann, Jürgen Walter, Jóakim Kistowski, Samuel Kounev. "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time". In: *Proceedings of the 26th IEEE International Conference on Software Architecture* (*ICSA*). 2018, pp. 135–147.

[Str+22] Martin Straesser, Johannes Grohmann, Jóakim Kistowski, Simon Eismann, Andre Bauer, Samuel Kounev. "Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling". In: *Proceedings of the 13th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2022.

[Gro+21b] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, Samuel Kounev. "SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications". In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2021, pp. 165–176.

[Bau+21] André Bauer, Marwin Züfle, Simon Eismann, Johannes Grohmann, Nikolas Herbst, Samuel Kounev. "Libra: A Benchmark for Time Series Forecasting Methods". In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2021, pp. 189–200.

[Gro+20] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev, Jörg Domaschka. "Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS". In: *Proceedings of the 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (*MASCOTS*). 2020, pp. 1–8.

[Her+20] Stefan Herrnleben, Piotr Rygielski, Johannes Grohmann, Simon Eismann, Tobias Hossfeld, Samuel Kounev. "Model-Based Performance Predictions for SDN-Based Networks: A Case Study". In: *Proceedings of the 2020 International Conference on Measurement, Modelling and Evaluation of Computing Systems* (*MMB*). 2020, pp. 82–98.

[Gro+19a] Johannes Grohmann, Simon Eismann, Sven Elflein, Manar Mazkatli, Jóakim Kistowski, Samuel Kounev. "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (*MASCOTS*). 2019, pp. 309–322.

[Kis+18] Jóakim Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, Samuel Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis,*

and Simulation of Computer and Telecommunication Systems (*MASCOTS*). 2018, pp. 223–236.

Short Conference Papers

[Lez+22] Mark Leznik, Johannes Grohmann, Nina Kliche, Andre Bauer, Daniel Seybold, Simon Eismann, Samuel Kounev, Jörg Domaschka. "Same, Same, but Dissimilar: Exploring Measurements for Workload Time-series Similarity". In: *Proceedings of the 13th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2022.

[Bez+19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André Hoorn, Mónica Villavicencio, Jürgen Walter, Felix Willnecker. "How is Performance Addressed in DevOps?" In: *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). 2019, pp. 45–50.

Workshop Papers

[Eyk+20] Erwin Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, Alexandru Iosup. "Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark". In: *Companion of the 11th ACM/SPEC International Conference on Performance Engineering* (*ICPE-C*). 2020, pp. 26–31.

[GEK19] Johannes Grohmann, Simon Eismann, Samuel Kounev. "On Learning Parametric Dependencies from Monitoring Data". In: *Proceedings of the 10th Symposium on Software Performance* (*SSP*). 2019.

[Gro+19b] Johannes Grohmann, Simon Eismann, Andre Bauer, Marwin Zuefle, Nikolas Herbst, Samuel Kounev. "Utilizing Clustering to Optimize Resource Demand Estimation Approaches". In: *Proceedings of the 4th IEEE International Workshops on Foundations and Applications of Self* Systems* (*FAS*W*). 2019, pp. 134–139.

[Bau+19] André Bauer, Simon Eismann, Johannes Grohmann, Nikolas Herbst, Samuel Kounev. "Systematic Search for Optimal Resource Configurations of Distributed Applications". In: *Proceedings of the 4th IEEE International Workshops on Foundations and Applications of Self* Systems* (*FAS*W*). 2019, pp. 120–125.

[Ack+18] Vanessa Ackermann, Johannes Grohmann, Simon Eismann, Samuel Kounev. "Black-box Learning of Parametric Dependencies for Performance Models". In: *Proceedings of 13th International Workshop on Models@run.time* (*MRT*). 2018.

[Wal+17] Jürgen Walter, Simon Eismann, Nikolai Reed, Samuel Kounev. "Providing Model-Extraction-as-a-Service for Architectural Performance Models". In: *Proceedings of the 10th Symposium on Software Performance* (*SSP*). 2017.

Vision, Position, and Tutorial Papers

[Dom+21] Jörg Domaschka, Mark Leznik, Daniel Seybold, Simon Eismann, Johannes Grohmann, Samuel Kounev. "Buzzy: Towards Realistic DBMS Benchmarking via Tailored, Representative, Synthetic Workloads". In: *Companion of the 12th ACM/SPEC International Conference on Performance Engineering* (*ICPE-C*). 2021, pp. 175–178.

[GEK18] Johannes Grohmann, Simon Eismann, Samuel Kounev. "The Vision of Self-Aware Performance Models". In: *Companion of the 26th IEEE International Conference on Software Architecture Companion* (*ICSA-C*). 2018, pp. 60–63.

[Wal+18] Jürgen Walter, Simon Eismann, Johannes Grohmann, Dušan Okanovic, Samuel Kounev. "Tools for Declarative Performance Engineering". In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering* (*ICPE-C*). 2018, pp. 53–56.

[Iff+18] Lukas Iffländer, Jürgen Walter, Simon Eismann, Samuel Kounev. "The Vision of Self-Aware Reordering of Security Network Function Chains". In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering* (*ICPE-C*). 2018, pp. 1–4.

[Eyk+18] Erwin Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, Simon Eismann. "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures". In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering* (*ICPE-C*). 2018, pp. 21–24.

Technical Reports

[Eis+20c] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. *A Review of Serverless Use Cases and their Characteristics*. Tech. rep. 2020.

Software Contributions

[Kis+19] Jóakim Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, Samuel Kounev. *TeaStore*. Standard Performance Evaluation Corporation Research Group (SPEC RG) Tool. `https://research.spec.org/tools/overview/teastore.html`. 2019.

Chapter 1

Introduction

Cloud computing promises computing as a utility as companies no longer need to maintain their own data center [Jon+19]. Instead, they can provision virtually unlimited compute, network, and storage within minutes. This removes the upfront capital investment required to build and maintain a data center and replaces it with a flexible pay-as-you-go model. Traditionally, cloud computing is realized in the form of virtual machines that can be created and deleted within minutes, which enables companies to flexibly adapt the available resources to the current demand. Due to these benefits, cloud computing is becoming widely adopted as 81% of enterprises are using cloud computing according to the 2020 IDG Cloud Computing Survey[IDG20] and Gartner reports that cloud spending increased by 40.7% in 2020 from $45.7 billion to $64.3 billion [Gar20].

To bring cloud computing even closer to the promise of computing as a utility, Amazon Web Services introduced AWS Lambda, the first FaaS platform, in 2014 [AWS14]. It abolishes the notion of virtual machines and instead allows developers to execute code on-demand in response to events with continuous scaling while having to pay only for the time used with sub-second metering. Recognizing the potential of FaaS, other cloud providers such as Microsoft Azure, Google Cloud, and IBM Cloud released their own FaaS platforms in 2016 [Eis+20c]. FaaS is rapidly adopted by cloud users with Allied Market Research estimating that the global FaaS market generated $3 billion in 2018 and will reach $24 billion by 2026 [Res20].

In recent years, cloud providers have further introduced many managed services, which are also known as Backend-as-a-Service (BaaS), for databases, messaging buses, and storage that also abstract the notion of virtual machines in favor of a pay-per-use model. Applications composed of these managed services and multiple FaaS functions have become widely known as *serverless applications* and their underlying computational model as *serverless computing*. Kounev et al. define serverless computing as a cloud computing paradigm offering a high-level application programming model, based on utilization-based billing, that allows one to develop and deploy cloud applications without allocating and managing virtualized servers and resources or being concerned about other operational aspects with utilization-based billing [Kou+21]. This definition is quite broad to encompass future developments in this area, such as fully managed container platforms, also known as Container-as-a-Service (CaaS). However, for the purpose of this thesis, we will consider serverless

computing as the combination of FaaS and BaaS, as this reflects the current state of the practice.

In 2019, 40% of companies reported adoption of serverless computing according to a survey by O'Reilly [ORe19] and is steadily increasing as DataDog finds that AWS lambda invocations have increased by 350% from 2019 to 2021 [Dat21]. MarketsAndMarkets Analysis estimates that the serverless computing market size will grow from an estimated $7.6 billion in 2020 to $21.1 billion in 2021 [Mar20]. Overall, serverless computing is emerging as the potential next step in the evolution of cloud computing.

1.1 Problem Statement

Serverless computing is quickly growing in adoption both in industry and in academia. However, reminiscent of many other emerging technologies, there are some growing pains associated with this rapid growth. One of the most pressing points is the lack of information around many serverless computing topics, as the community is in disagreement on many properties, interactions, and tradeoffs of serverless applications and platforms [Eyk+19; Eis+21c]. This could present a major roadblock in the adoption of serverless computing, as a survey by O'Reilly already reports that the leading concern of companies that have not yet adopted serverless computing is the fear of the unknown [ORe19].

This is especially true for the performance of serverless applications and platforms as much existing performance engineering knowledge and many approaches are not directly transferable to serverless computing. For example, it is currently unclear what typical serverless applications look like, what type of performance they can offer, what their limitations are, or how stable their performance is. Without this knowledge, it is challenging to make an informed decision if serverless computing is well suited for a use case. Developers are often unaware which configuration options and design decisions for serverless applications exist, what their impact on performance is, or how they should make these design and configuration decisions. This results in many poorly designed serverless applications in practice which in turn can quickly undermine the public perception of serverless computing. Further, there are also many open questions regarding the design of serverless platforms, such as the impact of different scheduling approaches, placement algorithms, resource provisioning strategies, virtualization techniques, worker deprovisioning timings, or runtime environments. The lack of an in-depth understanding of the performance implications and tradeoffs of these design decisions can severely hinder the advancement of serverless platforms. Overall, there are many open questions regarding the performance of serverless applications and platforms, which threaten to slow down the adoption of serverless computing.
We consider the following to be among the most pressing challenges:

Challenge 1: Disagreement on the characteristics of serverless applications Currently, there exist only few, scattered, and often conflicting reports on what serverless applications look like, when they are well suited, and what the best practices for their implementation are. Therefore, developers, academics, and managers often have conflicting views on the common characteristics of serverless applications, which hinders their adoption.

Challenge 2: Unknown performance variability of serverless applications Developers have no insight into the resource environment that their application is running in. Therefore, the performance of serverless applications could theoretically change at any time if the provider makes changes to the underlying hardware or software stack. However, there is currently no data on how often this occurs and how large the resulting performance variability is.

Challenge 3: Lack of automated resource sizing of serverless functions Developers of serverless functions are still in charge of resource sizing, that is, selecting how much resources (CPU, memory, network, disk I/O, etc.) are allocated to each worker instance. However, selecting the optimal size of serverless functions is quite challenging, so developers often neglect it despite its significant cost and performance implications.

Challenge 4: Difficulty of estimating the cost of serverless workflows Serverless workflows facilitate the orchestration of multiple serverless functions for complex tasks. However, the pay-per-use model and the delayed nontransparent reporting by cloud providers make it challenging for developers to estimate the expected monetary cost of serverless workflows, which prevents informed business decisions.

Challenge 5: Limited understanding of design tradeoffs for serverless platforms Platform providers need to understand the impact of different architectural and algorithmic design decisions on the performance properties of the platform. The high cost and risk associated with evaluating the performance of alternative designs in practice hinders the advancement of serverless platforms.

Therefore, in this doctoral thesis, we introduce techniques to understand, optimize, and analyze the performance of serverless applications and platforms in order to address these challenges.

1.2 Shortcomings of the State-of-the-Art

Only few, scattered, and often conflicting reports address the characteristics of serverless applications. For example, although some of them claim significant cost savings by switching to serverless applications [AC17a; Lev20], others identify scenarios in

which higher costs are incurred compared to traditional hosting [Eiv17]. Similarly, although reports of successful serverless applications for data-intensive applications exist [Wit+20; Cre+19], other reports claim that serverless computing is not well suited for data-intensive applications [Hel+18]. For serverless computing, existing research has focused on serverless platforms and their performance properties [SL20]. Pioneering studies about the features, architecture, and performance properties of these platforms [Eyk+19; BA18; Fig+18; LSF18; Llo+18; Wan+18a] do not study systematic collections of applications. The few existing empirical studies that do not focus on platform properties do not consider application characteristics either [Sha+20; Lei+19]. The only existing collection of serverless applications is by Castro et al. [Cas+19], which introduces ten exemplary applications. Therefore, Challenge 1 is not yet addressed, as systematic studies about serverless applications are still missing.

The performance variability of virtual machines in cloud environments has been studied extensively [IYE11; SDQ10; LSL19]. However, many serverless platforms are not deployed on traditional virtual machines [Aga+20]. Many existing works on the performance evaluation of serverless platforms determine the performance characteristics of such platforms but do not investigate the stability of performance measurements [Eyk+19; BA18; Fig+18; LSF18]. Some existing studies also investigate the performance variability of serverless platforms and find that the underlying CPU model and the low performance isolation between co-located workers are major sources of performance variability [Llo+18; Wan+18a; LSF18]. However, all of these studies consider only the performance variation between repeated executions and do not investigate the performance variability over longer periods of time. When Wang et al. repeated a subset of their measurements after six months, they found significant changes in the platform behavior [Wan+18a]. This shows that the performance of serverless applications can change over time, but the frequency, size, and impact of the performance variability over longer periods of time is still unknown, which means that Challenge 2 is not yet addressed.

Considering the memory size optimization of serverless functions, there are many articles describing the impact of memory size on serverless functions, which highlight the complexity and importance of selecting the optimal memory size [Zha+19; Wan+18b; Fig+18; BA18; Str18; SL20]. Most existing approaches for the cost optimization of serverless functions either do not consider memory size or consider its impact as a user-provided parameter [KK17; Gun+19; Elg18; Boz+17]. The current best practice in industry is to measure the execution time of each memory size using performance tests followed by a manual analysis of the results [Cas20a]. There have been approaches proposed to reduce the number of required performance measurements by measuring a subset of memory sizes and interpolating the execution time for the remaining memory sizes [Akh+20; Ali+20]. All existing approaches to optimize the memory size of serverless functions require automated performance tests, which are time-consuming to implement and maintain [JH15; Bez+19]. Further, the

requirement for active performance tests makes these techniques inapplicable for cloud providers. This shows that Challenge 3 is not yet addressed.

Many empirical studies analyze the performance and cost of industry-standard workflow orchestrators that showcase significant cost differences between the available options [Lop+18; MAB21; WL21; Bar+19]. Academic researchers have further proposed many formalisms and execution frameworks for serverless workflows that aim to address the shortcomings of the first generation of serverless workflow orchestrators [Skl+19; Joh+19; Bur+21; Lop+20; Zha+20]. Existing approaches for the cost estimation of serverless functions and workflows require the user to estimate the function response time with a single mean value [Boz+17; Elg18], which does not consider the impact of input parameters on the function execution time. Queueing theory-based models can predict the impact of input parameters on the performance of traditional systems [Bon+05; Ack+18], but they are inapplicable for serverless solutions, as they require knowledge about the underlying resource landscape and deployment. To summarize, Challenge 4 is not yet addressed, as there is currently no approach that enables the accurate cost prediction of different configurations of a serverless workflow.

As to the analysis of design tradeoffs for serverless platforms, the currently popular approach is the implementation of research prototypes that implement a specific combination of tradeoffs combined with an experimental evaluation to analyze its performance properties. Examples of such prototypes include SOCK [Oak+18], SEUSS [Cad+20], Atoll [Sin+21], EMARS [SJ18], and SAND [Akk+18]. However, the implementation and evaluation of research prototypes is quite time-intensive and only enables the exploration of a specific configuration or a set of configurations. In traditional software systems, white-box performance modeling is a common technique to evaluate the impact of design tradeoffs [Reu+16; Ale+09; Kou+17]. White-box performance models capture the system architecture, the control flow within and between the software components, the deployment of components on physical resources, the performance properties of individual components, and the expected workload. These models can then be simulated to predict performance indices such as response time, throughput, or hardware utilization [Bro+15]. Unfortunately, white-box performance modeling is currently challenging to apply to serverless platforms due to two well-documented shortcomings of white-box performance modeling: a) white-box performance modeling requires the explicit modeling of parametric dependencies [Koz08; Ham09; Bon+05], that is modeling the impact of configuration parameters and user input on the system behavior, which is not feasible for a platform running thousands of different functions [Sha+20], and b) the simulation time of a white-box performance model grows exponentially with the number of components, which makes the analysis of large scale systems, such as serverless platform, infeasible [Nam+16; WFP07; Koz10]. White-box performance modeling techniques could be well suited to address Challenge 5, but they are currently not widely applied for serverless platforms due to their shortcomings.

To summarize, none of the challenges we introduce in Section 1.1 are addressed by the current state-of-the-art.

1.3 Goals and Research Questions

In the previous sections, we introduced five challenges around the performance engineering of serverless applications and platforms that could potentially slow down the adoption of serverless computing and the shortcomings of the state-of-the-art regarding these challenges. In the following, we present the five main goals of this thesis, with each goal designed to address one of the previously introduced challenges. For each goal, we also list a set of Research Questions (RQs) that need to be answered in order to address the respective goals.

Goal I: *Provide quantitative data on the common characteristics of modern serverless applications.*

- **RQ I.1**: *What are common characteristics of current serverless applications?*
- **RQ I.2**: *Is there a community consensus on the common characteristics of serverless applications?*

The first goal aims to address Challenge 1, the current disagreement on the characteristics (size, platform, programming language, etc.) of serverless applications that exists in the public dialogue. Resolving this disagreement requires well-founded quantitative data on the common characteristics of serverless applications. Towards this goal, the first research question concerns the generation of primary data on the common characteristics of serverless applications, while the second research question aims to substantiate the primary data by comparing it with data from existing studies.

Goal II: *Quantify the performance variability that serverless applications experience.*

- **RQ II.1**: *How much performance variability do common serverless applications experience?*
- **RQ II.2**: *Does the performance of serverless applications change over time?*

The second goal aims to address Challenge 2 by quantifying the performance variability that serverless applications experience due to variation in the underlying platforms. As discussed earlier, this is currently an unknown factor that can dissuade developers from adopting the serverless paradigm. The first research question aims at the quantification of the baseline performance variability of serverless applications, while the second research question is about quantifying the performance variability of serverless applications over longer time periods.

Goal III: *Develop an automated method to optimize the size of serverless functions.*

- **RQ III.1**: *How can a dataset on the impact of memory size for a vast number of functions be generated?*
- **RQ III.2**: *How can one predict the optimal size of serverless functions based on passive monitoring data?*

The third goal aims to address Challenge 3, the automated optimization of the resource sizing for serverless applications without active empirical measurements, so that cloud providers can automatically manage the function configuration instead of developers having to deal with it. The first research question concerns the generation of a large training dataset on the impact of memory size, without having access to a vast amount of serverless functions; the second research question aims at the automated size optimization of serverless functions based on passive monitoring data.

Goal IV: *Provide a technique to estimate the costs of serverless workflows.*

- **RQ IV.1**: *How can the execution time distribution of a serverless function be predicted based on its input parameters?*
- **RQ IV.2**: *Can the impact of restructuring a serverless workflow on its cost be predicted?*

The fourth goal aims to address Challenge 4 by predicting the billed costs of serverless workflows based on predictions for the performance of each serverless function within the workflow. The first research question concerns the prediction of the execution time distribution of individual serverless functions based on their input parameters in order to derive accurate cost predictions on a per-function level. The second research question aims to use these predictions to assess the impact of different configurations of a serverless workflow on its cost.

Goal V: *Provide an approach for the white-box performance modeling and simulation of serverless platforms.*

- **RQ V.1**: *How can the time required to simulate large systems such as serverless platforms be reduced?*
- **RQ V.2**: *How can relationships between parameters observed at runtime be utilized in white-box performance models?*

The fifth goal aims to address Challenge 5 by enabling the use of white-box performance modeling to analyze the performance properties of serverless platforms. The first research question aims to enable faster simulation times for white-box

performance models in order to support the analysis of large systems such as serverless platforms. The second research question targets the modeling of empirically determined relationships between model parameters, as the explicit modeling of parametric dependencies at design time is infeasible for large systems such as serverless platforms.

1.4 Contribution Summary

In the following, we summarize the three main contributions of this thesis, which address the challenges identified in Section 1.1 and the research goals defined in Section 1.3.

Contribution 1: Evaluation of the Characteristics and Performance of Serverless Applications Our first contribution aims to address Challenge 1 and Challenge 2 by improving the understanding of serverless applications on the basis of two empirical studies. The first study builds on resources created by the community to systematically collect a total of 89 serverless applications from open-source projects, academic literature, industrial white papers, and scientific computing projects. These applications are then characterized through a systematic and comprehensive pair-reviewing process with regard to 16 characteristics, such as execution pattern, workflow coordination, use of external services, and motivation for adopting the serverless paradigm. We complement this with a literature search, finding ten, mostly industrial, web surveys and datasets on the characteristics of serverless applications; we compare the results of these studies to our results with the aim to identify characteristics for which there is a consensus among multiple studies as well as points of disagreement. The second study is an exploratory case study on the stability of performance measurements of serverless applications. Unlike existing work, which mostly relies on microbenchmarks and single-function applications, we use a representative, production-grade serverless application for our case study. Using this application, we conduct two sets of performance measurements: (1) multiple repetitions of performance measurements under varying configurations to investigate the baseline performance variability, and (2) three daily measurements for ten months to create a longitudinal dataset and investigate the stability of performance tests over time.

As a result of these two studies, we find that the most commonly reported reasons for the adoption of serverless computing are cost savings for irregular or bursty workloads, avoidance of operational concerns, built-in scalability, and increased speed of development. Typical use cases for serverless applications include short-running tasks with low data volume and bursty workloads, but we also frequently came across latency-critical, high-volume core functionality as serverless applications. Further, we find that serverless applications are mostly implemented on AWS, in either Python or JavaScript, and use managed services. As to the performance

variability, we find that the performance variability of measurements conducted at the same time is comparable to the performance variability observed in traditional microservice-based or monolithic systems. However, we also find that there are short-term performance fluctuations where the performance changes from day to day. Additionally, we observe multiple long-term performance changes where undisclosed, provider-side changes permanently alter the performance of serverless applications.

These studies present the largest collection of serverless applications to date, by a factor of 8.9x over the next largest collection [Cas+19]. They provide the first systematic and comprehensive characterization of serverless applications, the first analysis of community consensus on the characteristics of serverless applications, as well as the first characterization of performance variability of serverless applications over longer periods of time. Therefore, this contribution addresses **Goal I** and **Goal II** by improving the understanding of serverless applications. The individual parts of this contribution have been published in a Standard Performance Evaluation Corporation Research Group (SPEC RG) technical report [Eis+20c], an IEEE Software article [Eis+21b], an IEEE Transactions on Software Engineering (TSE) article [Eis+21c], and an article in the Journal of Systems and Software (JSS) [Eis+22].

Contribution 2: Automating Operational Tasks of Serverless Applications Our second contribution introduces two approaches to address Challenge 3 and Challenge 4 by automating the operational tasks associated with deploying serverless applications. To address the task of resource sizing, that is, selecting how much CPU, memory, I/O bandwidth, etc. are allocated to a worker instance, we introduce an approach to predict the optimal memory size of serverless functions based on monitoring data for a single memory size. To achieve this goal, we first implement a serverless function generator capable of generating a large number of synthetic serverless functions by combining representative function segments. Next, we measure the execution time and resource consumption metrics of 2 000 synthetic functions for six different memory sizes on a public cloud. Finally, we construct a multi-target regression model to predict the execution time of a serverless function for previously unseen memory sizes based on the execution time and resource consumption metrics for a single memory size. The second operational task that we address in this contribution is the cost optimization of serverless workflows. First, we apply machine learning to predict the response time and output parameter distributions for individual serverless functions. Next, we show how Mixture Density Networks (MDNs) can be used to accurately predict the response time and output parameter distributions of serverless functions. These individual function models are integrated into a workflow model that describes the parameter relationships within the workflow. Finally, a Monte-Carlo simulation traverses the workflow

model and samples distributions from the individual function models to derive cost predictions for serverless workflows.

In our evaluation, our memory size optimization model—which was trained on data from synthetic functions—successfully predicts the execution time of other memory sizes based on monitoring data from a single memory size with an average prediction error of 15.3% for four realistic serverless applications. It selects the optimal memory size for 79.0% and the second-best memory size for 12.3% of the serverless functions. Using the memory sizes selected by our approach results in an average speedup of 39.7% while simultaneously reducing average costs by 2.6%. Our cost prediction approach predicts the response time distribution and the distribution of the output parameters of five representative Google Cloud Functions with a mean accuracy of 96.1% in a case study in the context of audio transcription workflows. For two workflows composed of these functions, our approach achieves a mean workflow cost prediction accuracy of 96.2%.

The approaches introduced in this contribution enable cloud providers to provide memory size recommendations for developers as well as to automatically optimize the cost of serverless workflows based on accurate, context-sensitive cost predictions. Therefore, this contribution addresses **Goal III** and **Goal IV** by automating the operational tasks associated with serverless applications. It has been published in a full research paper at the 13th ACM/SPEC International Conference on Performance Engineering (ICPE 2022) [Eis+20b] and a full research paper at the 22nd ACM/IFIP International Middleware Conference (MIDDLEWARE 2022) [Eis+21a], which received the Best Student Paper Award.

Contribution 3: Enabling White-Box Performance Modeling and Simulation of Serverless Platforms Our third contribution aims to address Challenge 5 by introducing two approaches that extend white-box performance models to make them applicable for the analysis of serverless platforms. In this contribution, we build upon and extend the Descartes Modeling Language (DML), a representative white-box performance model tailored towards resource management at runtime. The DML has been developed and extended in the context of multiple PhD theses [Bro14; Hub14; Wal19b]. However, the approaches presented in this contribution can be transferred to any architectural software performance model which uses a component notation [BKR09; WW04; His+02; Göb+04].

The first approach is directed at speeding up the simulation time required to solve white-box performance models. This is necessary as the simulation time increases with system size making the analysis of large-scale systems, such as serverless platforms infeasible. We introduce a generic modeling approach that enables a parallel and integrated description of subsystems as both fast-to-solve black-box performance models and as traditional queueing models. We provide a transformation of the integrated queueing/statistical model to Queueing Petri Nets (QPNs) and extend an existing discrete event simulation solver for QPNs to support black-box performance

models. The second approach addresses the issue that white-box performance modeling does not support the integration of empirically observed relationships between model parameters. We propose a novel approach to modeling empirical parametric dependencies in architectural performance models. To derive performance prediction, a dependency resolution algorithm transforms the empirical information from the model into a directed graph and resolves this graph to derive a fully parameterized model.

In our evaluation, we apply the proposed approach for the simulation of white-box performance models to a distributed, component-based system of medium size. Our experiments show that the approach maintains sufficient prediction accuracy and achieves speedups of up to 94.8%. In two case studies in the context of a media store, our approach for the modeling and solution of empirical parametric dependencies achieves a mean prediction error for utilization and response time of less than 5% and 10%, respectively.

The approaches proposed in this contribution address two limitations of existing white-box performance modeling techniques in order to enable their application for the analysis of design tradeoffs of serverless platforms. Therefore, this contribution addresses **Goal V** by extending white-box performance models for the analysis of serverless platforms. The results have been published in a full research paper at the 15th ACM/SPEC International Conference on Software Architecture (ICSA 2018) [Eis+18] and a full research paper at the 16th ACM/SPEC International Conference on Software Architecture (ICSA 2019) [Eis+19].

1.5 Thesis Outline

In the following, we describe the remaining structure of this thesis. First, Part I introduces the fundamental notions and concepts related to this thesis and discusses the state-of-the-art in detail. Chapter 2introduces the background on serverless computing including Function-as-a-Service, Backend-as-a-Service, and pricing models as well as the performance engineering concepts of performance testing, white-box performance modeling, and black-box performance modeling. Furthermore, we discuss the current state-of-the-art regarding the challenges this thesis aims to address in Chapter 3.

In Part II, we present the main contributions of this thesis. In particular, Section 4 discusses the two empirical studies we conducted to improve the understanding of serverless applications and performance variance, Section 5 presents our approaches for the automated optimization of serverless function sizes and workflow configuration, and Section 6 introduces our approaches to close two of the gaps in white-box performance modeling that make it inapplicable for the analysis of serverless platforms.

Part III then discusses the results of our studies and evaluates the proposed approaches. First, Section 7 discusses the results of our empirical studies on the

characteristics and performance variability of serverless applications. Then, Section 8 evaluates the function size optimization on four realistic serverless applications and the workflow cost optimization on two speech transcription workflows. Lastly, Section 9 evaluates the proposed approach to speed up the simulation time of white-box performance models on a large synthetic application and evaluates our approach to capture empirical relationships in white-box performance models on two examples in the context of a media store.

Finally, Part IV concludes this thesis. In Section 10, we summarize the contributions and Section 11 describes the potential future work.

Part I

Fundamentals and Related Work

Chapter 2

Fundamentals

In this chapter, we outline the fundamental concepts that lay the foundation for our research. In Section 2.1, we define the term serverless computing and introduce the concepts of FaaS and BaaS. Finally, Section 2.2 introduces common performance engineering concepts.

2.1 Serverless Computing

In the following, we provide a definition for serverless computing, the main subject of this thesis, and discuss the related terms Function-as-a-Service and Backend-as-a-Service. Further, we introduce the concepts behind Function-as-a-Service platforms, the serverless pricing model, and function sizing.

2.1.1 Definition

In a recent Dagstuhl seminar with over fifty serverless computing experts from industry and academia, we extensively debated a definition for serverless computing [Kou+21]. After multiple days of intense discussion, we agreed on the following definition:

Definition: *Serverless computing* is a cloud computing paradigm offering a high-level application programming model that allows one to develop and deploy cloud applications without allocating and managing virtualized servers and resources or being concerned about other operational aspects. The responsibility for operational aspects, such as fault tolerance or the elastic scaling of computing, storage, and communication resources to match varying application demands, is offloaded to the cloud provider. Providers apply utilization-based billing: they charge cloud users in proportion to the resources that applications actually consume from the cloud infrastructure, such as computing time, memory, and storage space.

This definition focuses around the concepts of *NoOps* and *Utilization-based Billing*, compared to other definitions that focus on properties such as stateless, granular billing, and event-driven architectures. Today, Function-as-a-Service (see Section 2.1.2) and Backend-as-a-Service (see Section 2.1.3) are the most common im-

plementations of serverless computing, which are both covered by the new, extended definition. As the definition focuses on conceptual properties, instead of implementation-based limitations, such as short execution time, this definition covers the growing set of technologies and evolving programming models that taken together will provide the basis for next-generation serverless platforms and applications.

2.1.2 Function-as-a-Service

While some startups such as PiCloud announced Function-as-a-Service platforms starting from 2010[1], mainstream adoption began in 2014, when Amazon Web Services (AWS) announced AWS Lambda, the first Function-as-a-Service platform from a major cloud provider[2]. This started a race from the other cloud providers to develop their own Function-as-a-Service solution, with IBM functions[3], Google Cloud Functions[4], Azure Functions[5] all released in 2016. There are also numerous open-source Function-as-a-Service platforms that are actively being developed [Li+19]. Allied Market Research estimates that the global Function-as-a-Service market generated $3.01 billion in 2018 and will reach $24.00 billion by 2026[6], which shows that Function-as-a-Service is rapidly adopted in industry.

With Function-as-a-Service (also often known as serverless functions), developers write business logic as isolated functions, specify the memory and CPU resources that the function should be executed with, and define triggers for the execution of the business logic. Triggers can either be HTTP requests or cloud events such as a new message in a queue, a new database entry, a file upload, or an event on an event bus. The developer provides the code for the serverless functions and the triggers; then the cloud provider guarantees that the code is executed whenever a trigger occurs, independent of the number of parallel executions. In contrast to classical Infrastructure-as-a-Service (IaaS) platforms, developers are not billed for the time resources are *allocated* but rather for the time resources are actively *used*. So under this pay-per-use model, developers are billed based on the consumed CPU time, so function execution time multiplied by the specified CPU time.

Serverless functions promise seamless scaling of arbitrary code. In order to do so, each function needs to be *ephemeral*, *stateless*, and executed in a *provider-managed runtime environment*. Serverless functions are ephemeral, which means that developers can not assume that the execution environment stays available after a request is processed. The ephemeral nature of serverless functions also enforces them to be stateless as for example, values saved to static variables or initialized connections

[1] https://techcrunch.com/picloud-launches-serverless-computing-platform-to-the-public/
[2] https://aws.amazon.com//2014/11/13/introducing-aws-lambda/
[3] http://www-03.ibm.com/press/us/en/pressrelease/49158.wss
[4] https://venturebeat.com/2016/02/09/google-launched-its-answer-to-aws-lambda/
[5] https://azure.microsoft.com/announcing-general-availability-of-azure-functions/
[6] https://www.alliedmarketresearch.com/function-as-a-service-market-A06072

might no longer be available when the function is executed a second time. Further, developers need to select a provider-managed runtime environment, such as for example Java 8, Java 17, or Python 3. Initially, this was a quite large constraint as AWS Lambda supported only JavaScript upon release, but now the long-term support (LTS) versions of most popular languages are available.

For the most part, serverless functions fulfill their promise of seamless scaling, however, they are suffering from the performance challenge of so-called *cold starts*. If no function instances are available to handle an incoming request, a new function instance is started to handle the request. These cold starts could initially take multiple seconds, but cloud providers are continuously working to reduce this startup time. Current studies indicate that a typical cold start only incurs a three-digit millisecond overhead today, for an in-depth analysis of cold start times, we refer to an article series by Mikhail Shilkov[7].

Initially, serverless functions were used for inherently stateless applications, such as chat bots [Yan+16] or single-page applications [Rad16], but have since been shown to support complex use cases such as neural network training [Fen+18] and serving [IMS18], seismic imaging [Wit+20], sequence comparison [Niu+19a], task offloading [Fou+19], or social media platforms [AC17a].

2.1.3 Backend-as-a-Service

There is currently no widely accepted definition for Backend-as-a-Service and there is quite some ambiguity in how the term is currently used. It is also often used interchangeably with Mobil-Backend-as-a-Service, that is, managed backends for mobile applications. For the context of this thesis, we consider Backend-as-a-Service as the combination of *managed services* and *third-party APIs*.

Managed services implement application primitives, such as message brokers (queues, pub/sub, etc.), request routing, file storage, databases, secret stores, or key-value stores. Developers can instantiate and configure these services via a management console, a CLI, or more commonly as Infrastructure-as-Code and can then interact with these services from their code using software development kits (SDKs) offered by the cloud provider. The cloud provider is responsible for the availability, reliability, scalability, performance, deployment, and maintenance of these services. Managed services implement different forms of the pay-per-use pricing model, depending on the nature of the service. For example, a queueing service might charge per message delivered by the queue, whereas blob storage might charge per megabyte of upload and download, as well as a storage charge per megabyte of data stored per month. While the complexity and exact nature of the pricing models differ between managed services, they all charge for *usage* of the service rather than for *allocation* of resources as is the case in IaaS.

[7] https://mikhail.io/serverless/coldstarts/

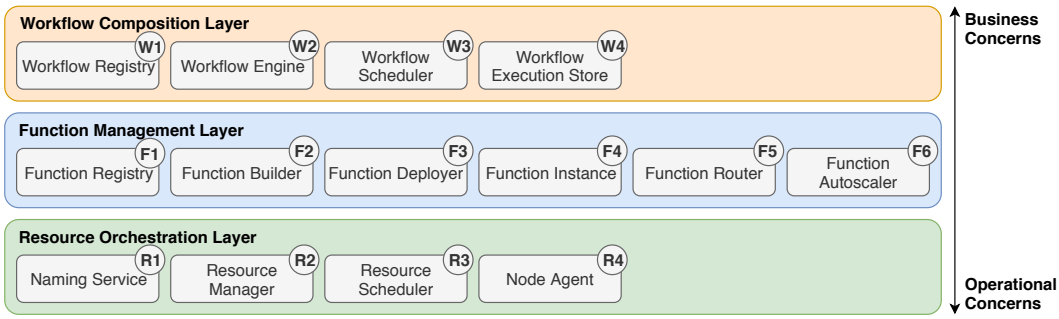


Figure 2.1: The SPEC RG reference architecture for FaaS platforms [Eyk+19].

Third-party APIs differ from managed services in two areas: a) they are offered by a third party instead of the cloud provider, and b) they implement specific application logic, rather than application primitives. A great example for a third-party API is Stripe[8], which is a service that offers managed payment workflows, which can be integrated into an application in the form of API calls. Other examples of services offered by third-party APIs include chats, ticketing, surveys, user management, and mailing lists. Generally, third-party APIs implement application fragments, that can function on their own whereas managed services implement functionality that can be used to build applications. However, this line can get blurred quickly, for example, both AWS and Auth0[9] offer a service for the management and authentication of users and it is unclear whether to count this as a managed service or a third-party API. Therefore, for the remainder of this thesis, we will use the term managed services interchangeably with Backend-as-a-Service to also include any managed third-party APIs.

The use of managed services offers many benefits to developers. First, it offers the classic benefits of software reuse in the form of libraries, such as improved development speed and lower susceptibility to bugs. Further, as either a cloud provider or a third-party vendor take care of the runtime management of the applications, they likely offer better performance and higher availability as they are already operating these services for years. Finally, it reduces operational overhead as it reduces operational overhead since developers do not need to operate these services.

2.1.4 Function-as-a-Service Platforms

Despite the rapidly increasing adoption of serverless computing, there is no community-wide consensus on the fundamental architecture building blocks of Function-as-a-Service platforms. In collaboration with the SPEC RG Cloud group[10], we

[8] https://stripe.com/
[9] https://auth0.com/
[10] https://research.spec.org/working-groups/rg-cloud.html

worked on distilling a reference architecture for Function-as-a-Service platforms based on a systematic analysis of 47 diverse open-source and closed-source platforms, ranging from workflow composition engines (e.g., Fission Workflows and AWS Step Functions), to single-function engines (e.g., Apache OpenWhisk and AWS Lambda), to cloud resource managers (e.g., Kubernetes) [Eyk+19]. The resulting reference architecture is depicted in Figure 2.1. In the following, we discuss the *resource orchestration layer*, *function management layer*, and *workflow composition layer*, and their building blocks in detail.

2.1.4.1 Resource Orchestration Layer

At the lowest level in the reference architecture, the *resource orchestration layer* is responsible for the management of physical resources of a cluster of machines. The components in this layer manage the operational lifecycle of the containers or virtual machines, which are consolidated on physical resources. We separate this generic resource layer from the FaaS-specific layers, to indicate where the FaaS platforms fit into the existing cloud infrastructure, and to capture the common approach of FaaS platforms to delegate resource management to more mature systems. The *resource orchestration layer* consists of:

1. **Naming Service**: provides cluster-wide unique and consistent naming to resources. This allows components to identify each other.
2. **Resource Manager**: manages the available resources of cluster-nodes through node agents, ensuring that the state of the resources conforms with the desired state.
3. **Resource Scheduler**: determines which actions are needed to ensure that the current state of the resources converges towards the desired state. The scheduler decides for each job, on which resources it should be deployed.
4. **Node Agent**: is deployed on each node in the cluster. It monitors the local resources, informs the resource manager, and executes instructions it receives from the *resource manager*.

2.1.4.2 Function Management Layer

The *function management layer* contains the core components responsible for the operational lifecycle of individual FaaS functions: deploying function instances, executing functions triggered by events, and elastically scaling functions. Whereas the *resource orchestration layer* is concerned with the management of arbitrary resources, the *function management layer* manages arbitrary functions. In this layer, the components rely on the lower-level layer for the correct management of the resources. At its core, the *function management layer* consists of:

1. **Function Registry**: is a function repository that is often further split into a *function metadata store*, for low-latency look-ups of function metadata, and a *function store* containing the binaries of the function (the function code).

2. **Function Builder**: turns function sources into deployable functions. Functions typically have to undergo compilation, packaging, etc, before they are stored in the *function registry* or deployed by the *function deployer*.

3. **Function Deployer**: combines the configuration stored in the *function registry*, the parameters supplied by the requester, and other factors into a decision of how the function should be deployed. The deployment of the *function instance* itself is delegated to the *resource orchestration layer*.

4. **Function Instance**: is a self-contained worker—typically a container—capable of handling function executions. For scalability, a function can have multiple, concurrent *function instances*.

5. **Function Router**: routes incoming requests or events to the correct *function instance*. If no *function instance* is available, the *function router* queues the events to await the deployment of new instances.

6. **Function Autoscaler**: monitors the demand and supply of resources, and elastically scales the number of *function instances*—adding or removing instances as needed.

2.1.4.3 Workflow Composition Layer

Modern applications require the orchestration of multiple functions and the management of the inter-function state, which is the responsibility of the higher-level *workflow composition layer*. As with the *resource orchestration layer*, we rely on the extensive existing work on workflow management systems for the components of this layer. The difference with original workflow systems is that serverless workflows are smaller, executed more frequently, and have more demanding performance requirements. The *workflow composition layer* consists of:

1. **Workflow Registry**: serves as a repository of workflows. To be admitted to this registry, a workflow typically requires validation and compilation of its individual tasks.

2. **Workflow Engine**: is responsible for monitoring workflow executions. It takes the appropriate action based on decisions from the *workflow scheduler*, such as triggering the execution of functions in the workflow whose predecessors have completed.

Provider	AWS	Azure	Google	IBM
Invocation cost	$\$0.2 * 10^{-6}$	$\$0.2 * 10^{-6}$	$\$0.2 * 10^{-6}$	-
Memory [GB-s]	$1.7 * 10^{-5}$	$1.6 * 10^{-5}$	$2.5 * 10^{-6}$	$1.7 * 10^{-5}$
CPU [GHz-s]	-	-	$1.0 * 10^{-5}$	-
Billing interval	1ms*	1ms*	100ms	100ms

Table 2.1: Comparison of the pricing models of popular serverless function platforms. *changed in 2021 from 100ms.

3. **Workflow Scheduler**: decides which functions to execute when. It makes these decisions based on a number of factors, including the current state of the workflow execution and historical data.

4. **Workflow Execution Storage**: ensures the persistence of data of workflow executions. To ensure reliable workflow executions, a database holds the state of workflow executions.

2.1.5 Serverless Pricing Models

One of the key characteristics of serverless computing compared to traditional cloud computing models is the billing per *use* instead of *allocated* resources. For serverless functions, the billing scheme between different cloud providers differs slightly, but generally implements the same concepts: 1) a flat invocation cost that is billed per function execution, 2) a charge for memory size of the function multiplied by the execution time, 3) a charge for the CPU size of the function multiplied by the execution time, and 4) the execution time is rounded up to the nearest 100ms. Table 2.1 gives an overview of the exact pricing model for serverless functions by AWS, Azure, Google, and IBM at the time of writing. All providers except for IBM charge a flat invocation cost of $\$0.2 * 10^{-6}$. At AWS, Azure, and IBM users can select function sizes only by memory size, and the CPU size is scaled correspondingly, so they only charge for memory time. Google, on the other hand, allows users to define the memory and CPU size of a function independently, therefore they have separate billing for the consumed memory and CPU time. Typically, the billed memory/CPU time is rounded *up* to the nearest 100ms interval, however, Azure uses per millisecond billing with a minimum of 100ms. Recently AWS also switched to per millisecond billing [11]. As an example, a function that runs on AWS for three seconds with a memory size of 512 MB would cost:

$$3s \cdot 0.5GB \cdot 0.00001667\$ + 0.0000002\$ = 0.0000252\$$$

[11] https://aws.amazon.com/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/

where 0.00001667$ is the price per consumed GB-s and 0.0000002$ is the static overhead charge (0.7% of the total execution cost). For a more in-depth discussion of the impact of this pricing model, we refer to an article by Eivy et al. [Eiv17].

A direct comparison of the pricing models of managed services across providers is challenging as there often is no direct counterpart for every service [Yus+19]. Therefore, we will go over the popular services from AWS to exemplify the billing models behind managed services. Amazon SQS, a *managed queue*, has a comparatively simple pricing model with a flat charge per message posted to the queue. This amount of this flat charge depends on the volume of requests per month and differs between unordered and First In, First Out (FIFO) queues. For Amazon SNS, a *managed pub/sub* service, the billing model is more complex. First, there is again a charge per message that depends on if FIFO is required, however, every message is divided into chunks based on message size, that each counts as an individual message. Further, there are data transfer charges that only apply to some target destinations. Finally, there are additional charges for mobile push, email, and HTTP notifications. Amazon S3, a *managed blob storage*, offers four different six different storage tiers based on the required availability, durability, and access pattern. For each of these storage tiers, different rates apply for the following charges: GB/month of stored data, PUT/COPY/POST/LIST requests, other requests, data retrieval requests, GB of data retrieved, and data transfer to the Internet. Amazon DynamoDB, a *managed NoSQL database*, differentiates between provisioned capacity and on-demand billing (the serverless computing option). For the on-demand billing, developers are charged a flat fee for every read and write request as well as a monthly fee per stored GB. Additional charges are linked to features such as global tables, backups, acceleration, streams, and data transfer to some other services. These are the pricing models for just four out of the over 200 services AWS currently offers, which shows how the pricing of serverless applications that combine many of these services can get quite complicated.

2.1.6 Serverless Function Size

The main benefit of serverless functions compared to traditional compute solutions is that the cloud provider opaquely takes care of common resource management tasks such as resource provisioning, software deployment, or auto-scaling [AC17a; Eyk+19]. However, there is still one resource management task that cloud providers leave to developers: *resource sizing*. Resource sizing is the task of selecting how much CPU, memory, I/O bandwidth, etc. are allocated to a worker instance [Sin+20; AL19; Pir+15]. Most cloud providers implement the resource sizing of serverless functions as a configurable memory size, where other resources such as CPU, network, or I/O are scaled accordingly[12][13]. Selecting an appropriate resource size is essential

[12] https://aws.amazon.com/lambda/pricing/
[13] https://cloud.ibm.com/functions/learn/pricing

as it can often result in a faster execution at a lower cost. However, selecting an appropriate resource size is challenging. A recent survey revealed that 47% of the serverless functions in production use the default memory size setting, indicating that developers often neglect resource sizing[14].

The relationship between the memory size of a serverless function, the cost per function execution, and the function execution time is quite counter-intuitive. A common assumption is that a higher memory size results in a faster execution at a higher price, since the allocated CPU, I/O, network, etc. capacity scales linearly with the selected memory size. However, this is not the case due to the pricing scheme most cloud providers employ, where the cost of an execution is calculated based on the consumed GB-s of memory, that is, the execution time multiplied by memory size. Increasing the memory size increases the cost per second, but also decreases the execution time as more resources are allocated. As each function's execution time scales differently with additional resources, every function has a unique cost/performance tradeoff.

Figure 2.2 shows how the execution time and cost per execution vary for four different functions with different memory sizes based on data from [Cas20b]. The function *InvertMatrix* creates and inverts a random matrix. Here, we can see that increasing the memory size from 128MB to 256MB decreases the execution time by 49.6%, with only a 1% increase in cost. For larger memory sizes, the execution time still decreases almost linearly. The second function, *PrimeNumbers*, calculates the first million prime numbers a thousand times, which is another CPU-intensive task. Interestingly, the execution time of this function scales super-linearly with increased memory sizes up to 2048MB, which results in a 92.9% faster execution with simultaneously 13.3% reduced costs. Using a memory size of 3008MB further speeds up the execution time, but it increases the execution cost. The third function, *DynamoDB*, executes three queries against a DynamoDB table, which is a serverless database. Here, the execution time decreases roughly linearly from 128MB to 512MB, resulting in an 86.6% decreased execution time at a similar cost. However, further increasing the memory only slightly reduces the execution time while increasing costs by 587.5%. Lastly, the *API-Call* function calls an external API. Here, increasing the memory has minimal impact on the execution time and only increases the cost per execution.

We can conclude that: i) the impact of memory size configurations on execution time differs from function to function, ii) predicting the execution time for a memory size is challenging, as even two seemingly CPU-intensive and two network-intensive functions behave differently, and iii) selecting an appropriate memory size is important as it can drastically improve performance at a similar or reduced cost.

[14]https://www.datadoghq.com/state-of-serverless/

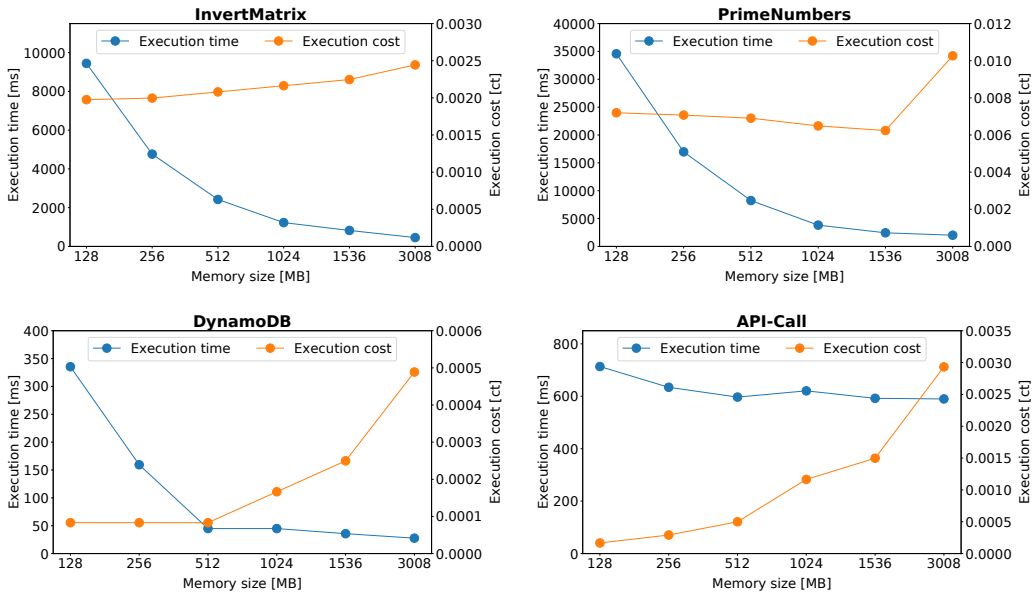


Figure 2.2: The mean execution time and cost for four serverless functions (adapted from [Cas20b]).

2.2 Performance Engineering

In this section, we introduce the fundamental performance engineering concepts related to this thesis. Section 2.2.1 covers performance testing, Sections 2.2.2 and 2.2.3 introduce white-box and black-box performance modeling respectively, and finally, Section 2.2.4 discusses automated performance optimization.

2.2.1 Performance Testing

Performance testing is the process of measuring and ascertaining a system's performance-related aspects (e.g., response time, resource utilization, and throughput) under a particular workload [JH15]. Performance testing helps to determine compliance with performance goals and requirements, identify bottlenecks in a system, and detect performance regressions. A typical performance testing process starts with designing the performance tests according to the performance requirements. These performance tests are then executed in a dedicated performance testing environment, while the system under test is continuously monitored to collect system runtime information including performance counters (e.g., response time and CPU utilization), the system's execution logs, and event traces. Finally, performance analysts analyze the results of the performance testing.

Conducting a performance test in the production environment may have a negative impact on the users of the production environment. Hence, performance tests are often conducted in test (or staging) environments. However, it is challenging to predict the production performance from a performance test on a smaller, less powerful testing environment. For example, there is a discrepancy between performance test results from a virtual and physical environment. Hence, the testing environment should ideally be equal to the production environment. In addition, a heterogeneous testing environment makes the analysis of performance testing results more challenging making a homogeneous testing environment more desirable.

In regards to the operational profile, that is, the user behavior, used in a performance test, there are two types of performance tests: API-focused performance tests and realistic performance tests. API-focused performance tests stress only a single API endpoint and quantify the performance of this endpoint in isolation. Realistic performance tests on the other hand aim to closely depict the behavior of real users. API-focused performance tests are easier to set up, maintain and interpret. However, realistic performance tests more closely mirror the expected performance in the production environment, as they correctly depict the overall system utilization, which often impacts the performance of individual endpoints.

During the execution of a software system, it often takes some time to reach its stable performance level under a given load. One typical example is that Java programs need some time for the Just-In-Time (JIT) compilation before applications achieve a steady state. During performance testing, the period before the software system reaches steady-state is commonly known as the warm-up period, and the period after that is considered as the steady-state period. There are many reasons for the warm-up period, such as filling up buffers or caches, performing JIT compilation, and absorbing temporary fluctuations in the system state. Since performance during the warm-up period may fluctuate, in practice, performance engineers often remove the duration of the unstable phase (i.e., warm-up period) of the performance test and only consider the steady-state period in the performance test results. The most intuitive way to determine the warm-up period is to simply remove a fixed duration of time from the beginning of the performance testing results. We refer to a review by Mahajan and Ingalls [MI04] for an overview of existing techniques to determine the warm-up period.

One of the common use cases for performance tests is performance regression testing, which aims to determine which code change caused a deterioration in the performance of a software product. To determine this, single or multiple performance tests are executed for each commit, and the results are analyzed to determine after which commit the performance degraded. This process is often implemented as part of a CI/CD pipeline. To avoid the need for manual analysis, techniques from the area of change point detection (determining when a significant change in a time series occurs) are used. For an overview of existing change point detection techniques, we refer to a survey by Aminikhanghahi et al. [AC17b].

2.2.2 White-box Performance Modeling

White-box performance modeling aims to build a model of an application based on detailed information about the internals of the application, such as the underlying hardware infrastructure, the distribution of software components on this hardware, the control flow within and between the software components, the resource consumption of the software components, and the usage profile. Therefore, it can only be used if all this information is available to the performance engineer.

Examples for white-box performance models are queueing networks, layered queueing networks, colored Petri nets, stochastic Petri nets, or stochastic process algebras. To predict the expected performance of modeled applications, these models can be solved either analytically or based on simulations. Analytical solution approaches, such as mean-value analysis, impose strict restrictions on the structure of the underlying model, whereas simulation-based model solvers, such as QPME[15], LQNS[16], or JMT[17], can be applied to a larger class of models, but the simulation of larger applications can get quite time-intensive.

A key parameter of these models is the so-called resource demand or service demand, which is defined as "the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU or hard disk) in a system over all visits excluding any waiting times" [Spi+15]. The accuracy and level of detail of the resource demands used in a white-box performance model is one of the key influencing factors for its prediction accuracy, however, measuring them is challenging. Therefore, a number of estimation approaches based on, for example, Kalman-filters, queueing theory, regression models, or maximum likelihood estimation have been proposed. For a comprehensive evaluation of these approaches, we refer to a review by Spindler et al. [Spi+15].

One of the factors that inhibit the usage of, for example queueing models, in practice is the semantic disconnect between the perspective of an engineer who is thinking in terms of servers and software components and the theoretical concepts of queues or Markov chains. Architectural performance models aim to bridge this semantic gap by enabling the software engineer to model the application using the concepts of servers, software components, interfaces, etc. These models are then automatically transformed to stochastical models such as queueing models or stochastic process algebras to predict the performance. Examples of such architectural models are the Descartes Modeling Languague (DML)[18], Unified Modeling Language (UML) Modeling and Analysis of Real-Time and Embedded systems

[15] https://se.informatik.uni-wuerzburg.de/software-engineering-group/tools/qpme/
[16] http://www.sce.carleton.ca/rads/lqns
[17] http://jmt.sourceforge.net/
[18] https://se.informatik.uni-wuerzburg.de/software-engineering-group/tools/dml/descartes-modeling-language/

(MARTE)[19] or the Palladio Component Model (PCM)[20].

The main selling point of white-box performance models is their capability of analyzing hypothetical scenarios without relying on extrapolation. By adapting parameters of the model, white-box performance models can answer questions, such as: What is the expected response time if the number of users doubles? How much would the performance be improved by adding an additional server? How much do I need to speed up the implementation of a component until it is no longer the bottleneck in the system? However, white-box performance models also come with the downside that they are very time-intensive to build and maintain.

Therefore, a number of techniques have been developed to automatically extract white-box performance models. The component architecture and control flow are usually extracted either from tracing information or via static code analysis. The resource demands are traditionally extracted by combining measurement-based approaches with the resource demand estimation approaches discussed above. Finally, the information about the deployment and the underlying resource landscape is derived from deployment descriptors and resource managers, such as Helm charts in the case of Kubernetes.

2.2.3 Black-box Performance Modeling

In many scenarios, software developers do not have white-box access to all components of their applications. The reasons can range from cloud providers obscuring platform details or the usage of third-party APIs, to the fact that understanding the details of for example an open-source database might be too time-consuming. Therefore, black-box performance modeling techniques aim to model application performance without knowledge of its internal structure and processes.

For most black-box performance modeling techniques, the general idea is the same. First, the configuration space, that is, the number of variation points and their potential values are defined. Then, for a subset of these configuration points, the performance (e.g., response time, throughput, or utilization) is measured using performance tests. Finally, a stochastic model or a machine learning model is fit based on the performance of the measured configurations. The resulting model can then predict the performance of the unmeasured configurations by inter/extrapolating from the measured configurations.

One of the key parameters of such approaches is the number of measured configurations. On the one hand, measuring additional configurations tends to increase the accuracy of the model, but on the other hand, there is a time and monetary investment associated with every measured configuration. Therefore, black-box performance modeling approaches aim to determine the minimal number of measurements that offer an acceptable prediction accuracy. A common technique here is

[19] https://www.omg.org/spec/MARTE/

[20] https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model

to incrementally increase the number of measurements until k-fold cross-validation reports an acceptable prediction accuracy.

Another angle to optimize black-box performance models is the selection of the measured configurations. Intuitively, it makes sense to select the measured configurations so that they cover the full configuration space evenly. Another approach is to iteratively select the next measured configuration based on in-model confidence, which enables the approach to allocate more samples to the areas of the configuration space that shows higher performance variability. Interestingly, in a study by Alves et al. [Alv+20], no measurement point selection strategy was able to outperform random sampling.

The final key parameter that impacts the quality of a black-box performance model is the selected modeling approach. Examples for modeling techniques include stochastic regression models (e.g., linear regression, Multivariate Adaptive Regression Splines (MARS), or kriging), decision tree-based models (e.g., Classification and Regression Trees (CART), M5 trees, or cubist forests), and optimization techniques (e.g., genetic programming). There have been several studies comparing these techniques, but there is currently no consensus on a single best technique as the best performing technique depends on the data set [Wes+12; Noo+13].

An emerging trend to further reduce the number of required measurements is the usage of transfer learning techniques. Here, a model is first trained on system A, and then later transferred to system B. To adapt the model to a new system, additional measurements on the new system are required so that the model can learn about the differences between the two systems. The underlying idea is that many relationships hold across most software systems, such as response time increases with higher load, and therefore do not need to be relearned for every software system. In general, the efficiency of transfer learning for black-box performance modeling highly depends on the similarity of the two systems, with it performing better for very similar systems, such as for example two variants of the same system, and transfer learning being potentially detrimental if the systems are too dissimilar.

2.2.4 Automated Performance Optimization

Optimizing the performance of a software system is a classic performance engineering task, where the performance engineer attempts to find the system configuration (deployment, resource allocation, cluster size, component configuration, etc.) that exhibits the optimal performance properties (response time, throughput, resource utilization, reliability, durability, etc.). A number of techniques have been proposed to automate this process, which all follow a similar process. First, a *selection criterion* determines which system configuration should be investigated next, then a *configuration evaluation* approach is used to quantify the performance properties of this system configuration. Next, a *multi-objective optimization* determines if this

configuration is better than the existing system configurations. Finally, a *termination criterion* determines whether additional system configurations should be explored.

The two common types of *selection criterion* are search-based and optimization-based approaches. Search-based techniques aim to provide strong coverage of the configuration space to find which configuration performs best, either through a full exploration of the configuration space or based on heuristics if the configuration space is too large. Optimization-based techniques, on the other hand, use an initial system configuration and aim to improve up this configuration. While search-based approaches tend to identify overall better system configurations, optimization-based techniques tend to identify good system configurations that are close to the initial configuration. This makes search-based approaches best suited for greenfield applications, whereas optimization-based approaches are better suited for brownfield applications.

For automated performance evaluation, it is essential that the system properties of interest (e.g., response time, throughput, utilization) can be quantified for every system configuration selected by the *selection criterion*. If very high accuracy is required, this is usually conducted using performance tests (see Section 2.2.1), however, this can be very time- and cost-intensive. Alternatives are the use of white-box performance modeling (see Section 2.2.2) or black-box performance modeling (see Section 2.2.3) in scenarios where a slightly lower accuracy is tolerable.

The automated comparison of system configurations can be challenging even after determining their performance properties. For example, configuration A might have a faster response time and better throughput, whereas configuration B has a lower cost and better resource utilization. Such solutions (where no other configuration is strictly better) are called Pareto-optimal. This challenge is usually addressed using multi-objective optimization techniques, such as a priori methods, a posteriori methods, and interactive methods. Here, we refer to a survey by Marler et. al. on multi-objective optimization techniques [MA04]. Finally, once a termination criterion such as a target performance, a time/cost budget, or sufficient coverage of the configuration space is reached, the automated performance optimization returns the best configuration it found alongside the expected performance properties of this configuration.

Chapter 3

State-of-the-Art

This thesis makes contributions in multiple areas related to both serverless computing and software performance modeling. In both of these fields, there already exists a large number of related work that serves as the foundation upon which our contributions build. In the following, we discuss the current state-of-the-art on both serverless computing (Section 3.1) and software performance modeling (Section 3.2). We focus on the following topics, which are closely related to the contributions presented in this thesis: serverless applications (Section 3.1.1), performance variability of serverless platforms (Section 3.1.2), and the cost prediction for serverless applications (Section 3.1.3) as well as hybrid performance models (Section 3.2.1) and parametric dependencies (Section 3.2.2).

3.1 Serverless Computing

Since the announcement of AWS Lambda brought serverless computing into the mainstream, researchers have increasingly investigated both serverless applications and serverless platforms. In the following, we give an overview of the current scientific state-of-the-art in regard to serverless applications (Section 3.1.1), performance variability of serverless platforms (Section 3.1.2), and the cost prediction for serverless applications (Section 3.1.3).

3.1.1 Serverless Applications

Serverless computing is an emerging technology with an increasing impact on our modern society, and increasing adoption by academia and industry. As it is an emerging technology, researchers are still trying to understand for which classes of applications serverless computing is a good fit, and what the characteristics of serverless applications in the wild are. In the following, we discuss existing case studies for serverless applications (Section 3.1.1.1) and attempts to characterize serverless applications (3.1.1.2).

3.1.1.1 Serverless Application Case Studies

In the early stages of serverless computing adoption, researchers conducted a number of case studies to determine the applicability and challenges of serverless computing for different domains.

One of the first such case studies was conducted by Lehva et al. [LMM17], who designed a messenger chatbot. They find that serverless messenger chatbots are not only feasible but highly scalable, low-maintenance, and cost-efficient. As potential challenges for serverless applications, they find limited debugging and complex pricing models.

Fouladi et al. [Fou+17; Fou+19] investigate the feasibility of offloading tasks from consumer computers to the cloud using serverless functions. Here, serverless offers the advantage, compared to virtual machines, of being able to set up a highly parallel execution environment within seconds for a small price. They showcase the benefits of this approach for tasks such as video processing [Fou+17], software compilation, unit testing, and object recognition [Fou+19].

Data analytics is another infrequently executed task that could profit from the serverless pay-per-use model. Müller et. al. [MMA20] evaluate the applicability of serverless computing for distributed data analytics. They find a number of limitations that make this challenging but introduce Lambada, a framework that can be used to overcome these challenges.

Tu et al. [TLL18] and Feng et al. [Fen+18] showcase that serverless functions can be used for neural network serving and training respectively. Before, the potentially large model size and the long training time of neural networks were considered prohibitive, but the increasing maturity of serverless platforms alleviated many of the previous roadblocks.

Hellerstein et al. [Hel+18] argued in 2018 that serverless computing is not well suited for scientific computing tasks. However, in recent years, a number of successful serverless scientific computing use cases were published. Witte et al. apply serverless computing for seismic imaging [Wit+20], Cresp et al. for SNP Genotyping [Cre+19], Niu et al. for sequence comparison [Niu+19b], and Vazquez et al. for magnetic field identification [VL18].

To summarize, serverless computing has been shown to be applicable in a wide range of domains, even in domains that were initially considered unsuited.

3.1.1.2 Serverless Application Characterization

Despite a large number of case studies on the applicability of serverless computing, there are very few academic efforts to characterize serverless applications, which would help answer important questions, such as *Why developers build serverless applications?*, *When are serverless applications useful?*, or *How are serverless applications implemented in practice?*.

Castro et al. [Cas+19] discuss ten applications collected from reports of successful adoption of serverless computing by large companies. They argue that each of these applications is a representative of a class of applications, such as event processing, API composition, or map-reduce. However, they do not provide any characteristic analysis on top of these collected use cases.

Eskandani et al. [ES21] collect a large data set of serverless open-source applications that use the serverless framework[1] from Github. The authors argue that this data set can provide the foundation for a number of studies such as the detection of serverless security patterns, the analysis of serverless anti-patterns, the investigation of how serverless applications evolve, and the analysis of the characteristics of serverless applications.

To summarize, there is to the best of our knowledge no existing study that examines the characteristics of serverless applications.

3.1.2 Performance Variability of Serverless Platforms

Performance variability (that is, non-steady or non-uniform performance either over time or across replicated instances) of cloud platforms due to, for example, the scheduling, placement, or hardware heterogeneity of cloud resources has been widely discussed for IaaS offerings, and initial studies for serverless systems also exist. In the following, we introduce the existing work on the performance variability of IaaS cloud offerings (Section 3.1.2.1) and the existing work on the performance variability of serverless offerings (Section 3.1.2.2).

3.1.2.1 Performance Variability of IaaS Cloud Offerings

One of the first major studies on the performance variability of IaaS cloud offerings was conducted by Schad et al. [SDQ10] in 2010, who investigated the performance variability of EC2, the IaaS offering of AWS. Towards this goal, they conducted hourly performance measurements of a map-reduce task for one month on both EC2 and a local cluster. They find that EC2 experiences a significantly larger performance variation compared to the local cluster. Upon further investigation, they determine that the performance falls into two groups depending on which CPU model is used by the underlying hardware.

Iosup et al. [IYE11] investigate the performance variability of AWS and Google Cloud based on performance data collected by Hyperic's CloudStatus team, which contains performance indicators for multiple cloud services of each provider. Based on the data set for the year 2009, they determine that the performance variability they observe is not purely random but can contain both monthly and yearly patterns. They also find several time periods exhibiting stable performance.

[1] https://www.serverless.com/

In 2016, Leitner and Cito [LC16] benchmark the performance of the IaaS solutions of AWS, Google Cloud, Microsoft Azure, and IBM Cloud six times a day for a period of a month using multiple microbenchmarks. Based on this data set, they investigate 15 hypotheses that they derived based on a systematic literature survey. Among other things, they find reduced hardware heterogeneity and no clearly defined daily or weekly patterns that would explain the observed performance variability.

To investigate the impact of the performance variability of Virtual Machines (VMs) on performance assurance activities (e.g., performance testing and microbenchmarking), Laaber et al. [LL18; LSL19] evaluated the variability of microbenchmarking results in different virtualization environments and analyzed the results from a statistical perspective. They found that not all cloud providers and instance types are equally suited for performance microbenchmarking.

3.1.2.2 Performance Evaluation of Serverless Platforms

A number of empirical measurement studies to evaluate the performance of serverless applications have been conducted, which we discuss in the following with a focus on performance variability.

Lloyd et al. [Llo+18] examined the infrastructure elasticity, load balancing, provisioning variation, infrastructure retention, and memory reservation size of AWS Lambda and Azure Functions. They found that cold and warm execution times are correlated with the number of containers per host, which makes the number of containers per host a major source of performance variability.

Wang et al. [Wan+18a] conducted a large measurement study that focuses on reverse engineering platform details. They found variation in the underlying CPU model used and low performance isolation between multiple functions on the same host. When they repeated a subset of their measurements about half a year later, they found significant changes in the platform behavior.

Lee et al. [LSF18] analyzed the performance of CPU, memory, and disk-intensive functions with different invocation patterns. They found that file I/O decreases with increasing numbers of concurrent requests and that the response time distribution remained stable for a varying workload on AWS.

Yu et al. [Yu+20] compared the performance of AWS Lambda to two open-source platforms, OpenWhisk and Fn. They found that Linux CPU shares offer insufficient performance isolation and that performance can degrade when co-locating different applications.

However, while the performance of FaaS platforms has been extensively studied, there has been little focus on the stability of these measurements over time.

There have also been a number of measurement tools and benchmarks developed for serverless applications and platforms. Cordingly et al. [CSL20] introduced the Serverless Application Analytics Framework (SAAF), a tool that allows profiling FaaS workload performance and resource utilization on public clouds; however, it

does not provide any example applications. They find that the performance variability of a CPU-intensive microbenchmark roughly doubles with parallel execution compared to sequential execution due to the hardware heterogeneity.

Figiela et al. [Fig+18] introduced a benchmarking suite for serverless platforms and evaluated the performance of AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Functions. The workloads included in the benchmarking suite consist of synthetic benchmark functions, such as a Mersenne twister or Linpack implementation. They found that function instances on AWS Lambda are executed on four different processor models and are regularly recycled.

Kim et al. [KL19] proposed FunctionBench, a suite of function workloads for various cloud providers. The functions included in FunctionBench closely resemble realistic workloads, such as video processing or model serving, but they only cover single functions and not entire applications. In summary, there has been a strong focus on benchmarks and tooling around FaaS but much less focus on realistic applications.

For further details on the current state of the performance evaluation of serverless offerings, we refer to an extensive multi-vocal literature review by Scheuner et al. [SL20]. This review also finds that the reproducibility of the surveyed studies is a major challenge.

3.1.3 Cost Optimization of Serverless Applications

In a serverless computing model, most operational tasks are offloaded to the cloud provider. However, engineering for cost-performance is still a developer responsibility [Eyk+18]. Therefore, we discuss the current state-of-the-art on the economics of serverless computing (Section 3.1.3.1), the cost prediction of serverless functions (Section 3.1.3.2), and the optimization of function size (Section 3.1.3.3).

3.1.3.1 Economics of Serverless Computing

There have been multiple publications discussing the general economic aspects and pricing models of serverless computing.

In the work of Adzic and Chatley [AC17a], two industrial case studies of companies migrating from traditional hosting options to serverless computing are presented. The companies reported cost savings of 66% and 95% respectively after switching to serverless computing. The authors also discuss the non-constant response times of serverless functions as a limitation of current serverless platforms.

Eivy et al. [Eiv17] claim that while the cost of serverless computing seems simple on the surface, they are surprisingly complicated in practice. They discuss the issue of rounding up the function execution times to 100 ms and that response time estimates require deploying and testing the function. They also compare a serverless solution to traditional hosting in a case study with a large-scale API, where the cost for the serverless solution is almost trice the cost for the VM-based solution.

Vazquez et al. [VL18] conducted a study on the applicability of serverless computing for data-intensive applications. They compare a solution based on AWS Lamba to using EC2 to process data collected by the MARS Express orbiter from the European Space Agency. In their case study, both solutions incur a similar cost, but the serverless solution is roughly twice as fast.

3.1.3.2 Cost Prediction and Optimization of Serverless Functions

The cost prediction and subsequent optimization of serverless functions and workflow enable the cost-efficient operation of serverless applications.

In the work of Boza et al. [Boz+17], an approach using model-based simulations to compare the cost of reserved VMs, on-demand VMs, and serverless functions is introduced. The authors propose to model serverless functions as $M(t)/M/\infty$ queues, which assumes constant function response times and does not consider the impact of input parameters. The authors further conducted a survey with 96 participants, which revealed that many companies rely on reserved VMs to simplify financial planning.

Another approach to optimize the cost of serverless workflows by deciding whether to fuse multiple functions into a larger function and which memory limit should be allocated to a serverless function is proposed in the work by Elgmal [Elg18]. This approach also relies on a constant value instead of a distribution for the response time of the serverless function and does not consider the impact of input parameters on the response time of a serverless function.

Gunasekaran et al. [Gun+19] propose the use of serverless functions in combination with VM-based hosting to enable Service Level Objective (SLO) and cost-aware resource procurement. To enable the cost-aware decision-making between VM-based hosting and serverless functions, the authors also rely on cost predictions for the serverless functions. This approach also relies on a constant response time of the serverless function and does not consider the impact of input parameters on the response time of a serverless function.

However, none of these approaches consider the influence of input parameters on the execution time and therefore execution cost, or the need to predict execution time distributions for the accurate prediction of billed cost.

3.1.3.3 Analysis and Optimizing of Serverless Function Size

Zhang et al. [Zha+19] analyze the impact of different configuration options of serverless functions on their cost and performance. They observe that the impact of memory sizes is non-trivial and highly related to the workload. In 2018, Wang et al. [Wan+18b] conducted one of the largest measurement studies of serverless functions to date. They find that memory size impacts not only the function execution time but also the cold start duration. Figiela et al. [Fig+18] introduce a benchmarking framework for serverless functions on AWS, GCloud, and IBM Cloud. They find that

the impact of memory size on execution time differs between providers and that finding the best performing memory size is non-trivial. Back et al. [BA18] evaluate the performance and cost model of public and private serverless function platforms using microbenchmarks. They find that the relation between memory size and cost is not linear and depends on the cloud provider.

These works showcase the need for the automated memory size optimization of serverless functions, as it significantly impacts both the performance and cost of serverless functions. To the best of our knowledge, there have currently been three approaches proposed for the automated function size optimization for serverless functions.

The first approach is the AWS power tuning tool, a popular open-source tool that measures the impact of different memory sizes on the execution time and cost of a serverless function [Cas20a]. A step function workflow coordinates the deployment, performance measurement, and result collection for a set of predefined memory configurations.

The second approach is COSE [Akh+20], which aims to reduce the number of required measurements using Bayesian optimization. By learning a performance model describing the relationship between memory size and execution time based on fewer measurement points, COSE can reduce the number of required performance measurements.

Lastly, BATCH [Ali+20] is a framework for efficient machine learning serving on serverless platforms. It relies on a profiler that measures a subset of the potential configurations (that is, memory size, batch size, and timeout) and employs a multivariable polynomial regression model to estimate the performance of the remaining configurations.

To summarize, all existing approaches for the memory size optimization of serverless functions, combine sparse measurements with interpolation or modeling to determine the optimal memory size, an approach that is commonly applied for data analytics systems [Ven+16; Ali+17; HDB11]. However, all existing approaches require measuring multiple function sizes.

3.2 Software Performance Models

Software performance models provide a powerful tool enabling performance prediction for software systems. These performance predictions can be used for many purposes, such as capacity planning [BKR09] and automated resource management [Hub+17]. For serverless platforms and systems, many aspects of software performance models can be directly reused, however, there are also open challenges [Eyk+18; MK21]. In the following, we discuss the current state-of-the-art on hybrid performance models (Section 3.2.1) and parametric dependencies (Section 3.2.2), which attempt to address some of these challenges.

3.2.1 Hybrid Performance Models

Hybrid performance models aim to negate the downsides of white-box performance models (long simulation time) and black-box performance models (limited extrapolation) by combining them into a single model. In the following, this section discusses the current state of white-box performance models (Section 3.2.1.1), black-box performance models (Section 3.2.1.2), model reduction techniques (Section 3.2.1.3), and hybrid performance models (Section 3.2.1.4).

3.2.1.1 White-box Performance Models

The UML extension MARTE [ZJI15] is a commonly used architectural performance model that focuses on real-time and embedded systems by extending the UML modeling formalism with performance annotations. UML MARTE has later been extended by the DICE profile to enable modeling of technology-specific aspects of big data frameworks such as Hadoop or Spark [CP17].

Grassi et al. introduce Kernel LAnguage for PErformance and Reliability analysis (CLAPER) [GMS07], a kernel language based on the Meta-Object Facility (MOF), aims to simplify the transformation of design time models, such as UML, to analysis models. By acting as an intermediary model between design and analysis models, CLAPER significantly reduces the number of required model transformations compared to a full mapping from design models to analysis models.

The Palladio Component Model (PCM) [Reu+16] is a white-box performance model for the prediction, analysis, and optimization of software projects during design time. The PCM model is separated into five sub-models: (1) the component repository, (2) the system assembly, (3) the resource environment, (4) the deployment, and (5) the usage profile.

The Descartes Modeling Language (DML) [Gro+21; Hub+17] is a white-box performance model to guide resource allocation at runtime. Towards this goal, DML supports the automated, continuous extraction of model parameters from monitoring data and the automated, performance-aware self-adaptation at runtime.

For an in-depth overview of white-box performance models, we refer to the surveys by Koziolek [Koz10] and Balsamo et al. [Bal+04].

In order to predict performance indices such as response time, throughput, or hardware utilization, for each considered system state (i.e., system configuration including resource allocations), these white-box performance models are transformed to predictive stochastic models, such as queueing networks [AM20], layered queueing networks [IPW18], colored Petri nets [JK15], stochastic Petri nets [ZH19], or stochastic process algebras [MHS19]. These stochastic models can be solved either analytically or based on an event-based simulation [Bro+15].

Analytical solution approaches, such as mean-value analysis [ZCS07], impose strict restrictions on the structure of the underlying model making them inapplicable for complex software systems. Simulation-based model solvers, such as

QPME [KD09], LQNS [Fra+09], or JMT [BCS09], can be applied to a larger class of models, but for large software systems, the time required for the simulation often becomes prohibitive [Nam+16; WFP07; Koz10]. To summarize, white-box performance models are a powerful tool to analyze the performance of software systems, however, the simulation time can be a limiting factor for larger systems.

3.2.1.2 Black-box Performance Models

Black-box performance models are also known as performance prediction functions [Wes+12], software performance curves [FH12; WM10], performance predictions using machine learning [Kwo+13] or performance predictions using statistical techniques [The+10]. These approaches train machine learning/statistical models on measurement data, which is usually collected during dedicated measurements. These models are used to infer the performance of a software system with different workloads or configurations.

Thereska et al. [The+10] build a black-box performance model to predict the performance of several Microsoft applications, such as the Office suite or Visual Studio. By instrumenting the applications, the authors collect data from several hundred thousand real users. CART is used to filter relevant features, followed by a similarity search to derive performance predictions.

Kwon et al. [Kwo+13] use a regression model to predict the performance of Android applications to determine whether the task can be efficiently offloaded. The authors train the regression model not only on the input parameters and the current hardware utilization but also on values calculated during the program execution.

Westermann et al. [Wes+12] compare four techniques for the construction of black-box performance models: MARS, CART, Genetic programming, and Kriging. In their case studies, MARS significantly outperformed the other three approaches. Additionally, the authors evaluate three different measurement point selection algorithms, which reduce the required number of dedicated performance measurements.

Noorshams et al. [Noo+13] evaluate the accuracy of regression techniques for the black-box performance prediction of storage systems: linear regression, MARS, CART, M5 Trees, and Cubist Forests. The authors propose an approach to optimize the parameterization of the individual algorithms. With their parameter optimization, MARS and Cubist outperform CART, M5 Trees, and the linear regression in their case study.

Faber et al. [FH12] use genetic programming to derive software performance curves. They introduce parameter optimization approaches and a technique to prevent overfitting for genetic programming. In their evaluation, the optimized genetic programming approach outperforms an unoptimized MARS model.

To summarize, black-box performance models can predict the impact of changes in workload intensity and workload parameterization well but are unreliable when predicting the impact of changes to the system or its deployment.

3.2.1.3 Performance Model Reduction

Various techniques to reduce or simplify performance models have been proposed. Some techniques aim to improve the model solution speed whereas others attempt to make the model easier to understand for human users.

Queueing networks can be simplified by replacing a number of nodes with a so-called flow-equivalent server [Bol+98; Laz+84]. A flow-equivalent server is a load-dependent queue, which perfectly emulates the delay caused by a subsystem. The resulting network can be solved faster using analytical approaches, such as convolution or mean value analysis. However, flow-equivalent servers come with two drawbacks: i) the representation as a load-dependent queue is cumbersome for discrete-event-simulation and ii) in order to construct a flow-equivalent server, the short-circuited network needs to be solved once for every possible number of concurrent users, so in order to construct a flow-equivalent server for a system with a thousand users, the model needs to be solved a thousand times.

The method of surrogate delays by Jacobson et al. [JL81] enables an analytical solution for models with simultaneous resource possession. The approach requires a model, where the primary resource is estimated by a delay and another one where the secondary resource is estimated by a delay. While this method does reduce the initial model, it requires an additional model and does not provide any advantages when solved using simulation.

A more recent approach by Islam et al. [IPW15] reduces the size of layered queueing networks by aggregating activities, tasks, and entries of non-bottleneck resources. This approach is extended by an improved approach for the identification of tasks that can be safely aggregated [IPW18]. As this approach permanently reduces the model, it is no longer possible to accurately analyze the impact of reconfigurations or deployment changes for non-bottleneck resources.

To the best of our knowledge, there is no existing model reduction approach for simulation-based solvers that enables faster model solution while retaining the full flexibility and accuracy of the initial model.

3.2.1.4 Hybrid Performance Models

Some existing approaches aim to utilize hybrid performance models, that is, combinations of black-box and white-box performance models in order to negate some of the downsides of the individual formalisms.

Therefore, Noorshams et al. [Noo+14] propose a methodology to enrich software architecture modeling approaches with black-box I/O performance models. Storage access is modeled as traditional resource access, but the response time of the storage system is determined by a black-box model. This approach enables accurate performance predictions for software systems using dedicated storage systems, without having to explicitly model them. This approach is limited to storage systems and

does not allow to dynamically switch between the black-box model and a traditional queueing model for the storage system.

Woodside et al. [WPS02] propose the concept of performance-related completions, where an abstract model element is later replaced with a concrete model, once the information is available. Happe et al. [Hap+10] extend this approach by using platform-independent model skeletons. The platform-specific details are derived based on measurements from automatically generated test drivers. This approach incorporates measurements in performance models in order to improve the parameterization of the performance model and therefore the performance prediction accuracy.

To the best of our knowledge, no existing approach allows to dynamically switch between black-box response time models and queueing model representations for generic components or sub-systems.

3.2.2 Parametric Dependencies

Providing accurate performance predictions using white-box performance models requires the modeling of several system properties. Specifically, it requires the explicit modeling of dependencies between different model parameters, for example, the resource demand of a service might depend on the value of its input parameters. Modeling such parametric dependencies expands the range of system settings that can be accurately modeled [BHK14; BCK07; Koz08; KKR10]. Formal modeling of parametric dependencies allows predicting the impact of changing workloads, system reconfigurations, and deployment changes using a single performance model. The SPEC RG Cloud[2] also identified modeling the impact of parameters on the performance of serverless functions as a performance challenge for Function-as-a-Service architectures [Eyk+18]. Therefore, in the following, we present the current state-of-the-art on modeling parametric dependencies (Section 3.2.2.2) and the automated learning of parametric dependencies (Section 3.2.2.1).

3.2.2.1 Automated Learning of Parametric Dependencies

Manual modeling of parametric dependencies is quite cumbersome and therefore a number of techniques for the automated learning of parametric dependencies have been proposed.

Krogmann et. al. [KKR10] propose an approach to extract a model that depicts the influence of the execution platform, the usage profile, and the control flow on the resource demand of software components. The models are learned at runtime from monitoring data, bytecode counts, and static bytecode analysis using genetic programming. The resulting model can predict the performance behavior of a software component in scenarios with previously unobserved execution platform,

[2] https://research.spec.org/working-groups/rg-cloud/

Modeling Feature	Static Parameter Model [WW04; GM01] [Zsc10; LGF05]	Dynamic Parameter Model [Koz08; Sit+01] [Ham09; Bon+05]	Persistent Parameter Model [HBR14]
In- and output parameters	×	✓	✓
Component-level dep.	×	✓	✓
Dependency chaining	×	✓	✓
Instance-level dep.	×	×	×
Multiple descriptions	×	×	×
Correlations as dep.	×	(✓)	(✓)

Table 3.1: Support of parametric dependency modeling features by existing performance modeling approaches.

usage profile, and control flow. However, the bytecode counter monitoring incurs significant overheads.

Brosig et. al. [BHK11] forgo the requirement for bytecode counter monitoring and propose an approach to learn the general model structure, model parameters, and probabilistic parameter dependencies based on run-time monitoring data. The extraction of probabilistic parameter dependencies assumes that every target variable can only depend on a single influencing factor and then estimates the probability mass function based on the observed tuples of target and influencing variables.

Mazkatli et. al. [Maz+20] argue that learning the full performance model for every deployment in a DevOps environment is too costly and time-intensive and therefore propose an approach for the incremental extraction of performance models including parametric dependencies. For every commit, the changes in the source code are analyzed to determine which parts of the model have potentially changed. Next, adaptive monitoring allows to selectively monitor only the relevant parts of the application. Finally, the model is updated and a self-validation phase determines if the resulting model can accurately depict the changed performance of the system.

3.2.2.2 Modeling Parametric Dependencies

We categorize existing performance modeling approaches according to their parametric dependency modeling capabilities into four categories:

a) Modeling formalisms having *no parameter model* lack modeling features to model parameters or their impact on the performance of individual components. Such models cannot capture changing workload mixes and system configurations within a single model. Examples include stochastic formalisms, like Queueing Network (QN), QPN, Layered Queueing Network (LQN), and process models, as well as some architectural models [GMW97; MM02; WI03].

b) Meta-models with a *static parameter model*, as presented in [WW04; GM01; Zsc10; LGF05], allow for the parametrization of component instances to model the

influence of instance-specific parameters on component performance. However, these approaches require each component instance to be parameterized individually. They lack features to model the impact of deployment changes on component performance. Instance-specific parameters can be hardware parameters, such as processing rate or number of cores, or software parameters, such as operating system, hypervisor, or number of virtual cores [Zsc10]. Parameters can also be used to model different component configurations, as for example, two instances of a compression component with different compression rates. Resource demands and external call frequencies can be described as a linear combination of the static parameters [GM01].

c) *Dynamic parameter models* are supported for example, by PCM [Koz08], RESOLVE [Sit+01], HAMLET [Ham09], ROBOCOP [Bon+05]. While these models suit design-time analysis, they encounter deficiencies at run-time. This class of models allows the modeling of input [Bon+05] and output parameters for components and external calls [Koz08]. The resolution of parameter values propagates from caller to callee which makes parameter values dependent on the deployment context. Correlations that do not propagate from caller to callee cannot be modeled.

d) *Persistent parameter models* allow modeling internal states for subsystems and components based on parameters [HBR14]. Internal state means that the performance of consecutive calls to the same component are no longer statistically independent. These parameters may change during the model simulation impacting the component behavior. Modeling the internal state increases the prediction accuracy, but cannot accurately model run-time specific behavior.

Table 3.1 compares the capabilities of existing modeling approaches with regard to modeling of parametric dependencies. The symbol $\times$ means the approach does not support the modeling feature, (✓) indicates partial support, and ✓ stands for complete support. To summarize, none of the related approaches provides means to model correlations as dependencies, instance-level dependencies, or multiple descriptions of the same variable.

Part II

Contributions

Chapter 4

Evaluation of the Characteristics and Performance of Serverless Applications

Many aspects of serverless applications are currently poorly understood, which could hinder the widespread adoption of serverless computing. There exist only few, scattered, and often conflicting reports on the characteristics of serverless applications, which means that developers, academics, and managers often have conflicting views on the characteristics of serverless applications, as described in Challenge 1. Further, the performance of serverless applications could theoretically change at any time if the provider makes changes to the underlying hardware or software stack. As outlined in Challenge 2, there is currently no data on how often the performance of serverless applications changes and how impactful these changes are.

Our first contribution aims to address Challenge 1 and Challenge 2 by improving the understanding of serverless applications on the basis of two empirical studies. The first study systematically collects a total of 89 serverless applications and analyzes them in regard to characteristics, such as execution pattern, workflow coordination, use of external services, and motivation for adopting the serverless paradigm. We complement this data with an analysis of the community consensus regarding these characteristics. The second study is an exploratory case study on the stability of performance measurements of serverless applications using a representative, production-grade serverless application. Our longitudinal analysis investigates the daily performance variation over a period of ten months.

In the following, Section 4.1 introduces our efforts towards the systematic and comprehensive analysis of the characteristics of serverless applications, and Section 4.2 introduces our case study for the analysis of the performance variability of serverless applications in general and over longer periods of time.

4.1 Characteristics of Serverless Applications

There exist only few, scattered, and sometimes conflicting reports addressing important questions such as *Why do developers build serverless applications?*, *When are serverless applications useful?*, or *How are serverless applications implemented in practice?* For example, although some report significant cost savings by switching to

serverless applications [AC17a; Lev20], others identify a higher cost compared to traditional hosting in some scenarios [Eiv17]. Similarly, although reports of successful serverless applications for data-intensive applications exist [Wit+20; Cre+19], other reports claim that serverless computing is not well suited for data-intensive applications [Hel+18]. As a third and last example, although a recent study differentiates between containers and serverless computing and finds the former to be preferable for latency-critical tasks [Cha18], others see them as connected [Pah+19; Mae+20] or report successfully applying serverless computing to latency-critical, user-facing traffic [Orf18]. Having concrete information on these topics would be valuable for developers, to guide decisions on whether serverless computing is a suitable paradigm for their specific application.

This study targets **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*) by systematically gathering the largest collection of serverless applications to date, the comprehensive characterization of serverless applications, and the first analysis of community consensus on the characteristics of serverless applications. Building on resources created by the community [PAM19; SA19], we systematically collect a total of 89 serverless applications from four different sources. This is the largest collection of serverless applications to date, by a factor of 8.9x over the next largest [Cas+19]. Next, we characterize each application from our collection, through a systematic and comprehensive pair-reviewing process, with regard to 16 characteristics, such as execution pattern, workflow coordination, use of external services, and motivation for adopting serverless computing. This addresses **RQ I.1** (*"What are common characteristics of current serverless applications?"*). Finally, we conduct a literature search, finding ten, mostly industrial, web surveys and datasets on the characteristics of serverless applications. We compare the results of these studies to this study, to identify characteristics for which there is a consensus among multiple studies and investigate points of disagreement, which answers **RQ I.2** (*"Is there a community consensus on the common characteristics of serverless applications?"*).

The remainder of this section is structured as follows: Section 4.1.1 describes our methodology for the collection of serverless applications, Section 4.1.2 details our process to identify the characteristics of serverless applications, and Section 4.1.3 outlines our methodology for the identification of community consensus. Finally, Section 4.1.4 discusses the limitations and threats to the validity of this study. The results of this study are discussed in Section 7.1.

4.1.1 Collecting Serverless Applications

Serverless applications have been described in many kinds of materials written for experts including peer-reviewed academic publications, open-source projects, blog posts, podcasts, talks, and provider-reported success stories. The field is only a few years old, so any of these types of materials could include meaningful and unique

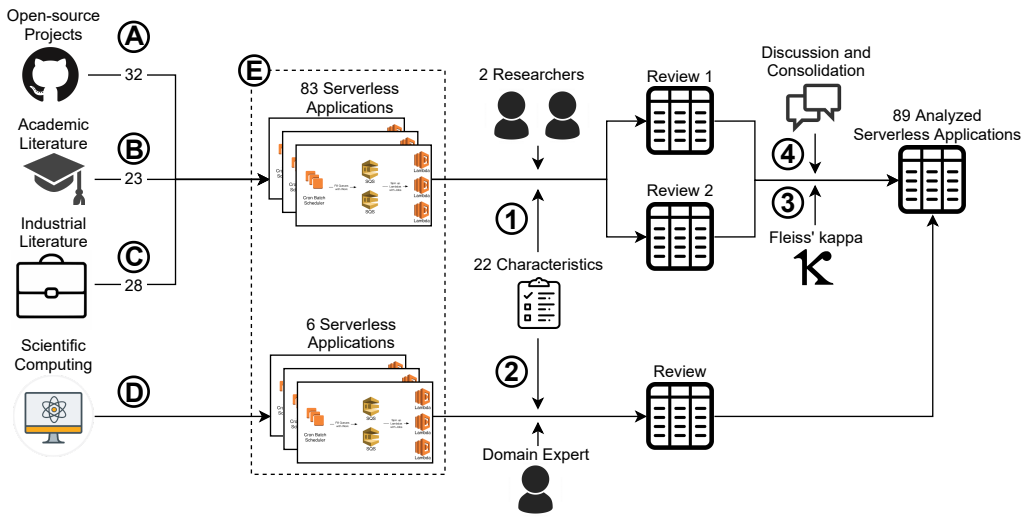


Figure 4.1: Methodology for serverless application collection (left part, Section 4.1.1) and characterization (right part, Section 4.1.2).

material. We aim to create a process for collecting a large number of descriptions of serverless applications, spanning this range of materials judiciously and without a strong selection bias toward one or another. Our aim is not that the process should be exhaustive; doing so while the field is still growing and new applications are still emerging would not be useful in the long term. Figure 4.1 shows the result of our use of this process—a large, varied sample, obtained from the following sources:

Open-source projects (Figure 4.1, component A): We start with an existing dataset on open-source serverless projects [PAM19]. We remove small and inactive projects based on the number of files, commits, contributors, and watchers. Next, we manually filter the resulting dataset to retain only projects that implement serverless applications. Finally, we select only projects that have an active and appreciative community (projects with over 50 stars). This results in a set of 32 serverless applications from open-source projects.

Academic literature (Figure 4.1, B): We base our search on an existing, community-curated dataset on literature for serverless computing of over 180 peer-reviewed articles [SA19]. As we were familiar with five additional publications describing serverless applications, we contribute them to the community-curated dataset and include them in this study. We first filter all the articles based on title and abstract and remove any articles that implement only a single function for evaluation purposes or do not include sufficient detail to enable a review. This results in 23 serverless applications from academic literature.

Industrial literature (Figure 4.1, C): There are many blog posts by companies or individuals, talks at industry conferences, and provider-reported success stories that describe serverless applications. We filter the case studies reported by the major serverless providers (AWS, Azure, Google, and IBM) and select from them the solutions that depend on serverless technology. We also include the ten applications reported in a recent article [Cas+19], which until this work is the largest public collection of serverless applications from industrial literature. We further extend this collection with industrial literature describing serverless applications in main industry events (e.g., KubeCon). This process results in 28 serverless applications from industrial literature.

Scientific computing (Figure 4.1, D): The scientific computing community is showing increasing interest in serverless solutions (e.g., at NASA [Wal19a] and CERN [Blo+19]). However, most of these applications are still at an early stage, with scarce public information. To address this deficit of public data, we collect and include information from the German Aerospace Center (DLR) and the German Electron Synchrotron (DESY) in this work. This results in 6 serverless applications from the area of scientific computing.

For each of these sources, we use the same predefined inclusion (I) and exclusion (E) criteria to determine if an application should be included in our dataset:

I1 Concrete application. Real-world use is a plus.

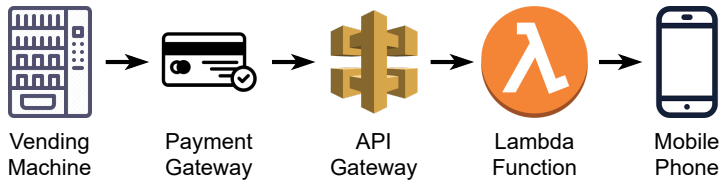
I2 Application description has sufficient detail to conduct a meaningful review. (Exclude high-level descriptions that lack technical detail.)

E1 Serverless platforms and frameworks, as these are not serverless applications.

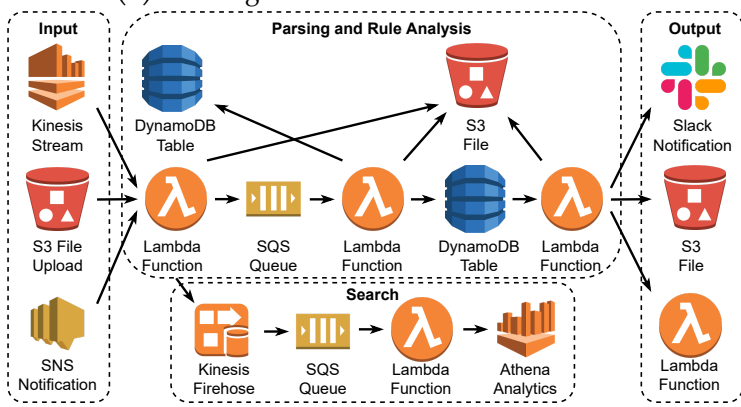
E2 Boilerplate code and simple technology demonstrations, as they do not constitute real-world applications.

I3/E3 Include only one out of multiple academic papers describing the same use case. For example, many academic papers discuss serverless neural network serving [IMS18; Bha+19; TLL18], but we only include a single representative paper.

Based on this methodology, we collect a diverse dataset of 89 serverless applications from open-source projects, academic literature, industrial literature, and scientific computing. This dataset (see Figure 4.1, component E) is publicly available as part of our replication package. Out of the total of 89 applications, 55% are used in production, and 53% are open-source. Researchers can use this dataset to study different applications, which facilitates extracting meaningful patterns and could trigger new designs. The dataset can also help with identifying representative



(a) Vending machine backend at Coca-Cola.



(b) StreamAlert by AirBnB.

Figure 4.2: Two examples of serverless applications from our collected dataset of 89 serverless applications.

applications, which can later be used for the evaluation of novel approaches and in empirical studies. Engineers can find useful examples in the dataset and identify areas in which serverless computing is successfully applied, to help decide whether to adopt serverless computing and to select blueprints for similar use cases. Platform providers can extract knowledge on how their products are used and thus optimize them, and any gaps in adoption that can point out deficits in current platform capabilities.

Figure 4.2 shows two example serverless applications from our dataset. Figure 4.2(a) depicts the serverless backend of Coca-Cola vending machines—an operation that handles 30 million requests per year. Figure 4.2(b) illustrates the open-source application StreamAlert, by Airbnb, which allows the validation of security rules on streams of log data. Both applications use the *cloud provider* AWS but are implemented in different *programming languages*. The architecture of these applications is different: whereas the vending machine uses a single *external service*, a managed cloud API gateway, StreamAlert uses several, including managed databases, managed streaming, managed queues, and managed storage. They use different *trigger types*, HTTP requests for the vending-machine backend, cloud events for StreamAlert. The *number of serverless functions* also differs: whereas the vending machine backend uses a single serverless function, StreamAlert consists of many serverless functions.

The workload of both applications further differs in *execution pattern*, *burstiness*, and *data volume*. The vending machine backend focuses on cost savings as the *motivation* behind adopting serverless computing, whereas StreamAlert seems to choose serverless computing to avoid operational overheads. Motivated by this comparison, we focus in the next section on analyzing these and more characteristics for all serverless applications in our dataset.

4.1.2 Determining the Characteristics

Figure 4.1, points 1–4, gives an overview of our methodology to identify the characteristics of serverless applications. Although we apply the methodology described in the following on the dataset collected in Section 4.1.1, the characterization methodology we introduce here could be applied to other, similar datasets. This level of generality allows further comparison between studies, a feature we leverage to conduct our own cross-community study, in Section 4.1.3.

We first identify and formalize the set of investigated characteristics through a multi-round process. In an initial round, we start from a set of questions, and each collaborator suggests characteristics independently, based on expertise. In the next round, we merge similar characteristics and retain only the characteristics that at least two collaborators consider relevant, in this work, 22 characteristics. Based on group discussion, we further define for each characteristic either the range of values or an exhaustive set of potential values, as applicable. For some characteristics, we cannot define a set of potential values before reviewing the applications. For these characteristics, we use text fragments during the review. Using thematic coding [CA96; GMN11], we extract codes and treat those as values for these characteristics.

We then conduct an initial round of reviews (Figure 4.1, label ①). Each application is assigned two reviewers out of a pool of seven available reviewers. We manually adjust a few reviewer assignments to reduce the number of coinciding reviewer pairs. Subsequently, each reviewer individually assigns values to all characteristics of their assigned applications. For the scientific applications, a different approach was necessary, because many were not publicly available at the time of our review. Therefore, these applications are reviewed by a single domain expert, which was either involved in the development of the application or in direct contact with the development (Figure 4.1, label ②). Our replication package contains descriptions of the scientific use cases and outlines which domain experts were consulted for each application.

Each review of an application characterizes it according to 22 characteristics: *cloud platform*, *programming languages*, *external services*, *trigger types*, *number of functions*, *execution pattern*, *burstiness*, *data volume*, *application type*, *function runtime*, *latency relevance*, *motivation*, *cost/performance tradeoff*, *resource bounds*, *locality requirements*, *update frequency*, *domain*, *is it a workflow?*, *workflow coordination*, *workflow structure*, *workflow size*, and *workflow internal parallelism*. For each, the result is typically a

value, one of the possible values for that dimension. However, if the information to determine a characteristic for a serverless application is not available, we label the characteristic as "Unknown" for this application.

After completing the initial round of reviews, we calculate the Fleiss' kappa to quantify the level of agreement between the reviewers [Gwe14] (Figure 4.1, label ③). We exclude all characteristics that use thematic coding and all characteristic assignments where at least one reviewer assigned more than one value, as the Fleiss' kappa can not be calculated in these cases. As each characteristic has a different number of possible values, we calculate Fleiss' kappa value for each characteristic individually and then quantify the overall agreement with a weighted average over the individual Fleiss' kappa value of each characteristic. This results in a Fleiss' kappa value of 0.48, which can be interpreted as "moderate agreement" [LK77].

In the following discussion and consolidation phase (Figure 4.1, label ④), the reviewers compare their notes and try to reach a consensus for the characteristics with conflicting assignments. For most conflicts, consolidation turned out to be a quick process, as the most frequent type of conflict was that one reviewer found additional documentation that the other reviewer did not find. In only a few cases, the two reviewers still have different interpretations of a characteristic; these conflicts are discussed among all collaborators to ensure that characteristic interpretations are consistent. Following this process, we were able to resolve all conflicts.

Our process is data-driven, so it also has to account for missing or malformed data. For 6 characteristics (*resource bounds*, *locality requirements*, *update frequency*, *domain*, *workflow internal parallelism*, and *cost/performance tradeoff*), many applications are assigned the "Unknown" value, that is the reviewers were not able to determine the value of this characteristic, as the required information was missing in the documentation. Therefore, we exclude these characteristics without sufficient information available.

For the remaining 16 characteristics, the percentage of "Unknowns" ranges from 0–19%, with two outliers at 25% and 30%. These "Unknowns" are excluded in the percentage values presented in Section 7.1. A breakdown per characteristic of the "Unknown" percentages is available in our replication package. Additionally, for a single characteristic (the *application type*), the list of potential values turned out to be inadequate, so we repeat the mapping for this characteristic with a new set of potential values that were again derived with the above described process.

4.1.3 Finding Community Consensus

Because the field of serverless applications is relatively new and fast-evolving, reaching community consensus about application patterns and best practices is both desirable and challenging. This section aims to analyze existing studies that analyze one or several characteristics that we also study, and determine if overlapping studies

corroborate our findings or contradict them. The results from this comparisonare a necessary first step towards reaching community consensus.

We aim to find and compare with existing studies in the community on the characteristics of serverless applications. Our methodology consists of three parts: a literature search to identify related studies, mapping their findings to our framework, and quantifying the degree of agreement.

4.1.3.1 Identification of Related Studies

To identify existing surveys and datasets that also investigate at least one of the characteristics investigated in this work, we conducted a literature search. As we are mostly looking for industrial studies and datasets, we use Google as the search engine with the following search term:

$$(\text{“}serverless\text{”}\ OR\ \text{“}faas\text{”})\ AND$$
$$(\text{“}dataset\text{”}\ OR\ \text{“}survey\text{”}\ OR\ \text{“}report\text{”})$$
after: 2018-01-01

This search term looks for any combination of either serverless or FaaS alongside any of the terms: dataset, survey, or report. We further limit the search to articles since 2018, as serverless computing is a fast-moving field, and therefore any older studies are likely outdated. This search term results in a total of *173 unfiltered results*.

To validate if using only a single search engine is sufficient, and the search term is broad enough, we check if the seven studies that we were already familiar with appear in the results. Because the search results include all these studies, we conclude our literature search is broad enough.

We identified relevant results as follows. In the first iteration, to keep primary sources, we filter out results that do not report original data. We remove all reports on secondary data, where the original study was already contained in the search results. This process results in a total of *16 primary studies*. Finally, we determine for each primary study if they investigate one of our characteristics. This resulted in a total of *10 related studies*, which Table 4.1 summarizes. The related studies include 7 surveys and 3 datasets, surveys from 19 to 2 400 participants, and reports between 2018 and 2020.

In the following, we give a short description of each related study, as the methodology and context of each study are important for the correct interpretation of their results.

Serverless Community Study (SCS) This is an online survey among 583 participants from the serverless community, conducted in April 2020. It mostly focuses on end-user concerns, such as how far the end-user is in adopting serverless computing and what challenges they experience.

Table 4.1: Overview of the related studies.

Study	Year	Type	Participants	Source
SitW	2020	Dataset	-	https://bit.ly/3bl2vHM
TSoS	2020	Dataset	-	https://bit.ly/2Zp9zOh
FtLoS	2020	Dataset	-	https://bit.ly/3diWZrY
SCS	2020	Survey	583	https://bit.ly/37p56j4
FSS	2020	Survey	~150	https://bit.ly/2ZsIVUM
OSS	2019	Survey	>1500	https://bit.ly/3dnViJH
MMS	2018	Survey	182	https://bit.ly/3dpcJd6
DSS	2018	Survey	19	https://bit.ly/3qybX15
CNCF	2018	Survey	2400	https://bit.ly/2M2sjQz
GtST	2018	Survey	608	https://bit.ly/3biElxO

Serverless in the Wild (SitW) In 2020, researchers at Microsoft published one of the first comprehensive characterization studies of the workloads of a major (and closed-source) serverless platform. For this, they released all function invocations on the Azure Functions platform for two weeks.

Mixed-Method Study (MMS) The academic mixed-method study combines semi-structured practitioner interviews with 12 experts, a systematic review of 50 grey literature articles, and a quantitative survey covering 182 responses to investigate FaaS software development in industrial practice. Our study only compares against their web survey results from early 2018.

The State of Serverless (TSoS) This study compiles usage data from the customer base of Datadog, a vendor of serverless monitoring solutions. This data was published in early 2020 and focuses solely on AWS Lambda.

O'Reilly Serverless Survey (OSS) In June 2019, O'Reilly surveyed over 1,500 participants from diverse locations, companies, and industries on the adoption of serverless computing.

Guide to Serverless Technologies (GtST) As part of the ebook, "Guide to Serverless Technologies", The New Stack surveyed 608 participants interested in serverless technology. The survey participants were primarily recruited through the company's newsletter and their social media reach-out.

For the Love of Serverless (FtLoS) New Relic, a vendor for a serverless monitoring solution, analyzed serverless computing trends in 2020, based on data covering a sample set of the trillions of serverless events that their product processes.

Fastly Serverless Survey (FSS) Soon after the launch of the beta version of Compute@Edge, Fastly conducted in the beta community a survey about trends and challenges.

Dashbird Serverless Survey (DSS) In 2018, Dashbird surveyed its customers on why they switched to serverless computing, what problems they were trying to solve, and the biggest benefits and drawbacks. The 19 companies in the survey use Dashbird's observability solution on AWS workloads.

CNCF Survey (CNCF) The Cloud Native Computing Foundation regularly surveys its community about the adoption of cloud-native technologies. The 2018 survey includes some questions on serverless adoption and platforms.

4.1.3.2 Mapping the Results to our Framework

Because the related studies (identified in Section 4.1.3.1) offer different *answer options* than our study, we map their options to ours. In many cases, this is straightforward, for example, when mapping "HTTP" to "HTTP Request".

When the granularities of offered options differ between studies, we aggregate lower-granularity options to match the higher granularity. In case the lower-granularity options include multiple answers, we select only the highest value instead of aggregating values, to avoid counting a single study participant multiple times. We provide a detailed account of the mapping for each characteristic and related study as part of our replication package.

4.1.3.3 Quantifying the Degree of Agreement

For many studies, some information required for traditional meta-analysis techniques [BG01], such as cohort size, is unavailable. This prevents the direct application of these meta-analysis techniques.

We propose an agreement metric, a total that equally weighs the agreement of the reported percentage values and the agreement of the reported ranking:

$$A_t = 0.5 \times A_p + 0.5 \times A_r$$

where A_t represents the total agreement, A_p the agreement of the reported percentage values, and A_r the agreement of the reported ranking, with A_p and A_r defined in the following.

We calculate the percentage agreement as the weighted Mean Absolute Percentage Error (MAPE), with the reported percentage value of each answer as the weight:

$$A_p = \sum_{i=1}^{N} Min(1, u_i) \times \frac{|u_i - t_i|}{u_i}$$

where N denotes the number of answer-options; and u_i/t_i are the percentage value reported for option i in our study and the related study, respectively. The formula caps the MAPE for each option at 100%, as otherwise, options with very low percentage values would dominate the MAPE [Mak93]. In some cases, one of the studies allows a participant to select multiple options, while the other study only allows for a single option. To compare these results, we calculate a scaling factor based on the percentage difference of the largest reported values by both studies and scale the results from the study with multiple answers per participant accordingly.

We calculate the agreement regarding the reported ranking as follows:

$$A_r = \frac{S(u, t) + 1}{2}$$

We use Spearman's rank correlation coefficient $S(u, t)$, a common metric to quantify the similarity of two rankings [MWL10]. As Spearman's r value ranges from [-1, 1], we scale it to [0,1] so it has the same scale as A_p. Therefore, the resulting A_t also lies in the range $[0, 1]$.

Finally, we categorize scores in the range $[0.8, 1]$ as very *high agreement*, $[0.6, 0.8)$ as *high agreement*, $[0.4, 0.6)$ as *medium agreement*, $[0.2, 0.4)$ as *low agreement*, and $[0, 0.2)$ as *very low agreement*. We acknowledge that these categories are somewhat arbitrary. However, based on a manual inspection of the results, they seem to capture the individual studies' level of agreement quite well. Our replication package includes the mapped data alongside the resulting scores, to enable readers to conduct manual inspections of the degree of agreement.

4.1.4 Limitations & Threats to Validity

We discuss potential threats to validity and mitigation strategies for internal validity, construct validity, and external validity.

4.1.4.1 Internal Validity

Manual data extraction can lead to inaccurate or incomplete data. To mitigate this threat, we established and discussed a review protocol before reviewing, continuously discussed upcoming questions during the review process, and performed redundant reviews through multiple reviewers. Our review protocol established an exhaustive list of potential values for each characteristic and configured automated validation, which immediately highlighted deviations from these values.

For characteristics with thematic coding, we continuously refined their values in regular meetings during the review process. To address potential individual bias, we performed two independent reviews for each application, quantified the inter-rater agreement after an initial review round through Fleiss' kappa, and resolved each disagreement in an extended discussion and consolidation phase. The goal of this study is to capture and analyze the current state of serverless applications. However, due to our methodology, the collected sources can be several years old and therefore possibly represent already overhauled systems and architectures. As we published all the underlying data, follow-up studies can also focus on the development of different characteristics over time.

4.1.4.2 Construct Validity

To align this study's goal (i.e., comprehensive understanding of existing serverless applications) with the data extraction, we compiled a list of 22 characteristics covering six different aspect groups. We conducted and discussed this selection process in an international working group with collaborators from five different institutions. This kind of effort ensures that the construct has broad validity, but not necessarily that it is valid for the entire community: other researchers might consider different characteristics as relevant. For the purpose of this study, we excluded Container-as-a-Service, such as applications using AWS Fargate or Google Cloud Run, which also fall under a broader definition of serverless computing [Kou+21]. While analyzing these types of applications could also yield interesting findings, we consider it outside the scope of this work. Serverless computing is an emerging technology, therefore it is possible and likely that the characteristics of serverless applications change within the next five years. However, the goal of this study is to provide a snapshot of the characteristics of serverless applications at the time of writing. We include a detailed replication package that enables the faithful replication of this study at a later time, which will allow to draw conclusions about how the state of serverless applications changed. We analyzed consensus between our study and ten related studies and found many points of consensus and good (often high or very high) levels of agreement overall. However, to determine the level of agreement with existing studies, we could not use established meta-analysis techniques, as some related studies did not disclose essential information (for example, cohort size). Therefore, it is possible that the level of agreement we compute does not correctly reflect the actual degree of agreement between the studies. To account for this, we included a detailed breakdown of the study results and their comparison as part of our replication package, enabling the community to conduct other comparisons, independently.

4.1.4.3 External Validity

Our study was designed to cover applications from open source projects, academic literature, and industrial literature, but we cannot claim generalizability to all serverless applications. For open-source projects, we filtered non-trivial projects from the most popular open-source repository but might have missed projects published in other repositories. Our academic literature collection is based on a curated dataset on serverless literature. Our comparison study uses a similar methodology. Although we validated the resulting collections against our knowledge and a small set of test articles, the methodology does not guarantee validity: we might have missed more recent articles, or articles not found by our process and unknown to all collaborators. Applications from industrial literature mostly focus on provider-reported case studies, an existing collection of industrial applications, and sources known to the authors. Our scientific computing applications are limited to institutions in a single country, Germany. We only partially cover applications in industry and science, as many of them remain unpublished, and others provide insufficient details to conduct a meaningful review. Other studies, for example, on FaaS platforms [Eyk+19], suffer from the same limitations.

4.1.5 Summary

Despite the many perceived advantages of serverless computing, the characteristics of serverless applications are still not well understood. This presents a major roadblock for the adoption of serverless computing in practice. Therefore, this contribution furthers the understanding of serverless applications by systematically collecting the largest collection of serverless applications to date from open-source projects, academic literature, industrial literature, and scientific computing. In order to answer **RQ I.1** (*"What are common characteristics of current serverless applications?"*), we characterize each application from our collection, through a systematic and comprehensive pair-reviewing process, with regard to 16 characteristics. Then, **RQ I.2** (*"Is there a community consensus on the common characteristics of serverless applications?"*) is answered by comparing our results to ten, mostly industrial, web surveys, and datasets. Therefore, this contribution achieves **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*). The results of this study are discussed in detail in Section 7.1. This contribution has been published as a SPEC RG technical report [Eis+20c], an IEEE Software article [Eis+21b], and a Transactions on Software Engineering (TSE) article [Eis+21c].

4.2 Performance Variability of Serverless Applications

One of the key requirements for reliable performance tests is ensuring that an identical resource environment is used for all tests [Eis+20a]. However, with serverless applications, developers have no control over the resource environment. Worse yet, cloud providers expose no information to developers about the resource environment [Wan+18a]. Therefore, information such as the number of provisioned workers, worker utilization, worker version, virtualization stack, or underlying hardware is unavailable. Furthermore, cold starts (requests where a new worker has to be provisioned) are a widely discussed performance challenge [Lei+19; Eyk+18]. This begs the question if performance tests of serverless applications are stable.

In this section, we present an exploratory case study to address **Goal II** (*"Quantify the performance variability that serverless applications experience."*) using the serverless airline booking application, a representative, production-grade serverless application [Les19; Ser19]. First, we conduct multiple repetitions of performance tests under varying configurations to answer **RQ II.1** (*"How much performance variability do common serverless applications experience?"*) and then three daily measurements for ten months to investigate **RQ II.2** (*"Does the performance of serverless applications change over time?"*). This study is the first to provide quantitative data on the performance variability of serverless applications over long periods of time. Unlike the microbenchmarks and single-function applications often used by existing work, it analyses the performance behavior of a complex, realistic application. We further include a detailed replication package that enables the replication of our study on future configurations of serverless platforms.

The remainder of this section is structured as follows: Section 4.2.1 introduces the system under test, Section 4.2.2 details our measurement methodology, and Section 4.2.3 introduces the analysis plan for each research question we address in the study. Finally, Section 4.2.4 discusses the limitations and threats to the validity of this study. The results of this case study are discussed in Section 7.2.

4.2.1 Serverless Airline Booking

The Serverless Airline Booking Application (SAB)[1] is a fully serverless web application that implements the flight booking aspect of an airline on AWS. It was presented at AWS re:Invent as an example for the implementation of a production-grade full-stack app using AWS Amplify [Les19]. The SAB was also the subject of the AWS Build On Serverless series [Ser19]. Customers can search for flights, book flights, pay using a credit card, and earn loyalty points with each booking.

The frontend of the SAB is implemented using CloudFront[2], Amplify[3]/S3, Vue.js[4],

[1] https://github.com/aws-samples/aws-serverless-airline-booking
[2] https://aws.amazon.com/cloudfront/
[3] https://aws.amazon.com/amplify/
[4] https://vuejs.org/

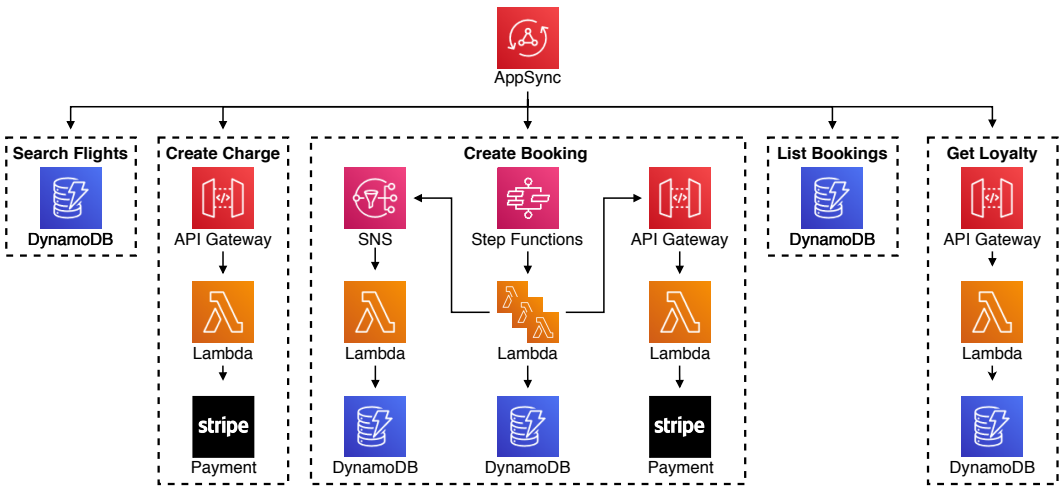


Figure 4.3: Architecture and API endpoints of the serverless airline booking application.

the Quasar framework[5], and Stripe Elements[6]. This frontend sends GraphQL queries (resolved using AWS AppSync) to five backend APIs, as shown in Figure 4.3:

- The *Search Flights* API retrieves all flights for a given date, arrival airport, and departure airport from a DynamoDB table using the DynamoDB GraphQL resolver.

- The *Create Charge* API is implemented as an API gateway that triggers the execution of the *CreateStripeCharge* lambda function, which manages the call to the Stripe API.

- The *Create Booking* API reserves a seat on a flight, creates an unconfirmed booking and attempts to collect the charge on the customer's credit card. If successful, it confirms the booking, and awards loyalty points to the customer. In case the payment collection fails, the reserved seat is freed again, and the booking is canceled. This workflow is implemented as an AWS Step Functions workflow that coordinates multiple lambda functions. The functions *ReserveBooking* and *CancelBooking* directly modify DynamoDB tables, the *NotifyBooking* function publishes a message to SNS, which is later consumed by the *IngestLoyalty* function that updates the loyalty points in a DynamoDB table. The *CollectPayment* and *RefundPayment* functions call the Stripe backend via an application from the Serverless Application Repository.

[5]https://quasar.dev/
[6]https://stripe.com/en-de/payments/elements

- The *List Bookings* API retrieves the existing bookings for a customer. Similar to the *Search Flights* API, this is implemented using a DynamoDB table and the DynamoDB GraphQL resolver.

- The *Get Loyalty* API retrieves the loyalty level and loyalty points for a customer. An API Gateway triggers the lambda function *FetchLoyalty*, which retrieves the loyalty status for a customer from a DynamoDB table.

We selected the SAB for our case study after investigating the serverless applications we collected in Section 4.1. We chose the SAB over other potential applications due to its comparatively large size and its usage of many different managed services. It is also running on AWS, the by far most popular cloud provider for serverless applications (see Section 4.1), and it uses both Python and JavaScript to implement the serverless functions, the two most popular programming languages for serverless applications (see Section 4.1).

4.2.2 Measurement Methodology

We deploy the frontend via Amplify [7] and the backend services via either the Serverless Application Model[8] or CloudFormation[9] templates depending on the service. The serverless nature of the application makes it impossible to specify the versions of any of the used services, as DynamoDB, Lambda, API Gateway, Simple Notification Service, Step Functions, and AppSync all do not provide any publicly available version numbers.

For the load profile, customers start by querying the *Search Flights* API for flights between two airports. If no flight exists for the specified airports and date, the customer queries the *Search Flights* API again, looking for a different flight. We populated the database so that most customers find a flight within their first query. Next, they call the *Create Charge* API and the *Create Booking* to book a flight and pay for it. After booking a flight, each customer checks their existing bookings and loyalty status via the *List Bookings* API and the *Get Loyalty* API. This load profile is implemented using the TeaStore load driver [Kis+18].
Based on some initial test runs, the SAB was only able to serve about 50 requests per second with our load profile. For it to serve at least 500 requests per second we made the following changes:

- **Stripe integration** The Stripe API test mode has a concurrency limit of 25 requests. After contacting the support, we adapted the application to distribute the requests to the Stripe API across multiple Stripe keys. Additionally, we

[7] https://aws.amazon.com/amplify/
[8] https://aws.amazon.com/serverless/sam/
[9] https://aws.amazon.com/cloudformation/

observed several request timeouts from the Stripe API. Therefore, we reconfigured the Stripe integration to timeout and retry long-running requests, which significantly reduced the number of failed requests.

- **Lambda concurrency limit** Each AWS account comes with many so-called service quotas, which limit, for example, the number of VMs an account can create. For serverless offerings, such as DynamoDB or Lambda, the service quotas limit the maximum throughput/concurrent request to provide an upper limit for the costs a serverless application can incur as unlike auto scaling groups, there is no upper limit for the scaling of serverless offerings. We requested an increase of the Lambda concurrent executions service quota from the default of 1.000 to 5.000, which was granted after about two days.

- **Systems Manager Parameter Store** Some Lambda functions retrieve configuration parameters from the Systems Manager Parameter Store for every request. At higher load levels, the default transactions per second limit of the Systems Manager Parameter Store started to become a limiting factor. First, we implemented caching for these configuration parameters, but the initial wave of cold starts still exceeded the default transaction limit. Therefore, we enabled the higher throughput option of the System Manager Parameter Store, which increases the throughput to up to 1.000 requests per second.

- **Step Functions Workflows** In our initial test runs, Step Functions was responsible for 60-70% of the incurred costs. AWS introduced Step Functions Express Workflows[10] in 2019. Step functions express workflows use the same Amazon States Language Specification as standard step function workflows but are tailored towards cost-effectively orchestrating short running workflows. Switching to an express workflow reduced the step functions costs by about 95-99% while maintaining the same functionality.

In terms of monitoring data, we collect the response time of each API call via the load driver. Additionally, we collect the duration, so the execution time of every lambda function. We exclude the duration of the lambdas *ChargeCard* and *FetchLoyalty*, as the response times of the APIs *Create Charge* and *Get Loyalty* mostly consist of the execution times of these lambdas. We cannot collect any resource-level metrics such as utilization or number of provisioned workers, as AWS and most other major serverless platforms do not report any resource-level metrics.

For our experiments, we perform measurements with 5 req/s, 25 req/s, 50 req/s, 100 req/s, 250 req/s, and 500 req/s to cover a broad range of load levels. Additionally, we vary the memory size of the lambda functions between 256 MB, 512 MB, and 1024 MB, which covers the most commonly used memory sizes [Dat20]. For each measurement, the SAB is deployed, put under load for 15 minutes, and then

[10] https://aws.amazon.com/2019/12/introducing-aws-step-functions-express-workflows/

torn down again. We perform ten repetitions of each measurement to account for cloud performance variability. Additionally, we run the experiments as randomized multiple interleaved trials, which have been shown to further reduce the impact of cloud performance variability [AB17]. To minimize the risk of manual errors, we fully automate the experiments (for further details we refer to our replication package). These measurements started on July 5th, 2020, and continuously ran until July 17th, 2020.

Additionally, we set up a longitudinal study that ran three measurement repetitions with 100 req/s and 512 MB every day at 19:00 from Aug 20th, 2020 to Jun 20th, 2021. The measurements were automated by a Step Functions workflow that is triggered daily by a CloudWatch alarm and starts the experiment controller VM, triggers the experiment, uploads the results to an S3 bucket, and shuts down the experiment controller VM again. We stopped the longitudinal study on Jun 20th, 2021, as AWS removed the Python 2.7 runtime, which is used in some functions of the SAB, on this date[11].

To ensure reproducibility of our results, the fully automated measurement harness and all collected data from these experiments are available in our replication package.[12]

4.2.3 Research Questions and Analysis Plan

In the following, we introduce and motivate the two research questions we aim to address in this case study and the corresponding analysis plans.

RQ II.1 How much performance variability do common serverless applications experience?

A common goal of performance tests is to measure the steady-state performance of a system under a given workload. Hence, it is essential that practitioners understand how long it takes for serverless applications to reach stable performance (i.e., how long is the warm-up period) in order to plan the duration of their performance tests accordingly. Aside from the general aspects that influence the initial performance instability, such as the environment and application optimizations (e.g., CPU adaptive clocking and cache setup), serverless applications also encounter cold starts. A cold start occurs when a request cannot be fulfilled by the available function instances, and a new instance has to be provisioned to process the upcoming request. Cold starts can incur significantly higher response times [Wan+18a; Fig+18]. Hence, in this research question, we investigate how long is the warm-up period in our experiments and the role of cold starts in the stability of the warm-up and steady-state experiment phases. Further, practitioners have no way to ensure that two different

[11] https://aws.amazon.com/blogs/compute/end-of-support-for-python-2-7-in-aws-lambda/
[12] https://github.com/ServerlessLoadTesting/ReplicationPackage

Algorithm 4.1: Warm-up Period Identification Heuristic.

```
   Result: warmupInSeconds
 1 threshold = 0.01
 2 stable = False
 3 warmupInSeconds = 0
 4 global_mean = mean(ts)
 5 while stable == False do
 6 |   ts = remove5secs(ts) // Remove 5 seconds of data
 7 |   warmupInSeconds += 5
 8 |   new_mean = mean(ts)
 9 |   delta = abs((new_mean - global_mean) / global_mean)
10 |   if delta < threshold then
11 |   |   stable = True
12 |   else
13 |   |   global_mean = new_mean
14 |   end
15 end
```

performance tests are executed in similar resource environments, given that deployment details in serverless applications are hidden from developers. Hence, we study how the inherent variance in deployed serverless applications impacts the stability *between* performance tests.

Analysis Plan. To determine the duration of the warm-up period, we initially tried to use the MSER-5 method [WCS00], which is the most popular method to identify the warm-up period in simulations [MI04; HRD10]. However, this approach was not applicable due to the large outliers present in our data, a well-documented flaw of MSER-5 [SS06]. Therefore, we employ a heuristic to identify the warm-up period. Our heuristic, shown in Algorithm 4.1, gradually removes data from the beginning of the experiment in windows of five seconds and evaluates the impact of doing so on the overall mean results. If the impact is above a threshold (we used 1% in our experiments), we continue the data removal procedure. Otherwise, we consider the seconds removed as the warm-up period and the remainder as the steady-state phase of the performance test experiment. Similar to MSER-5 [WCS00], we label any measurement where the detected warm-up period is larger than 40% of the measurement as unstable. This regulation is necessary, as warm-up period detection approaches become unreliable once the steady-state period is not considerably longer than the warm-up period. In these scenarios, either longer measurements are required until a steady-state can be detected or the system under test never reaches a steady-state (e.g., due to a growing number of entries in a database).

To evaluate the impact of cold starts on the experiment stability, we analyze the distribution of cold start requests across the two phases of performance tests: warm-up period and steady-state period. Then, we evaluate the influence of cold start requests on the overall mean response time, considering only cold start requests that occurred after the warm-up period. To test for statistically significant differences, we use the unpaired and non-parametric Mann-Whitney U test [MW47]. In cases where we observe a statistical difference, we evaluate the effect size of the difference using the Cliff's Delta effect size [LFC03], and we use the following common thresholds [Rom+06] for interpreting the effect size:

$$\text{Effect size } d = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147 \\ small(S), & \text{if } 0.147 < |d| \leq 0.33 \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474 \\ large(L), & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

Note that not all request classes provide information about cold starts. This information is only available for the six lambda functions, as the managed services either do not have cold starts or do not expose them. Therefore, we report the cold start analysis for the following six request classes: *CollectPayment*, *ConfirmBooking*, *CreateStripeCharge*, *IngestLoyalty*, *NotifyBooking*, and *ReserveBooking*.

Next, we evaluate the variation of the mean response time across experiment runs and study the influence of experiment factors such as the load level and function size. We focus on evaluating the steady-state performance of performance tests. Hence, we discarded the data from the first two minutes of the performance test runs (warm-up period) and calculated the mean response time for the steady-state phase, that is the remaining 13 minutes of experiment data. To evaluate the stability of the mean response time across runs, we first exclude outliers within an experiment that fall above the .99 percentile. Then, we calculate the coefficient of variation of the response time across the ten repetitions, per workload level and function size. The coefficient of variation is the ratio of the standard variation to the mean and is commonly used as a metric of relative variability in performance experiments [CSL20; LC16]. Similarly, we test statistically significant differences using the Mann-Whitney U test [MW47] and assess the effect size of the difference using the Cliff's Delta effect size [LFC03].

RQ II.2 Does the performance of serverless applications change over time?

The first research question focuses on the stability of performance tests conducted within the same time frame. However, the opaque nature of the underlying resource environments introduces an additional challenge: the underlying resource environment may change without notice. This might result in both short-term performance fluctuations (e.g., due to changing load on the platform) or long-term performance

changes (e.g., due to software/hardware changes). Therefore, in this research question, we conduct a longitudinal study on the performance of our SUT, to investigate if we can detect short-term performance fluctuations and long-term performance changes.

Analysis Plan. We analyze the results of our longitudinal study, which consists of three measurement repetitions with 100 requests per second and 512 MB memory size every day for ten months. First, to determine if there are any significant changes in the distribution of the measurement results over time, we employ the change point detection approach from Daly et al. [Dal+20]. To reduce the sensibility to short-term fluctuations, we use the median response time of the three daily measurements and configure the approach with $p = 0$ and 100,000 permutations. Second, upon visual inspection, it seemed that the variation between the three daily measurement repetitions was less than the overall variation between measurements. To investigate this, we conducted a Monte Carlo simulation that randomly picks 100,000 pairs of measurements that were conducted on the same day and 100,000 measurement pairs from different days. We calculated and compared the average variation between the sample pairs from the same day and from different days. Finally, to investigate if the observed performance variation could be misinterpreted as a real performance change (regression), we conducted a second Monte Carlo simulation. We randomly select two sets of ten consecutive measurements that do not overlap and test for a significant difference between the pairs using the Mann–Whitney U test [MW47]. For each detected significant difference, we calculate Cliff's Delta [LFC03] to quantify the effect size. Similar to our first Monte Carlo simulation, we repeat this selection and comparison 100,000 times. Further implementation details are available in our replication package.[13]

4.2.4 Limitations & Threats to Validity

This section first introduces the limitations of our study and then discusses the threats to validity that arise from these limitations. We consider the following to be the main limitations of this study:

- **Single system under test.** This study uses only a single system under test, the serverless airline booking application.
- **Single cloud platform.** This study is limited to AWS and does not consider any other cloud providers.
- **Constant load.** This study does not investigate the impact of varying load patterns as the experiments all use constant load.

[13] https://github.com/ServerlessLoadTesting/ReplicationPackage

- **Black-box view.** As this study is conducted on a public cloud, it is limited to the metrics exposed by the cloud provider.

In the following, we discuss the threats to the construct, internal, and external validity that arise from these limitations [Woh+12]. Construct validity examines the relation of the measurements to the proposed research questions. Internal validity examines the trustworthiness of the cause-and-effect relationship, that is the existence of alternative explanations for findings, and external validity considers how well the results can be generalized.

4.2.4.1 Construct Validity

In our experiments, we measured only the response time and function execution time; other metrics might show different effects. Out of the commonly used performance metrics, we did not consider CPU utilization and throughput. However, measuring the throughput is unusual for serverless applications due to their built-in scalability, and CPU utilization is currently not exposed by AWS. Further, we limited our experiments to performance tests with a constant load; performance tests with varying load might behave differently. Constant load is commonly used for performance tests, whereas varying load is more commonly used for load and stress testing. However, further research is required to understand the effects of performance tests under varying load.

4.2.4.2 Internal Validity

As the MSER-5 method for determining the duration of the warm-up period was not applicable to our data, we used a custom heuristic. It might be possible that this heuristic does not appropriately capture the length of the warm-up period. Based on a visual inspection of a large subset of the experiments, we found that the heuristic seems to capture the warm-up period well. Our replication package can be used to repeat this visual inspection.

Another threat to the validity of our results is that performance experiments in the cloud can suffer from a high degree of uncertainty. To mitigate this threat, we followed recommended practices for conducting and reporting cloud experiments [Pap+19] and used randomized multiple interleaved trials [AB17] to reduce measurement variability. Further, we provide a fully automated measurement harness that enables the replication of our measurements. For the longitudinal study, we perform three measurement repetitions each day at the same time to mitigate measurement variability, but we do not attempt to further control for performance variability as the study was intended to investigate the variability.

4.2.4.3 External Validity

Our case study used only a single system under test, which might limit the generalizability of our results. However, the serverless airline booking application is larger (uses more functions) than the average serverless application [Eis+21b; Sha+20], so independent parts of the application could also be considered multiple applications. Further, most of the properties we measure are more dependent on the underlying cloud platform than the application itself. However, it is possible that a different application, such as a scientific computing application with long-running functions might behave differently. While our experiments were conducted on one application only, our methodology is applicable to any application.

Another threat is that we conduct measurements on a single cloud platform. Although AWS is by far the most popular cloud provider for serverless applications, with 55%-70% of serverless applications running on AWS (see Section 4.2), further research is required to determine if our findings are transferable between cloud providers.

4.2.5 Summary

The performance variability of serverless applications over large periods of time is still poorly understood. In this contribution, we conducted an exploratory case study on the performance stability of serverless applications. First, we answered **RQ II.1** (*"How much performance variability do common serverless applications experience?"*) by analyzing multiple repetitions of performance tests under varying configurations. Next, we collected and analyzed a longitudinal dataset consisting of three daily measurements for ten months to answer **RQ II.2** (*"Does the performance of serverless applications change over time?"*). Together, this provides the first quantitative data on the performance variability of serverless applications over time and therefore achieves **Goal II** (*"Quantify the performance variability that serverless applications experience."*). The results of this study are discussed in detail in Section 7.2. This contribution has been published as an article in the Journal of Systems and Software (JSS) [Eis+22].

Chapter 5

Automating Operational Tasks of Serverless Applications

One of the key value propositions of serverless computing is that the cloud provider handles most operational tasks associated with running a serverless application. However, there are some serverless computing-specific operational tasks that put an additional burden on developers. As discussed in Challenge 3, developers of serverless functions are in charge of selecting how much resources are allocated to each worker instance. Selecting the optimal size of serverless functions is quite challenging, so developers often neglect it despite its significant cost and performance implications. Further, serverless workflows are commonly used to orchestrate multiple serverless functions. However, the pay-per-use model and the delayed, nontransparent reporting by cloud providers make it challenging to estimate the expected cost of workflow as outlined in Challenge 4.

Our second contribution introduces two approaches to address Challenge 3 and Challenge 4 by automating operational tasks associated with serverless applications. We introduce an approach to predict the optimal memory size of serverless functions based on monitoring data for a single memory size. We implement a synthetic function generator and construct a multi-target regression model to predict the impact of function size based on a large synthetic dataset. The second approach presented in this contribution addresses the cost optimization of serverless workflows. First, we use MDNs to accurately predict the response time and output parameter distributions of serverless functions. These individual function models are integrated into a workflow model and a Monte-Carlo simulation derives cost predictions for serverless workflows.

In the following, Section 5.1 introduces our approach for the automated serverless function size optimization and Section 5.2 introduces our workflow cost estimation approach.

5.1 Optimizing the Size of Serverless Functions

There is still a resource management task that serverless platforms leave to developers: *resource sizing*. Resource sizing is the task of selecting how much CPU, memory,

I/O bandwidth, etc. are allocated to a worker instance [Sin+20; AL19; Pir+15]. Most cloud providers implement the resource sizing of serverless functions as a configurable memory size, where other resources such as CPU, network, or I/O are scaled accordingly [Ser20b; Clo20]. Selecting an appropriate resource size is essential as it can often result in a faster execution at a lower cost. However, selecting an appropriate resource size is challenging. A recent survey revealed that 47% of the serverless functions in production have the default memory size, indicating that developers often neglect resource sizing [Dat20].

In this section, we introduce an approach to predict the optimal memory size of serverless functions based on monitoring data for a single memory size, which targets **Goal III** (*"Develop an automated method to optimize the size of serverless functions."*). First, we implement a serverless function generator capable of generating a large number of synthetic serverless functions by combining representative function segments. Next, we measure the execution time and resource consumption metrics of 2 000 synthetic functions for six different memory sizes on a public cloud, which addresses **RQ III.1** (*"How can a dataset on the impact of memory size for a vast number of functions be generated?"*). Finally, we answer **RQ III.2** (*"How can one predict the optimal size of serverless functions based on passive monitoring data?"*) by constructing a multi-target regression model to predict the execution time of a serverless function for previously unseen memory sizes based on the execution time and resource consumption metrics for a single memory size.

The remainder of this contribution is structured as follows: Section 5.1.1 gives an overview of the proposed approach, Section 5.1.2 describes our process for the generation of a large training dataset on the influence of varying memory sizes on different serverless functions, and Section 5.1.3 details how we use multi-target regression modeling and multi-objective optimization to determine the optimal memory size for previously unobserved serverless functions. Finally, Section 5.1.4 discusses the current limitations of the proposed approach.

5.1.1 Architecture Overview

Figure 5.1 gives a graphical overview of our approach to predict the optimal memory size of serverless functions based on monitoring data collected for a single memory size. During the offline phase, the *Synthetic Function Generator* creates many synthetic serverless functions, which are then instrumented with our *Resource Consumption Monitoring*. During the *Dataset Generation*, we run performance tests to obtain the resource consumption metrics and execution times for all memory sizes of thousands of synthetic serverless functions. By applying *Multi-target Regression Modeling* to the resulting dataset, we generate a performance model that can predict the execution time for all memory sizes of a real function based on monitoring data for a single memory size. The *Memory Size Optimization* utilizes these predictions to determine the optimal memory size. Our implementation of the proposed approach is limited to

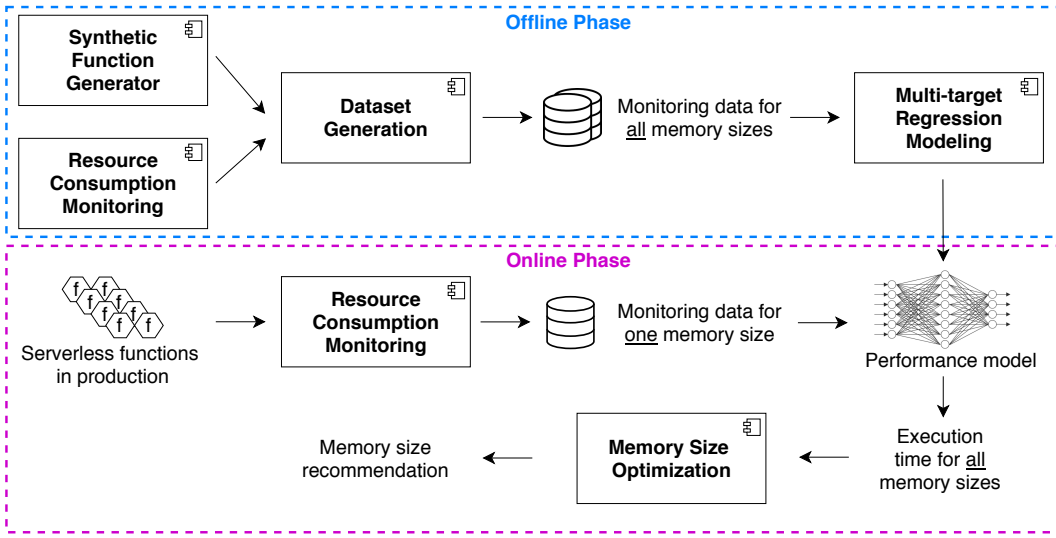


Figure 5.1: Overview of the proposed approach.

AWS Lambda and the language Node.js as they are by far the most common platform and programming language for serverless functions [Eis+21b; Dal20]. However, we are confident that it can be transferred to other platforms and programming languages.

5.1.2 Generation of a Large Training Dataset

In the following, we discuss how we generated a large training dataset describing the influence of varying memory sizes on different serverless functions. Section 5.1.2.1 describes the synthetic function generator, Section 5.1.2.2 explains our resource consumption monitoring, and Section 5.1.2.3 outlines our data collection process.

5.1.2.1 Synthetic function generator

Learning to derive how different memory sizes influence the execution time based on resource consumption metrics requires a large dataset covering a wide variety of different functions. Unfortunately, there are not enough easily benchmarkable open-source functions available [Eis+20c]. Therefore, we propose to generate synthetic serverless functions by combining representative function segments. Each function segment represents the smallest granularity of common tasks in serverless functions. Additionally, each function segment has to provide its own inputs to simplify load generation (e.g., a function segment that performs image manipulation comes with several images). Further, each function segment provides setup and teardown code of all external services it uses (e.g., databases or messaging queues).

For the selection of function segments, we investigated common tasks from the survey presented in Section 4.1. We implemented sixteen function segments covering, among other things, CPU-intensive tasks, image manipulation, format conversion, data compression, and interaction with files and external services, such as DynamoDB or S3. The function generator documentation from our replication package includes detailed descriptions of all implemented function segments[1]. While it is not possible to cover all possible functionality serverless functions can implement, our function segments should enable the generation of a large number of synthetic serverless functions with varying resource consumption profiles. In the future, the number of implemented segments can be easily extended if specific resource consumption profiles are required.

The function generator randomly combines these function segments and wraps them in a Lambda handler. The resulting deployment package and *template.yaml* file can be deployed as an AWS CloudFormation stack using the Serverless Application Model. Additionally, it generates scripts to set up and tear down the required services from the segments. The function generator keeps a list of already generated function hashes to ensure that no function is generated twice. For additional details on the function generator implementation, we refer to the documentation in our replication package.

5.1.2.2 Resource consumption monitoring

We propose to predict the execution time of serverless functions for different memory sizes based on the resource consumption metrics and execution time for a single memory size. However, Lambda currently does not support monitoring of resource consumption metrics out of the box. In general, Lambda's monitoring capabilities are quite lacking, which spawned several third-party monitoring solutions (e.g., by Epsagon[2], Datadog[3], and Dynatrace[4]). However, these third-party monitoring solutions focus on tracing requests across a serverless application. If they include resource consumption metrics, they are limited to basic metrics such as CPU utilization.

Therefore, we implement custom resource consumption monitoring to cover a wide variety of resource consumption metrics. Table 5.1 gives an overview of the metrics our resource consumption monitoring can collect and how the metrics are obtained. The execution time is monitored by timing the execution of the monitored function. A set of resource consumption metrics can be obtained via the Node.js library `'Process'`. This includes information about the CPU time consumed by user and system processes, the number of voluntary and involuntary context switches, the

[1] `https://github.com/Sizeless/ReplicationPackage`
[2] https://epsagon.com/
[3] https://www.datadoghq.com/
[4] https://www.dynatrace.com/

Table 5.1: Metric sources and collected metrics

Metric Name	Metric Source
Execution time	process.hrtime()
User CPU time	process.cpuUsage()
System CPU time	process.cpuUsage()
Vol Context Switches	process.resourceUsage()
Invol Context Switches	process.resourceUsage()
File system reads	process.resourceUsage()
File system writes	process.resourceUsage()
Resident set size	process.memoryUsage()
Max resident set size	process.resourceUsage()
Total heap	process.memoryUsage()
Heap used	process.memoryUsage()
Physical heap	v8.getHeapStatistics()
Available heap	v8.getHeapStatistics()
Heap limit	v8.getHeapStatistics()
Allocated memory	v8.getHeapStatistics()
External memory	process.memoryUsage()
Bytecode metadata	v8.getHeapCodeStatistics()
Bytes received	/proc/net/dev/
Bytes transmitted	/proc/net/dev/
Packages received	/proc/net/dev/
Packages transmitted	/proc/net/dev/
Min event loop lag	perf_hooks
Max event loop lag	perf_hooks
Mean event loop lag	perf_hooks
Std event loop lag	perf_hooks

number of times the file system had to perform I/O, information about the resident set, and the heap usage. Additional information about the heap is collected from the underlying V8 JavaScript engine. Information about the bytes and packages received and transmitted via the network is read using Linux counters in `/proc/net/dev/`. Finally, the Node.js event loop is monitored via the `perf_hooks` library. Within a traditional VM or container, many additional resource consumption metrics could be collected. However, many of these metrics are either unavailable within serverless functions, always return zero, or return unrealistic values [Wan+18b].

To implement the resource consumption monitoring, we employ a wrapper-style approach, where the monitoring itself implements the Lambda entry point. Whenever the Lambda is triggered, we log the initial metric values for all monitored metrics. Next, the entry point of the original monitored Lambda is called, resulting

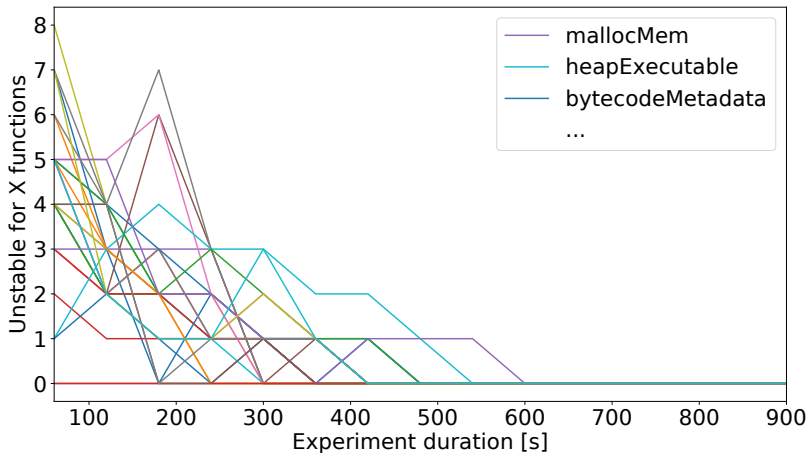


Figure 5.2: Number of functions for which each metric is unstable for with different measurement duration.

in a normal execution of this function. After the monitored function is finished, the metrics are polled again and the difference between before and after the function execution is calculated. The resulting metric values are written to a DynamoDB table. This call to DynamoDB does not affect the collected resource consumption metrics, as it occurs after the metric collection is finished. Finally, the response from the monitored function is returned.

This wrapper-style approach has been the best practice to monitor Lambda functions since the inception of this service [Cui20], even though it creates a slight performance overhead. However, note that this overhead does not impact the execution time measurements since we only measure the inner function execution. Recently, AWS previewed Lambda Extensions, which are processes that can run concurrently to the Lambda execution, similar to sidecars for containers [Woo20]. Lambda Extensions are mostly targeted at making the monitoring of Lambda functions more efficient, and accordingly, the resource consumption monitoring could be reimplemented as a Lambda extension once they reach general availability.

5.1.2.3 Data Collection

In this work, we conducted extensive performance experiments to create a large dataset describing how different functions are impacted by different memory sizes. Towards this goal, we randomly generated 2 000 different synthetic functions using our synthetic function generator and equipped them with our resource consumption monitoring. To manage the large number of required performance measurements, we implemented a fully automated measurement harness in Go that relies on Vegeta[5]

[5] `https://github.com/tsenart/vegeta`

as a load driver and enables parallelization of the experiments. For the implementation details of the measurement harness, we refer to our replication package.

Before we generate the dataset, we need to determine how long each performance experiment needs to run until the reported metrics are stable. To investigate this, we generated 50 functions and measured their execution time and resource consumption metrics for fifteen minutes at 30 requests per second. Next, we tested for each collected metric if the samples across all requests within the first minute, first two minutes, first three minutes, and so on come from the same distribution as the values collected during the full experiment. Figure 5.2 shows the results when using the Mann–Whitney U test [MW47] to test for similarity. We can see that even after one minute, all metrics are already stable for over 80% of the investigated functions. If we apply Cliff's delta [Cli93] for the differences observed after one minute, all differences are already considered negligible. After ten minutes of experiment time, `mallocMem` is the last metric to become stable for all functions according to the Mann-Whitney U test. We select ten minutes as the experiment duration for the dataset generation to ensure that stable metrics are collected.

Finally, we used the measurement harness to measure the execution time and resource consumption metrics for 2 000 functions across six different memory sizes (128MB, 256MB, 512MB, 1024MB, 2048MB, 3008MB), including the smallest and largest available memory sizes on AWS for ten minutes each at 30 requests per second with an exponentially distributed inter-arrival time. This amounts to 12 000 performance measurements, 120 000 minutes of experiment time, 216 000 000 Lambda executions, and roughly $2 000 worth of Lambda compute time. The resulting dataset is publicly available as part of our replication package.

5.1.3 Determining the Optimal Function Size

In the following, we describe our approach to determine the optimal function size using multi-target regression modeling (Section 5.1.3.1) and multi-objective optimization (Section 5.1.3.2).

5.1.3.1 Multi-target regression modeling

We formulate the task of predicting the execution time of a serverless function across all memory sizes based on monitoring data from a single memory size as a multi-target regression problem. Hence, for each monitored memory size (*base memory size*) we train a regression model that predicts the remaining five memory sizes (*target memory sizes*) based on the average monitored execution time and average resource consumption metrics. To equalize the scale of the target variables, we express all target execution times as ratios of the input execution time.

We start out with a simple neural network with three layers of 128 neurons trained for 200 epochs. Using this model, we conduct several feature engineering and selection rounds, inspired by [Gro+19b]. The first round of feature selection

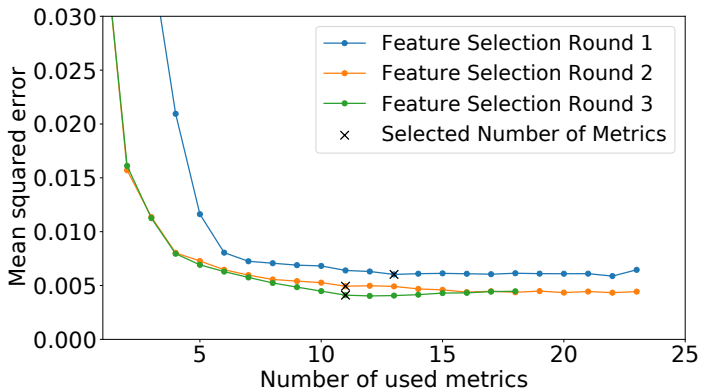


Figure 5.3: Accuracy and selected metrics for three sequential forward feature selection rounds.

uses our initial features F_0, that is the mean execution time and the mean of each resource consumption metric. Figure 5.3 shows that the accuracy increases until we reach thirteen features, so we discard all remaining metrics for the reduced feature set F_1. Next, we construct relative features that normalize the F_1 features by execution length to obtain F_2. For example, in addition to the total number of context switches, F_2 also contains the context switches per second. Running the sequential forward feature selection again (see Figure 5.3) shows an increased model accuracy. We again decrease the number of features in F_3 by selecting only the eleven most promising features from F_2. Finally, for each remaining metric in F_3, we add the standard deviation and coefficient of variation and run the feature selection for the third time to receive the final feature set F_4. This results in only a slight accuracy increase, but further reduces the number of base metrics required as all eleven final metrics in F_4 are calculated using the metrics heap used, user CPU time, system CPU time, voluntary context switches, bytes written to file system, and bytes received over the network. Therefore, our approach requires monitoring only these six metrics when applied in practice.

In order to explore the inner workings of our model and therefore to understand how different characteristics influence the scaling of a function with additional resources, we employ partial dependence plots. A partial dependence plot shows the marginal effect of a feature on the model prediction, which makes it a common explainability tool for machine learning models [Gol+15]. Figure 5.4 shows the partial dependence plots for the six most impactful features of our model for a base size of 128MB. It shows that the relative time spent in user and system space (i.e., the CPU utilization caused by the function) have the largest impact on the scaling behavior, with a higher CPU utilization resulting in higher expected speedups for increased memory size. This is in line with current assumptions that CPU-intensive functions

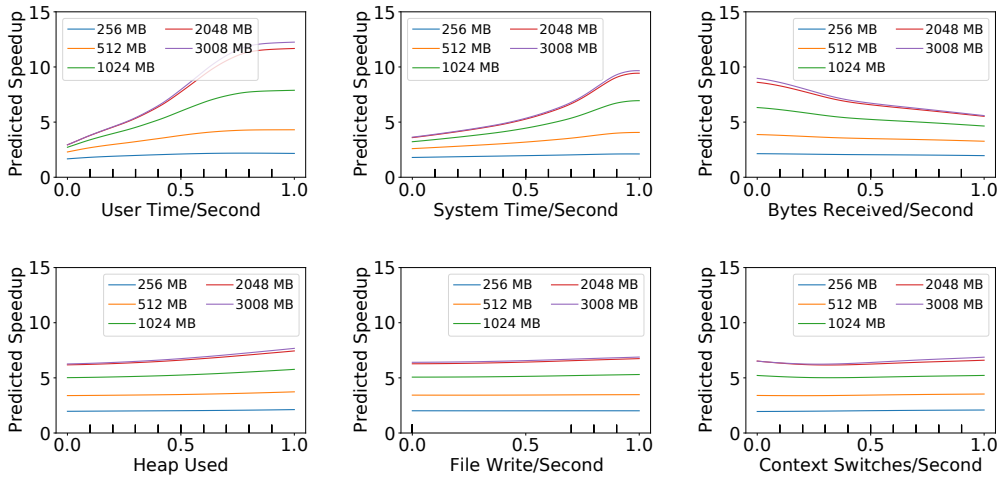


Figure 5.4: Partial dependence plots of the model with basesize 128MB, showcasing the impact of six most important input features on the predicted speedup for different memory sizes. X-axis values are scaled to [0,1]. The partial dependence plots for the remaining features can be found in our replication package.

benefit the most from larger memory sizes [Cas20b]. Interestingly, the number of bytes received per second correlates negatively with the predicted speedup, so a function that is network-intensive will scale worse with larger memory sizes. Further, the heap used (so the memory used by the application) also has an impact on the predicted speedup. If a function uses a lot of its available memory, adding additional memory would reduce memory swapping, whereas for a function that already has sufficient memory available additional memory would not be beneficial. While the remaining features also impact the final prediction, they seem to be mostly used to fine-tune the prediction. To summarize, we find that the predicted speedup mostly depends on the CPU utilization, network activity, and memory used by the function.

Next, we conduct a grid search to tune the hyperparameters of the model. Table 5.2 shows the tuned parameters, the ranges for each parameter, and the parameter value selected by the grid search. The final model uses the Adam optimizer, a MAPE loss function, 200 epochs, an L2 regularization of 10^{-2}, and four layers. With about three minutes, the training overhead is negligible.

As our approach relies on monitoring data from the function running in production with a single memory size, we investigate if a certain monitored memory size provides better prediction accuracy than others, as it would then make sense to deploy the function with this memory size initially. Therefore, we run ten iterations of five-fold cross-validation with a random split for each base memory size. Table 5.3 shows the resulting mean squared error, mean absolute percentage error, coefficient of determination (R^2), and explained variance score, which are common metrics to

Table 5.2: Parameter range and selected parameters for the hyperparameter optimization.

Parameter	Parameter range	Selected
Optimizer	SGD, Adam, Adagrad	Adam
Loss	MSE, MAE, MAPE	MAPE
Epochs	200, 500, 1000	200
Neurons	64, 128, 256	256
L2	0, 0.0001, 0.001, 0.01	0.01
Layers	2, 3, 4, 5	4

Table 5.3: Mean Squared Error (MSE), MAPE, R^2, and explained variance for each base memory size (in MB) based on cross-validation.

Basesize	128	**256**	512	1024	2048	3008
MSE	0.005	**0.003**	0.004	0.009	0.010	0.015
MAPE	0.066	**0.046**	0.040	0.031	0.033	0.036
R^2	0.986	**0.977**	0.971	0.970	0.954	0.958
ExpVar	0.987	**0.979**	0.974	0.972	0.962	0.963

determine the performance of regression models [GS01]. We select 256MB as the default basesize, as it shows the best mean squared error, the second-highest R^2 and explained variance scores, and a good mean absolute percentage error.

5.1.3.2 Memory size optimization

Automatically determining the optimal memory size for a serverless function results in a standard multi-objective optimization problem, as we want to optimize for both performance and cost. A common approach to determine a single, optimal solution for multi-objective optimization problems is to use a parameterizable tradeoff function that combines the objectives into a single score [Rao19]. This requires scores of the same scale for each objective, which we calculate for each memory size m_x as follows:

$$S_{cost}(m_x) = \frac{\text{cost}(m_x)}{\min_{\forall m_i \in M} \text{cost}(m_i)},$$

$$S_{perf}(m_x) = \frac{\text{executionTime}(m_x)}{\min_{\forall m_i \in M} \text{executionTime}(m_i)}$$

with M being the set of all available memory sizes and the functions cost() and executionTime() returning the predicted cost and execution time for a given memory

size m_i. Note that for monitored memory sizes the observed values can be used. Both scores have a minimum of 1, which indicates an optimal cost/execution time, and each value above one indicates the percentage deviation from the optimum, so a S_{cost} of 1.5 would indicate a 50% increased cost compared to the lowest possible cost for this function. Therefore, both scores use the same scale and are humanly interpretable. For the final objective function, these scores are combined using a configurable tradeoff value $t \in [0, 1]$ as follows:

$$S_{total}(m_x) = t \cdot S_{cost}(m_x) + (1 - t) \cdot S_{perf}(m_x)$$

The value for t can be set by the system operator based on their preferences, where $t = 0.5$ would indicate that a one percent increase in cost compared to the best possible cost is worth the same as a one percent increase in execution time compared to the best possible execution time is worth the same. Whereas a value of $t = 0.75$ would result in accepting an X% increase in cost only if it would result in a 3X% decrease of execution time.

In order to apply this tradeoff function to optimize the memory size of serverless functions, we first use our model to predict the execution time for all memory sizes and then calculate the predicted cost for a function execution based on the pricing model of the cloud provider, as it only depends on the execution time. Next, we calculate S_{total} for each memory size and select the memory size with the lowest S_{total} score:

$$OptSize = \arg \min_{\forall m_x \in \{128,256,512,1024,2048,3008\}} S_{total}(m_x)$$

5.1.4 Limitations & Threats to Validity

While we consider our approach a significant improvement over the current state-of-the-art, there still are limitations and threats to the validity to be discussed.

First, we limited the problem along two dimensions to reduce the cost associated with generating the synthetic dataset. AWS actually supports adjusting the memory size from 128MB to 3008MB in 64MB increments, whereas the dataset in this paper is limited to only six different memory sizes. However, the approach from [Ali+20] could be used to interpolate the values for the 64MB increments. Second, our approach currently only supports a single cloud provider and a single programming language. Transferring the approach to other providers and languages does not pose any conceptual challenges and only requires an extension of the synthetic dataset as no Node.js-specific metrics were used.

Second, we did not evaluate the performance overhead caused by our resource consumption monitoring. While this overhead does not impact the measured metrics and execution times, it might hinder adoption in practice. However, there is already a commercial monitoring solution that tracks the CPU usage and network activity

of serverless functions using Lambda Extensions, which indicates that the overhead of monitoring system-level metrics for serverless functions is reasonable [Cui20]. Therefore, monitoring six system-level metrics for about ten minutes should not cause any issues.

Third, a shift in the workload of an application can change the performance properties of a serverless function, for example, the workload becomes substantially burstier, which causes more cold starts, or the payload size increases, which causes longer execution times. These workload shifts would also change the resource consumption metrics, so our model could be used to predict the optimal memory size for the changed function behavior again. To alleviate this, the input-sensitive performance models proposed in Section 5.2 could be integrated with the approach from this contribution.

Finally, serverless platforms are introducing new features and performance improvements regularly, which raises the question of the longevity of our model. In our evaluation of this contribution (see Section 8.1), the measurements for the hello retail application were conducted nine months after the training dataset was created and there is no significant deterioration in prediction accuracy pointing. However, there might be breaking changes on the provider side that would invalidate our model. To avoid having to regenerate the full training dataset, one could explore transfer learning techniques that freeze the initial layers of our model and retrain only with a much smaller new dataset.

5.1.5 Summary

Developers of serverless applications are still in charge of is resource sizing, that is selecting how much resources are allocated to each worker. In this contribution, we introduced an approach to predict the optimal resource size of serverless functions using monitoring data of a single memory size. First, we answered **RQ III.1** (*"How can a dataset on the impact of memory size for a vast number of functions be generated?"*) by introducing a synthetic function generator and a resource consumption monitoring approach. Using these, we generated a large dataset on how functions with different resource consumption behavior scale with increasing memory sizes. Based on this dataset, we answered **RQ III.2** (*"How can one predict the optimal size of serverless functions based on passive monitoring data?"*) by training a multi-target regression model capable of predicting the execution time of a serverless function for all memory sizes based on monitoring data for a single memory size. These predictions then enable the automated optimization of a serverless function's memory size. Unlike existing approaches based on performance testing, our approach only requires monitoring data that can be collected in production as opposed to dedicated performance tests, which shows that we have achieved **Goal III** (*"Develop an automated method to optimize the size of serverless functions."*). For developers, this removes the effort required to implement and maintain representative performance

tests. For cloud providers, it enables memory size recommendations, similar to the AWS Compute Optimizer for virtual machines [Ser20a], which so far was infeasible as cloud providers cannot run performance tests on user functions. The prediction accuracy, memory size optimization, and the potential speed-ups/cost savings are evaluated using four realistic serverless applications in Section 8.1. This contribution has been published as a full research paper at the International Middleware Conference (MIDDLEWARE) [Eis+20b], which has received the best student paper award.

5.2 Optimizing the Cost of Serverless Workflows

Many major cloud providers use the same cost model for serverless functions, where the cost of a function execution depends on: i) the response time of a function rounded up to the nearest 100 ms[6], ii) the memory allocated to the function, and iii) a static charge for every invocation [AC17a]. While many organizations report significant cost savings by switching from traditional hosting options to serverless solutions [IBM17; Wil17; AC17a], an inhibiting factor for the adoption of serverless solutions in practice is the difficulty of estimating the expected costs of serverless functions and workflows [Eyk+18; AC17a; BA18]. A reason for this is that, in contrast to traditional hosting options, the cost of a function depends directly on its input parameters—since the response time distribution of a function depends on its input parameters. For example, the time required to resize an image depends on its original size. Therefore, the cost of resizing an image depends on its original size as well. This is exacerbated in workflows, where function outputs are often propagated to succeeding functions. Hence, the cost and response time of functions contained within a workflow can be erratic, which makes predicting the cost for the overall serverless workflow challenging.

In this section, we propose a methodology for the cost prediction of serverless workflows to address **Goal IV** (*"Provide a technique to estimate the costs of serverless workflows."*). First, we apply machine learning to predict the response time and output parameter distributions for individual serverless functions. Standard regression techniques, such as SVR, MARS, or random forest, can only be used to predict the mean response time of a function. However, accurate cost estimations require the prediction of response time distributions, because many cloud providers round the billed execution time up to the nearest 100 ms. Therefore, we show how MDNs can be used to accurately predict the response time and output parameter distributions of serverless functions, which answers **RQ IV.1** (*"How can the execution time distribution of a serverless function be predicted based on its input parameters?"*). These individual function models are composed to a workflow model that describes the parameter relationships within the workflow. Finally, a Monte-Carlo simulation traverses the workflow model and samples distributions from the individual functions models to derive cost predictions for serverless workflows and thereby addresses **RQ IV.2** (*"Can the impact of restructuring a serverless workflow on its cost be predicted?"*).

The remainder of this section is structured as follows: Section 5.2.1 gives an overview of the proposed approach. Next, Sections 5.2.2 and 5.2.3, go in-depth on the prediction of function response time and output parameter distributions and our approach to derive cost estimates for serverless workflows based on the individual function models. Finally, Section 5.2.4 discusses the limitations and threats to the

[6] AWS recently switched to rounding to the nearest millisecond, but this pricing model is still common among the other cloud providers.

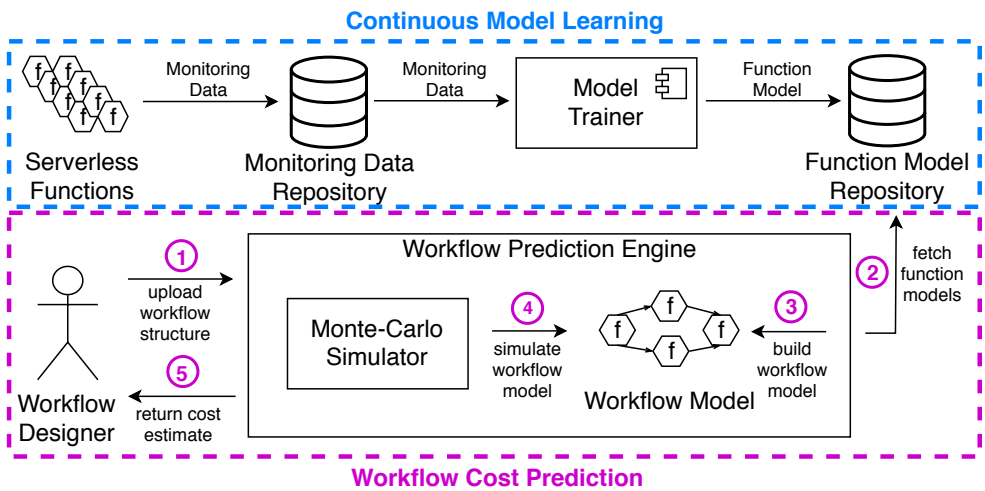


Figure 5.5: Overview of the proposed approach for the cost prediction of serverless workflows.

validity of the proposed approach. The prediction accuracy of our approach and the potential cost savings are evaluated in Section 8.2.

5.2.1 Architecture Overview

The proposed approach shown in Figure 5.5 can be separated into two phases, the continuous model learning process, and the workflow cost prediction process. During the continuous model learning process, the existing `Serverless Functions` are monitored. For any functions that are not already deployed in production, microbenchmarks can be used to generate monitoring data [BA18]. The resulting monitoring data is stored in a `Monitoring Data Repository` (e.g., Prometheus, InfluxDB, or a managed monitoring solution from the cloud provider). Periodically, the `Model Trainer` is triggered to train models that describe the response time and output parameter distributions of the serverless functions based on their input parameters, which is discussed in detail in Section 5.2.2. As the `Model Trainer` could make use of GPU-based acceleration during the model learning, it could be deployed in a distributed data analytics cluster with GPU acceleration, such as a Spark or Hadoop cluster. The resulting models are then stored in the `Function Model Repository`, which due to the infrequent access pattern can be a cloud data storage, such as Amazon S3, Google Cloud Storage, or Azure Storage.

The workflow cost prediction process is triggered when a workflow designer uploads the workflow. Next, the `Workflow Prediction Engine` fetches the models for all functions contained in the workflow from the `Function Model Repository`. These function models are then composed into a `Workflow Model` based on the structure uploaded by the workflow designer. To derive cost predictions from

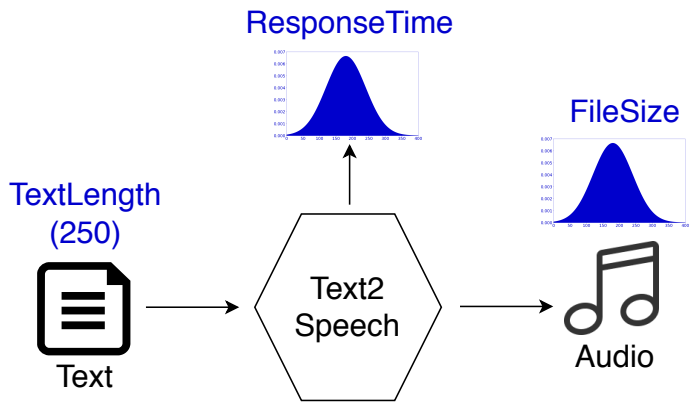


Figure 5.6: For multiple text segments of length 250, a distribution of response times and output file sizes can be observed for a function that transcribes text into speech.

the `Workflow Model`, the `Monte-Carlo Simulator` simulates the `Workflow Model`. Finally, the derived cost estimates are returned to the workflow designer. The `Workflow Prediction Engine` could be implemented as a serverless function, as it has an infrequent, potentially bursty access pattern and model inference rarely relies on GPU acceleration [Zha+18], which is currently not supported for serverless functions [Hel+18].

5.2.2 Response Time and Parameter Distribution Prediction

We train an individual model for the response time and for each output parameter of a serverless function based on monitoring data from the `Monitoring Data Repository`. This monitoring data contains the response time and parameterization for each request to the serverless function. Most machine learning techniques require numeric input, while the parameters of a function call are not necessarily numeric values. Examples of non-numeric values include strings, lists, binary data, etc. We do not address the task of creating numeric features based on this data, as there is extensive prior work targeting the automated extraction of numeric features based on function input parameters [Gro+19a; KKR10].

For the repeated execution of a serverless function with identical input parameter characteristics, a distribution of response times and output parameters can be observed. To illustrate this, we implemented and evaluated a function called *Text2Speech*, which transcribes text segments to speech, as shown in Figure 5.6. Transcribing multiple text segments with a length of 250 characters, we observe a distribution of the response time due to variation in the performance and saturation of the hardware executing the function. Additionally, we also observe varying values for the size of the resulting audio file. However, both the response time and the resulting file size are closely correlated to the length of the transcribed text segment.

Figure 5.7: Comparison between billed response time and mean response time of normal distribution.

Predicting the distribution of the response time of a serverless function is important to estimate the resulting costs. Predicting only the expected mean response time can lead to inaccurate cost predictions, as all major FaaS providers round the billed response time up to the nearest 100 ms. Figure 5.7 shows this for a simulated serverless function with a normally distributed response time with a mean of 180 ms and a standard deviation of 60 ms. If we would solely use the mean response time of 180 ms and round to the nearest 100 ms, we would predict that an execution of this function is billed for 200 ms on average. However, looking at the actual probabilities of being billed 100 ms (9.12%), 200 ms (53.93%), 300 ms (34.67%), and 400 ms (2.28%), results in a mean billed time of 230.11 ms. Therefore, accurate cost estimations for serverless functions and workflows require predicting the response time distribution instead of only the mean response time.

Common regression techniques, such as SVR, MARS, or random forest can only be used to predict the mean response time of a serverless function. Therefore, we propose the usage of so-called MDNs [Bis94]. Bishop et al. propose the idea to use a dense neural network to parameterize a Gaussian mixture model. A mixture model describes the probability density function of a random variable as a linear combination of m Gaussian kernels:

$$p(y|x) = \sum_{i=1}^{m} \alpha_i(x) * \phi_i(y|x) \tag{5.1}$$

with α_i as the mixing factor ($\sum_{i=1}^{m} \alpha_i(x) = 1$) and $\phi_i(y|x)$ as a Gaussian kernel with mean μ_i and standard deviation σ_i. Provided with a large enough number of kernels, a Gaussian mixture distribution can approximate any probability distribution with an arbitrary accuracy [Gea89]. Simply put, a Gaussian mixture distribution is the

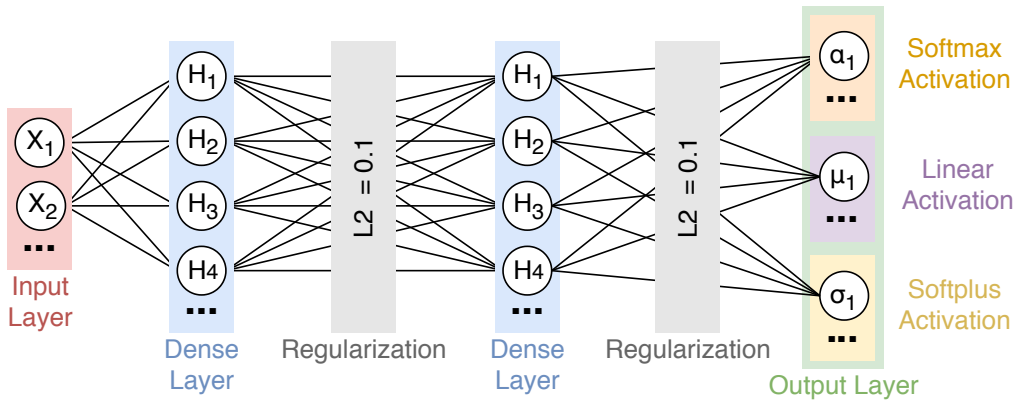


Figure 5.8: Mixture density network architecture for the prediction of response time and output parameter distributions of serverless functions with α as mixing coefficients, μ as kernel means, and σ as kernel standard deviations.

weighted average of m normal distributions. Parameterizing a Gaussian mixture distribution requires the weights, means, and standard deviations of the m normal distributions. In a mixture density network, these parameters are estimated using a dense neural network. As the weights, means, and standard deviations are not contained within the training data set, traditional loss functions for regression, such as MSE, Mean Squared Logarithmic Error (MSLE), or Mean Absolute Error (MAE) cannot be applied. Instead, most mixture density networks use the negative log-likelihood function as a loss function, which is defined as:

$$\ell(x) = -log(p(y|x)) \tag{5.2}$$

For each sample in a training batch, the logarithm of its occurrence likelihood is calculated and then negated, as neural network optimizers aim to minimize the loss function.

Figure 5.8 shows the network layout we propose to use for the prediction of the response time and output parameter distributions of serverless functions. It consists of an input layer, two dense hidden layers, two regularizations, and an output layer that aggregates over the three layers describing the mixing coefficients α_i, the means μ_i and the standard deviations σ_i of the gaussian kernels. The input layer contains a neuron for each input parameter, so the overall network has rather few input neurons. The output layer has a total of $m * 3$ neurons, but our evaluation showed that the prediction of the response time and output parameter distributions requires usually less than five kernels. Therefore, the total number of output neurons remains usually below fifteen. With limited input and output neurons, the dense layers require only a comparatively low number of neurons (200 were sufficient during our case study).

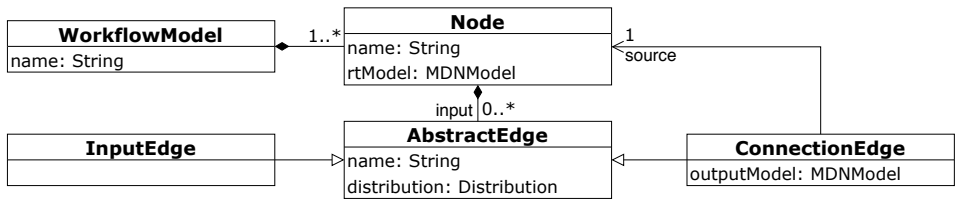


Figure 5.9: Meta-model for the workflow model.

Some input parameters have a large range of values, such as the file size. For such input parameters, it is possible to only have a single observation for a specific input parameter value. Additionally, the response time for serverless functions is prone to outliers due to function cold starts [Bal+17]. If such an outlier is the only sample for its input parameter value, the neural network will overfit by parameterizing the mixture distribution for this specific input parameter value much larger than for adjacent input parameter values. In order to prevent this type of overfitting, we apply L2 regularization after each dense layer. An L2 regularization (also known as ridge regularization or Tikhonov regularization [RN09]) adjusts the cost function for the gradient descent learning by adding the squared Euclidean norm of the corresponding layers weight matrix [CMR09]. Therefore, the L2 regularization penalizes model complexity. In our use case, this is a desirable property as we assume that the relationship between an input parameter and the observed response time distribution is roughly continuous.

The dense layers use the widespread rectified linear unit (relu) activation function [LBH15]. The output layer for the mixing coefficients uses the softmax activation function to guarantee that the mixing coefficients sum up to one. As no restrictions apply for the means of the linear kernels, the corresponding output layer uses a linear activation function. For the prediction of response time distributions, it could be restricted to positive values. However, there might be edge cases in which the distribution of an output parameter might contain negative values. As a standard deviation is restricted to values greater than or equal to zero, the corresponding output layer should also be restricted accordingly as otherwise the negative loss likelihood can no longer be calculated. Bishop et al. originally proposed the usage of an exponential activation function [Bis94]. However, this is reported to potentially lead to numerical instability [Bor15]. As alternatives, we tested the softplus activation function [GBB11] and an exponential linear unit (elu) activation function [CUH15] with an offset of 1. The convex nature of the softplus activation function enabled the network to fit linear kernels with a small standard deviation, whereas the elu + 1 activation function consistently skewed towards kernels with large standard deviations. Linear kernels with a small standard deviation are useful in mixture density networks to explain subpopulations. Therefore, we choose the softplus activation function for the standard deviation output layer. Finally, the three output

layers are concatenated to form a single output layer.

5.2.3 Workflow Cost Prediction

In Section 5.2.2, we propose the usage of mixture density networks to predict the response time and output parameter distribution of individual serverless functions for a concrete input parameter value. However, in a serverless workflow, the input parameters of each function are a distribution instead of a concrete value, because they are the output of previous functions. We propose to run a Monte-Carlo simulation [Vos08] on a model of the workflow to determine the response time and output parameter distribution for a given distribution of input parameters.

As shown in Figure 5.9, the proposed workflow model is an extended, directed acyclic graph (DAG), a common formalism to model workflows [VVI18]. As a simplification, we assume that control and data flow are identical. Each `WorkflowModel` consists of a number of `Nodes`. A `Node` represents a single execution of a serverless function and should be named after the function it represents. Every `Node` contains a number of `AbstractEdges`, which represent the input parameters to the function and also should be named accordingly. Each edge describes the name of a parameter and its corresponding distribution. There are two sub-classes of `AbstractEdge`, namely `InputEdge` and `ConnectionEdge`. An `InputEdge` represents an input to the workflow and characterizes the distribution of an input parameter to the first `Nodes` in the workflow. On the contrary, `ConnectionEdges` serve both as input parameters to nodes and output parameters from nodes. They characterize the output distribution of a return parameter of a node, which is usually an input parameter to another `Node` in the workflow. Therefore, `ConnectionEdges` contain an `MDNModel` that can predict the `Distribution` of the output parameter the `ConnectionEdge` represents, based on the input parameters of the corresponding `Node`. `ConnectionEdges` additionally reference the `Node` of which the output parameter originated from. Similarly, each `Node` contains a `MDNModel` that can be used to estimate the response time distribution of the serverless function represented by the `Node` based on its input parameters. Since all edges always describe the input parameters of nodes, it is possible to add output parameters that do not impact the cost of the workflow execution.

Algorithm 5.1 requires a `WorkflowModel` as an input and provides an estimation of the costs for executing this workflow. First, at lines 2-4 of Algorithm 5.1, all input distributions are solved by iterating over all input edges of all nodes of the workflow and recursively solving them. The `SOLVE` function described at lines 14-23 returns at line 16, if a given edge already has a distribution. This is the case for `InputEdges` for example, as they are already parameterized as input. However, if the distribution of an edge is unknown, the distribution of the edge can be estimated by applying the Monte-Carlo simulation on the given MDN model of the edge and the distributions of its input parameters. However, as the input parameters might also be unknown, all dependent edges are first solved by recursively calling SOLVE on them. This

Algorithm 5.1: Workflow Model Traversal

```
1  function ESTIMATECOSTS(workflowModel):
2      for edge in workflowModel.nodes.input do
3          SOLVE(edge)
4      end
5      workflowCost = 0
6      for node in workflowModel.nodes do
7          rDist = SIMULATE(node.rtModel, node.input)
8          functionCost = ESTIMATECOST(rDist)
9          workflowCost += functionCost
10     end
11     return workflowCost
12 End function
13
14 function SOLVE(edge):
15     if edge.distribution != NULL then
16         return
17     end
18     inputDists = edge.source.input
19     for dependency in inputDists do
20         SOLVE(dependency)
21     end
22     edge.distribution = SIMULATE(edge.model, inputDists)
23 End function
```

recursion is guaranteed to be finite, as DAGs are not allowed to contain circles and all input edges are already parameterized. After the recursion ends at line 5 of Algorithm 5.1, all input distributions to all nodes in the given workflow model are known. Hence, lines 6-11 iterate over all nodes in the workflow engine, and sums up the estimated cost for each predicted response time distribution. Finally, the total costs can be returned at line 11.

Algorithm 5.2 details how the Monte-Carlo simulation derives the distribution of response times or output parameters of a serverless function based on the distribution of its input parameters. It uses the mixture density networks described in Section 5.2.2, that predict the expected distribution for concrete input parameter values. In lines 5-9 of Algorithm 5.2, the algorithm draws a sample from the probability distribution of each input parameter.

Next, at lines 10-11, the mixture density model is used to predict the expected distribution for this set of input parameters, and the resulting distribution is added to a list. The more samples are used in a Monte-Carlo simulation, the more precise

Algorithm 5.2: Monte-Carlo Simulation

```
1  function SIMULATE(MDNModel, paramDists):
2      numSamples = 5000
3      resultDistList = new List()
4      for i = 1; i ≤ numSamples; i++ do
5          params = new List()
6          for param in paramDists do
7              sample = param.drawSample()
8              params.add(sample)
9          end
10         dist = MDNModel.predict(params)
11         resultDistList.add(dist)
12     end
13     return new MixtureDistribution(resultDistList)
14 End function
```

Algorithm 5.3: Cost Estimation

```
1  function ESTIMATECOST(respDist):
2      numSamples = 5000
3      sumCosts = 0
4      for i = 1; i ≤ numSamples; i++ do
5          sample = respDist.drawSample()
6          billedIntervals = CEIL(sample/BILLINGINTERVAL)
7          sumCosts += billedIntervals * CPUCOST
8          sumCosts += billedIntervals * MEMORYCOST
9          sumCosts += EXECUTIONCOST
10     end
11     return sumCosts/numSamples
12 End function
```

the resulting estimation becomes, at the cost of increased computation time. As rare events are not expected in our use case, 5,000 samples likely provide a sufficient prediction accuracy. The resulting list of probability distributions is then composed to a mixture distribution with equal weights, which can be seen as the average over the individual distributions.

Algorithm 5.3 shows the adapted Monte-Carlo simulation used to predict the average cost for a function estimation based on its response time distribution. First, 5,000 samples are drawn from the response time distribution of the serverless function. In line 6, the algorithm calculates the number of billed intervals by dividing

the response time sample by the size for the billing interval and rounding up. The number of billed intervals is then used for calculating the cost for the CPU time and the memory time, by multiplying them at lines 7 and 8 of Algorithm 5.3. Some cloud providers do not split the costs of CPU time and memory time; in this case lines 7 and 8 can be concatenated and replaced by just one multiplication with the charged amount per interval. Additionally, each sample is charged a constant blanket fee per execution, that is, the invocation cost for each function execution. After calculating all samples, the costs for each sample are summed up and divided by the number of samples to determine the average execution cost at line 11. The static variables `BILLINGINTERVAL`, `CPUCOST`, `MEMORYCOST` and `EXECUTIONCOST` depend on the pricing of the selected cloud provider and can be parameterized accordingly.

5.2.4 Limitations & Threats to Validity

Serverless functions can be provisioned with different memory limits, which indirectly also changes the processing power allocated to each function instance. Our approach currently does not take this into consideration and assumes that if a function is used in a workflow, its memory limit is not changed. While we consider this assumption reasonable, This approach could be integrated with the approach proposed in Section 5.1 to determine the impact of different memory sizes.

Besides costs for CPU time, memory time, and a flat execution cost, cloud providers usually also charge for network egress, that is the amount of data leaving their data center or a regional zone. Our approach currently does not consider this type of costs as the specification of the workflow model does not contain any information about when data leaves a regional zone or the data center of the cloud provider. However, our approach is already capable of estimating the size of the output data of a serverless function and if the workflow model is extended accordingly, it should also be possible to predict the egress costs.

The proposed approach considers functions as black-boxes that can only be monitored at the interface level. Therefore, it does not explicitly model potential external calls within the serverless functions. For external calls to other serverless solutions such as serverless object storage (e.g., S3 Buckets or Google Cloud Storage), serverless databases (e.g. AWS Aurora or Google Cloud Datastore), serverless event management (e.g., AWS SNS or Google Cloud Pub/Sub) or serverless in-memory data storage (e.g., AWS Elasticache) should not impact the response time prediction accuracy, as the load-independent response time of these calls is correctly modeled within the mixture density network describing the response time of the function issuing the external call. External calls to non-serverless services can negatively impact the response time prediction accuracy, as the load-dependent behavior of these external calls is not captured by our approach. In practice, this should be neglectable as synchronous external calls in serverless functions are a major anti-pattern as they cause double-billing [Bal+17].

5.2.5 Summary

To provide complex functionality, serverless functions are often assembled into workflows. However, estimating the costs of these serverless workflows is challenging as the response time and therefore the costs of a serverless function depend on its input parameters, which are propagated from prior functions within the workflow. In this contribution, we propose a methodology to predict the costs of serverless workflows. To answer **RQ IV.1** (*"How can the execution time distribution of a serverless function be predicted based on its input parameters?"*), we apply mixture density networks to predict the distribution of a function's response time and its output parameters. The resulting models are then combined into a workflow model. Based on this workflow model, a Monte-Carlo simulation derives cost estimates for the workflow execution, which answers **RQ IV.2** (*"Can the impact of restructuring a serverless workflow on its cost be predicted?"*). The approach presented in this contribution achieves **Goal IV** (*"Provide a technique to estimate the costs of serverless workflows."*) by providing accurate cost predictions for previously unobserved serverless workflows. Using our approach, solution architects can make informed decisions when choosing between a serverless workflow and a traditionally hosted workflow by providing concrete numbers for the costs of the serverless workflow. Based on our cost predictions, workflow designers can compare alternatives without time-intensive experimentation. Additionally, this contribution represents a first step towards fully automated workflow optimization using multi-objective optimization techniques, analogously to existing tools for traditional software systems [Ale+09; Mar+10]. The prediction accuracy of our approach and the potential cost savings are evaluated in Section 8.2. This contribution has been published as a full research paper at the International Conference on Performance Engineering (ICPE) [Eis+20b].

Chapter 6

Enabling White-Box Performance Modeling and Simulation of Serverless Platforms

Providers of serverless platforms need to understand the impact of different architectural and algorithmic design decisions on the performance properties of the platform. The high cost and risk associated with evaluating the performance of alternative designs in practice stymies the advancement of serverless platforms. In traditional software systems, white-box performance modeling is a common technique to evaluate the impact of design tradeoffs. Unfortunately, white-box performance modeling is currently challenging to apply to serverless platforms due to two well-documented shortcomings of white-box performance, as discussed in Challenge 5: a) white-box performance modeling requires the explicit modeling of parametric dependencies at design time, and b) the simulation time of a white-box performance model grows exponentially with the number of components.

In this section, we introduce two approaches targeted towards **Goal V** (*"Provide an approach for the white-box performance modeling and simulation of serverless platforms."*). The first approach speeds up the simulation time required to solve white-box performance models by enabling the parallel modeling of subsystems as fast-to-solve black-box performance models and as traditional queueing models. Further, we provide a transformation of the hybrid model toQPNs and extend an existing solver to support hybrid models. The second approach addresses the shortcoming that white-box performance modeling does not support the integration of empirically observed relationships between model parameters, such as resource demands, branch probabilities, and in-/output parameters. We propose a novel approach to modeling empirical parametric dependencies in architectural performance models. To derive performance prediction, a dependency resolution algorithm derives a fully parameterized model from the empirical data.

In the following, Section 9.1 introduces our approach for the simulation of fine-grained deployments, and Section 9.2 introduces our approach for the modeling and solution of empirical parametric dependencies.

6.1 Simulation of Fine-grained Deployments

The time required to simulate a model is an important factor for performance models. Nambiar et al. [Nam+16] identify faster model solution as a criterion for performance modeling success, Woodside et al. [WFP07] find that reduced run-time would increase the adoption of simulation-based solvers and Koziolek et al.[Koz10] state that the time required to solve a model is a limiting factor for the level of detail a performance model can contain.

In this contribution, we introduce a hybrid modeling approach that aims at **RQ V.1** (*"How can the time required to simulate large systems such as serverless platforms be reduced?"*). This generic modeling approach enables a parallel and integrated description of subsystems as statistical response time models (*black-box performance models*) and as traditional queueing models (*white-box performance models*) in order to speed up model simulation. The proposed approach allows users to dynamically select the appropriate modeling composition of statistical and queueing sub-models for each analysis run. We provide a transformation of the integrated queueing/statistical model to QPN [KB06] and extend an existing discrete event simulation solver for QPN to support black-box performance models. Additionally, we investigate the impact of replacing components or full subsystems by statistical response time models on the architectural performance model's accuracy and simulation time.

The remainder of this section is structured as follows: Section 6.1.1 describes our hybrid meta-modeling approach that extends component-based performance models with the statistical response time models, Section 6.1.2 describes our algorithm to extract statistical response time models from monitoring data, and Section 6.1.3 introduces an approach to transform architectural performance models containing statistical response time models to an extended QPN. Finally, Section 6.1.4 discusses the current limitations of the proposed approach. Our approach is evaluated in detail in Section 9.1.

6.1.1 Hybrid Meta-Model

In general, our modeling approach can be applied to any architectural software performance model which uses a component notation. Examples for software performance models which could be extended using our approach include the PCM [BKR09], CBML [WW04], PECT [His+02], or COMQUAD [Göb+04] as all of these models rely on the notion of repository components with some kind of performance description which are instantiated as assembly components. The meta-model proposed in this paper would not be directly applicable to performance models, such as ROBOCOP [Bon+05], which do not differentiate between repository and assembly components. However, the general concept of including statistical response time models in architectural performance models is also applicable to these performance models but would require a different modeling approach.

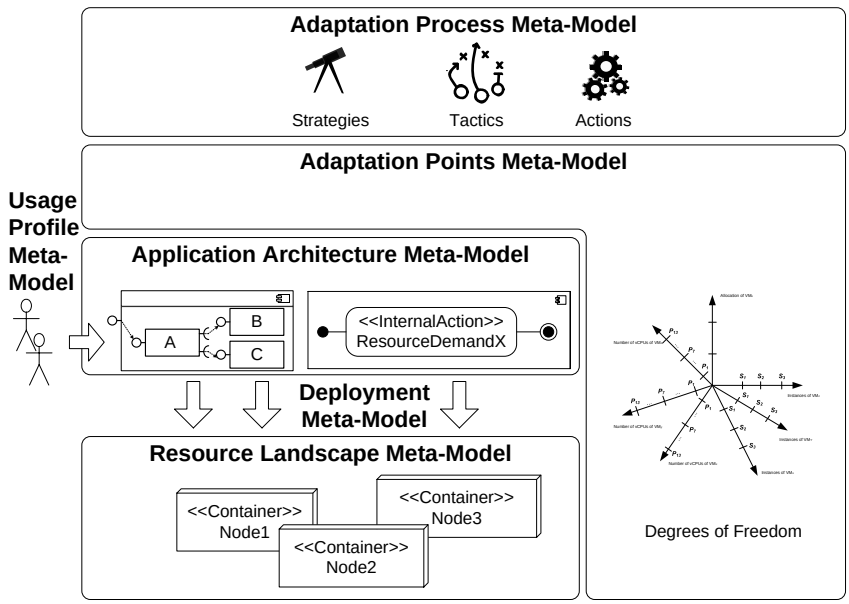


Figure 6.1: Structure of the DML meta-model (source: [Hub+17]).

We show how our approach can be applied to DML [BHK14], a performance model for component-based systems in data centers. It is representative of component-based performance models and was already evaluated in a number of case studies [Hub+17; Kou+16; Eis+18; Spi+19]. The meta-model of DML is separated into six submodels as shown in Figure 6.1:

Application architecture The application architecture meta-model consists of a component repository and a component assembly. The repository specifies the interfaces and components of a system. For each component the performance-relevant properties can be specified via resource demands, control flow operations (loops, branches, etc.), and calls to interface providing roles. In the assembly, these components are instantiated and connected to each other according to their interface providing and requiring roles.

Resource landscape The physical resources in a data center such as compute or storage nodes are contained in the resource landscape meta-model. The DML provides support for virtualized environments, such as VMs or containers as nested resources.

Deployment The deployment meta-model maps the instantiated assembly components from the application architecture meta-model to the physical resources defined in the resource landscape meta-model.

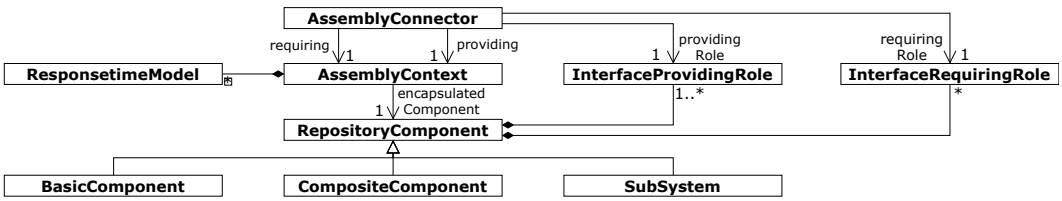


Figure 6.2: Meta-model for assembly with response time model integration.

Usage profile The workload is specified in the usage profile meta-model, similar to UML use cases and UML activities. Here, DML supports open and closed workloads, as well as detailed user sessions.

Adaptation points The possible adaptations to a system at run-time are limited, as not everything can be changed during system operation. Therefore, the adaptation points meta-model describes which elements of the resource landscape and application architecture can be adapted at run-time.

Adaptation process The adaptation process meta-model describes how a dynamic system reacts if its environment changes during operation. The DML supports three granularities that allow building composite adaptation processes: strategies, tactics, and actions.

Component assembly in DML is modeled similarly to UML, as shown in Figure 6.2. Every `RepositoryComponent` is either a `BasicComponent`, a `CompositeComponent` or a `SubSystem`. Every `RepositoryComponent` specifies a number of `InterfaceProviding-` and `InterfaceRequiringRoles`, which describe the functionality a component provides and what functionality should be provided by external components. A `RepositoryComponent` can be instantiated multiple times as `AssemblyContexts`. These component/subsystem instances are connected with each other by `AssemblyConnectors`, which connect `InterfaceProvidingRoles` to `InterfaceRequiringRoles`.

Traditionally, performance descriptions are modeled on the level of repository components to enable reuse of the performance descriptions [BKR09; WW04; His+02; Göb+04]. However, statistical response time models describe the response time of a specific component instance and the response time of a component is influenced by its required services and deployment [Koz10]. Therefore, we propose to enrich component instances in the form of `AssemblyContexts` with `Statistical-Models` describing its response time. This allows replacing component instances, composite component instances, and subsystem instances with statistical response time models during model solution.

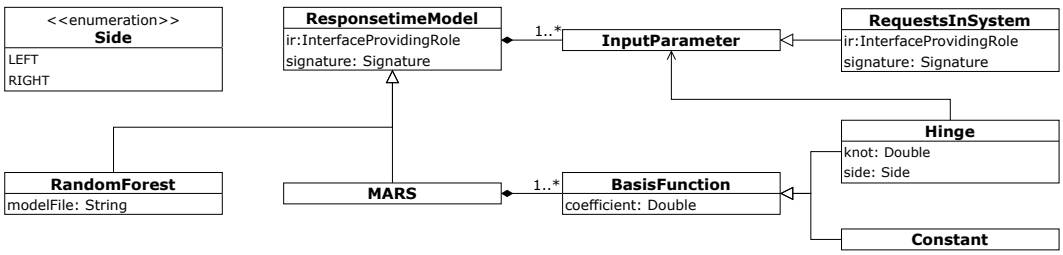


Figure 6.3: Meta-model for response time models.

Every response time model describes the response time for a specific workload class, as shown in Figure 6.3. In DML, a workload class can be uniquely identified by an `InterfaceProvidingRole` and a corresponding `Signature`. Therefore, every `ResponsetimeModel` references an `InterfaceProvidingRole` and a `Signature`. Every `ResponsetimeModel` has a number of `InputParameters` based on which the response time for a specific request is calculated. Currently, the only supported `InputParameter` is `RequestsInSystem`. Upon the arrival of a new request for which a response time has to be calculated, this parameter counts how many requests of a workload class (specified by an `InterfaceProvidingRole` and a `Signature`) are currently being processed by the component/subsystem. The `InputParameter` interface provides an extension point to include further parameters that influence the response time of a component/subsystem, for example, workload parameters such as the size of an input file. Response time models can either be a white-box, that is a human-readable function, or a black-box, such as a machine learning model. We model one of each, in order to showcase how they can be integrated into the model. As a black-box approach, we use a trained `RandomForest` model. Every `RandomForest` references a file containing the machine learning model. This model predicts the response time of a component/subsystem by using the `InputParameters` as features. `MARS` is a regression technique that results in a human-readable function, which can therefore be modeled explicitly. Every `MARS` model consists of a sum of `BasisFunctions`, which are either a static `Constant` or a `Hinge`. A `Hinge` $H(i, k, c, s)$ is a function over an `InputParameter` i with the following form:

$$H(i,k,c,s) = \begin{cases} c * max(0, k - i) & \text{for } s = LEFT, \\ c * max(0, i - k) & \text{else} \end{cases} \tag{6.1}$$

with a constant c, a knot k, and a side s. For further information on random forests or MARS see [Ho95] and [Fri91], respectively.

Our modeling approach provides three key characteristics: (i) It provides an extension point to include any regression or machine learning model. Although deep learning approaches might often be unfeasible due to the required quantity of training data, including them would be possible. (ii) Our approach annotates component/subsystem instances with response time models. The response time of

a component/subsystem depends on the deployment platform and the required services, which means the response time, unlike resource demands, cannot be specified for generic components. (iii) The presented meta-model enables parallel modeling of a component/subsystem as a traditional queueing system and as a response model. As the appropriate modeling granularity depends on the predicted performance metrics and model adaptations, this enables adapting the model for each individual request.

6.1.2 Statistical Response Time Model Extraction

Response time models can be extracted from either run-time monitoring or via dedicated measurements. There is a large body of work on the construction of response time models using dedicated measurements, for example, [Wes+12; Noo+13; FH12]. This research usually focuses on the intelligent selection of measurement points.

In the following, we present an approach to extract response time models from run-time monitoring. We assume generic monitoring data that consists of a set of monitoring records $rec = (wc, st, cp)$, where wc denotes the request's workload class, st and cp represent the absolute start and completion time of the request, respectively. Training regression or machine learning models requires a set of observations of a target variable and a number of features that will be used to predict the target variable. In our case, the target variable is the response time of a single request and the features are the number of requests of each workload class that are currently being processed within the component/subsystem. Therefore, an observation $obs = (rt, wc, req_{wc_1}, req_{wc_2}, ...)$ consists of the observed response time rt, the workload class of the processed request wc and the number of requests being processed in the component/subsystem upon arrival of the request for every workload class req_{wc_i}.

Algorithm 6.1 shows how we extract the training data for the regression and machine learning approaches based on run-time monitoring data. The algorithm receives a list of records $recordList$ as input and returns a list of observations $observations$. It maintains a list of requests that are currently being processed in the component/subsystem $requestsInSystem$ and a map $wcToCount$ that counts the number of requests currently being processed for each workload class in order to speed up the computation. In line 5, the algorithm iterates over the list of monitoring records (we assume that this list is ordered by request arrival time). For every monitoring record, an observation is created in lines 12-16. The observation consists of the response time of the request $record.cp - record.st$, the workload class of the request $record.wc$ and the number of requests already in the system for each workload class $wcToCount.get(wc_i)$. Afterwards, the request is added to the list of requests currently in the system $requestsInSystem$ and the counter for the number of requests currently in the system $wcToCount$ is increased for the workload class of the request.

Algorithm 6.1: Training data extraction algorithm

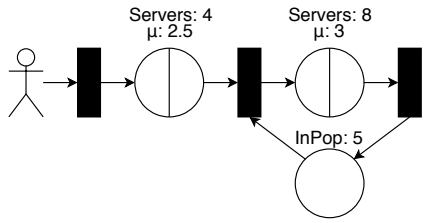
```
1  function EXTRACTTRAININGDATA(recordList):
2      observations = {}
3      requestsInSystem = {}
4      wcToCount = {wc_1 → 0, wc_2 → 0, ...}
5      for record in recordList do
6          for request in requestsInSystem do
7              if request.cp < record.st then
8                  requestsInSystem.remove(request)
9                  wcToCount.decrease(request.wc)
10             end
11         end
12         observations.add(new Observation(
13                                 record.cp − record.st,
14                                 record.wc,
15                                 wcToCount.get(wc_1),
16                                 wcToCount.get(wc_2), ...))
17         requestsInSystem.add(request)
18         wcToCount.increase(request.wc)
19     end
20     return observations
21 End function
```

Prior to this, the algorithm checks if any of the requests currently in the system has departed prior to the arrival of the current request in lines 6-11. It iterates over every request currently in the system and evaluates if the completion timestamp of the request $request.cp$ is smaller and therefore prior to the start timestamp of the new record $record.st$. Every request that finishes is removed from the list of requests currently in the system $requestsInSystem$ and the counter for the number of requests in the system $wcToCount$ is decreased for the workload class.

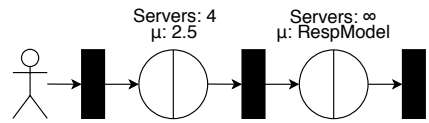
The presented algorithm converts common run-time monitoring data that consists only of start and completion time for each request and its request class to a format that enables training of most machine learning approaches and the fitting of stochastic models. The corresponding processes to train a MARS or random forest model on this data is described in [Ho95] and [Fri91], respectively.

6.1.3 Simulation of Hybrid Performance Models

Traditionally, architectural performance models are transformed into a solution formalism such as a QN, LQN, or QPN. DML, the modeling formalism used in this



(**a**) Initial queueing model



(**b**) queueing model with response time model

Figure 6.4: Example for the integration of statistical response time models in queueing models.

thesis, derives performance predictions by transforming the model to a QPN that is subsequently simulated using simQPN [KB06]. For detailed information on the transformation to QPN, we refer to [Bro14].

We extend the existing formalism by introducing a new distribution for the processing time, where the processing time is calculated by a statistical response time model based on the number of requests already being served. As the statistical response time model already considers queueing time, the corresponding queueing place has infinite servers to avoid duplicating the queueing time. It is important to note that each statistical response time model only describes the response time for a single request class. Therefore, a queueing place that serves two request classes has to contain two response time models.

Figure 6.4 shows an example how a subsystem can be replaced by a statistical response time model. In Figure 6.4a the original queueing model is depicted. It consists of two queues, which represent two components and a place with five initial tokens, that is used to model a limited software thread pool of five for the second component. Figure 6.4b shows the system if the second component is replaced by a statistical response time model. The full representation of the component (the queueing place and the ordinary place for the software thread pool) is replaced by a single queueing place with infinite servers and a processing rate which is described by a statistical response time model. This example describes the replacement of a single, coarsely modeled component. For more complex components or subsystems, the full model describing the component/subsystem is still replaced by a single queue with infinite servers and a statistical response time model. Assuming the statistical response time model perfectly predicts the response time of the subsystem/component, we can assume that the model with the statistical response time model predicts the same values for the following performance indices as the original model:

- Throughput of the model
- Overall response time of the model

- Utilization of all remaining queues

As the derivation of a single value from a response time model is significantly faster than the simulation of a request through a subsystem, the overall model solution time should be greatly reduced.

6.1.4 Limitations & Threats to Validity

While our approach shows significant benefits over the current state-of-the-art, there still are limitations and threats to the validity to be discussed.

Currently, the statistical response time models are trained based on the observed concurrency levels for each request class. However, the literature suggests that the parameterization of requests within a single request class can significantly impact resource demands [Koz10; Ack+18]. Our approach could be extended to take this into account by having these parameters implement the *InputParameter* interface shown in Figure 6.3.

Most components are stateless, but for a stateful component, the response time can be influenced by its internal state [HBR14]. The approach presented in this contribution is not able to capture this behavior accurately. In theory, response time models could be able to accurately capture such behavior if data on previous requests is included in the training data and a machine learning technique that can handle non-linear correlations is used. However, since serverless functions enforce statelessness anyway, this is outside the scope of this thesis.

The experimental evaluation of this contribution (see Section 9.1) considers a restricted scenario as there is no co-location of components on the same physical resources and the only investigated adaptation is the load level. Further experimentation including scenarios with co-located components, adaptations to the system architecture, and changing component implementations is required to derive a definitive rule set describing when statistical response time models can safely be integrated in architectural performance models.

Lastly, the exact results for the attained speedups by the integration of statistical response time models in architectural performance models are limited to MARS models. Further experimentation using different machine learning algorithms to build statistical response time models could provide additional insights. However, for most machine learning algorithms the training time is the limiting factor and deriving predictions from a previously trained model is comparatively fast [RN09].

6.1.5 Summary

Architectural performance models provide a powerful tool for software architects to evaluate and improve the performance of a software system and its architecture. However, the time required to simulate large or detailed models has been identified as a limiting factor by many researchers [Nam+16; WFP07; Koz10]. This

contribution addresses this issue by integrating statistical response time models into architectural performance models. The appropriate model composition depends on the performance metrics of interest. Our approach enables the modeling of each component and subsystem as a queueing network and a statistical response time model in parallel. This allows to dynamically tailor the system description for each analysis run. We provide a transformation of the integrated queueing/statistical model to QPN [KB06] and extend an existing discrete event simulation solver for QPN to support black-box performance models. Our approach enables software architects to analyze larger systems, performance engineers can explore more detailed models and self-adaptive systems can explore additional adaptation options within the same time period due to the faster model solution, which achieves **RQ V.1** (*"How can the time required to simulate large systems such as serverless platforms be reduced?"*). The prediction accuracy of our approach and the achieved simulation speedups are evaluated in Section 9.1. This contribution has been published as a full research paper at the International Conference on Software Architecture (ICSA) [Eis+19].

6.2 Simulation of Parametric Dependencies

At run-time, variables and dependencies can be learned and continuously updated based on available monitoring data [Spi17]. However, parts of the system may be inaccessible to monitoring as instrumentation might negatively impact the overall system performance [Ehl+11]. At design time, many details on component interactions remain open, limiting concrete specifications of parametric dependencies to aspects independent of component instantiation. In contrast, a run-time environment can supply sufficient information (e.g., using monitoring) to identify dependencies for specific component instances.

Existing architectural performance modeling formalisms [Koz08; Sit+01; Ham09; Bon+05; Ost+14] apply design-time assumptions when modeling parametric dependencies. In particular, we identified the following major limitations: (1) Existing parameter dependency models only support the modeling of dependencies per component type, not between specific component instances. However, in a run-time scenario, where dependencies between specific component instances can be obtained from monitoring data, modeling them becomes necessary. (2) No support for parameterization based on multiple parametric dependencies, which would allow for selecting alternative input parameters based on what is measurable. (3) Existing solutions [Koz08; Ham09; Bon+05] can not incorporate and benefit from correlations if they are not based on previously executed parameters from the call path, such as backward correlations.

In this section, we present a novel approach to integrate empirical parametric dependencies in architectural performance models of component-based software systems to address **RQ V.2** (*"How can relationships between parameters observed at run-time be utilized in white-box performance models?"*). Our approach enables the modeling of parametric dependencies on a component instance level, multiple independent dependencies describing one variable, and modeling of correlations as parametric dependencies. The dependency resolution algorithm transforms its information to a directed graph and resolves this graph to derive a fully parameterized model.

The remainder of this contribution is structured as follows: Section 6.2.1 introduces two motivating examples and provides a detailed problem statement. Section 6.2.2 describes the proposed parametric dependency modeling approach and Section 6.2.3 presents the corresponding dependency resolution approach. Finally, Section 6.2.4 discusses the limitations of the proposed approach. Finally, Section 6.2.5 summarizes the contribution. A detailed evaluation of the proposed approach is presented in Section 9.2.

6.2.1 Motivating Example

We illustrate general deficiencies for the modeling of parametric dependencies based on a video store application and then derive the general problem statement.

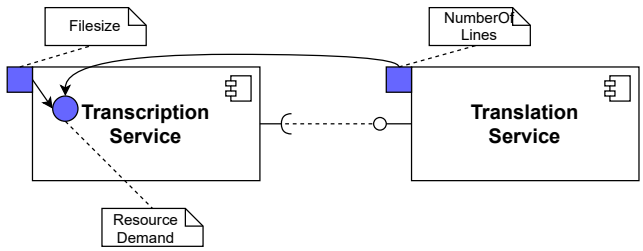


Figure 6.5: Generation of new subtitles.

6.2.1.1 Video Store Example

The challenges of modeling parametric dependencies in run-time scenarios can be easily highlighted using an online video store application, similar to YouTube, Netflix, or Amazon Video. The video store in our example provides its videos with subtitles in different languages. The subtitles are automatically generated and translated. We introduce two use cases that underline common requirements for modeling parametric dependencies at run-time. The first use case covers the automatic generation of new subtitles by transcribing the audio using machine learning and subsequent translation to different languages. In the second use case, the video store retrieves subtitles from a subtitle repository with a cache.

Use case 1: This use case is inspired by YouTube's automatically generated captions[1]. To automatically generate subtitles for newly uploaded videos, a transcription service uses machine learning techniques to transcribe the video's audio track. The resulting subtitles are then automatically translated by a translation service, as shown in Figure 6.5. The resource demand of the transcription service can be derived from the size of the file it transcribes. Another way to characterize this resource demand is to reverse engineer it using the resulting number of subtitle lines.

This use case motivates two novel modeling features: First when designing a performance model it might not be known for which parameters monitoring data will be available at run-time. Therefore, both dependencies have to be modeled and the decision which of these two dependencies should be used to characterize the resource demand of the transcription service should be made at run-time. Secondly, this use-case motivates the modeling of correlations as parametric dependencies. Variables can be described by correlations instead of relying on cause and effect. In general, the output of a method may encapsulate information about its internal execution process. In concrete, the correlation between the number of lines and the transcription resource demand opens an additional valid way to derive the resource demand.

Use case 2: This use case is inspired by video-on-demand providers, such as Netflix or Amazon Video. When a user requests a video, the subtitle repository

[1] https://googleblog.blogspot.de/2009/11/automatic-captions-in-youtube.html

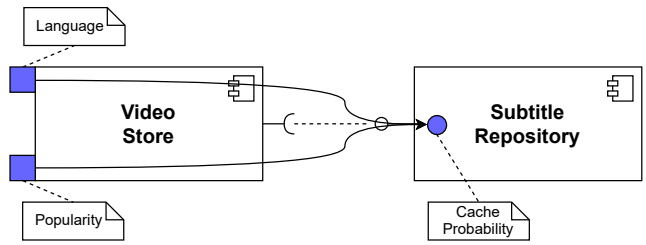


Figure 6.6: Retrieval of subtitles from the subtitle repository.

provides subtitles in the corresponding language, as shown in Figure 6.6. The subtitle repository implements a cache, which contains the frequently requested subtitles. The retrieval of subtitles from the cache causes a lower resource demand than retrieving subtitles from the database. The cache probability depends on both the popularity and language of the requested video, as subtitles in a frequent language for popular videos are more likely to be in the cache. This interaction can not be generalized for all subtitle repository instances. Other subtitle repository instances might be used in a different context, where the access frequency does not depend on the subtitle language or video popularity. An example for this would be a component creating backups, which iterates over all subtitles, independent of language or popularity. This showcases the need to model dependencies on an instance level in addition to component-level dependencies. At run-time, correlations between parameters can be learned from monitoring data for the deployed system. These correlations are only applicable for specific component instances, not for all instances of a component.

6.2.1.2 Problem Statement

Illustrated by the video store example, we identified the following general requirements to model parametric dependencies for component-based systems in run-time scenarios:

Instance-level dependencies Instance-level dependencies describe interrelations between component instances. Modeling these dependencies for component types, as supported by existing approaches, would apply the dependency to all instances of the component.

Multiple descriptions The description of parameters using multiple independent parametric dependencies provides alternatives for run-time model parameterization. A typical use case can be, for instance, the specification of component-level dependencies and instance-level dependencies for the same variable.

Correlations as dependencies At a runnable system state, monitoring data may reveal correlations between parameters. Modeling these correlations as dependencies can be used to derive characterizations in case parameters can not be measured. Existing parameter models can only capture strictly causal

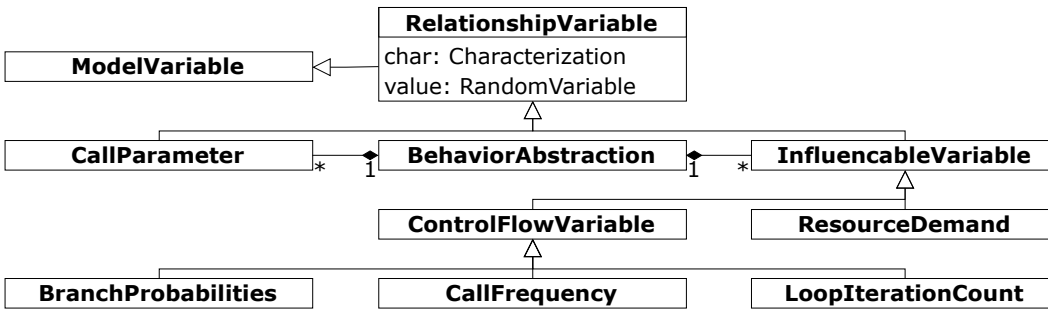


Figure 6.7: Model variables in DML.

correlations as dependencies since they enforce that a parameter may only depend on prior parameters from the same call path [Koz08; Ham09; Bon+05]. Existing approaches [Koz08; Sit+01; Ham09; Bon+05; Ost+14] can not model the parametric dependencies that occur in our video store example. In the following, we propose a modeling and resolution approach that provides native support for the above-described modeling features.

6.2.2 Parametric Dependency Modeling

We integrate our work for modeling and resolution of parametric dependencies into a representative architectural performance modeling formalism, the DML [BHK14; BHK11]. Therefore, we briefly introduce its relevant parts. The DML meta-model is made of five sub-models: repository, assembly, resource environment, deployment, and usage profile. The repository model defines blueprints for the software components that get assembled and connected in the assembly model. The deployment model describes how these components are distributed across the hardware resources defined in the resource environment model. The usage profile model contains the workload definition. For a detailed introduction to the DML meta-model, we refer to [BHK14].

The integration of novel modeling features requires the identification of the affected meta-model parts. The resource landscape and the deployment do not affect parametric dependencies. Only the application architecture meta-model contains information about parameters and dependencies between them. In the application architecture, components are modeled, instantiated, and connected to other components using interface providing and requiring roles. Every interface signature provided by a component has a corresponding behavior description. It may contain control-flow descriptions such as loops, branches, and forks. Additionally, behavior descriptions contain resource demands and external calls to other components. The specification of parametric dependencies builds upon three main concepts: (i) variables, (ii) parameters, and (iii) relationships.

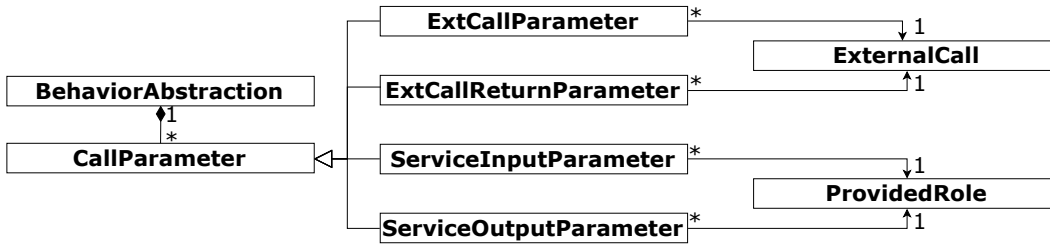


Figure 6.8: Call parameters in DML.

Variables depict a core concept for parametric dependency modeling. Figure 6.7 shows the variable and parameter types to which dependency descriptions may refer to. Every variable inherits from `RelationshipVariable`. Each `RelationshipVariable` can either be an `InfluencableVariable` or a `CallParameter`, which are both contained in a `BehaviorAbstraction`. An `InfluencableVariable` can be a `ResourceDemand`, a `BranchProbability`, a `CallFrequency`, or a `LoopIterationCount`. Any `RelationshipVariable` contains a `value` attribute of type `RandomVariable` describing its distribution. The `value` attribute can be `NULL` if its distribution is to be derived based on parametric dependencies. The `Characterization` of a `RelationshipVariable` indicates whether the variable value is explicitly modeled or should be characterized using monitoring data.

Calls to components can contain parameters that influence the behavior of the called component. The value returned by a call to another component can also influence the calling component's behavior. Common examples include file sizes and list lengths. In DML, such parameters are modeled as `CallParameters`, which are depicted in Figure 6.8. `CallParameters` include `ExtCallParameters`, `ExtCallReturnParameters`, `ServiceInputParameters`, and `ServiceOutputParameters`. `ServiceInputParameters` together with `ServiceOutputParameters` model a component's input and output parameters. Therefore, they reference a `ProvidedRole`, which describes the interface and signature the parameter belongs to. The counterparts to these parameters are the `ExtCallParameters` and the `ExtCallReturnParameters`, which specify the input and output parameters for a call to another component. If two component instances are connected via an assembly, the `ExtCallParameter` and the respective `ServiceInputParameter` share the same distribution; the same applies to `ServiceOutputParameters` and `ExtCallReturnParameters`.

In order to model dependencies both on repository level and on component instance level, DML provides two types of `Relationships`, as shown in Figure 6.9. A `DependencyRelationship` represents a dependency on the repository level and is therefore modeled as part of the component blueprint in the `Repository`. `CorrelationRelationships` on the other hand are used to model dependencies on a component instance level. Therefore it is modeled as part of the `System` and refers to two or more `AssemblyContexts`, which represent specific component instances.

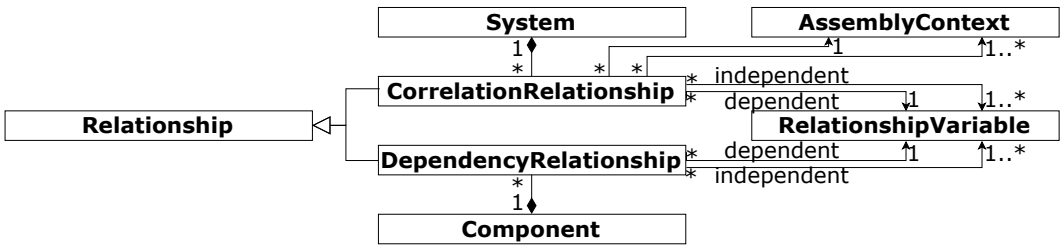


Figure 6.9: Relationships in DML.

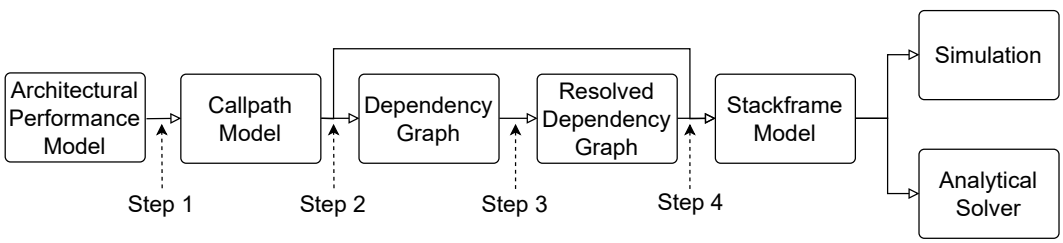


Figure 6.10: Dependency resolution process.

Both types of `Relationships` contain an equation which can be used to derive the value of the dependent `RelationshipVariable` from the one or more independent `RelationshipVariables`.

6.2.3 Parametric Dependency Resolution

The second major objective of our approach is to automatically derive performance indices from models using our novel and existing dependency modeling features. The automated prediction requires a resolution of the declarative parametric dependency description to enable model analyses. Besides functional requirements, we formulate the following design goals for the realization of our dependency resolution approach: (i) modularity in order to improve maintainability and (ii) independence of concrete stochastic model solvers.

To achieve modular design, we decompose the resolution into the multi-step dependency resolution process depicted in Figure 6.10. It consists of the following steps:

Step 1 Extraction of possible call paths through the system from the architectural model into a `CallpathModel`.

Step 2 Transformation of the model parameters and the dependencies between them into a directed graph, denoted as `DependencyGraph`.

Step 3 Resolution of parametric dependencies within the `DependencyGraph` using our dependency resolution algorithm to generate the `ResolvedDependencyGraph`.

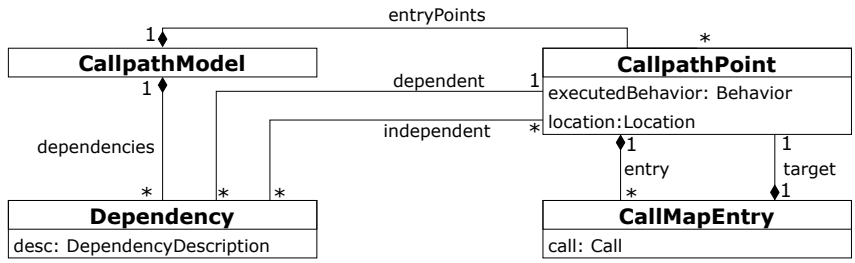


Figure 6.11: Callpath meta-model.

Step 4 Combination of the information contained in the `CallpathModel` and the `ResolvedDependencyGraph` in order to generate a `StackframeModel`. The `StackframeModel` represents a solution-ready model that can be solved using existing simulators and analytical solvers by transforming it to their respective analysis format [Hub+17].

The resolution of parametric dependencies should not rely on a specific prediction formalism in order to be reused in transformations to different stochastic models and respective analytical or simulation-based solvers. Through the transformation to the `StackframeModel`, our dependency resolution is independent of specific stochastic model solvers. In addition, this enables the reuse of all existing DML solvers. The description of employed intermediate models allows for their reuse in the context of further architectural performance modeling formalisms. In the following, we detail the employed intermediate meta-models and our dependency resolution algorithm.

6.2.3.1 Intermediate Meta-Models

The dependency resolution algorithm applies a set of intermediate graph meta-models. In this section, we describe the `CallpathModel`, the `DependencyGraph`, and the `StackframeModel`.

Callpath Model The `CallpathModel`, depicted in Figure 6.11, captures all paths a request may take through the system. It contains a set of `CallpathPoints` representing entry points to the system. Each `CallpathPoint` specifies the `Location` where its `Behavior` is executed. A location does not refer to a physical node in the system, but to an assembled software component that contains the executed behavior. For every `Call` in the `Behavior` of a `CallpathPoint`, a `CallMapEntry` references another `CallpathPoint` representing the `Call's` target `Behavior` and the assembly component it is executed in. Every possible path that leads to the execution of a `Behavior` can be described by a single `CallpathPoint` as it contains a reference to its predecessor. Additionally, the `CallpathModel` contains `Dependencies`, which connect two or more `CallpathPoints`. The `Behavior` description of `CallpathPoints` contains the independent and dependent parameters. The nature of the depen-

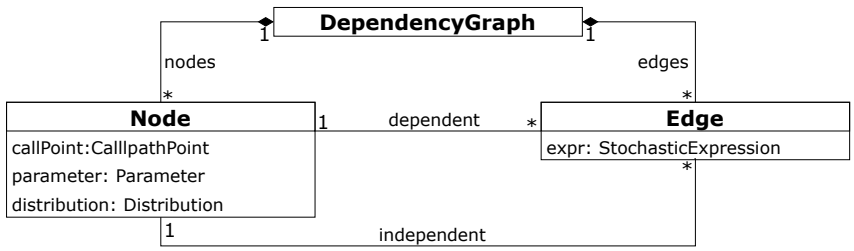


Figure 6.12: Dependency graph meta-model.

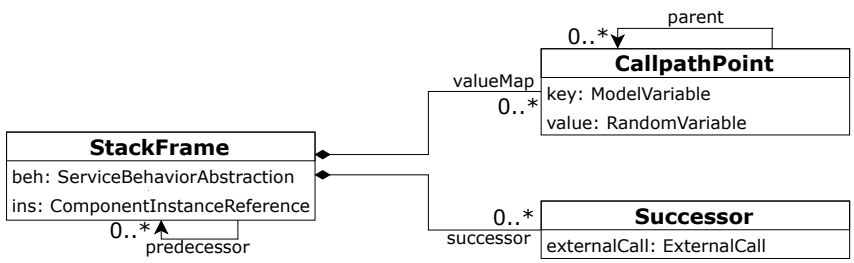


Figure 6.13: Stackframe model from [Hub+17].

dency is abstracted in a `DependencyDescription`. The meta-model only contains the connections between `CallpathPoints`. Component level dependencies do not have to be modeled in the `CallpathModel` since the location of its dependent and independent components is captured within the model.

Dependency Graph The `DependencyGraph`, depicted in Figure 6.12, contains information about the model parameters and the dependencies between them. Every `Node` in the Graph represents a `Parameter` and a call path to it, which is represented by a `CallpathPoint`. This means that one `Parameter` can be contained in multiple `Nodes` if it lies on multiple call paths, allowing a `Parameter` to have different values depending on the call path. If the distribution of the `Parameter` in this call path is known, the value is saved in its `Distribution`, otherwise it is `NULL`. The `Nodes` are connected by directed `Edges` that model the dependencies between them. Every `Edge` connects one dependent `Node` to one or more independent `Nodes`. The `Edge's` `StochasticExpression` describes how the distribution of the dependent `Node` can be calculated if the values of the independent `Nodes` are known.

Stackframe Model We reuse the `StackframeModel` presented in [Hub+17] as output for our dependency resolution algorithm. The `StackframeModel` represents a solution ready form of an architectural performance model. In the `StackframeModel`, the sophisticated aspects of architectural performance models are resolved, simplifying the transformation into solution models such as QPNs or LQNs, as described in [Hub+17]. Figure 6.13 shows the meta-model of the `StackframeModel`. It consists of a series of `StackFrames` which describe a `ServiceBehaviorAbstraction` and

Algorithm 6.2: Dependency Resolution Algorithm

```
1  function SOLVEDEPENDENCIES(dependencyGraph):
2      hasChanged = true
3      while hasChanged == true do
4          hasChanged = false
5          for Edge in dependencyGraph.edges do
6              allIndependentsCharacterized = true
7              for Node in Edge.independents do
8                  if Node.value == null then
9                      resolvable = false
10                 end
11             end
12             if resolvable == true then
13                 Edge.dependent.value = Edge.calc()
14                 hasChanged = true
15             end
16         end
17     end
18 End function
```

a `ComponentInstanceReference`, which describe the assembly instance on which the behavior is executed. For every `ExternalCall` in the `StackFrame's` behavior, a `Successor` annotates which `StackFrame` is called by the `ExternalCall`. Similarly, a `ValueMapEntry` annotates every `ModelVariable` in the `StackFrame's` behavior with a `RandomVariable` describing its distribution.

6.2.3.2 Resolution Process

This section details how our dependency resolution algorithm uses the intermediate meta-models to transform a model containing parametric dependencies to an analysis-ready model without dependencies. The approach consists of the following four steps:

Step 1 The `CallpathModel` can be extracted from an architectural performance model by creating `CallpathPoints` for all possible entry points. Next, iterating over all newly created `CallpathPoints`, a `CallMapEntry`, and a target `CallpathPoint` has to be created for every `Call` inside the `CallpathPoint's Behavior`. This needs to be recursively repeated for all `CallpathPoints` created by this step, until every call path ends in a `CallpathPoint` who's `Behavior` does not contain any `Calls`.

Step 2 The `CallpathModel` can be transformed to the `DependencyGraph` by iterating over all `CallpathPoints` in the `CallpathModel` and executing the following tasks on them:

- Create a Node for every parameter in the CallpathPoint Behavior.
- Create an Edge for every component level dependency described in the CallpathPoint's Behavior.
- Create an Edge for every pair of input and output parameter in the CallpathPoint's Behavior and its predecessor's Behavior.
- Create an Edge for every instance level dependency modeled in the CallpathModel who's dependent is contained in the CallpathPoint's Behavior.

This generates a complete DependencyGraph, which can be used to derive values for all unknown parameters in the following step.

Step 3 The DependencyGraph contains Nodes with known distributions and Nodes with unknown distributions. If all independent Nodes of an Edge have distributions, the distribution of the dependent Node can be derived. To determine a resolution order that resolves distributions for all Nodes, we use the dependency solver algorithm shown in Algorithm 6.2. It iterates over all Edges in the dependency graph and evaluates if every independent Node of the Edge is already characterized. Should this be the case, the algorithm computes the distribution of the dependent Node of the Edge. The algorithm repeats these steps until no changes occur in the dependency graph after iterating over the Edges. This resolves distributions for all Nodes and therefore for every occurrence of every parameter in the system.

Step 4 We transform the CallpathModel and the ResolvedDependencyGraph to a StackframeModel. The CallpathPoints and the CallMapEntries connecting them can be directly mapped to StackFrames and Successor. For every parameter in a CallpathPoint Behavior, its distribution can be found in the corresponding Node of the ResolvedDependencyGraph. We transform it to a ValueMapEntry of the StackframeModel. The result of the whole process is a fully parameterized StackframeModel, which can be analyzed using a variety of analytical approaches and simulations [Hub+17].

6.2.4 Limitations & Threats to Validity

While we consider our approach a significant improvement over the current state-of-the-art, there still are limitations and threats to the validity to be discussed.

Algorithm 6.2 for the resolution of parametric dependencies could potentially never terminate. However, since each iteration reduces the number of nodes without characterization, it will always terminate. In case no new node was characterized, the algorithm also terminates. Whenever a parametric performance model contains cyclic dependencies, one of the dependencies making up the cycle can be ignored since it describes an already known variable. Assuming that all dependencies describe the variable with the same accuracy, no information is lost.

There is no guarantee that our approach will be able to resolve distributions for all variables. In case the parameter model contains insufficient information to derive distributions for all variables, our algorithm still resolves as many distributions as possible and then returns the partially resolved `DependencyGraph`. This partially resolved `DependencyGraph` contains the information which variables could not be resolved and measurement values for which parameters would allow to resolve the missing distributions.

If the distributions of two variables can be derived from the same parameter or share a common ancestor in the `DependencyGraph`, then the resulting model wrongly assumes that the distributions of the two variables are statistically independent. This problem is also inherent to existing approaches [Bro+15]. In contrast to existing work, our approach allows to automatically detect this by checking whether two variables share a common ancestor in the `DependencyGraph`. This allows to inform the user that the parameterization might be inaccurate.

6.2.5 Summary

This contribution provides novel modeling features for parametric dependencies and a corresponding solution algorithm, which extends state-of-the-art modeling with run-time-specific features. In particular, it enables parametric dependencies on component instance level, multiple descriptions of a single variable, and modeling of correlations as dependencies. Next, we introduce a dependency resolution algorithm that transforms the model to a directed graph and resolves this graph to derive a fully parameterized model. The presented approach enables the utilization of empirically observed parametric dependencies improving the ability to reflect system behavior within architectural performance models based on higher flexibility of inputs, which shows that we have answered **RQ V.2** (*"How can relationships between parameters observed at runtime be utilized in white-box performance models?"*). This results in higher performance prediction accuracy, which is valuable for various purposes such as capacity planning [MV00], bottleneck analysis [ST07], design tradeoff analysis [KAM13], and proactive auto-scaling [BHK17]. A detailed evaluation of the proposed approach is presented in Section 9.2. This contribution has been published as a full research paper at the International Conference on Software Architecture (ICSA) [Eis+18].

Part III

Results & Evaluation

Chapter 7

Results for the Characteristics and Performance of Serverless Applications

7.1 Serverless Application Characterization

In this chapter, we present the results of our serverless application characterization, which consists of the collection of serverless applications, their characterization by multiple reviewers, and the analysis of consensus among existing studies. We present our results for **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*) in the context of the following two research questions, that we defined in Section 1.3:

- **RQ I.1** (*"What are common characteristics of current serverless applications?"*)
- **RQ I.2** (*"Is there a community consensus on the common characteristics of serverless applications?"*)

In the following, Section 7.1.1 introduces the results of our characterization of 89 serverless applications from open-source projects, academic literature, industrial literature, and scientific computing. Next, Section 7.1.2 discusses our findings on the consensus between our results and ten existing, mostly industrial studies. Our replication package is introduced in Section 7.1.3 and finally, Section 7.1.4 summarizes our findings.

7.1.1 Characteristics Analysis

In the following, we describe the serverless application characterization results in the context of six common questions about serverless applications.

How are serverless applications implemented?

When AWS revealed Lambda in 2014, it was the only FaaS platform offered by a major cloud provider, and it only supported JavaScript functions. Since then, tens of serverless platforms have emerged [Eyk+19, §3], offering support for diverse programming languages. The capabilities and performance features of these platforms

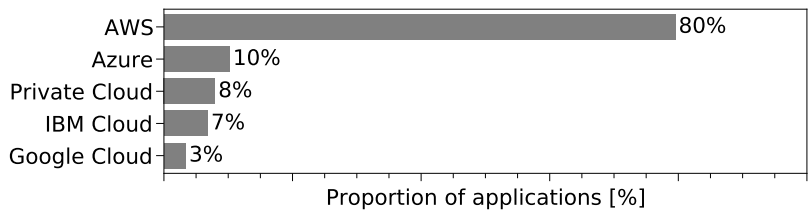


Figure 7.1: Cloud provider used for serverless applications.

and languages have been studied extensively [SL20]. We focus here on the state of practice in *implementing* serverless applications.

We analyze the collection of serverless applications introduced in Section 4.1.1. Figure 7.1 shows that 80% of the applications in our dataset are using the cloud provider AWS, whereas the other major cloud providers are used a lot less often (10% Azure, 7% IBM Cloud, and 3% Google Cloud). Although AWS also has the largest market share when it comes to IaaS at 47.8% [Gar18], this difference alone is not enough to explain why so many of the applications in our dataset are using AWS. A potential explanation is that AWS Lambda was released two years before any other large cloud provider released their Function-as-a-Service solution, which means this platform is likely more mature and there was more time for customers to adopt its serverless features. Since then, many open-source Function-as-a-Service solutions launched [Eyk+19], yet we do not see significant adoption for them in our dataset (a combined 8%, mostly by scientific applications). We observe that most applications in our dataset use managed cloud services that would not be available in a private cloud environment; this could explain the low adoption of the open-source Function-as-a-Service solutions and also spur innovators in serverless technology to consider more carefully the ecosystem where their platforms can work.

Interpreted languages may be better suited for serverless applications than compiled languages because compiled languages suffer from longer cold-starts [Mal19]. Figure 7.2 corroborates this rule-of-thumb: JavaScript (42%) and Python (42%) are the most popular programming languages. Serverless applications are also written in Java (12%), C/C++ (11%), or C# (8%); few use Go (5%) or Ruby (2%). However, this may change, as the usage of ahead-of-time compilation, for example, for Java, has been shown to alleviate the difference in cold-start duration [MG20].

Finding 1: Currently, AWS is the dominating platform for serverless applications (80%), and most applications are implemented in either JavaScript or Python (42% each) (**RQ I.1**).

What does a typical serverless architecture like?

Developers looking to implement serverless applications need to make many archi-

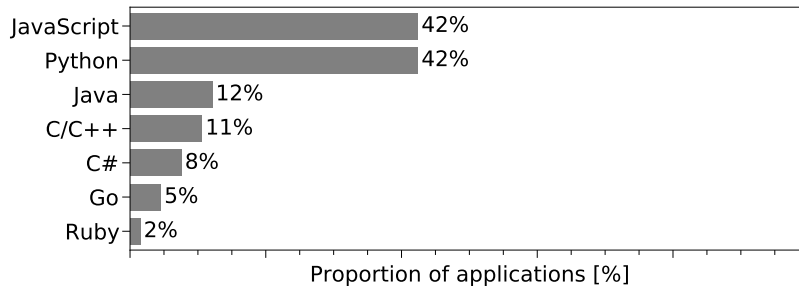


Figure 7.2: Programming language used for serverless applications.

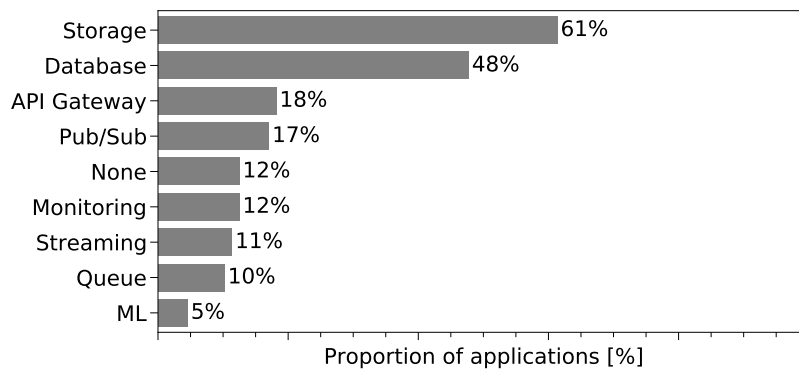


Figure 7.3: Managed services used by serverless applications.

tectural decisions, such as which external services to use, how many functions to use, and how they are triggered. Understanding patterns in serverless architecture can guide the general discourse on serverless applications and provide a valuable guideline for developers starting to build serverless applications.

Figure 7.3, label *None*, shows that only 12% of the serverless applications in our dataset do not use any managed service. This suggests serverless applications are typically created by combining serverless functions for compute and managed cloud services for other operations. The most frequently used managed services in our dataset are storage (61%) and databases (48%). Serverless functions are stateless, therefore all application state needs to be persisted in external storage and databases. The second class of most frequently used external services are managed messaging services, including publish/subscribe solutions (17%), streaming solutions (11%), and queueing solutions (10%). Serverless functions often use such messaging services to store their output if it needs to be processed further.

Cloud providers offer different ways to trigger the execution of serverless functions. As Figure 7.4 depicts, we find that 48% of the selected applications use HTTP triggers, and 41% use cloud event triggers (e.g., as a new message in a queue, or a

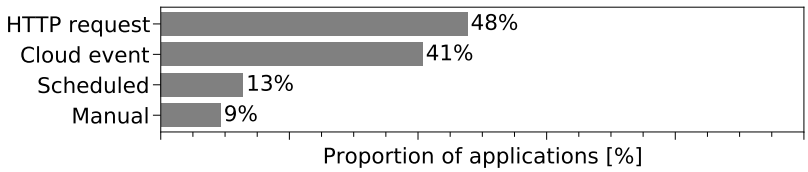


Figure 7.4: Trigger types used in serverless applications.

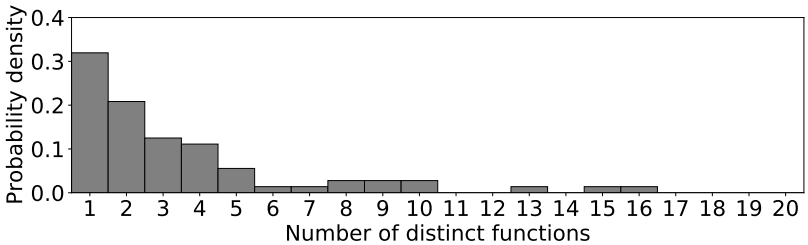


Figure 7.5: Number of serverless functions per serverless application.

new entry in a database). Generally, HTTP triggers are commonly used to expose functionality to users, whereas cloud events help coordinate multiple cloud functions. This is a significant change from microservices, which typically rely on API calls to coordinate multiple services. One of the reasons for this change towards an event-driven architecture could be that synchronous calls between serverless functions cause double billing [Bal+17]. A smaller number of applications use schedule-based triggers (13%) or manually triggered functionality (9%). These triggers are usually used for orchestration or management tasks.

Figure 7.5 shows that the number of functions per application is relatively low: Only 7% of the applications in our dataset use more than 10 functions, and 82% use 5 or fewer functions. This suggests, firstly, that the use of external services reduces the amount of (internal) code required to build an application. Secondly, the functionality encapsulated by a serverless function is between a microservice and an API endpoint, as the applications we review do not wrap every programming function as a serverless function [SMM18]. The term "serverless functions" might be misleading, as they are not related to the programming concept of functions.

Finding 2: Serverless applications typically use cloud storage (61%), cloud databases (47%), and cloud messaging (38%). They use few cloud functions: 82% of serverless applications use 5 or fewer functions (**RQ I.1**).

What are common traffic patterns for serverless applications?

Traffic patterns—namely, execution patterns, burstiness characteristics, and data

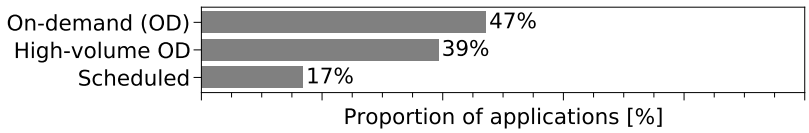


Figure 7.6: Execution pattern of serverless applications.

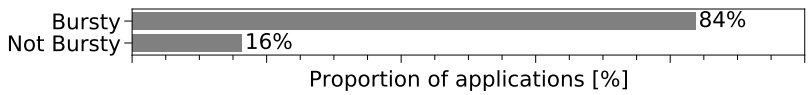


Figure 7.7: Burstiness of the workload of serverless applications.

volumes—can reveal how serverless platforms are used. Applications can be executed *on-demand* when a user interacts with the application or a cloud event occurs; we further classify the on-demand execution as regular *on-demand* or *high-volume on-demand*. Applications can also be *scheduled* to run at specific times, for example, to perform cleanup tasks during off-hours.

Regarding the *execution patterns*, Figure 7.6 shows that most applications are triggered on-demand (86%), out of which more than half are high-volume invocations, associated with business-critical functions. Only 17% of the applications are triggered by a periodic schedule. Through an in-depth analysis, we find that about half of the scheduled applications execute *operations & monitoring* functions, highlighting how the serverless model has been adopted—in many cases—to automate operations, software management, and DevOps pipelines. We also note that the high prevalence of on-demand triggered applications, and specifically, high-volume on-demand patterns, is well supported by the industry trends of reducing overheads (i.e., function start-up time), and of providing quick and seamless function auto-scaling mechanisms.

Regarding *burstiness*, we classify applications as having potentially bursty workloads or non-bursty workloads. A *bursty application* follows a workload pattern that includes sudden and unexpected load spikes, or a significant amount of sustained noise and variation in intensity. We classify an application as *non-bursty* if the workload is guaranteed to rarely or never experience bursts (e.g., if all executions are scheduled and known in advance); otherwise, the workload is bursty. When humans trigger function executions, the workload pattern can be bursty, as user behavior can rarely be scheduled or reliably controlled. As Figure 7.7 shows, we classify more than 84% of the workload patterns we analyzed as bursty; only 16% have a clear non-bursty pattern. As one of the strengths of serverless computing is its seamless scalability, together with the general ease of operations, it comes as no surprise that most of the applications can indeed experience bursty workload patterns.

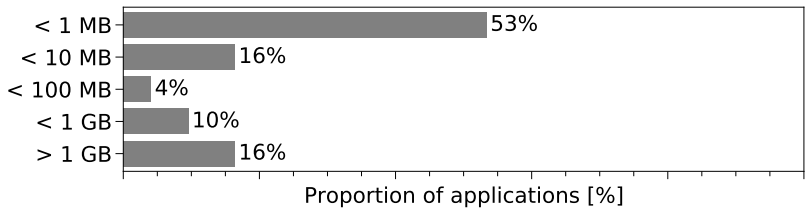


Figure 7.8: Data volume handled by serverless applications.

Finally, we analyze the *data volume* or load that the serverless applications issue on the network and storage devices. We classify applications into: volumes of *less than 1 MB* per execution, *less than 10 MB*, *less than 100 MB*, *less than 1 GB*, and *more than 1 GB*. Exact numbers rarely appear in our sources so this classification is based on reviewers' estimates. Figure 7.8 shows (i) more than half of the applications (53%) in the smallest category of data volumes and 16% in the next (<10 MB) and (ii) the second peak (16%) in the largest category (>1 GB). The resulting distribution appears bimodal, but this might be an artifact of the binning intervals.

Finding 3: Most serverless applications have potentially bursty workloads (84%). Serverless applications are often used for high-traffic workloads (39%) (**RQ I.1**).

What are serverless applications used for?

A common assumption is that serverless applications are suitable for operations tasks and batch jobs, as their traffic patterns profit from the pay-per-use model. For example, Netflix uses AWS Lambda for operations tasks, such as video encoding, file backup, security audits of EC2 instances, and monitoring. However, the core functionality—the website and app backend, and video delivery—is still running on traditional IaaS cloud services [Lau18]. Contrary to this popular belief, and as Figure 7.9 depicts, we find that the most common serverless applications in our dataset are implementing APIs (29%) or are processing frequent events (streams) asynchronously (28%). Example use cases for these types of applications are serverless backends for web, mobile, or IoT applications. Still, a significant portion of serverless applications focus on processing batch tasks (24%) and on automating operations tasks (20%).

Another common assumption is that serverless applications are not suitable for complex analysis tasks. In contrast, in our dataset, 25% of applications contain functions with an estimated runtime of over one minute. Among these applications are scientific workloads, such as SNP Genotyping [Cre+19] or seismic imaging [Wit+20], showing an increased adoption of serverless computing for complex analysis tasks.

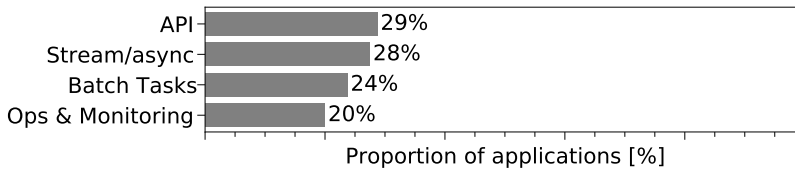


Figure 7.9: Application type of serverless applications.

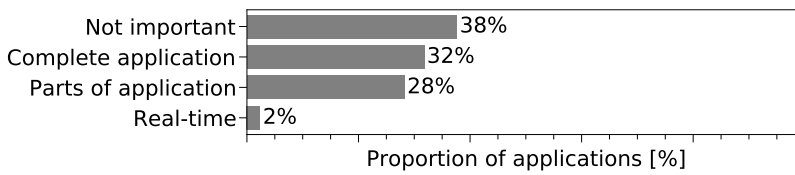


Figure 7.10: Latency requirements of serverless applications.

The high percentage of APIs is somewhat surprising, as a common argument against serverless applications is that cold starts make them unsuitable for applications with low-latency requirements or focus on tail-latency. However, we find that serverless applications are used for latency-critical tasks. As shown in Figure 7.10, 38% of the selected serverless applications have no latency requirements. However, 32% of the serverless applications have latency requirements for all functionality, 28% have partial latency requirements, and 2% even have real-time requirements.

Finding 4: Serverless applications are not limited to any specific types of applications, as they are commonly used to implement APIs (29%), stream/async processing (28%), batch tasks (24%), and operations tasks (20%) (**RQ I.1**).

Why are practitioners choosing serverless computing?

Several potential benefits of serverless applications have been proposed: reduced operational effort, faster development due to the heavy use of Backend-as-a-Service, and near-infinite scalability of serverless applications. Many also discuss significant cost savings from switching to serverless computing. However, these benefits are not generally agreed upon, for example, cost savings have come under scrutiny [Eiv17]. To understand why practitioners choose to adopt serverless computing, we investigate the descriptions and documentation of applications in our dataset.

We could not determine the reasons behind the adoption of serverless computing for about 30% of the applications in our dataset, as the documentation did not mention explicitly why serverless computing was chosen. We analyze the remainder and depict the results in Figure 7.11. The main driver is cost—mentioned by 47% of the remaining applications. While serverless computing is not *per se* cheaper

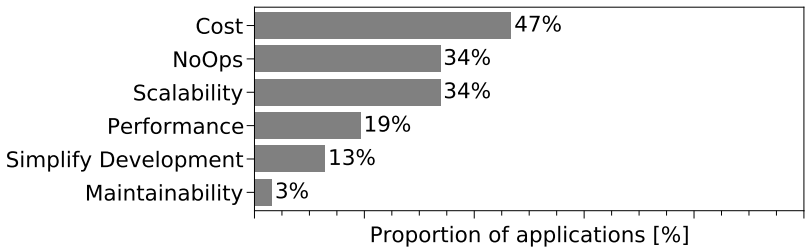


Figure 7.11: Motivation for building serverless applications.

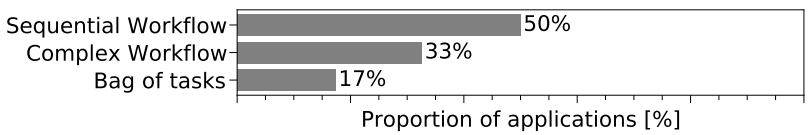


Figure 7.12: Structure of serverless workflows.

than IaaS hosting, the pay-per-use model and ability scale to zero, reduce costs in scenarios where the IaaS resources are underutilized. Serverless applications can also offer seamless, virtually infinite scaling. Scalability is mentioned as a reason for serverless adoption by 34% of the applications in our dataset. The third main reason for choosing serverless computing over traditional hosting options is reduced operational overhead because server management is no longer done by applications. A few applications also reported improved performance (19%) and faster development speed (13%) as reasons for serverless adoption.

Finding 5: Reduced hosting costs of serverless applications (47%), reduced operation effort (34%), and high scalability (34%) are the main drivers for serverless adoption (**RQ I.1**).

How complex are serverless applications?

Although initially, serverless computing focused on simple applications, comprised of mainly small functions, there has been increasing interest in using serverless computing for more complex applications. Such applications can be expressed as *serverless workflows*, which orchestrate the dependencies between multiple functions.

From our dataset, we find a significant percentage of applications already structured as serverless workflows (31%). Examples of such serverless workflows can range from simple workflows to large scientific workflows [Wit+20].

Similar to workflows in other fields, we can classify these serverless workflows into specific patterns based on how the function calls (or tasks) are structured within these orchestrations. As Figure 7.12 depicts, we find that half of the workflows are

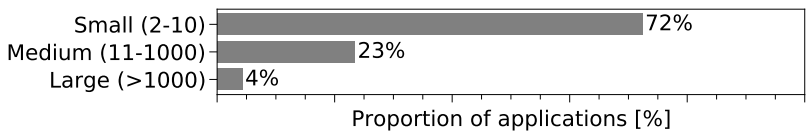


Figure 7.13: Size of serverless workflows.

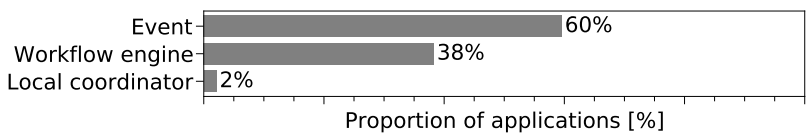


Figure 7.14: Coordination of serverless workflows.

sequential in nature, where tasks are executed one after the other. We also find *bags of tasks* (17%), where a set of tasks can execute without a particular order or inter-dependency. Finally, a third (33%) of the applications are defined as *complex workflows*, that is, workflows that include structures such as conditional branches and loops. The diversity and level of complexity indicate designers of workflow management systems are soon to be engaged in a competition over new features and optimizations.

Another key differentiator between the selected applications is workflow size (shown in Figure 7.13). Most applications (72%) have rather small workflows, consisting of 2 to 10 tasks. These tend to be applications for business processes and data pipelines, such as the multi-step provisioning of developer machines at Autodesk[Wil17]. Almost a quarter of serverless workflows (23%) contain 11 up to 1,000 tasks. Finally, a few serverless workflows (4%) consist of more than 1,000 tasks—typically, large scientific workflows requiring custom workflow engines to run.

A final factor in serverless workflows is the approach used to orchestrate their execution. Overall, as Figure 7.14 highlights, we found that most serverless workflow applications (60%) use *event-based mechanisms*—such as file uploads triggering the execution of functions—to implicitly orchestrate entire workflows by ensuring that the result of one function triggers the next. About a third of the workflows (38%) are managed by a *dedicated workflow management* system, such as AWS Step Functions, Google Workflows, or Azure Durable Functions. Besides these cloud-based orchestration methods, we also identify a less-common approach, *local coordination* (2%), in which the orchestration complexity is deferred to the client-side. Although this is less robust than other methods, it is used for one-off workflows, for example, the distributed build-workflows of gg [Fou+19].

Table 7.1: Degree of agreement with existing studies. A - denotes that the study did not investigate this characteristic and a (-) denotes that the results are incomparable due to differences in the question or answer options.

Characteristic	SCS	SitW	MMS	TSoS	OSS	GtST	FtLoS	FSS	DSS	CNCF
Platform	Very High	-	(-)	-	Very High	High	-	-	High	-
Language	Medium	-	High	Very High	-	Medium	Very High	-	-	-
External Services	(-)	(-)	(-)	(-)	-	-	-	-	-	-
Trigger Types	-	Very High	-	-	-	-	-	-	-	-
Number of Functions	(-)	Very High	High	-	-	Low	-	-	-	-
Execution Pattern	-	(-)	-	-	-	-	-	-	-	-
Burstiness	-	High	-	-	-	-	-	-	-	-
Data Volume	-	-	-	-	-	-	-	-	-	-
Application Type	Very High	-	(-)	-	-	High	-	(-)	-	(-)
Function Runtime	-	High	-	High	-	-	(-)	-	-	-
Is Latency relevant?	-	-	-	-	-	-	-	-	-	-
Motivation	Medium	-	Medium	-	High	High	-	Low	-	Low
Is it a workflow	-	-	-	-	-	-	-	-	-	-
Workflow coordination	-	-	-	-	-	-	-	-	-	-
Workflow structure	-	-	-	-	-	-	-	-	-	-
Workflow size	-	-	-	-	-	-	-	-	-	-

Finding 6: Almost a third (31%) of the serverless applications are workflows. Most workflows are of simple structure, small, and short-lived (**RQ I.1**).

7.1.2 Consensus Analysis

Here, we analyze the degree of agreement between the results from our study and from other studies. This meta-analysis can identify meaningful corroboration between the different studies: For the characteristics that appear both in our study and in others, a high degree of agreement with the existing studies would increase the credibility of these results. A high degree of agreement can also suggest that the results for characteristics that have not yet been investigated by any other study are also credible.

Table 7.1 summarizes the results; the degrees of agreement are defined in Section 4.1.3.3. We find that 8 characteristics are also investigated by other studies, as indicated by rows where very low to very high items appear. Among these characteristics, *platform*, *language*, and *motivation* are analyzed by 4–6 other studies. For 6 characteristics, we are the first to investigate them in peer-reviewed material. For the remaining 2 characteristics, *external services*, and *execution pattern*, our study uses options incomparable with other studies that considered the aspect.

In general, we find a high degree of agreement with the existing studies. For each characteristic where we observe only a low or medium level of agreement with another study, we also observe high or very high agreement with another study, pointing towards differences between these studies. Only for the *motivation* characteristic, there are multiple studies relatively to which we observe low and

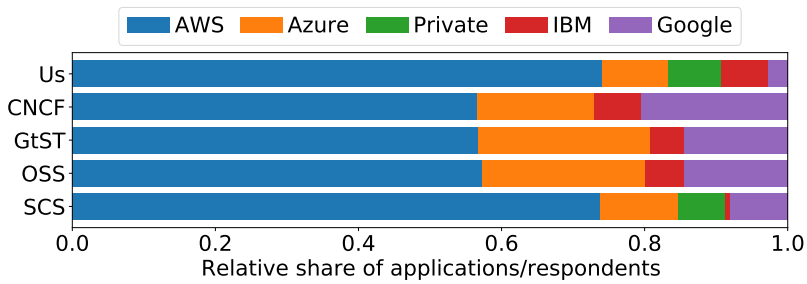


Figure 7.15: Comparison of results for used cloud provider.

medium levels of agreement, suggesting there might be some information that our study missed.

In the following, we provide a qualitative comparison of the results from our study and the comparison studies for the eight characteristics analyzed by at least one of the comparison studies. Here, we focus on points of agreement and disagreement between the studies to obtain corroborating evidence for our findings and identify characteristics that require further investigation.

Platform and Programming Language

Five independent studies indicate that AWS is the most popular serverless provider, followed by Microsoft Azure and Google Cloud. All five studies report a relative share of applications per respondent above 50% for AWS, as shown in Figure 7.15. Azure comes second in all studies except CNCF, where Google Cloud is slightly more popular. Google Cloud is ranked third in all studies except ours, which reports IBM Cloud to be more popular. IBM Cloud is the last major public serverless provider mentioned by all studies. CNCF, GtST, and SCS mention many other serverless platforms such as Cloudflare Workers, Twilio Functions, or Huawei FunctionStage. However, we excluded these hosted platforms due to low popularity (<5%) and only being mentioned by few studies. SCS and our study grouped installable platforms into the private cloud category, including Apache OpenWhisk, Knative, Kubeless, and OpenFaas. For the other studies, the private cloud category could not be calculated due to incompatible reporting.

Figure 7.16 shows six studies agreeing that JavaScript and Python are the dominant programming languages in serverless applications, followed by Java and C#. The tie between JavaScript and Python in our study highlights that both languages are similarly popular across all six studies, with a minor trend towards JavaScript being more popular. Compared to broadly distributed surveys, Java appears to be more popular among enterprise newsletter respondents from GtST and the enterprise-focused survey and interview study MMS. The .NET platform with C# is also present in all six studies but generally less popular than Java. Four studies report Go as a

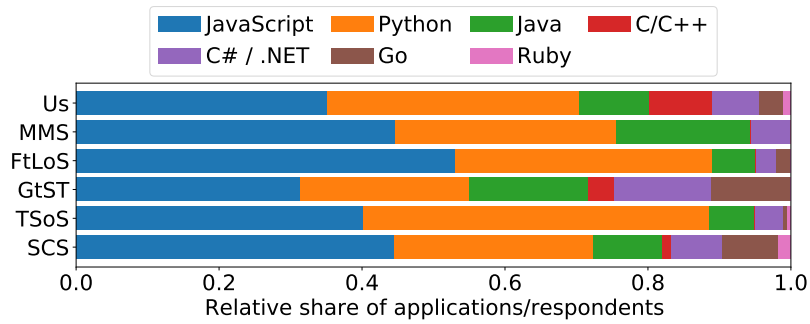


Figure 7.16: Comparison of results for used programming languages.

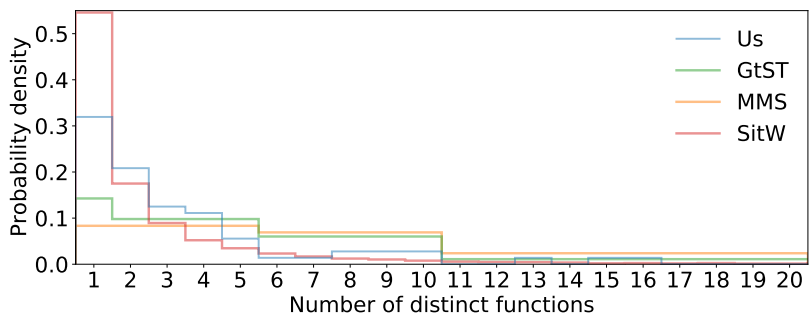


Figure 7.17: Comparison of function numbers per application. The long tail of the distributions is not shown (GtST: 10.3% > 25 functions, MMS: 16% > 20 functions, SitW: 0.25% > 20 functions, Us: 1.1% > 20).

strong contender for catching up with C#. Ruby remains a niche language listed by only three studies. In contrast to other studies, our study found C/C++ to be similarly popular to Java and C#. We assume that other studies mostly ignored this category because it refers to C/C++ binaries running under another officially supported runtime. For GtST and TSoS, the share of programming languages is derived from telemetry data of deployed functions (rather than applications) based on their runtime configuration for the cloud provider AWS.

Consensus 1: AWS is the most popular serverless provider with an over 50% market share, followed by Microsoft Azure and Google Cloud (**RQ I.2**).

Consensus 2: JavaScript and Python are the most popular programming languages for serverless applications. Different studies find next a mix of Java, C#, and C/C++ (**RQ I.2**).

Table 7.2: Comparison of results for trigger types.

Study	HTTP	Event	Scheduled	Manual	Orchestration
SitW	0.641	0.363	0.292	-	0.094
Us	0.474	0.402	0.124	0.093	-

Number of Functions

The number of functions per serverless application was also investigated by GtST, MMS, and SitW. Figure 7.17 shows a histogram of the number of functions determined by each study. The coarse binning used by the surveys prevents a detailed analysis, but a clear trend is visible. Both GtST and MMS find applications to consist of more functions than we do; the difference is larger for MMS. We hypothesize that this is due to differences in survey methodologies. SitW finds more single-function applications than all other studies. As SitW is specific to Azure, while the other studies predominantly cover AWS, we hypothesize that serverless applications on AWS are larger than on other platforms due to the higher maturity of the AWS serverless ecosystem. Despite disagreements on the exact distribution, all studies agree that at least 64% of serverless applications have ten or fewer functions.

Consensus 3: Nearly two-thirds of serverless applications have 10 or fewer functions (**RQ I.2**).

Trigger Types

The only other study on how serverless functions are triggered is SitW. Table 7.2 shows the results for the triggers HTTP request, cloud event, and scheduled triggers. For SitW, we aggregated the results for queue, storage, and event trigger as cloud events, as our definition of cloud event includes those. Both studies agree that HTTP requests and cloud events are the most common triggers for serverless functions. However, SitW finds that scheduled triggers are similarly common, whereas we find them less common than HTTP and event triggers. SitW covers only functions deployed on Azure and reports a larger share of single-function apps than any other study. We hypothesize there is a difference in serverless usage between the providers, with serverless computing being used more for timer-based, single-function utility applications at Azure than at, for example, AWS.

Consensus 4: HTTP requests and cloud events are the most common triggers for serverless functions (**RQ I.2**).

Burstiness

The only other study that included information related to burstiness is the SitW

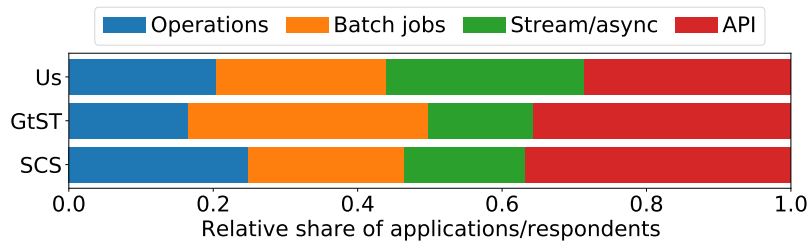


Figure 7.18: Comparison of results for application type.

study. We note that there is no standardized way of characterizing burstiness [GB08; Ali+14]. The SitW study reports the coefficient of variation (c_v) of the inter-arrival times of application invocations, which we used to derive burstiness levels B in terms of the c_v following the metric proposed by Goh and Barabasi [GB08]. The SitW results are in high agreement with the results in our study: both studies agree that more than half of the serverless applications exhibit bursty workloads. Specifically, we found 81% of the applications exhibit bursty workloads, while 57% of the applications from the SitW study exhibited bursty behavior.

Consensus 5: More than 50% of serverless applications have potentially bursty workloads (**RQ I.2**).

Application Type

Comparing the different application type studies is not straightforward, as each study introduces its own classification type. We, therefore, had to map the categorizations of the other studies to match our taxonomy. The details can be found in our replication package. Figure 7.18 shows that GtST and SCS agree with the observation that serverless tasks are used for all areas of computing, including operations, batch jobs, streaming or asynchronous data, or standard API operations. Although the individual percentages differ, for example, GtST and SCS both assign higher importance to API applications and less to stream operations, the overall picture is quite similar. In fact, all studies agree that at least 20% of serverless applications implement operations tasks, batch jobs, async processing, and APIs each.

Consensus 6: There is no dominant application type, but several types are common (**RQ I.2**).

Function runtime

The runtime of serverless functions was not covered in any of the surveys, only the datasets from Azure (SitW), Datadog (TSoS), and New Relic (FtLoS) cover this information. Due to our study methodology, we only estimated if the runtime of

Table 7.3: Comparison of results for function runtime.

Study	Function Runtime	
	<1 min	>= 1 min
SitW	0.960	0.040
TSoS	0.965	0.035
Us	0.750	0.250

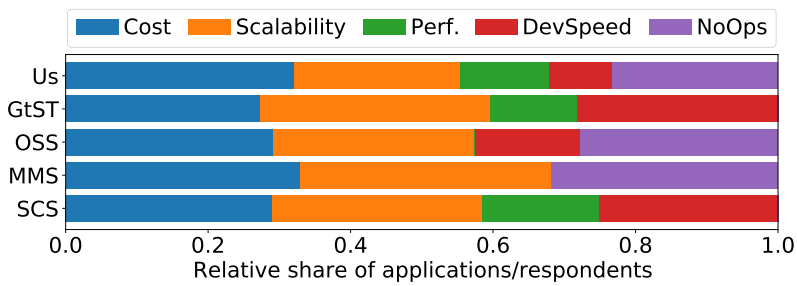


Figure 7.19: Comparison of motivation for building serverless applications.

any function of an application is less than a minute or if it is likely to run longer. Thus, we can not compare our results to the data from FtLoS, as they focus on the runtime distribution below five seconds. As Table 7.3 shows, the studies agree that most functions run for less than a minute, but SitW and TSoS find that over 95% of serverless functions run for less than a minute, compared to the 75% we found. The main reason for this difference should be that SitW and TSoS analyze per function runtime, whereas we analyze how many applications contain one or more functions that run longer than a minute, so the results are not directly comparable. We note that even though TSoS focuses on AWS and SitW on Azure, both agree that over 95% of serverless functions run for less than a minute.

Consensus 7: At least 75% of the serverless functions run for under 1 minute (**RQ I.2**).

Motivation

Figure 7.19 compares the motivations for building serverless applications. The studies had a wide range of possible and overlapping options; we grouped the answers of the comparison studies to fit the options that we identified in our study. This means we compared the studies based on motivations for cost reduction, scalability, performance, developer productivity (DevSpeed), and reduced operational complexity (NoOps). We find that cost reduction and scalability are common and key motivations mentioned in all studies. A reason for this could be that cost reduc-

tion and scalability properties are typical concerns for serverless developers and operators. In contrast, the other motivations—performance, developer productivity, and reduced operational complexity—vary in relative share or are even completely absent in some studies. We believe this is due to the surveys providing options for participants to select from; those options tend to be biased by what the survey is focused on. One survey (GtST) focused heavily on motivations related to the developer productivity, whereas another (MMS) targeted the operation of serverless applications.

Consensus 8: Cost, scalability, and NoOps are major drivers of serverless computing adoption. Some studies also find increased development speed as a major driver of serverless computing adoption (**RQ I.2**).

7.1.3 Replication Package

We include a detailed replication package that provides in-depth details about our methodology and enables the faithful replication of this study at a later time, which will allow to draw conclusions about how the state of serverless applications changed. For the collection of serverless applications, it documents the selection of applications from open-source projects and academic literature based on community datasets. Additionally, it contains descriptions of the scientific computing applications along with our contacts for each application. Finally, it contains a list of all applications collected in this study. For the characterization of serverless applications, it documents the initial reviewer ratings and the scripts to calculate the inter-rater agreement as well as the breakdown of the "Unknown" values. Finally, it contains the characteristics of all surveyed applications and the scripts to reproduce the figures. For the consensus analysis, it documents the search results and the subsequent filtering, the mapping of investigated characteristics to our characteristic framework for every study, and the scripts to calculate the agreement and reproduce all figures. The full replication package is available in the form of a GitHub repository[1]. This comprehensive replication package should provide full transparency for our study and enable the replication at a later point to determine if and how the state of serverless applications has changed.

7.1.4 Summary

In this section, we presented the results for our study addressing **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*). Our analysis spotlights the following main findings that answer **RQ I.1** (*"What are common characteristics of current serverless applications?"*): (I) The most commonly reported reasons for the adoption of serverless computing include cost savings for irregular

[1] https://github.com/ServerlessApplications/ReplicationPackage

or bursty workloads, avoidance of operational concerns, built-in scalability, and increased speed of development, (II) Typical scenarios include short-running tasks with low data volume and bursty workloads, but we also frequently found latency-critical, high-volume core functionality as serverless applications, and (III) Serverless applications are mostly implemented on AWS, in either Python or JavaScript, and use BaaS. Further, we find that seven out of these findings are corroborated by multiple, independent studies, which answers **RQ I.2** (*"Is there a community consensus on the common characteristics of serverless applications?"*). By providing answers to these two research questions, we have achieved **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*).

7.2 Performance Variability of Serverless Applications

In this section, we present the results of our performance variability case study. The first dataset we collected contains multiple repetitions of performance tests under varying configurations to investigate the performance impact of the latter, and the second dataset contains three daily measurements for ten months to create a longitudinal dataset and investigate the stability of performance tests over time. We present our results for **Goal II** (*"Quantify the performance variability that serverless applications experience."*) in the context of the following two research questions, that we defined in Section 1.3:

- **RQ II.1** (*"How much performance variability do common serverless applications experience?"*)
- **RQ II.2** (*"Does the performance of serverless applications change over time?"*)

In the following, Section 7.2.1 introduces the results of the analysis of both performance variability datasets in the context of our research questions. Next, Section 7.2.2 discusses how our findings impact the performance testing stages: design, execution, and analysis. Our replication package is introduced in Section 7.2.3 and finally, Section 7.2.4 summarizes our findings.

7.2.1 Case Study Results

We now present the results of our empirical study in the context of our research questions.

RQ II.1 How much performance variability do common serverless applications experience?

Table 7.4 shows the maximum warm-up period in seconds, observed across all experiments per workload level. In most experiments, we observe that the maximum warm-up period out of the ten repetitions lasts less than 30 seconds (37 out of 48 experiment combinations). With exception of *IngestLoyalty*, all workload classes exhibit a shorter warm-up period as the load increases. The average warm-up period in experiments with 500 requests per second was 27 seconds, half of the warm-up period observed in runs with 5 requests per second (52 seconds). The function *Get Loyalty* never reaches a steady-state under high load, as it implements the performance anti-pattern "Ramp" due to a growing number of entries in the database [SW02]. We also note that, contrary to the workload, the function size (memory size) has no influence on the warm-up period: in most cases, the difference of the warm-up period across function sizes (256 MB, 512 MB, 1024 MB) is not significant ($p > 0.05$), with a negligible effect size for the few significantly different cases ($d < 0.147$). In the following, we opt to conservatively consider the first 2

Table 7.4: Maximum warm-up period in seconds across ten repetitions of all function sizes.

Request Class	Workload (reqs/s)					
	5	**25**	**50**	**100**	**250**	**500**
CollectPayment	15	10	10	10	10	10
ConfirmBooking	25	15	15	65	10	15
CreateStripeCharge	15	15	15	15	20	15
Get Loyalty	70	60	45	30	10	–
IngestLoyalty	15	25	55	75	125	155
List Bookings	115	70	55	45	25	15
NotifyBooking	20	40	40	15	60	10
Process Booking	65	45	20	10	10	15
ReserveBooking	20	20	10	10	10	15
Search Flights	135	80	50	30	30	20

minutes of performance tests as part of the warm-up period for any subsequent analysis.

Result 1: The warm-up period lasts less than 2 minutes in the vast majority of our experiments (**RQ II.1**).

Table 7.5 depicts the average percentage of cold start requests across different request classes, the share of cold start requests that occur in the warm-up period, and whether cold starts after the warm-up period significantly impact the mean response time. We consider cold starts to impact the results if there is a significant difference between the mean response time with and without cold starts in the steady-state phase. As we observe similar results for all request classes, below we only discuss *CollectPayment*. On average, cold start requests in *CollectPayment* make up for 0.93% of the total number of requests. However, since they mostly concentrate in the first two minutes of the experiment (99.5%), they are discarded with the warm-up period. The remaining cold start requests (0.5%) that occur throughout the run of our performance test did not significantly impact the response time (Mann-Whitney U test with $p > 0.05$).

Result 2: The vast majority (>99%) of cold starts occur during the warm-up period. Cold start requests that occur after the warm-up period (<1%) do not impact the measurements (**RQ II.1**).

Given that cold starts occur mostly during the warm-up period, we wanted to assess if the warm-up period is composed solely of cold start requests. Is it enough to simply drop cold start requests from the experiment and consider all other requests as part of the steady-state performance measurements? Table 7.6 shows the difference of

Table 7.5: Average occurrence of cold start requests in the performance tests per request class.

Request Class	% Cold Start	% Occurrence <=2 min	% Occurrence >2 min	Impact?
CollectPayment	0.93	99.5	0.05	No
ConfirmBooking	0.72	99.5	0.05	No
CreateStripeCharge	0.44	99.9	0.01	No
IngestLoyalty	1.01	99.2	0.02	No
NotifyBooking	1.04	99.9	0.01	No
ReserveBooking	0.40	99.8	0.02	No

Table 7.6: Difference of the maximum warmup-period in seconds between experiments including all requests vs. experiments filtering out the cold starts.

Request Class	Workload (reqs/s)					
	5	25	50	100	250	500
CollectPayment	-5	–	-5	–	–	–
ConfirmBooking	–	–	–	–	–	–
CreateStripeCharge	-10	-5	-5	-5	-5	-10
IngestLoyalty	–	–	-10	–	–	-10
NotifyBooking	–	-20	-20	–	–	–
ReserveBooking	-5	-5	–	–	–	–

the warm-up period considering all requests for the workloads that expose cold start information (the one shown in Table 7.4), versus the warm-up period calculated by filtering the cold start requests from the experiment. In the majority of the experiments (22 out of 36 combinations), we observe no difference between dropping or keeping the cold start requests in the duration of the warm-up period. Some request classes, however, exhibited shorter periods of warm-up once we filter out cold start requests, as the high response time of cold start requests contributes to the warm-up period. For instance, the experiment with *CreateStripesCharge* showed a consistent reduction of the warm-up period of at least 5 seconds (our heuristic's window size) for all the workload sizes. It is important to note, however, that the warm-up period — while shorter for some classes — is not only influenced by cold starts.

Result 3: In the majority of experiments, removing the cold starts does not shorten the warm-up period (**RQ II.1**).

Figure 7.20 shows a heat map of the coefficient of variation observed in 10 repe-

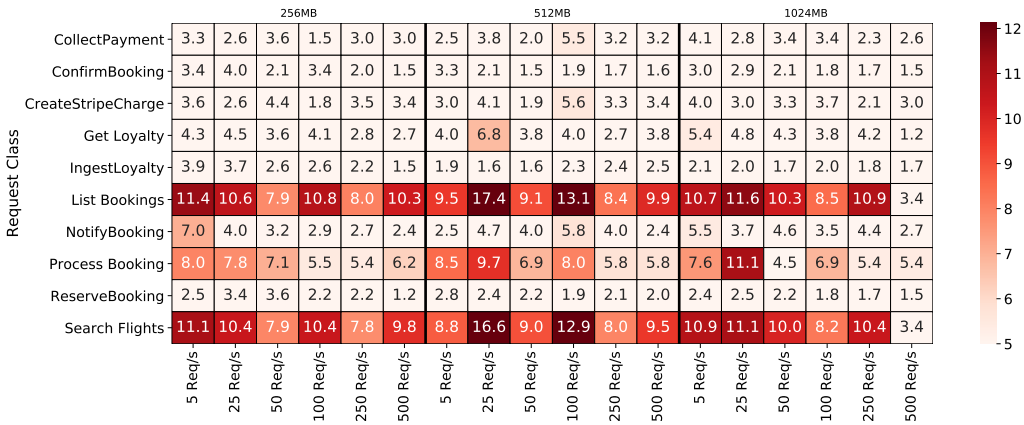


Figure 7.20: Coefficient of variation of the .99 mean across 10 repetitions per request class, load level, and function size.

titions of all experiments. With the exception of three request classes, *List Booking*, *Process Booking*, and *Search Flights*, most of the other experiments show a coefficient of variation of less than 5% of the mean (125 out of the 132 experiments). The observed coefficient of variation is also in line with reported variation in other serverless benchmarks [CSL20], which was reported to be 5 to 10% when executing synthetic workloads in the AWS infrastructure. This suggests that the studied serverless application performance tests are more stable than most traditional performance tests of cloud applications (IaaS). Cito and Leitner [LC16] reported that performance variations of performance tests in cloud environments are consistently above 5%, reaching variations above 80% of the mean in several I/O-based workloads. The two classes with higher variability of the results, *List Booking*, and *Search Flights*, both use an Amplify resolver to retrieve data from DynamoDB. Our findings indicate that this AWS-managed resolver might suffer from a larger performance variability.

Result 4: We observe that the vast majority of experiments (160 out of 180) exhibits a coefficient of variation below 10% of the mean response time (**RQ II.1**).

Figure 7.21 shows the response time of the *ConfirmBooking* request class, in which we observe that as the workload increases, the average response time decreases for all function sizes. This is true across almost all experiments, where the response time observed in the scenario with 500 requests per second is significantly faster than scenarios with only 5 requests per second (Mann-Whitney U $p < 0.05$), often to large effect sizes (Cliff's delta $d > 0.474$). Moreover, the stability of the obtained average response time (across 10 repetitions) also improves slightly, from 4.6% on average across all experiments with 5 reqs/s, to 3.3% on experiments with 500 reqs/s (see Figure 7.22). Our findings suggest that the workload in the studied serverless

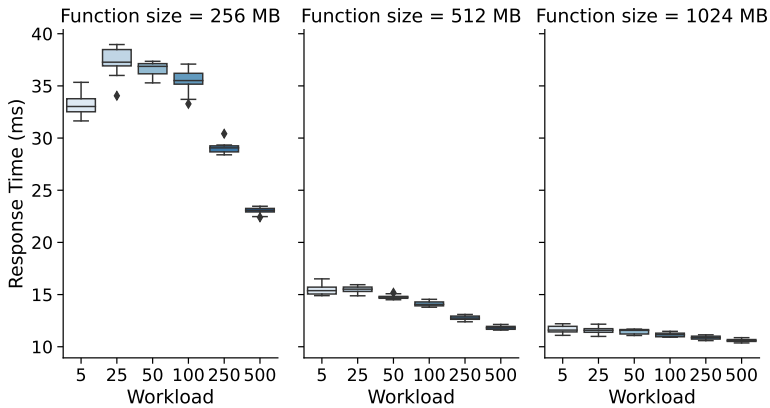


Figure 7.21: Response time of ten repetitions of *ConfirmBooking* performance tests, per workload level and function size.

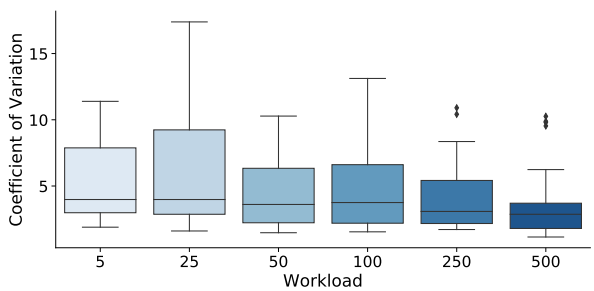


Figure 7.22: Distribution of coefficients of variation across all request classes and function size, per workload (reqs/s).

application showed an inverse relationship to measured performance, that is, the higher the workload we tested the faster was the average response time, the opposite of what is expected in most typical systems (bare-metal, cloud environments). It is important to note that, given the cost model of serverless infrastructure, performance tests with 500 requests per second cost 100 x more than tests with 5 requests per second. Therefore, the small gain in stability is unlikely to justify the much higher costs of running performance tests in practice.

Result 5: We observe improvement in the response time and result stability in scenarios with higher workloads (**RQ II.1**).

We note in Figure 7.21 that the *ConfirmBooking* average response time is considerably faster when the function size is 512 MB or larger. However, we do not observe any significant difference in the stability of the experiments (coefficient of variation)

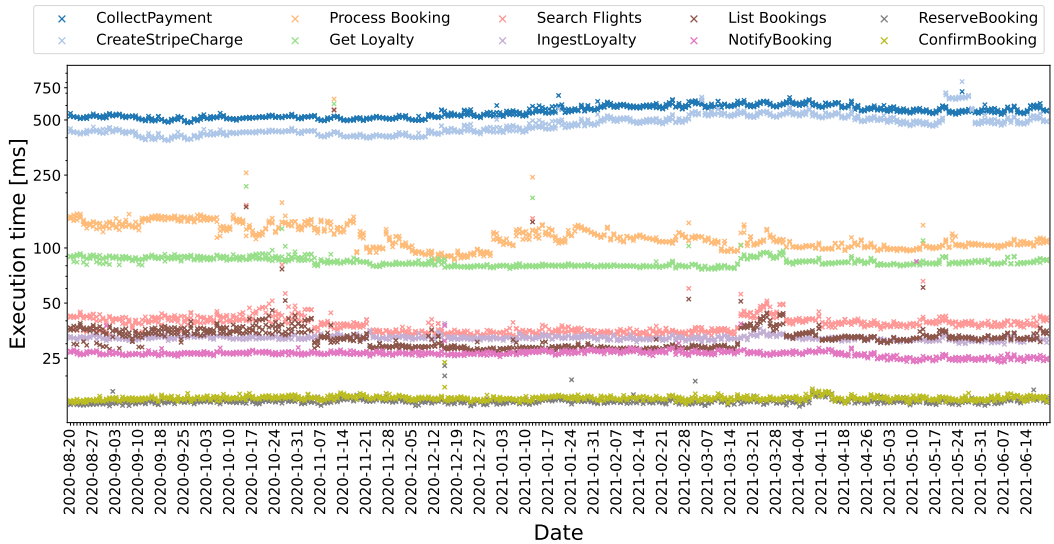


Figure 7.23: Mean response time over a period of ten months.

across different function sizes (Mann-Whitney U test $p > 0.05$). This means that the amount of memory allocated for the function has an impact on its response time (expected), but exerts no significant influence on the stability of experiments.

Result 6: While the response time improves on larger function sizes, the stability of the tests is not affected significantly by the allocated memory (**RQ II.1**).

To summarize, we find that the performance of serverless applications becomes stable quite quickly and that their stability in general is comparable to the stability of traditional software systems (coefficient of variation below 10%), which answers **RQ II.1** (*"How much performance variability do common serverless applications experience?"*)

RQ II.2 Does the performance of serverless applications change over time?

Figure 7.23 presents the average response time of each API endpoint during the study periods. We can clearly observe fluctuations in performance. For example, the response time of the API *Process Booking* has demonstrated large fluctuation after October 2020. Table 7.7 compares the variation of performance between measurements from the same day and across different days (overall) using a Monte Carlo simulation. We find that in all of the API endpoints, the average variation between two random measurements is higher than the variation between measurements from the same day. For example, *Process Booking* has an average variation of 17.2% when considering all measurements, which is more than four times the average variation between measurements from the same day (3.5%).

Table 7.7: Comparison of average performance variation between two measurements from either the same day or different days.

Request class	Same-day Variation	Overall Variation
ConfirmBooking	2.1% ± 2.6%	2.8% ± 3.0%
CreateStripeCharge	2.2% ± 2.1%	13.3% ± 11.0%
Get Loyalty	3.0% ± 20.0%	7.0% ± 22.0%
IngestLoyalty	1.6% ± 1.6%	2.9% ± 2.4%
List Bookings	7.0% ± 59.3%	16.2% ± 65.3%
NotifyBooking	2.3% ± 8.5%	4.3% ± 8.4%
Process Booking	3.5% ± 16.9%	17.2% ± 20.4%
ReserveBooking	2.4% ± 3.2%	3.2% ± 3.8%
CollectPayment	1.9% ± 2.0%	8.3% ± 6.1%
Search Flights	7.1% ± 50.1%	13.2% ± 49.6%

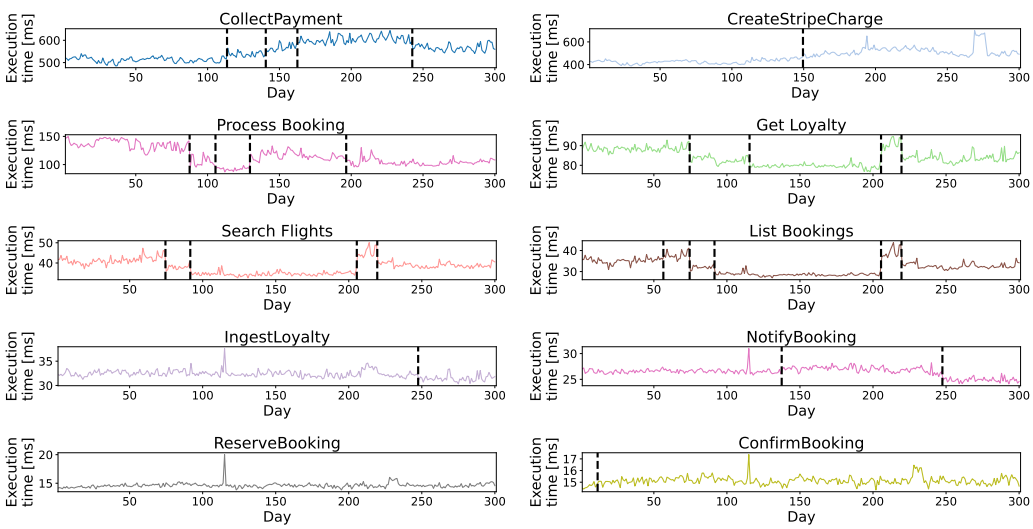


Figure 7.24: Detected change points for each workload class, note the different y-axis scales.

Result 7: There were short-term performance fluctuations during our longitudinal study, despite no changes to the application (**RQ II.2**).

Figure 7.24 presents the detected long-term performance changes in the different APIs, according to the change point detection. Although some API endpoints have more change points than others, all of the API endpoints, except for *ReserveBooking*, have gone through at least one change point (the change point in *ConfirmBooking* might also be a false positive, as it is quite close to the experiment start). There exist as

Table 7.8: Percentage of at least negligible, small, or medium differences according to Mann-Whitney U test and Cliff's delta based on Monte Carlo simulation.

Request class	Negligible+	Small+	Medium+
ConfirmBooking	54.0%	17.4%	6.8%
CreateStripeCharge	11.0%	3.8%	1.7%
Get Loyalty	21.6%	7.1%	2.9%
IngestLoyalty	43.3%	14.0%	5.6%
List Bookings	19.2%	6.8%	3.1%
NotifyBooking	37.1%	11.9%	4.7%
Process Booking	14.0%	5.0%	2.3%
ReserveBooking	57.0%	18.5%	7.0%
CollectPayment	13.2%	4.4%	1.9%
Search Flights	24.3%	7.9%	3.3%

many as five change points during the observation period for an API endpoint. The impact of the performance change may be drastic. For example, the API endpoints *Search Flights* and *List Bookings* have similar performance changes where the response time is drastically reduced twice. On the other hand, the response time of some API endpoints, for example, *CollectPayment*, increases in each change point, leading to a potential unexpected negative impact on the end-user experience. Finally, most of the change points for the different API endpoints do not appear at the same time, which may further increase the challenge of maintaining the performance of the serverless applications.

Result 8: We detect long-term performance changes during the observation period (**RQ II.2**).

Table 7.8 shows the results of conducting the Mann-Whitney U test and measuring Cliff's delta between two groups of consecutive, non-overlapping samples based on our Monte Carlo simulation (described in Section 15). We find that for four API endpoints, almost half of the comparisons have a statistically significant performance difference, even though the serverless application itself was *identical* throughout the observation period. On the other hand, most of the differences have lower than medium effect sizes. In other words, the magnitude of the differences may be small and negligible, such that the impact on end-users may not be drastic. However, there still exist cases whether large effect sizes are observed. Practitioners may need to be aware of such cases due to their large potential impact on end-user experience.

Result 9: Short-term performance fluctuations and long-term performance changes could be considered as false performance regressions (**RQ II.2**).

To summarize, we find that there are impactful short-term and long-term performance changes over longer periods of time for serverless applications (**RQ II.2** (*"Does the performance of serverless applications change over time?"*)).

7.2.2 Impact on Performance Testing Stages

According to Jiang et al. [JH15], performance tests consist of three stages: (1) designing the test, (2) running the test, and (3) analyzing the test results. Based on the findings from our case study, we identified multiple properties of performance tests of serverless applications that practitioners should consider in each of these stages, as shown in Figure 7.25.

7.2.2.1 Design Phase

During the design of a performance test, the key factors are the workload (which types of requests in which order), the load intensity (the number of requests), and the duration of the performance test.

Unintuitive performance scaling (D1). One of the key selling points of serverless platforms is their ability to seamlessly, and virtually infinitely scale with increasing traffic [Eis+21b]. Therefore, the classical approach of running performance tests at increasing load intensities, to see how much the performance deteriorates, becomes obsolete. We find in our experiments that the performance still differs at different load levels, however, and perhaps counterintuitively, the execution time decreases with increasing load. This property impacts how to plan performance tests. For example, a developer might run a performance test at 200 requests per second and find that the performance satisfies the SLA; however when the application is deployed and receives only 100 requests per second, it might violate the SLA. Therefore, developers need to consider that the worst performance is no longer observed at the highest load. Depending on the use case, performance testing strategies could aim to: (a) quantify the expected performance by aiming to match the production load level, (b) understand how different load levels impact the performance by measuring a range of load intensities, or (c) aim to find the worst-case performance with a search-based approach.

Load intensity to cost relationship (D2). Traditionally, the cost of a performance test is independent of the load intensity and depends only on the number of deployed VMs and the duration of the experiment. For a serverless application, this relationship is inverted due to the pay-per-use pricing model of serverless computing. Due to this per-request pricing, the costs of a performance test have a linear relationship to the total number of requests in a performance test, for example, a performance test with 50 requests per second costs ten times as much as a performance test with

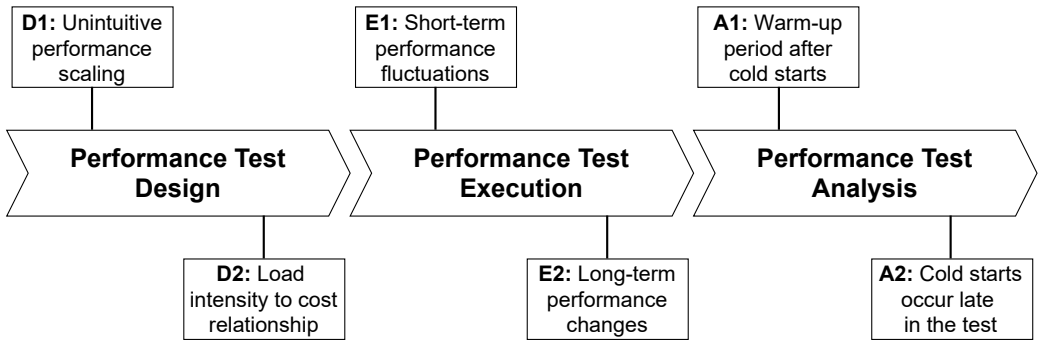


Figure 7.25: Properties of serverless computing that influence the different performance test stages.

5 requests per second. This changes how developers should think about the costs of a performance test. Additionally, increasing the load intensity from five requests per second to 500 requests per second resulted in only a minor increase in result stability in our case study. Therefore, running more repetitions of a performance test at low load intensity instead of a single, large test could result in more stable results at the same cost. However, further experiments in this direction are required to determine how much this increases the result stability.

7.2.2.2 Execution Phase

For the execution of a performance test, performance engineers need to decide when and how the test is executed. The technical implementation of a performance test is mostly unaffected by the switch to serverless applications, as most tooling for the performance testing of HTTP APIs (e.g., for microservice applications) can be reused. However, we find that there are two properties of serverless applications that influence the scheduling of performance tests.

Short-term performance fluctuations (E1). We find that the performance of serverless applications can suffer from short-term (daily) performance variation. While performance variation has also been observed for virtual machines [IYE11; LC16], we find that the variation between measurements conducted on different days is larger than for measurements conducted on the same day for serverless applications. Depending on the goal of a performance test, this has different implications. If the goal is to compare the performance of two alternatives (e.g., to answer the question if the performance of an application changed between two commits), then the measurements for both alternatives should be conducted on the same day. On the other hand, if the goal of a performance test is to quantify the performance of an application, the measurement repetitions should be spread across multiple days as this will result in a more representative performance.

Long-term performance changes (E2). We detect a number of long-term performance changes that caused the performance of the application to permanently change in our case study, despite no changes being made to the application itself. We hypothesize that these performance changes are caused by updates to the software stack of the serverless platform; however, most serverless services do not offer any publicly available versioning that could be used to corroborate this. Unlike the short-term fluctuations, this issue can not be combated by running a larger number of measurement repetitions or by adopting robust measurement strategies such as multiple randomized interleaved trials [AB17]. When comparing two alternatives, they should be measured at the same time to minimize the chance of a long-term performance change occurring between the measurements, which is currently not necessarily the case, for example, for performance regression testing. Quantifying the performance of a serverless application is no longer a discrete task, but rather a continuous process, as the performance of a serverless application can change over time.

7.2.2.3 Analysis Phase

In this phase, the monitoring data collected during the execution phase is analyzed to answer questions related to the performance of the SUT. A key aspect of this phase is the removal of the warm-up period to properly quantify the steady-state performance.

Warm-up period after cold starts (A1). The performance of a serverless application is generally separated into cold starts, which include initialization overheads, and warm starts, which are considered to have reached the steady-state phase and yield a more stable performance. We find that a performance test can still have a warm-up period even after excluding cold starts. A potential reason might be that, for example, caches of the underlying hardware still need to be filled before steady-state performance is reached. This indicates that in the analysis of performance test results, the warm-up period still needs to be analyzed and excluded. For our data, MSER-5, the current best practice to determine the warm-up period [MI04; WCS00], was not applicable due to large outliers present in the data, a well-documented flaw of MSER-5 [SS06]. Therefore, future research should investigate suitable approaches for detecting the warm-up period of serverless applications.

Cold starts occur late in the test (A2). Another aspect about cold starts is that for a constant load, one could expect to find cold starts only during the warm-up period. In our experiments, we found that while the vast majority of cold starts occur during the warm-up period, some cold starts are scattered throughout the experiment. This might be, for example, due to worker instances getting recycled [Llo+18]. While these late cold starts did not significantly impact the mean execution time, they

might impact more tail-sensitive measures such as the 99^{th} percentile. Therefore, performance testers need to keep the possibility of late cold starts in mind while analyzing performance testing results.

7.2.3 Replication Package

Performance measurements of public cloud environments are per definition only a snapshot of the performance at the time of measurement [LC16; AB17; IYE11]. The performance properties can change whenever the cloud provider upgrades its hardware, switches to newer versions of the underlying operating system or virtualization technology, introduces new optimizations or features for the offered services, or changes any number of configuration parameters [Eis+20a; IYE11]. To increase our results' longevity, we provide a replication package that allows other researchers to replicate our findings and enables tracking if and how the reported performance properties evolve over time. This is in line with the recently proposed methodological principles for the reproducible performance evaluation of public clouds by Papadopoulos et al. [Pap+19].

Our replication package[2] consists of two parts: (a) the experiment harness used to run the performance measurements and (b) the data and analysis scripts used in the presented analysis. We provide the experiment harness as a Docker container that replicates all measurements conducted in this study with a single CLI command from any Docker-capable machine. To simplify the reuse of this harness in other studies, experiments can be specified as JSON files, including measurement duration, load intensity, load pattern, measurement repetitions, and system configuration. The second part of our replication package is a CodeOcean capsule containing the collected measurement data and the scripts for the analysis presented in this paper. The CodeOcean capsule enables a one-click replication of our analysis either on the measurement data we collected or on new measurement data collected using our measurement harness.

7.2.4 Summary

In this section, we present the results for our study towards **Goal II** (*"Quantify the performance variability that serverless applications experience."*). Among other things, we find that the performance of serverless applications becomes stable quite quickly and that their stability in general is comparable to the stability of traditional software systems (coefficient of variation below 10%), which answers **RQ II.1** (*"How much performance variability do common serverless applications experience?"*). In regard to **RQ II.2** (*"Does the performance of serverless applications change over time?"*), we find that there are both short-term and long-term performance changes over longer periods of time for serverless applications. Further, we find that there are serverless

[2] `https://github.com/ServerlessLoadTesting/ReplicationPackage`

computing-specific changes and pitfalls to all performance test phases: design, execution, and analysis. These insights show that we have addressed both research questions and therefore achieved **Goal II** (*"Quantify the performance variability that serverless applications experience."*).

Chapter 8

Evaluating the Automation of Operational Tasks of Serverless Applications

8.1 Serverless Function Size Optimization

In this chapter, we present the evaluation of our approach for the serverless function size optimization. We design our evaluation to answer the following three evaluation questions to determine if we achieved **Goal III** (*"Develop an automated method to optimize the size of serverless functions."*):

- **EQ III.1**: *Can our model, trained on a synthetic dataset, accurately predict the execution time of realistic serverless functions?*
- **EQ III.2**: *Are the execution time predictions provided by our approach sufficient to determine the optimal memory size of serverless functions?*
- **EQ III.3**: *How large are the benefits in terms of decreased cost and execution time of our proposed approach?*

In the following, Section 8.1.1 introduces the four realistic serverless applications we used to evaluate our approach. Next, Section 8.1.2 analyzes the accuracy of our execution time predictions, Section 8.1.3 investigates the memory size optimization, and Section 8.1.4 evaluates the obtained cost savings and execution time speedups. Our replication package is introduced in Section 8.1.5 and finally, Section 8.1.6 summarizes our findings.

8.1.1 Evaluation Systems

In order to investigate these research questions, we conduct performance measurements for the following four case study systems:

Airline Booking This application was the subject of the AWS Build On Serverless series [Ser19] and presented at AWS re:Invent as an example for the implementation

of a production-grade full-stack app using AWS Amplify [Les19]. The airline booking application implements the flight booking aspect of an airline. Customers can search for flights, book flights, pay using a credit card, and earn loyalty points with each booking. It consists of eight serverless functions, the managed services S3, SNS, Step Functions, and API Gateway, as well as an external payment provider. The workload consists of 200 requests per second that sequentially access all application features for ten minutes. The measurements for this case study were conducted in October 2020, so two months after the training dataset and incurred costs of ~500$.

Facial Recognition This case study is part of the AWS Wild Rydes workshop and was also used in the evaluation of Costless [Elg18], another cost optimization approach for serverless functions. In this application, users of a fictional transportation app, Wild Rydes, upload their profile picture, which triggers the execution of a workflow that performs facial recognition, matching, and indexing. It consists of six serverless functions, however, we removed the notification function as it is a no-op stub. This application makes heavy use of AWS Rekognition, a service that was not contained in our function segments. The workload consists of only ten requests per second for five minutes, as AWS Rekognition is comparatively expensive. Therefore, for this case study, our approach will have less monitoring data available to learn from. The measurements for this case study were conducted in December 2020, so four months after the training dataset and incurred costs of ~400$.

Event Processing This application was introduced in [Yus+19], where the authors investigate the challenges of migrating serverless applications across cloud providers by migrating different systems across multiple cloud providers. For our case study, we use the AWS implementation of the IoT-inspired event processing system, where the data obtained from multiple sensors are aggregated for further processing. It consists of seven serverless functions and uses the API Gateway, SNS, SQS, and AWS Aurora, none of which are used in our function segments. Compared to the other two applications, the functions of this application exhibit very fast execution times. The workload consists of 10 requests per second that sequentially access all application features for ten minutes. The measurements for this case study were conducted in December 2020, so four months after the training dataset and incurred costs of ~50$.

Hello Retail This application from the online retailer Nordstrom won the inaugural serverless architecture competition at Serverlessconf Austin [McK17]. The application models the product catalog of an online shop. Notably, it includes a workflow for the addition of new products that outsources the product image acquisition to photographers. This application consists of seven functions and uses Kinesis, API Gateway, Stepfunctions, DynamoDB, and S3. The workload consists of 10 requests per second that sequentially access all application features for ten

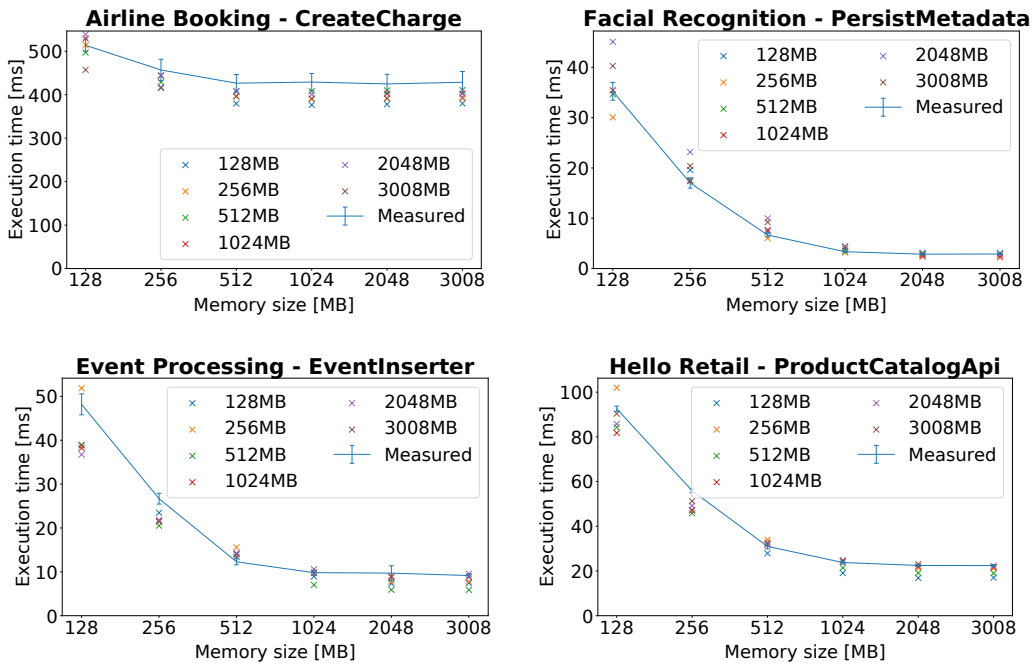


Figure 8.1: Example for the measured and predicted execution time for a serverless function of each serverless application.

minutes. The measurements for this case study were conducted in May 2021, so nine months after the training dataset and incurred costs of ~30$.

For all four case studies, we conducted ten measurement repetitions for each memory size to account for cloud performance variability, in line with existing guidelines for performance measurements of cloud applications [Pap+19]. We run the experiments as randomized multiple interleaved trials, which has been shown to reduce performance variability [AB17]. To reduce the chance of human error and enable the replication of our measurements, we implemented automated measurement harnesses for each case study, which are available in our replication package[1]. Considering the number of functions and external services used, these applications have an above average complexity [Sha+20; Eis+21b].

8.1.2 Execution Time Prediction

The underlying idea of our approach is to learn how memory size impacts the execution time of a function based on a large dataset obtained from synthetic functions

[1] `https://github.com/Sizeless/ReplicationPackage`

Table 8.1: Relative prediction error based on monitoring data from 256MB for the **airline booking application**.

Targetsize	128	512	1024	2048	3008
IngestLoyalty	10.2	4.7	16.8	19.6	19.8
CaptureCharge	0.4	7.3	8.8	7.6	8.4
CreateCharge	0.1	6.8	9.2	7.7	8.6
CollectPayment	26.1	16.8	26.7	30.3	29.2
ConfirmBooking	5.7	9.6	11.8	6.1	5.0
GetLoyalty	11.2	16.6	27.3	32.7	33.1
NotifyBooking	1.3	1.1	1.3	4.1	4.3
ReserveBooking	1.3	11.6	16.3	11.8	8.3
All functions	7.0	9.3	14.8	15.0	14.6

Table 8.2: Relative prediction error based on monitoring data from 256MB for the **facial recognition application**.

Targetsize	128	512	1024	2048	3008
FaceDetection	0.8	0.9	15.8	2.8	1.3
FaceSearch	5.6	5.1	20.7	15.2	13.5
IndexFace	12.0	18.9	9.2	17.0	18.8
PersistMetadata	14.7	9.5	4.3	6.9	9.7
CreateThumbnail	30.7	6.5	25.1	10.4	6.4
All functions	12.7	8.2	15.0	10.5	9.9

with realistic resource consumption profiles. To investigate if the knowledge our model learned from the synthetic dataset can be transferred to realistic functions, we train the model on the synthetic dataset as described earlier and ask it to predict the execution time of the twenty-seven serverless functions from our four case study systems across all memory sizes based on monitoring data obtained from a single memory size. Figure 8.1 shows the measured execution time and standard deviation across ten measurement repetitions for a function from each application in blue. The colored crosses show the predictions for different basesizes. The individual functions scale differently with increasing memory sizes, and our approach is generally able to predict these differences quite well. It also shows that there are some inaccuracies, especially when predicting the execution time of 128MB. The graphs for the remaining nineteen functions can be viewed in the CodeOcean capsule of our replication package.

Table 8.1 shows the relative prediction error of the airline application for a base

Table 8.3: Relative prediction error based on monitoring data from 256MB for the **event processing application**.

Targetsize	128	512	1024	2048	3008
EventInserter	7.6	26.8	0.4	19.8	14.8
FormatForecast	7.7	9.1	7.1	4.6	6.3
FormatState	8.9	2.2	6.1	5.6	8.2
FormatTemp	3.1	3.4	9.6	5.5	9.7
GetLatestEvents	23.8	56.5	70.9	50.3	47.6
ListAllEvents	18.0	34.3	119.4	131.9	132.2
IngestEvent	10.5	11.5	16.1	20.8	20.9
All functions	11.4	20.5	32.8	34.1	34.2

Table 8.4: Relative prediction error based on monitoring data from 256MB for the **hello retail application**.

Targetsize	128	512	1024	2048	3008
EventWriter	5.7	2.1	4.9	7.8	7.3
PhotoAssign	0.8	1.4	1.0	0.2	0.9
PhotoProcessor	32.6	21.4	42.2	53.3	52.9
PhotoReceive	6.5	0.1	3.1	4.6	2.6
PhotoReport	2.0	6.5	11.1	16.2	18.4
ProductCatalogApi	10.2	9.7	1.0	5.2	5.6
ProductCatalogBuilder	11.0	7.3	2.8	14.5	16.3
All functions	9.8	6.9	9.4	14.5	14.8

size of 256MB, which we identified as the preferred basesize as described in Section 5.1.3.1. Our approach is able to predict the execution time for 128MB and 512MB accurately with below 10% relative error, and the relative prediction error increases to around 15% for the execution time with 1024MB, 2048MB, and 3008MB. The predictions for the functions of the facial recognition application (Table 8.2) show similar prediction accuracies with 512 MB and 3008 MB below 10% relative error and the other three memory sizes with below 15% error. For the functions of the event processing application, the predictions for 128MB again exhibits a relative prediction error of around 10%, however, the relative prediction error for 512MB, 1024MB, 2048MB, and 3008MB is quite large with 20-35%, as shown in Table 8.3. This is mostly due to a single function where the predictions are over 100% off. While the very high relative prediction error is partly due to the low absolute values at higher memory sizes (prediction ∼40ms, real ∼20ms), the approach also underestimates

how well the function scales with additional resources. Finally, the prediction error for the hello retail application is below 10% for 128MB, 512MB, and 1024MB and below 15% for 2048 MB and 3008 MB.

Result 1: Across 27 serverless functions, our approach achieved an average prediction error of 15.3% (**EQ III.1**).

8.1.3 Memory Size Optimization

The goal of our approach is to select the optimal memory size for a serverless function after monitoring it with a single base size. Therefore, the goal of this experiment is to verify, whether our approach is applicable to find the optimal memory size based on the execution time predictions for the previously unobserved memory sizes.

To investigate if the predictions are accurate enough to determine the optimal memory size, we apply the optimization approach from Section 5.1.3.2 using the execution time predictions and compare the selected memory size to the optimal memory size determined based on the measured execution time. We run the optimization for three different tradeoff parameters, $t = 0.75$ which prioritizes cost, $t = 0.5$ which shows no preference, and $t = 0.25$ which prioritizes performance. Figure 8.2 shows the ranking of the selected memory sizes for each tradeoff parameter grouped by application. For a tradeoff parameter of 0.75, our approach selects the optimal memory size for 74.0% of the functions, for a tradeoff parameter of 0.5 it selects the optimal memory size for 81.4% of the functions, and for a tradeoff parameter of 0.25, it also selects the optimal memory size for 81.4% of all functions. If our approach does not select the optimal memory size, it usually selects the second-best memory size and only rarely the third or fourth-best memory size.

Result 2: Our approach selects the optimal memory size for 79.0% of the serverless functions and the second-best memory size for 12.3% of the serverless function (**EQ III.2**).

8.1.4 Cost Savings and Speedup

We have shown that our approach is capable of selecting either the optimal or a close to optimal memory size. In a next step, we are investigating what the actual benefits of using the memory sizes selected by our approach are, that is how much costs can be saved and how much the function execution can be sped up.

To quantify these benefits, we calculate the relative change in cost and execution time between the memory size selected by our approach for the tradeoff factors of 0.75, 0.5, and 0.25. Table 8.5 shows the average percentage cost savings and execution time speedup obtained by switching to the memory sizes recommended by our approach. For $t = 0.5$, the cost increase by 12.0%, but the average function execution is speed up by 46.7%. If the optimization is configured to favor speed

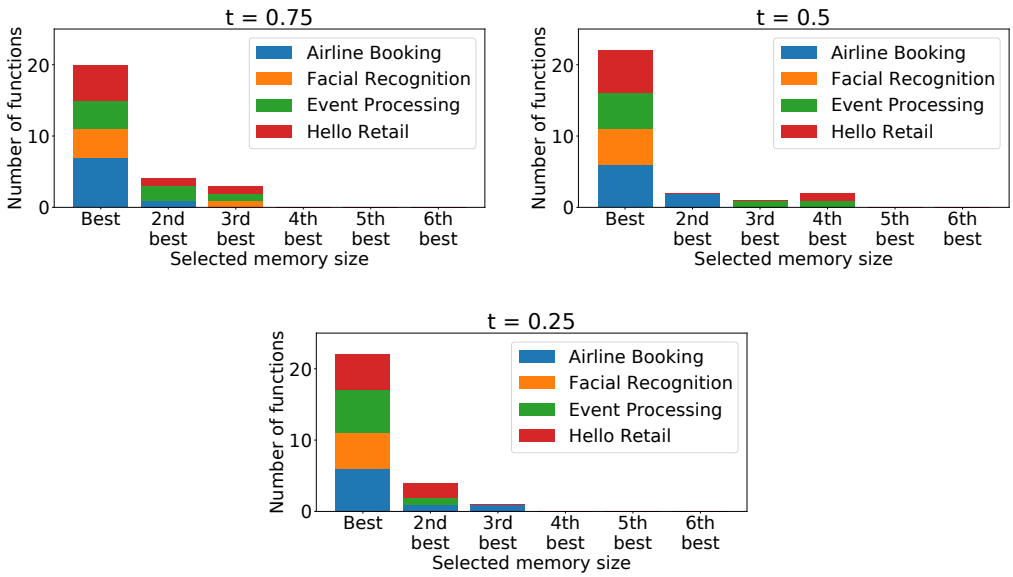


Figure 8.2: Number of functions for which our approach selects the X best approach for three different tradeoff parameters.

Table 8.5: Cost savings and application speedup for four applications using our approach.

Application	t = 0.75		t = 0.5		t = 0.25	
	Cost savings	Speedup	Cost savings	Speedup	Cost savings	Speedup
Airline Booking	15.6%	28.5%	4.5%	31.9%	-12.3%	34.1%
Facial Recognition	-2.9%	67.5%	-2.9%	67.5%	-17.4%	70.6%
Event Processing	2.8%	31.2%	-17.0%	47.8%	-30.5%	57.5%
Hello Retail	-8.4%	41.1%	-32.4%	47.7%	-63.8%	55.7%
All Applications	2.6%	39.7%	-12.0%	46.7%	-31.3%	52.5%

over cost ($t = 0.25$), the execution is sped up further (52.5%) and the cost increases by 31.3%. If the optimization is configured to favor cost over speed ($t = 0.75$), the cost savings increase to 2.6%, and the achieved speedup decreases by about seven percent to 39.7%. This also shows that the tradeoff parameter correctly influences the behavior of the optimization. As cost savings from the memory size optimization are generally lower than execution time speedups, it seems that $t = 0.75$ achieves the most balanced optimization result. Therefore, we would recommend using this tradeoff factor for the automated memory size optimization.

Result 3: Applying our approach to four realistic serverless applications saves on average 2.6% costs and speeds up the functions by 39.7% (**EQ III.3**).

8.1.5 Replication Package

To facilitate replication of our results and extensions of our approach, we attempt to provide an extensive three-part replication package. This is in line with a recent guideline on reproducible measurements in cloud environments [Pap+19]. The first part of our replication package consists of docker containers containing the benchmark harnesses for the airline booking, facial recognition, and event processing applications that require only the injection of valid AWS credentials to reproduce the measurement results shown in our evaluation. The second part of our replication package consists of the function generator and resource consumption monitoring introduced in this evaluation alongside instructions to reproduce the measurements to create our training dataset. The third part is a CodeOcean capsule containing the generated dataset covering 12 000 performance measurements (216 000 000 lambda executions) as well as the implementation of the multi-target regression modeling used in our approach. This enables a 1-click replication of all analyses conducted in the scope of this work, and all tables/figures presented in the evaluation. The replication package is available in form of a Github repository[2]. Our replication package was evaluated at the artifact evaluation track of the International Middleware Conference (MIDDLEWARE) and has received the *ACM Artifact Evaluated - Available* and *ACM Artifact Evaluated - Functional* badges.

8.1.6 Summary

In this section, we evaluated whether the approach proposed in Section 5.1 fulfills **Goal III** (*"Develop an automated method to optimize the size of serverless functions."*) using four realistic serverless applications. We answer **EQ III.1** (*"Can our model, trained on a synthetic dataset, accurately predict the execution time of realistic serverless functions?"*) by showing that our approach achieved an average prediction error of 15.3% across 27 serverless functions. It selects the optimal memory size for 79.0% of the serverless functions and the second-best memory size for 12.3% of the serverless function, which addresses **EQ III.2** (*"Are the execution time predictions provided by our approach sufficient to determine the optimal memory size of serverless functions?"*). Finally, we cover **EQ III.3** (*"How large are the benefits in terms of decreased cost and execution time of our proposed approach?"*) by showing that using our approach to optimize the memory size of four realistic serverless functions saves on average 2.6% costs and speeds up the functions by 39.7%. Answering these three evaluations questions shows that our approach achieves **Goal III**. By automating the memory size optimization for serverless functions, our approach removes a resource management task that developers still need to deal with in serverless functions and thus makes serverless functions more serverless.

[2] `https://github.com/Sizeless/ReplicationPackage`

8.2 Cost Optimization of Serverless Workflows

In this chapter, we present the evaluation of our approach for the cost optimization of serverless workflows. We design our evaluation to answer the following three evaluation questions to determine if we achieved **Goal IV** (*"Provide a technique to estimate the costs of serverless workflows."*):

- **EQ IV.1**: *Are mixture density networks capable of accurately predicting the distribution of the response time and the output parameters of a serverless function?*
- **EQ IV.2**: *Can the proposed algorithm and the underlying machine learning models for the individual functions accurately predict the costs of a previously unobserved workflow?*
- **EQ IV.3**: *What is the required time for training and workflow prediction? Is the overhead feasible for a production environment?*

In the following, Section 8.2.1 introduces the serverless functions and workflows we used to evaluate our approach. Next, Section 8.2.2 evaluates the accuracy of response time and parameter distribution predictions, Section 8.2.3 evaluates the accuracy of the cost predictions for serverless workflows, and Section 8.2.4 analyzes the time requirements of our approach. Our replication package is introduced in Section 8.2.5 and finally, Section 8.2.6 summarizes our findings.

8.2.1 Evaluation Setup

We implemented the following five audio utility functions on Google Cloud Functions using Python with the following input and output parameters:
Text2Speech Transcribes text files into audio files, a functionality that is commonly used to increase accessibility, automate phone banking, or in smart home assistants like Alexa or Google Assistant. It uses the google text-to-speech Python library `gTTS` (v2.0.3), which returns an MP3 file. This function has the following input and output parameters:

- **[Input]** *TextLength:* Length of the text that needs to be transcribed, measured in number of characters.
- **[Output]** *FileSize:* Size of the resulting MP3 file in bytes.

ProfanityDet Detects racial slurs, sexually explicit language, and general expletives in a text segment. The implementation is based on the Python library `profanity` (v1.1.0), which implements a blacklist-based filter. This function has the following input and output parameters:

- **[Input]** *TextLength:* Length of the text in which the profanities are detected, measured in number of characters.

- **[Output]** *ProfanityCount:* Number of detected profanities alongside their location in the text.

Conversion Converts an MP3 file to a WAV file. This conversion tends to increase the file size, but many applications require raw WAV files as input. The conversion is performed using the Python library `pydub` (v0.23.1), a wrapper for the `ffmpeg` library, which is available in all Google Cloud function instances. This function has the following input and output parameters:

- **[Input]** *FileSize:* Size of the MP3 file that is converted.
- **[Output]** *FileSize:* Size of the resulting WAV file in bytes.

Censor Censors segments of a WAV file, based on a list of time segments that should be censored. For the censoring, all samples within the segments that are censored are muted using the `pydub` (v0.23.1) library. This function has the following input and output parameters:

- **[Input]** *FileSize:* Size of the file that is censored in bytes.
- **[Input]** *ProfanityCount:* Number of detected profanities.
- **[Output]** *FileSize:* Size of the censored audio file in bytes.

Compression Compresses a WAV audio file by reducing the sampling rate and sample width. On an initial set of audio files, the compression achieves compression rates of about 60-90%. This function has the following input and output parameters:

- **[Input]** *FileSize:* Size of the audio file prior to compression, measured in bytes.
- **[Output]** *FileSize:* Size of the audio file after compression, measured in bytes.

We deploy each function to Google Cloud Functions with 512 MB memory, the Python 3.7 runtime, and a timeout of 60 seconds. The implementations of these functions are available as part of our replication package. In the following, we first investigate the capability of the proposed mixture density networks to accurately predict the distribution of a function's response time and its output parameters. Next, we apply our cost-prediction algorithm to two distinct workflows composed of these functions and compare the cost predictions to the actual observed costs.

8.2.2 Response Time and Parameter Distribution Predictions

As described in Section 5.2.2, the mixture density networks can be trained on monitoring data collected during function operation or using microbenchmarks. In this case study, we use microbenchmarks to create the data set for the training of the mixture density networks as these functions are currently not deployed in production. The workload for each function consists of 50 requests/second with varying input parameters. The monitoring data for the first three minutes of a measurement is discarded as a warm-up phase. The next ten minutes are used as training data for the mixture density network models. For the evaluation in this paper, we additionally collect the monitoring data of the following 50 minutes as our validation data set.

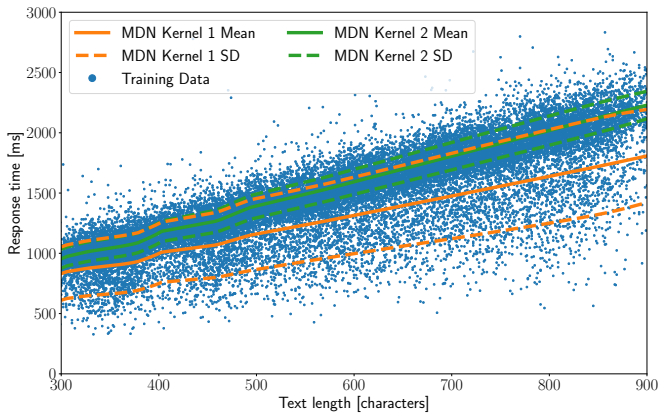


Figure 8.3: MDN model for the response time of the *Text2Speech* function.

This larger validation data set is only required for the evaluation presented in this paper and is not necessary to apply our approach in practice. For the experiments presented in this paper, we parameterize the network as following. We use the Adam optimizer [KB14] with a learning rate of 0.001. We train for 500 epochs with a batch size of 32 and to prevent overfitting an early stopping criterion terminates the training process if the negative log-likelihood does not decrease by more than 0.001 for 10 epochs [YRC07]. Additionally, we apply model check-pointing to save the best model achieved during training as the model accuracy decreased at times during the training process. In order to determine the appropriate number of kernels for the gaussian mixture model, we apply basic hyper-parameter optimization based on the observed negative log-likelihood during model training to select between 1, 2, 3, 4 or 5 kernels.

As an example for the resulting models, Figure 8.3 shows how the mixture density networks fit the training data for the response time of the *Text2Speech* function. The input parameter TextLength varies from roughly 300 to 900 characters and the resulting response time ranges from roughly 400 ms up to 3000 ms. There is a clear correlation between the length of the transcribed text and the response time of the *Text2Speech* function. However, for each input text length, a broad range of response times is observed. The mixture density network describes this distribution using two normal distributed kernels. The green normal distribution (MDN Kernel 2) is used to fit the bulk of occurring response times and the orange normal distribution (MDN Kernel 1) is used to describe the scattered lower response times. Note that these two distributions are not weighted equally, instead, the green kernel has a larger weight than the orange kernel.

Figure 8.4 shows how the predicted distributions (in red) compare to the observed empirical distributions from the validation data set for text lengths of 300, 600, and 900. To derive the empirical distributions we apply a gaussian kernel density

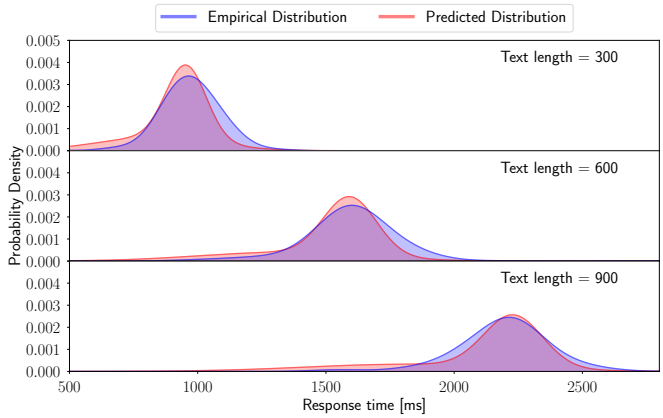


Figure 8.4: Comparison of measured response time distribution in the validation data set and predicted response time distribution for three different text lengths.

Function	Parameter	1 kernel	2 kernels	3 kernels	4 kernels	5 kernels
Text2Speech	Response time	5.3%	4.2%	4.1%	6.4%	**4.5%**
Text2Speech	FileSize	0.6%	0.3%	1.1%	**0.4%**	0.6%
Conversion	Response time	13.2%	38.3%	3.4%	3.3%	**3.3%**
Conversion	FileSize	0.9%	**1.2%**	7.8%	9.0%	16.4%
Compression	Response time	13.1%	4.3%	5.2%	4.4%	**3.6%**
Compression	FileSize	0.2%	1.7%	0.4%	**0.2%**	3.5%
ProfanityDet	Response time	38.7%	32.9%	12.8%	9.4%	**4.6%**
ProfanityDet	ProfanityCount	14.5%	69.0%	12.8%	**12.3%**	14.0%
Censor	Response time	9.5%	10.1%	8.5%	**8.2%**	9.1%
Censor	FileSize	**1.0%**	0.6%	0.7%	1.5%	7.9%

Table 8.6: Relative Wasserstein distance [%] between validation dataset and predictions of MDNs with 1-5 kernels. Kernel count selected by hyperparameter optimization highlighted in bold.

estimation [BGK+10] with a bandwidth of 0.4. The predicted distributions capture both the mean of the empirical distribution and the shape of the distribution well. The shape of the predicted distributions is slightly left heavy compared to a normal distribution due to the addition of the second kernel.

Next, we investigate the impact of the number of kernels used in the mixture density network. Table 8.6 shows the prediction error of mixture density networks with one to five kernels for the response time and output parameter distributions of the five functions. As a measure for the similarity of two distributions, we use the

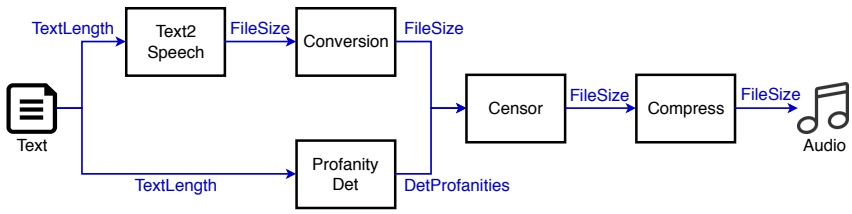
Wasserstein metric [AGS08] defined for two distributions u and v as:

$$l(u, v) = \int_{-\infty}^{\infty} |U - V| \tag{8.1}$$

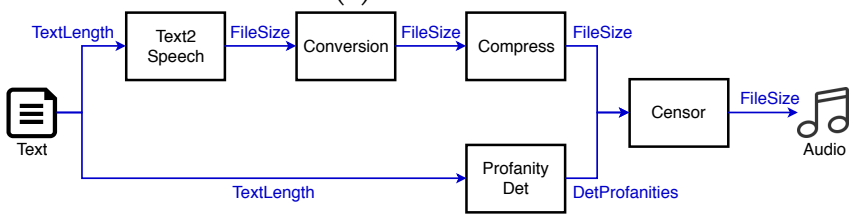
with U and V as the cumulative distribution function of u and v, respectively. Generally speaking, the Wasserstein metric quantifies how far a sample from a set of samples drawn from u has to be moved on average in order to transform the set of samples drawn from u to a set of samples drawn from v. As the absolute values of the Wasserstein metric are difficult to interpret, we calculate the relative Wasserstein metric by dividing the absolute Wasserstein metric by the mean of the empirical distribution as proposed in [Maj+18]. This relative Wasserstein metric enables us to quantify the prediction accuracy for a single input value. As a mixture density network predicts a different distribution for each input value, we calculate the weighted average over all values of the input distribution with the number of empirical samples as a weight.

Table 8.6 shows the weighted average of the relative Wasserstein metric of mixture density networks with one to five kernels for the response time and output parameter distributions of the five functions and the kernel number selected by the hyper-parameter optimization. Generally, it seems that response time distributions are harder to predict than output parameter distributions. This is intuitive, as the output parameter of a function for a certain input is often constant, for example transcribing a text segment multiple times results in the same audio file each time, but the response time varies between executions. An outlier in this regard is the output parameter ProfanityCount of the function *ProfanityDet*, which has a higher error compared to the other parameters. This does not necessarily indicate a bad model fit as the target parameter ProfanityCount is an integer value of less than ten in most cases. The relative Wasserstein metric assigns high percentage errors for even small deviations between integer distributions with a small range of values that also includes zero.

Regarding the kernel count, these results show that there is no kernel count that is ideal for every function. While three to five kernels seem to generally produce accurate performance predictions, the prediction error for the FileSize parameter of the *Conversion* function with five kernels is 16.4%, while using two kernels results in a prediction error of 1.2%. This shows that selecting an individual number of kernels for each function is necessary. The hyper-parameter optimization based on the observed negative log-likelihood during model training reliably selects a kernel count that provides accurate predictions. In four out of ten scenarios, the hyper-parameter optimization does not select the ideal number of kernels, but the prediction accuracy of the selected kernels is always within one percentage point of the ideal kernel count.



(**a**) WorkflowA.



(**b**) WorkflowB.

Figure 8.5: The two alternatives for the transcription and censoring workflow. The blue arrows indicate parameters passed to a function from a previous function.

Result 1: Across all functions, response time, and parameter distributions, the mixture density network models selected by the hyper-parameter optimization achieve a prediction accuracy of 96.1% (**EQ IV.1**).

8.2.3 Workflow Cost Predictions

For the evaluation of our workflow cost prediction algorithm, we consider the following scenario: A workflow designer is looking to build a workflow that turns short text segments into speech and censors any profanities within the text segment. For this task, he comes up with the two different workflows shown in Figure 8.5. In both workflows, the input text is first passed to the *Text2Speech* function and then converted to a WAV file using the *Conversion* function. In parallel, the text is also passed to the *ProfanityDet* function in order to identify any profanities within the text. In WorkflowA , any identified profanities are censored first using the *Censor* function and afterwards the audio file is compressed. In WorkflowB, the audio file is compressed prior to the censoring. While it is a reasonable assumption that WorkflowB might be cheaper, manually quantifying the cost difference is currently challenging for a workflow designer.

We apply the algorithm proposed in Section 5.2.3 in combination with the mixture density network models with two kernels from Section 8.2.2. To measure the actual execution cost of both workflow alternatives, we implement both workflows using Google Cloud Composer (a managed Apache Airflow service). The implementation of the workflows is available as part of our replication package. At the time of

Workflow	Metric	Invocations	CPU Time	Memory Time	**Total**
WorkflowA	Measured cost [cent]	$2.00 * 10^{-6}$	$8.60 * 10^{-5}$	$1.40 * 10^{-5}$	$\mathbf{1.02 * 10^{-4}}$
WorkflowA	Predicted cost [cent]	$1.79 * 10^{-6}$	$9.08 * 10^{-5}$	$1.35 * 10^{-5}$	$\mathbf{1.06 * 10^{-4}}$
WorkflowA	Relative prediction error	10.3%	5.5%	3.4%	**4.0%**
WorkflowB	Measured cost [cent]	$2.00 * 10^{-6}$	$3.80 * 10^{-5}$	$6.00 * 10^{-6}$	$\mathbf{4.60 * 10^{-5}}$
WorkflowB	Predicted cost [cent]	$1.79 * 10^{-6}$	$3.70 * 10^{-5}$	$5.52 * 10^{-6}$	$\mathbf{4.43 * 10^{-5}}$
WorkflowB	Relative prediction error	10.3%	2.6%	8.0%	**3.6%**

Table 8.7: Comparison between measured and predicted cost for a single workflow execution for both workflows in EUR.

writing, the billing reporting for Google Cloud Functions is quite coarse-grained. An example of the most detailed reporting currently possible: On June 7th, 2019 you paid 43.7$ for 4,370,000 GHz-seconds CPU time of Cloud Functions, 30.5$ for 12,200,000 GB-seconds memory time of Cloud Functions, and 5.5$ for 13,750,000 invocations of Cloud Functions. The smallest time frame for cost reports is a full day and there is no capability to report costs for a specific function or function execution. Additionally, no costs are reported until the free tier of 2 million invocations, 400,000 GB-seconds memory time, and 200,000 GHz-seconds CPU time are used up.

Based on these limitations, we use the following approach to experimentally evaluate the costs of both workflows. First, we purposefully use up the capacity of the free tier by executing arbitrary functions. Next, we reserve a day for each experiment where no other functions are executed. During this day, we execute the first workflow 5,000 times with text segments with a normally distributed length ($\mu = 500$, $\sigma = 50$). At the start of the next day, we take the aggregated costs for the day and divide them by 5,000 in order to get the average cost per workflow execution. We repeat the same process for the second workflow.

Table 8.7 shows the measured costs per workflow execution, the predicted costs using our approach, and the resulting relative prediction error. At first glance, the measured prices seem unrealistically low. However, this is mostly due to the unfamiliar pricing scheme of cost per execution. If we were to assume that a n1-standard-2 VM (2 vCPU, 7.5GB memory) from Google Cloud (currently priced at $0.0950 per hour) can handle 5 requests per second, this would result in a cost of $5.3 \cdot 10^{-6}$ per request. Consequently, costs of $1.02 \cdot 10^{-4}$ and $4.60 \cdot 10^{-5}$ to execute a workflow consisting of five functions is cheap, but within reason.

Existing approaches that rely on cost estimations for serverless functions and workflow simply round the mean observed function response time up to the nearest 100ms [Boz+17; Elg18]. Applying this cost estimation approach results in a cost estimate of $4.06 \cdot 10^{-5}$ for both workflows as it assumes that the function execution cost is independent of its context. However, a single execution of WorkflowA costs $1.02 \cdot 10^{-4}$ ct, whereas an execution of WorkflowB costs only about half as much even

though both workflows provide the same functionality. This results in a relative prediction error of 60.2% for WorkflowA and 11.8% for WorkflowB using the naive cost estimate.

Our approach on the other hand accurately predicts this cost difference between the two workflows. It predicts the charged costs for invocations, CPU time, and memory time with a prediction error ranging from 2.6% to 10.3%. An interesting observation is that the measured and predicted cost for the number of invocations differs. As in the workflows from our case study, the number of function invocations is static, our approach correctly predicts that 25,000 functions (5 executions per workflow * 5,000 workflow executions) will be executed. The billed costs on Google Cloud are rounded up to full cents, which causes this observed difference between measured and predicted costs for function invocations.

Result 2: Our approach predicts costs for the execution of both workflows with an error of 4.0% and 3.6% respectively, resulting in an average cost prediction accuracy of 96.2% (**EQ IV.2**).

8.2.4 Overhead Analysis

The proposed approach enables accurate cost predictions for serverless workflows. In order to ensure that the proposed approach is applicable in practice, we investigate the time required to train the machine learning models for each function and the time required for the Monte-Carlo simulation.

The following experiments were conducted using an Intel® Core™ i5-4690K CPU with 3.50 GHz. We measure the time required to train the MDN model for the response time distribution and the MDN model for the output parameter for each function from our case study with the hyper-parameter optimization to determine the appropriate number of kernels, which requires training MDN models. Figure 8.6 shows the result of repeating this measurement ten times as a boxplot. The training process for each function which includes training ten mixture density networks takes between 10 and 15 minutes, with the *Censor* function as a small outlier with a median training time of 20 minutes. This difference can be attributed to an increased model complexity due to the additional input parameter. In general, we consider these training times acceptable as the training is performed offline and can be easily parallelized.

Additionally, we measure the time required to derive the cost predictions for a workflow using the Monte-Carlo simulation. Predicting the costs of WorkflowA requires 16.34±0.30 (N=10) seconds, whereas the predictions for WorkflowB require 14.20±0.03 (N=10) seconds. A user looking to compare these two workflow alternatives would need to wait about 30 seconds.

Result 3: Our approach requires less than 30 minutes for the model training and less than 30 seconds for the cost prediction (**EQ IV.3**).

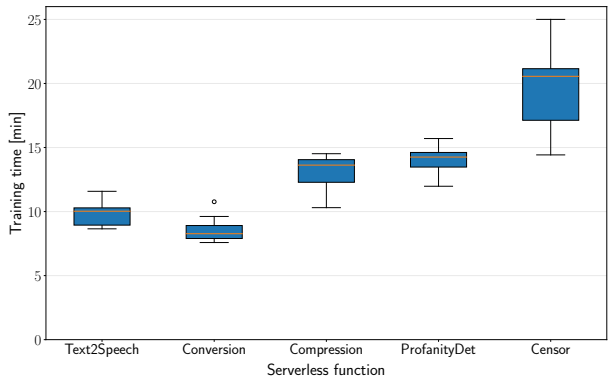


Figure 8.6: Training time for both models of each serverless functions with hyper-parameter optimization.

8.2.5 Replication Package

The replication package for this chapter consists of two parts: 1) reproducing the performance measurements for the audio processing functions in the Google Functions environment and 2) reproducing the performance predictions using the proposed approach based on the collected measurement data. In order to enable quick reproduction of the performance measurements, we packaged the scripts for the experiment automation as a docker container. The original measurement data used in this analysis and the python code implementing the proposed approach are available in form of a CodeOcean capsule, which enables quick and easy reproduction of our results. This replication package is available on Zenodo[3]. It was evaluated at the artifact evaluation track of the International Conference on Performance Engineering (ICPE) and has received the *ACM Artifact Evaluated - Available* and *ACM Artifact Evaluated - Functional* badges.

8.2.6 Summary

In this section, we evaluated whether the approach proposed in Section 5.2 fulfills **Goal IV** (*"Provide a technique to estimate the costs of serverless workflows."*). We answer **EQ IV.1** (*"Are mixture density networks capable of accurately predicting the distribution of the response time and the output parameters of a serverless function?"*) by showing that our approach predicted the response time and parameter distributions of serverless functions with an average prediction accuracy of 96.1%. The costs for the workflow execution are predicted by our approach with an average prediction accuracy of 96.2%, which addresses **EQ IV.2** (*"Can the proposed algorithm and the underlying machine learning models for the individual functions accurately predict the costs of a previously*

[3]https://doi.org/10.5281/zenodo.3612479

unobserved workflow?"). Finally, we answer **EQ IV.3** (*"What is the required time for training and workflow prediction? Is the overhead feasible for a production environment?"*) by showing that our approach requires less than 30 minutes for the model training and less than 30 seconds for the cost prediction, which we consider to be acceptable to use our approach in production. Answering these three evaluations questions shows that our approach achieves **Goal IV**. Using our approach, solution architects can make informed decisions when choosing between a serverless workflow and a traditionally hosted workflow and workflow designers can compare alternatives without time-intensive experimentation. Additionally, our approach represents a first step towards fully automated workflow optimization using multi-objective optimization techniques, analogously to existing tools for traditional software systems [Ale+09; Mar+10].

Chapter 9

Evaluating the Adaptation of White-Box Performance Models for Serverless Platforms

9.1 Simulation of Fine-grained Deployments

In this chapter, we present the evaluation of our approach for the simulation of fine-grained deployments. We design our case study in answer the following three evaluation questions to determine if we have successfully answered **RQ V.1** (*"How can the time required to simulate large systems such as serverless platforms be reduced?"*):

- **EQ V.1**: *Does replacing parts of the performance model with statistical response time models decrease prediction accuracy?*
- **EQ V.2**: *What are limiting factors for applying statistical response time models?*
- **EQ V.3**: *How much can the integration of statistical models decrease the time required to simulate a system?*

In the following, Section 9.1.1 introduces the component-based system we used to evaluate our approach. Next, Section 9.1.2 evaluates the accuracy of our response time and utilization predictions, and Section 9.1.3 evaluates the simulation time speedups. Finally, Section 9.1.4 summarizes our findings.

9.1.1 Evaluation System

We analyze a medium-sized, distributed system with seven components featuring diverse performance properties. We construct a traditional performance model for this system and extract statistical response time models as described in Section 6.1.2. Next, we compare the prediction accuracy of the traditional performance model to using different statistical response time models. Finally, we analyze how much applying different statistical response time models speeds up model solution.

For this case study, we deploy a system composed of components with synthetic resource demands on seven virtual machines with two 2.6 GHz cores and 4 GB

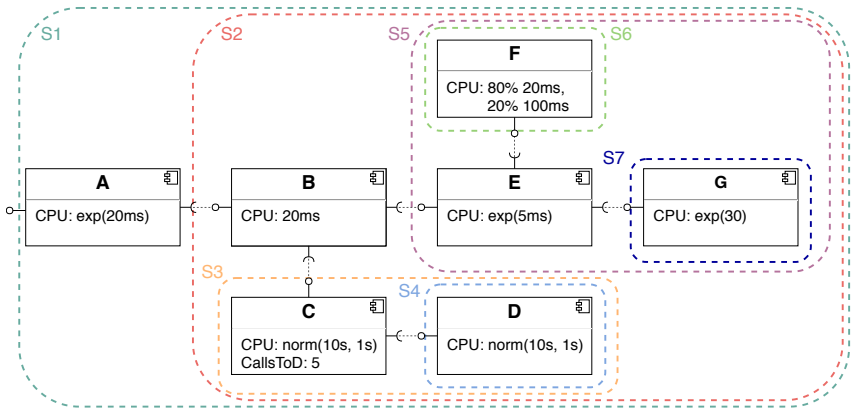


Figure 9.1: System consisting of components A-Z with their performance properties and the different models sections S1-S7 that can be replaced by statistical response time models.

memory, each. The virtual machines are deployed in a CloudStack cluster (version 4.9 with KVM) consisting of eight HPE ProLiant DL160 Gen9 hosts with eight 2.6 GHz cores and 32 GB RAM each. Hyper-threading was disabled on all hosts to avoid race conditions. The architecture of the system is shown in Figure 9.1. It consists of seven components with constant, exponentially distributed, and normally distributed resource demands. Additionally, component C contains a loop that calls component D five times and component F contains branches where its resource demand is 20 ms 80% of the time and 100 ms otherwise. For our experiments, we use the load driver from [KDK18].

First, we build a traditional performance model using DML, based on the resource demands shown in Figure 9.1. After some initial analysis, we added network delays of about 3 ms to each external call, which we model as resource demands on a resource with infinite servers, as the network delays appear to be static. In order to collect monitoring data to construct the statistical response time models, we put the system under varying loads, ranging from 10 requests per second to 50 requests per second for ten minutes. Next, we use the algorithm presented in Section 6.1.2 to generate a set of training data to train a MARS model on. For the MARS model we use the py-earth python package[1] with the following parameters:

- minspan_alpha = 5
- endspan_alpha = 5
- sample_weight = 1 + 1/(concurrency + 1)

[1] `https://contrib.scikit-learn.org/py-earth/content.html`

Load [Req/s]	Measured / Predicted Utilization [%] A	B	C	D	E	F	G
10	11.4 / 9.9	11.3 / 10.0	6.6 / 5.0	25.7 / 25.0	4.0 / 2.5	18.0 / 18.0	15.3 / 15.0
15	15.9 / 15.0	16.0 / 14.9	9.6 / 7.5	37.3 / 37.4	5.4 / 3.7	26.2 / 26.9	22.2 / 22.5
20	20.7 / 20.1	21.2 / 20.1	12.8 / 10.1	48.9 / 50.2	6.7 / 5.1	34.7 / 36.2	28.6 / 30.1
25	25.6 / 25.1	25.8 / 25.1	15.3 / 12.6	60.6 / 62.8	7.9 / 6.3	42.9 / 45.3	36.1 / 37.6
30	30.2 / 29.9	30.4 / 29.9	11.7 / 15.0	72.7 / 74.7	9.4 / 7.5	50.9 / 54.1	42.8 / 42.5
35	35.3 / 35.0	35.3 / 35.0	20.8 / 17.5	88.5 / 87.8	10.9 / 8.8	60.0 / 63.0	50.5 / 52.4
40	38.5 / 40.0	39.1 / 40.0	24.2 / 20.0	96.7 / 99.9	12.2 / 10.0	67.9 / 72.0	55.3 / 59.9

Table 9.1: Comparison of measured utilizations of components A-G and the predictions by the queueing theory model.

Load [Req/s]	Measured / Predicted Responsetime [ms] A	B	C	D	E	F	G
10	226 / 209	200 / 185	93 / 81	16 / 14	83 / 82	42 / 40	34 / 34
15	229 / 215	204 / 192	95 / 85	16 / 14	85 / 84	43 / 41	35 / 35
20	236 / 226	211 / 202	99 / 92	17 / 16	87 / 87	45 / 43	35 / 36
25	254 / 245	228 / 221	112 / 106	20 / 19	92 / 92	47 / 47	38 / 37
30	283 / 281	257 / 256	136 / 132	25 / 24	96 / 99	49 / 51	40 / 40
35	339 / 391	312 / 365	184 / 232	34 / 44	102 / 109	54 / 58	42 / 43
40	4805 / 21614	4778 / 21587	4632 / 21438	924 / 4287	120 / 124	68 / 68	45 / 47

Table 9.2: Comparison of measured response times of components A-G and the predictions by the queueing theory model.

These parameters are derived based on systematic experimentation to optimize the internal GCV error score of the MARS model, which describes how well the model fits the training data. The sample weights prioritize lower concurrency levels in order to decrease the influence of measurements during overload scenarios. Using this approach, we extract a total of seven statistical response time models, shown in Figure 9.1 as S1-S7. These response time models have varying sizes. For example, S1 replaces the full system with a statistical response time model whereas S4 only replaces a single component. We integrate these response time models into the DML model as described in Section 5.1.3.1.

9.1.2 Prediction Accuracy

In a first step, we validate the prediction accuracy of the traditional performance model to ensure its validity. We ran seven experiments with a constant load of 10-40 requests per second for three minutes and measured the utilization and response time for every component. Table 9.1 compares the measured utilization to the predictions of the traditional queueing theory model. Most predictions are accurate with an error of less than two percentage points. For components C, D, F, and G

Statistical Models	Predicted Utilization [%]						
	A	B	C	D	E	F	G
None	25.1	25.1	12.6	62.8	6.3	45.3	37.6
S1	0	0	0	0	0	0	0
S2	25.0	0	0	0	0	0	0
S3	25.0	25.0	0	0	6.3	45.1	37.5
S4	25.0	25.0	12.5	0	6.2	44.9	37.5
S5	25.0	25.0	12.5	62.5	0	0	0
S6	25.0	25.0	12.5	62.5	6.3	0	37.5
S7	25.0	25.0	12.5	62.6	6.3	45.1	0
S3 + S5	25.0	25.0	0	0	0	0	0
S3 + S6	25.0	25.0	0	0	6.3	0	37.5
S3 + S7	24.9	25.0	0	0	6.2	45.0	0
S4 + S5	25.0	25.0	12.5	0	0	0	0
Measured	25.6	25.8	15.3	60.6	7.9	42.9	36.1

Table 9.3: Predicted utilization at 25 req/s when applying different statistical models (see Figure 9.1) and the measured values as baseline.

the prediction is slightly off in the high load scenarios, but still always less than five percentage points. Table 9.2 shows the measured response times along the predicted values. Here, the prediction error is $\leq$10% for the experiments with 10-30 requests per second. For 35 requests per second, the model overestimates the system response time but remains below the 30% considered sufficient for capacity planning [MV00]. At 40 requests per second, the response time predictions are inaccurate, which is expected as the system is under excess load, so there is no steady-state for the response time. Overall, the accuracy of the model seems sufficient and will be our comparison values for the following experiments.

Next, we apply the different statistical response time models S1-S7 as well as some combinations of them (S3 + S5, S3 + S6, S3 + S7, S4 + S5) and compare the resulting prediction accuracy to the traditional performance model without statistical response time models ("None"). Table 9.3 shows the resulting utilization predictions. We performed this analysis for all load levels, but show only the results for 25 requests per second due to space constraints. The full dataset is available online[2]. The first observation here is that the model predicts a utilization of zero for any component which is replaced by a statistical model. This can be explained by the fact, that the statistical models only predict a response time, but do not schedule any resource demands. Therefore, the first limitation for the integration of statistical models in performance models is that we can not replace components or subsystems that are deployed on physical resources of which we want to predict the utilization.

[2] `https://github.com/SimonEismann/ICSA2019/blob/master/results.xlsx`

Statistical Models	Predicted Responsetime [ms]						
	A	B	C	D	E	F	G
None	245	221	106	19	92	47	37
S1	266	-	-	-	-	-	-
S2	263	239	-	-	-	-	-
S3	265	240	131	-	86	42	36
S4	284	260	145	26	91	46	37
S5	253	228	105	18	99	-	-
S6	252	227	105	18	98	55	35
S7	252	227	106	19	98	46	44
S3 + S5	270	245	130	-	95	-	-
S3 + S6	269	244	131	-	90	48	34
S3 + S7	270	245	130	-	91	42	41
S4 + S5	295	267	145	26	99	-	-
Measured	254	228	112	20	92	47	38

Table 9.4: Predicted response time at 25 req/s when applying different statistical models (see Figure 9.1) and the measured values as baseline.

The average prediction accuracy across all utilization predictions without statistical models is 9.2%±9.4, compared to 9.1%±9.4 for the average prediction accuracy across all utilization predictions using models containing statistical response time models. Therefore, we conclude that the utilization predictions remain accurate. Table 9.4 shows the same data for the response time predictions. Here, when we replace a component with a statistical model, we lose the ability to predict the response time of any components called by this component as the statistical response time model does not contain any notion of external calls. This is a conceptual limitation for the use of statistical response time models within performance models. The response time predictions are in some cases more accurate than their queueing model counterparts and worse in other cases. The average prediction accuracy across all response time predictions without statistical models is 6.3%±6.4, compared to 12.6%±20.6 for the average prediction accuracy across all response time predictions using models containing statistical response time models. While the prediction accuracy deteriorates slightly, it remains overall satisfactory.

It is interesting to note here, that the prediction accuracy is not tied to the size of the system that is replaced by a statistical response time model. For example, the S4 + S5 scenario replaces four components and achieves the worst prediction accuracy, whereas S1 replaces seven components but results in better prediction accuracy. Instead, the prediction accuracy seems to be tied to how well the MARS model is able to fit the response time data, that is the better the MARS model predicts the monitoring data, the more accurate the overall predictions become. Here, the investigation of other machine learning approaches such as random forests would

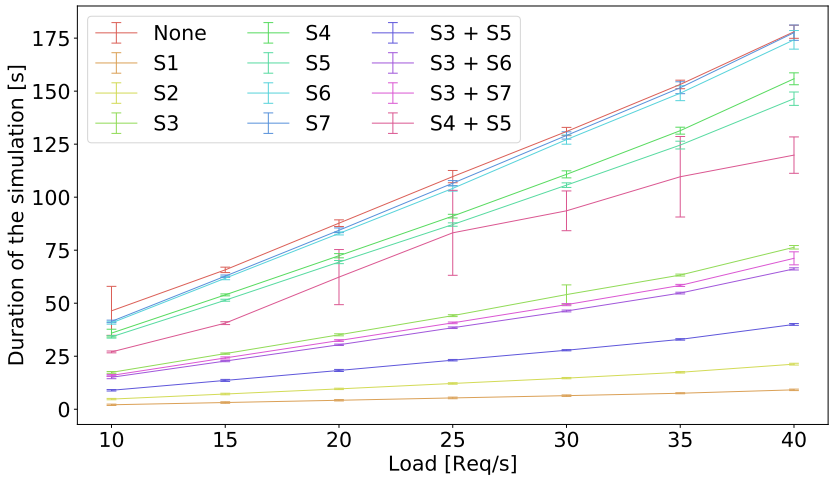


Figure 9.2: Wall clock time required to simulate 200.000 seconds of simulated time when applying different statistical models (see Figure 9.1).

be of interest.

Result 1: Integrating statistical response time models in architectural performance models does not invalidate the performance predictions (**EQ V.1**).

Result 2: The prediction of some performance metrics is no longer possible when integrating statistical response time models in architectural performance models (**EQ V.2**).

9.1.3 Simulation Time Analysis

The previous experiment shows that replacing parts of the performance model makes the prediction of some performance indices impossible, but does not negatively impact the accuracy of the remaining predictions. In the next step, we analyze the speedup that can be gained in these scenarios. In order to measure the simulation time, we measure the wall clock time required to simulate a total of 200 000 seconds for each scenario from the previous experiment. The measurements were performed on an out-of-the-box Lenovo X1 Yoga Gen 2 Thinkpad with an i7-7600U CPU with up to 2.80 GHz. As there is some variation in the results, we repeat the measurements 20 times each and calculate the standard deviation.

Figure 9.2 shows the required wall clock time to simulate 200 000 seconds for load levels from 10 to 40 requests per second. The required time increases linearly with the number of requests per second. This behavior is expected, as doubling the number of requests per second roughly doubles the number of events that need to be processed within the event-based simulation. The model without any statistical response time

models is the slowest and takes up to 178 seconds. The fastest is S1 with an average simulation time of 9.1 seconds for 40 requests per second, resulting in a speedup of 94.8%. On the other hand, S6 and S7 result in almost no speed up, as they only replace a single component with a statistical response time model. Curiously, S4 also replaces only a single component but still speeds the simulation up by ∼60%. The difference here seems to be that component D is called five times as often as components E/F. So we can conclude that the speedup depends on the number of calls to the replaced components. Based on S1-S7 we can observe that the achieved speedup is also related to the size of the system which is replaced by a statistical response time model. Lastly, the results for models containing multiple statistical response time models (S3+S5, S3+S6, S3+S7, and S4+S5) indicate that applying multiple models increases the speedup compared to their singular counterparts (S3, S4, S5, S6, S7).

Result 3: Integrating statistical response time models in architectural performance models speeds up simulation time by up to 94.8% (**EQ V.3**).

9.1.4 Summary

In this section, we evaluated whether the approach proposed in Section 6.1 answers **RQ V.1** (*"How can the time required to simulate large systems such as serverless platforms be reduced?"*) using a medium-size component-based application. We answer **EQ V.1** (*"Does replacing parts of the performance model with statistical response time models decrease prediction accuracy?"*) by showing that the integration of statistical response time models in white-box performance models does not invalidate performance predictions. However, our evaluation revealed that the prediction of some performance metrics is no longer possible, which addresses **EQ V.2** (*"What are limiting factors for applying statistical response time models?"*). Finally, we answer **EQ V.3** (*"How much can the integration of statistical models decrease the time required to simulate a system?"*) by showing that tailoring the integrated statistical response time models to the required metrics can speed up the simulation time of architectural performance models by up to 94.8%. Answering these three evaluation questions shows that our approach answers **RQ V.1**, by enabling faster solution of architectural performance models while retaining the prediction accuracy and capability to predict previously unseen scenarios of traditional, queueing theory-based performance models. Our approach enables software architects to analyze larger systems, performance engineers benefit from the ability to build more detailed models and self-adaptive systems can explore additional adaptation options within the same time period due to the faster model solution. Further, it represents a step towards solving a problem that impedes widespread adoption of performance models in practice: models that are either too large or too detailed cannot be simulated within a reasonable time frame [Nam+16; WFP07; Koz10].

9.2 Modeling of Parametric Dependencies

In this section, we present the evaluation of our approach for the modeling of parametric dependencies. We design our evaluation to answer the following three evaluation questions to determine if we have successfully answered **RQ V.2** (*"How can relationships between parameters observed at runtime be utilized in white-box performance models?"*):

- **EQ VI.1**: *Is the modeling of dependencies on the component instance required in some cases?*
- **EQ VI.2**: *Can we provide multiple characterizations of a parametric resource demand based on alternative parameters that enable accurate performance predictions?*
- **EQ VI.3**: *Does modeling parametric dependencies on component instance-level improve the prediction accuracy?*

In the following, Section 9.2.1 introduces the case study setup we used to evaluate our approach. Next, Section 9.2.2 describes how we used the previously introduced concepts to model the dependencies within our case study, and Section 9.2.3 evaluates how much our approach improves the performance prediction accuracy. Finally, Section 9.2.4 summarizes our findings.

9.2.1 Case Study Setting

For our evaluation, we implemented parts of the motivating video store required for analysis. All of our experiments run on an HPE ProLiant DL160 server. It features an Intel Xeon E5-2650 v3 processor with 10 cores at 2.4 GHz and 32 GB RAM. To provide an implementation of the video transcription service for the first case study, we integrate the established open source speech recognition software CMUsphinx[3] into a Java EE servlet application. The processing time of the transcription service depends on the properties of the respective video. The length of a video, as well as the number of subtitles (counted as number of lines), impact the processing overhead. While the number of subtitle lines can be monitored from the Java interface, the monitoring of the file size requires a more intrusive monitoring. For the evaluation, we collected a training data set and an evaluation data set, each consisting of 25 videos with 10 to 30 seconds of English-spoken content from YouTube. Based on these videos, we investigate the response times of the transcription service for a mixture of videos.

The second case study monitors the impact of different parametrizations on the system performance for subtitle retrieval. At the retrieval, subtitles have to be queried from a database if not cached within the subtitle repository component. Subtitles in frequent languages and for popular videos are more likely to be in the

[3] https://cmusphinx.github.io/

Region	ENG	SPA	GER	FR	IT	POL	RUM
USA	83%	17%	0%	0%	0%	0%	0%
EU	25%	11%	21%	15%	14%	9%	5%

Table 9.5: Assumed language mix within the workload.

Popularity Class	Number of Products	Product Likelihood
Frequent access	200	0.3%
Moderate access	800	0.033%
Long tail	9000	0.0011%

Table 9.6: Product popularity classes.

cache. Consequently, the likelihood of a subtitle being in the cache is influenced by the requested language as well as the popularity of the video. We assume two workload mixes depicted in Table 9.5, denoted as USA and EU. We assume American traffic to be predominated by English and Hispanic customers, whereas European traffic consists of a wide variety of languages. The popularity of videos follows a long tail distribution [BHS06]. As shown in Table 9.6, the store provides 10.000 videos from which the 200 most frequently accessed videos make up for over half of the traffic. The next 800 most frequently accessed videos cause 30% of the traffic. The remaining 9000 long-tail videos cause only 10% of the traffic. For each video, subtitles can be requested in seven languages. While the cache allows storing 250 subtitles at a time, the number of subtitles sums up to a total of 70.000.

9.2.2 Dependency Modeling

Figure 9.3 models the video transcription use case. The CPU resource demand of the `TranscriptionService` is modeled as a `ResourceDemand` variable. Two `ServiceInputParameters` specify the file size and the number of lines parameters. Further, we model a `DependencyRelationship` describing the relationship between the file size and the resource demand on the component level. To describe the relationship between the number of lines and the resource demand of the `TranscriptionService`, we select a `CorrelationRelationship` to model the instance-level dependency.

Figure 9.4 shows modeling of the subtitle provider use case. The behavior of cache hits and misses is modeled as a `BranchAction` with an unknown `BranchingProbability`. The branch denoting a cache-hit is connected to a lower resource demand than the branch for cache misses. The input parameters `Language` and `Popularity` are modeled as `ServiceInputParameters`. The dependency between these parameters and the branching probability is represented by a `CorrelationRelationship` with two independents to model an instance-level dependency.

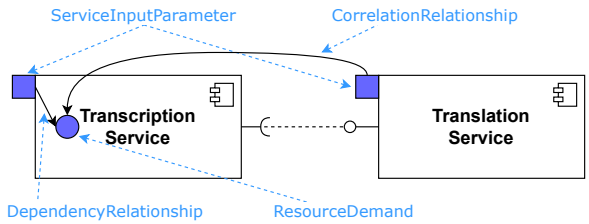


Figure 9.3: Modeling the video transcription case study.

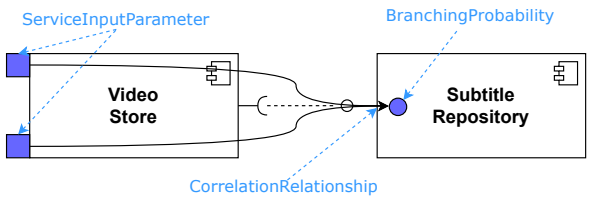


Figure 9.4: Modeling the subtitle generation case study.

To characterize the parametric dependencies, we measure the video file size, the amount of generated subtitles in number of lines, and corresponding resource demands for the training data set. Then, we fit polynomial functions up to fourth-order that describe the resource demand from the file size and the number of lines. Our investigations show that increasing polynomial degrees does not significantly improve the prediction. Therefore, we conclude a linear correlation and model both aspects using first-order functions. The CPU resource demand description of the transcription service based on the file size in kilobyte is:

$$ResourceDemand = 43.2 \times FileSize - 1662.5$$

Additionally, we model the resource demand based on the number of lines parameter as:

$$ResourceDemand = 2253.5 \times NumberOfLines + 3894.2$$

In this use case, modeling the correlation between the number of lines parameter and the transcription service resource demand allows to accurately estimate the resource demand. This provides an additional description of the resource demand, which can be used as a fallback in case the size of the video files cannot be monitored.

The dependency between the language distribution, the popularity distribution, and the probability of accessing the database to retrieve a subtitle follows Wallenius' noncentral hypergeometric distribution [Fog08], which can be approximated as a

binomial distribution. This approximation leads to the following formula describing the probability for a cache hit:

$$\mathrm{P}(hit) = \sum_{l}^{Languages} \sum_{p}^{Popularities} \frac{\mathrm{P}(p)^2 \times \mathrm{P}(l)^2 \times 250}{|p|} \tag{9.1}$$

For all popularity levels p and all languages l, their respective occurrence probabilities $\mathrm{P}(p)$ and $\mathrm{P}(l)$ are squared and multiplied by the cache capacity, which is 250. Finally, the resulting value is divided by the number of subtitles in the language l and popularity class p, which is equivalent to the number of videos in the popularity class p, denoted as $|p|$.

Instance-level dependencies are required to capture the influence of the video popularity and subtitle language influence on the cache probability of the subtitle provider. We argue that this dependency has to be modeled on component instance-level, that is for this specific component instance. In contrast, the traditional approach would model it as a dependency on repository component-level, that is a dependency that applies to all subtitle provider instances. However, this dependency can not be applied to every subtitle provider instance. For example, if a subtitle provider instance is used by a backup component, the dependency would not apply. The backup component iterates over all subtitles leading to a cache probability of zero, since no subtitle is requested twice. However, the dependency from Equation 9.1 would derive a non-zero cache probability.

Result 1: It is necessary to model some dependencies on the component instance-level rather than on the repository component-level (**EQ VI.1**).

9.2.3 Prediction Accuracy

After dependency resolution, we solve the resulting `StackFrame` model reusing a transformation to QPN and subsequent simulation using SimQPN, a discrete event simulation tool for QPNs [KB06]. Table 9.7 shows the residual standard error, R-squared metric, and p-value for estimations of four polynomial orders with either the file size or the number of subtitle lines as input. The low p-value indicates a significant relationship between the input parameter and the response time, which justifies modeling it. The residual standard error of the number of lines estimation exceeds the error of file size estimation by about 10%. Similarly, the file size estimation captures the variance within the sample slightly better, as described by the R-squared metric. While the file size estimation produces more accurate predictions, the estimation using the number of lines is sufficient in case the file size cannot be monitored. Our experiments show that the correlation between the resource demand and the number of lines parameter provides an accurate description of the resource demand.

Input	Polynomial order	p-value	Residual std. error	R-Squared
File size	1	8.5e-13	3023	0.896
	2	1.4e-11	3078	0.897
	3	6.3e-11	3017	0.906
	4	9.6e-11	2826	0.921
Number of lines	1	1.8e-11	3445	0.865
	2	1.7e-11	3363	0.877
	3	5.2e-10	3338	0.884
	4	2.9e-09	3357	0.889

Table 9.7: Evaluation of the CPU resource demand estimations.

Region	Metric	Eval.	Load-Level high	med	low	lowest
EU	Utilization	measured	79.6	39.2	19.7	9.4
USA	Utilization	measured	57.2	29.9	15.4	8.0
Both	Utilization	predicted	70.7	35.3	17.6	8.8
EU	Relative Prediction Error		11.2	9.9	10.7	5.9
USA			23.6	18.1	14.3	10.6

Table 9.8: Comparison of the measured utilization to a model-based prediction without parametric dependencies.

Result 2: Multiple accurate descriptions of the same variable can be derived based on different input (**EQ VI.2**).

To demonstrate that each of the two descriptions allows for accurate performance predictions, we analyze the accuracy of response time predictions for each description. We configure an exponentially distributed request inter-arrival rate with an average delay of 60 s which corresponds to a CPU utilization of around 45%. The requests randomly select a video from the evaluation data set. The measured average response time was 44207 ms. The model predicts an average response time of 45518 ms when using the file size estimation and 46252 ms for the number of lines estimation. This corresponds to an accuracy of 97.03% and 95.38%, respectively. While using the file size estimation achieves a better accuracy, the accuracy of both predictions is sufficient for capacity planning [MV00].

Modeling this case study without parametric dependencies corresponds to using the average monitored value for the branching probability. Analysis results presented in Table 9.8 show this to be highly inappropriate to capture the system behavior due

Region	Metric	Eval.	Load-Level			
			lowest	low	med	high
EU	Utilization	measured	9.4	19.7	39.2	79.6
		predicted	9.8	19.8	39.0	79.3
	Resp. time	measured	21.0	27.0	39.0	149.0
		predicted	23.1	26.7	37.0	106.0
USA	Utilization	measured	8.0	15.4	29.9	57.2
		predicted	7.2	14.4	28.8	57.4
	Resp. time	measured	17.0	18.0	22.0	45.0
		predicted	16.2	18.3	23.6	43.9

Table 9.9: Comparison of measured performance metrics to predictions using parametric dependencies.

to an average utilization prediction error of 13%. When modeling the dependency between the language and popularity distribution and the branching probability of the subtitle repository, the DML model is capable of accurately predicting the utilization for all eight scenarios, with a relative error below 5%, as shown in Table 9.9. The response time prediction error is below 10%, except for a high load of EU traffic. Here, the model-based prediction underestimates the response time by 28.86%. This outlier is still within a 30% margin considered to be acceptable for capacity planning [MV00]. By modeling the cache probability as a parametric dependency, our model can accurately predict the impact of different language distributions on the performance of the system.

Result 3: Using parametric dependencies reduced the average relative utilization prediction error from 13% to below 5% (**EQ VI.3**).

9.2.4 Summary

In this section, we evaluated whether the approach proposed in Section 6.2 answers **RQ V.2** (*"How can relationships between parameters observed at runtime be utilized in white-box performance models?"*) in two case study scenarios. We answer **EQ VI.1** (*"Is the modeling of dependencies on the component instance required in some cases?"*) by showing that it is necessary to model some dependencies on the component-level rather than on the repository component-level. Next, we show that multiple accurate descriptions of the same variable can be derived based on different input, which addresses **EQ VI.2** (*"Can we provide multiple characterizations of a parametric resource demand based on alternative parameters that enable accurate performance predictions?"*). Finally, we cover **EQ VI.3** (*"Does modeling parametric dependencies on*

component instance-level improve the prediction accuracy?") by showing that in our case studies the use of parametric dependencies reduced the average relative utilization prediction error from 13% to below 5%. Answering these three evaluations questions shows that our approach achieves **RQ V.2**. Applying the presented approach, software architects benefit from an improved ability to reflect system behavior within architectural performance models based on a higher flexibility of inputs. This resulting increased input sensitivity increases the prediction accuracy for the analysis of design tradeoffs and other what-if analysis scenarios.

Part IV

Conclusion

Chapter 10

Summary

Serverless computing abolishes the notion of virtual machines and instead allows developers to execute code on-demand in response to events with continuous scaling while having to pay only for the time used with sub-second metering. In recent years, cloud providers have further introduced many managed services for databases, messaging buses, and storage, that also abstract the notion of virtual machines in favor of a pay-per-use model. The adoption of serverless computing is quickly growing due to the reduced operational overhead, perceived cost savings, and increased development speed.

However, the serverless community faces significant new challenges due to the paradigm shift and is in disagreement regarding many serverless computing topics. This is especially the case for the performance of serverless applications and platforms as much existing performance engineering knowledge and many existing approaches are not directly transferable to serverless computing. This results in many poorly designed serverless applications in practice which in turn can quickly undermine the public perception of serverless computing. Further, there are also many open questions regarding the design of serverless platforms. The lack of an in-depth understanding of the performance implications and tradeoffs of these design decisions can severely hinder the advancement of serverless platforms.

In this thesis, we address the lack of performance knowledge surrounding serverless applications and platforms from multiple angles: we conduct empirical studies to further the understanding of serverless applications and platforms, we introduce automated optimization methods that reduce the knowledge overhead of operating serverless applications, and we enable the analysis of design tradeoffs by extending white-box performance modeling. In the following, we summarize the individual contributions and revisit the goals and research questions laid out in Section 1.3.

Contribution 1: Evaluation of the Characteristics and Performance of Serverless Applications As our first contribution, Chapter 4 presents two empirical studies that aim to improve the understanding of serverless applications. The first study addresses **Goal I** (*"Provide quantitative data on the common characteristics of modern serverless applications."*) by conducting the first systematic and comprehensive characterization of serverless applications. We collect 89 serverless applications and

characterize them, which answers **RQ I.1** (*"What are common characteristics of current serverless applications?"*). To answer **RQ I.2** (*"Is there a community consensus on the common characteristics of serverless applications?"*), we further compare the results of our study with ten existing surveys and datasets. We find that the most commonly reported reasons for the adoption of serverless computing are cost savings for irregular or bursty workloads, avoidance of operational concerns, built-in scalability, and increased speed of development. Typical use cases for serverless applications include short-running tasks with low data volume and bursty workloads, but we also frequently found latency-critical, high-volume core functionality as serverless applications. Further, we find that serverless applications are mostly implemented on AWS, in either Python or JavaScript, and use managed services. With more details beyond this summary presented in Chapter 4 , we are confident about having achieved **Goal I**.

The second study investigates the stability of performance measurements of serverless applications which aligns with **Goal II** (*"Quantify the performance variability that serverless applications experience."*). We first quantify the baseline performance variability of serverless applications based on multiple repetitions of performance measurements under varying configurations, which addresses **RQ II.1** (*"How much performance variability do common serverless applications experience?"*). To answer **RQ II.2** (*"Does the performance of serverless applications change over time?"*), we conduct a longitudinal study consisting of three daily measurements for ten months. We find that the performance variability of measurements conducted at the same time is comparable to the performance variability observed in traditional microservice or monolithic systems. However, we also find that serverless applications experience short-term and long-term performance changes. These findings show that we have achieved **Goal II**.

Contribution 2: Automating Operational Tasks of Serverless Applications Our second contribution introduces two approaches to automate the operational tasks associated with serverless applications. The first approach automates resource sizing by predicting the optimal memory size of serverless functions, which addresses **Goal III** (*"Develop an automated method to optimize the size of serverless functions."*). First, we measure the execution time and resource consumption metrics of a large collection of functions generated by our synthetic function generator on a public cloud, which targets **RQ III.1** (*"How can a dataset on the impact of memory size for a vast number of functions be generated?"*). Then, to answer **RQ III.2** (*"How can one predict the optimal size of serverless functions based on passive monitoring data?"*), we construct a multi-target regression model to predict the execution time of a serverless function for previously unseen memory sizes based on the execution time and resource consumption metrics for a single memory size. In our evaluation on four realistic serverless applications, our approach selects the optimal memory size for 79.0%

of the serverless functions, which saves on average 2.6% costs and speeds up the functions by 39.7%. These results show that our approach achieves **Goal III**.

The second operational task that we automate in this contribution is the cost prediction of serverless workflows, which addresses **Goal IV** (*"Provide a technique to estimate the costs of serverless workflows."*). First, we show how Mixture Density Networks (MDNs) can be used to accurately predict the response time and output parameter distributions of serverless functions, which answers **RQ IV.1** (*"How can the execution time distribution of a serverless function be predicted based on its input parameters?"*). These individual function models are integrated into a workflow model that a Monte-Carlo simulation traverses to derive cost predictions for serverless workflows, which addresses **RQ IV.2** (*"Can the impact of restructuring a serverless workflow on its cost be predicted?"*). In our evaluation in the context of audio transcription workflows, our cost prediction approach predicts the response time distribution and the distribution of the output parameters of five representative Google Cloud Functions with a mean accuracy of 96.1%. For two workflows composed of these functions, our approach achieves a mean workflow cost prediction accuracy of 96.2%, which shows that our approach achieves **Goal IV**.

Contribution 3: Enabling White-Box Performance Modeling and Simulation of Serverless Platforms Our third contribution introduces two approaches aimed towards **Goal V** (*"Provide an approach for the white-box performance modeling and simulation of serverless platforms."*). The first approach speeds up the simulation time required to solve white-box performance models, which targets **RQ V.1** (*"How can the time required to simulate large systems such as serverless platforms be reduced?"*). Our approach enables a parallel description of subsystems of a white-box performance model as fast-to-solve black-box performance models. We provide a transformation of the integrated queueing/statistical model to QPNs and extend an existing discrete event simulation solver for QPNs to support black-box performance models. Our evaluation shows that our approach maintains prediction accuracy and achieves speedups of up to 94.8% for a component-based system of medium size, which shows that our approach successfully achieves **RQ V.1**.

The second approach provides an integration of empirically observed relationships between model parameters in white-box performance models, which targets **RQ V.2** (*"How can relationships between parameters observed at runtime be utilized in white-box performance models?"*). We propose a novel approach to modeling empirical parametric dependencies in architectural performance models. To derive performance prediction, a dependency resolution algorithm transforms the empirical information from the model into a directed graph and resolves this graph to derive a fully parameterized model. In two case studies in the context of a media store, our approach for the modeling and solution of empirical parametric dependencies achieves a mean prediction error for utilization and response time of less than 5% and 10% respectively, which shows that our approach successfully answers **RQ V.2**.

Together, these two approaches achieve **Goal V** (*"Provide an approach for the white-box performance modeling and simulation of serverless platforms."*).

Summary In this thesis, we present three core contributions regarding the performance engineering of serverless applications and platforms. Together, these contributions address all five challenges introduced in Section 1.1. We address the lack of performance-related knowledge from three directions: (i) we investigate the characteristics and performance of serverless applications, (ii) provide automation for the operational tasks associated with serverless applications, and (iii) adapt white-box performance models to make them applicable to analyze the design tradeoffs of serverless platforms. As a result, these contributions further the understanding of the performance of serverless applications and represent a significant advance in the performance engineering of serverless applications and platforms.

Chapter 11

Open Challenges and Future Work

Finally, we discuss open challenges in the performance engineering of serverless applications and platforms that we did not address in this thesis and motivate future work in this research area.

Empirical analysis of serverless platforms and developers In this thesis, we conducted two empirical studies to further the understanding of serverless applications. These studies are by no means exhaustive, there are multiple further directions for future work in this area. While there have been a multitude of studies that characterize the performance of serverless platforms [Llo+18; Wan+18a; LSF18; Yu+20; Fig+18], most of these studies treat the serverless platforms as a black-box and are limited to describing the performance properties they observe. Here, a tracing-based, white-box benchmark of serverless platforms could elucidate the reasons behind the observed performance [Eyk+20]. For example for tail-latencies, it would be insightful to determine which part of the request, such as computation, network latency, external calls, or orchestration, is causing the slowdown. There is also a distinct lack of studies that focus on the development workflow and lifecycle of serverless applications, which is a perspective that is currently missing for a holistic view of serverless applications. Studies that investigate developer pain points, gaps in current workflows, and DevOps practices, similar to the survey by Leitner et. al. [Lei+19], and compares these practices to reports by developers of traditional systems, could help pinpoint the novel aspects and challenges related to the development of serverless applications.

Resource abstraction layer We have proposed an approach to automatically optimize the resource sizing of serverless functions, which automates the most cumbersome configuration parameter of serverless functions. However, serverless applications do not consist solely of serverless functions, instead, they make heavy use of managed services as described in our survey on the characteristics of serverless applications. These managed services come with a multitude of configuration parameters, for example, for a managed queue on AWS, developers need to configure among other things the type of queue, encryption, deduplication window and scope, number of retries, maximum message size, retention period, visibility

timeout, delivery delay, and receive wait time. While each individual configuration is not as cumbersome to determine as the resource size of serverless functions, the sheer number of configuration parameters places a high burden of knowledge on developers [Lei+19; LP20]. Here, an additional layer of abstraction on top of these resources that allows developers to specify their use case and derives good defaults from the selected use case and context information could help to significantly reduce the number of parameters that developers need to know about and configure.

Holistic cost modeling One of the key selling points of serverless computing is the fine-granular pay-per-use billing model, instead of the traditional pay-per-reservation billing model [Cas+19]. However, this also makes it challenging to estimate the expected costs of an application during the design of the application. In the context of this thesis, we introduced an approach to estimate the costs of serverless workflows composed of serverless functions. Other cost estimation approaches also focus elusively on serverless functions [Boz+17; Elg18; Gun+19]. Therefore, none of these approaches enable the holistic cost estimation of serverless applications. Here, a model-based approach, that enables developers to model the cost-relevant aspects of an application could fill this gap. Based on this model, cost estimations could be derived analytically based on the cloud provider cost models. The existing approaches that address specific cost prediction complexities could be integrated in this holistic model to enable the cost estimation of serverless applications during the design phase.

Simulating the transient behavior of serverless platforms In the context of this thesis, we have extended white-box performance models to address shortcomings that previously made them inapplicable for the analysis of serverless platforms. This means that white-box performance models can now be used to analyze the steady-state behavior of serverless platforms. However, serverless functions are often short-lived and ephemeral, so their transient behavior is quite important to the operation of a serverless platform. This includes for example the analysis of provisioning and de-provisioning techniques [Sha+20], request routing strategies [ABV18; LC21], or resource placement approaches [Mah+19; BKN21]. Existing simulations for serverless platforms do not support such fine-grained analysis [MK21]. Building on top of our approaches to alleviate the shortcomings of white-box performance models, white-box performance models could be further extended to support the simulation of transient behavior. Towards this goal, architectural performance models would need to be extended to enable the modeling of serverless function-specific features, such as cold starts, and provide interfaces for run-time strategies, such as resource provisioning, request routing, or resource placement. Further, this would require a simulation environment targeted towards the analysis of transient phases, instead of the commonly targeted steady-state phases. The resulting framework would enable

the time- and cost-efficient analysis of design decisions for the run-time behavior of serverless platforms.

Understanding the evolution of serverless computing Serverless computing is still an emerging technology, with consistent innovation from providers and developers. This is apparent from the large number of novel platform improvements that are proposed [Sin+19; Oak+18; Cad+20] and the constant stream of novel application domains that are adopting serverless computing [Niu+19a; Fou+19; AC17a]. The fast innovation cycle means that study results can become outdated in a matter of years, as evidenced by studies reporting conflicting results for the characteristics of serverless applications [Wit+20; Eiv17] and measurement studies reporting differing performance properties [Llo+18; Wan+18a; LSF18]. We believe that this necessitates a methodological change, where studies are not conducted to capture a snapshot of the analyzed properties, but rather are regularly repeated to capture the evolution of serverless applications and platforms over time. A challenge in this direction is that many studies can not be externally reproduced due to missing documentation [SL20]. To address this, we provide detailed replication packages for the empirical studies conducted in the context of this thesis.

List of Figures

List of Tables

List of Algorithms

Acknowledgements

This thesis would mot have been possible without the aid and support of many people. First of all, I would like to thank my advisor Prof. Samuel Kounev. I first met him during my time as a research assistant at the 6th Descartes anniversary party, where his talk on logical fallacies made a lasting impression on me. Since then, he has been a constant source of inspiration and motivation for me. His creative ideas, critical feedback and analytical thinking helped me develop as a researcher. Further, he has been supportive and trusted me when I wanted to explore a new direction for my thesis.

I may have never pursued a PhD without Dr. Jürgen Walter who took me under his wing during my bachelor thesis, subsequent work as a research assistant, master thesis and first year as a PhD student. I learned a lot during this time about academia, research and paper writing from his mentorship. After his graduation, Dr Joakim von Kistowski and later Dr. Nikolas Herbst took over as my mentors and guided me in my research journey.

I would like to thank my colleagues at the University of Würzburg who were always ready to discuss ideas, problems, share thoughts, and provide feedback. In no particular order (hoping to not forget anyone): Lukas Iffländer, Christian Krupitzer, Simon Spinner, Piotr Rygielski, Aleksandar Milenkoski, Fabian Brosig, Vanessa Borst, Bohdan Dovhan, Marius Hadry, Maximilian Meißner, Stefan Herrnleben, Dennis Kaiser, Robert Leppich, Veronika Lesch, Thomas Prantl, Florian Spiess, and Marwin Züfle. In particular I would like to thank in Johannes Grohmann, André Bauer, Norbert Schmitt, Andre Gräubel, Christoph Hagen, Lukas Beierlieb, and Martin Sträßer for many fun discussions over lunch, which kept me in good spirits whenever experiments did not want to work.

I also want to thank Fritz Kleemann, Susanne Stenglin, and Erika Littmann, who patiently supported me with my silly technical and administrative issues.

My special thanks to my research assistants Simon Trapp and Long Bui whose support enabled me to pursue as many different projects during my PhD. I would also like to thank Lars Hick, Kai Schaffarczyk, and Simon Hümmer for supporting me as students as part of their bachelor's or master's theses.

Further, I would also like to thank all co-authors who collaborated with me on various projects. In particular, the SPEC RG research groups on cloud computing and devops performance influenced my research through numerous discussions, inspiring talks, and critical feedback, here I would like to thank: Alexandru Iosup, Cristina Abad, Andre van Hoorn, Cor-Paul Bezemer, Weiyi Shang, Joel Scheuner,

Diego Elias Costa, Erwin van Eyk, Lizhi Liao, and more. I also would like to thank Jörg Domaschka Leznik, Daniel Seybold, and Mark Leznik from the University of Ulm for a fruitful and always fun collaboration.

Finally, I would like to thank my family and friends for their support and encouragement throughout the years, which made it all possible.

Bibliography

[ABV18] Cristina L Abad, Edwin F Boza, Erwin Van Eyk. "Package-aware scheduling of FaaS functions". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 101–106 (see page 188).

[AB17] Ali Abedi, Tim Brecht. "Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 287–292 (see pages 64, 68, 146, 147, 151).

[Ack+18] Vanessa Ackermann, Johannes Grohmann, Simon Eismann, Samuel Kounev. "Black-box Learning of Parametric Dependencies for Performance Models". In: *Proceedings of 13th International Workshop on Models@run.time (MRT)*. 2018 (see pages 5, 103).

[AC17a] Gojko Adzic, Robert Chatley. "Serverless Computing: Economic and Architectural Impact". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 884–889 (see pages 3, 17, 22, 35, 48, 84, 189).

[Aga+20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, Diana-Maria Popa. "Firecracker: Lightweight virtualization for serverless applications". In: *Proceedings of the 17th USENIX symposium on networked systems design and implementation*. 2020, pp. 419–434 (see page 4).

[AL19] Preyashi Agarwal, J. Lakshmi. "Cost Aware Resource Sizing and Scaling of Microservices". In: *Proceedings of the 2019 4th International Conference on Cloud Computing and Internet of Things*. 2019, pp. 66–74 (see pages 22, 72).

[Akh+20] Nabeel Akhtar, Ali Raza, Vatche Ishakian, Ibrahim Matta. "COSE: Configuring Serverless Functions using Statistical Learning". In: *Proceedings of the IEEE 2020 Conference on Computer Communications*. 2020, pp. 129–138 (see pages 4, 37).

[Akk+18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, Volker Hilt. "SAND: Towards High-Performance Serverless Computing". In: *Proceedings of the 2018 USENIX Annual Technical Conference*. 2018, pp. 923–935 (see page 5).

[Ale+09] Aldeida Aleti, Stefan Bjornander, Lars Grunske, Indika Meedeniya. "ArcheOpterix: An Extendable Tool for Architecture Optimization of AADL Models". In: *Proceedings of the 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. 2009, pp. 61–71 (see pages 5, 94, 166).

[Ali+20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, Evgenia Smirni. "Batch: machine learning inference serving on serverless platforms with adaptive batching". In: *Proceedings of the 2020 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 972–986 (see pages 4, 37, 81).

[Ali+14] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjöstedt-de Luna, Johan Tordsson, Erik Elmroth. "Measuring cloud workload burstiness". In: *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing*. 2014, pp. 566–572 (see page 132).

[Ali+17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, Ming Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. 2017, pp. 469–482 (see page 37).

[Alv+20] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel. "Sampling effect on performance prediction of configurable systems: A case study". In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 277–288 (see page 28).

[AGS08] Luigi Ambrosio, Nicola Gigli, Giuseppe Savaré. *Gradient flows: in metric spaces and in the space of probability measures*. Springer Science & Business Media, 2008 (see page 161).

[AC17b] Samaneh Aminikhanghahi, Diane J Cook. "A survey of methods for time series change point detection". In: *Knowledge and information systems* 51.2 (2017), pp. 339–367 (see page 25).

[AM20] Mahmoud Awad, Daniel A. Menasce. "IModel: Automatic Derivation of Analytic Performance Models". In: *ACM ToMPECS* 5.2 (2020) (see page 38).

[AWS14] AWS. *Introducing AWS Lambda*. `https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/`. 2014 (see page 1).

[BA18] Timon Back, Vasilios Andrikopoulos. "Using a microbenchmark to compare function as a service solutions". In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. 2018, pp. 146–160 (see pages 4, 37, 84, 85).

[BKN21] Bharathan Balaji, Christopher Kakovitch, Balakrishnan Narayanaswamy. "FirePlace: Placing Firecracker Virtual Machines with Hindsight Imitation". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 652–663 (see page 188).

[Bal+17] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, Olivier Tardieu. "The Serverless Trilemma: Function Composition for Serverless Computing". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2017, pp. 89–103 (see pages 89, 93, 122).

[Bal+04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, Marta Simeoni. "Model-based performance prediction in software development: A survey". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310 (see page 38).

[Bar+19] Daniel Barcelona-Pons, Pedro Garcia-Lopez, Alvaro Ruiz, Amanda Gomez-Gomez, Gerard Paris, Marc Sanchez-Artigas. "Faas orchestration of parallel workloads". In: *Proceedings of the 5th International Workshop on Serverless Computing*. 2019, pp. 25–30 (see page 5).

[BHK17] André Bauer, Nikolas Herbst, Samuel Kounev. "Design and Evaluation of a Proactive, Application-Aware Auto-Scaler". In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering*. 2017, pp. 425–428 (see page 115).

[BKR09] Steffen Becker, Heiko Koziolek, Ralf Reussner. "The Palladio Component Model for Model-driven Performance Prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22 (see pages 10, 37, 96, 98).

[BCS09] Marco Bertoli, Giuliano Casale, Giuseppe Serazzi. "JMT: performance engineering tools for system modeling". In: *SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 10–15 (see page 39).

[Bez+19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André Hoorn, Mónica Villavicencio, Jürgen Walter, Felix Willnecker. "How is Performance Addressed in DevOps?" In: *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2019, pp. 45–50 (see page 4).

[Bha+19] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, Gabor Karsai. "Barista: Efficient and scalable serverless serving system for deep learning prediction services". In: *Proceedings of the 2019 IEEE International Conference on Cloud Engineering*. 2019, pp. 23–33 (see page 50).

[Bis94] Christopher M Bishop. *Mixture density networks*. Tech. rep. 1994 (see pages 87, 89).

[Blo+19] Jakob Blomer, Gerardo Ganis, Simone Mosciatti, Radu Popescu. "Towards a serverless CernVM-FS". In: *EPJ Web of Conferences*. Vol. 214. 2019, p. 09007 (see page 50).

[Bol+98] Gunter Bolch, Stefan Greiner, Hermann Meer, Kishor S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, 1998 (see page 40).

[Bon+05] E. Bondarev, P. With, M. Chaudron, J. Muskens. "Modelling of input-parameter dependency for performance predictions of component-based embedded systems". In: *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*. 2005, pp. 36–43 (see pages 5, 42, 43, 96, 105, 108).

[BCK07] Egor Bondarev, Michel Chaudron, Erwin Kock. "Exploring Performance Trade-offs of a JPEG Decoder Using the Deepcompass Framework". In: *Proceedings of the 6th International Workshop on Software and Performance*. 2007, pp. 153–163 (see page 41).

[Bor15] Oliver Borchers. *A Hitchhiker's Guide to Mixture Density Networks*. `https://towardsdatascience.com/a-hitchhikers-guide-to-mixture-density-networks-76b435826cca`. 2015 (see page 89).

[BGK+10] Zdravko I Botev, Joseph F Grotowski, Dirk P Kroese. "Kernel density estimation via diffusion". In: *The annals of Statistics* 38.5 (2010), pp. 2916–2957 (see page 160).

[Boz+17] Edwin F Boza, Cristina L Abad, Monica Villavicencio, Stephany Quimba, Juan Antonio Plaza. "Reserved, on demand or serverless: Model-based simulations for cloud budget planning". In: *Proceedings of the 2017 IEEE Second Ecuador Technical Chapters Meeting*. 2017, pp. 1–6 (see pages 4, 5, 36, 163, 188).

[BG01] Sarah E Brockwell, Ian R Gordon. "A comparison of statistical methods for meta-analysis". In: *Statistics in medicine* 20.6 (2001), pp. 825–840 (see page 56).

[Bro14] Fabian Brosig. "Architecture-Level Software Performance Models for Online Performance Prediction". PhD thesis. Karlsruhe Institute of Technology, 2014 (see pages 10, 102).

[BHK11] Fabian Brosig, Nikolaus Huber, Samuel Kounev. "Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems". In: *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering*. 2011 (see pages 42, 108).

[BHK14] Fabian Brosig, Nikolaus Huber, Samuel Kounev. "Architecture-Level Software Performance Abstractions for Online Performance Prediction". In: *Elsevier Science of Computer Programming Journal (SciCo)* 90 Part B (2014), pp. 71–92 (see pages 41, 97, 108).

[Bro+15] Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, Samuel Kounev. "Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures". In: *IEEE TSE* 41.2 (2015), pp. 157–175 (see pages 5, 38, 115).

[BHS06] Erik Brynjolfsson, Yu Jeffrey Hu, Michael D. Smith. "From Niches to Riches: The Anatomy of the Long Tail". In: *Sloan Management Review* 47.2 (2006), pp. 67–71 (see page 175).

[Bur+21] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn. "Serverless Workflows with Durable Functions and Netherite". In: *arXiv preprint arXiv:2103.00033* (2021) (see page 5).

[Cad+20] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, Jonathan Appavoo. "SEUSS: Skip Redundant Paths to Make Serverless Fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020 (see pages 5, 189).

[Cas20a] Alex Casalboni. *AWS Lambda Power Tuning*. `https://github.com/alexcasalboni/aws-lambda-power-tuning`. 2020 (see pages 4, 37).

[Cas20b] Alex Casalboni. *Deep dive: finding the optimal resources allocation for your Lambda functions*. `https://bit.ly/3ihJBF7`. 2020 (see pages 23, 24, 79).

[CP17] Giuliano Casale, Dorina Petriu. "Tulsa: A Tool for Transforming UML to Layered Queueing Networks for Performance Analysis of Data Intensive Applications". In: *Proceedings of the 14th International Conference on Quantitative Evaluation of Systems*. Vol. 10503. 2017, p. 295 (see page 38).

[Cas+19] Paul Castro, Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54 (see pages 4, 9, 33, 48, 50, 188).

[Cha18] Mike Chan. *Containers vs. Serverless: Which Should You Use, and When?* `https://bit.ly/3rwMqpx`. 2018 (see page 48).

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2015. eprint: `arXiv:1511.07289` (see page 89).

[Cli93] Norman Cliff. "Dominance statistics: Ordinal analyses to answer ordinal questions." In: *Psychological bulletin* 114.3 (1993), p. 494 (see page 77).

[Clo20] Google Cloud. *Cloud Functions Pricing*. `https://bit.ly/3BYhOCv`. 2020 (see page 72).

[CA96] Amanda Coffey, Paul Atkinson. *Making sense of qualitative data: complementary research strategies.* Sage Publications, 1996 (see page 52).

[CSL20] Robert Cordingly, Wen Shu, Wes J Lloyd. "Predicting Performance and Cost of Serverless Computing Functions with SAAF". In: *Proceedings of the 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress*. 2020, pp. 640–649 (see pages 34, 66, 139).

[CMR09] Corinna Cortes, Mehryar Mohri, Afshin Rostamizadeh. "L 2 regularization for learning kernels". In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. 2009, pp. 109–116 (see page 89).

[Cre+19] Rodrigo Crespo-Cepeda, Giuseppe Agapito, Jose Luis Vazquez-Poletti, Mario Cannataro. "Challenges and opportunities of amazon serverless lambda services in bioinformatics". In: *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. 2019, pp. 663–668 (see pages 4, 32, 48, 124).

[Cui20] Yan Cui. *AWS Lambda Extensions: What are they and why do they matter*. 2020 (see pages 76, 82).

[Dal+20] David Daly, William Brown, Henrik Ingo, Jim O'Leary, David Bradford. "The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 67–75 (see page 67).

[Dal20] Jeremy Daly. *Serverless Community Study*. `https://github.com/jeremydaly/serverless-community-survey-2020`. 2020 (see page 73).

[Dat20] Datadog. *The State of Serverless*. 2020 (see pages 63, 72).

[Dat21] Datadog. *The State of Serverless*. `https://www.datadoghq.com/state-of-serverless/`. 2021 (see page 2).

[Ehl+11] Jens Ehlers, Andre Hoorn, Jan Waller, Wilhelm Hasselbring. "Self-adaptive Software System Monitoring for Performance Anomaly Localization". In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. 2011, pp. 197–200 (see page 105).

[Eis+20a] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, André Hoorn. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2020, pp. 265–276 (see pages 60, 147).

[Eis+21a] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, Samuel Kounev. "Sizeless: Predicting the Optimal Size of Serverless Functions". In: *Proceedings of the 22nd International Middleware Conference (MIDDLEWARE)*. 2021, pp. 248–259 (see page 10).

[Eis+22] Simon Eismann, Diego Elias Costa, Lizhi Liao, Cor-Paul Bezemer, Weiyi Shang, Andre Hoorn, Samuel Kounev. "A Case Study on the Stability of Performance Tests for Serverless Applications". In: *Journal of Systems and Software* (2022) (see pages 9, 69).

[Eis+20b] Simon Eismann, Johannes Grohmann, Erwin Eyk, Nikolas Herbst, Samuel Kounev. "Predicting the Costs of Serverless Workflows". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2020, pp. 265–276 (see pages 10, 83, 94).

[Eis+19] Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim Kistowski, Samuel Kounev. "Integrating Statistical Response Time Models in Architectural Performance Models". In: *Proceedings of the 27th IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 71–80 (see pages 11, 104).

[Eis+21b] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (2021), pp. 32–39 (see pages 9, 59, 69, 73, 144, 151).

[Eis+21c] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, Alexandru Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* (2021) (see pages 2, 9, 59).

[Eis+20c] Simon Eismann, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. *A Review of Serverless Use Cases and their Characteristics*. Tech. rep. 2020 (see pages 1, 9, 59, 73).

[Eis+18] Simon Eismann, Jürgen Walter, Jóakim Kistowski, Samuel Kounev. "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time". In: *Proceedings of the 26th IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 135–147 (see pages 11, 97, 115).

[Eiv17] Adam Eivy. “Be wary of the economics of" Serverless" Cloud Computing”. In: *IEEE Cloud Computing* 4.2 (2017), pp. 6–12 (see pages 4, 22, 35, 48, 125, 189).

[Elg18] Tarek Elgamal. “Costless: Optimizing the cost of serverless computing through function fusion and placement”. In: *Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing*. 2018, pp. 300–312 (see pages 4, 5, 36, 150, 163, 188).

[ES21] Nafise Eskandani, Guido Salvaneschi. “The Wonderless Dataset for Serverless Computing”. In: *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositorie*. 2021, pp. 565–569 (see page 33).

[Eyk+19] Erwin Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup. “The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms”. In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18 (see pages 2, 4, 18, 19, 22, 59, 119, 120).

[Eyk+18] Erwin Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, Simon Eismann. “A SPEC RG Cloud Group’s Vision on the Performance Challenges of FaaS Cloud Architectures”. In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE-C)*. 2018, pp. 21–24 (see pages 35, 37, 41, 60, 84).

[Eyk+20] Erwin Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, Alexandru Iosup. “Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark”. In: *Companion of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE-C)*. 2020, pp. 26–31 (see page 187).

[FH12] Michael Faber, Jens Happe. “Systematic Adoption of Genetic Programming for Deriving Software Performance Curves”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. 2012, pp. 33–44 (see pages 39, 100).

[Fen+18] Lang Feng, Prabhakar Kudva, Dilma Da Silva, Jiang Hu. “Exploring serverless computing for neural network training”. In: *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. 2018, pp. 334–341 (see pages 17, 32).

[Fig+18] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, Maciej Malawski. “Performance evaluation of heterogeneous cloud functions”. In: *Concurrency and Computation: Practice and Experience* 30.23 (2018), e4792 (see pages 4, 35, 36, 64, 187).

[Fog08] Agner Fog. “Calculation methods for Wallenius’ noncentral hypergeometric distribution”. In: *Communications in Statistics-Simulation and Computation®* 37.2 (2008), pp. 258–273 (see page 176).

[Fou+19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, Keith Winstein. “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers”. In: *Proceedings of the 2019 USENIX Annual Technical Conference*. 2019, pp. 475–488 (see pages 17, 32, 127, 189).

[Fou+17] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, Keith Winstein. "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads". In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. 2017, pp. 363–376 (see page 32).

[Fra+09] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, Salem Derisavi. "Enhanced modeling and solution of layered queueing networks". In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 148–161 (see page 39).

[Fri91] Jerome H Friedman. "Multivariate adaptive regression splines". In: *The annals of statistics* (1991), pp. 1–67 (see pages 99, 101).

[GMW97] David Garlan, Robert Monroe, David Wile. "Acme: An Architecture Description Interchange Language". In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. 1997, pp. 7–14 (see page 42).

[Gar20] Garner. *Gartner Says Worldwide IaaS Public Cloud Services Market Grew 40.7% in 2020*. `https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020`. 2020 (see page 1).

[Gar18] Gartner. *Worldwide IaaS Public Cloud Services Market Grew 31.3%*. 2018 (see page 120).

[Gea89] Loyd Geary. *Mixture Models: Inference and Applications to Clustering*. Vol. 152. Royal Statistical Society, 1989, pp. 126–127 (see page 87).

[GS01] Stanton A Glantz, Bryan K Slinker. *Primer of Applied Regression & Analysis of Variance, ed*. McGraw-Hill, Inc., 2001 (see page 80).

[GBB11] Xavier Glorot, Antoine Bordes, Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323 (see page 89).

[Göb+04] Steffen Göbel, Christoph Pohl, Simone Röttger, Steffen Zschaler. "The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects". In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. 2004, pp. 74–82 (see pages 10, 96, 98).

[GB08] Kwang-Il Goh, Albert-Laszlo Barabási. "Burstiness and memory in complex systems". In: *Europhysics Letters* 81.4 (2008), p. 48002 (see page 132).

[Gol+15] Alex Goldstein, Adam Kapelner, Justin Bleich, Emil Pitkin. "Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation". In: *Journal of Computational and Graphical Statistics* 24.1 (2015), pp. 44–65 (see page 78).

[GM01] Hassan Gomaa, Daniel A. Menasce. "Performance Engineering of Component-Based Distributed Software Systems". In: *Performance Engineering: State of the Art and Current Trends*. 2001, pp. 40–55 (see pages 42, 43).

[GMS07] Vincenzo Grassi, Raffaela Mirandola, Antonino Sabetta. "Filling the Gap Between Design and Performance/Reliability Models of Component-based Systems". In: *Journal of Systems and Software* 80.4 (2007), pp. 528–558 (see page 38).

[Gro+21] Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, Samuel Kounev. "SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31 (see page 38).

[Gro+19a] Johannes Grohmann, Simon Eismann, Sven Elflein, Manar Mazkatli, Jóakim Kistowski, Samuel Kounev. "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2019, pp. 309–322 (see page 86).

[Gro+19b] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, Diego Lugones. "Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning". In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 149–162 (see page 77).

[GMN11] Greg Guest, Kathleen M MacQueen, Emily E Namey. *Applied thematic analysis*. Sage Publications, 2011 (see page 52).

[Gun+19] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, Chita Das. "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud". In: *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. 2019, pp. 199–208 (see pages 4, 36, 188).

[Gwe14] Kilem L Gwet. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. LLC, 2014 (see page 53).

[Ham09] Dick Hamlet. "Tools and experiments supporting a testing-based theory of component composition". In: *ACM Transactions on Software Engineering and Methodology* 18.3 (2009), p. 12 (see pages 5, 42, 43, 105, 108).

[Hap+10] Jens Happe, Steffen Becker, Christoph Rathfelder, Holger Friedrich, Ralf H. Reussner. "Parametric Performance Completions for Model-driven Performance Prediction". In: *Perform. Eval.* 67.8 (2010), pp. 694–716 (see page 41).

[HBR14] Lucia Happe, Barbora Buhnova, Ralf Reussner. "Stateful Component-based Performance Models". In: *Softw. Syst. Model.* 13.4 (2014), pp. 1319–1343 (see pages 42, 43, 103).

[Hel+18] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, Chenggang Wu. "Serverless computing: One step forward, two steps back". In: *arXiv preprint arXiv:1812.03651* (2018) (see pages 4, 32, 48, 86).

[HDB11] Herodotos Herodotou, Fei Dong, Shivnath Babu. "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-Intensive Analytics". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 2011 (see page 37).

[His+02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, Kurt C. Wallnau. "Packaging Predictable Assembly". In: *Proceedings of the IFIP/ACM Working Conference on Component Deployment*. 2002, pp. 108–124 (see pages 10, 96, 98).

[Ho95] Tin Kam Ho. "Random Decision Forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. 1995, p. 278 (see pages 99, 101).

[HRD10] Kathryn Hoad, Stewart Robinson, Ruth Davies. "Automating warm-up length estimation". In: *Journal of the Operational Research Society* 61.9 (2010), pp. 1389–1403 (see page 65).

[Hub14] Nikolaus Huber. "Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures". PhD thesis. Karlsruhe Institute of Technology, 2014 (see page 10).

[Hub+17] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, Manuel Bähr. "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". In: *IEEE Transactions on Software Engineering* 43.5 (2017), pp. 432–452 (see pages 37, 38, 97, 111, 112, 114).

[IBM17] IBM. *Serverless Architectures in Banking: OpenWhisk on IBM Bluemix at Santander*. `https://developer.ibm.com/code/videos/tech-talk-replay-build-faster-banking-apps-ibm-cloud-functions/`. 2017 (see page 84).

[IDG20] IDG. *2020 Cloud Computing Study*. `https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/`. 2020 (see page 1).

[IYE11] Alexandru Iosup, Nezih Yigitbasi, Dick Epema. "On the Performance Variability of Production Cloud Services". In: *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2011, pp. 104–113 (see pages 4, 33, 145, 147).

[IMS18] Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski. "Serving deep learning models in a serverless platform". In: *Proceedings of the 2018 IEEE International Conference on Cloud Engineering*. 2018, pp. 257–262 (see pages 17, 50).

[IPW15] Farhana Islam, Dorina Petriu, Murray Woodside. "Simplifying Layered Queuing Network Models". In: *Proceedings of the European Workshop on Performance Engineering*. 2015, pp. 65–79 (see page 40).

[IPW18] Farhana Islam, Dorina Petriu, Murray Woodside. "Choice of Aggregation Groups for Layered Performance Model Simplification". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 241–252 (see pages 38, 40).

[JL81] Patricia A. Jacobson, Edward D. Lazowska. "The Method of Surrogate Delays: Simultaneous Resource Possession in Analytic Models of Computer Systems". In: *SIGMETRICS* 10.3 (1981), pp. 165–174 (see page 40).

[JK15] Kurt Jensen, Lars M Kristensen. "Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems". In: *Communications of the ACM* 58.6 (2015), pp. 61–70 (see page 38).

[JH15] Zhen Ming Jiang, Ahmed E. Hassan. "A Survey on Load Testing of Large-Scale Software Systems". In: *IEEE Trans. Software Eng.* 41.11 (2015), pp. 1091–1118 (see pages 4, 24, 144).

[Joh+19] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, Amanda Tan. "SWEEP: accelerating scientific research through scalable serverless workflows". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. 2019, pp. 43–50 (see page 5).

[Jon+19] Eric Jonas. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. 2019 (see page 1).

[KL19] Jeongchul Kim, Kyungyong Lee. "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service". In: *Proceedings of the 12th IEEE International Conference on Cloud Computing*. 2019, pp. 502–504 (see page 35).

[KB14] Diederik P. Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: `arXiv:1412.6980` (see page 159).

[KDK18] Jóakim Kistowski, Maximilian Deffner, Samuel Kounev. "Run-time Prediction of Power Consumption for Component Deployments". In: *Proceedings of the 15th IEEE International Conference on Autonomic Computing*. 2018 (see page 168).

[Kis+18] Jóakim Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, Samuel Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2018, pp. 223–236 (see page 62).

[KB06] Samuel Kounev, Alejandro Buchmann. "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation". In: *Performance Evaluation* 63.4-5 (2006), pp. 364–394 (see pages 96, 102, 104, 177).

[KD09] Samuel Kounev, Christofer Dutz. "QPME: a performance modeling tool based on queueing Petri Nets". In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 46–51 (see page 39).

[Kou+16] Samuel Kounev, Nikolaus Huber, Fabian Brosig, Xiaoyun Zhu. "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures". In: *Computer* 49.7 (2016), pp. 53–61 (see page 97).

[Kou+17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, Xiaoyun Zhu. *Self-Aware Computing Systems*. Springer-Verlag GmbH, 2017 (see page 5).

[Kou+21] Samuel Kounev. "Toward a Definition for Serverless Computing". In: *Serverless Computing (Dagstuhl Seminar 21201)*. Vol. 11 (5). 2021. Chap. 5.1 (see pages 1, 15, 58).

[KAM13] Anne Koziolek, Danilo Ardagna, Raffaela Mirandola. "Hybrid Multi-Attribute QoS Optimization in Component Based Software Systems". In: *Journal of Systems and Software* 86.10 (2013), pp. 2542–2558 (see page 115).

[Koz08] Heiko Koziolek. "Parameter dependencies for reusable performance specifications of software components". PhD thesis. Universität Oldenburg, 2008 (see pages 5, 41–43, 105, 108).

[Koz10] Heiko Koziolek. "Performance evaluation of component-based software systems: A survey". In: *Performance evaluation* 67.8 (2010), pp. 634–658 (see pages 5, 38, 39, 96, 98, 103, 173).

[KKR10] Klaus Krogmann, Michael Kuperberg, Ralf Reussner. "Using genetic search for reverse engineering of parametric behavior models for performance prediction". In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 865–877 (see pages 41, 86).

[KK17] Jörn Kuhlenkamp, Markus Klems. "Costradamus: A cost-tracing system for cloud-based software services". In: *Proceedings of the International Conference on Service-Oriented Computing*. 2017, pp. 657–672 (see page 4).

[Kwo+13] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, Yunheung Paek. "Mantis: Automatic performance prediction for smartphone applications". In: *Proceedings of the 2013 USENIX Annual Technical Conference*. 2013, pp. 297–308 (see page 39).

[LL18] Christoph Laaber, Philipp Leitner. "An evaluation of open-source software microbenchmark suites for continuous performance assessment". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 119–130 (see page 34).

[LSL19] Christoph Laaber, Joel Scheuner, Philipp Leitner. "Software microbenchmarking in the cloud. How bad is it really?" In: *Empirical Software Engineering* 24.4 (2019), pp. 2469–2508 (see pages 4, 34).

[LK77] J Richard Landis, Gary G Koch. "The measurement of observer agreement for categorical data". In: *Biometrics* (1977), pp. 159–174 (see page 53).

[Lau18] Mart Laul. *Serverless Case Study - Netflix*. 2018 (see page 124).

[Laz+84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984 (see page 40).

[LBH15] Yann LeCun, Yoshua Bengio, Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), p. 436 (see page 89).

[LSF18] Hyungro Lee, Kumar Satyam, Geoffrey Fox. "Evaluation of Production Serverless Computing Environments". In: *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. 2018, pp. 442–450 (see pages 4, 34, 187, 189).

[LC21] Youngsoo Lee, Sunghee Choi. "A Greedy Load Balancing Algorithm for FaaS Platforms". In: *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing (ICCBDC)*. 2021, pp. 63–69 (see page 188).

[LMM17] Jyri Lehvä, Niko Mäkitalo, Tommi Mikkonen. "Case study: building a serverless messenger chatbot". In: *Proceedings of the International Conference on Web Engineering*. 2017, pp. 75–86 (see page 32).

[LC16] Philipp Leitner, Jürgen Cito. "Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds". In: *ACM Trans. Internet Technol.* 16.3 (2016) (see pages 34, 66, 139, 145, 147).

[Lei+19] Philipp Leitner, Erik Wittern, Josef Spillner, Waldemar Hummer. "A mixed-method empirical study of Function-as-a-Service software development in industrial practice". In: *Journal of Systems and Software* 149 (2019), pp. 340–359 (see pages 4, 60, 187, 188).

[LP20] Valentina Lenarduzzi, Annibale Panichella. "Serverless testing: Tool vendors' and experts' points of view". In: *IEEE Software* 38.1 (2020), pp. 54–60 (see page 188).

[Les19] Heitor Lessa. *Production-grade full-stack apps with AWS Amplify*. `https://www.youtube.com/watch?v=DcrtvgaVdCU`. 2019 (see pages 60, 150).

[Lev20] Ella Levinson. *Serverless Community Survey 2020*. 2020 (see pages 3, 48).

[Li+19] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, Dan Li. "Understanding Open Source Serverless Platforms: Design Considerations and Performance". In: *Proceedings of the 5th International Workshop on Serverless Computing*. 2019, pp. 37–42 (see page 16).

[LGF05] Yan Liu, Ian Gorton, Alan Fekete. "Design-level performance prediction of component-based applications". In: *IEEE Transactions on Software Engineering* 31.11 (2005), pp. 928–941 (see page 42).

[Llo+18] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, Shrideep Pallickara. "Serverless computing: An investigation of factors influencing microservice performance". In: *Proceedings of the 2018 IEEE International Conference on Cloud Engineering*. 2018, pp. 159–169 (see pages 4, 34, 146, 187, 189).

[LFC03] Jeffrey D. Long, Du Feng, Norman Cliff. *Ordinal Analysis of Behavioral Data*. John Wiley & Sons, 2003 (see pages 66, 67).

[Lop+20] Pedro Garcia Lopez, Aitor Arjona, Josep Sampe, Aleksander Slominski, Lionel Villard. "Triggerflow: trigger-based orchestration of serverless workflows". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 3–14 (see page 5).

[Lop+18] Pedro Garcia Lopez, Marc Sanchez-Artigas, Gerard Paris, Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, David Arroyo Pinto. "Comparison of faas orchestration systems". In: *Companion of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing*. 2018, pp. 148–153 (see page 5).

[Mae+20] Pieter-Jan Maenhaut, Bruno Volckaert, Veerle Ongenae, Filip De Turck. "Resource Management in a Containerized Cloud: Status and Challenges". In: *J. Netw. Syst. Manag.* 28.2 (2020), pp. 197–246 (see page 48).

[MI04] Prasad Mahajan, Ricki Ingalls. "Evaluation of Methods Used to Detect Warm-Up Period in Steady State Simulation". In: *Proceedings of the 36th conference on Winter simulation*. 2004, pp. 663–671 (see pages 25, 65, 146).

[MK21] Nima Mahmoudi, Hamzeh Khazaei. "SimFaaS: A Performance Simulator for Serverless Computing Platforms". In: *arXiv preprint arXiv:2102.08904* (2021) (see pages 37, 188).

[Mah+19] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, 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*. 2019, pp. 203–213 (see page 188).

[Maj+18] Szymon Majewski, Michal Aleksander Ciach, Michal Startek, Wanda Niemyska, Blazej Miasojedow, Anna Gambin. "The Wasserstein Distance as a Dissimilarity Measure for Mass Spectra with Application to Spectral Deconvolution". In: *Proceedings of the 18th International Workshop on Algorithms in Bioinformatics*. Vol. 113. 2018, 25:1–25:21 (see page 161).

[Mak93] Spyros Makridakis. "Accuracy measures: theoretical and practical concerns". In: *International journal of forecasting* 9 (1993), pp. 527–529 (see page 57).

[Mal19] Nathan Malishev. *AWS Lambda Cold Start Language Comparisons, 2019 edition*. `https://bit.ly/ColdStartComp`. 2019 (see page 120).

[MW47] H. B. Mann, D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *Ann. Math. Statist.* 18.1 (1947), pp. 50–60 (see pages 66, 67, 77).

[Mar20] MarketsAndMarkets. *Serverless Architecture Market*. `https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html`. 2020 (see page 2).

[MA04] R Timothy Marler, Jasbir S Arora. "Survey of multi-objective optimization methods for engineering". In: *Structural and multidisciplinary optimization* 26.6 (2004), pp. 369–395 (see page 29).

[Mar+10] Anne Martens, Heiko Koziolek, Steffen Becker, Ralf Reussner. "Automatically Improve Software Architecture Models for Performance, Reliability, and Cost Using Evolutionary Algorithms". In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. 2010, pp. 105–116 (see pages 94, 166).

[MAB21] Anil Mathew, Vasilios Andrikopoulos, Frank J Blaauw. "Exploring the cost and performance benefits of AWS Step Functions using a data processing pipeline". In: *Proceedings of the IEEE/ACM 14th International Conference on Utility and Cloud Computing*. 2021 (see page 5).

[Maz+20] Manar Mazkatli, David Monschein, Johannes Grohmann, Anne Koziolek. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *Proceedings of the 2020 IEEE International Conference on Software Architecture*. 2020, pp. 23–34 (see page 42).

[McK17] John McKim. *Serverless Event Sourcing at Nordstrom*. `https://bit.ly/3bVX6bp`. 2017 (see page 150).

[MV00] Daniel A. Menasce, A. F. Almeida Virgilio. *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*. PTR, 2000 (see pages 115, 170, 178, 179).

[MHS19] Michalis Michaelides, Jane Hillston, Guido Sanguinetti. "Statistical abstraction for multi-scale spatio-temporal systems". In: *ACM Transactions on Modeling and Computer Simulation* 29.4 (2019), pp. 1–29 (see page 38).

[MG20] Sascha Moellering, Steffen Grunwald. *Field Notes: Optimize your Java application for AWS Lambda with Quarkus*. `https://amzn.to/3mqZYBg`. 2020 (see page 120).

[MM02] Adrian Mos, John Murphy. "A Framework for Performance Monitoring, Modelling and Prediction of Component Oriented Distributed Systems". In: *Proceedings of the 3rd International Workshop on Software and Performance*. 2002, pp. 235–236 (see page 42).

[MMA20] Ingo Müller, Renato Marroquin, Gustavo Alonso. "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 115–130 (see page 32).

[MWL10] Jerome L Myers, Arnold Well, Robert Frederick Lorch. *Research design and statistical analysis*. Routledge, 2010 (see page 57).

[Nam+16] Manoj Nambiar, Ajay Kattepur, Gopal Bhaskaran, Rekha Singhal, Subhasri Duttagupta. "Model Driven Software Performance Engineering: Current Challenges and Way Ahead". In: *SIGMETRICS Perform. Eval. Rev.* 43.4 (2016), pp. 53–62 (see pages 5, 39, 96, 103, 173).

[Niu+19a] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, Ka Yee Yeung. "Leveraging Serverless Computing to Improve Performance for Sequence Comparison". In: *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. 2019, pp. 683–687 (see pages 17, 189).

[Niu+19b] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, Ka Yee Yeung. "Leveraging serverless computing to improve performance for sequence comparison". In: *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. 2019, pp. 683–687 (see page 32).

[Noo+13] Qais Noorshams, Dominik Bruhn, Samuel Kounev, Ralf Reussner. "Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013, pp. 283–294 (see pages 28, 39, 100).

[Noo+14] Qais Noorshams, Roland Reeb, Andreas Rentschler, Samuel Kounev, Ralf Reussner. "Enriching Software Architecture Models with Statistical Models for Performance Prediction in Modern Storage Environments". In: *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*. 2014, pp. 45–54 (see page 40).

[ORe19] O'Reilly. *O'Reilly serverless survey 2019: Concerns, what works, and what to expect*. `https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect/`. 2019 (see page 2).

[Oak+18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *Proceedings of the 2018 USENIX Annual Technical Conference*. 2018, pp. 57–70 (see pages 5, 189).

[Orf18] Antoni Orfin. *How Droplr Scales to Millions With The Serverless Framework*. `https://bit.ly/3ejIWTu`. 2018 (see page 48).

[Ost+14] Per-Olov Ostberg, Henning Groenda, Stefan Wesner, James Byrne, Dimitrios S Nikolopoulos, Craig Sheridan, Jakub Krzywda, Ahmed Ali-Eldin, Johan Tordsson, Erik Elmroth. "The CACTOS vision of context-aware cloud topology optimization and simulation". In: *Proceedings of the Cloud Computing Technology and Science*. 2014, pp. 26–31 (see pages 105, 108).

[Pah+19] Claus Pahl, Antonio Brogi, Jacopo Soldani, Pooyan Jamshidi. "Cloud Container Technologies: A State-of-the-Art Review". In: *IEEE Trans. Cloud Comput.* 7.3 (2019), pp. 677–692 (see page 48).

[Pap+19] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tůma, Alexandru Iosup. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* 47.8 (2019), pp. 1528–1543 (see pages 68, 147, 151, 156).

[PAM19] Ilja Pavlov, Susanne Ali, Tauhid Mahmud. *Serverless Development Trends in Open Source: a Mixed-Research Study*. Bach. Thesis. 2019 (see pages 48, 49).

[Pir+15] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Rajkumar Buyya. "Efficient Virtual Machine Sizing for Hosting Containers as a Service". In: *Proceedings of the 2015 IEEE World Congress on Services*. 2015, pp. 31–38 (see pages 22, 72).

[Rad16] Ben Rady. *Serverless single page apps: Fast, scalable, and available*. Pragmatic Bookshelf, 2016 (see page 17).

[Rao19] Singiresu S Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 2019 (see page 80).

[Res20] Allied Market Research. *Function-as-a-Service Market*. `https://www.alliedmarketresearch.com/function-as-a-service-market-A06072`. 2020 (see page 1).

[Reu+16] Ralf H Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, Klaus Krogmann. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016 (see pages 5, 38).

[Rom+06] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, Linda Devine. "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices". In: *Proceedings of the Annual meeting of the Southern Association for Institutional Research*. 2006, pp. 1–51 (see page 66).

[RN09] Stuart Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009 (see pages 89, 103).

[SJ18] Aakanksha Saha, Sonika Jindal. "EMARS: Efficient Management and Allocation of Resources in Serverless". In: *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing*. 2018, pp. 827–830 (see page 5).

[SS06] Burhaneddin Sandıkçı, İhsan Sabuncuoğlu. "Analysis of the behavior of the transient period in non-terminating simulations". In: *European Journal of Operational Research* 173.1 (2006), pp. 252–267 (see pages 65, 146).

[ST07] N Sato, Kishor S Trivedi. "Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks". In: *Proceedings of the International Conference on Service-Oriented Computing*. 2007, pp. 107–118 (see page 115).

[SDQ10] Jörg Schad, Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz. "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance". In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 460–471 (see pages 4, 33).

[SL20] Joel Scheuner, Philipp Leitner. "Function-as-a-Service Performance Evaluation: A Multivocal Literature Review". In: *Journal of Systems and Software* (2020) (see pages 4, 35, 120, 189).

[Ser19] Amazon Web Services. *Build On Serverless - Architect an Airline Booking Application*. `https://pages.awscloud.com/GLOBAL-devstrategy-OE-BuildOnServerless-2019-reg-event.html`. 2019 (see pages 60, 149).

[Ser20a] Amazon Web Services. *AWS Compute Optimizer*. `https://go.aws/3G5R0Tp`. 2020 (see page 83).

[Ser20b] Amazon Web Services. *AWS Lambda Pricing*. `https://go.aws/3aTYQRv`. 2020 (see page 72).

[Sha+20] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, Ricardo Bianchini. *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*. 2020. arXiv: `2003.03423 [cs.DC]` (see pages 4, 5, 69, 151, 188).

[Sin+20] Rayman Preet Singh, Bharath Kumarasubramanian, Prateek Maheshwari, Samarth Shetty. "Auto-sizing for Stream Processing Applications at LinkedIn". In: *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing*. 2020 (see pages 22, 72).

[Sin+21] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, Aditya Akella. "Atoll: A Scalable Low-Latency Serverless Platform". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 138–152 (see page 5).

[Sin+19] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, Aditya Akella. *Archipelago: A Scalable Low-Latency Serverless Platform*. 2019. arXiv: `1911.09849 [cs.DC]` (see page 189).

[Sit+01] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, A. L. N. Reddy. "Performance Specification of Software Components". In: *SIGSOFT Softw. Eng. Notes* 26.3 (2001), pp. 3–10 (see pages 42, 43, 105, 108).

[Skl+19] Tyler J Skluzacek, Ryan Chard, Ryan Wong, Zhuozhao Li, Yadu N Babuji, Logan Ward, Ben Blaiszik, Kyle Chard, Ian Foster. "Serverless workflows for indexing large scientific data". In: *Proceedings of the 5th International Workshop on Serverless Computing*. 2019, pp. 43–48 (see page 5).

[SW02] Connie U Smith, Lloyd G Williams. "New software performance antipatterns: More ways to shoot yourself in the foot". In: *Proceedings of the Int. CMG Conference*. 2002, pp. 667–674 (see page 136).

[SA19] Josef Spillner, Mohammed Al-Ameen. *Serverless Literature Dataset*. `https://doi.org/10.5281/zenodo.1175423`. 2019 (see pages 48, 49).

[SMM18] Josef Spillner, Cristian Mateos, David A. Monge. "FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC". In: *High Performance Computing*. 2018, pp. 154–168 (see page 122).

[Spi17] Simon Spinner. "Self-Aware Resource Management in Virtualized Data Centers". PhD thesis. University of Würzburg, 2017 (see page 105).

[Spi+15] Simon Spinner, Giuliano Casale, Fabian Brosig, Samuel Kounev. "Evaluating Approaches to Resource Demand Estimation". In: *Performance Evaluation* 92.C (2015), pp. 51–71 (see page 26).

[Spi+19] Simon Spinner, Johannes Grohmann, Simon Eismann, Samuel Kounev. "Online model learning for self-aware computing infrastructures". In: *Journal of Systems and Software* 147 (2019), pp. 1–16 (see page 97).

[Str18] Bernd Strehl. *The largest benchmark of Serverless providers*. `https://bit.ly/3BYmwzX`. 2018 (see page 4).

[The+10] Eno Thereska, Bjoern Doebel, Alice X. Zheng, Peter Nobel. "Practical Performance Models for Complex, Popular Applications". In: *SIGMETRICS Performance Evaluation Review* 38.1 (2010), pp. 1–12 (see page 39).

[TLL18] Zhucheng Tu, Mengping Li, Jimmy Lin. "Pay-per-request deployment of neural network models using serverless architectures". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. 2018, pp. 6–10 (see pages 32, 50).

[VL18] Jose Luis Vazquez-Poletti, Ignacio Martin Llorente. "Serverless computing: from planet mars to the cloud". In: *Computing in Science & Engineering* 20.6 (2018), pp. 73–79 (see pages 32, 36).

[Ven+16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*. 2016, pp. 363–378 (see page 37).

[VVI18] Laurens Versluis, Erwin Van Eyk, Alexandru Iosup. "An Analysis of Workflow Formalisms for Workflows with Complex Non-Functional Requirements". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 107–112 (see page 90).

[Vos08] David Vose. *Risk analysis: a quantitative guide*. John Wiley & Sons, 2008 (see page 90).

[WI03] Kurt Wallnau, James Ivers. *Snapshot of CCL: A Language for Predictable Assembly*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 2003 (see page 42).

[Wal19a] Jeff Walter. *Systematic Data Transformation to Enable Web Coverage Services and ArcGIS Image Services within ESDIS Cumulus Cloud*. 2019 (see page 50).

[Wal19b] Jürgen Christian Walter. "Automation in Software Performance Engineering Based on a Declarative Specification of Concerns". PhD thesis. Universität Würzburg, 2019 (see page 10).

[Wan+18a] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, Michael Swift. "Peeking behind the curtains of serverless platforms". In: *Proceedings of the 2018 USENIX Annual Technical Conference*. 2018, pp. 133–146 (see pages 4, 34, 60, 64, 187, 189).

[Wan+18b] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, Michael Swift. "Peeking behind the curtains of serverless platforms". In: *Proceedings of the 2018 USENIX Annual Technical Conference*. 2018, pp. 133–146 (see pages 4, 36, 75).

[WL21] Jinfeng Wen, Yi Liu. "A measurement study on serverless workflow services". In: *Proceedings of the 2021 IEEE International Conference on Web Services*. 2021, pp. 741–750 (see page 5).

[Wes+12] Dennis Westermann, Jens Happe, Rouven Krebs, Roozbeh Farahbod. "Automated inference of goal-oriented performance prediction functions". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 190–199 (see pages 28, 39, 100).

[WM10] Dennis Westermann, Christof Momm. "Using Software Performance Curves for Dependable and Cost-efficient Service Hosting". In: *Proceedings of the 2Nd International Workshop on the Quality of Service-Oriented Software Systems*. 2010, 3:1–3:6 (see page 39).

[WCS00] K.P. White, M.J. Cobb, S.C. Spratt. "A comparison of five steady-state truncation heuristics for simulation". In: *Proceedings of the 2000 Winter Simulation Conference Proceedings*. Vol. 1. 2000, pp. 755–760 (see pages 65, 146).

[Wil17] Alan Williams. *Autodesk Goes Serverless in the AWS Cloud, Reduces Account-Creation Time by 99%*. `https://amzn.to/2Q3X0pV`. 2017 (see pages 84, 127).

[Wit+20] Philipp A Witte, Mathias Louboutin, Henryk Modzelewski, Charles Jones, James Selvage, Felix J Herrmann. "An event-driven approach to serverless seismic imaging in the cloud". In: *IEEE Transactions on Parallel and Distributed Systems* 31.9 (2020), pp. 2032–2049 (see pages 4, 17, 32, 48, 124, 126, 189).

[Woh+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012 (see page 68).

[Woo20] Julian Wood. *Introducing AWS Lambda Extensions – In preview*. `https://amzn.to/3qhiJHJ`. 2020 (see page 76).

[WFP07] Murray Woodside, Greg Franks, Dorina C. Petriu. "The Future of Software Performance Engineering". In: *Proceedings of the 2007 Conference on the Future of Software Engineering*. 2007, pp. 171–187 (see pages 5, 39, 96, 103, 173).

[WPS02] Murray Woodside, Dorin Petriu, Khalid Siddiqui. "Performance-related Completions for Software Specifications". In: *Proceedings of the 24th International Conference on Software Engineering*. 2002, pp. 22–32 (see page 41).

[WW04] Xiuping Wu, Murray Woodside. "Performance Modeling from Software Components". In: *Proceedings of the 4th International Workshop on Software and Performance*. 2004, pp. 290–301 (see pages 10, 42, 96, 98).

[Yan+16] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian. "Building a chatbot with serverless computing". In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. 2016, pp. 1–4 (see page 17).

[YRC07] Yuan Yao, Lorenzo Rosasco, Andrea Caponnetto. "On early stopping in gradient descent learning". In: *Constructive Approximation* 26.2 (2007), pp. 289–315 (see page 159).

[Yu+20] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, Haibo Chen. "Characterizing serverless platforms with serverlessbench". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 30–44 (see pages 34, 187).

[Yus+19] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, Christian Müller. "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 273–283 (see pages 22, 150).

[ZJI15] Mohd ZM Zaki, Dayang NA Jawawi, Mohd Adham Isa. "Integrated MARTE-based Model for Designing Component-Based Embedded Real-Time Software". In: *International Journal of Software Engineering and Its Applications* 9.3 (2015), pp. 157–174 (see page 38).

[Zha+20] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, Vincent Liu. "Fault-tolerant and transactional stateful serverless workflows". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 2020, pp. 1187–1204 (see page 5).

[Zha+19] Miao Zhang, Yifei Zhu, Cong Zhang, Jiangchuan Liu. "Video processing with serverless computing: a measurement study". In: *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 2019, pp. 61–66 (see pages 4, 36).

[ZCS07] Qi Zhang, Ludmila Cherkasova, Evgenia Smirni. "A regression-based analytic model for dynamic resource provisioning of multi-tier applications". In: *Proceedings of the Fourth International Conference on Autonomic Computing*. 2007, pp. 27–36 (see page 38).

[Zha+18] Qingchen Zhang, Laurence T Yang, Zhikui Chen, Peng Li. "A survey on deep learning for big data". In: *Information Fusion* 42 (2018), pp. 146–157 (see page 86).

[ZH19] Armin Zimmermann, Thomas Hotz. "Integrating simulation and numerical analysis in the evaluation of generalized stochastic Petri nets". In: *ACM Transactions on Modeling and Computer Simulation* 29.4 (2019), pp. 1–25 (see page 38).

[Zsc10] Steffen Zschaler. "Formal specification of non-functional properties of component-based software systems". In: *Software and Systems Modeling* 9.2 (2010), pp. 161–201 (see pages 42, 43).