43 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

A Simulation Framework for Function as a Service

Johannes Manner

University of Bamberg Press

43 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 43

University of Bamberg Press
2024

A Simulation Framework for Function as a Service

Johannes Manner

University
of Bamberg
Press
2024

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Dr. h. c. Frank Leymann
Tag der mündlichen Prüfung: 21.12.2023

Herstellung und Druck: Prime Rate, Budapest
Umschlaggestaltung: University of Bamberg Press

ISSN: 1867-6197 (Print)
ISBN: 978-3-86309-978-7 (Print)
eISSN: 2750-8560 (Online)
eISBN: 978-3-86309-979-4 (Online)

URN: urn:nbn:de:bvb:473-irb-929135
DOI: https://doi.org/10.20378/irb-92913

Words are like bees

–

some leave a sting and others create honey

-UNKNOWN-

Acknowledgments

Over the last six years at the university I had so many great and enriching moments with a wide variety of people for which I am very grateful! Unfortunately, I cannot mention all of them by name, so I would like to take this opportunity at the beginning to say a big thank you to those who have walked a part of this interesting way with me.

I would particularly like to thank my supervisor Prof. Dr. Guido Wirtz for providing me with the chance to grow with the challenges research- and teaching-wise. Especially the undergraduate course *programming complex systems (PKS)*, where low level concurrency mechanism were taught, enhanced my way of thinking. Also special thanks to the other members of my dissertation committee, Prof. Dr. Udo Krieger and Prof. Dr. Sven Overhage and their valuable feedback during the colloquium. I would like to thank my external examiner Prof. Dr. Dr. h. c. Frank Leymann. I got to know him as one of the main organizers of SummerSoC, a summer school on service oriented computing in Crete, which I attended four times. I heard a lot of inspiring talks and made good friends and met one of the kindest people I ever met - Stavros, an exquisite chef and bartender. I will really miss the raki nights 🍸.

It's impossible to have a great workplace without amazing people. First and foremost all current and former members of Distributed Systems Group (DSG), Sebastian Böhm, Robin Lichtenthäler, Stefan Winzinger, Stefan Kolb, Andreas Schönberger, Simon Harrer and Matthias Geiger. You were there anytime to support me when I needed advice. I cannot think about a better place to work! Without Cornelia Schecher's help in administrative matters I would not be able to write these lines but would still be filling out forms. I am also very grateful that we share the same passion - beekeeping 🐝 - and used spare time to share knowledge with each other, talking about the weather and the tons of honey we will harvest. Also I really enjoyed the non-sense talks on Conversational Fridays, a joint work with the members of SWT research group, in particular Eugene Yip who comes up with the funniest topics ever. The last two years, I was engaged in committee work together with colleagues from other faculties, in particular Nicole K. Konopka, Susann Sachse-Thürer and Johannes Zenk. Without you I probably would have resigned to convince people to fight for more digital processes. My résumé after two years: The mills grind slowly but they grind. A constant in these six years journey was ZEUS, a small workshop on services aimed for supporting young researchers and early PhD students. My appreciation goes to Oliver Kopp who is still the driving force of this workshop and all the other steering and PC

members. The feedback, reviews and discussion were more valuable for my work than the big conferences I attended.

The other side of my PhD position was teaching which I really loved! I had a lot of students challenging me and vice versa. I am especially proud of the papers we wrote together, in particular with Robin Hartauer, Benedikt Full, Tobias Heckel and Martin Endreß. With Martin I even wrote two. Many thanks to him and Tobias which also contributed to my research prototype. We had a lot of implementation fun years ago during my first university project as a PhD student.

My particular appreciation goes to my proof-readers Leonie Fidler, Sebastian Böhm and Robin Lichtenthäler. Without your help the text would not be half as good! Thanks for your countless hours in reading paper drafts, looking at concurrency code and finally reading the next 190 pages also on the behalf of all the others who read this work!

Last but not least, I am grateful for my housing project at *Brenni* and all the flat mates who have become friends. Finally, a lot of love to my parents and sister for their ongoing support in all my efforts. I apologize for any inconvenience this may have caused you ☺.

Kurzfassung

Serverless Computing wird als Wegbereiter für den Betrieb großer Anwendungen gesehen. Obwohl Entwickler und Forscher diesen Begriff oft verwenden, ist das Konzept, über das sie eigentlich sprechen möchten, als Function as a Service (FaaS) bekannt. Bei diesem neuen Servicemodell schreiben FaaS Nutzer einzelne Funktionen und stellen diese auf Cloud-Plattformen bereit. Der Cloud-Anbieter kümmert sich um alle betrieblichen Belange - somit scheint es sich aus der Sicht des Nutzers um serverloses Computing zu handeln.

Dennoch ist es bei den meisten kommerziellen FaaS-Plattformen notwendig, einige Funktionskonfigurationen vorzunehmen, da sie die Ressourcenzuweisungen, insbesondere für CPU und Hauptspeicher, beeinflussen. Vergleicht man wesentliche Cloud Computing Charakteristiken bei den Modellen Platform as a Service und FaaS, so zeigt sich bei den beiden Dimensionen Elastizität und Granularität der Serviceabrechnung eine Verbesserung. FaaS ist das erste Cloud Servicemodell, das Funktionen bei Bedarf innerhalb weniger Millisekunden skaliert. Aufgrund unabhängiger Skalierung und starker Isolation durch virtualisierte Umgebungen können Funktionskonfigurationen als unabhängig von anderen Cloud-Funktionen angesehen werden. Daher sind keine Noisy Neighbor Probleme zu beobachten. FaaS-Plattformen messen die Ausführungszeit in Millisekunden und stellen sie Nutzern auf Grundlage der Funktionskonfiguration in Rechnung (Granularität der Serviceabrechnung). Dies führt zu neuen Leistungs- und Kostenabwägungen.

In dieser Arbeit wird ein Simulationsansatz vorgeschlagen, um diesen Trade-off in einer frühen Entwicklungsphase zu untersuchen. Eine Alternative zu unserem Simulationsansatz wäre die Funktionen mit verschiedenen Konfigurationen produktiv zu betreiben, die Ausführungsdaten von mehreren FaaS-Plattformen zu analysieren und die Konfiguration anzupassen. Dies würde allerdings zu Mehrkosten und einem höheren Aufwand führen. Um eine realistische Simulation zu ermöglichen, sollten die Entwicklungs- und Produktionsumgebung so ähnlich wie möglich sein. Diese Ähnlichkeit wird auch als Dev-Prod-Parität bezeichnet. Basierend auf einer neuen Methodik zum Vergleich verschiedener virtualisierter Umgebungen, können Benutzer unseres Simulationsframeworks Funktionen auf ihren Rechnern ausführen und die Laufzeiteigenschaften verschiedener Konfigurationen auf verschiedenen Cloud-Plattformen untersuchen. Eine Visualisierung der lokalen Simulationen hilft den Nutzern dabei, eine geeignete Funktionskonfiguration zu wählen, um den erwähnten Trade-off den Anforderungen entsprechend bestmöglich aufzulösen.

Abstract

Serverless Computing is seen as a game changer in operating large-scale applications. While practitioners and researches often use this term, the concept they actually want to refer to is Function as a Service (FaaS). In this new service model, a user deploys only single functions to cloud platforms where the cloud provider deals with all operational concerns – this creates the notion of server-less computing for the user.

Nonetheless, a few configurations for the cloud function are necessary for most commercial FaaS platforms as they influence the resource assignments like CPU time and memory. Due to these options, there is still an abstracted perception of servers for the FaaS user. The resource assignment and the different strategies to scale resources for public cloud offerings and on-premise hosted open-source platforms determine the runtime characteristics of cloud functions and are in the focus of this work. Compared to cloud offerings like Platform as a Service, two out of the five cloud computing characteristics improved. These two are rapid elasticity and measured service. FaaS is the first computational cloud model to scale functions only on demand. Due to an independent scaling and a strong isolation via virtualized environments, functions can be considered independent of other cloud functions. Therefore, noisy neighbor problems do not occur. The second characteristic, measured service, targets billing. FaaS platforms measure execution time on a millisecond basis and bill users accordingly based on the function configuration. This leads to new performance and cost trade-offs.

Therefore, this thesis proposes a simulation approach to investigate this trade-off in an early development phase. The alternative would be to deploy functions with varying configurations, analyze the execution data from several FaaS platforms and adjust the configuration. However, this alternative is time-consuming, tedious and costly. To provide a proper simulation, the development and production environment should be as similar as possible. This similarity is also known as dev-prod parity. Based on a new methodology to compare different virtualized environments, users of our simulation framework are able to execute functions on their machines and investigate the runtime characteristics for different function configurations at several cloud platforms without running their functions on the cloud platform at all. A visualization of the local simulations guide the user to choose an appropriate function configuration to resolve the mentioned trade-off dependent on their requirements.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ACPI	Advanced Configuration and Power Interface
ACU	Azure Compute Unit
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CaaS	Container as a Service
CFS	Complete Fair Scheduler
cgroups	control groups
CLI	Command Line Interface
CMS	Cambridge Monitor System
CNCF	Cloud-Native Computing Foundation
CNY	Chinese Yuan Renminbi
CP	Control Program
CRI	Container Runtime Interface
DSG	Distributed Systems Group
DTO	Data Transfer Object
EKS	Elastic Kubernetes Service
FaaS	Function as a Service
GFLOPS	Giga FLoating Point Operations Per Second
GKE	Google Kubernetes Engine
HPC	High Performance Computing
HWP	Hardware-Managed P-states
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
ICU	International Telecommunication Union
JIT	Just-in-Time
JPA	Java Persistence API
JVM	Java Virtual Machine

JWT	JSON Web Token
K8s	Kubernetes
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
libOS	Library Operating System
LXC	Linux Container
ML	Machine Learning
NIST	National (US) Institute of Standard and Technology
OAS	OpenAPI Specification
OCI	Open Container Initiative
OLTP	Online Transaction Processing
ORM	Object Relational Mapping
OS	Operating System
PaaS	Platform as a Service
PCI	Peripheral Component Interconnect
QoS	Quality of Service
REST	Representational State Transfer
RTT	Round Trip Time
SaaS	Software as a Service
SLA	Service Level Agreement
SLR	Structured Literature Review
SPEC	Standard Performance Evaluation Corporation
SUT	System Under Test
TCO	Total Cost of Ownership
TPC-C	Transaction Processing Performance Council Benchmark C
UI	User Interface
USB	Universal Serial Bus
vCPU	virtual CPU
VM	Virtual Machine
VMM	Virtual Machine Monitor
XaaS	Everything as a Service

Part I.

Background and Problem Identification

1. Introduction

Parts of this chapter have been taken from [174, 180–182, 185].

Severless Computing is seen as a game changer in operating large-scale applications. For the first time, developers are able to fully focus on the implementation of their apps rather than on configuring servers. However, when deploying a cloud function to a commercial FaaS platform, a user has to specify some settings which influence the scaling of resources and therefore the machine's configuration. The first chapter of this work states this configuration problem in Section 1.1 and proposes a solution for it. A summary of the main contributions published while working on the dissertation project is given in Section 1.2 followed by a detailed agenda stating research questions and the research methodology. This introduction is concluded by an outline in Section 1.4[1].

1.1. Context

When cloud computing started to gain popularity, it promised many advantages to developers. These advantages are essentially the cloud computing capabilities defined by the National (US) Institute of Standard and Technology (NIST) [192]: deliver computing resources elastically at scale, offer measured services where the user pays per use, enable on demand self-service, pool resources and access services via standardized network interfaces. Back in September 2011, NIST defined three *as a Service* models, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), to shape the cloud computing landscape and provide a common understanding of terms and characteristics. The main differentiation made between the three models is based on the responsibility the user (or the provider) has:

IaaS is a model to "provision [...] fundamental computing resources where the consumer is able to deploy and run arbitrary software" [192, p. 3]. An IaaS provider handles procurement of machines and provides basic compute, storage and networking resources. The user is free to use this generic toolbox to run their preferred Operating System (OS) or other custom libraries and tools.

PaaS is more restrictive in the sense that a consumer is able "to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider" [192,

[1] All links in the thesis have been last accessed on 24th of July, 2023. Their status for preserving the content when addressed for this thesis is archived at Zenodo [176].

p. 2-3]. This loss in flexibility compared to IaaS comes with pre-configured runtime environments which allows consumers to focus on the core business capabilities of their application rather than patching their systems.

SaaS is at the end of a spectrum. Here the consumers "use the provider's applications running on a cloud infrastructure" [192, p. 2] out of the box with limited application-specific configuration options. In this service model, all operational work is handled by the cloud provider.

Since the initial classification of *as a Service* offerings by NIST is restricted to these three service models and there was no update since their initial publication, the Everything as a Service (XaaS) movement started [69]. As the name implies this model is an umbrella term of new service models which e.g. transfer database offerings to the cloud [146] or rethink and recombine established services. A lot of *new* service models originated from this XaaS movement but all of them can be primarily attributed to one of the three originally defined service models.

In late 2014, Amazon Web Services (AWS) introduced a new compute service called AWS Lambda[2] which attracted a lot of attention in industry and academia. With Lambda, AWS implemented a platform where a user has the option to deploy ephemeral and stateless cloud functions which run only upon request. Based on their product ideas, FaaS was introduced as a new service model tightly related to PaaS. The most important difference is the granularity: On PaaS platforms, self-contained applications are being deployed, whereas on FaaS platforms, only single-purposed, stateless cloud functions can be deployed. A further movement in more granular services from microservices to nanoservices is observable [3, 173, 256]. Due to this focus on a single functionality, some cloud computing characteristics, especially *rapid elasticity* and *measured service,* have been improved to such an extent that it is justifiable to speak of a new service model. FaaS is seen as the next evolution in cloud computing by several researchers [41, 250, 275]. What's more, ten years after their initial "view on cloud computing" [5], researchers at the Berkeley University published an update [113] in which they claim that Serverless will dominate the cloud in future. As FaaS is the predominant reason for the Serverless hype where a cloud computing user hands over all operational work to the provider, many practitioners and researchers use the terms Serverless and FaaS interchangeably even if technically they want to refer to FaaS. Since there is no common terminology yet, the two terms Serverless and FaaS will be differentiated in Section 3.1. As a short preview it can be said that we categorize FaaS as a subset of Serverless technologies since it is not the only cloud service offering which hides server configuration and management from the user. For example, all SaaS offerings also hide them from the user. Therefore, in the following FaaS is explicitly used to address the cloud function concept.

When looking at the cloud computing characteristics listed at the beginning, it can be seen that *rapid elasticity* and *measured service* are the focus of FaaS. Due to the stateless nature of cloud functions, a FaaS platform is conceptually capable

[2] https://aws.amazon.com/de/about-aws/whats-new/2014/11/13/introducing-aws-lambda/

of starting a new instance for every single request and tearing it down right after executing the function. Furthermore, the influence of one cloud function on a related function, like executing two functions in sequence, is reduced to a minimum since the functions are decoupled by communicating only via events. Therefore, functions can be scaled independently of each other. There is no noisy neighbor problem at public cloud providers [17] and a solution to overcome this problem in on-premise open-source FaaS systems is proposed in Section 6.4. Thus, a user can optimize the cloud function configuration for each function in isolation dependent on its specific requirements. During periods without user requests, there is no bottom line of always running cloud function instances. This enables real elasticity from zero to an arbitrary number of instances. In most of the current cloud platforms and open source tools, the platform starts a function instance for the first request and keeps the instance warm for a short time without billing the user. This leads us to the second important improvement – the *measured service* characteristic.

FaaS is the first compute service where the public cloud provider offers a pay-as-you-go billing model. A user is only charged for the time their functions are actually running. AWS Lambda launched the first commerical offering with an already fine-grained billing model of 100 millisecond chunks in 2015 but announced to improve their billing model to a millisecond basis in December 2020[3]. This unique billing scheme for public cloud providers raises new performance/cost trade-off questions. This is in contrast to on-premise hosted open-source platforms, where no metering and therefore no billing guidelines exist. The question of how to price on-premise hosted solutions is still unanswered. To address this trade-off, there are two important factors to consider. Firstly, the execution time of a cloud function determines the price based on the millisecond billing scheme. Secondly, the configuration of cloud function instances is used for scaling resources and also defines the price per millisecond. For example, doubling the resource assignment for a function results in doubling the price per millisecond but would also halve the execution time in an ideal world. In such a scenario a user would be charged equally for all configurations, as can be seen in Fig. 12 in the experiment of Figiela and others [79]. Leaving the ideal world scenario of functions fully utilizing the assigned resources, a user is facing the challenge to configure cloud functions appropriately based on their requirements to meet latency constraints without wasting resources and in the end money.

When comparing this configuration problem with applications deployed to a PaaS environment, two aspects make it apparent why the configuration problem and the implications for cloud functions are worth investigating. The first reason is the time period during which a PaaS application is running. Virtual Machines (VMs) are typical deployment targets in PaaS. Applications run for hours not milliseconds, also during idle periods. The price per request is thus influenced by the

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

utilization of the system. For cloud functions, instead, each request is executed and billed independently without any instances being idle. The second aspect is scaling. In PaaS scenarios one instance is always running to serve incoming requests, independent of the workload. Further instances are deployed for example when the CPU utilization of the first instance exceeds a certain threshold. Therefore, a user chooses the configuration in such a way that the application instance can handle predicted peak loads and configures the platform to scale further instances if needed. Based on the elasticity property of FaaS there is no consideration whether the function can handle peak loads. If a peak occurs, the FaaS platform scales horizontally by creating new instances. The performance/cost trade-off is solely based on the chosen programming language and the function configuration determining the vertical resource assignment for every single cloud function instance. Choosing the right configuration isn't easy and guiding a user to find a suitable one is the motivation of this thesis project.

1.2. Contributions

To provide the aforementioned user guidance for a proper cloud function configuration, three contributions were made to the scientific community:

(C1) New hypotheses, concepts and methods to build a simulation framework for cloud functions are proposed to predict the runtime behavior for different configuration parameters during the development process. New insights on how to calibrate different virtualized execution environments are gained by calibration data which are used to calculate equivalent settings for the local developer's machine and cloud platforms.

(C2) The research prototype implements the proposed methods and evaluates them within the already published papers. Furthermore, the tool is able to simulate arbitrary cloud functions locally as long as they are packaged as Open Container Initiative (OCI) compliant images and presents guidance for cloud function configuration via its web User Interface (UI).

(C3) This work provides an understanding about the applied scaling strategies for public cloud providers as well as on-premise open-source hosted FaaS platforms. Many research papers doing empirical research did not consider multi-threaded functions despite the capabilities of several cloud platforms to assign resources of multiple cores to a single cloud function. This often led to incomplete or wrong conclusions about the resource scaling strategies of cloud providers. By analyzing these research papers and performing experiments with multi-threaded cloud functions, we overcome misinterpretations and guide users to consider multi-threaded code.

The main contribution of the dissertation project is to build a simulation framework for FaaS (C1) to find the best function configuration for a cloud function

deployed to a FaaS platform based on the user's requirements. To reach this goal, we propose a simulation to be run on the developer's machine at an early stage of the software development lifecylce, since deploying functions, executing them and collecting data for analysis is time-consuming and introduces further efforts. A precondition for a meaningful user guidance is parity of the local (dev) and cloud environment (prod). This dev-prod parity is one of the Twelve-Factor app principles[4] to build SaaS applications. It does not necessarily mean that developers have to buy the same hardware as used by the cloud provider in their computing centers. Rather, it suggests that developers configure their system and functions locally in a comparable way to the cloud. This calibration of the local developer's machine and the corresponding cloud platform is the main contribution of the work in hand. The proposed calibration and the executed simulations were validated via benchmarks on different FaaS platforms as well as local machines. In the end, the implemented tool executes several simulations and gives guidance to users on how to configure a cloud function based on the user constraints.

Individual aspects of this process have already been published. Table 1.1 summarizes all papers submitted to peer-reviewed conferences and workshops. The technical report [174] describes and explains the research prototype. The author of this thesis is also the first author of all published papers except for the "Considerations for Portability" [96] which originated from a master thesis. The work in hand is based on these already published papers, brings them into a consistent shape and adds further insights. At the beginning of each section, references indicate whether the respective section is based on already published work.

Table 1.1.: Publications by type and year.

	Type	Year	Where
[175]	Conference	2023	CLOUD
[180]	Conference	2022	CLOUD
[96]	Conference	2022	CLOSER
[174]	Technical Report	2021	Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik
[185]	Conference	2021	CLOUD
[181]	Conference	2021	SOSE
[179]	Summer School	2019	SummerSoC
[183]	Journal	2019	SummerSoC
[173]	Workshop	2019	ZEUS
[182]	Workshop	2018	WoSC

The following enumeration is a detailed list of the relevant publications for this thesis already presented in Table 1.1:

[175] J. Manner: "A Structured Literature Review Approach to Define Serverless Computing and Function as a Service", In Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), 2023.

[4] https://12factor.net/dev-prod-parity

[180] J. Manner and G. Wirtz: "Resource Scaling Strategies for Open-Source FaaS Platforms compared to Commercial Cloud Offerings", In Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), 2022.

[96] R. Hartauer, J. Manner and G. Wirtz: "Cloud Function Lifecycle Considerations for Portability in Function as a Service", In Proceedings of International Conference on Cloud Computing and Service Science (CLOSER), 2022.

[174] J.Manner: "SeMoDe – Simulation and Benchmarking Pipeline for Function as a Service", Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 105, University of Bamberg Press, November 2021.

[185] J. Manner, M. Endreß, S. Böhm and G. Wirtz: "Optimizing Cloud Function Configuration via Local Simulations", In Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), 2021.

[181] J. Manner and G. Wirtz: "Why Many Benchmarks Might Be Compromised", In Proceedings of the IEEE International Conference on Service-Oriented System Engineering (SOSE), 2021.

[179] J. Manner and G. Wirtz: "Impact of Application Load in Function as a Service", In Proceedings of Symposium and Summer School On Service-Oriented Computing (SummerSoC), 2019.

[183] J. Manner, S. Kolb and G. Wirtz: "Troubleshooting Serverless functions: a combined monitoring and debugging approach", In SICS Software-Intensive Cyber-Physical Systems, 2019.

[173] J. Manner: "Towards Performance and Cost Simulation in Function as a Service", In Proceedings of Central European Workshop on Services and their Composition (ZEUS), 2019.

[182] J. Manner, M. Endreß, T. Heckel and G. Wirtz: "Cold Start Influencing Factors in Function as a Service", In Proceedings of the Workshop on Serverless Computing (WoSC), 2018.

Apart from the vision paper [173], the portability consideration work [96] and the term definitions [175], the methodological parts of all other papers were evaluated using the research prototype *SeMoDe* [174]. This research prototype forms the second contribution (C2). The acronym was coined at the beginning of the dissertation project and stands for **Se**rverless **Mo**nitoring and **De**bugging. It was inspired by the first paper which was published in Software-Intensive Cyber-Physical Systems Journal [183]. Based on the design of this research, a small benchmarking component was needed for executing functions and analyzing the logs. This

small component and the possibilities to make empirical research caught our attention which resulted in the overall idea presented in 2019 as a vision paper at ZEUS [173]. A detailed introduction of the capabilities of `SeMoDe` is presented in Section 4.3. `SeMoDe` is freely available under a MIT license[5] and implemented in Java using SpringBoot and Docker containers for executing simulated functions on a local machine. It is hosted on GitHub[6] and deployed on the DSG cluster[7].

Overcoming misinterpretation of multi-threaded code (C3) was motivated by both, research and teaching. When reading details of some research papers, e.g., Fig. 2 in [154], Fig. 1. in [72], Sec. 6.2 in [48], Sec. 6 in [189], Tab. 3 in [17] or Fig. 7 in [70], it was apparent that the authors of these papers did not mention nor consider multi-threaded cloud functions. The authors wonder why they faced constant execution times despite increased resources (vertical scaling). Since most platforms provide only a configuration of memory and scale CPU and other resources proportionally, it is not apparent without reading the details of the documentation at which level a cloud function gets assigned more than a single core. The authors of the aforementioned papers tried to explain these phenomena with short execution times inducing greater prediction errors, the rounding of execution times to 100 ms blocks (at AWS Lambda back in 2021), or could not find a reason at all. None of these explanations are convincing when reading the papers and looking at the presented data. To the best of the author's knowledge, only a single publication [293] includes multi-threaded functions in their evaluation but also made a misinterpretation due to the resource setting chosen. This observation lead to the assumption that resource allocation (vertical scale-out) and multi-threading on a function level are often neglected. Most experiments address the rapid scalability property of FaaS platforms and only look at horizontal scale-out [17, 119]. Additionally, researchers argue that cloud functions are often used when implementing *glue code* [54, 110, 148], for example to integrate microservices with each other. A vertical scale-out is not in focus in such cases, but experiments dealing with different cloud function configurations which ignore this aspect are bound to infer incorrect interpretation and results.

Another motivation to address this aspect was lecturing an advanced undergraduate course on concurrency programming in Java. Most undergraduate computer science courses focus on sequential programming within a single thread despite the fact that since IBM released their first multi-core processor[8] 20 years ago, there have been many hardware achievements with regard to parallel computing. To close the gap between the single threaded undergraduate courses and great textbooks like "Java Concurrency in Practice" [87] or "Effective Java" [26], we wrote the "Lecture Notes: Concurrency Topics in Java" [178] to summarize insights of

[5] https://opensource.org/licenses/MIT
[6] https://github.com/johannes-manner/SeMoDe
[7] https://semode.pi.uni-bamberg.de/
[8] https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/

the challenges and pitfalls students encounter when learning to write concurrent code and to guide them in their efforts.

During the dissertation project the author had the pleasure to work in various roles within the scientific community. He served as a program committee chair for the Central European Workshop on Services and their Composition (ZEUS) and edited the proceedings from 2020-2022 [184, 186, 187]. The idea to connect young researchers to allow them to discuss their visions for the dissertation projects is one of ZEUS's success factors. The constructive atmosphere helped to develop new ideas and to stay focused. Additionally, many papers were reviewed for several conferences namely the Conference on Software Engineering and Knowledge Engineering (SEKE2019-2021), Conference on Utility and Cloud Computing (UCC2019-2021), Conference on Omni-layer Intelligent Systems (COINS2020-2021), Conference on Cooperative Information Systems (COPIS2022) and Conference on Cloud Computing (CLOUD2022).

1.3. Research Questions

The following research questions are more specific about the contributions and already outline the main part of the thesis. They are clustered in five sections: Conceptualization, Benchmarking FaaS Platforms, Achieving Dev-Prod Parity, Providing User Guidance and Improving Cold Starts.

1.3.1. Conceptualization

As mentioned before there is still no widely accepted terminology for the new service model FaaS due to the continuously evolving research area. This motivates the first part of this thesis as well as the first research question:

Research Question 1.1:
Which characteristics define a FaaS offering?

In particular, the term *FaaS* is differentiated from *Serverless Computing* by giving an overview of existing definitions and summarizing key characteristics of a FaaS offering. We try to understand the evolution of terms based on search trends and share best practices on how to be precise with regards to term usage while still ensuring visibility. Since FaaS is perceived as a new cloud service model, the next research questions addresses the relation to other as a Service models especially the ones defined by NIST:

Research Question 1.2:
How is FaaS related to other service models in the cloud computing landscape?

Furthermore, a market analysis is conducted to identify currently available providers and open-source solutions and their technical realization. Especially the

different virtualization options for building a FaaS platform like Unikernels, VMs, or containers are discussed. In addition, an overview is provided on how different platforms implement scaling strategies, in particular how they assign resources to the deployed cloud functions. For open-source platforms, it is apparent that Kubernetes (K8s) seems to be a common abstraction [12, 113, 180] to delegate scaling and other operational tasks to.

1.3.2. Benchmarking FaaS Platforms

A quote from W. Edwards Deming (1900-1993), "without data, you're just another person with an opinion.", sparked two research questions about how to benchmark FaaS platforms and what best practices of benchmark approaches and their data generation could look like. First, it is necessary to understand the current state of research by looking at benchmarking studies which were conducted during the last years. Based on a Structured Literature Review (SLR), initially done while writing a technical documentation of the research prototype [174] and updated for this thesis, the following question is answered:

Research Question 2.1:
Which tools and experiments do currently exist for benchmarking FaaS platforms?

What is evident when reading relevant research papers is that only a few experiments specify all their settings or make source code and data publicly available. Nevertheless, documentation of all settings is essential to interpret the results explained in the papers and give other researchers the option to reproduce research and therefore verify results. In an SLR conducted by Kuhlenkamp and Werner [137], they found that only three out of 26 FaaS benchmarking experiments published until 2018 provided all the necessary information. Furthermore, when looking at other research, like Fig. 4 in [167] where two different clusters of measured execution times exist, phenomena are apparent but some data is missing to properly interpret the results. In the aforementioned case, the missing data is the VM and machine configuration. Based on these observations, the following research question wants to tackle this problem of incomplete data acquisition by specifying a catalog of necessary data:

Research Question 2.2:
How should a FaaS experiment be documented and which items are necessary for data evaluation?

Explanations are given on how relevant data is measured and stored persistently with the help of the research prototype. SeMoDe's system architecture as well as the database model is aligned with insights about incomplete experiment documentation and aspects identified when answering research question 2.2.

1.3.3. Achieving Dev-Prod Parity

After understanding the systems' resource allocation via benchmarking, the main concern of the work in hand is achieving dev-prod parity and simulating the execution behavior of cloud functions. This work and the presented data focus on simulating CPU intensive functions like LINPACK, recursive Fibonacci, and prime number search as often done in empirical FaaS research, e.g. in [138, 182, 185, 213, 255, 271]. The main reasons for choosing these functions are a good understanding of their runtime characteristics, their wide-spread use, and therefore comprehensible results and discussions.

One of the Twelve-Factor App guidelines recommends that development and production environments should be "as similar as possible"[4]. The idea of this dissertation project to simulate a cloud function during development in order to make a prediction about the runtime behavior comes with a few challenges. The machine performing the simulation should be equipped comparably with the deployment target. While performing an experiment on one of the local machines at the chair, a situation occurred where the scaling of resources was unpredictable due to the CPU frequency scaling algorithms used. This leads to the following question:

Research Question 3.1:
How can a consistent CPU scaling behavior across various processors and scaling algorithms be achieved and visualized?

Experiments were performed to test the CPU scaling behavior under different Linux kernel settings. We used the introduced benchmarking facilities to collect data on a public FaaS platform and calibrated the local environment accordingly. On the local machine, Docker containers are used to control assigned resources using container quotas (cgroups). On the provider side, the computing performance depends on the selected resource setting. To perform the calibration, functions which implement LINPACK [65, 66] - a widely used benchmark assessing CPU performance - were executed. The results are measured in Giga FLoating Point Operations Per Second (GFLOPS). When executing the calibration function for different quotas or function settings, the GFLOPS value indicates the container respectively cloud function's performance. This data is used to compute linear regression models for making the scaling of resources statistically and graphically visible. After a consistent scaling of resources locally and for the respective FaaS platform has been achieved, the next question arises:

Research Question 3.2:
How can two distinct virtualized execution environments be made comparable?

SPRUNT [258] emphasizes that processor implementations are mostly abstracted by *program characterization events*, like floating point events. Since the calibration

already computes GFLOPS, this data is used to compute the linear regression models to determine a mathematical function by equating the two regressions. At this point, settings of the target FaaS platform can be selected and comparable resource assignments for executing the local simulation can be determined. This leads to the evaluation and the next research question:

Research Question 3.3:
Do resource configurations based on calibration lead to accurate predictions on a provider-hosted FaaS platform in the cloud?

To evaluate the proposed methodology, functions were executed locally and in the cloud with the computed settings from RQ 3.2 to compare the execution times and compute trends. As mentioned before, a lot of research papers do not consider multi-threaded functions. Therefore, experiments were performed with several single as well as multi-threaded functions. This is particularly interesting when comparing the different CPU equivalents especially for configurations where a setting exceeds the resource equivalent of more than one core. Besides fortunate coincidence, the one CPU equivalent differs between the local environment and the target platform. This should be kept in mind when interpreting the results.

Up until now, we only considered resource scaling in the cloud where a lot of empirical research has already been conducted and documented by two SLRs about benchmarking FaaS systems [174, 240]. These SLRs state that performance benchmarking is rarely done for open-source offerings which results in unfair comparisons to the cloud since Quality of Service (QoS) attributes and proper resource scaling algorithms are not in place for open-source platforms. Therefore, the last research question in this chapter is concerned with this issue:

Research Question 3.4:
How can resource scaling strategies be applied to on-premise open-source FaaS platforms in a manner that is equivalent to cloud strategies?

If the resource scaling strategy for on-premise open-source platforms is equivalent to those on a cloud FaaS platform, the introduced simulation approach can be extended to open-source platforms as well. As a notable side effect, the limiting of resources for the execution instance of a cloud function on an on-premise hosted open-source FaaS platform can solve the noisy neighbor problem. As already mentioned, this problem is not apparent in the cloud as research showed [17]. Furthermore, different configurations of on-premise hosted cloud functions allow a pricing per invocation as done by public cloud providers and provides a company with the option to transform their IT department's method of settlement from a cost to a profit center.

1.3.4. Providing User Guidance

Performance and cost are two conflicting goals a user should consider when configuring a cloud function. As stated in the beginning, FaaS is the first service model where a user pays for the compute time they consume without idling. After presenting empirical data which shows that the simulation approach predicts execution times reliably, a user guidance is offered by clarifying the following question:

Research Question 4:
How can developers be supported in making reasonable decisions about their cloud function configurations?

With the help of `SeMoDe`, execution data of the local simulations are presented to the user graphically. Simulated data are displayed for different function configurations and a user of the prototype can select a target configuration. Based on this, an ideal cost function is displayed helping the user to assess whether the function profits from an resource increase/decrease and how this is related to the cost of the function execution. The UI additionally includes information about single- respectively multi-core limits to raise awareness when a cloud function configuration exceeds this limit.

1.3.5. Guidance for Improving Cold Starts

The hype about FaaS can be explained by the elastic scalability of functions per request. One downside of this property is that it entails a lot of cold starts. Cold starts happen when starting a function instance for the first time and introduce additional latency for spinning up the execution environment. Due to performance reasons, FaaS providers do not shut down the cloud function instances immediately. Subsequent executions use already existing containers to profit from a provisioned execution environment. This is the reason why cold starts are one of the most discussed performance aspects in empirical FaaS research, e.g. in [157, 182]. To mitigate the problem, there are some anti-patterns to avoid cold starts by artificially keeping the function instances warm. This is often done by pinging the endpoint on a regular basis [145, 273] to fake actual demand. To avoid such mitigation strategies, the last research question deals with influential factors and provides an understanding of how the cold start overhead can be reduced:

Research Question 5:
Which factors influence the cold start behavior of a function besides the function configuration?

To answer this question a hypotheses based investigation was performed and complementary insights from related work were added. It was evident during

data evaluation that the user perceived cold start times include additional overheads not included in the metering service of the FaaS platform. When computing the difference between a cold and a warm execution on the same container instance, situations occurred where the reported cold start overhead as measured by the metering service in the cloud was 7 times lower than the difference between cold and warm execution on the client side [182]. Therefore, since the aim is to guide users to configure their functions properly, implementation decisions of developers are questioned in order to reduce the cold start period for the user.

1.4. Outline

The outline of this thesis is as follows: The most important foundations, namely virtualization, benchmarking and simulation, are introduced in Section 2. These foundations form the background of this work and conclude Part I.

Part II is devoted to Function as a Service. RQ1.1 and RQ1.2 are answered in Section 3 where a conceptualization of FaaS and a distinction to established as a Service models are provided. Commercial and open-source FaaS platforms as well as their corresponding resource scaling strategies are listed followed by a typical architecture of a FaaS worker node.

The main contributions are addressed in Part III where RQ2-RQ5 are answered. Current benchmarking approaches are discussed in Section 4 to answer RQ2.1. This detailed discussion resulted in a checklist and documentation guidance for FaaS experiments as raised by RQ2.2. To enforce the checklist and comply to a reproducible experimental design, a custom benchmark research prototype is proposed - `SeMoDe`. The architecture of the research prototype is discussed along the insights from the SLR. During the experiments described in Section 5, the custom scaling driver `intel_pstate` showed a non-linear scaling of resources. This motivated research on RQ3.1. The idea to equalize two virtual execution environments provides an answer to RQ3.2 which is discussed in Section 6 where an evaluation of the proposed methodology confirms that it leads to accurate predictions, which was challenged by RQ3.3. For RQ3.4, the proposed methodology is applied to an open-source platform to incorporate comparable research with public cloud offerings. Section 7 contains the answer to RQ4 as it explains how `SeMoDe` can be used to provide guidance for users to select a proper cloud function configuration. It also addresses cold start influencing factors thereby answering RQ5.

Part IV concludes the thesis by discussing the most important competing approaches to the work in hand. Furthermore, the main contributions are summarized and a vision on performance aware computing is proposed for future work.

2. Theoretical and Technical Foundations

Parts of this chapter have been taken from [185].

First of all there are some important foundations which need to be laid down. Section 2.1 introduces the range of virtualization and container technology options. It provides a detailed overview of technology stacks and their benefits and drawbacks.

In Section 2.2, benchmarking and related terms are defined and characteristics for proper benchmarking are stated. Based on the discussion about benchmarking, a discussion about simulation per se and its approaches follows in Section 2.3.

2.1. Virtualization

2.1.1. Motivation

"One server, one application" [223, p.9] was the modus operandi of running applications and operating servers in the early days of computing. In the 1950s, sharing of resources between processes or even users was not considered due to security concerns, performance, portability and complexity issues [227]. Since then and due to improvements in hardware as described by Moore's Law, components on an integrated circuit doubled every year [163]. Hence, servers have become more powerful and therefore underutilized with serving only a single application.

In a first step, this underutilization problem was tackled by multiprocessing. This means that every process should be isolated and unaffected by other processes of the same user on the same machine. However, since technology evolved further, even multiple processes of a single user did not fully utilize a machine. Therefore, the idea of multiprocessing advanced even further to multi-user processing which means that each user can run several processes while it seems as if they were the only user on the system [227]. Consolidation and containment [223, 269] were the driving forces for this development. *Virtual Machines* and *containers* help to realize this facet of isolation in order to better utilize physical machines. These developments lead to the abstraction of hardware by implementing specialized software, in particular hypervisors and container runtimes. In the VM case, several users of the system are able to run their own OS for each VM, whereas containers enable users to run their own virtualized OS in an isolated process of a shared host OS. Both developments also lead to the possibility of migrating applications from one physical machine to another [269].

This means that virtualization is the enabler of cloud computing. It creates flexibility for the provider and user of cloud services as well as for operation engineers hosting on-premise applications. Therefore, in the following sections, an introduction is given to the two most important virtualization techniques namely virtual machines and unikernels. Furthermore, container technology is discussed as an alternative to VMs. At the end of this section, performance experiments are examined which compare VM, container and bare metal performance. Application virtualization like the Java Virtual Machine (JVM) will not be considered here.

2.1.2. Virtual Machine

Virtual Machines are "a feature of a computer's operating system that allows multiple other systems to be run on the same computer, each with its own operating environment"[9]. Based on this definition, VMs are standalone, encapsulated *systems* abstracted from the physical machine via virtual components necessary to run own OSs. Before introducing the building blocks of a VM solution, we offer a little bit of history and a retrospective on early term usage.

In the mid 1960s, IBM started to develop VMs. Back then, they named their host OS Cambridge Monitor System (CMS) and the software for creating and managing VMs Control Program (CP) [88, 227]. The first VM going by today's standards was called CP-40/CMS for an IBM System/360 (model 40) in the late 60s [227]. Popek and Goldberg formulated formal virtualization requirements which a Virtual Machine Monitor (VMM) nowadays referred to as hypervisor [239][10], or CP in IBM terms, has to fulfill to create virtual machines as an "*efficient, isolated duplicate* of the real machine" [221, p. 413]:

1. "Provides an environment for programs which is essentially identical with the original machine".
   This requirement entails that an application is executable on a virtual machine despite it being designed for the physical machine without modifying the source code. The only exception are system resources as for example less memory is attached to a VM or some resources are temporarily unavailable since they are in use by another concurrently running VM [221]. Paravirtualization [285] approaches where the guest OS has been adapted to a

[9] https://www.oxfordlearnersdictionaries.com/definition/english/virtual-machine

[10] There is no common terminology what characterizes and distinguishes a hypervisor from a VMM in literature. In the early days, the term VMM was more common like in [88, 221]. Nowadays the term hypervisor is more often used. This observation is confirmed when searching on dblp for the term *Hypervisor* (548 entries on 23rd of September 2022) and *Virtual Machine Monitor* (74 entries on 23rd of September). The two terms are most often used in recent literature as synonyms like in [1, 227, 239]. But there is also research like Bugnion and others [32] which defines the VMM as a special part of the hypervisor controlling CPU and memory virtualization. Based on these insights, the work in hand also subsumes all virtualization components necessary to run VMs under the term hypervisor.

hardware abstraction of the hypervisor like in Xen [19] are not considered as hypervisors in their sense.

2. "Programs [...] show at worst only minor decreases in speed".
   Popek and Goldberg explained in more detail that a substantial number of instructions are directly executed on the processor without additional hypervisor interception. Therefore, simulators and emulators like QEMU [22][11] which translate or intercept commands from the virtual to the physical machine are no hypervisor in their sense due to performance downsides [32].

3. "VMM is in complete control of system resources".
   The VM is unable to access system resources other than those assigned to it and can only reclaim unused resources which were already assigned to the VM [221]. VMs are isolated from each other based on disjunctively assigned system resources.

Nowadays, hypervisors are categorized in three groups. Figure 2.1 shows them and some implementations currently available. The classification is based on the technology stack, the component in control of resources and how the interaction with the hardware is organized.

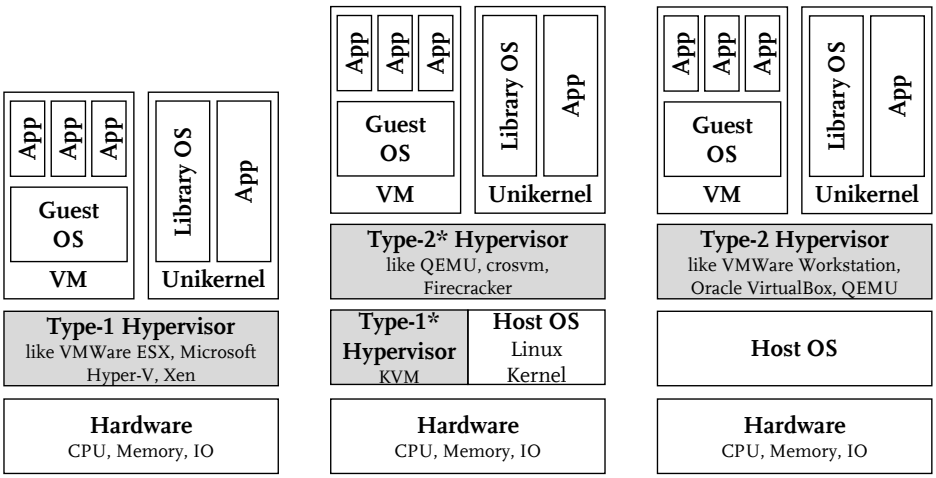


Figure 2.1.: Hypervisors and their corresponding technology stack based on the responsibility of resource allocation [1, 32, 223].

As indicated by Figure 2.1, in a type-1 hypervisor scenario - also called bare metal virtualization [239] - the hypervisor runs directly on top of the hardware. Para-virtualization solutions, where the guest OS has to be adapted to the hypervisor interface, are included since they are also recognized as bare metal virtualization [19]. Established software products in this category are VMWare ESX[12],

[11] https://www.qemu.org/
[12] https://www.vmware.com/content/vmware/vmware-published-sites/us/products/esxi-and-esx.html.html

Microsoft's Hyper-V[13] and Xen[14]. A type-2 hypervisor, also called hosted virtualization [239], uses some features from the host OS and handles the virtualization functionality by hardware emulation. VMWare Workstation[15], Oracle's VirtualBox[16] and QEMU are examples for type-2 hypervisors. A type-2 hypervisor therefore encapsulates virtualization logic and uses basic OS features from the host OS. In contrast, a type-1 hypervisor re-implements OS features and therefore combines basic resource allocation features, scheduling and the virtualization logic. A recent performance investigation [61] showed that type-1 hypervisors are more performant which can be explained by direct interaction with the hardware without an indirection using a host OS and the corresponding hardware emulation done by the hypervisor.

Indicated by the project website[17], Kernel-based Virtual Machine (KVM) [125] has both notions of hypervisors and consists of a *kernel component* and a *userspace component*. The focus of KVM is to extend the Linux OS to support the creation of virtual machines. It has been integrated in the Linux kernel since version 2.6.20 as a kernel module. Every Linux OS now has built-in support for VM virtualization [94]. This has two advantages: KVM solely focuses on virtualization features and directly interacts with the hardware for handling virtual machines like type-1 hypervisors despite reimplementing OS functionalities like schedulers. Since KVM is not a standalone hypervisor, it is considered a type-1* hypervisor. The second advantage is that it directly benefits from OS Linux kernel updates. The *userspace component* was also developed within the KVM projects and is therefore tightly integrated with the kernel component. The use of underlying OS and virtualization features make it a type-2 hypervisor. Since QEMU version 1.3[17], the KVM userspace component was integrated in the emulator and extended QEMU's capabilities to become a virtualization solution on top of a Linux host OS. Due to this coupling and the improved virtualization part, it is considered a type-2* hypervisor. Both subtypes (1* and 2*) form a unit. The combination of KVM/QEMU has shown the best performance for CPU, disk, network and IO workloads in the aforementioned experiment [61] when compared to VMWare ESX, XenServer, VMWare Workstation and Oracle VirtualBox. Alternatives to QEMU as a type-2* hypervisor are Google's crosvm[18] and its successor Firecracker [1][19] implemented by AWS. Crosvm and Firecracker are called microVM hypervisors. Their design goal is to provide strong isolation in multi-tenant environments and reduce the performance overhead VMs typically introduce when booting a guest OS and emulating several virtual devices like Universal Serial Bus (USB), Peripheral Component Interconnect (PCI) etc. Firecracker claims that they can boot application code

[13] https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/
[14] https://xenproject.org/
[15] https://www.vmware.com/products/workstation-pro.html
[16] https://www.virtualbox.org/
[17] https://www.linux-kvm.org/page/Main_Page
[18] https://chromium.googlesource.com/chromiumos/platform/crosvm/
[19] https://firecracker-microvm.github.io/

within 125 ms [1] which was empirical confirmed in an experiment comparing QEMU's microVM[20] with Firecracker [195]. The developers of AWS achieved this by implementing a custom Linux kernel with less system calls, removing emulation for several devices like USB and having pre-loaded execution environments (microVMs).

As already implied when introducing microVMs, OSs booted inside a VM can be full blown Linux, Windows or MacOS distributions with several running applications managed by the respective OSs, see Figure 2.1. Hypervisor studies revealed that booting VMs can take seconds up to minutes in the worst case [204, 228]. For scenarios where scalability and latency are critical requirements, such boot times are not acceptable. There are two approaches documented in literature and supported by industry to overcome this booting issue for hypervisor based virtualization: unikernels and tailored OSs.

Unikernels are "specialized, sealed, single-purpose lib[rary]OS VMs that run directly on the hypervisor" [164, p. 462]. This has several benefits for the compiled unikernel. Firstly, only OS features needed by the application are included in the final artifact during the build process. Secondly, since configuration parameters, hardware architecture etc. are known at compile time, the compiler can optimize the unikernel [164]. And finally, the underlying hypervisor abstraction isolates unikernels from other VMs or unikernels running on the same physical machine, so no security mechanisms have to be implemented by the unikernel itself. This enables direct execution of commands in user space, regardless of whether the invoked functionality is implemented by the Library Operating System (libOS) or the application. A dedicated kernel space is not necessary which makes context switches and system calls obsolete [126, 222]. In 2011, Porter and others [222] already predicted that cloud services will adopt unikernels due to their performance advantages over VMs but generic VM services like AWS EC2 will still. Since 2018, Firecracker supports user-created unikernels build upon the libOS OSv [1, 126][21]. Running only a single functionality is perfect for the unikernel approach, therefore, the developers of AWS Lambda and Firecracker decided to support it as one option in their public cloud offering. But there is one drawback. A reconfiguration of the application is only possible when recompiling the unikernel.

The second approach to deal with scalability and latency requirements are tailored OSs. Such approaches omit features like USB support similar to the corresponding hypervisor solutions which are not necessary for a compute offering in the cloud where a developer has no option to use USB devices. In this regard, also the number of syscalls can be reduced to a minimum as is the case of a recent research prototype: Lupine Linux [142]. Opposed to unikernels, they can execute a number of applications which comply to the tailored restrictions without recompilation.

[20] https://qemu.readthedocs.io/en/latest/system/i386/microvm.html
[21] https://osv.io/

2.1.3. Container Technology

When talking about containers, most (non-IT) people probably think about container ships. Therefore, the online dictionary definition of containers is not surprising: "a large metal or wooden box of a standard size in which goods are packed so that they can easily be lifted onto a ship, train, etc. to be transported"[22].

The question is now, how this definition is related to containers in computer science. The main aspect here is standardization. With shipping containers, arbitrary goods can be transported on any means of transport while inside the container packaging specific to the particular goods can be used. With OCI compliant containers, arbitrary applications can be run on any kind of system while inside the container the specific libraries required by the application are included. Compared to VMs, where each virtual instance has its own OS and is therefore self-contained and independent, containers are dependent on the host as shown in Figure 2.2.

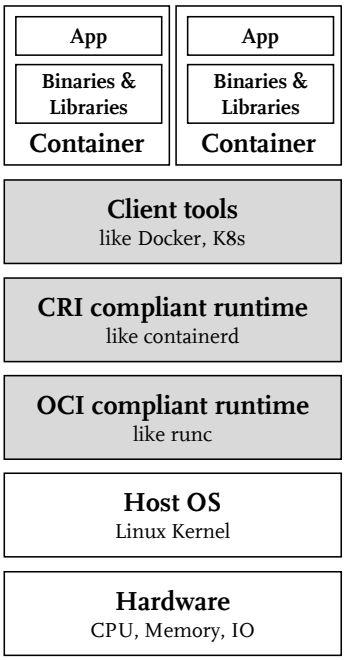


Figure 2.2.: Containers and their corresponding technology stack [25, 75].

Containers are not a virtualization technology since they have no "own operating environment"[9] for each container, but are often considered as an alternative to VMs due to their capabilities of isolating applications from each other. Several different projects contributed to this isolation security- and performance-wise by making them independent of each other [227]. In 2008, Linux Container (LXC) was the first project for creating and running containers [25]. The most important features are user namespaces, seccomp policies, kernel capabilities and control groups (cgroups):

[22] https://www.oxfordlearnersdictionaries.com/definition/english/container

namespaces are an abstraction used for isolating a number of processes from others. For example the root directory (`chroot` functionality) can be defined for each namespace. Each process assigned to this namespace can only access this part of the file system. Namespaces are hierarchically structured and can contain child namespaces with further restrictions [262].

seccomp stands for secure computing. It introduces system call filtering by limiting the process to assigned profile-based system calls. This reduces the possible attack surface for other processes running on the same kernel [49].

capabilities enable an unprivileged user process to execute some superuser functionality. Each process can be individually configured with a set of capabilities like binding a socket to a port[23].

cgroups limit the resource consumption of a process. The limits for CPU, memory, IO and network bandwidth are necessary when operating multi-tenant environments to prevent starvation of other processes running on the same machine [262].

These basic Linux kernel features and other features like AppArmor[24] or SELinux[25] are also used by other container management tools. One of these other management tools is Docker. At the beginning, Docker used LXC for creating images and executing containers but later extended LXC with two APIs for managing kernel and application features [25, 193]. The production ready version was released in June 2014[26] and revolutionized the way developers interact with containers. Docker uses the userspace components of an OS as a base for building images and a daemon process, the docker engine, for managing containers and other resources like networking, IO and storage[27]. This centralized management of basic compute features by a single daemon process is a big difference compared to the container management of LXC and improves portability of containers between different machines [200].

To improve the portability aspect across vendors of containerization software, Docker and other leading companies founded the Open Container Initiative (OCI) in 2015 under the umbrella of the Linux foundation. They published three specifications namely a runtime specification (runtime-spec), an image specification (image-spec) and a distribution specification (distribution-spec)[28]. When talking about OCI compliant images in this work, the compliance to the image-spec is meant. It defines how an image is constructed out of a layered file system and corresponding metadata. A corresponding project for building such compliant

[23] https://man7.org/linux/man-pages/man7/capabilities.7.html
[24] https://apparmor.net/
[25] http://selinuxproject.org/page/Main_Page
[26] https://docs.docker.com/engine/release-notes/prior-releases/#100-2014-06-09
[27] https://docs.docker.com/engine/
[28] https://opencontainers.org/about/overview/

images is *BuildKit*[29] which implements the OCI image-spec and is used by client tools like Docker. For starting and running containers, *runc* is widely used [75]. It is a low level tool for "spawning and running containers on Linux according to the OCI specification"[30] and uses the host kernel's aforementioned features like namespaces etc. to provide secure containers. *runc* is the reference implementation of the OCI runtime-spec [75]. It was donated by Docker in 2015 to OCI. Since *runc* is a low level tool, it is not recommended for end user usage[30]. Therefore, another software layer was introduced to manage the lifecycle of containers. Compliant tools like *containerd*[31] implement features like network support, pulling and pushing images to repositories for sharing them with others and further management features. *containerd* is open source, a Cloud-Native Computing Foundation (CNCF) graduated project and used by Docker per default. It delegates the creation and execution of containers to *runc*. With the rise of the container orchestration tool K8s [33], a third important interface was proposed: Container Runtime Interface (CRI). Container runtimes which implement this interface can be used by K8s and changed without recompiling the whole system[32]. The CRI consists of a gRPC API defining basic operations for sandbox and container operations delegated to tools like *containerd* and transitively *runc*.

2.1.4. Performance Considerations

VMs and containers enable multi-tenant environments where different actors deploy their software in a secure way via hypervisor and kernel features. These additional layers introduce performance overheads which are important to keep in mind for production use cases. This section examines different experiments to answer the question to which extent the discussed technologies influence performance considering VMs, containers and bare metal deployments.

Table 2.1.: Performance experiments comparing Bare Metal (BM), Virtual Machine (VM), and Container Technology (CT).

	Deployment Option			**Investigated Resource**		
	BM	VM	CT	CPU	MEM	IO
Arif et al. [4]	X	X		X	X	X
Debab & Hidouci [58]		X	X	X	X	X
Espe et al.[75]			X	X	X	X
Kozhirbayev & Sinnott [134]	X		X	X	X	X
Leitner and Cito [147]		X		X	X	X
Li et al.[151]	X	X	X	X	X	X
Morabito et al.[199]	X	X	X	X	X	X

[29]https://github.com/moby/buildkit
[30]https://github.com/opencontainers/runc
[31]https://github.com/containerd/containerd
[32]https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

Arif and others [4] compared bare metal and VM deployments based on CPU, memory and IO metrics. They used two web applications with an additional on-premise database for some experiments on identically constructed physical machines. Their results indicate that it isn't possible to predict the performance of VMs by applying a single scalar factor to the performance of bare metal machines. They showed that the level of overhead introduced depends on the executed functionality. The distribution of absolute values for the same metric is different on each deployment target for each application. Normalizing the values and correlating the metrics uncovers some trends and makes a comparison between the physical and virtual deployment option possible. One example the authors mentioned was the correlation of throughput to CPU utilization and IO operations. CPU and memory performance showed only minor discrepancies when comparing correlations between the physical and virtual environment whereas IO metrics were more varied. A prediction for CPU and memory behavior is therefore only possible for the same group of applications, e.g. CPU-bound functions, whereas IO operations are hardly predictable due to the emulation introduced by hypervisors. These results are in line with research on public IaaS providers [147]. Within the same instance category the performance metrics for memory and CPU only deviate negligibly when deployed on the same hardware. IO metrics, however, show up to 95% deviation when using the standard deviation as a measure to compare the different compute resources. Other publications come to the same conclusions for comparing bare metal with VM performance [151, 199]. Morabito and others[199] for example performed different tests for CPU, memory and IO on bare metal, KVM-based VMs, two container runtimes and the unikernel OS^v, also based on KVM. Their focus was particularly on CPU benchmarks. For a *noploop* benchmark, they showed that the unikernel solutions was the fastest with 2.249 ms, followed by bare metal (2.391 ms), Docker as container runtime (2.393 ms) and KVM-based VMs (2.397 ms). This means that the unikernel application is 6.6% faster compared to the VM. For another CPU benchmark, LINPACK, OS^v for example performed worst compared to the other alternatives, but the relative difference between the solutions was between 1% and 2%, which still allows to make good predictions.

While the findings were largely similar when comparing the containers to bare metal performance regarding CPU and memory, there was a difference in the decline in IO performance. In experiments [134, 151], IO throughput only decreases up to 11% in relation to the bare metal deployment which is an improvement compared to VM deployments. Multi-tenancy was not a concern of the mentioned studies. Furthermore, there are studies comparing the different options for OCI compliant runtimes like [58, 75]. Their focus is on comparing native container engines with microVM approaches like kata containers[33] and Firecracker. As can be seen from these studies, there is no separation between container and VM solutions in today's research. It is rather a continuum dependent on the use case and the security requirements.

[33] https://katacontainers.io/

2.1.5. Summary

VMs and container technology provide developers with an abstraction from physical machines. These technologies are the basis for multi-tenant environments on-premise and in the cloud. Based on their technology stacks, they offer different kinds of isolation with each VM being self-contained and booting its own guest OS. Containers, on the other hand, share the host and are therefore considered to be more vulnerable to one container escaping from its sandbox and corrupting other tenants running on the same host kernel.

We can conclude that performance of bare metal deployments are comparable to VM or container solutions when looking at the distribution. Especially CPU and memory metrics only show minor differences between the different deployment targets. Isolation, security and performance are often discussed in VM and container research. Since containers start up within milliseconds and VMs provide better isolation, tools like Firecracker combine the strengths of both options and offer developers to run their software containerized on top of microVMs. For a deployment solution abstracting the specific hardware, VMs and containers are standardized options which allow a configuration of resources depending only on the requirements and independent of the hardware used.

2.2. Benchmarking

2.2.1. Definition

In the traditional word sense, a benchmark is "a mark on a workbench used to compare the lengths of pieces so as to determine whether one was longer or shorter than desired"[34]. Another dictionary definition also focuses on a similar aspect in a more abstract way as it describes benchmarks as "something that can be measured and used as a standard that other things can be compared with"[35]. When extracting the key aspects out of these two generic definitions, a benchmark has three aspects: an object under investigation, a measure to describe it and the possibility to compare it with similar things.

In computer science, the process of benchmarking is about comparing Systems Under Test (SUTs). The most important parts, which should be defined upfront, are the measurement methodology and a reasoning why the *experimental design* is as specified. The selection of suitable *metrics* is important for the mentioned comparison as well as how these metrics are gathered. The last aspect is the *workload pattern* and how the SUT is stressed [105, 132]. If the workload is kept constant across experiments, differences in the results can be attributed to the SUTs only, therefore enabling a fair comparison of the SUTs. These three aspects comprise a benchmark in the computer science domain. As they influence each other, re-

[34] https://www.spec.org/spec/glossary/#benchmark
[35] https://www.oxfordlearnersdictionaries.com/definition/english/benchmark_1

searchers have to be aware of the inter-dependencies between these three elements in order for results to be scientifically usable.

2.2.2. Metrics

When designing a benchmark, the selection of meaningful metrics and their properties are important to improve the quality of the benchmark. Bermbach and others [23] interpret the IEEE standard 1061, *Standard for a Software Quality Metrics Methodology* [243], with regard to cloud services. Their work is based on a prior study by Kaner and Bond [115] in which they propose a multidimensional analysis of "direct metrics" to combine and correlate them. They state that such an analysis improves the expressiveness of conducted studies. According to the IEEE 1061 standard, a direct metric "does not depend upon a measure of any other attribute" [243, p.2]. An example for this would be the total number of requests made by a benchmark. Indirect or derived metrics are the combination of several direct metrics. An example would be throughput which is the number of requests per specific time period.

To validate and provide a basis for the interpretation of a series of metrics, the standard mentioned above defines general validity criteria: Correlation, Tracking, Monotonicity, Discriminative Power, Predictability and Reliablity. *Correlation* demands that there should be a "strong linear association" [243] between the dependent (D) and independent (I) metric. To show this association between metrics, researchers often use linear regression models in their work [50, 180, 185, 298]. Lilja [153] even stated that linearity is one requirement for a *useful* metric to enable "accurate and detailed comparisons". But this requirement introduces a limitation for the relation between two metrics. Others argue that there can be also a different kind of correlation [23]. *Tracking* means that a change in I results in a change in D within a reasonable time frame. A counter-example would be a case where the change of I affects D only after hours or days which makes a correlation of the two metrics challenging. *Consistency* [115, 243] respectively *Monotonicity* [23] describes the fact that D should increase monotonically when I increases. The standard [243] as well as the interpretations [23, 115] only refer to monotonically increasing metrics, however, monotonically decreasing scenarios are also possible. The amount of available disk space considering a benchmark with increasing disk writes over time would be an example for this. The previous examples already followed the *discriminative power* criterion which states that the metrics, D and I, should be chosen in a way that several measures of the same metric can be clearly contrasted to each other. An example, which is not discriminative, is to group measures of CPU utilization in categories like low, medium, high where two points categorized as low are not further distinguishable. The next aspect *Predictability* is a step ahead. Benchmarking is the process of observing and comparing similar SUTs with each other. If we are able to predict D based on I since we gained knowledge about the behavior by observing our SUT and using the historical data as input for

a prediction model, we are half way towards simulating a system. So *Predictability* already builds a bridge from benchmarking to simulation which will be discussed in the next section. The last criterion - *Reliablity* - summarizes the others: A metric is considered reliable when it confirms to these five criteria "for at least P% of the application of the metric" [243, p.12]. The value of P can be specified based on the use case and characteristic of the experiment.

2.2.3. Workload Pattern

The second integral part of a benchmark is the workload submitted to stress the SUT. There is a trade-off between controlled experiments with artificially created workloads and others which mimic real world use cases [23].

Artificial workloads have the advantage that an experimenter designs the workload in a way which allows to draw clear conclusions. For example, if the aim is to test the scalability of a system, two approaches are conceivable: For the first approach the benchmark could be constructed to send a number of concurrent requests to a single instance of an application at t_0 followed by a wait time until all requests have been answered and the application is idle again. Then the next bunch of requests can be sent at t_1 to understand the capability to handle concurrent requests, hot paths within the application and resource utilization on the selected deployment target. When testing for example Java use cases, a second and third bunch of requests is necessary to reach a steady state after all Just-in-Time (JIT) optimizations are performed. In a second scalability approach the multi-tenancy influence of several tenants deployed on the same system could be investigated by sending a single or multiple requests to every tenant. In both scenarios, a base case workload is needed where only a single request is sent at a time. This enables a researcher to investigate how scalability respectively multi-tenancy change the system behavior [80]. We conducted an experiment implementing the first approach: We concurrently sent a single request to every tenant deployed on a single-node open-source FaaS system and specified a sufficient wait time until the system was idle again before sending the next requests [180].

Trace-based workloads, on the other hand, are more representative [80] as the workload stems from real world examples like an anonymized data set from Twitter [43] or from the parallel workload archive [77]. Since collecting such traces in detail introduces overhead in production, these traces are rare and often only aggregated values like the throughput are present. A re-engineering based on these aggregated values is necessary to specify synthetic workloads based on real world traces [64]. The benefit of having synthetic workloads is the possibility to easily tune and scale them dependent on the SUT and the research questions [105].

Another classification of workloads is their arrival pattern. Schroeder and others differentiate between open, closed and partly-open workloads [244]. In closed workloads, the interaction between users and the SUT is as follows: A number of users send requests, the SUT processes them and responds to the users. They pro-

cess the response and send the next request to the SUT after some time. In such a setup, the maximum amount of concurrent requests possible is determined by the number of users within this closed system. It also means that the arrival pattern of requests is known upfront. Such a design was chosen when performing the aforementioned multi-tenancy experiment in order to draw clear conclusions for the specific setup [180]. Open workload designs, on the other hand, have an unpredictable number of users and therefore an a priory unknown amount of concurrently executed requests. Especially for schedulers, load balancers and scaling mechanisms are used for these kind of workloads. If the system is composed out of several services, an experiment designer should be aware of the scaling capabilities of the individual components. There might be situations in which one component scales perfectly fine and the next one in the pipeline becomes a bottleneck. Such a design would reveal weak spots in the overall system design but also renders some metrics, e.g. an end-to-end latency metric, useless.

The third category are partly-open workloads where new users arrive to request the SUT and users who already interacted with the SUT leave the system or make further requests. This workload category is the default for web and Online Transaction Processing (OLTP) applications [244]. A recent study about load generation tools lists characteristics and future challenges [53]. The authors classified them based on the three arrival patterns, closed, open and partly-open. Examples for such tools are httpperf[36] or JMeter[37] to name the most prominent ones.

2.2.4. Quality Criteria for Experimental Benchmark Design

To assess different SUTs and make benchmarks comparable to each other, it is important to have quality criteria in mind during the design phase, especially when thinking about the benchmark methodology. Different criteria were defined in literature [23, 80, 103, 279] and a selection is presented in the following. These criteria also influence the selection of suitable metrics and workloads to support the intended benchmark goals.

Relevant The most important criterion is the *relevancy* of the benchmark. At design time, researchers and practitioners need to define and note the "intended use of the benchmark" [279, p. 334] in order not to lose sight of the big picture. Two questions are helpful when considering the relevancy of the benchmark. Firstly, is there a target audience who is interested in studying the results of the benchmark? Secondly, are the results relevant for a longer period or outdated after the next release of the SUT?

Based on the scenario and use case of the specific benchmark there could also be situations where it is reasonable to conduct experiments for a small target audience where the results are only valid for a single release cycle. One example

[36] https://linux.die.net/man/1/httperf
[37] https://jmeter.apache.org/

for such a scenario is to perform stress testing during development and give developers immediate feedback about the SUT.

Representative Another quality aspect which directly influences the target audience is whether the benchmark and its methodology are *representative*. This is the case when SUTs are tested under reasonable conditions like the "use of hardware in a manner that is similar to consumer environments" [103, p. 22]. The workload pattern is also crucial for a representative experiment. Tackling the scalability property of a system cannot be done when using a single user workload with a single request at a time. The insights gained from a benchmark can be furthermore representative for a set of similar problems, respectively algorithms, if the SUT and its functionality is categorized like often done for CPU intensive workloads [23]. This is an important criterion when an experimenter wants to transfer the insights gained from one problem domain to a similar one. One example for such a scenario is the standard Transaction Processing Performance Council Benchmark C (TPC-C) [267]. Written down as specification, TPC-C is a typical OLTP use case for testing read and write performance of databases during transactions. The measures and also the measurement methodology are specified by the standard to guide users by their implementations. Therefore, the level of freedom in implementing the specification is limited and various alternatives for databases are comparable based on this standard in the OLTP domain. Such standardization efforts path the way to consolidated experiments being relevant and representative. The interested reader is also referred to the benchmarking efforts of Standard Performance Evaluation Corporation (SPEC)[38], another corporation where researchers and industry experts work on the topic of standardized, widely adopted benchmarks for different problem domains.

Repeatable The documentation of the measurement methodology and all necessary configuration parameters enables other researchers to interpret the results correctly. The most important information is the hardware used, the software stack and its configuration, the workload generator and pattern etc. This precise documentation is the foundation for *repeatable* experiments [279]. Literature studies reveal a huge discrepancy between this desired state and the reality of published experiments. Kuhlenkamp and Werner [137] performed an SLR in the FaaS domain and reported that only three out of 26 experiments were repeatable based on the information provided. Another investigation by Kalibera and Jones [114] revealed that a "majority" out of 122 papers were not repeatable based on the information documented by the study authors. These studies are in line with Lorena Barba's keynote in which she talked about "12 Ways to Fool the Masses with Irreproducible Results"[39] in 2021 [15]. She makes points that go beyond the execu-

[38] https://www.spec.org/

[39] Keynote at IEEE International Parallel and Distributed Processing Symposium, https://www.youtube.com/watch?v=R2-GuH-6VFU.

tion/repetition of an experiment and are focused on the processing of data and the resulting publication. However, they should also already be considered during design time in order to get the raw data in a processable format. Barba's first aspect is that some researchers do not publish raw data of their experiments or make it accessible only upon request. This statement is based on some surveys in which researchers tried to gather raw data from already published research. One study investigated 315 data projects with data being recoverable in only 26% of the cases [51]. Another study, where the journal they looked at had a policy to submit raw data together with the manuscript, revealed a slightly better return share of 44% out of 204 randomly selected studies [260]. Not publishing raw data has two effects on the repeatability. Diagrams, figures and statistical evaluation within the papers are used to extract raw data by other researchers. This procedure is cumbersome and error prone since oftentimes the scripts to generate these artifacts aren't published either, Lorena Barba's last point in the enumeration of fooling the masses. Therefore, it is often unclear if for example outliers were included in the plots, which correlation measure was used to compute the statistics etc. The second aspect especially hampers the interpretation when redoing the experiment on own hardware which typically diverges from the hardware used to perform the original experiment. When conducting reproducibility studies, the concrete value of specific metrics is not in focus but the same objective and conclusion from the experiment [279]. Published raw data helps to assess the difference and enables researchers to investigate whether these are introduced by a slightly different setup or whether the original published data has some flaws limiting repeatability.

Fair The last statement about comparing the original published raw data and the data gathered from a repetition of the experiments lead us to *fairness*, the next quality criterion for good benchmarks. One aspect of fair comparisons is to avoid specific optimizations of some tools where comparable alternatives have no corresponding counterpart. A typical example for such specific optimization is the implementation of special SQL statements in relational databases by the database vendors. Furthermore, Huppler mentioned the tendency of designers to over-optimize benchmarks for a target environment, like UNIX, which implicitly affects other environments negatively [103]. When performing a comparison between a Java in-memory database and a relational database [81], we found unfair elements in the original evaluation conducted by the vendor of the in-memory database. The execution time for several transactions were measured based on the server processing time by wrapping controller methods with a timer. We found three generalizable flaws. Firstly, fair benchmarks should be conducted by a neutral consortium which is not directly associated with a SUT but has the expertise to conduct the benchmark. Often this is not the case, when vendors publish data to substantiate their marketing claims as they have a high interest showing the strength of their tools by hiding drawbacks. Unbiased, objective organizations like research institutions or councils can overcome this issue and provide fair compar-

isons between comparable alternatives. Secondly, in the aforementioned experiment, the in-memory database is specialized in storing a Java object graph and is therefore language dependent. Since Java uses JIT compilation and compiles often executed methods at a later point in time, the first minutes of an experiment are the warm-up phase in which these functions are executed by an interpreter. Data generated in this phase should be discarded until reaching a steady state period, when all compilations are finished [103]. Thirdly, the selection of metrics determines the fairness. A realistic scenario for an OLTP use case is the interaction of a frontend application requesting data via a Representational State Transfer (REST) Application Programming Interfaces (APIs) from a backend. In this case, the metric which is important from a frontend perspective is the overall request-response time including the network latency, unmarshalling of sent data, preprocessing by middleware layers and finally processing of the request within a controller method. Selecting only the server processing time to compare two alternatives like in the initial in-memory database example, IO overhead of relational database queries are predominant and distort competition [81].

Trustworthy In the age of "alternative facts", *trustworthiness* is an important aspect, especially when publishing data and drawing conclusions. Two view points can be distinguished, an outside and an inside perspective on the benchmark and the SUT. Specifications or standards like the mentioned TPC-C benchmark are published by unbiased organizations. They already provide some kind of outside perspective due to their neutrality. Furthermore, they have been trained by people who have the authority and the knowledge to assess if an implementation of a specification is compliant with the standard. These external auditors [103] and the independent organizations improve the trustworthiness of benchmark designs. Besides these two outside factors, inside factors are for example testing efforts, input restrictions and cross-validation of metrics. Unit tests guarantee that the SUT produces correct results under different circumstances, like using different compiler settings, runtime parameters etc. Other testing efforts like architectural tests or penetration tests can reveal if the specification is implemented correctly at an early stage in the benchmark design. Another aspect is the set of valid input parameters or the workload. Valid entries limit the number of possible configurations which increases the validity of results. These aspects should be documented but also asserted to prevent benchmarks being executed with invalid values [103, 279]. For example, the resource limitations when starting a SUT has an impact on the execution behavior of the system. When considering a multi-threaded application, the minimum of available cores should be greater than one. A chosen resource limitation which is equivalent to less than a single core would prevent concurrent executions. The conclusions drawn from such misconfigured experiments are often error-prone and not trustworthy. A last aspect is the cross-validation of different metrics in the data post-processing phase of the benchmark when the experiment has already been conducted. Metrics can be independent of each other

or they can affect each other positively or negatively. These relations between metrics are oftentimes known upfront like the influence on system utilization when increasing the number of requests and support benchmark designers to make results plausible.

Economical A benchmark should be "worth the investment" [103, p. 28]. Huppler stated that this criterion is often not considered during the initial benchmark design since the technical consideration how to build the system, which workload to use, how many runs etc. define the boundaries. Nevertheless, being *economical* is intuitive since the trade-off between financial investments and information gain is already inherent in the criterion name. One example which is often considered in literature [103, 279] is the monetary expenditure conducting the already mentioned TPC-C benchmark. TPC-C results are ordered by the maximum throughput measured in transactions per minute for benchmark C - tpmC. At the time of writing this thesis, the first entry in the list[40] was submitted on 18th May, 2020 executed on Alibaba Cloud Elastic Compute Service Cluster[41] with a maximum of 707,351,007 tmpC. The total system cost was 2,814,509,552 Chinese Yuan Renminbi (CNY)[42]. This benchmark shows the capabilities of a specific system but is only reasonable for a limited number of companies in the market. Defining and documenting the intended goals associated with the benchmark helps in creating an environmental setup and choosing an appropriate workload which is affordable.

Usable The last aspect is how *usable* a benchmark is, which is the case if it can be deployed and used on comparable systems without major modifications and if the benchmark results are understandable. Portability in this dimension means to use generic features and avoid using special ones [23]. Examples are the many different SQL dialects and specialized routines for interacting with relational databases or the capabilities of one compute service of a cloud provider compared with another. Besides the same provided functionality, interfaces are normally different as shown in a portability study on the cloud function lifecycle [96]. We made some suggestions on how to overcome the heterogeneity of interfaces and harmonize them through wrappers making the benchmark also usable on other platforms without code modification. Understandability is another aspect of usability. The benchmark should be designed and documented in a way that an interested reader, who is familiar with the domain under test, is able to understand the design objectives and can interpret the results [23].

[40] https://www.tpc.org/tpcc/results/tpcc_results5.asp
[41] https://www.tpc.org/1803
[42] Considering the exchange rate of the 29th of November 2022, this is equivalent with 379,601,373 €

2.2.5. Distinction to Related Concepts

Benchmarking is one way to understand a SUT by treating it as a black box. It is characterized by making experiments with different types of workloads, using an appropriate measurement methodology and suitable metrics within a reasonable period as discussed in the prior sections. Profiling, tracing and monitoring are often used in conjunction with benchmarking, but the former two are more intrusive and collect more information for individual requests to unravel the black box and gain further insights. The latter, on the other hand, is a continuous process to observe the black box and selected metrics but is not limited to a specific experiment as benchmarking is.

Profiling is "the act of collecting useful information about [...] something so that you can give a description of [...] it"[43]. In computer science, profiling is the process of generating a profile of e.g. the resource consumption of an application. Tracing describes the process of following a single request until the end to understand the different control flows of an application and their impact on the overall performance. And finally, monitoring is the process "to watch and check something over a period of time"[44]. In the following, we present a selection of studies each of which has a focus on one of these three aspects but also includes elements of the others.

The *need for profiling* [283] is argued based on application management, resource considerations and the cost aspect of a SUT. *Application* management of a SUT means that the resources are configured properly to avoid performance degradation. *Resource* considerations should avoid an over-provisioning situation. And finally the *cost* perspective often balances the two prior aspects. Profilers for example introduce a lot of overhead by instrumenting code and exposing more events but improve the information gain [201]. This process should be considered when benchmarking a SUT is not sufficient and further insights are needed. One profiling approach for example uses additional information to generate test cases. The authors filtered faulty executions a posteriori and supported developers by troubleshooting their SUTs. The test cases served as a starting point for debugging as well as enriched the test suite for the individual application [183]. Cuomo and others [52] implemented some wrapper for often used components to derive more runtime metrics, generate profiles and use them when executing their simulations. Ren and others [229] presented the way how Google profiles their data centers from an infrastructure point of view. They provide metrics for physical machines onto which multiple tenants are deployed as well as enable application profiling. Each Google cloud service includes a custom library which exposes profiling information when requested.

Tracing approaches also introduce overhead since unique identifiers for every trace are processed in every component of the distributed system to assemble

[43] https://www.oxfordlearnersdictionaries.com/definition/english/profiling
[44] https://www.oxfordlearnersdictionaries.com/definition/english/monitor_2

the trace at a later point in time. With their instrumentation tool, Mace and others [162] enabled a recording of distributed application topologies. They introduced a happened-before join operator to allow the user to investigate traces across component or application boundaries. Another tracing approach was used when suggesting a way to construct integration test scenarios by modeling an application as a graph [286] and using this graph to understand the data flow of it [288]. Also the public cloud providers offer tracing services to understand the different paths within a complex application like AWS X-Ray[45] inspired by ray tracing.

In contrast to intrusive approaches, non-intrusive approaches focus on monitoring to observe the current system state. Monitoring is a permanent process which enables users to define thresholds for several metrics to get alerted when something unexpected happens. Pi and others [218] used the container API as a source to collect metrics for implementing a feedback control tool in a distributed environment. Docker as the de facto container standard also provides some metrics via its *docker stats* API[46].

2.2.6. Summary

Benchmarking a SUT is the basis for collecting data about the runtime behavior of an application under different circumstances. This process enables researchers and practitioners to make informed decisions and test their hypotheses. It is an integral part of this thesis since the collected test data builds the foundation for simulation discussed in the next chapter.

When talking about the building blocks of a good benchmark - metrics, workloads and benchmark designs - documentation of all properties and influential factors is a key aspect for high-quality experiments. Also publishing raw data, scripts for post-processing and a release of the SUT belongs to the documentation aspect. This additional effort is still not widely incentivised by scientific publishers but appreciated by other researchers interested in details not included in papers and one aspect of good scientific practice.

Besides the listed characteristics from literature, a good benchmark is further characterised by its focus, namely that only a single dimension is subject of investigation. When comparing two applications which, for example, provide the same business functionality, but are implemented in different languages, the hardware should be identical. Furthermore, contrasting goals exist when designing a benchmark like reducing the experiment duration to save money. This is reasonable, but the documentation about decisions made for several dimensions is important for comprehensible experiments.

[45] https://aws.amazon.com/xray/
[46] https://docs.docker.com/engine/reference/commandline/stats/

2.3. Simulation

2.3.1. Definition

A simulation is "a situation in which a particular set of conditions is created artificially in order to study or experience something that could exist in reality"[47]. This implies that the real world object, which should be simulated, is abstracted and the focus is on the important conditions respectively features of the object a user is interested in. The main benefit of such an approach is that due to this abstraction an engineer will get feedback quickly before manufacturing the product. Simulation experiments can be conducted within minutes or hours giving a first impression of the design and characteristics of the object under investigation. It is used in many cases like training, analysis and decision support [135]. A main concern is the accuracy of a simulation, in particular whether there is parity between the set of conditions and the features of the real object.

In computer science, simulations can be divided into two groups: software simulation and hardware simulation. A designer of a hardware simulation experiment builds a model of the real world object and uses this model for analysis in experiments [144]. Such models are used to understand processor or system designs in an early development phase without having to construct the products physically [249]. This branch of simulation approaches is out of scope of this work. On the other hand, software simulations are used to understand the runtime behavior of software deployed on different targets, for example on-premise compared to cloud deployments [180, 185].

2.3.2. Quality Criteria

Besides the general software engineering quality criteria listed in literature, like by Boehm and others [27] or Sommerville [253], simulation systems have further quality criteria and processes. Balci [7] proposed a complete lifecycle of simulation studies with several tasks. It starts from the real world problem via modeling and experiments and ends with the result discussion. Over the years, he expanded his considerations to include quality criteria, in particular verification and validation considerations [8]. Others identified 29 quality factors by performing an SLR about simulation modeling and its quality assessment [60]. Robinson concluded based on his terminology forming that criteria in literature are differently addressed, categorized and composited [234]. Based on these findings from literature, only two quality criteria - accuracy and credibility - are particularly distinctive for a simulation approach and will be discussed in the following.

Accurate The definition of simulation already reveals the problem of *accurate* experiments. The model which is built to simulate the real world object can only

[47] https://www.oxfordlearnersdictionaries.com/definition/english/simulation

ever be an approximation. Therefore, it is obvious that the level of detail with which the model is created is a trade-off between modeling and runtime costs of the simulation on the one hand and the accuracy of the simulation results on the other hand [144]. A precise documentation of the design decisions is important for a proper interpretation of the experiments. Descriptions of aspects which are neglected as they currently aren't relevant are just as important as those which are included in the model. Additionally, to be accurate, a simulation experiment has to document the validation process by e.g. specifying tests [7]. These self-validation steps give other researchers the opportunity to work on the simulation model and refine it while staying in the functional boundaries. Furthermore, these steps make the experiment reproducible by others. Similar to benchmarking studies, repeatability based on the provided information is also a problem in simulation research as Pawlikowski and others found that a "vast majority of simulation experiments" is not repeatable [212, p. 137].

Other terms used for describing aspects of accuracy in literature are fidelity [60], verification, testing [7] or validity [234]. All of these terms and their explanations have in common that they deal with a precise mapping of the real world object to the model.

Credible While for accuracy the focus is on a neutral and objective investigation, the question whether a simulation is *credible* is more subjective. The focus of simulations is to compare alternatives within an early development phase of a product or software system and decide - based on limited information - which approach is more favorable. Therefore, another target group besides software engineers are decision makers who decide which option to choose on the basis of simulation results [234]. Awareness for the assumptions needs to be communicated in an early phase and repeated when presenting the results so that decision makers perceive the simulation approach and therefore the simulation results as trustworthy [144]. As for the first quality criterion, a proper documentation of the assumptions made and the consensus of all stakeholders in the simulation process improves the credibility of the solution - also for future simulations. To guide decision makers, a proper visualization of the simulation results supports the assessment of different alternatives [60]. Additionally, a discussion of a validation process and a proper reasoning create trust [144]. So, credibility is a soft criterion and due to its subjective nature not easily quantifiable but crucial for the success of a simulation process and the conclusions drawn from it. Other terms used in literature which address a similar notion are acceptability [7] or judgment of the process quality [234].

2.3.3. Distinction to Related Concepts

The most closely related term to simulation is emulation[48]. In an industrial publication by Starner and Chessin [259], there is no clear differentiation between the two terms as the authors described emulation as a process based on the simulation model to test different alternatives, which is seen in academic literature as a simulation step [120]. The differentiation between both terms is important for an understanding of the capabilities of an emulator compared to a simulator. According to the Oxford Learner's Dictionary, emulation is "the act of a computer or computer program working in the same way as another computer or program and performing the same tasks"[49]. An emulator is therefore capable of replacing parts of the real world object or the whole one despite unchanged system functionality.

Simulation approaches can be a first step towards emulation solutions. Within the simulation process, the real object is abstracted via models for analyzing and studying the domain. Then an engineer is capable of building an emulator with the core features based on the simulation model. This procedure saves time and money by testing products before they are physically constructed [120, 191]. In an experiment for Covid-19 simulation models, Sürer and Plumlee used this approach. Their simulation model produced a lot of unusable results based on the input parameters. These parameter constellations were filtered to shrink the result set before building an emulator based on these insights [263].

2.3.4. Summary

Due to its advantage of producing early feedback with reduced cost, simulation is often used in engineering. Especially when having the two quality criteria - accuracy and credibility - in mind, experiments can support engineers and managers to decide between several system designs or architectures. It is important to indicate the level of accuracy needed for making a decision at the beginning of the project. This makes the requirements to the simulation process clear, helps to create a model with the desired quality and supports the analysis of the real world object in an early development stage.

[48] This observation is confirmed when searching on dblp for the search string "simulation emulation", which resulted in 164 entries on 13th of December 2022, https://dblp.uni-trier.de/search?q=simulation+emulation.

[49] https://www.oxfordlearnersdictionaries.com/definition/english/emulation

Part II.

Function as a Service

3. Conceptualization of Function as a Service

Parts of this chapter have been taken from [175, 180, 185].

In this chapter, RQ1.1 *(Which characteristics define a FaaS offering?)* and RQ1.2 *(How is FaaS related to other service models in the cloud computing landscape?)* are supported.

Having a common understanding and precise use of terminology is an important foundation to embed a new technology in an already present technology stack. Therefore, Section 3.1 defines *Function as a Service* and *Serverless Computing* based on characteristics extracted from literature to differentiate the two terms which are often used to describe similar concepts. FaaS is recognized as a new cloud computing model. Therefore, in Section 3.2 it is embedded in a "as a Service" classification based on the provider and user view on this new service model. The following sections include a market overview of current public cloud providers as well as open-source alternatives (Section 3.3) and strategies to assign resources to cloud function instances (Section 3.4). The conceptualization is concluded with a proposal for an architecture of a FaaS worker node in Section 3.5 and a short summary.

3.1. Differentiation of Serverless Computing and Function as a Service

3.1.1. Motivation

Serverless functions [9, 97, 108, 172, 264, 284], *cloud functions* [113, 203, 242, 257, 274], server-aware vs. server-less [9] and serverful computing [113] are terms to describe and define a new computing model which abstracts servers. In addition to this plethora of different terms, the interpretation of this new service model is diverse. Leitner and others [148] focus especially on the operation work a user has to do. They argue that a developer does not have to do any form of operations work anymore when using this new computing model. From this point of view, this new paradigm enhances DevOps to *NoOps* [148]. Another interpretation is that "server-less platforms can be considered an evolution of Platform as a Service"[42, p.47].

The "Serverless Predictions" [159] forecast a *disaggregation of computing resources* for the next decade but don't offer a proper terminology for this change. All of this indicates why Hellerstein and others [98] state on their first page that "the notion of serverless computing is vague enough to allow optimists to project any number of possible broad interpretations on what it might mean". When searching for definitions and literature about this new cloud computing concept this ambiguity is a real problem. For this reason, this part is dedicated to conceptualize FaaS and Serverless Computing to avoid further confusion.

Since Serverless respectively FaaS are considered as a new cloud service model, the NIST definition on cloud computing [192] is the first source for getting reliable information and check if FaaS is indeed a new cloud service model. All "as a Service" offerings defined by NIST have a common central aspect which is that a cloud provider offers as a Service for infrastructure (I), platforms (P) and software (S). FaaS is no exception but it also provides a platform for executing arbitrary user defined functions (F)[50]. It also attracts a lot of attention in industry. Gartner predicts that 50% of global companies will use a FaaS service in 2025[51]. It is also seen as the next big evolution in cloud computing in scientific literature [112, 113, 233, 242]. Despite the overall agreement that this new cloud computing model abstracts most operational tasks, the majority of publications use the term *Serverless Computing* (or only *Serverless*) while addressing core concepts and characteristics of FaaS. Most authors are aware that Serverless is an oxymoron [113] and misnomer [18, 73, 215, 289] but still use the term in their publications. Therefore, the following SLR based on the guidelines of Kitchenham and Charters [124] provides insights in a structured way to define both terms. The contribution to the scientific community is two-fold. The provided SLR is the first structured approach in literature, to the best of our knowledge, to define Serverless Computing and FaaS and distinguish the two terms based on a set of characteristics (RQ1.1). These characteristics can serve as a checklist for further publications to help decide which term to use. Furthermore, they give guidance if new services offered are either categorizable as FaaS or Serverless or none of the two. Secondly, cloud service offerings provide different levels of abstraction and control. As an outlook to this conceptualization perspective, we classify FaaS from a provider and user control perspective in Section 3.2. This can point users to decide which service offering to use based on their requirements and the level of technological freedom they need. When discussing the SLR results, numbers are presented to which extend authors use the term *Serverless* compared to *FaaS*. Furthermore, search trends at Google's search engine provide data and give hints why authors choose a misnomer instead of being more precise in their wording.

[50] It is especially noteworthy that the biggest scientific cloud computing conference, IEEE CLOUD, lists FaaS with the other three established service models in their call for papers as research areas, https://conferences.computer.org/cloud/2023/cfp/.

[51] https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing

3.1.2. Related Work

There are several other SLRs which are related to this chapter as they searched literature in a reproducible way in order to better understand the following areas of research concerning FaaS: The studies deal with benchmarking [137, 174], performance evaluations [240] and tooling in the FaaS domain [294]. In another systematic investigations, Taibi and others [264] examined 24 publications and found 32 patterns for building applications. Eismann and others [73] looked at 89 applications to decide when to use serverless offerings whereas Yussupov and others [296] chose a different dimension and considered the given requirements to find a suitable FaaS platform. They investigated the ten most prominent platforms at that time and compared them based on their characteristics. Despite the many different formulated questions and structured research approaches already available, there is currently no structured approach to identify and discuss literature for a conceptualization and definition of Serverless and FaaS.

3.1.3. First Definition Approaches

The first step of our criteria collection process is to look at existing definitions for Serverless and FaaS. For this initial approximation of the terms, we look at the glossary sections of the Cloud-Native Computing Foundation. CNCF was chosen based on its vendor neutrality as well as the focus on cloud technologies. It is, therefore, a valid source where interested researchers and practitioners may look for a definition[52]. However, these definitions are not peer-reviewed nor discussed within the scientific community and reflect the opinion of the corresponding authors at CNCF. Nevertheless, they are recognized as a starting point in our discussion to scientific sound definitions.

> "**Serverless** is a cloud native development model that allows developers to build and run applications without having to manage servers. There are still servers in serverless, but they are abstracted away from app development. A cloud provider handles the routine work of provisioning, maintaining, and scaling the server infrastructure. Developers can simply package their code in containers for deployment. Once deployed, serverless apps respond to demand and automatically scale up and down as needed. Serverless offerings from public cloud providers are usually metered on-demand through an event-driven execution model. As a result, when a serverless function is sitting idle, it doesn't cost anything."[53]
>
> "**Function as a Service** (FaaS) is a type of serverless cloud computing service that allows executing code in response to events without maintaining the

[52] The two definitions are archived on Zenodo in the bundle with all the other SLR information [177].

[53] https://glossary.cncf.io/serverless/

> complex infrastructure typically associated with building and launching microservices applications. With FaaS, users manage only functions and data while the cloud provider manages the application. This allows developers to get the functions they need without paying for services when code isn't running."[54]

Based on these definitions for Serverless and FaaS we can identify important technological evolutions, which serve as input for an SLR: The authors of the glossary at CNCF define FaaS as a subset of Serverless Computing. When we compare the two definitions, we recognize that both are quite similar. However, the characteristics of the *Serverless* CNCF definition are more abstract compared to the *Function as a Service* definition as it focuses on the obligations of providers. From a developer point of view, the focus is limited on services packaged as standardized units. When considering the benefits of VMs security-wise and containers performance-wise as explained in Section 2.1, users upload code and provider package this code in OCI compliant images and run them within their cloud platforms. These images could be a skeleton of self-contained functions but also arbitrary services like databases or queuing systems. Presented and discussed in the next section, these related concepts , i.e. microservices, virtualization and containers, are important to consider for a proper conceptualization. They help to narrow down the context of Serverless and FaaS and serve as another kind of input for our SLR.

3.1.4. Related Technologies

It is also worthwhile to look at the technologies and concepts which are related to FaaS in order to come up with a better definition. A first aspect is system architecture where microservices seem a natural fit for self-contained functions. A microservice implements and supports a dedicated business capability which is normally comprised of several functionalities coupled in a service [68, 202]. How big or small microservices should be is an ongoing debate [245, 265]. However, when services are designed to only provide a single functionality - as is the case with FaaS - then the size should be even smaller. To differentiate between microservices and these smaller services, Wolff introduced the term *nanoservice* [290].

To build applications out of microservices and nanoservices, orchestration is a central aspect which has to be considered to understand the scope of *Serverless* approaches. When using orchestration tools, developers focus on declarative descriptions of their application architecture. Such a declarative, Infrastructure as Code (IaC) based approach abstracts servers from the developer and is considered as *Serverless* [292]. Public cloud providers achieve orchestration and management of their infrastructure and therefore a *Serverless* developer experience via a combination of IaC tools like AWS Cloud Formation[55] and capabilities of their platforms

[54] https://glossary.cncf.io/function-as-a-service/
[55] https://aws.amazon.com/cloudformation/

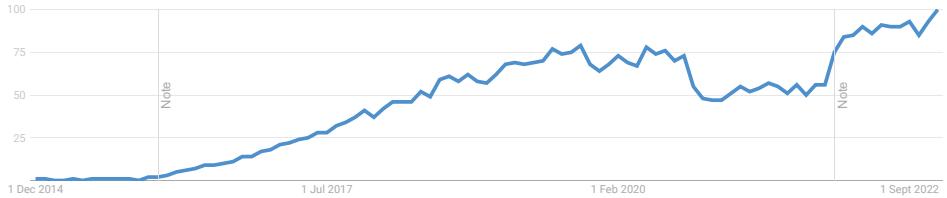
like offering auto-scaling features at Google App Engine[56]. When hosting open-source Serverless software like FaaS platforms, the job of a provider, which is often an in-house IT department, is to provide a notion of abstract computing resources for a developer. Due to the abstraction introduced with VMs and containers, the bare metal management of servers can be abstracted from a developer in an on-premise scenario but basic configuration options remain. Developers often have to specify runtime requirements to operate their implemented services, in particular how much memory and how many logical cores are necessary to operate the specific service. Proprietary solutions to provide a set of predefined VM configurations which developers can select from are one possible option for the in-house IT department but cumbersome to operate and error prone. VMs have stronger isolation guarantees and a better isolation of multiple tenants on the same host. However, to deploy and run nanoservices, the runtime characteristics where startup duration can take seconds for a single VM are not appropriate. Hence, as seen in Section 2.1.4, the runtime characteristics of container are more suitable for deploying such services but come along with a weaker isolation of tenants.

With the rise of K8s, an open source container orchestrator preview-released in 2014 and officially released in 2015[57], containers have become the default level of abstraction. Operation and management aspects are handled by the K8s orchestrator which comply with the provider tasks in the CNCF *Serverless* definition, for example by implementing a self-healing property of services or providing cluster management features like adding new nodes. IaC features are provided by the declarative K8s deployments and service configurations. Some authors are of the opinion that the *Serverless* movement is tightly related to the rise of K8s [25, 31, 296]. Therefore, it is important to consider that K8s features might influence the existing definitions of *Serverless*. It also explains why K8s is used as one of the search terms in a quantitative assessment of search trends in the next section.

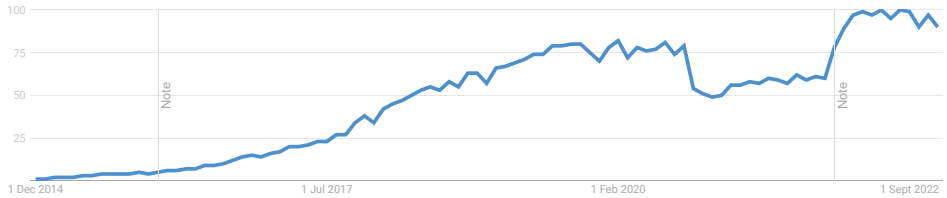
3.1.5. Search Trends at Google's Search Engine

To get an idea about the general usage of the terms Serverless and FaaS in articles, blog posts and the internet in general, this paragraph examines different search terms and their relevance over time as recorded by GoogleTrends [46]. It gives a first hint on the frequently used terms at Google's search engine and can be an indication why some terms are used more often in literature compared to others. The results for the Google Search Trends[58] query for *Serverless*, *Kubernetes*, *Function as a Service* and *AWS Lambda* can be found in Figure 3.1. The terms Serverless and Function as a Service were chosen to see their dissemination and understand the term usage in the web in general. K8s is identified as an important tool for orchestration where the K8s orchestrator is an interface for the user which abstracts

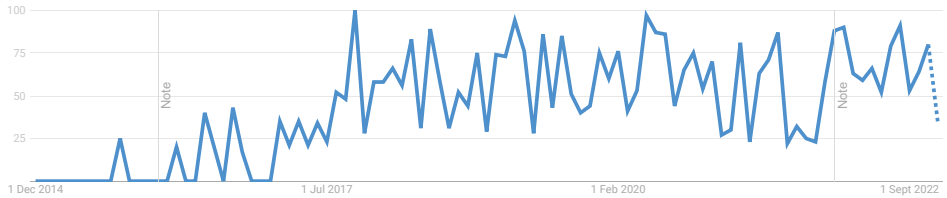
[56] https://cloud.google.com/appengine/docs/standard
[57] https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/
[58] https://trends.google.com/trends/explore?date=2014-12-20%202022-12-20



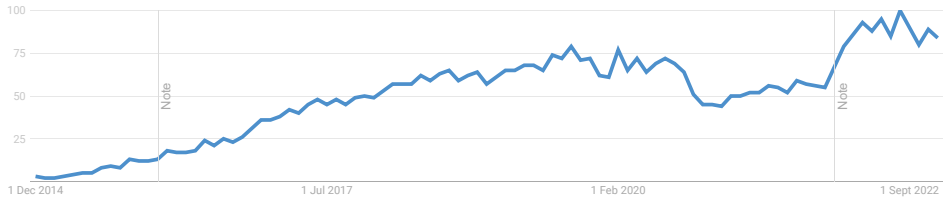
(a) Severless



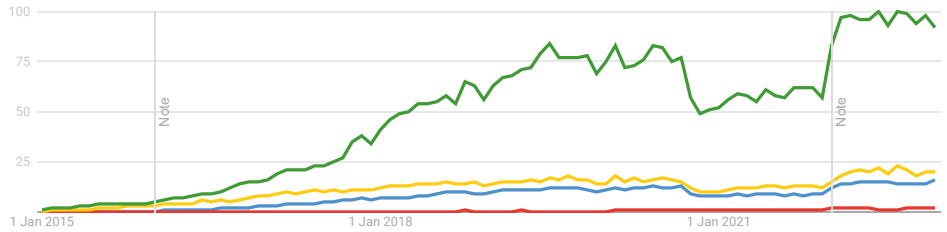
(b) Kubernetes



(c) Function as a Service



(d) AWS Lambda



(e) Kubernetes (green), AWS Lambda (yellow), Serverless (blue) and Function as a Service (red)

Figure 3.1.: Google Search Trends for the keywords *Serverless*, *Kubernetes*, *Function as a Service* and *AWS Lambda* from December 2014 until December 2022 as well as their relative interest to each other.

a lot of operational aspects. And finally, AWS Lambda as the first commercial FaaS platform created a milestone and also influenced the way practioners and researchers see the Serverless and FaaS domain. The graphics from Figure 3.1 cover the time span from December 2014 to December 2022. The vertical lines labeled "Note" are only a reference that improvements to the data collection system were

conducted in 2016 and 2022, respectively. This has no further impact on the interpretation of the charts. All lines are normalized based on the maximum of search interest for each search term individually as can be seen in subfigures (a)-(d). This maximum has the value 100 and other values are direct proportional to this maximum. Subfigure (e) puts all search terms in relation to show their comparative relevance.

In the previous section, we discussed the obligations of a computing platform provider as indicated by the CNCF *Serverless* definition and identified K8s as an important tool for building Serverless applications. Therefore, we start our Google-Trends analysis with a comparison of *Serverless* (subfigure (a)) and *Kubernetes* (subfigure (b)). It can be seen that both terms gain importance over time and their search trends show a similar distribution. Both terms indicate a *kind of operational abstraction*. Furthermore, both terms are considered important within the IaC and DevOps community [93, 252]. When comparing the relative frequency of these terms used in queries in subfigure (e), Kubernetes as a search term was used four times more often than Serverless. The reasons could be its practical nature and the configurations problems developers experiencing when they implement their applications. In addition, K8s is used by many companies as their container orchestration platform which explains its relative importance compared to Serverless.

As indicated by the CNCF definition, FaaS is seen as a subset of Serverless technologies, so the next term we want to consider is *Function as a Service*. In subfigure (c), we see a curve with a wide amplitude without a clear upward or downward trend. Based on this and the Serverless trend curve, a take-away could be that the two terms are not considered together. Taking into account subfigure (e) as well, the relative importance of Function as a Service compared with Serverless or even K8s is negligible. So an interim conclusion here is, that interested users looking for FaaS services and characteristics, use other search terms.

In 2014, the same year a preview version of K8s was released, the most important technological development of cloud service offerings with regards to FaaS was made by AWS, when they announced a new service called *Lambda*[59] at their re:invent developer conference. It was first available to the public in April 2015[60]. In their release note, they described their new service with the following attributes: event-driven execution model, provide management of compute resources, startup within milliseconds, billing per request as long as the function is running, auto-scaling on demand, metering in millisecond granularity[59]. Since AWS Lambda was the first available public service in the market which is in line with the CNCF definition of FaaS by being more concrete about the technical realization and the features a user can expect, it is reasonable to look at the search term trend for *AWS Lambda* in subfigure 3.1(d). The trend is comparable with the curve of Serverless but starting earlier with the end of 2014 which is after the re:Invent

[59]https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/
[60]https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html

presentation of Lambda in November. The relative importance in Figure 3.1(e) between AWS Lambda and Serverless trends is striking. Since FaaS is seen as a subset of Serverless based on CNCF definitions and AWS Lambda is a commercial FaaS platform, both terms describe the same kind of technology. This could be why the two terms are tightly related to each other in GoogleTrends.

3.1.6. Structured Literature Review

3.1.6.1. Identified Characteristics

For a structured scientific assessment of Serverless and FaaS, we need a solid foundation for differentiating and defining the terms. We choose a characteristics based approach, which means that we look at several terms, existing definitions and established standards to extract distinctive features. The previous section dealt with the general usage of terms in the web. Together with the definitions of the CNCF and related technologies like microservice, K8s and AWS Lambda in mind, we now have a set of relevant terms for a scientific, more structured assessment of Serverless and FaaS. As already mentioned in the motivation, another important source for general service model characteristics in the cloud is the NIST definition of cloud computing [192, p. 2]. They state five essential characteristics which form a cloud computing model: On-demand self-service, broad network access, resource pooling, rapid elasticity and measured service.

On-demand self-service means that a user of the service can "unilaterally" provision as many resources as they need and access them via standardized protocols over a *network*. By doing so, a provider serves multiple clients on a shared fleet of resources and *pools its resources*. This pooling leads to multi-tenancy scenarios where providers have to offer isolated units on a single physical machine to execute the code of each user separately. Deployment of services is fully managed by the provider and a user can only make choices on a "higher level of abstraction" like a geographical region or datacenter. However, they cannot configure on which VM, physical machine or rack an application should run. *Rapid elasticity* describes scaling up and down on demand which can give the user the illusion of infinite resources on the platform. And the last NIST characteristic describes an implementation of a *measured service* which is often realized by metering the user based on a pay-per-use strategy like hours for VMs or occupied memory for storage services.

To embed the CNCF definitions in a larger context and relate the NIST characteristics to it, in the following we will perform a SLR to find a solid body of literature to define both terms. Table 3.1 presents the extracted characteristics of the three definitions ("Serverless" and "FaaS" from CNCF and "Cloud Computing" from NIST) and shows their overlap in some of the characteristics. The two NIST characteristics *on-demand self-service* and *rapid elasticity* were combined with the *scaling* property of the Serverless CNCF definition. *Broad network access* was

Table 3.1.: Characteristics included in Cloud Native Computing Foundation definitions for FaaS and Serverless as well as the essential characteristics of cloud computing defined by NIST.

	Characteristic	Included In Definition		
		Serverless	FaaS	NIST
c1	Abstracted server management by a platform provider	X	X	
c2	Self-contained services complying to a standard, e.g. OCI compliant images	X		
c3	Scaling, on-demand self-service, rapid elasticity	X		X
c4	On-demand metering, pay-as-you-go	X		X
c5	Event-driven execution model	X	X	
c6	No idling cost	X	X	
c7	Resource pooling, multi-tenancy			X

not considered as a distinctive characteristic since all platforms are accessible via standardized protocols like HTTP.

3.1.6.2. Search Process

SLRs make research reproducible which is a quality aimed for in all research efforts within this dissertation project. Furthermore, an aim of this review is to find other characteristics to define the term FaaS and differentiate it from Serverless. One lesson learned from conducting SLRs is to justify the search strategy [30]. To get a comprehensive set of literature about Serverless and FaaS, we use the following generic search string at Google Scholar[61], ACM digital library[62] and IEEE Xplore[63]:

```
(Function as a Service OR FaaS OR Serverless) AND definition
```

The search was performed on 11th of January 2023. We adapted the search string based on the respective interfaces of the three search engines. Furthermore, we developed a review protocol to document the decisions made along the way[64]. K8s was not included in the search string since the focus is on a differentiation between FaaS and Serverless as well as to understand the term usage in the research community. Nevertheless, one question of the protocol approaches the importance of K8s within the identified publications in order to give credit to its widespread use.

When approaching the first engine, Google Scholar, we got about 24,300 results for our search combination, which can be explained based on the broad scope of the terms as well as the relevancy of the topic. To get a manageable set of publications, we follow an approach also used in other SLRs [130, 264] to include the first 100 entries sorted by relevance. The same quantity problem is present at ACM

[61] https://scholar.google.de/
[62] https://dl.acm.org/search/advanced
[63] https://dl.acm.org/search/advanced
[64] The review protocol as well as the results after each step in the SLR process are available at Zenodo [177].

digital library when performing a full-text search as more than 4,000 results were returned. In contrast to Google Scholar, the digital libraries of ACM and IEEE are capable of searching in the abstract as well as in other parts of the publications. Since the abstract of a paper is a short summary of the presented work which usually contains all relevant keywords, it was chosen as the search space for the first part of the phrase to find articles focused on FaaS and Serverless. The term *definition* could be included anywhere in the document since limiting this term only to the abstract would have resulted in merely 5 publications within ACM Digital Library and 8 results from IEEE Xplore.

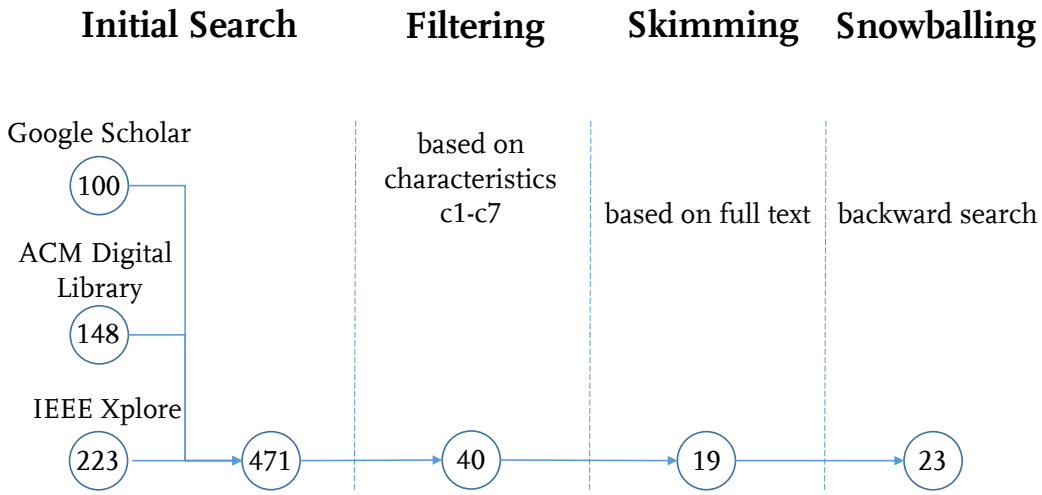


Figure 3.2.: Structured Literature Review conducted on 11th of January 2023 for differentiating FaaS and Serverless.

Figure 3.2 shows the SLR process and indicates that 471 entries were gathered from the three libraries. After filtering the entries based on the title with the characteristics c1-c7 from Table 3.1 in mind, 40 publications were left for a more detailed assessment based on their full texts. Skimming the papers was problematic since a lot of publications mentioned characteristics in several sections. So, we read the full texts of these papers and got a first set of relevant literature. The inclusion criteria was the presence of solid parts or sections where the authors describe Serverless or FaaS or both terms. Only a few papers had proper definition sections where the terms are defined, e.g. [9, 42, 113, 131, 133, 152, 233, 246, 274, 291], but also these papers name some of the characteristics in other sections as well. 21 papers were excluded from the further process as they either had a different focus like patterns [295] or were follow-up publications where the same authors published several papers with the same point of view on the terms, e.g. van Eyk and others in 2017 [274] followed by two publications in 2018 [275] and 2019 [276]. Only the first published papers of these authors were included. The last step was a backward search to find other relevant papers. Eight further papers were identified; four of them relevant based on the questionnaire. In the end of the SLR process, we ended up with 23 publications as listed in Table 3.2.

3.1.6.3. Discussion

Table 3.2.: Summary of Structured Literature Review publications for term definitions of Serverless (S) and FaaS (F).

Paper	Year	Terms Used		Differentiation/Usage	Characteristics
		S	F		
Gannon et al. [84]	2017	XX			c1-c3,c5
Lynn et al. [161]	2017	XX	X	Used as synonyms.	c1-c8
Baldini et al. [9]	2017	XX	X	FaaS one version of Serverless. Term FaaS only used once.	c1-c6,c8
Roberts & Chapin [233]	2017	X	X	Serverless comprises BaaS & FaaS.	c1-c6,c8
Van Eyk et al. [274]	2017	X	XX	Serverless comprises BaaS & FaaS as well as parts of PaaS and SaaS.	c1,c3-c5,c7-c8
Wang et al. [282]	2018	XX	X	Used as synonyms. Term FaaS only used once.	c1-c4,c6-c8
Castro et al. [42]	2019	XX	X	FaaS as the dominant use case of Serverless. Used as synonyms.	c1-c6,c8
Jonas et al. [113]	2019	XX	X	Serverless comprises BaaS & FaaS.	c1-c8
Jangda et al. [108]	2019	XX	X	Used as synonyms.	c1,c3,c5-c6,c8
Leitner et al. [148]	2019	X	XX	FaaS "most prominent implementation" of Serverless.	c1-c8
Hellerstein et al. [98]	2019	X	X	FaaS "core of Serverless offerings".	c1,c3-c5,c7-c8
Wu et al. [291]	2020	XX	X	Used as synonyms.	c1-c8
Koschel et al. [131]	2021	XX	X	Serverless comprises BaaS & FaaS.	c1-c5,c7-c8
Denninnart & Salehi [59]	2021	XX	X	Serverless comprises BaaS & FaaS.	c1-c8
Hassan et al. [97]	2021	XX	X	Serverless comprises BaaS & FaaS.	c1-c8
Kounev et al. [133]	2021	XX	X	Serverless comprises BaaS & FaaS as well as parts of PaaS and SaaS.	c1-c8
Schleier-Smith et al. [242]	2021	XX	X	Serverless comprises BaaS & FaaS.	c1,c3-c6,c8
Marin et al. [188]	2022	XX	X	Serverless comprises BaaS & FaaS.	c1-c5,c8
Ngo et al. [203]	2022	(X)	XX	Used Serverless only once as keyword.	c1-c3,c5
Li et al. [152]	2022	XX	X	Serverless comprises BaaS & FaaS, stuck to definition of Jonas et al. [113].	c1-c8
Shafiei et al. [246]	2022	XX	X	Serverless as a "generalization" of BaaS & FaaS.	c1-c8
Wen et al. [284]	2022	XX	X	Serverless comprises BaaS & FaaS.	c1-c5,c8
Mampage et al. [172]	2022	XX	X	Serverless comprises BaaS & FaaS.	c1-c8

After reading the first papers, *statelessness* was another term which was often used to describe Serverless respectively FaaS approaches. Therefore, it was included in the literature review process as the eighth characteristic (**c8**). Table 3.2 lists the authors, publication year and the term usage for every paper. For the term usage, XX indicates the predominantly used term (Serverless (S) or FaaS (F)) and X if the other term was also used. This term usage is especially important in order to form a definition as it indicates whether the papers distinguish between the two terms or if they use them as synonyms. There are only two papers of our set which name only one of the two terms. Gannon and others [84] only used the term Serverless in their paper where they discussed cloud-native applications and see Serverless Computing as a natural fit for such applications. Due to the year of the publication in 2017 and the focus on their paper, this term usage is reasonable. Ngo and others [203] on the other hand used Serverless only once as a keyword. Their focus is on FaaS platforms and their scalability. A reason for using still Serverless as a keyword could be that the authors are aware that other

researchers often use Serverless as a search term despite looking for publications about FaaS. All other papers used both terms.

Table 3.3.: Characteristics (c1-c8) and their occurrences in the Structured Literature Review papers.

	Characteristic	Included In Publications	Percentage
c1	Abstracted server management by a platform provider	[9, 42, 59, 84, 97, 98, 108, 113, 131, 133, 148, 152, 161, 172, 188, 203, 233, 242, 246, 274, 282, 284, 291]	100 %
c2	Self-contained services complying to a standard, e.g. OCI compliant images.	[9, 42, 59, 84, 97, 113, 131, 133, 148, 152, 161, 172, 188, 203, 233, 246, 282, 284, 291]	83 %
c3	Scaling, on-demand self-service, rapid elasticity	[9, 42, 59, 84, 97, 98, 108, 113, 131, 133, 148, 152, 161, 172, 188, 203, 233, 242, 246, 274, 282, 284, 291]	100 %
c4	On-demand metering, pay-as-you-go	[9, 42, 59, 97, 98, 113, 131, 133, 148, 152, 161, 172, 188, 233, 242, 246, 274, 282, 284, 291]	87 %
c5	Event-driven execution model	[9, 42, 59, 84, 97, 98, 108, 113, 131, 133, 148, 152, 161, 172, 188, 203, 233, 242, 246, 274, 284, 291]	96 %
c6	No idling cost	[9, 42, 59, 97, 108, 113, 133, 148, 152, 161, 172, 233, 242, 246, 282, 291]	70 %
c7	Resource pooling, multi-tenancy	[59, 97, 98, 113, 131, 133, 148, 152, 161, 172, 246, 274, 282, 291]	61 %
c8	Statelessness	[9, 42, 59, 97, 98, 108, 113, 131, 133, 148, 152, 161, 172, 188, 233, 242, 246, 274, 282, 284, 291]	91 %

Table 3.3 shows the characteristics and the list of papers where these characteristics were included, an inverted view on the information of Table 3.2. All characteristics are included in the majority of the papers and abstracted server management (c1) and scaling on-demand (c3) in particular are named in every included paper. Also most of the other characteristics like self-contained artifacts (c2), pay-as-you-go (c4), event-driven execution model (c5) and statelessness (c8) are used to describe Serverless and FaaS in more than 83 % of the papers. The other two aspects are less common but nevertheless important for a conceptualization as well as to stress platform design prerequisites. No idling cost (c6) is explicitly included as a characteristic since FaaS platforms improved the pay-as-you-go dimension in such a way that only the running instances are charged. This might explain why only 70 % of the selected authors named this as a distinctive aspect. If the provider optimizes and keeps functions alive after a first invocation, the user is not charged for this effort but profits from already provisioned instances without cold start effects like starting a JVM or loading dependencies which results in general in a shorter billing duration. The last characteristics in our list is resource pooling respectively multi-tenancy (c7) which is an inherent characteristic in cloud computing for achieving higher utilization for a fleet of servers. This aspect has performance implications for building cloud platforms in general and FaaS platforms in particular. See chapter 2.1 for a discussion about virtualization options.

Before discussing the SLR and insights from the current view on Serverless and FaaS offerings in detail, definitions are provided based on the introduced characteristics and the structured review of literature in mind.

Definition 3.1 (Serverless Computing)
Serverless Computing is a generic computing approach where servers are abstracted from the user. All operational aspects like managing physical machines and providing VMs are tasks of the service provider. A service provider is typically a public cloud provider but also in-house IT departments can act as service providers for their own development teams. They offer interfaces for service users like SDKs, CLIs or OCI compliant runtimes where the focus is on code, configuration options or deploying/uploading artifacts. All offerings are pay-per-use. Scaling and provisioning of additional computing, storage or network resources happens without user interference based on the demand of the deployed applications and ecosystem services like data storage. A provider of a serverless platform pools resources and serves multiple users on the same physical machine. Tenants are isolated via virtualization approaches in particular VMs.

The above definition of *Serverless Computing* contains the characteristics c1-c4 and c7. Some authors use Serverless and FaaS as synonyms [42, 108, 161, 282, 291]. As indicated by the following definition of FaaS, we disagree with this view and argue that some characteristics describing FaaS are unique to this service model as well as some characteristics are too specific for a generic computing principle like Serverless.

Definition 3.2 (Function as a Service)
Function as a Service (FaaS) is a compute service model in line with other as a Service offerings. The core of this concept are event-triggered, single-scoped functions, i.e. cloud functions, which scale up and down on demand without user interaction. This auto-scaling property is one of the unique characteristics of FaaS. Such platforms conceptually start a single function instance for each request and tear this instance down again after the function is executed. Therefore, cloud functions are inherently stateless. Which means that platform providers are able to offer a genuine pay-as-you-go billing model where users are only charged when the cloud function is running as instances scaled to zero results in no idling costs. Other comparable computing models like PaaS based offerings always have at least a single instance running to serve requests.

We see c3, c5-c6 and c8 as unique aspects for defining FaaS. Scaling is already included in the Serverless definition but a key aspect why so many practitioners and researchers talk and write about FaaS and therefore stressed again. From a conceptual point of view, a single function is started upon each request and teared down afterwards which renders a FaaS offering stateless. This focus on stateless cloud functions eases scalability and pay-as-you-go but comes along with an increased number of services (nanoservice vs. microservice) as well as additional ef-

fort needed to get a FaaS offering in production. Storage, routing and messaging are only a few examples of backend services needed to build FaaS applications. These services are subsumed under the term Backend as a Service (BaaS) and often characterized Serverless since the service provider manages the servers (c1), scales the resources on-demand (c3) and meters a user on demand for the used capacity (c4), e.g. API requests, storage etc. Combined with the multi-tenancy (c7) and self-containedness of these services (c2), all characteristics of our Serverless definition are fulfilled. Since the number of BaaS offerings needed to get FaaS in production is higher than e.g. in a PaaS scenario, some authors coin this stronger kind of vendor lock-in as ecosystem lock-in [28, 96, 127].

Furthermore, when approaching this new area of computing from an application point of view, it is not surprisingly that most of the authors [59, 97, 113, 131, 133, 152, 172, 188, 233, 242, 246, 274, 284] see Serverless as an umbrella term for BaaS and FaaS as can be seen in Table 3.2. All papers from our SLR published in 2021 and 2022 except Ngo and others [203] share this view on Serverless, BaaS and FaaS whereas only three [113, 233, 274] out of twelve papers did so in the period from 2017 to 2020. Another insight from the literature review is that most authors write about FaaS in the sense of the presented definition but use the term Serverless in their publications in the majority of occurrences [9, 59, 97, 98, 108, 152, 161, 172, 282, 284, 291]. Wang and others [282] for example used an ambiguous wording as their GitHub repository is called "faas_measure" but the term Serverless is most often used in the paper. This term usage is mirrored in the search trends where users most often use Serverless in their query. Additionally, early authors in 2017 most often used the term Serverless which resulted in a self-reinforcing effect.

From the search trends presented in Section 3.1.5, two terms have been missing from this discussion so far: K8s and AWS Lambda. K8s is included in nine of the 23 papers, so there is a relation between Serverless and FaaS on the one hand and K8s on the other. But this relation is not as strong which is indicated by the search term relevancy over time for the terms in Figure 3.1. K8s is seen as a kind of orchestration layer [161] or used for achieving security aspects [97] but most of these papers argue that K8s is used as a foundation for open source FaaS platforms [233] and present data which open source platforms are using K8s [59, 152, 172]. Other research conducted which is not included in the selected papers confirm this view on K8s and open source projects [149, 180, 197, 207, 276]. This seems reasonable when looking at its features[65] like auto scaling and declarative deployments. To summarize, K8s is an important technology in the Serverless domain, especially as a foundation of open source FaaS platforms.

The other search term which attracted a lot of attention with regard to Serverless is AWS Lambda. It is included in all the selected papers from the SLR despite two of them not naming AWS Lambda in their full texts [133, 188]. As a pioneering platform in the FaaS domain also other studies revealed this important position.

[65] https://kubernetes.io/docs/concepts/overview/

A glimpse of this dominance in FaaS research is an SLR conducted by Hassan and others where 78 of their selected papers use AWS Lambda for experiments (2nd place Apache OpenWhisk included in 23 papers) [97]. A more general study showed that 86% of survey respondents had experience with the cloud provider AWS in general (2nd place Azure reached 31%) [148]. To summarize, the unique features of AWS Lambda released in 2015 contributed to the ongoing Serverless hype. The tightly related curves in Figure 3.1 which are showing a high connection of the terms and a growing relevance over time strengthen this statement. Since most of the papers name this platform and also conduct research with it, AWS Lambda influenced the point of view on Serverless and FaaS offerings.

3.1.7. Conclusion

Even though FaaS complies with the Serverless definition and can be seen as a kind of Serverless Computing, a more precise use of terminology is needed in the cloud computing research area to properly distinguish between both terms. The presented SLR provides a foundation to this and also incorporates related terms like nanoservices and associated technologies like IaC, K8s and AWS Lambda to differentiate Serverless and FaaS. The usage of search terms indicates that users usually do not use FaaS. Instead other search terms like Serverless and AWS Lambda have higher relevance. The provided definitions for Serverless Computing and FaaS as well as the classification of BaaS in this section are a contribution towards a more precise use of terminology. Based on the structured approach and the gathered characteristics, RQ1.1 *which characteristics define a FaaS offering* can now be seen as answered.

3.2. Differentiation to Established Cloud Service Models

As already acknowledged in the last section, FaaS and BaaS are two types of Serverless offerings. The question which now arises is RQ1.2: *How is FaaS related to other service models in the cloud computing landscape?* In 2016, Adrian Cockcroft, a former vice president at AWS, wrote on Twitter: "If your PaaS can efficiently start instances in 20 ms that run for half a second, then call it serverless."[66]. We know from the definitions provided that we would call it FaaS, but there are also PaaS offerings like Google App Engine[56] where a user does not have to care about servers and their configurations. This results in a serverless experience based on the introduced characteristics from Table 3.3. To answer RQ1.2 we look at the provided data from the SLR again and discuss it from a provider and user point of view. We look at the three established service models IaaS, PaaS and SaaS as well as on FaaS and BaaS. Since cloud functions are often packaged as OCI compliant images, we further incorporate Container as a Service (CaaS) in our discussion and present a

[66] https://twitter.com/adrianco/status/736553530689998848

spectrum of cloud service models as can be seen in Figure 3.3. Note that the axis labels are different from low to high on the y-axis showing the provider control over the software stack and from high to low on the x-axis for the user control.

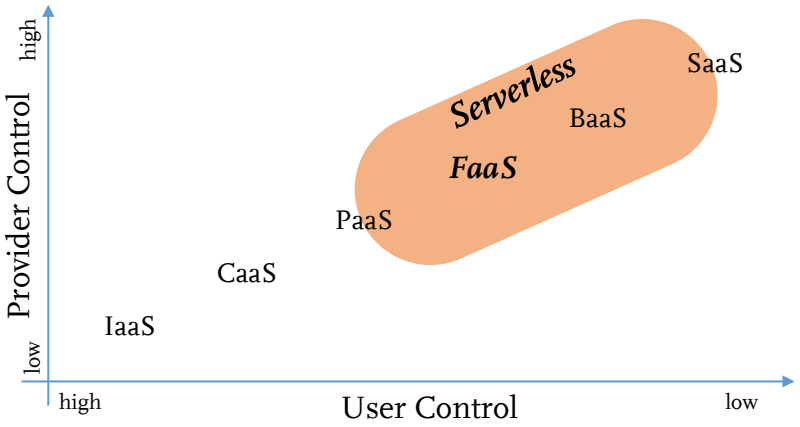


Figure 3.3.: Classification of FaaS and Serverless and relation to other as a Service offerings based on the user and provider control [130, 133, 274].

Since a user of a CaaS offering has to care about updating and patching libraries etc., it is seen from an infrastructure point of view to lie somewhere between IaaS and PaaS. A service user has more freedom but also more obligations compared to a PaaS offering [219]. In line with other authors from our SLR [133, 274] we see Serverless as a set of technologies ranging from some PaaS offerings to SaaS offerings where FaaS and BaaS are in between and fully considered Serverless. It is service-dependent if a PaaS or SaaS offering is considered Serverless based on the introduced characteristics and the definition. The classification of Figure 3.3 answers RQ1.2 and also provides our view on Serverless technologies.

3.3. FaaS Offerings over Time

We have already seen in the search trends in Section 3.1.5 that early products in the market like AWS Lambda have an impact on how a new domain is formed. Based on this aspect, we look at public cloud FaaS offerings (orange boxes) as well as open-source alternatives (blue boxes) in Figure 3.4. It gives a first glance about the number of platforms as well as their timely distribution. For a detailed information about the release date for public cloud offerings conduct Table 3.4. The initial and latest commits for open-source FaaS platforms are listed in Table 3.5.

The platforms were identified based on literature [97, 172, 207, 294] and the CNCF Serverless landscape[67]. The latter also includes other services we would consider Serverless but are out of scope of our FaaS definition domain. Therefore a selection was made based on the documentation and general information and only as FaaS platforms identified services were considered in the following. The

[67]https://landscape.cncf.io/serverless

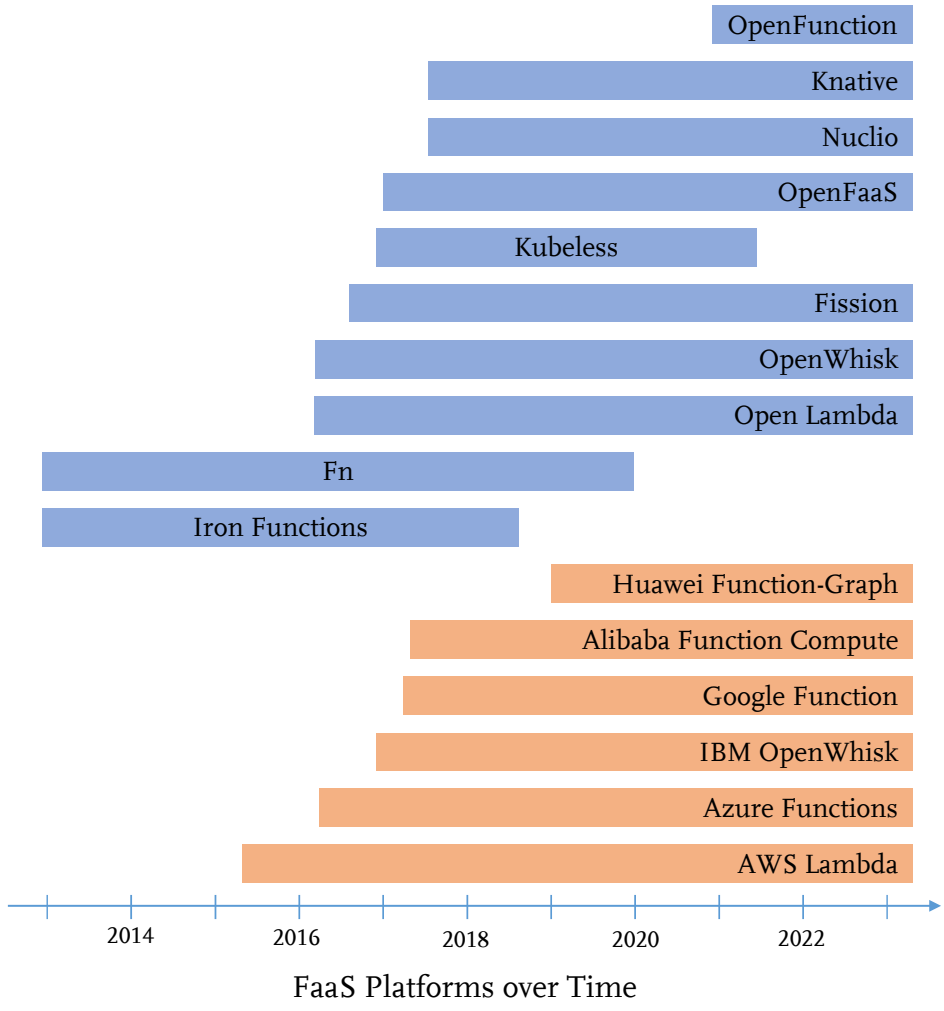


Figure 3.4.: Public cloud provider FaaS platform (orange boxes) and their open-source counterparts (blue boxes) over time.

public cloud providers included are in chronological order: AWS Lambda started in April 2015, AzureFunctions[68] in March 2016, IBM Cloud Functions[69] in December 2016 and Google CloudFunctions in March 2017[70]. These four commercial offerings are often considered in research papers as SLRs show [97, 174, 294]. There are also other public offerings but they are seldom considered in research. For these platforms the number of papers is small and authors of the papers are often affiliated with the companies providing the service. FunctionGraph from Huawei, started in January 2019[71] was recognized twice in an SLR from Yussupov and others [294], where at least one author of the papers was affiliated with Huawei at the time they published their papers [44, 281]. For Alibaba Cloud Function Compute,

[68]https://azure.microsoft.com/en-us/blog/introducing-azure-functions/
[69]Initial release was called Bluemix OpenWhisk. Bluemix was the former name of IBM's cloud offering and OpenWhisk was chosen since IBM Cloud Functions are a provider managed deployment of Apache OpenWhisk, https://rabbah.io/openwhisk.html.
[70]https://cloud.google.com/functions/docs/release-notes
[71]https://support.huaweicloud.com/intl/en-us/wtsnew-functiongraph/index.html

started in April 2017[72] the situation is similar when looking for example at a publication of Wang and others [280]. The last included Asian FaaS platform Baidu Cloud FunctionCompute[73] was not considered in the following since the website was not available in English at the time of writing.

The other platforms, illustrated as blue boxes in Figure 3.4, are open-source. Their source code is available on GitHub[74]. Since there is no public release note for most of these platforms, the initial commits are used as a starting point. In the footnotes of the included platforms, hyperlinks lead to the repository and the initial commit for: IronFunctions [75], Fn[76], OpenLambda[77], OpenWhisk[78], Fission[79], Kubeless[80], OpenFaaS[81], Nuclio[82], Knative[83] and OpenFunction[84]. Two aspects attract attention when looking at the open-source platforms in Figure 3.4. IronFunctions, Fn and Kubeless are stale and not further developed based on their last commits on GitHub. For Kubeless, the last feature specific commit based on the commit message was made on 3rd of May 2021. VMWare decided in late 2021 that they no longer actively develop and maintain Kubeless. The second obvious observation is about IronFunctions and Fn. Their initial commits were made on New Year's Eve 2012. Two projects started on the same day is not a coincident. Fn and IronFunctions share the same initial commit since Fn is a fork of IronFunctions.

In the conceptualization in Section 3, AWS Lambda was attributed as the first FaaS platform which attracted a lot of attention, but there are two open source platforms with an earlier start also providing a FaaS offering. So the first thing is, to ask about the attention these open-source projects had prior to the AWS Lambda offering to correct the prior statement about AWS Lambda if necessary. One kind of attention in the open source community are GitHub stars, the developer pendant to likes in social media. Figure 3.5 shows the distribution of GitHub stars over time, collected with an open source tool called Star History[85]. Precise numbers for the initial/last commit and the starts are included in Table 3.5. It shows that Fn and IronFunctions had their first stars mid of 2016 respectively mid of 2017 which confirms that the tools prior to the AWS Lambda release had negligible impact. Another hint for this statement is the initial README from the IronFunctions repository. The developers stated that IronFunctions started as a small

[72] https://www.alibabacloud.com/help/en/function-compute/latest/product-dynamic-2017
[73] https://intl.cloud.baidu.com/product/cfc.html
[74] Initial, last commit and the number of GitHub stars were updated on the 6th of March 2023.
[75] https://github.com/iron-io/functions
[76] https://github.com/fnproject/fn
[77] https://github.com/open-lambda/open-lambda
[78] https://github.com/apache/openwhisk
[79] https://github.com/fission/fission
[80] https://github.com/vmware-archive/kubeless
[81] https://github.com/openfaas/faas
[82] https://github.com/nuclio/nuclio
[83] https://github.com/knative/serving
[84] https://github.com/OpenFunction/OpenFunction
[85] https://github.com/star-history/star-history

prototype for implementing a reverse proxy and restarting failed workers as well as implementing some auto-scaling features. There were no deployment support at the beginning to build up a cluster of computing nodes but there was already a notion of scaling to zero instances within their project. In August 2018, IronFunctions received its last commit. There is no hint at the GitHub repository if the platform is still under development, but since there were no commits the last years it is highly likely that the contributors stopped their efforts in enhancing this platform. Fn, the fork of IronFunctions, is nowadays the core of Oracle's FaaS offering[86]. Oracle still offers its FaaS platform but the development at GitHub looks stale since December 2019. If Oracle does some closed-source development effort is not accessible.

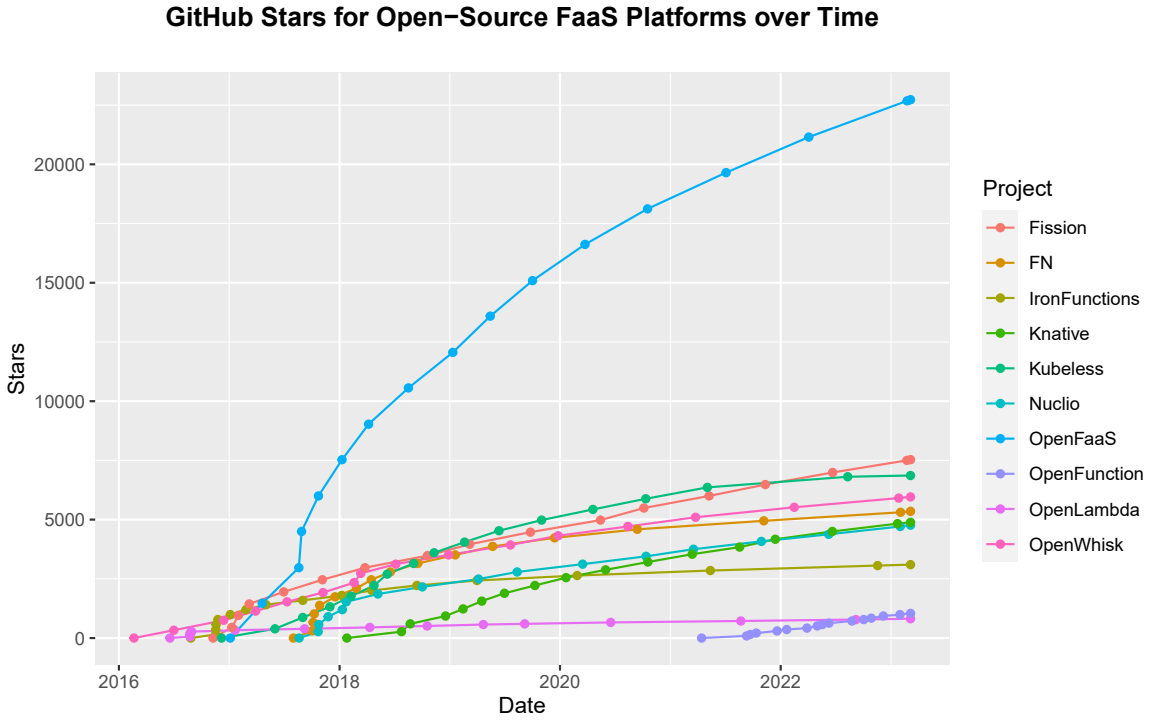


Figure 3.5.: GitHub stars for open-source FaaS offerings over time.

From the GitHub star Figure 3.5 and the remarks, we can state that our conceptualization is still valid. Additionally noteworthy is the sharp increase of GitHub stars for most of the platforms between end of 2016 until end of 2017, the same period where also most of the public cloud provider offerings released their services. This indicates that a lot of developers searched for open-source alternatives and starred the respective platforms.

[86]https://www.oracle.com/cloud/cloud-native/functions/

3.4. Resource Scaling Strategies

To build a simulation framework for cloud functions, we need an understanding of resource scaling strategies of different platforms and how they establish resource configurations. Only platforms with a comprehensible strategy can be simulated. From the documentation of the corresponding services, the current public cloud provider offerings are the first set of platforms we have a closer look at. An overview of their scaling strategies, release date and resource boundaries are listed in Table 3.4. Furthermore, one column indicates if the allocated instances can be configured in a way to have more than a single core. Cloud function instances allocated with a multi-core configuration are only fully utilized when deploying multi-threaded source code. This is an important aspect for a simulation approach and often misinterpreted in research [17, 48, 70, 72, 154, 189] as emphasized by the third contribution (c3) of this work. As already said in Section 3.1.5, AWS Lambda was the first public cloud provider offering for cloud functions. Therefore their scaling strategy set new standards which has to be considered when assessing the other offerings.

Table 3.4.: Summary of resource scaling strategies and limits of selected public cloud provider FaaS offerings.

Provider	Released	Multi-Threading	Scaling of Resources	Boundaries
AWS Lambda	04/2015	☑	CPU proportionally to memory	128-10240 MB, up to 6 vCPUs
Google Cloud Functions	03/2017	☑	9 predefined configurations	128 MB/0.0833 vCPU - 32,768 MB/8 vCPUs
Azure Functions	03/2016	☐ / ☑	Allocated dependent on hosting plan, for consumption plan only a single instance type (☐), premium plan selection of three instance types (One vCPU/3.5 GB RAM; Two vCPU/7 GB RAM; Four vCPU/14 GB RAM) (☑)	
IBM Cloud Functions	12/2016	☑	CPU not specified, memory MB-wise	128-2048 MB
Alibaba Cloud Function Compute	04/2017	☑	CPU and memory independently. GPUs available.	0.05-16 vCPUs, 128-32,768 MB
Huawei Cloud FunctionGraph	01/2019	☑	CPU scaling proportionally to memory	128-4096 MB

In the following, we shortly describe the public cloud provider offerings in chronological order based on their release dates. "[AWS] Lambda allocates CPU power in proportion to the amount of memory configured"[87]. Research in 2021 confirms this statement [185]. Memory can be allocated MB-wise in the range between 128 – 10240 MB with up to 6 vCPUs[88]. Google Cloud Functions offers 9

[87] https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html
[88] https://aws.amazon.com/de/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/

runtime configurations where memory and vCPUs are not linearly scaled[89]. The minimal configuration is 128 MB/0.0833 vCPU and the maximal configuration is 32768 MB/8 vCPUs. Azure Functions offers several resource plans with different possibilities[90]. The different hosting plan options determine the resource allocation and scaling of function instances under different load conditions. For the consumption plan there is only a single instance type available where a single core is assigned. Therefore, multi-threaded functions would not profit compared to single-threaded functions with regards to execution time. For the premium plan, three configurations are possible which can be selected based on roles in the Azure portal. These types include a single core instance with 3.5 GB memory and two multi-core settings with 2 virtual CPUs (vCPUs)/7 GB and 4 vCPUs/14 GB[91]. The configuration of the functions is somewhat hidden in the documentation. Also their scaling property is different compared to other public cloud offerings, i.e. AWS Lambda, IBM Cloud Functions and Google Cloud Functions. Due to these reasons, it is difficult to compare Azure Functions with other public cloud provider platforms [17]. IBM Cloud Functions hosts the open-source FaaS platform Apache OpenWhisk for public cloud customers. Currently there is an option to scale memory from 128 – 2048 MB[92]. There is no description in the documentation how CPU resources are scaled. When looking at the source code of Apache OpenWhisk[93], the documentation there states that CPU scaling can be enabled and if so the resources are scaled linearly based on the memory setting. From an experiment [176] we know that this scaling is not linear based on the memory setting. It looks like that CPU shares are fixed when analyzing the execution time of the CPU intensive functions. Furthermore, we were able to confirm based on the implementation of a multi-threaded prime number search that the cloud function instances have more than one vCPU assigned independent of the memory setting. In our experiment, instances with a low memory setting (256 MB) performed better in this multi-threaded scenario compared to instances with a high memory setting (2048 MB). As the only included public provider, Alibaba Cloud[94] offers an independent scaling of memory and vCPU. For memory, the boundaries are between 128 MB and 32,768 MB. vCPU shares can be selected in a range from 0.05 to 16. Huawei Cloud FunctionGraph is the last offering in our list of public cloud providers and also the youngest one. They follow the approach of AWS Lambda and Google Cloud Functions and allocate vCPU resources based on the selected memory setting. At their documentation, they provide a formula to

[89] https://cloud.google.com/functions/pricing
[90] https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale
[91] https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal
[92] https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits
[93] https://github.com/apache/openwhisk/blob/.../WhiskPodBuilder.scala
[94] https://www.alibabacloud.com/help/en/function-compute/latest/instance-types-and-instance-modes

calculate the assigned vCPUs: $memory/128 * 0.1\ vCPU + 0.2\ vCPU$[95]. They allow for a range from 128 MB to 4096 MB memory for a cloud function instance which would result, based on the presented formula, in 3.4 vCPUs for 4096 MB[96].

When looking at the resource scaling strategies in Table 3.4 all platforms have in common that they offer configurations where users have multiple cores for their functions. Different to microservices or even applications which comprise of several functionalities executed in parallel, the utilization of a cloud function instance is fully dependent on the capability of the function to use these resources. Therefore, multi-threading is one important dimension when doing public cloud FaaS research and has to be considered for a simulation approach. Only a few publications address this multi-threading aspect in literature, e.g. [180, 185, 293, 300].

Table 3.5.: Summary of resource scaling strategies and limits of ten open-source FaaS offerings.

Platform	First Commit	Last Commit	K8s Deployment	Memory Scaling	CPU Scaling	GitHub Stars
Apache OpenWhisk	19/02/2016	01/03/2023	☑	128-2048 MB	dependent on memory scaling	5,955
Kubeless	16/11/2016	03/05/2021	☑	N/A	N/A	6,862
OpenFaaS	22/12/2016	28/01/2023	☑	based on K8s limits	based on K8s limits	22,723
Fission	19/08/2016	28/02/2023	☑	boundaries (min/max)	boundaries (min/max)	7,528
OpenLambda	02/02/2016	23/02/2023	☐	N/A	N/A	815
Nuclio	04/06/2017	02/03/2023	☑	based on K8s limits	based on K8s limits	4,765
Knative	30/01/2018	02/03/2023	☑	based on K8s limits	based on K8s limits	4,886
OpenFunction	05/12/2020	28/02/2023	☑	N/A	N/A	1,040
IronFunctions	31/12/2012	20/08/2018	☐	N/A	N/A	3,098
Fn	31/12/2012	19/12/2019	☑	based on CLI run parameter	N/A	5,350

The open-source FaaS landscape is a bit different. Since there is no provider managing a service, the resource and transitively cost perspective is often not discussed for open-source platforms. To make informed decisions about the function configurations and to be able to calculate Total Cost of Ownership (TCO) of a cloud function hosted on premise, resource scaling strategies of open-source tools are important. Furthermore, without such strategies and an assignment of computing resources to cloud functions, there is no guarantee for the computing resources available on the machine due to other tenants which leads to unpredictable performance situations and Service Level Agreement (SLA) violations. The import-

[95] https://support.huaweicloud.com/intl/en-us/ae-ad-1-devg-functiongraph/-functiongraph_02_0420.html

[96] https://support.huaweicloud.com/intl/en-us/usermanual-functiongraph/-functiongraph_01_1828.html

ance of comparable resource scaling approaches for open-source FaaS platforms compared to commercial cloud offerings is discussed in Section 6.4 in detail.

Table 3.5 summarizes information about the open-source platforms from the prior section. For Apache OpenWhisk[97] already mentioned as the public offering of IBM Cloud Functions, there is an option to scale memory from 128 – 2048 MB[98]. A CPU limit like for the memory setting is under development, but the pull request under review looks stale[99]. Therefore, the assignment of CPU resources to a cloud function instance is dependent on the used infrastructure management tool. OpenWhisk's recommended local deployment option is K8s where scaling of memory for functions is possible with an environment variable. The next open-source platform Kubeless is currently without a maintainer since VMware stopped their support for the project. Since the documentation is no longer accessible, an assessment of the memory and CPU scaling cannot be made. OpenFaaS offers two deployment options. The single node deployment is offered via *faasd* where resource limits are not available. The recommended deployment option is again K8s where resource limits for memory and CPU can be set independently[100]. This feature is based on K8s facilities to restrict resources within the deployment YAML adapted by OpenFaaS. Another K8s based tools is Fission. Limits are assigned to function environments[101] where boundaries of minimum and maximum CPU or memory can be specified via environment variables. OpenLambda [99] is a research prototype based on Linux containers. It uses the cgroups capabilities of Linux containers for min/max memory and CPU boundaries. It has no K8s support. Nuclio[102] and Knative[103] have the same resource assignment strategy. Both tools point to the K8s reference on using resources[104] and obviously rely on a K8s deployment. Using resource restrictions for OpenFunction was not detectable. It offers a Knative based deployment option which might allow to restrict resources but that is only speculation and can not be checked based on the tool documentation. Due to it's Knative deployment option, the checkbox for K8s deployment is checked. IronFunctions uses docker containers for the function's server and API. Due to its focus on implementing a reverse proxy and autoscaling as already mentioned, there is no information for a multi-node environment. Also they do not offer a K8s deployment. Since Fn builds upon IronFunctions, the same holds true for the tight docker integration as well as for the missing K8s support within the project, but there is a Helm chart to deploy Fn on a K8s cluster. Another dif-

[97] https://openwhisk.apache.org
[98] https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits
[99] https://github.com/apache/openwhisk/pull/4648
[100] https://docs.openfaas.com/reference/yaml/#function-memorycpu-limits
[101] https://fission.io/docs/usage/function/executor/
[102] https://nuclio.io/docs/latest/reference/function-configuration/function-configuration-reference/
[103] https://knative.dev/docs/serving/services/configure-requests-limits-services/
[104] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

ference to IronFunctions is the possibility to configure functions in a declarative way via YAML files where memory configuration is one aspect[105].

As can be seen from Table 3.5, most open-source platforms offer a K8s deployment option, where only OpenLambda, a research prototype, and IronFunctions, started its developement in the pre K8s era, lack in this dimension. From the remaining eight platforms, three have no information how resources for cloud function instances are assigned based on their documentations, i.e. Kubeless, OpenFunction, Fn. Another offering, Apache OpenWhisk, states that it scales CPU based on the selected memory setting, but the pull request is unchanged for years. The remaining four, which is half of this group use resource limits based on container settings (Fission) or the K8s resource limits (OpenFaaS, Nuclio, Knative). Delegating these settings to the deployment capabilities of K8s passes the responsibility for resource allocation and utilization to the container orchestration platform. This finding is in line with those of other researchers, e.g. [11, 31, 113]. They lead us to argue that K8s could be a higher layer of abstraction for implementing open-source FaaS platforms and hosting cloud functions. It is another aspect which fosters the statement that K8s is tightly related to the hype about Serverless technologies.

3.5. Architecture of a FaaS Platform Worker Node

In Section 2.1, we already discussed the topic of virtualization with VMs and containers. We use the hypervisor design (see Figure 2.1 middle) with KVM as Type-1* hypervisor and a compatible Type-2* hypervisor as listed in the figure. Also the other two options described in the virtualization paragraph could be used to abstract the hardware. There are several other publication which describe how a FaaS platform is build from a more abstract level. These include all the important building blocks, i.e. a reverse proxy, a scheduler and worker nodes [118, 149, 188, 276]. We therefore focus on a more detailed architecture of a typical FaaS worker node as shown in Figure 3.6.

Since we build a simulation framework, the SUT as well as our simulator should have a similar tool stack as already mentioned in Section 1.2. We frame these comparable technology stacks as dev-prod parity[4]. This parity is important as Arif and others [4] showed. They compared physical and virtual environments and came to the conclusion that a comparison is only possible when the systems run on a comparable technology stack. We already know from the conceptualization prior in this section that FaaS platforms are virtualized. Now there are two options to choose from: VMs and containers. For public FaaS platforms, there is an additional requirement we have to consider. The NIST characteristics indicate that there is a shared fleet of servers handling requests from several customers. Therefore one requirement for our FaaS worker node architecture is isolation of

[105] https://github.com/fnproject/docs/blob/master/fn/develop/func-file.md

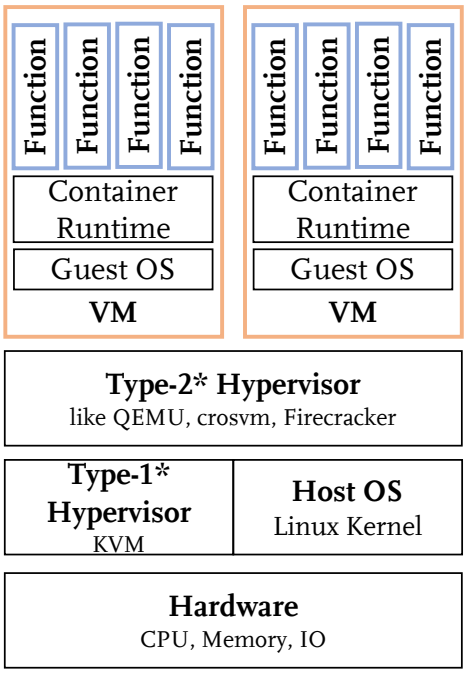


Figure 3.6.: A suggested architecture for a FaaS worker node.

workloads. In Figure 3.6 we achieve this isolation via VMs where every running VM has exactly one assigned customers. Within this secure execution context, a container runtime as another level of abstraction serves multiple functions of the same customer. Therefore, the customer profits from fast boot-times of containers and a secure VM enclave in theory. Dependent on the platform, this design has performance implications when VMs are not pre-provisioned. In such cases performance can vary up to 15 times as shown by an experiment of Lloyd and others [157]. For open-source platforms the default level of abstraction are containers. To the best of the author's knowledge none of the discussed open-source tools describe these two-face security design as presented in Figure 3.6. An explanation could be the implicit assumption that open-source platforms are often deployed on-premise and only used by a single customer which makes the first level of isolation obsolete.

3.6. Summary

In this chapter, we discussed the origin of terms and how FaaS integrates into the cloud computing service landscape. We furthermore listed different FaaS offerings, discussed their resource scaling strategies and ended with a typical architecture of a FaaS worker node.

Based on a proper understanding of the research domain and an overview of available tools, we have chosen a public cloud provider service and an open-source platform where we predominantly conducted experiments with in the course of this dissertation project. As public cloud offering, AWS Lambda was selected due

to its dominant position by forming a new area of computing platforms, its market leadership and the fact that it is the most often used platform in research papers [97]. For open source platforms, OpenFaaS is the clear leader with regards to stars which also implies its maturity and a strong developer community. It is also often used in scientific literature to perform experiments with an open-source platform to improve the platform itself [12] or compare the capabilities of the platform with other open-source platforms [11, 13, 149, 150, 197] or to the public cloud [180]. Before we go into the details of the experiments performed with AWS Lambda (Sections 6.3, 6.4 and 7.2) and OpenFaaS (Section 6.4), the next sections discuss some of the foundations for a benchmarking and simulation framework for FaaS.

Part III.

A Benchmarking and Simulation Framework for Function as a Service

4. Benchmarking FaaS Platforms

Parts of this chapter have been taken from [174, 182, 185].

In this chapter, RQ2.1 *(Which tools and experiments do currently exist for benchmarking FaaS platforms?)* and RQ2.2 *(How should a FaaS experiment be documented and which items are necessary for data evaluation?)* are supported.

In Section 4.1, an SLR is performed to answer RQ2.1. During the work on the SLR an included secondary study revealed a lack of reproducibility within the research efforts in the FaaS domain. This motivated a more thorough investigation concerning the aspects needed for a proper data evaluation to understand the technical realization of resource scaling strategies of FaaS platforms. One result is a checklist for data generation to enable a fair evaluation of different FaaS platforms in Section 4.2. Tooling is one approach to deal with reproducibility and guarantee that requests are executed for several experiments in the same way. Therefore, Section 4.3 gives insights how the research prototype `SeMoDe` enables other researchers to conduct experiments in a reproducible and well documented way and includes capabilities to present pre-processed data.

4.1. Current Benchmarking Approaches and Tools

Benchmarking FaaS platforms is directly motivated by the research goal of this dissertation project. To achieve this goal of proposing a simulation framework, we need an understanding of the different platforms, verify their documented scaling strategies and harvest data as an input for our simulation system. Eivy [74, p.9] claims that the "real execution is the only valid test". Therefore, we look at already published work in the FaaS research domain with a focus on benchmarking approaches and tools. These research efforts investigate special properties of the platforms as well as ways to optimize the cloud functions runtime behavior. By performing another SLR we approach this body of knowledge in a structured way and answer the first research question of this chapter, *which tools and experiments do currently exist for benchmarking FaaS platforms?* After defining a review protocol [176], dblp computer science bibliography[106] was picked as a literature database since it includes work from journals and conference proceedings including different publishers like ACM and IEEE.

[106] https://dblp.uni-trier.de/

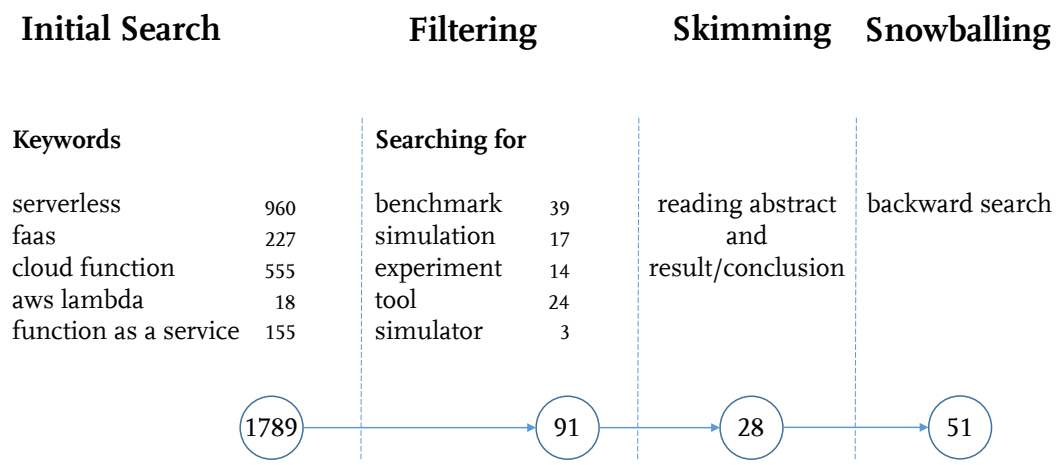


Figure 4.1.: Strucutred Literature Review conducted on 13th of March 2023 to identify empirical FaaS research.

Figure 4.1 shows the SLR process and number of collected papers in each step. Since the advanced search at dblp was disfunctional when combining several search terms with an OR operator, the decision was made to conduct multiple searches with the keywords presented in the initial search phase: *serverless*, *faas*, *cloud function*, *aws lambda* and *function as a service*. Conducting a single query for each of these search terms is equivalent with a more advanced search having a single search string where the search terms are concatenated with ORs. The usage of generic, broad keywords was a conscious decision to get a broad set of literature which is the basis for the further steps. The initial search phase when combining the result sets of the five queries resulted in 1915 entries. To handle the literature, the bibliography manager JabRef[107] was used. One feature of JabRef is to remove duplicates within a literature database. After applying this feature to our initial set of literature, we ended up with 1789 unique entries. Since this section is about finding competing approaches to the work in hand, all publications mentioned in Table 1.1 were excluded. The remaining 1789 entries were then filtered by the search feature of JabRef. Similarly to the search on dblp, the decision was made to perform a single search for *benchmark*, *simulation*, *experiment*, *tool* and *simulator*. The filtering keywords were chosen due to the focus on understanding benchmark and simulation approaches in FaaS as well as looking at experiments and tools. We consider benchmarking experiments in this section and discuss simulation approaches in Section 6.2. Nevertheless, each simulation needs input data in form of raw data from benchmarking experiments. When writing about publications with a simulation focus in this section, an emphasis is put on the data generation and therefore the benchmarking within these papers.

The filtering process was done on the title and other meta information like the conference or journal name. After removing six duplicates, 91 papers were skimmed in the next phase based on their abstract, result and conclusion section. For

[107] https://www.jabref.org/

most of them, it was sufficient to read the abstract to evaluate their relevance. To remove doubts on ambiguous papers, the research questions and the results part were read to make a well justified decision. After the skimming phase, 28 papers were left which were read in detail while answering the questions of the review protocol. Since the search was only done on dblp and the keywords were quite generic for the initial search, an additional backward search was conducted for every publication to identify further relevant research. The snowballing process was performed for all 28 papers in the third step and also the papers which were identified during snowballing. After finishing this recursive snowballing process, 51 papers were part of the final set. The following tables show the literature set by categorizing them in secondary performance studies (Table 4.1), papers which are not directly related to experiments on public cloud providers (Table 4.2) and experiments which are important for the prototype design in Section 4.3 and the improvement of empirical public cloud FaaS research (Table 4.3). The focus on public cloud provider experiments has two reasons. The first reason is the number of publications conducted to understand the scaling strategies and resource assignments of public cloud providers. The second aspect is that the body of knowledge for open-source FaaS platforms is limited. Only our work at IEEE CLOUD 2022 [180] where we put emphasis on resource assignments for open source platforms and Zhang and others [300] dealt with resource restrictions for open source platforms.

Table 4.1.: Secondary studies within the Structured Literature Review on benchmarking and simulation approaches.

Authors	Year	Title
Kuhlenkamp & Werner [137]	2018	Benchmarking FaaS Platforms: Call for Community Participation
Yussupov et al. [294]	2019	A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools
Scheuner & Leitner [240]	2020	Function-as-a-Service performance evaluation: A multivocal literature review

Table 4.1 includes secondary studies which address specific aspects within the FaaS research domain. These secondary studies are not directly related to benchmarking and simulation experiments since they are not publishing new data and insights from practical experiments but they give guidance and an overview of relevant literature for specific issues. Yussupov and others [294] motivate the work on a consistent benchmark and simulation framework due to their SLR being focused on different platforms and tools. Their work was one of the foundations of the RADON [39] project. One goal of this EU horizon project is the development of an IDE for serverless applications. Performance aspects are discussed in Scheuner's and Leitner's work [240] where they included 112 sources from academic and gray literature. Especially interesting is the design of different experiments found in literature and their impacts on the data evaluation and therefore on the

results. Kuhlenkamp and Werner [137] motivated the community to participate in the benchmarking process by revealing issues about the reproducibility of conducted research. The authors stated in their third research question that only 3 out of 26 experiments were reproducible based on the provided information included in the experiment and data evaluation description. To overcome this problem of partly-documented experiments, a checklist is proposed in the next section which gives an answer to RQ2.2.

Table 4.2.: Primary studies within the Structured Literature Review on benchmarking and simulation approaches which were included based on the literature process but not directly related to public cloud provider experiments.

Authors	Year	Title
van Eyk et al. [277]	2020	Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark
Mohanty et al. [197]	2018	An Evaluation of Open Source Serverless Computing Frameworks
Govind & González-Vélez [90]	2021	Benchmarking Serverless Workloads on Kubernetes
Lin et al. [156]	2021	BBServerless: A Bursty Traffic Benchmark for Serverless
Zhang et al. [300]	2021	An Experimental Analysis of Function Performance with Resource Allocation on Serverless Platform
Das et al. [56]	2018	EdgeBench: Benchmarking Edge Computing Platforms
Jeon et al. [109]	2019	A CloudSim-Extension for Simulating Distributed Functions-as-a-Service
Bardsley et al. [18]	2018	Serverless Performance and Optimization Strategies
Gan et al. [83]	2019	An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems
Kritikos & Skrzypek [136]	2019	Simulation-as-a-Service with Serverless Computing
Malla & Christensen [169]	2019	HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)
Malawski et al. [168]	2020	Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions
Sadaqat et al. [237]	2022	Benchmarking Serverless Computing: Performance and Usability
de Carvalho & de Araújo [38]	2022	Orama: A Benchmark Framework for Function-as-a-Service
Hancock et al. [95]	2022	OrcBench: A Representative Serverless Benchmark
Palepu et al. [208]	2022	Benchmarking the Data Layer Across Serverless Platforms
McGrath and Brenner [190]	2017	Serverless Computing: Design, Implementation, and Performance
Gias & Casale [85]	2020	COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms
Somu et al. [254]	2020	PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications
Quaresma et al. [225]	2021	Validation of a simulation model for FaaS performance benchmarking using predictive validation
Ustiugov et al. [270]	2021	Benchmarking, Analysis, and Optimization of Serverless Function Snapshots

Table 4.2 contains the publications, which were classified as related to presented empirical FaaS research in this work but not directly supporting efforts in building the benchmark and simulation framework. Nevertheless, the contributions of these papers are important to stress the context and related research areas. The papers were categorized and each cutting line separates one subset from another.

The first bibliographic entry [277] is a vision paper about how a benchmark in the FaaS area should look like with a focus on real world experiments to overcome microbenchmarks. This vision paper was not further assessed since no raw performance respectively function configuration data were provided.

The next set of papers performed experiments with open-source platforms. Mohanty and others [197] evaluated the open source platforms Fission, Kubeless and OpenFaaS. They focused on testing how concurrent users and the autoscaling property, which is based on Kubernetes Horizontal Pod Autoscaler, influence the number of running function instances. Another K8s related research was performed by Govind and González-Vélez [90]. They tried to understand the parallelism and scalability properties of OpenFaaS deployed on a multi-node master K8s cluster. During their experiments, they recognized performance degradation for a specific number of users when starting new instances. Since there is no documented resource assignment, the default for OpenFaaS and K8s is to create more functions than the system can handle under peak load as shown by one of our experiments on OpenFaaS [180]. This multi-tenancy phenomena is discussed in detail in Section 6.4. To put emphasis on the scaling of resources for open source platforms, Zhang and others [300] deployed OpenWhisk on a K8s cluster. Especially the application of K8s limits to function instances is noteworthy since they describe in their paper that multi-threaded functions indeed profit from a resource increase beyond a single core, whereas memory intensive, single-threaded functions do not. A unique experiment design characteristic of their investigation is that they consider memory and CPU in combination. Furthermore their data is promising for a better understanding of resource allocation and assignment strategies for open-source platforms based on K8s. A drawback is that the source code is not open-source and therefore a detailed assessment of their function implementation is not possible. The next experiment [156], as the name already implies, puts emphasis on bursty benchmarks. The authors described four use cases for testing their new benchmarking platform namely web application, big data, streaming and machine learning use cases. They deal with an important topic and argue comprehensibly why they put so much effort in an assessment of bursty workloads. Nevertheless, in the course of their benchmark approach and questions for a resource aware FaaS configuration, this paper does not state any function configuration besides noting that they use the same K8s cluster configuration and the default configuration for all targeted platforms.

The same holds true for the edge computing frameworks, where the first work used AWS Greengrass[108] and Azure IoT Edge[109] for latency measurements and cost comparisons [56]. The second publication [109] implemented an extension for CloudSim [36], a popular simulator for cloud offerings, with the aim to support a distributed FaaS architecture comprised of cloud, fog and edge. This layering of an

[108] https://aws.amazon.com/greengrass/
[109] https://azure.microsoft.com/en-us/services/iot-edge/

application where parts can be deployed in several of the aforementioned areas and shifted within a distributed architecture is also known as osmotic computing [278].

Another subarea are use case implementations using FaaS platforms as execution environment for simulations [136], cloud functions as part of an e-commerce system [18], building microservices architecture with predictable performance [83] or checking the High Performance Computing (HPC) capabilities of cloud function offerings compared to IaaS [169] or against each other [168]. A last use case paper was a survey to understand how well public FaaS platforms are understood and can be used out of the box to implement applications [237].

The following three publications try to understand the effects of backend services. Orama is a framework with six built-in use cases to understand the interaction with data storage solutions and API gateways, the most common integration scenario for FaaS [38]. Related to this research is the work of Palepu and others [208]. They tested the data transfer rates on different FaaS platforms for different data storage solutions. The last work in this category is early work on OrcBench [95]. The focus here is to cluster execution traces from an Microsoft Azure dataset with 52,000 functions and 8.8 billion invocations and reengineer common execution time and invocation pattern.

The last group of papers optimized some aspects of FaaS platforms or the management of cloud functions by providing a custom implementation or configuration. These aspects are out of scope for implementing a benchmark and simulation pipeline for assessing function performance and characteristics on public FaaS platform offerings, but reveal interesting results by highlighting different shortcomings in today's FaaS offerings. Quaresma and others [225] built a simulator to predict their prior introduced suppression of garbage collection during function execution and compare this to invocations on public providers. Solving the startup of *unnecessary* instances in over-provisioning scenarios is researched by Gias and Casale [85]. They implemented a queuing based approach for on-premise platforms which breaks the scaling on demand principle of FaaS. The next work [190] implemented a prototype in .NET deployed on Azure's cloud platform, compared it to public offerings and showed some benefits especially for throughput (executions per second). Another work [254] suggests a tool for deploying function and other components to different providers. The function deployment can also, as described in the paper, be done by using the serverless framework[110]. Especially at early publications, the serverless framework was often used also by other papers included in this SLR, like [6, 139, 167, 189]. Nevertheless, the benefits over generic IaC tools like Terraform[111] or provider specific ones like AWS CloudFormation[112] are not clear in the aforementioned papers. Noteworthy in this last group of papers is the effort of Ustiugov and others [270].

[110]https://www.serverless.com/
[111]https://www.terraform.io/
[112]https://aws.amazon.com/cloudformation/

Their major research goal is to provide a serverless open source playground[113] for experimentation on various layers of the system stack like Firecracker hypervisor, see Section 2.1.

When assessing the publications of Table 4.2, the finding of Kuhlenkamp and Werner [137] is not surprising that only a minority of experiments documented all information to perform them again.

Table 4.3 includes competing approaches and presents data from public cloud provider experiments. When checking the public FaaS platforms within the experiments, the ordering of platforms is similar to Scheuner and Leitner [240] where AWS Lambda is the most investigated platformed followed by Azure Functions, Google Cloud Functions and lastly IBM OpenWhisk. Important for repeatable research is to enable other researchers to reproduce the experiments. Only 18 out of the 27 publications have open sourced their prototypes which renders already one third not reproducible.

Different languages and their effects on execution time were investigated by Jackson and Clynch [107] for AWS Lambda and Azure Functions. What is missing in their paper is the memory setting for AWS Lambda, which has an influence on the execution time [185]. Another language comparison is done by Kuntsevich and others [141] tackling a locally hosted OpenWhisk installation. Due to the nature of the paper being a demo, they only included first results on the execution time behavior for concurrent requests. There is no notion of K8s limits applied to functions. The data published by Martins and others [189] contains noise in form of their evaluation based on the Round Trip Time (RTT). A comparison of their RTT and the elapsed time measured on the provider platform would enrich the data and cancel out some network delays, platform scheduling etc. Furthermore, the HelloWorld use cases in different languages are limited in their interpretation since there are constellations where a function running in Java is faster than a comparable JavaScript counterpart as shown in [182]. Hence, a selection of the programming language and the function configuration should be use-case dependent. One experiment in Martin's work is commonly used in other research, e.g. [138, 182, 185, 213, 255, 271], where a Fibonacci function was executed at different memory settings on different providers to understand the resource scaling strategies. Therefore, this function was also used for experiments in latter parts of this thesis. As argued before [185], microbenchmarking can unravel platform mysteries when designing experiments in a way that a single aspect is isolated. For that reason, Kim and Lee [121, 123] published a benchmark suite comprised of microbenchmarking and application level benchmarks. Their collection of different function implementations can serve as foundation for experiments to discuss common cloud functions and applications in research. Nevertheless, they argued that microbenchmarking is important but not sufficient for the requirements real world use cases present. Authors of other real world experiments like SeBS [48] and BeFaaS [91] enforced that argument but are aware that real world use cases

[113] https://github.com/ease-lab/vhive

Table 4.3.: Primary studies within the Structured Literature Review on benchmarking and simulation approaches which present data on public cloud provider experiments.

Authors	Year	Title
Back & Andrikopoulos [6]	2018	Using a Microbenchmark to Compare Function as a Service Solutions
Figiela et al. [79]	2018	Performance evaluation of heterogeneous cloud functions
Jackson & Clynch [107]	2018	An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions
Kuntsevich et al. [141]	2018	Demo Abstract: A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform
Lee et al. [145]	2018	Evaluation of Production Serverless Computing Environments
Lloyd et al. [157]	2018	Serverless Computing: An Investigation of Factors Influencing Microservice Performance
Malawski et al. [167]	2018	Benchmarking Heterogeneous Cloud Functions
Pawlik et al. [211]	2018	Performance evaluation of parallel cloud functions
Wang et al. [282]	2018	Peeking Behind the Curtains of Serverless Platforms
Bortolini & Obelheiro [29]	2019	Investigating Performance and Cost in Function-as-a-Service Platforms
Giménez-Alventosa et al. [86]	2019	A framework and a performance assessment for serverless MapReduce on AWS Lambda
Kim & Lee [121]	2019	FunctionBench: A Suite of Workloads for Serverless Cloud Function Service
Kim & Lee [122]	2019	Practical Cloud Workloads for Serverless FaaS
Pellegrini et al. [214]	2019	Function-as-a-Service Benchmarking Framework
Copik et al. [48]	2020	SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing
Kuhlenkamp at al. [139]	2020	Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications
Maissen et al. [166]	2020	FaaSdom: a benchmark suite for serverless computing
Martins et al. [189]	2020	Benchmarking Serverless Computing Platforms
Yu et al. [293]	2020	Characterizing serverless platforms with serverlessbench
Grambow et al. [91]	2021	BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms
Lin & Khazaei [154]	2021	Modeling and Optimization of Performance and Cost of Serverless Applications
Pons and López [17]	2021	Benchmarking Parallelism in FaaS Platforms
Ristov et al.[231]	2022	Colder Than the Warm Start and Warmer Than the Cold Start! Experience the Spawn Start in FaaS Providers
Scheuner et al. [241]	2022	TriggerBench: A Performance Benchmark for Serverless Function Triggers
Akhtar et al. [2]	2020	COSE: Configuring Serverless Functions using Statistical Learning
Eismann et al. [72]	2021	Sizeless: Predicting the Optimal Size of Serverless Functions
Mahmoudi & Khazaei [165]	2021	SimFaaS: A Performance Simulator for Serverless Computing Platforms

are often not that precise in their conclusions due to noise in the collected data. However, they share insights on the adaptation of the technology. SeBS [48] concludes that IO bound functions in general do not profit from cloud the function execution model due to the double billing problem [10]. They also included a cost analysis and compared their real world examples to IaaS solutions. Despite being also real world conform by implementing a webshop and traffic light use case, BeFaaS [91] has a different focus on federated workloads where their traffic light application is comprised of a cloud and edge layer. The latter is implemented by using their tinyFaaS [217] system. Some information like the configurations for the cloud functions is missing and, therefore, the comparison of different providers is limited.

An important aspect in FaaS research as already mentioned is to understand the resource scaling strategies of public FaaS providers for choosing the right configuration dependent on the use case. Early work [6] already published data for different memory settings and their cost implication, but did not relate nor statistically assess the interrelation between performance and cost. AWS Lambda claims that it "allocates CPU power in proportion to the amount of memory configured"[114] and bills the user on millisecond granularity where the price also increases linearly in the same way CPU and other resources do. One study confirmed this statement in an ideal world with CPU intensive functions [29] which corresponds to our findings [185]. Resource scaling and runtime prediction is not that deterministic on the other platforms, in particular at Google Cloud Functions and IBM OpenWhisk. Maissen and others [166] documented different processors and cold start times by executing the cloud functions in various regions. These cold start times come from the fact that FaaS providers reuse cloud function instances within a provider dependent period until the cloud function faces a shutdown. This optimization is hidden from the FaaS customer. Also Giménez-Alventosa and others [86] investigated differently equipped execution environments. Of particular interest is their data analysis where they grouped the invocations of the functions by the executing VM. In the resulting histogram, there were deviations of up to 30% in execution time for the same functionality and configuration. Since they missed to also record the physical machine's information, first of all the CPU specifications, the reason for this huge deviation remains unanswered. When relating these measures with other experiments [50, 206, 213], one explanation could be the different physical machines used for deploying the VMs and executing the cloud functions. Such a variation of different execution environments results in an unfair business model where a provider bills a customer up to 30% more for the same service. But there is also research where performance across availability regions is consistent [231]. Their interesting results reveal that AWS shows a consistent execution behavior across three different regions where only a single CPU was used to execute the functions. IBM Cloud Functions showed a scattered execution behavior with a variety of CPUs used and on Google the cold starts were faster than the warm ones.

[114] https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

These insights were one reason to include the VM identification and the hardware specification of the executing machine in the checklist in Section 4.2 to support a proper data evaluation. Since we already know that the physical host executing the function determines the resource assignment and ultimately the execution time, Wang and others work [282] is helpful as they found a way to uniquely identify VMs when executing the cloud functions. Their approach differs from the VM uptime approach suggested by Lloyd and others [157] and is collision free. Back in 2018, they recognized performance drops when a lot of function request were made concurrently, leading to a placement strategy where AWS Lambda executes multiple cloud functions on the same VM. Therefore, performance isolation was not guaranteed. Another concurrency study [145] was motivated to compare FaaS to an IaaS VM offering. A data intensive application was used to show the fit of FaaS for distributed computing where also other researchers see the benefits of this computing model [112]. The aforementioned multi-tenancy problems were not present in this publication. In 2020, Barcelona-Pons and López [17] also tackled the request parallelism in FaaS, however, they did not recognize a noisy neighbor effect in their data. This finding is a hint that the AWS Lambda platform improved in this property which is also supported by the AWS Security[115] documentation where the software stack AWS Lambda is running on is depicted. This stack is quite similar to the architecture proposed in Section 3.5 where VMs are applied for security reasons and containers for fast start-ups. They also discovered in their work that the different scheduling strategies, how requests are distributed to instances, impact the fit of different platforms for application classes like IO bound functions and Azure Functions. In Table 3 of their work, the python function with 256 MB configured took 7.7 seconds whereas the 2048 MB function took 1.1 seconds. The cloud function was only 7 times faster despite having 8 times more resources. We know from the AWS Lambda documentation, that an increase of memory also scales CPU resources linearly where 1,769 MB is equivalent with one vCPU[116]. After this limit, only multi-threaded functions profit from a resource increase. This effect of resource scaling is also overlooked in other papers included in the SLR, e.g. [154, 241]. Lin and Khazaei state in Section 4.1 of their paper "when the allocated memory is greater than 1792 MB, the response time remains almost the same" [154, p. 624], but they give no explanation for this phenomena. Another work which discusses latency breakdowns based on tracing information claims that cloud function configuration beyond a single CPU is "inefficient for non-CPU-intensive load" [241, p. 7]. As already remarked, there are situations where cloud functions profit from a resource increase greater than a single core but only if the function is implemented in a multi-threaded way. Kuhlenkamp and others [139] were interested in another scaling aspect. They created five workloads with different characteristics to understand the scaling, i.e. the creation

[115] https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.html

[116] https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

of new cloud function instances. To distinguish between the execution time, understanding the network overhead, request scheduling etc. at the platforms, they used a multi step measurement methodology. They recorded the start and end time on their experiment machine which corresponds to the RTT of another publication [189] and compared this value to the execution time on the platform. Since they only executed a single function (prime number search) on a single memory setting, their insights on how different cloud function configurations influence the execution behavior of a function is limited, but related to understanding the impact of the application workload on the number of instances [179]. A different approach to assess routing properties and the number of instances is to implement a proxy [214] to understand the inner workings of platforms.

Another set of publications [79, 167, 211] were done by a research group from AGH University of Science and Technology in Poland. They raised questions about the computational performance proportional to function size, the network performance, overhead introduced by network and scheduling, reusage of application instances and their recycling time as well as the heterogeneous hardware. When they conducted their research on AWS Lambda back in 2017, the service was limited to a single core and 1536 MB memory, but they already found a linear relationship as stated by the AWS Lambda documentation. For other providers, they did not find that relation which is in line with other research [29]. In an ideal world, for embarrassingly parallel computation, the service implementation of AWS Lambda leads to a situation of constant cost when raising the memory setting of the function as depicted in their cost calculation in Figure 12 of their work [79]. Or to put it differently, a user gets the same portion of computing power over time for the same price as also confirmed in our work [185]. Similar to Maissen, they also found different CPU models executing the cloud function, which could also explain the scatter plot in Figure 5 in [79] (respectively Figure 4 in [167]), where two performance ranges for LINPACK were visible. The last included benchmarking paper in the SLR is another workbench, called Serverless-Bench [293]. They included an experiment with a multi-threaded application compared to parallel function instances. Since the memory configuration of the different cloud functions included in the plots are missing, the results are only partly interpretable. Their second implication is that splitting parallelizable parts of a function into several concurrently executed functions is use case dependent. We showed [185] that an efficient fork join implementation of a prime number search can also benefit from multi-threading. Nevertheless, their focus on splitting functions dependent on their resource needs, i.e. IO-bound function and CPU-bound functions, is interesting for further investigation.

Most of the papers discussed so far are benchmarking papers. Since the aim of this work is to build a simulation framework based on benchmarking data, we explicitly included *simulation* and *simulator* as search terms. Only a few publications were identified with a strong focus on simulating a FaaS offering. They also published data on how platforms react under different conditions which is a pre-

requisite for building a simulation model based on the data. The three included simulation papers are discussed in Section 6.2.2.2 in detail to keep the focus of this section on benchmarking approaches.

To motivate the following checklist in Section 4.2 and the research prototype in Section 4.3, a short summary of the presented SLR is given with the most important aspects for building a simulation framework.

Most of the papers included have some flaws in their data presentation. It is often unclear which configuration parameters were chosen. Furthermore, a lot of tools are not open-source limiting other researchers to investigate their benchmark and function implementation in detail. As observed by many researchers, the hardware stack within the same public cloud provider is different which has a direct effect on the execution time. Also multi-tenancy questions arise when not knowing the VM where the functions are executed on. A last important aspect from current research is the QoS assessment. A lot of experiments when using different memory settings focus on cost. When looking at a taxonomy for Serverless Resource Management [172], also latency is besides cost and resource efficiency an important goal. This motivates to investigate this tradeoff but also to keep latency QoS considerations in mind where FaaS customers might be willing to spent more money when latency drops, also when this does not fit the best performance/cost tradeoff.

4.2. Checklist for Performing FaaS Benchmarks

In the previous section, different benchmarking approaches were compared to argue which aspects are important for a reproducible and interpretable experiment design. A thorough documentation of open source research tools is the enabler for interpreting results and reproducing them [14] which motivates the following checklist. Each item is attached to some references, where details about the data extraction on the respective platform or the methodology is described in detail. As stated when writing about the flaws of some experiments, this checklist tries to help an experimenter to build a multi-dimensional data set for FaaS benchmarking research:

- **Physical Machine Configuration** - Obviously the physical machine and its performance influence measurements. A lot of experimenters do not include this information in their experiment description. The suggestion is to document at least the CPU model, the model number and the OS [166, 167]. Also the experimenter machine's performance could vary for different utilization levels where components of your benchmarking process are executed [185, 186]. A more detailed explanation of an inconsistent experimenter machine performance and a possible solution to fix this issue are presented in Section 5.

- **VM/Container Identification** - In the FaaS research area, most of the platforms are designed and make use of some virtualization layers, see Section 3.5 where a typical FaaS worker node architecture is depicted. To enrich benchmark data and enable an investigation of the multi-tenancy aspect, the executing VM has to be collected with runtime metrics. Unique strategies to identify the executing unit are important [282]. To identify the VM and CPU model, we use the `/proc/cpuinfo` and `/proc/stat` data from the shared file system of the Linux Host.

- **Function Configuration** - The function configuration is often omitted in papers as stated in the SLR. Since the resource scaling is determined by the configuration, i.e. the memory setting which influences the resource scaling as shown in Table 3.4, this information is vital for classifying the results [29, 185].

- **Function Runtime** - The programming language has a big influence on the execution time of the function when thinking about compiled languages like Java compared to interpreted languages like JavaScript [107, 141, 182, 189]. Also different runtime versions for the same programming language may vary in performance [143].

- **Data Measurement Procedure** - For the interpretation of the data, the measurement methodology is important to state in the experiment. As already discussed, the RTT is one metric to specify end-to-end latency but also contains noise due to routing and the platform middleware. We therefore propose a measurement methodology where each request submitted from an experimenter machine gets a unique identifier assigned. This identifier is further used within the payload of the request and attached to the execution logs on the platform. This procedure allows to map the local end-to-end RTT to the platform execution data where an assessment of network overhead and platform middleware can be made [139, 182].

- **Workload** - One best practice approach is to describe the workload and publish it with the corresponding data. This gives a first hint on the instance parallelism the FaaS platform has to handle [17, 139, 179].

- **Cold/Warm Distinction** - One of the major benefits of FaaS is scaling on demand. Since this results in function instance creation and cold starts when starting a container, it is important to state whether the functions faced a cold start or whether the function instance was reused [157, 182].

- **Multithreaded Implementation** - Due to the per-request execution model and the most fine-grained billing model on millisecond basis, the implementation of cloud functions directly influences the application/function characteristics. This is different to e.g. PaaS where single functionality

in the scope of a cloud function is hidden within a larger application/microservice. Therefore, the multithreading aspect when optimizing performance and costs of cloud functions is important to consider for FaaS experiments when using multi-core configurations [185, 293, 300]. This is different to long running applications in a PaaS area where multi-threaded implementations of pieces of functionality are suspended for code readability and maintenance. Avoiding multi-threaded code is also addressed by static code quality tools[117].

4.3. SeMoDe Web Application

We have implemented a research prototype called SeMoDe[118] to enforce the checklist when performing experiments and to comply with reproducibility efforts when conducting benchmarks. SeMoDe is a full stack app written in Java. PostgreSQL[119] is used as relational database solution, Spring Boot[120] as framework for the implementation of our application, Thymeleaf[121] as a server side templating engine and Open API Specification[122] to describe the implemented REST API. Figure 4.2[123] shows the different layers of the application. The prototype is also deployed at our university cluster where some of the experiments mentioned in later parts are available without user registration[124].

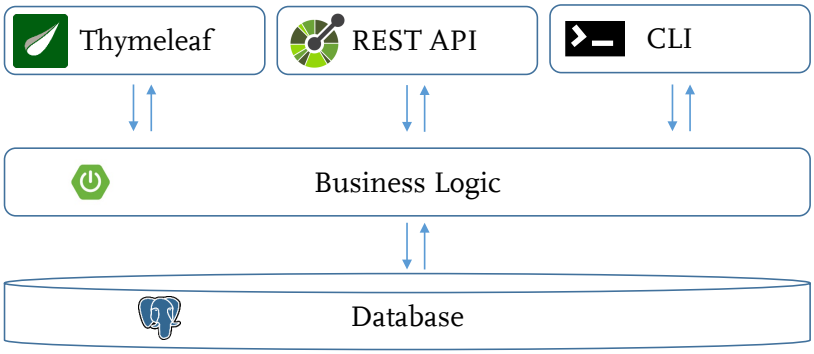


Figure 4.2.: Overall system architecture of the research prototype SeMoDe.

SeMoDe enforces the elements in the checklist but also supports developers to provide a proper documentation of their benchmark experiments which was often criticized when discussing the SLR papers. Based on a versioned setup configuration and a history in the database, each metric is associated with a setup and

[117] https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html#multithreaded-correctness-mt-correctness
[118] https://github.com/johannes-manner/SeMoDe
[119] https://www.postgresql.org/
[120] https://spring.io/projects/spring-boot
[121] https://www.thymeleaf.org/
[122] https://swagger.io/specification/
[123] Logos for Thymeleaf, OAS, Spring Boot, and PostgreSQL are from the corresponding tool sites.
[124] https://semode.pi.uni-bamberg.de

corresponding version. As stated, deploying cloud functions can be done via custom tools like the serverless framework or generic IaC tools like Terraform. One design decision of SeMoDe is to not rely on these tools and implement the deployment and monitoring of the function in one comprehensive tool. This gives the user the freedom to investigate special hypotheses and using the deployment information for starting the benchmark and getting the logs from different providers. It also supports the measurement procedure where uniquely identifiable requests are submitted and the logs of these executions are processed within the application and stored with the platform execution data. Furthermore, the simulation experiments are also conducted with the help of this prototype where an additional calibration step is explained in Section 5.

To understand the support SeMoDe provides, the database schema is introduced in Section 4.3.1. The aim of building a simulation framework is reflected in the database schema as well. Hence, benchmark related entities as well as simulation related entities are included. Therefore, the database schema gives a first explanation of important benchmark and simulation concepts. The package diagram explained in Section 4.3.2 gives a rough overview of the architecture of the prototype and highlights extension points when integrating new FaaS platforms. Three interaction mechanisms as shown in Figure 4.2 are described in Section 4.3.3. These three possibilities to access the capabilities of the prototype are described and their interaction is shown based on two common use cases when performing experiments with SeMoDe. Since one focus of this work is to get a solid foundation on benchmarking FaaS systems, special emphasis is put on the definition of different workloads, its submission within the prototype and the collected metrics during benchmarks in Section 4.4.

4.3.1. Database Schema

The database schema shown in Figure 4.3 visualizes all tables and their attributes. Primary keys are the first attribute of every entity and layouted in bold. The italic and bold-faced attributes are foreign keys. The cardinalities of the relationships are directed as implemented in the Java Persistence API (JPA) mapping within the entity classes in Java. They can be interpreted as follows:

A *setup configuration* is associated with exactly one *user*, whereas a *user* can have 0 to N *setup configuration*s.

The central table of the presented domain model is *setup configuration*. It is associated with a single *benchmark configuration* and a *calibration configuration*. We already know from Section 2.2 which elements comprise a benchmark. Within a benchmark configuration a user has to specify the workload (*benchmark mode* and *benchmark parameters*) and the function which should be deployed to the corresponding platform (*path to source*). Since SeMoDe supports deployment to several platforms, a user additionally specifies cloud provider or open-source tool dependent configuration parameters for deployment. Furthermore, after benchmarking a

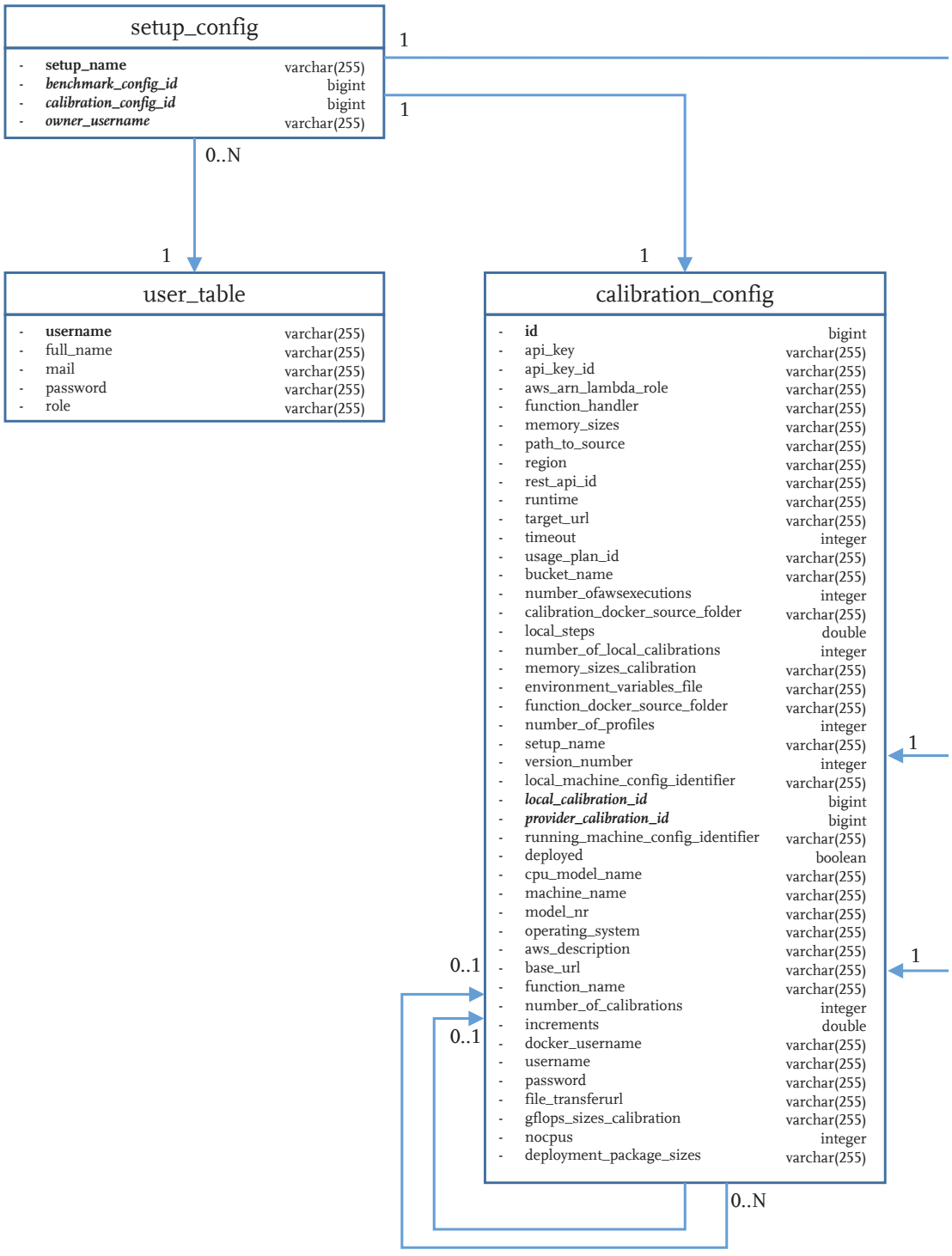


Figure 4.3.: Database schema of SeMoDe with a focus on benchmark and calibration entities.

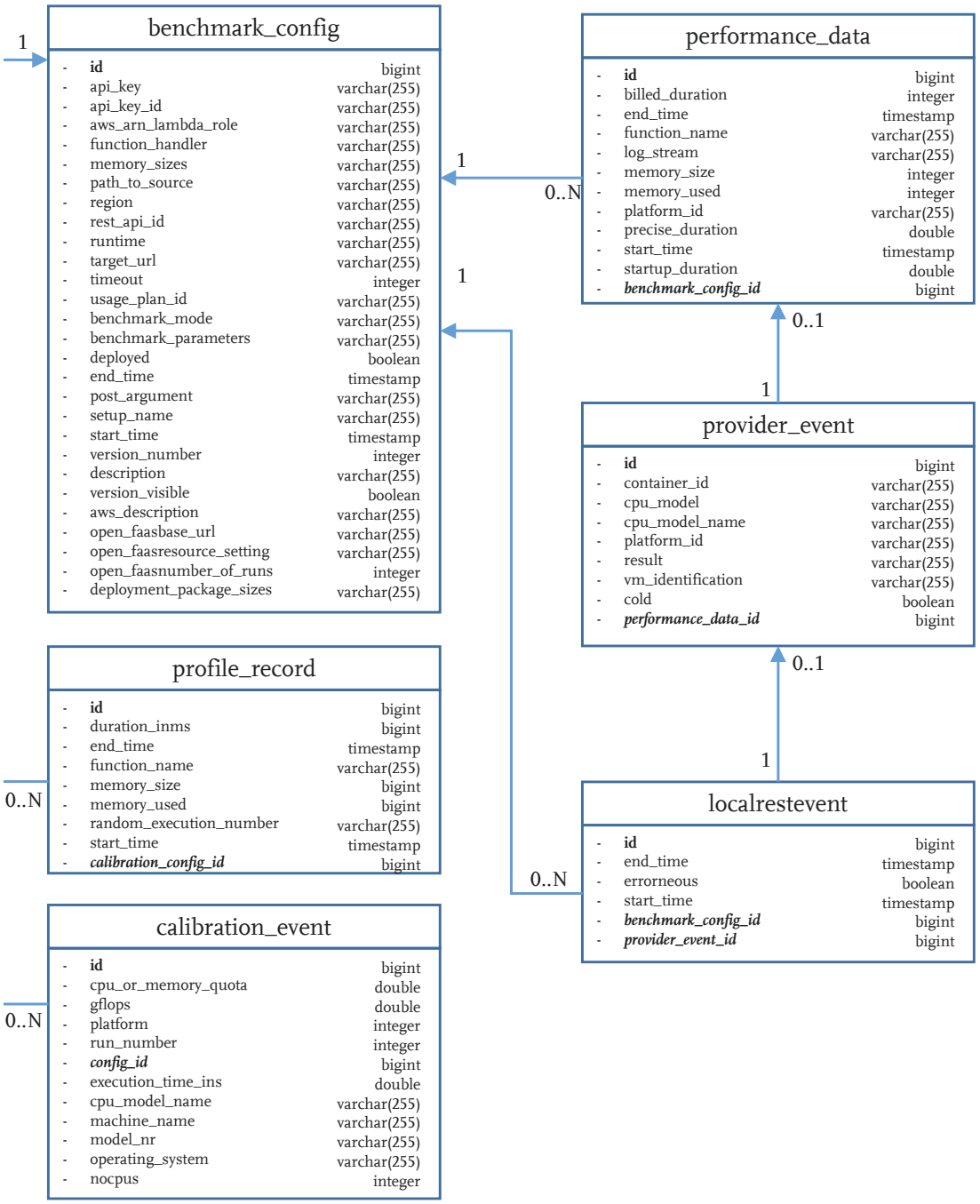


Figure 4.3.: Database schema of SeMoDe with a focus on benchmark and calibration entities.

platform, the logs are needed for a proper data evaluation where additional configuration parameters for the benchmark are stored to retrieve log data from the corresponding platform. All of this information is stored within a single *benchmark configuration*. Only the latest *version* of a configuration is editable. If a change occurs, the tool increments the version number which guarantees an unmodifiable documentation of the history.

Another reference from the *setup configuration* relates to a so called *calibration configuration*. The question now arises where are the simulation related classes and why is one of the most dominant entities in the data schema called *calibration configuration*. When looking at the dictionary again, to calibrate a tool means "to check the measurement on an instrument against a standard instrument, and adjust the first instrument to keep it accurate"[125]. Since we know from Section 4.1 that different public cloud providers and open-source tools scale their resources in different ways, this calibration is an approach to investigate the resource scaling strategies of providers. In addition, calibration data enables developers to make different systems comparable. This is the reason why the *calibration configuration* has two self-references. One is named *local calibration* and the other is named *provider calibration*. Both of them are needed to compare these two virtualized environments with each other, a more detailed explanation can be found in Section 6. The *profile record* table contains the simulation data, where an arbitrary function is executed based on calibration data. Conceptually, the `CalibrationConfig` has some references to other objects in our implementation to encapsulate specific aspects. For sake of simplicity, readability and speed of database retrieval of the experiment configuration, the `@Embedded` JPA annotation is used to store the attributes of these embedded classes in the same relational database table.

Both tables, *calibration configuration* and *benchmark configuration*, contain the *setup name* which enables a bidirectional relationship. This is not implemented in JPA but helps in optimizing retrieval operations for a single setup when using native queries, e.g. getting all versions for a specific setup by only querying a single table.

4.3.2. Package Diagram and Extension Points

This section explains the project structure and important files. It is a summary and overview of how the research prototype is constructed. A few concepts and classes which are important within later parts are already named but not discussed in detail. The package diagram depicted in Figure 4.4 is explained from the perspective to write connectors for new FaaS providers. Under `src/main/resources` Spring Boot's `application.properties` can be found where the settings for all properties, e.g. database connection, logging levels, custom environment variables etc. can be changed. The subfolder `templates` within the resources folder contains the Thymeleaf HTML templates and `static` content like CSS, images and Java-

[125] https://www.oxfordlearnersdictionaries.com/definition/american_english/calibrate

Script code. The application code is located under `src/main/java` predefined by a gradle convention.

Emphasis is put on classes which are especially important for the architecture of the prototype. Design decisions and extension points are highlighted. The main package `de.uniba.dsg.serverless` contains the classes for starting the SpringBoot application. Since a Spring web application was implemented, a web server, per default Tomcat, is started. To suppress the startup of the web server and use our prototype as CLI tool as shown in the overall system architecture in Figure 4.2, an `ArgumentProcessor` class is implemented which needs another system property set at runtime. See the comment within the argument processor class for further information on how to avoid starting the web server.

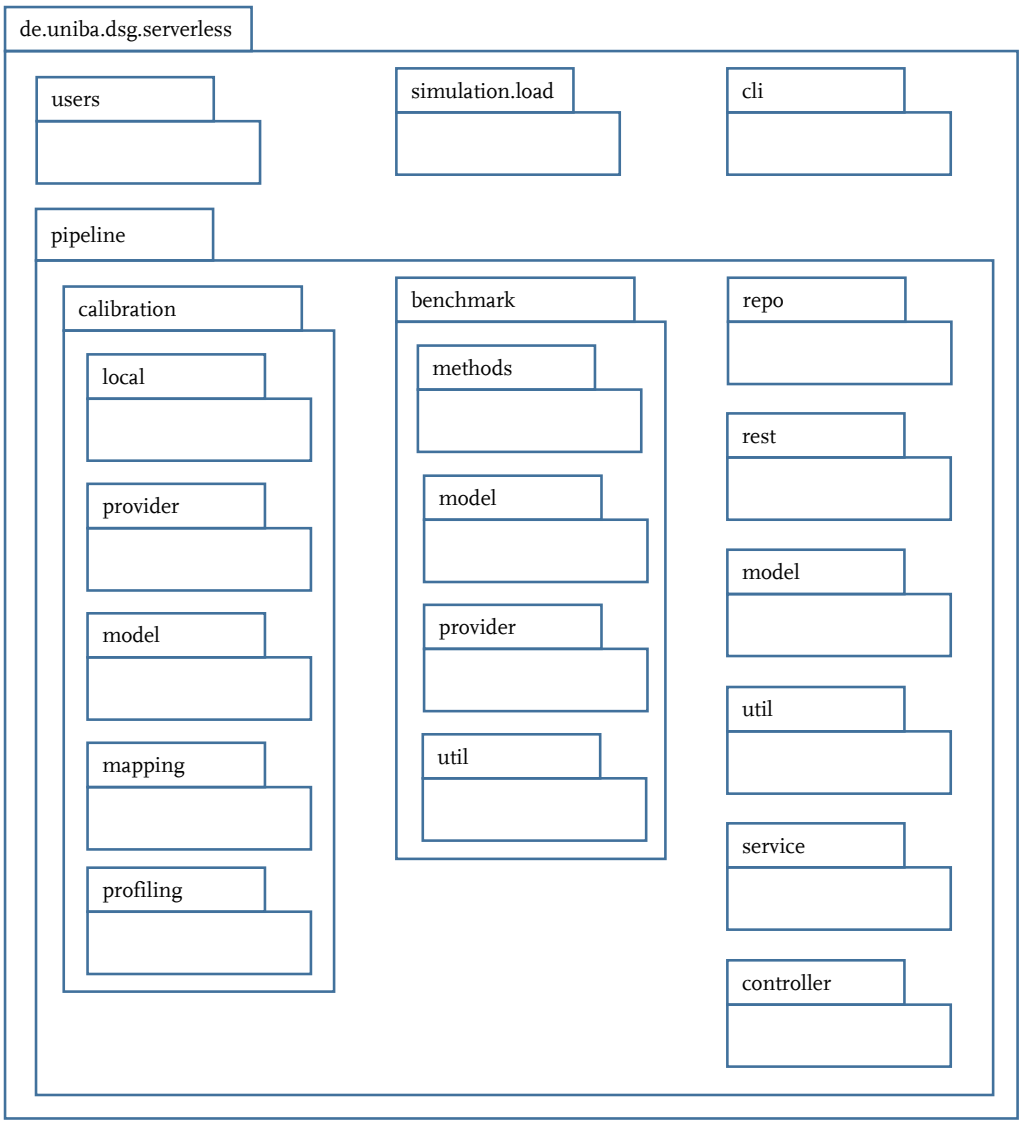


Figure 4.4.: UML package diagram of SeMoDe.

All other classes are included in four packages `simulation.load`, `users`, `cli` and `pipeline`. The first subpackage `simulation.load` contains the implementation of a static simulation approach published in 2019 [179] which is similar to

discussed work [165] in the SLR. When given an arbitrary load pattern and a simulation input (average execution time, cold start time and container warm period), the tool computes the number of cold starts and running instances at a given point in time. The `users` subpackage contains the model, controller and services for the user management of the application. Furthermore, it contains a session-based security configuration. The REST API on the other hand, is secured on a request basis where a user has to authenticate upfront and needs to request a JSON Web Token (JWT). The `cli` package contains a `UtitlityFactory` class where a builder pattern is implemented. Each `CustomUtility` has an attribute name which is one Command Line Interface (CLI) parameter to specify which functionality should be executed. This is also a mechanism to extend the prototype and enable developers to test some functionality before integrating it into the pipeline as well as to provide standalone features.

The most important package is `pipeline`. It contains the business logic for the benchmark and simulation framework. Before talking about these two packages, a short description is made for other relevant subpackages. All JPA related interfaces are included in `repo`. Spring Data JPA annotations are used where Hibernate is the Object Relational Mapping (ORM) implementation. It generates the queries at compile time based on a declarative, human readable approach where keywords known from SQL are combined within interface method names like `findByLastnameAndFirstname`[126]. Where this naming scheme is insufficient for expressing queries, we use native queries and interface based Data Transfer Object (DTO) projection[127]. In our `model` package, all configuration related classes are included, see Figure 4.3. The `benchmark.model` and `calibration.model` classes contain specific business related model classes which are necessary for performing benchmarks respectively calibrations. The packages `controller` and `service` contain classes which are annotated with Spring stereotypes of the same name. The architectural decision is to have no direct access from a controller to a repository class, so a controller class has only dependency injected service objects, but no repository objects. Furthermore, the service classes do not contain any state since the only source of truth is the database. Therefore, scaling and concurrency issues introduced by the middleware do not occur in `SeMoDe`. The `util` package includes the custom `SeMoDeException` as well as some static helper classes and Spring beans. As already mentioned in the introduction to this section, we also expose a REST API implemented in the `rest` package. This package also contains subpackages for `controllers`, `dtos`, `services` and a `security` implementation via JWT.

The remaining two packages `calibration` and `benchmark` implement the centerpiece of `SeMoDe`. The prototype contains of a few generic interfaces which are extension points to enable the support for different FaaS platforms and establish a generic interface to deal with their heterogeneity. Two of them are `Benchmark-`

[126] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

[127] https://www.baeldung.com/spring-data-jpa-projections#interface-based-projections

Methods and LogHandler in the benchmark package. The first is included in the methods subpackage, where currently only AWS is supported. This interface defines methods for deploying the functions to the platform, starting the benchmark, fetching data from the corresponding logging service and undeploying cloud functions to keep classes like the BenchmarkExecutor vendor independent. The latter is included in provider package where currently AWS CloudWatch is supported for investigating the logs and enable a vendor specific casting of the log data to generic *performance data* entries, see the database layout for further information. The benchmarks for the only open-source supported platform OpenFaaS is executed based on shell scripts.

The third extension point is the interface CalibrationMethods in calibration.provider. This mechanism is similar to the benchmark extension at BenchmarkMethods. Here, AWSCalibration, LocalCalibration and OpenFaaSCalibration implement this interface to deploy, start and undeploy the calibration. A local deployment is of course not necessary since we directly execute a Docker container to get the *calibration events*. The other subpackages in pipeline deal with different steps in the calibration and simulation pipeline. The statistical evaluation of the tool with linear regression models, computation of the coefficient of determination (R^2) and a mapping (MappingMaster) to compare different environments is included in mapping. Finally, profiling as the last subpackage of calibration, includes the classes for the local execution and simulation of a function based on the mapping information from the computed linear models.

After discussing extension points and configuration options from a developer point of view, the next section introduces the interaction mechanisms as shown by the system architecture of SeMoDe in Figure 4.2.

4.3.3. Interaction Mechanisms

4.3.3.1. Web UI

The web UI is designed to be the only possibility within SeMoDe to create new setups and change existing ones. When starting a new experiment and after registration, a *user* has to create a new *setup configuration*. Users of the application are categorized in two user groups: On the one side they can have the role USER limiting their capabilities to their own experiments, whereas on the other hand users with the role ADMIN can see and change the setup configuration of all users and their related entities. These are the only roles currently defined in the prototype. Figure 4.5 comments on the setup configuration overview page and shows the restrictions for ordinary users and capabilities for unauthenticated users which can look at publicly available benchmark information for selected configurations.

Figures 4.6 and 4.7 show the screenshots for the setups/{setupName}/benchmark[128] endpoint. The first screenshot contains all editable input fields to config-

[128] A curly bracket semantics is used in the style of specifying controller paths in Spring Web for dynamically changing fields.

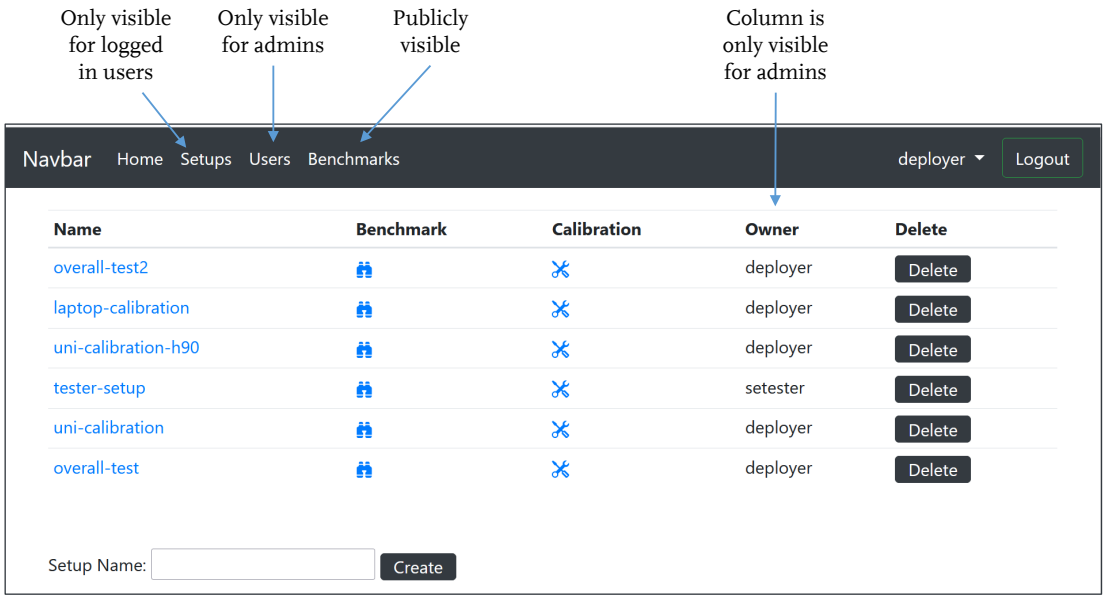


Figure 4.5.: Setup configuration Web UI of SeMoDe showing aspects of the implemented user management.

ure the benchmark. Via the *Versions* dropdown, a user can see the configuration of older versions. Technically, asynchronous requests via JavaScript (*AJAX*) are made to an endpoint to get the benchmark configuration based on the selected version. Only the latest version is editable for the input fields, in contrast to the description and version visible fields, which are editable for every version. The dropdown information is structured as follows:

"{version} - Platform events: {performance data entries} (database id)"

The number of performance data entries in the database indicates if a benchmark was executed based on this version and if the data was fetched from the corresponding provider. The performance entries and especially the attributes *memory used* and *precise duration* are later used for displaying the data graphically as can be seen in Figure 4.7. The primary key of the benchmark configuration is also displayed in parenthesis for an in-depth investigation and custom queries when accessing the database as an administrator. A custom *Description* is the next input field. This can be used to make notes about the configuration during the experiment planning and execution, but also for publicly marked experiments when the field *Version Visible* is set. Then a non-authenticated user can see a subset of the information presented here, more details about the publicly available benchmarks can be found on our website[124].

The following three fields are connected to each other. Different *Benchmarking Modes* are implemented to support clean test beds as well as arbitrary load patterns. Further details on the benchmarking modes implemented and the load pattern generation can be found in Section 4.4. Since the prototype is currently only capable of invoking REST endpoints, we added some *Post Body Arguments* in JSON format as input for the cloud function. When a benchmark is executed, the

experiment start and end time is set, displayed at *Experiment Time* and overwritten with null values when the next version is created and stored in database.

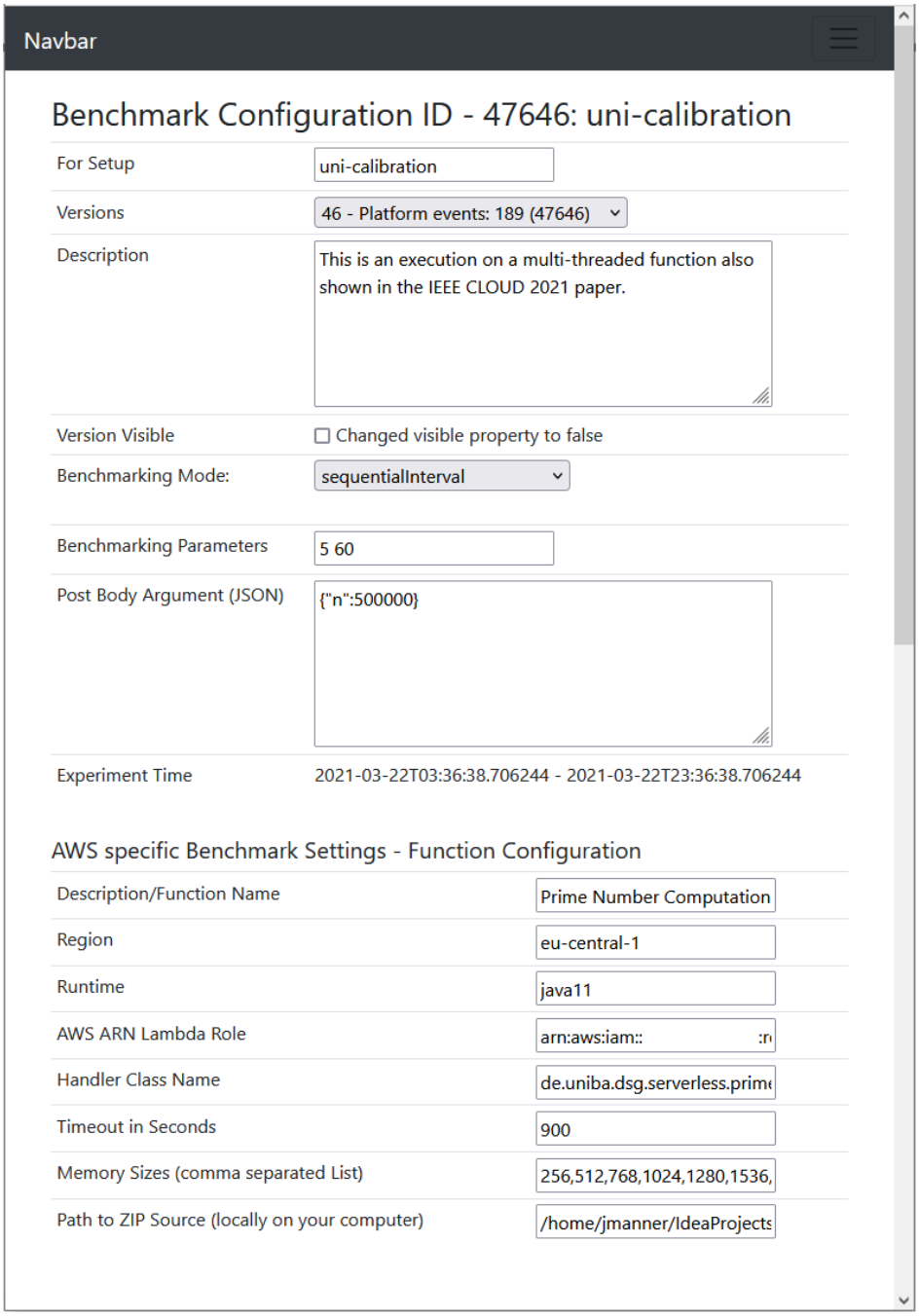


Figure 4.6.: Benchmark Web UI (I/II) to specify general information and AWS specific settings.

As mentioned before, currently only AWS and OpenFaaS are supported. Therefore, in the next part of the website, the AWS specific function configuration is set. The first fields are self-explanatory, where the *Description/Function Name* is added to the overall description. This enables the user to describe the function in this context. All other fields are similar to the fields at the AWS dashboard. There you can also use the ZIP upload option. *Deployment Internals* are the next part and

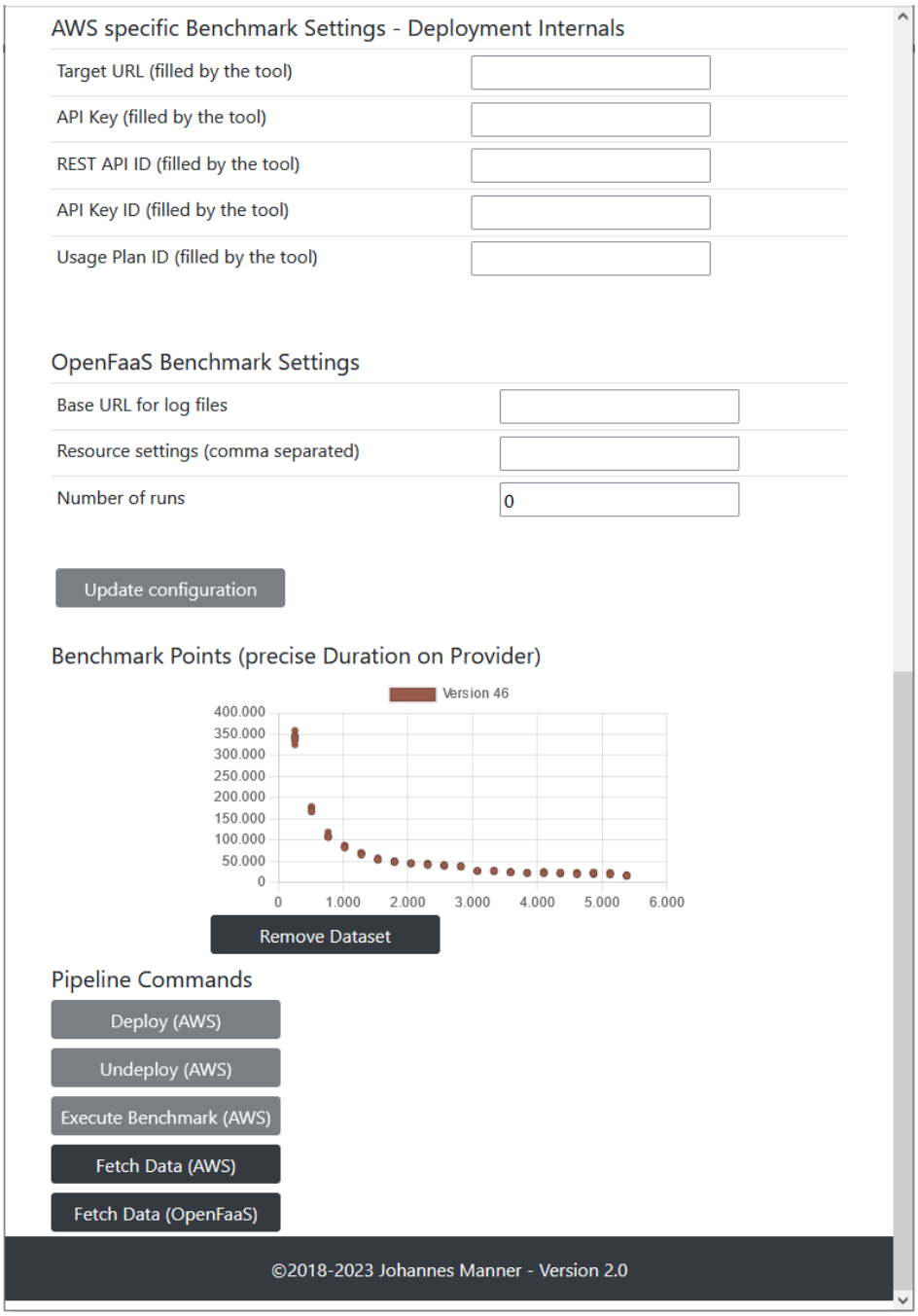


Figure 4.7.: Benchmark Web UI (II/II) plotting benchmark experiments and providing pipeline commands.

completely filled and cleared by the tool, see Figure 4.7. This procedure enables a check of specific parameters of the corresponding provider and allows for possible intervention by the experimenter when some components malfunction. It is furthermore needed to undeploy the deployed function at a later point in time. Via the *Update Configuration* button, a user of the tool can store a new version of the configuration. Whenever a version with associated performance data is selected, a new set of data points is included in the diagram where the last version data can also be removed via the *Remove Dataset* button. Also the actions *Deploy*, *Un-*

deploy and *Execute Benchmark* are disabled. *Fetch Data* from a provider's logging service like AWS CloudWatch is possible when an experimenter forgot to do so, but already fetched data cannot be fetched again.

4.3.3.2. Command Line Interface

Since parts of the overall benchmarking and simulation pipeline take hours or even days to complete, a CLI feature was implemented to run the application on the corresponding machines. The command in Listing 4.1 shows the invocation of the java application. It is vital to set the property `spring.main.web-application-type` to `NONE` here to suppress Spring's default behavior to start a web server. The & detaches the process from the console and via `> out.txt` the standard output is written to the file `out.txt`. This enables an error handling and investigation of details after the benchmark when running the prototype as a background process.

Listing 4.1: Start SeMoDe as CLI application.

```
1 $ java -jar -Dspring.main.web-application-type=NONE app.jar & > out.txt
```

4.3.3.3. REST API

As mentioned in Section 4.3.2, the basic security configuration is session based. For the REST API, another security `Configuration` class (`SecurityConfig.ApiSecurityAdapter`) is implemented to handle requests and use JWTs[129]. Therefore, a custom filter class (`JwtAuthenticationTokenFilter`) was implemented and registered in the pipe and filter security architecture of SpringBoot Security. A user has to send an `AuthenticationRequest` to the endpoint `/api/login` with the username and password. The tool generates a `JWTTokenResponse` with a Bearer token. In the following, the user has to include this token in every request as an authorization header. After some time (specified via the `jwt.expiration` property), the token expires and the user has to resend an authentication request to get a new token.

Figure 4.8 shows the Swagger UI displaying the OpenAPI Specification (OAS) of our prototype. It highlights, indicated by the lock, the authorization needed as described in the previous section. At the `/swagger` endpoint, the interested reader can access OAS and the benchmark configurations which are publicly available via the `/benchmarks` endpoint. The REST API is an additional option to access the data and at an early stage.

4.4. Invoking Cloud Functions

The previous sections gave a first impression on the structure of the prototype as well as on the capabilities of benchmarking, calibration, simulation and their

[129] https://datatracker.ietf.org/doc/html/rfc7519

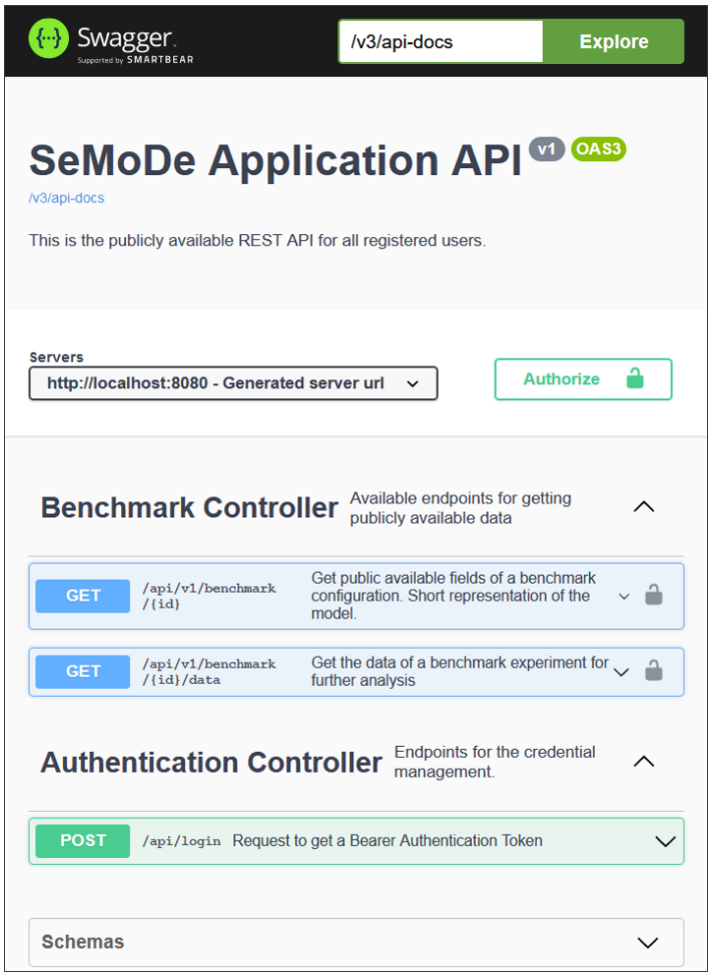


Figure 4.8.: OpenAPI Specification for exposed REST API.

configurations. To be able to make use of the prototype and to comply with its interfaces, a few things are important for the configuration of a benchmark experiment and its execution. Within the next Section 4.4.1, an introduction to the cloud function response is given, in particular which data has to be harvested on the platform to enable a proper data evaluation. An important aspect for multi-tenancy is the number of parallel requests which is directly influenced by the workload specification discussed in Section 4.4.2. Finally, the technical realization on submitting requests and discussing the measurement methodology in Section 4.4.3 concludes this chapter on benchmarking FaaS platforms.

4.4.1. Cloud Function Implementation

Each FaaS platform has a proprietary handler interface. A developer of a cloud function has to comply with these interfaces. A contribution on how to harmonize several interfaces is made in our work on *Cloud Function Lifecycle Considerations for Portability in Function as a Service* [96]. In the course of our prototype, we focused on HTTP calls, the predominantly used trigger for cloud functions. There is no additional constraints on the handler interface or the request structure made by

the prototype. We use standard HTTP POST requests to trigger the functions, but the response from the cloud function has to match the `ProviderEvent` structure to be parsed correctly by our implementation. This custom response includes all fields of the `provider_event` table or `ProviderEvent` class except for the id field, see the database schema in Section 4.3.1.

Listing 4.2: JSON response from a cloud function.

```
{
  "cold": true,
  "result": "12",
  "platformId": "4c609f17-a2c3-464b-b07e-075a20d40fdb",
  "containerId": "b7a54352-717b-4fec-8b85-65f0e0f220e8",
  "vmIdentification": "1680524966",
  "cpuModel": "63",
  "cpuModelName": "Intel(R) Xeon(R) Processor @ 2.50~GHz"
}
```

Listing 4.2 shows an exemplary response from a cloud function. We refer to the discussed approaches in the SLR to differentiate VMs and containers on different providers like [282] as well as using static attributes to see, if a new instance was started (attribute cold is true) or if an already provisioned function is reused (cold is false). Especially the CPU information is important to deal with the heterogeneity of hardware at different regions of a FaaS provider as researchers have documented performance variations in several experiments [50, 180, 206, 213].

4.4.2. Workload Specification within SeMoDe

As already shown in Figure 4.6, there is a dropdown for selecting the *benchmarking mode*. The tool supports five benchmark modes. The first four modes are artificially created to draw strong conclusions when conducting experiments, i.e. understanding the start-up of function instances, the influence on cold starts on the execution duration etc. The last option is for real world traces as described in Table 4.4. For each mode, the mandatory attributes are described and can be configured in the *Benchmarking Parameters* input field.

To have a generic data interface for submitting the workload, `SeMoDe` generates a csv file for the first four modes via the `LoadPatternGenerator` class. This file is then located in the folder `setups/setupName/benchmark/loadPattern.csv` and overwritten when further benchmarks are executed based on the same configuration.

4.4.3. Submitting Requests

So far, we discussed how to implement the SUT in Section 4.4.1 and to specify the workload in the prior section. What is missing as introduced in the benchmark Section 2.2 are metrics for a proper comparison.

Table 4.4.: Benchmark Mode and Benchmark Parameters to specify a custom workload

Benchmark Mode	Description	Benchmark Parameters
concurrent	Invoking the function under test once in parallel by executing a number of concurrent requests (NR).	NR: Number of requests.
sequentialInterval	Sequentially triggering a function with a fixed time interval (FT) between the starting point of the corresponding function invocations. The idea is to use only a single function instance in the cloud and execute a number of requests on that instance (N). This could lead to insights on JIT compilation capabilities of the platform or other optimizations made. Also unknown multi-tenancy aspects over time could be detected as well as clean-up periods, for example when the cloud function instance is terminated due to a termination of the parent VM. When choosing the parameters in a way that execution time of the function is greater than the interval, a sequential execution of the requests is not guaranteed. In such a case, the platform typically instantiates further cloud functions to serve the requests.	NR: Number of requests. FT: Seconds between request start times.
sequentialConcurrent	Combines the previous two invocations modes. The idea behind this mode is to assess the multi-tenancy and scaling properties of a FaaS platform. Based on the VM identification, an experimenter sees which requests are executed on the same VM respectively if the FaaS platforms create further instances or schedules the requests in a queue. The tool starts a number of requests (NEG) in parallel. After a specified time delay (DbG), the tool starts another set of requests (NEG) as often as specified by the experimenter (NoR). Here insights are possible again on the reuse of cloud function instances or if the FaaS provider deploys instances of the same customer on the same VMs.	NEG: Number of execution groups (parallel cloud function instances). NoR: Number of requests in each group (see sequentialInterval mode). DbG: Delay between start of group g and start of group g+1 in seconds.
sequentialChanging-Interval	This mode starts a number of requests (NR) in varying intervals between execution start times in a round robin fashion (LD). The reason behind this setting is to investigate serial or bursty patterns and their effects on FaaS platforms.	NR: Number of requests. LD: List of delays.
arbitraryLoadPattern	The function endpoints are triggered based on a csv file (FP). The file contains only one column and for each row a double value with a relative timestamp compared to the actual time when the generation is started, e.g. 0,0 means now and 5,0 means now + 5 seconds. This value indicates when a specific REST call, invoking the cloud function at the platform, should be submitted.	FP: File path of the csv load pattern file.

A typical and common measure assessing applications from a customer point of view is latency. To breakdown the latency in several parts, the start and end time on the user side are logged as depicted in Figure 4.9. Shown as orange boxes, these periods contain the transmission over the network, the scheduling on the FaaS platform and finally the execution plus returning the response. Which aspects are included in the metering service of the corresponding FaaS platform (blue box) is provider dependent and may vary significantly between providers. To have both measures, the user perceived latency and the provider's execution time, an experimenter can calculate the overhead which scheduling, network transmission etc. introduces. For latency critical applications, network latency is still an important aspect to consider.

When we start a request, we locally log the start and end time as well as a unique identifier (UUID). Those pieces of information, together with the result of the plat-

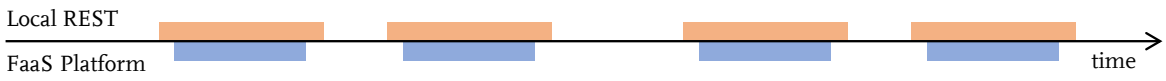


Figure 4.9.: User and platform perceived performance when executing a cloud function.

form (returned HTTP body from the API invocation, see Section 4.4.1) are stored in the tables `localrestevent` and `provider_event`. When we fetch the data from the metering respectively logging service of the corresponding provider, we analyze it and store the execution metrics in our `performance_data` table. Based on the unique identifier `platform_id` in `provider_event` and `performance_data`, the prototype is capable of matching the local REST event and the provider execution data from the specific log handler and associate the provider events with the corresponding performance data. In cases where, for example, the API gateway timeout is reached or unforeseen errors occur, no local rest events respectively provider events exist which would limit the possibilities to investigate the client perceived overhead. However, the data for investigating the platform performance at the provider can be stored in either way. This is important to keep in mind when analyzing experiments.

Since we are not using an established load generation and submission tool due to this tight integration on response data and its inter-relation, the question arises how to submit unknown workloads without concurrency limitations. Listing 4.3 contains the centerpiece of the benchmark invocation process. Since the prototype generically implements the `BenchmarkMethods` interface, a list of deployed methods with their endpoints is passed to this function. For the submission of requests, two executor services were used which handle the concurrent execution of tasks via thread pools. The reason for these two executors in line 5 and 6 is, that we want to schedule requests based on our generated load pattern (Benchmark Mode: arbitraryLoadPattern). This is only possible with a `ScheduledExecutorService`[130], but this executor service is limited in the number of concurrent threads which are determined by the factory method's `corePoolSize` attribute. Since we do not know the concurrency level a priori due to the arbitrary load pattern and function execution times, the core pool size cannot be determined. Another problem of using the scheduled executor is, that there is no runtime error when the scheduled requests exceed the core pool size. As given by the architecture of an executor service where the task submitters and the threads are decoupled by a queue, the requests are queued by the work queue of the thread pool and not executed at the time specified by the load pattern. Since `SeMoDe` requests the cloud function in a synchronous way, the executing thread blocks (IO wait).

[130] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ScheduledExecutorService.html

Listing 4.3: Centerpiece of the `BenchmarkExecutor` class.

```
1  public List<LocalRESTEvent> executeBenchmark(final List<BenchmarkMethods>
       benchmarkMethodsFromConfig) {
2
3    final List<Double> timestamps = this.loadLoadPatternFromFile();
4
5    final ScheduledExecutorService executor = Executors.newScheduledThreadPool(N);
6    final ExecutorService delegator = Executors.newCachedThreadPool();
7
8    final List<Future<LocalRESTEvent>> responses = new CopyOnWriteArrayList<>();
9
10   long tmpTimestamp = 0;
11   // for each provider
12   for (BenchmarkMethods benchmarkMethods : benchmarkMethodsFromConfig) {
13     if (benchmarkMethods.isInitialized()) {
14       // for each function
15       for (String functionEndpoint : benchmarkMethods.getUrlEndpointsOnPlatform())
             {
16         // for each timestamp
17         for (double timestamp : timestamps) {
18           tmpTimestamp = (long) (timestamp * 1000);
19           FunctionTrigger f = new FunctionTrigger(benchmarkMethods.getPlatform(),
               benchmarkConfig.getPostArgument(), new URL(functionEndpoint));
20           FunctionTriggerWrapper fWrapper = new FunctionTriggerWrapper(delegator,
               responses, f);
21           executor.schedule(fWrapper, tmpTimestamp, TimeUnit.MILLISECONDS);
22         }
23       }
24     }
25   }
26
27   // Shut down the first scheduled service. This means that all wrapper function
28   // trigger tasks are run and the function trigger tasks are submitted.
29   // Time to wait is the last timestamp from now on executing a function.
30   this.shutdownExecService(executor, tmpTimestamp + GATEWAY_TIMEOUT);
31   // Wait for the function trigger tasks to terminate
32   this.shutdownExecService(delegator, tmpTimestamp + FUNCTION_TIMEOUT);
33
34   // collecting the custom responses from the cloud platform
35   List<LocalRESTEvent> events = new ArrayList<>();
36   for (Future<LocalRESTEvent> futureEvent : responses) {
37     events.add(futureEvent.get());
38   }
39   return events;
40 }
```

This is the reason why `executor` is only used for asynchronously starting requests in a fire and forget manner. The synchronous execution is done by a `CachedThreadPool`[131] named `delegator` in the snippet which dynamically scales the number of threads in its thread pool. In line 19, the function trigger, a callable which synchronously executes HTTP requests, is created. This callable is wrapped in a runnable in line 20, which also gets a reference to the delegator (the autoscaling executor service) as well as a reference to the thread-safe list where the

[131] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html#newCachedThreadPool()

`Futures`[132], a handle to get the result of the computation later on, are collected. In line 21, the `executor` schedules the wrapped function triggers. Dependent on the start time of the scheduled callable, it executes a call to the delegator which executes the `FunctionTrigger`. After submitting the last request, the scheduled executor service terminates immediately in line 30. The call in line 32 blocks until the last synchronous invocation returns. It waits the timeout period for the cloud function to be sure that all functions can terminate correctly on the selected platform within the specific provider function timeout. After the execution, the futures and especially the results are added to the `events` list in line 37. In the following they are stored in the database.

`SeMoDe` is dependent on the system resources and the number of threads a system can handle. Another limiting factor is the amount of memory the system provides for the number of concurrent functions which can be executed in parallel. In this aspect, the implementation is similar to other load testing tools like JMeter[37] also implemented with Java. These limitations may be overcome by the usage of Project Loom[133] which is currently a preview released feature in Java 19. Loom implements virtual threads which have a similar memory and CPU footprint like ordinary Java objects. Therefore, this tool is predestined for IO intensive workloads like the one introduced in this section, waiting for the response of the cloud provider and is capable of submitting millions of requests.

[132] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Future.html

[133] https://openjdk.org/projects/loom/

5. Calibration of a Consistent Resource Scaling on a Developer's Machine

Parts of this chapter have been taken from [174, 181].

In this chapter, RQ3.1 *(How can a consistent CPU scaling behavior across various processors and scaling algorithms be achieved and visualized?)* is supported.

This section deals with an important aspect to make resource scaling on a machine visible. Based on the improvements in hardware, there are various requirements for specific situations which influence the performance and frequency scaling of the CPU like heat limits, different power consumption under different load conditions etc. For consistent experiments, an understanding of the used CPU frequency algorithms is necessary to select the right one. Therefore after a motivation, we recap some important fundamentals for P-states and the CPU frequency scaling options in the Linux kernel in Section 5.2. The next section discusses related work of benchmark characteristics and how P-states were addressed in other research. Section 5.4 describes the motivation of our work in greater detail and continues with the presentation of the problem. The next section proposes our methodology to answer RQ3.1 followed by an evaluation in Section 5.7. The discussion of the results and limitations of the proposed calibration is presented in Section 5.8. Finally, we conclude this section with ideas as well as next steps for future work.

5.1. Motivation

As already discussed in Section 3, different public cloud providers have several scaling strategies for the assignment of resources within the FaaS domain. When looking at even more generic offerings from public providers, VM offerings like Amazon EC2 or Azure Virtual Machine have several instance types with different resource sets assigned to them. To select a proper configuration, developers are for example guided on Azure with a rough performance estimation on an abstract measure called the Azure Compute Unit (ACU)[134]. It enables a comparison of different VM solutions based on their relative performance to their smallest VM configuration.

[134]https://learn.microsoft.com/en-us/azure/virtual-machines/acu

To the best of the author's knowledge, there is no notion of such an abstract computing measure for local environments introduced by a benchmark in the presented SLR nor in another experiment in research. As remarked, most of the experiments do not state the machine configuration at all. Therefore the question arose during the research efforts how to perform experiments in a comparable way on a local testbed being able to compare absolute values with the cloud offerings. A prerequisite we identified is to find such an abstract measure to approach the public cloud platform as well as the developer's machine to make them comparable. The strategies for assigning resources like for AWS Lambda or Huawei Cloud FunctionGraph are proportional to the memory setting. This introduces another challenge since a single abstract measure is not sufficient to compare different deployment alternatives. Rather a function must be defined to show the increase in computing power by increasing resources. Since most experiments and algorithms assume linearity of this assignment which is the best understood strategy [153], the same approach is chosen while building the simulation framework locally on a developer's machine.

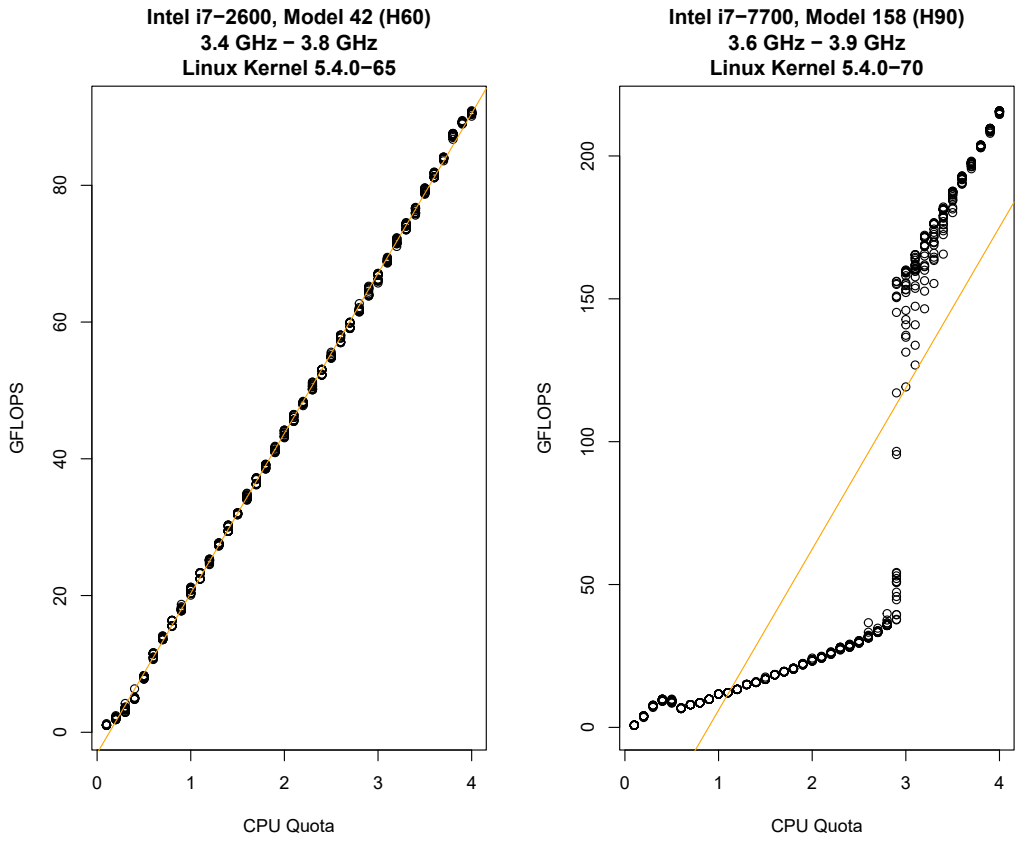


Figure 5.1.: Executed calibrations showed an inconsistent resource scaling on two local Ubuntu servers.

While conducting research and experimenting with different possibilities to compare hardware stacks with each other, we faced some unpredictable and non linear performance distributions on an Intel i7 processor shown in Figure 5.1. On

the x-axis, the used CPU resources assigned to the executed function is shown on our two quadcore machines. On the y-axis, an abstract compute measure, in particular Giga FLoating Point Operations Per Second (GFLOPS) is used to assess the performance at several utilization points. The orange line is a linear regression computed which is discussed in detail within the problem analysis Section 5.4. One of our machines showed three performance ranges due to power saving aspects (right diagram in Figure 5.1) whereas the other, an older machine, showed a consistent scaling of resources (left diagram).

There are many settings influencing each other and ultimately the performance of the CPU making it hard to trace performance variations back to a single factor. DELLs configuration of HPC servers [21, 129] or the SPEC bios settings descriptions[135] for their CPU benchmarks are examples for the plethora of configuration options. Simply disabling all powersaving or performance boost options is not an option since the default settings do not ensure a linear scaling. It is likely that also other investigations face this non linear performance distribution on machines used in their experiments without detecting it due to noise in the benchmarking data. We therefore called the published work which is the basis of this chapter, *why many benchmarks might be compromised* [181].

The problem of badly documented and therefore non-repeatable experiments gets even worse, when a series of experiments does not state hypotheses upfront and does not start with single, isolated experiments to confirm or reject these hypotheses before conducting load tests. Tools like JMeter allow to stress test a SUT, which can lead to predictions on how the system will behave under heavy load. CPU and other hardware resources are strained but often the implications of the hardware used is neglected [23, 50, 206]. Since applications are running on different hardware within the software lifecycle, e.g. there is a high likelihood that the test setup is different from the production setup, a mismatch of the runtime behavior and test results may appear. It is important to be aware of this fact when configuring one system based on the measured QoS of another system. General purpose processors, i. e., consumer processors, focus on optimizing the average-case performance and run in energy efficient modes by employing runtime performance enhancements. However, these techniques are normally not documented since they are Intellectual Property (IP) of the vendor. End users have limited control over them. Frequency scaling of the CPU is influenced by a lot of factors specified in the Advanced Configuration and Power Interface (ACPI) specification [268]. In particular, performance states (P-states), cooling requirements and turbo boost options influence the frequency scaling, power consumption and heat generation of the CPU. However, for load tests a linear scaling of resources is important for interpretable and fair results.

Therefore, we propose a simple function to calibrate hardware and understand the scaling algorithms determining CPU performance in order to allow for fair benchmarks. We calibrate the developer's machine for several artificial utilization

[135] https://www.spec.org/cpu2017/flags/Intel-Platform-Settings-V1.1.html

points. A running Docker environment is the only prerequisite. Our calibration approach can serve as a standard for normalizing results of benchmark experiments. This leads us to answer RQ3.1: *How can a consistent CPU scaling behavior across various processors and scaling algorithms be achieved and visualized?* We limit the discussion to Linux[136] and Intel CPUs[137] since they are the predominant technology used in the cloud.

5.2. Fundamentals

According to the ACPI specification[138], Control States (C-states) and Performance States (P-states) determine power consumption and frequency of the CPU. C-states handle the working mode of the machine, e.g. active or suspended. Since we assume an active one, C-states will not be considered in the following. P-states determine the CPU frequency and therefore the computational power. They can be changed by algorithms when demand changes. There are a few conflicting goals which need to be addressed when changing the P-states [268]. On the one hand, performance and on the other hand, power consumption, battery life, heat generation and fan noise. The options listed in the following are the implementation in the Linux kernel, where the CPU Frequency scaling (`CPUFreq`) subsystem consists of three components[139]:

- **Scaling Governors** - Each scaling governor implements an algorithm for reacting to changing computing demands and changes the CPU's frequency accordingly. Also mixed strategies where different scaling governors work together are reasonable to achieve a good system performance under various loads. Examples for governors are `performance` (highest frequency) or `powersave` (lowest frequency).
- **Scaling Drivers** - "Provide scaling governors with information on the available P-states (or P-state ranges in some cases) and access platform-specific hardware interfaces to change CPU P-states as requested by scaling governors."[140]
- **`CPUFreq` Core** - Basic code infrastructure framework which the other two components integrate with.

Since the scaling driver communicates with the hardware, vendor-specific options cannot be addressed by a generic implementation. Therefore, the integra-

[136] Market share Linux: https://www.rackspace.com/en-gb/blog/realising-the-value-of-cloud-computing-with-linux
[137] Market share Intel: https://www.statista.com/statistics/1130315/worldwide-x86-intel-amd-laptop-market-share/
[138] Especially Section 8 in the specification is of particular interest (pages 509ff. in [268])
[139] https://www.kernel.org/doc/html/v5.4/admin-guide/pm/cpufreq.html
[140] https://www.kernel.org/doc/html/v5.4/admin-guide/pm/cpufreq.html#cpu-performance-scaling-in-linux

tion of vendor-specific scaling drivers into the Linux kernel was introduced. Since SandyBridge (second generation of Intel processors), intel_pstate is such a scaling driver making it possible to also implement own scaling governors or overwriting existing ones. This driver circumvents the generic implementations and also adds new features where Hardware-Managed P-states (HWP) enable customized scaling algorithms to deal with specialties of each processor family and model. When HWP is turned off, the generic scaling information specified in ACPI tables are used.

5.3. Related Work

Since the scaling of CPU resources determines the CPU frequency and therefore the speed, we cluster previous research by their use of intel_pstate if this configuration is explicitly mentioned. The Linux kernel has been supporting this scaling driver since kernel version 3.9 got released in 2013 [89]. The kernel documentation describes[141] how P-states are translated into frequencies which is dependent on the specific processor model and family. Energy consumption grows proportionally with the frequency, therefore also ACPI compliant low power solutions are researched for the cloud [117].

Related work was identified based on two searches at google scholar and ACM digital library on the 26th of April 2023. intel_pstate was used as keyword which resulted in 14 entries at ACM digital library and 141 results on google scholar. Compared to other technical or conceptual search terms, the quantity of publications at the two search engines support the claim that this aspect is often neglected when designing benchmarks. The result list at google scholar contained a lot of presentations and also links to the Linux kernel documentation. As relevant identified conference and journal publications are discussed in the following. The papers have a relation to performance experiments and therefore named or investigated the CPU frequency scaling configurations. Papers which described that they used the intel_pstate driver but not documented its configuration were no further investigated, e.g. [210, 232]. The assumption is that these experiments use the default configuration which is an active intel_pstate driver plus powersave governor. The availability of hardware support is dependent on the processor, the default if available is with active hardware support.

Overall, there are four configuration options for the scaling driver[142]. Option one and two are to use intel_pstate scaling driver in active mode with (1) or without (2) hardware support (no_hwp). The third option is to use it in passive

[141] https://www.kernel.org/doc/html/v5.4/admin-guide/pm/intel_pstate.html#processor-support

[142] In the /etc/default/grub file, the GRUB_CMDLINE_LINUX_DEFAULT property can be changed to a value explained in the kernel documentation (https://www.kernel.org/doc/html/v5.4/admin-guide/pm/intel_pstate.html#kernel-command-line-options-for-intel-pstate). Via sudo update-grub, these changes can be applied and the system can be rebooted.

mode (3), whereas the last option is to disable `intel_pstate` (4). Disabling it results in the usage of the generic `acpi-cpufreq` scaling driver. Some vendor-specific hardware properties, which can be read by the `intel_pstate` scaling driver, cannot be used in this case. There are several reasons to do so. Becker and Chakraborty [20] fixed the CPU performance of their system by disabling this feature and also the turbo boost options. Reducing noise for a particular use case was another reason to disable it [67]. Some researchers, e.g. [37], wanted to be more flexible in changing the frequency by hand during their benchmarks. Since some of the generic scaling governors are overwritten (by using the same name) by `intel_pstate` and others are not usable, sometimes researchers [63, 82, 100, 140, 236] disabled it to use the generic ones. None of the aforementioned papers specified one of the first three options (1-3) explicitly in their papers. Since the default configuration is an active `intel_pstate` (for some models also with HWP enabled) we assume that a lot of benchmarks use this configuration when doing experiments on-premise.

There are also publications which deal explicitly with `intel_pstate` or describe its usage. The first one uses the default configuration where the `powersave` scaling governor is set to balance performance and energy consumption. Kang and others [116] investigated this setting and compared it to their own CPU scaling algorithms as well as to the standard governors implemented. They summarized that it takes some time until the frequency changes when using the default `powersave` govern for `intel_pstate` which does not meet their use case on latency critical network packet processing. Other researchers explicitly used the `performance` governor to prevent frequency changes and therefore variations in CPU performance [62, 92, 297]. Valter and others [272] would have used the `performance` governor as well but they used a remote hosted HPC cluster to execute their use case without the chance to reconfigure it. Noteworthy is their remark that the usage of `intel_pstate` with `powersave` is a potential source of error.

5.4. Problem Analysis

We already motivated the problem of a non-linear performance increase in Figure 5.1. In this Section, we discuss the problem in more detail and compute linear regression models to provide statistical data. H60 and H90 are Intel quad-core machines. We installed Ubuntu 20.04.2 and Docker for executing calibration functions. The machines' specifications are listed in Table 5.1.

To get several performance points for various artificial load settings as shown in Figure 5.1, we use the `cpus`[143] Docker CLI option to limit the CPU usage by containers. At each point in time, only a single container is running on the mentioned machines together with `SeMoDe` which collects the metrics. The impact of our prototype on the CPU utilization is negligible. We measured it using the Linux

[143] https://docs.docker.com/config/containers/resource_constraints/#cpu

Table 5.1.: Specifications of the two machines issued for the shown experiments.

	H60	H90
Processor	i7-2600	i7-7700
Model	42	158
Base Frequency	3.40 GHz	3.60 GHz
Turbo Boost	3.80 GHz	3.90 GHz
Linux Kernel	5.4.0-65	5.4.0-70

`sar` command in three system states: when no function is running, the prototype starts and the prototype idles. We did not see a noteworthy deviation to the clean system state[144].

As a calibration function, we executed LINPACK [65, 66] which solves linear equations as a CPU intensive function packaged in a Docker container at runtime. Each setting present in Figure 5.1 was executed 25 times by increasing the Docker `cpus` option by 0.1. Both machines use `intel_pstate` in active mode as their scaling driver and `powersave` as the scaling governor. HWP is enabled on H90 and not available on H60. At each share of the CPU, e.g. 0.5 `cpus`, the assigned portion of the CPU is nearly fully utilized due to the LINPACK characteristics. This is important to keep in mind when interpreting the diagrams and the subsequent results. In other words, we mimic artificial situations where a defined portion of the system is under heavy load and look at the performance of our system. For example when assigning 0.5 `cpus` on a system with four cores, the CPU utilization is around 12.5%[145].

The situation described here is contrived since in a normal load test, the impact of the frequency scaling might be hidden within the noise of other influencing factors. The CPU is normally not executed long enough at a given utilization to see the phenomenon in the data. Therefore, it is necessary to create a testbed where we can assess this influencing factor in isolation and make changes in the configuration visible.

As mentioned before, for laptops or machines with cooling problems etc., the choice of the scaling driver impacts the power consumption and heat generation. Furthermore, even different models within the same generation of a processor line have different impacts on the frequency scaling due to their specific hardware support for HWP. Being aware of this scaling phenomenon makes it easier for experimenters to choose a suitable scaling behavior for their benchmarks. This enables a performance estimation under low, moderate and high load of a system and does not jeopardize the results and, hence the conclusions drawn.

[144] Look at the following file which shows the utilization and changes when starting the prototype: https://github.com/johannes-manner/SeMoDe/files/6336159/utilization.txt.

[145] Due to other processes running on the system, the utilization is a bit higher, but shared services running in the background are negligible as can be seen for the prototype influence measured via `sar`.

Table 5.2.: Linear regression models for data presented in Figure 5.1.

	H60	H90 - 1.a
p-value	<2.2e-16	<2.2e-16
R^2	0.9995	0.7351
Intercept	-3.081	-50.340
Slope	23.400	56.357
Max GFLOPs	90.905	215.818

The orange lines in Figure 5.1 are based on linear regressions. Table 5.2 shows the statistics to the figures for H60 and H90. The coefficient of determination (R^2) for H60 shows a near ideal relationship between the dependent GFLOPS and the independent `cpus` variable. Therefore, the results of a benchmark on this machine are comparable and fair since doubling the resources results in doubling the GFLOPS. The intercept is negligible in this case and explainable due to inherent computational overhead. Contrary, on H90, the relationship between `cpus` and GFLOPS is good with R^2 being 0.7349, but it is obvious when looking at Figure 5.1 (right) that three performance ranges are visible from [0, 0.5], [0.6, 2.8] and [3.0, 4.0] with different slopes. When checking the available governors at H90, `powersave` and `performance` are active. The kernel documentation states that the "processor is permitted to take over performance scaling control"[146] when exceeding a threshold. When further looking at the different CPUs and their frequencies at runtime via tools like `turbostat`[147], we can see that the `powersave` scaling governor is used for the second interval operating at minimum frequency and the `performance` scaling governor is used for the first and third interval. Therefore, a fair comparison of SUT deployed on H60 and H90 is questionable since H90 performs worse under moderate load than under peak load.

This observation and the statistical evaluation already emphasize the need for a calibration of the CPU performance.

5.5. Methodology

We propose the following solution to RQ3.1. We use LINPACK benchmark as a CPU intensive calibration function and report the metrics specified in Table 5.2 to the user of our research prototype's CLI. Even though the case described in the previous section is contrived, it gives us the option to isolate CPU performance and to make changes in the configuration of `intel_pstate` *visible*. To be able to restrict CPU resources to a single function, we use the Docker CLI `cpus` option. This gives us the chance to artificially fix the CPU utilization at a given value and

[146] https://www.kernel.org/doc/html/v5.4/admin-guide/pm/intel_pstate.html#turbo-p-states-support

[147] https://www.linux.org/docs/man8/turbostat.html

understand the scaling of the CPU frequency and the performance by looking at the computed GFLOPS.

LINPACK is especially suited to assess the performance of multi-core hardware since it makes use of all available CPUs by being machine independent [65]. The same holds true for load testing tools, where concurrent users of a system can be simulated to stress the SUT. Other functions using the available resources in a similar way are also possible for this proposed calibration step, but LINPACK is well established in this domain. An excerpt of the LINPACK output is shown in Listing 5.1. We run our Docker container with a CPU share of 1.0 cpus on H90 in this example.

Listing 5.1: Sample LINPACK execution on H90 for a CPU share of 1.0.

```
$ docker run --cpus=1.0 jmnnr/linpack:v1
  ...
  Intel (R) Optimized LINPACK Benchmark data
  Current date/time: Fri Apr 16 11:10:52 2021
  ...

  ============ Single Runs ============

  Size   LDA    Align.  Time(s)    GFlops
  1000   1000   4       0.005      144.8319
  1000   1000   4       0.095      7.0746
  ...
  10000  25000  4       58.080     11.4819
  10000  25000  4       58.289     11.4406

  Performance Summary (GFlops)

  Size   LDA    Align.  Average  Maximal
  1000   1000   4       87.1978  144.8319
  5000   18000  4       10.9034  10.9638
  10000  25000  4       11.4612  11.4819

  Residual checks PASSED

  ...
```

The *Single Runs* and *Performance Summary* sections present the size of the matrix which is used for the linear computation and the leading dimension of A (LDA) which also determines the storage of arrays in memory. What is interesting in the *Single Runs* section is the problem size of the linear equation system. Problem size 1,000 reached quite a high number of GFLOPS. At this point in time the frequency scaling of the CPU is not stable and also the equations for this problem size are executed within a few milliseconds which distorts the accuracy of the CPU performance measurement in GFLOPS. In addition, the problem sizes are executed repeatedly for more stable results as can be seen for the two runs of problem size 10,000. Therefore, we use the average GFLOPS of the experiment with the largest problem size because the equations in this case run a sufficient period of time to get a stable scaling under this portion of CPU utilization. For sake of simplicity

and to be comparable, we package the LINPACK function and push the image[148] to Docker Hub, which is used by our prototype per default. We further parse the results (Listing 5.1) of LINPACK to get GFLOPS.

5.6. Web UI and Implementation

Figure 5.2 shows part of the web UI for the `setups/{setupName}/calibration` endpoint. The structure of the calibration webpage is similar to the benchmark webpage described in Section 4.3.3.1. The *Local Configuration* part includes the configuration for the local experiment machine and the upfront calibration of the used hardware. The *Local Steps* property describes the resource increase for the start of a Docker container, whose location is specified in the *Docker Source Folder* property. If there is nothing specified, the default is used which is the aforementioned image at Docker Hub. This CPU resource limitation is done via the `cpus` command line argument when starting a container. The *Number of local calibrations* describes how often the calibration is executed on every *Local Steps* setting. A graphical representation as well as a statistical summary of the selected calibration can be seen at the bottom of the diagram. For consistency reasons, the plotted data from Figure 5.1 is shown again here.

Since the calibration takes hours and days to execute, there is a CLI option to start this process[149]. The tool includes two properties as described in the README where a user of the tool can specify the number of runs for each CPU quota and the steps in which the CPU quota is incremented. The default values are 1 run (`runs`) with steps of 0.1 (`steps`). A single run is sufficient to get a confirmation through the statistical output if the performance scaling is appropriate for using the machine in further experiments or if a user has to check BIOS and other settings again. This feature and the introduced web UI answer RQ3.1 on *How can a consistent CPU scaling behavior across various processors and scaling algorithms be achieved and visualized.*

5.7. Evaluation

We now have an answer to RQ3.1, but still a configuration at H90 which does not comply with our prerequisite of a linear scaling of resources. In this evaluation section, we look at the four cases introduced in related work to iteratively assess different options and configure our machine accordingly. We solely focus on H90 since H60 shows an already acceptable CPU performance distribution under diverse load settings. For option one, we further investigate sub-cases to be more precise in drawing conclusions on this specific machine and show the most important settings.

[148] https://hub.docker.com/repository/docker/jmnnr/linpack
[149] https://github.com/johannes-manner/SeMoDe#hardware-calibration-feature

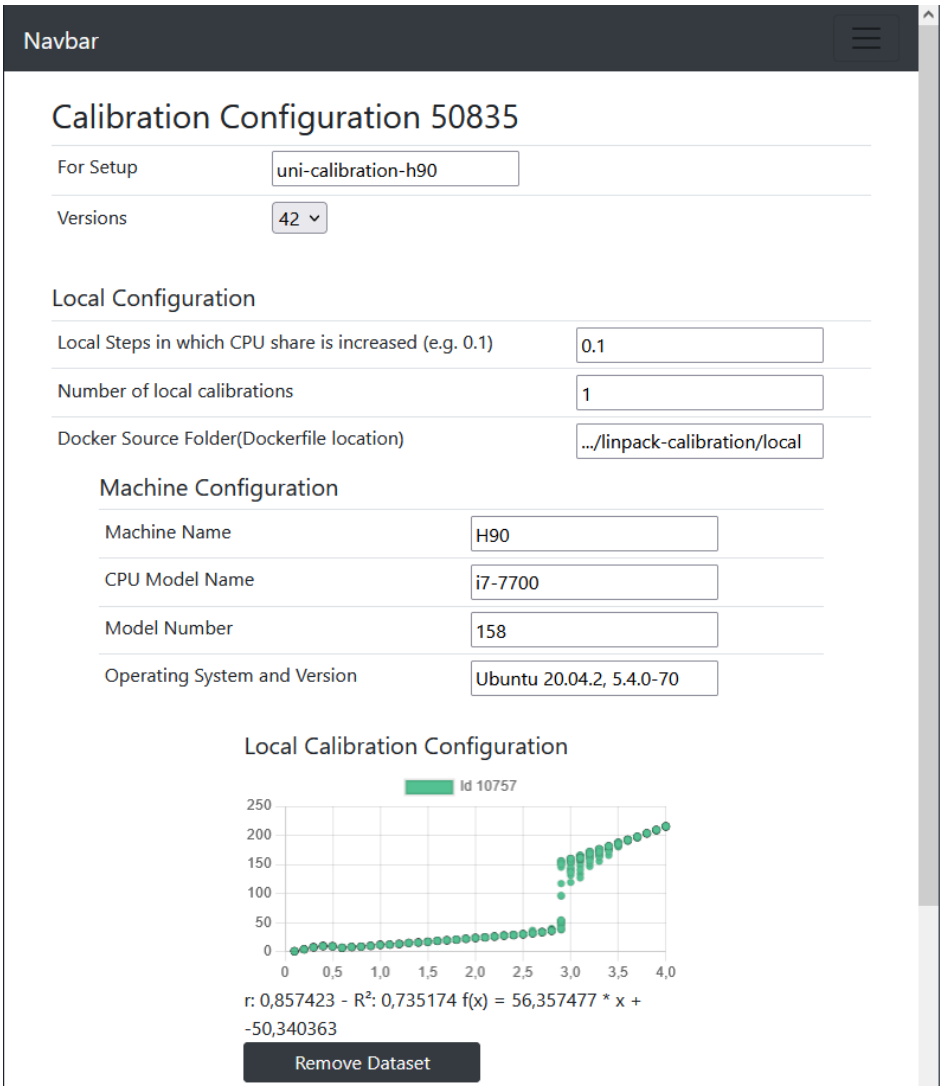


Figure 5.2.: SeMoDe web UI depicts a non-linear performance distribution for an Intel i7-7700, model 158 in one of our experiments.

1. Scaling driver `intel_pstate` in active mode with HWP support.
   a) Turbo boost on, `powersave` scaling governor.
   b) Turbo boost off[150], `powersave` scaling governor.
   c) Turbo boost on, `performance` scaling governor.
   d) Turbo boost off, `performance` scaling governor.
2. Scaling driver `intel_pstate` in active mode without HWP support, `powersave` scaling governor[151].
3. Scaling driver `intel_cpufreq` since `intel_pstate` is in passive mode. Scaling governor is `ondemand`[152].

[150] Change `/sys/devices/system/cpu/intel_pstate/no_ turbo` to "1" disables the turbo boost. "0" indicates an enabled turbo boost.
[151] `GRUB_CMDLINE_LINUX_DEFAULT="intel_pstate=no_hwp"`
[152] `GRUB_CMDLINE_LINUX_DEFAULT= "intel_pstate=passive"`

4. Scaling driver is here `acpi-cpufreq` and governor `ondemand`.[153].

For each setting (except for 1.a), we performed a single calibration run. In our methodology we propose that by default we only need a single run to assess the quality of a system. Especially the active HWP case is investigated further by changing the scaling governor to `performance` and enabling/disabling turbo boost. Options 2 to 4 are investigated in the default setting when updating `grub`, so turbo boost is enabled in all of these cases. The input for LINPACK is constant for all executions with three different matrix sizes (1,000, 5,000 and 10,000) as can be seen in Listing 5.1. We use the average of the highest problem size (10,000) for the statistical evaluation and the figures presented in the following.

The first configuration (1.a) was already shown in Figure 5.1 (right) and evaluated in Table 5.2. The CPU is configured with active HWP and turbo boost[154].

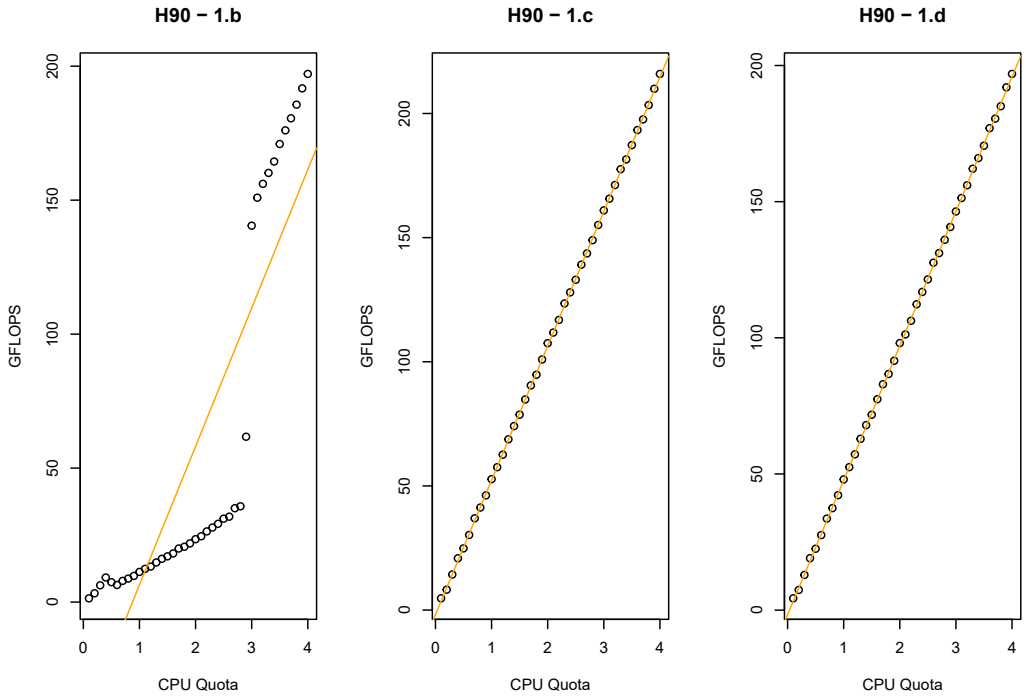


Figure 5.3.: Calibrating H90 in different settings by changing scaling governor and turbo boost.

The other three sub-configurations under option 1 are investigated in Table 5.3 and Figure 5.3. In 1.b we turned off turbo boost, resulting in the same distribution but the maximum achieved GFLOPS is around 10% lower, which is reasonable when looking at the base clock rate of 3.6 GHz and 3.9 GHz in turbo boost mode. The same observation can be made when comparing 1.c and 1.d with each other. The distributions are equal except for the absolute value of GFLOPS.

[153] `GRUB_CMDLINE_LINUX_DEFAULT= "intel_pstate=disable"`

[154] https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

Table 5.3.: Linear regression models for data presented in Figure 5.3

	H90 - 1.b	H90 - 1.c	H90 - 1.d
p-value	6.6e-16	<2.2e-16	<2.2e-16
R^2	0.7406	0.9999	0.9999
Intercept	-45.203	-1.820	-1.715
Slope	51.649	54.118	49.389
Max GFLOPS	197.101	215.939	196.908

The difference between 1.a/1.c and 1.b/1.d respectively is the scaling governor used. We exchanged the `powersave` with the `performance` governor[155]. For the `performance` governor, switching from one to the other algorithm does not happen since the CPU is operated at maximum frequency. Therefore, this configuration is a candidate for doing fair and repeatable benchmarks on H90. The drawback here is, that the power consumption is also at its maximum resulting in additional heat generation and power consumption.

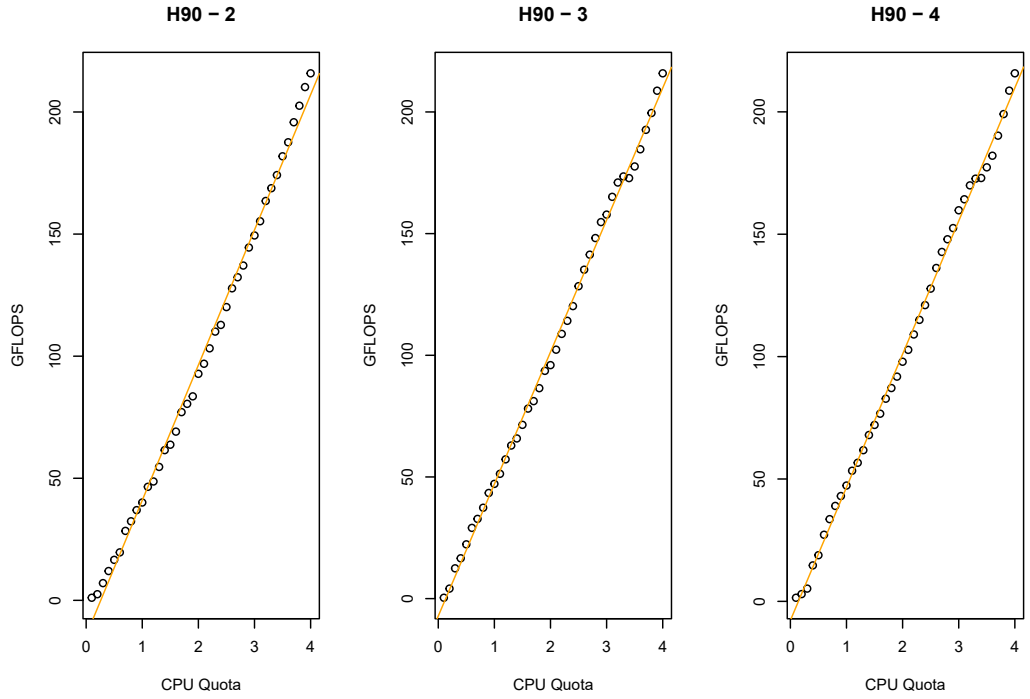


Figure 5.4.: Calibrating H90 in different settings by changing the scaling driver.

Option 2 to 4 are graphically presented in Figure 5.4 and statistically in Table 5.4. Compared to 1.a, the only difference of option 2 is an active `intel_pstate` without HWP support. The HWP support has an impact on the scaling algorithm and enables switching between `powersave` and `performance` governor as already seen in Figure 5.1 (right), whereas the system configured without HWP only uses the `powersave` governor which "selects P-states proportional to the current CPU utiliz-

[155] `sudo cpufreq-set --cpu n --governor performance` where n is a processor

Table 5.4.: Linear regression models for data presented in Figure 5.4

	H90 - 2	H90 - 3	H90 - 4
p-value	<2.2e-16	<2.2e-16	<2.2e-16
R^2	0.9953	0.9975	0.9976
Intercept	-14.378	-7.169	-7.775
Slope	55.362	54.254	54.395
Max GFLOPS	215.835	215.880	215.740

ation"[156] in this operation mode. This results in the undulations seen in Figure 5.4 (left).

Passivating `intel_pstate` (option 3) results in the usage of the `intel_cpufreq` scaling driver and the `ondemand` scaling governor. As stated in the documentation, the HWP support is also disabled. Compared to the second option, the performance behavior is quite similar, however, the governor uses the CPU load to determine the CPU frequency. Only 16 fixed P-states are provided by the generic ACPI frequency table which explains the non-linear scaling behavior. Most researchers disable `intel_pstate` and use the generic `acpi-cpufreq` scaling driver. Since option 3 and 4 use the same governor and the information available in the ACPI tables, their results are similar.

5.8. Conclusion

5.8.1. Discussion of the Results

The presented methodology enables a calibration of systems with respect to the CPU performance. We use LINPACK as a machine independent benchmark to assess the frequency scaling of the CPU and express the power via GFLOPS when solving linear equations. The approach showed one solution to solve the initially motivated situation, where an unpredictable scaling was present. We investigated the most important influencing factors for Intel CPUs under Linux namely the scaling driver and its corresponding governors. Due to vendor-specific implementations, it is possible to make use of the vendor-specific knowledge about the system components like `intel_pstate` showed. Nevertheless, in some configurations, this leads to performance distributions which are questionable when conducting fair and repeatable benchmarks for various load settings. The four options identified in related work cover only a small portion of the overall possible system configuration when taking all possible settings into consideration. Why a lot of researchers disabling `intel_pstate` to use the generic `acpi-cpufreq` scaling driver is not discussed explicitly in their work and also not reasonable when looking at the results of our evaluation. Therefore, it is important to conduct calibration experiments as ours upfront to find a good configuration for the machine

[156] https://www.kernel.org/doc/html/v5.4/admin-guide/pm/intel_pstate.html#powersave

which performs the benchmark with the SUT. Only option 1 was examined in detail by changing the scaling governor and testing the system with turbo boost enabled and disabled but without other fine-tuning. A recommendation based on our data is to use `intel_pstate` as scaling driver for Intel machines with HWP support enabled and set the `performance` scaling governor.

5.8.2. Threats to Validity

Single Machine - We only looked in detail at a single Intel processor (i7-7700, model 158) in this paragraph and assessed its specific configuration. For further generations of Intel processors (second generation onwards) `intel_pstate` is used as a scaling driver and we assume that the behavior shown here is also present at some of these machines dependent on the model and processor line.

Single Vendor - We only looked at Intel processors, but we assume that our methodology also works on other processors like AMDs.

Minimal Dimension - CPU performance was the only dimension we were interested for an initial calibration. To draw strong conclusions, we tried to reduce the influence of all other factors to a minimum. We are aware that the CPU performance is also influenced by cooling requirements, network access, hard disk speed, system bus, etc.

Sample Size - As mentioned in the evaluation, we only executed a single run for each experimental setup since we also propose to do this when using our methodology in practice. We have seen in Figure 5.1 that especially the transition from the second to the third performance interval is interesting since the effects of the scaling algorithm switch can be seen in greater detail for the 25 executions. There is a trade-off between execution time and accuracy of the results. However, in Figure 5.3 (left) we have seen that also a single execution is enough to show a non-linear scaling.

LINPACK Configuration - The problem size, number of runs and the LDA of the LINPACK benchmark can be specified as input. All calibration runs in this section were executed with the same set of parameters, which could bias the results since the LDA is also responsible for how the matrix A consisting the linear equations is stored in RAM.

5.8.3. Future Work

The plan for future work in this area is threefold. Firstly, we want to look at other factors related to the CPU and influencing the performance. First and foremost, we want to look at memory influences by changing the LINPACK input parameters. Also system bus capabilities and other components play a vital role which might require some specific calibration considerations to assess the quality of a system. Furthermore, the cooling capabilities are important to keep in mind when operating the system with the `performance` governor and in turbo boost mode. Fi-

nally, different Intel processors and other vendors are in focus of the next research efforts.

Secondly, we configured our machine H90 at maximum frequency when using the `performance` governor. Questions about the efficiency, in particular the used energy related to the computing power, are important for an assessment how sustainable such a configuration is for several workloads. This is one point of criticism when using LINPACK and use a performance oriented configuration.

A last idea within this calibration process is to establish an abstract computing measure to make different experiment setups comparable and quantifiable. Azure already did so by publishing abstract compute units which are boundaries at the moment due to the heterogeneous hardware used for offering some of their compute services. If research papers and public cloud providers would agree upon a set of parameters published with their experiments and services, also an easier assessment about the energy consumption and CO_2 equivalents could be made which supports companies to structure their sustainability strategies.

6. Simulating FaaS Platforms

Parts of this chapter have been taken from [174, 180, 181, 185].

In this chapter, RQ3.2 *(How can two distinct virtualized execution environments be made comparable?)*, RQ3.3 *(Do resource configurations based on calibration lead to accurate predictions on a provider-hosted FaaS platform in the cloud?)* and RQ3.4 *(How can resource scaling strategies be applied to on-premise open-source FaaS platforms in a manner that is equivalent to cloud strategies?)* are supported.

Simulating FaaS platforms is the central section of this work. A motivation for simulating cloud functions is stated in Section 6.1 by describing the situation and problem of resource configurations at FaaS platforms. Also our solution is shortly outlined. In Section 6.2 related work is discussed based on current profiling and simulation approaches. Section 6.3 suggests the aforementioned simulation approach for a public cloud provider by answering RQ3.2 and RQ3.3. The simulated data are compared to real executions in the cloud to provide an evaluation how accurate the simulations are. Section 6.4 includes work on open-source FaaS platforms where resource limits are applied in a way comparable to an on-premise hosted open-source platform at a public cloud provider. These considerations are prerequisites to enable fair comparisons of the capabilities of open-source and public cloud provider offerings and answer RQ3.4. In Section 6.5, we discuss our simulation approach and list threats to validity. Finally, we conclude with further ideas for ongoing research in Section 6.6.

6.1. Motivation

Most of the available cloud function offerings force the user to choose a memory or other resource setting and a timeout value. CPU is scaled based on the chosen settings, as summarized in Table 3.4 for public cloud providers. At a first glance, this seems like an easy task, but the tradeoff between performance and cost has implications on the quality of service of a cloud function.

Eivy [74, p. 9] claims that the "real execution is the only valid test" to figure out the most suitable configuration. This is reasonable to provide an understanding for absolute measures but is also time consuming and costly. The cloud function must be deployed first, executed based on a load profile in the testing phase, analyzed and finally reconfigured. Therefore, in this section we present a local sim-

ulation approach for cloud functions and support developers in choosing a suitable configuration for public cloud provider platforms and transfer this strategy to open-source tools as well. This stated simulation process is the first and main contribution of this work(C1). In our approach we do not state absolute values but relate local and platform measures to each other and support developers in their decision making process. The methodology we propose makes the cloud and local environment comparable and maps the local profiling data to a cloud platform and simulates the execution behavior of cloud functions locally. This reduces time during the development, enables developers to work with their familiar tools and lead to runtime predictions of functions in the cloud in order to find suitable configuration options. It is especially helpful when implementing multi-threaded cloud functions.

As already remarked, a cloud function has a single purpose compared to an application with several functions executed in parallel. Therefore, in the case of cloud functions, situations arise, where resources are assigned by the developer which cannot be fully utilized based on the cloud function characteristics, particularly for IO workloads or single-threaded functions. As a cloud function user, the possibilities to influence the runtime behavior are quite limited. Some authors state, e.g. [113], that either the developer should receive more configuration options or the optimal setting is automatically determined by the provider. There are offerings, where providers dynamically link memory and CPU resources to the cloud function container, but this further limits flexibility. There are occasions where a user wants to *waste* computing power to speed up the execution for a better user experience. Selecting a resource setting for public cloud offerings is comprehensible since the billing scheme depends on the resource setting. Double the resources results in doubling the price per unit.

In open-source research, this is not necessarily the case. Causes are heterogeneous hardware, but especially function and application resource limits are seldom applied and defaults are used[157]. Open-source tools are often compared to each other and the public cloud based on their capabilities but deployed functions and their runtime behavior are not comparable to the cloud per se. Making research on on-premise hosted open-source platforms comparable to experiments on public cloud providers is another dimension we consider for a holistic view on FaaS platforms.

Answering our research questions helps to identify function characteristics and their implications. We simulate the execution and directly focus on the cloud function runtime behavior itself in contrast to research where the FaaS platform serves as an execution environment to simulate other systems [136] or where the FaaS platform itself is the object under investigation [247]. Based on this assessment, a developer can select a fitting resource setting to prioritize cost over performance or vice versa depending on the business needs.

[157]This statement is an assumption based on the lack of documentation in most experiments.

6.2. Current Profiling and Simulation Approaches

6.2.1. Profiling Strategies

Profiling is the process of generating a profile of the resource consumption of an application, a virtual or physical environment over time. The need for profiling [283] also exists in the cloud function domain as the following three aspects attest: *Management* of cloud functions means that the resources are configured properly to avoid function performance degradation. *Resource* considerations should avoid an overprovisioning situation. And finally, the *cost* perspective in this pay per use model balances the two prior aspects. There are two types of approaches to profiling: Hardware profilers introduce less overhead by getting coarsely grained data. In contrast, software profilers introduce a lot of overhead by instrumenting code resulting in fine grained information [201]. Which approach to choose depends on the use case.

Table 6.1.: Comparison between monitoring and profiling approaches using different abstraction layers.

Abstraction Layer		Approach
intrusive	app layer	Dynamically or statically instrument events inside an application. Altering source code approaches, e.g. [52, 162, 183, 229, 287].
non intrusive	virtual layer	Periodically inspect the state of virtual resources by using APIs, e.g. [40, 205, 218].
	physical layer	Periodically inspect the state of the system using OS tools, e.g. [40, 229].

Table 6.1 is a selection of different profiling strategies. The ones which target the application layer are intrusive approaches since custom metrics or information can only be exposed on a source code level. Therefore, these approaches typically introduce some overhead which has to be in balance with the information gain. With their instrumentation tool, Mace and others [162] enabled a recording of distributed application topologies. They introduced a happened-before join operator to allow the user to investigate traces across component or application boundaries. Another intrusive approach used the additional information to generate test cases a posteriori for faulty executions to support developers to resolve runtime errors [183]. Cuomo and others [52] implemented some wrapper for often used components to derive runtime metrics during benchmarks and use them when executing their simulations. Ren and others [229] introduced the way how Google profiles their data centers from an infrastructure point of view but they also enable application profiling by a commonly used library. They collect heap allocation, lock contention, CPU time and other profiling metrics.

In contrast to intrusive approaches, non-intrusive profiling approaches focus on the management perspective and observe the current system state. Container technology and hypervisor (VM) based systems are typically the starting point when

profiling applications. Pi and others [218] used the container API as a source to collect metrics for implementing a feedback control tool in a distributed environment. Docker as the de facto container standard also supplies some metrics via its *docker stats* API[158]. IBM published their framework to profile and monitor their cloud infrastructure [205], too. They focus on the whole virtualization layer by collecting the memory and persistent state of containers and VMs. Casalicchio and Perciballi [40] used this API to conduct an experiment where they compared different metrics including CPU and memory on a native Linux environment and Docker. *Docker stats* and cAdvisor[159] were used as profiling sources in the container area.

Their research as well considered the physical layer, where they used OS tools like, e.g. the *mpstat* and *iostat* profilers of the native Linux kernel. As mentioned, Google also uses whole-machine profiles on a hardware basis to investigate the different applications and how they consume the machine's resources. In contrast to the application metrics, this data is hidden from a cloud service user. They collect CPU cycles, L1 and L2 cache misses, branch mispredictions and other hardware metrics [229].

6.2.2. Simulation Approaches and Tools

6.2.2.1. Cloud Simulation

Simulation of the cloud gets more important as a special issue on simulation in and of the cloud demonstrates [55]. Time and execution cost are the driving forces to simulate the cloud infrastructure upfront to estimate the probably achieved QoS. The most notable cloud computing simulators [16] are GridSim [34], a tool for simulating grid environments, SCORE[160] [78], a tool for data center simulation based on Google's Omega lightweight simulator, GreenCloud [128], a tool to investigate the energy consumption in data centers, and CloudSim[161]. Other simulation tools are listed in several secondary literature studies [76, 106, 226, 266].

CloudSim [36] is one of the first simulation environments for cloud computing and started with a focus on VM based simulation and federated clouds. This framework especially tackled inter-network components and their delays. With the rise of container technology, they extended their framework to simulate containers as well [220]. Their focus is on containerized cloud computing environments, i.e., studying resource management of containers holistically by looking at container scheduling, placement and their consolidation. Their research point of view is on the provider and not on a single container.

To validate simulated results, complementary approaches like benchmarking and monitoring a system emerged [52, 111]. Johng and others [111], for example,

[158] https://docs.docker.com/engine/reference/commandline/stats/
[159] https://github.com/google/cadvisor
[160] https://github.com/DamianUS/cluster-scheduler-simulator
[161] https://github.com/Cloudslab/cloudsim

developed an ontology based methodology where a mapping function between the different ontologies tries to compare the environments to achieve a closer relation between development and production environments. Execution of benchmarks is necessary for their approach to calibrate the simulation.

6.2.2.2. FaaS Simulation

FaaS simulation is a subarea of cloud simulation. Approaches present in literature can be divided into two categories: Firstly, the FaaS platforms are used as simulation engines where other systems are deployed to and investigated, like in [135, 136]. Secondly, some approaches simulate the FaaS platform itself and cloud functions are only deployed to validate the simulation in the specific experiments. However, there is a lack of such simulation systems [165].

In this section, we discuss current tools tackling this issue. The three simulation papers [2, 72, 165] from the SLR in Section 4.1 are discussed here in detail with a focus on their simulation approach as explained in the papers. Another paper [109] we identified within our SLR process was classified as out of scope and not included in our final SLR set. *DFaaSCloud*[162] [109] introduced a simulation framework for using functions in the continuum of core cloud and edge technologies by extending *CloudSim*. The executed functions are not mentioned in their research, which makes interpreting the results challenging.

An interesting approach and first ideas for a simulator focused on FaaS which simulates the scheduling, initialization etc. of new instances was introduced by Mahmoudi and Khazaei [165]. They included a short evaluation of their ideas and compared the results to platform data from AWS Lambda. They especially focused on scaling strategies, i.e. scale by request, a fixed concurrency threshold and metrics based scaling. Their research prototype `SimFaaS`[163] predicts cold start probability, average response time, rejection of request similar to our prior work [179]. One shortcoming of their methodology is that they only evaluate a single function at a single memory setting in their evaluation. Therefore, specific function characteristics are not within the scope of their work. The configuration of the function and its tuning was out of scope as well. Another work uses a Bayesian optimization model to find the best configuration for a cloud function [2]. They still need a few executions of a function on a cloud platform until the model converges to find its optima. While the authors discuss an important aspect of finding the best configuration, they solely focus on cost. They are not dealing with the trade-off between execution time and cost and neglecting constraints on latency. Furthermore, they did not consider configurations which increase a single CPU. From their Figure 1(c), the interested reader recognizes that the effects of scaling resources are not completely approached. Based on the shown diagrams, the executed function is single threaded. Therefore, it does not fully use the potential of the platform.

[162] https://github.com/etri/DFaaSCloud
[163] https://github.com/pacslab/simfaas/

An approach to predict the end-to-end latency for a collection of cloud functions building an application has been presented by Lin and others [154]. They proposed some profiling of the target cloud platform upfront to have some measure for the model and algorithms. Furthermore they made suggestions on how to solve two optimization problems when searching for minimum cost or the best performance.

Horovitz and others [101] built the self optimizing Machine Learning (ML) tool *FaaStest* by predicting the workload of a function and scheduled functions on VMs or on a cloud function platform. Since the workload is one of the determining factors influencing performance [179] with respect to cold starts and parallelism level on the platform, their research is important for simulating cloud function platforms but does not include function characteristics. Another approach to simulate FaaS is *FaaSSimulator*[230]. In contrast to *FaaStest*, this tool aims to support hybrid decisions by providing a spectrum between VM and FaaS solutions. Their work did not include function characteristics nor a technical setup description.

From related FaaS simulation approaches, *Sizeless*[164] [72] and *SAAF*[165] [50] are most closely related to our research. Eismann and others [72] proposed an approach to predict the best configuration. Their strategy to find a suitable configuration for a resource assignment only incorporates monitoring data where most other approaches perform specific performance tests as an input for their prediction models. They only rely on monitoring data for a single memory setting. While their approach is promising and easy to use for already deployed functions, the current state of the research prototype is limited to a single provider (AWS Lambda) and a single programming language (Node.js). Furthermore, the memory increase beyond 1769 MB is not discussed and, hence, neglecting multithreaded functions. Cordingly and others [50] focused on the multitenancy aspect, where various functions are executed on the same VMs. They also stressed the fact that cloud providers use different hardware. Therefore, the prediction of execution times is determined by the hardware which also directly influences the price. Linear regression models based on the Linux CPU time were used to calculate means and mean errors when profiling the functions. Concurrency was taken into account on a workload level, but not on a function implementation level.

To summarize, none of the presented approaches include function characteristics or concurrency when implementing a cloud function. We include these two aspects as important points in the concept and evaluation of our work.

6.2.3. Experiment Calibration

Calibrating local test-beds to use them for simulations is an already known approach for IaaS offerings. Zakarya and others [298] extended CloudSim to enable VM migration to save energy. Based on a small set of executions in the cloud,

[164] https://github.com/Sizeless
[165] https://github.com/wlloyduw/SAAF

they built linear regression models and were able to simulate their SUT with an accuracy of 98.6%.

Researchers in the FaaS area have also applied some kind of calibration steps in their research to compare measurements and draw conclusions. Back and Andrikopoulos [6] compared their experiments on a local Apache OpenWhisk[166] deployment using VirtualBox VMs to experiments on other commercial FaaS platforms. Jonas and others implemented a prototype to run map primitives on top of AWS Lambda [112]. They executed a matrix multiplication benchmark to measure the overall system performance in GFLOPS and also drew a histogram about the GFLOPS performance per CPU core, which shows a distribution of the CPU core performances. Different CPUs vary in their peak performance, e.g., 16 to 17 GFLOPS and 30 GFLOPS as shown in the above mentioned matrix multiplication case. This implies that different CPUs are used. In another research experiment [282], this assumption was confirmed by finding five different CPU models on AWS. Also co-location of VMs, where the containers are running in, cause multi-tenancy issues, which influence the runtime performance and explain slight deviations of measured values. Both aspects are only partly considered in the related research. Therefore, questions about their impact on the runtime behavior remain unresolved.

To conclude, there is research which enables a comparison of different environments but the used hardware and its influences were not discussed. Based on the chosen experiment, a comparison was possible but environments were not made comparable. This methodological gap in research is our motivation for RQ3.2 where we ask *"How can two distinct virtualized execution environments be made comparable?"*

6.3. Simulating Cloud Functions at Public Cloud Provider Platforms

The insights from the previous sections result in a profiling and calibration strategy to simulate cloud functions on a developer's machine to select a proper resource configuration. The presented process in Figure 6.1 summarizes the calibration, simulation and forecasting approach. Equations and Figures used in the following are added to the steps where they correspond to. This workflow is an essential part towards performance and cost simulation in FaaS. Our research prototype `SeMoDe` supports the overall process. Details are described in Section 7.1.

As already mentioned, the generated artifact at the end of the subprocess is a graphical representation of various local simulation runs which serves as a decision guidance to choose a suitable resource setting depending on the developer's needs. The following subsections explain the most important parts of this process

[166] https://openwhisk.apache.org/

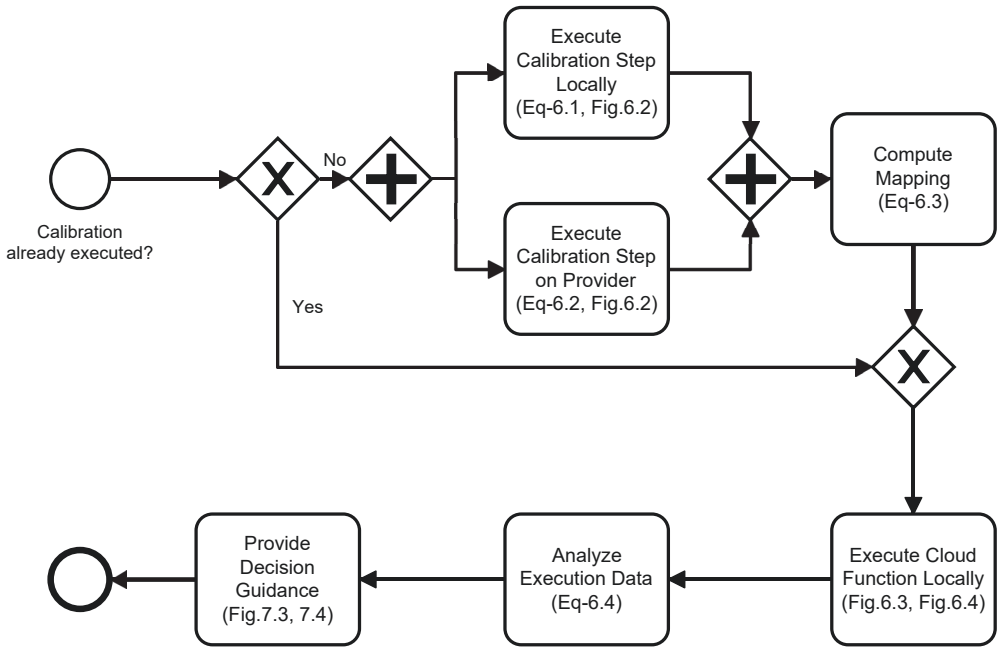


Figure 6.1.: Simulation process for achieving Dev-Prod parity, local simulations and a prediction to the public cloud.

in detail and give a concrete example, how to implement such a process for the integration of developer machines and FaaS platforms.

6.3.1. Achieving Dev-Prod Parity by Calibrations

"Calibration of parameters in simulation models is necessary to develop sharp predictions with quantified uncertainty" [263, Abstract]. This is the motivation to introduce a calibration step to compare the performance offered by cloud infrastructure with the performance of a local machine. Arif and others [4] already stated that a scalar factor to compare different environments is not enough. On our machines, we control the resources using container quotas (cgroups), and on the provider side, computing performance depends on the selected resource setting. As input for the calibration, users specify the granularity of the local calibration experiment and a set of providers, which are in focus for deployment. The calibration has to be executed on the platform provider once per resource setting. If the results are up-to-date and not outdated, we proceed with the execution of our functions locally as explained in the following section. If not, the calibration is divided into two tasks, which can be executed in parallel, and a following mapping step.

Sodan [251] compared different CPU architectures. Specific types of instruction sets were under investigation to optimize algorithms, as presented in Sprunt's work [258]. He describes often used *program characterization events*, for example the floating point event, to assess the SUT. This research is important for improving CPU architectures and optimizing algorithms in the area of high perform-

ance computing, but of limited relevance in the FaaS area since most of the providers use commodity hardware in their data centers. Sprunt emphasizes that processor's implementations are mostly abstracted by these *program characterizations events*. This allows an objective comparison of different hardware. On a conceptual level, such a comparison is needed for a simulation of the performance characteristics of a cloud function on the FaaS platform while executing it offline on a developer's machine. He stated that such a local simulation of the application stack, including the OS and processor information results in less accurate predictions for specialized algorithms and is ultimately not convincing. We overcome this problem in our approach since we are conducting established and well controllable experiments locally and on the specified FaaS platforms w.r.t. the mentioned program characterization events. Furthermore, we do not explicitly focus on specialized algorithms. The average-case business functionality is our concern. In a second step, we compute a function to equate the two application stacks and use it for our simulation task and the local execution of the functions.

6.3.1.1. Calibration Function

We use LINPACK, first introduced in 1979 [65] and still extending [66], as a hardware independent experiment on provider and user side. LINPACK is a package to solve linear equations of different complexity in single or double precision arithmetic. It is a de facto standard to compare CPU performance[167]. Dongarra stated that the LINPACK benchmark results not in a one-size-fits-all performance value, but the problem domain of solving linear equations is very common for any type of application. Therefore, the LINPACK results give a good hint about the CPU peak performance. It is also included in various micro-benchmark experiments and part of a workload suite for cloud function comparisons [121].

Most related to our calibration approach is work of Malawski and others [167]. They also use the LINPACK benchmark to compare the performance of AWS Lambda and Google Cloud Functions with different memory settings. Their results strengthen the hypothesis gained from previous work, that CPU resources are scaled linearly with the resource setting. AWS Lambda shows consistent linear scaling in GFLOPS performance, but a high variation in the results. Two performance ranges emerge when memory increases beyond 1024 MB. In previous research in 2018 [182], we also found these different levels of performance using a CPU intensive Fibonacci function to compare the different platform offerings but did not investigate this phenomenon in more detail back then. The original experiment and a follow-up investigation in 2023 are described in Section 7.2. Lee and others [145] used matrix manipulation to obtain the CPU performance of a Lambda function deployed on AWS. They submitted a workload and ascertained doubled execution time in the concurrent mode compared to the sequential execution, which yields to the multi-tenancy assumptions, where one of their functions

[167] https://www.top500.org/

acted as a noisy neighbor for another function. They also found similar absolute values like Malawski and others [167] with 19.63 GFLOPS for 1.5 MB memory setting and approximately 40 GFLOPS for 3 MB configuration but the reasons are unclear since some data is missing to interpret the technical infrastructure or other aspects influencing the response time and performance. These performance ranges and other literature read in the last years lead to the inclusion of the CPU model and model name as well as the VM identification in our checklist in Section 4.2 to draw strong conclusions on the results. This information is important to relate the hardware used in the experiments with the cloud function execution. Cloud providers may use various commodity hardware in different geographical regions or even in a single data center but LINPACK as a machine independent calibration function is a first step towards achieving dev-prod parity.

6.3.1.2. Calibration Mapping

For many FaaS providers, the resource setting directly determines the CPU resources linked to the container where the cloud function is executed. This is understandable since a cloud provider aims for a high utilization of the machine while providing a robust quality of service without interference for the functions running on it. The output of our previous calibration is the input of this mapping process. For the local machine and the FaaS cloud platform, we get two sets of execution data. The local machine data includes the GFLOPS achieved in relation to the CPU core shares. This is formalized in Equation 6.1:

$$f_{\text{local}}(y) \; = \; m_1 \; * \; y \; + \; t_1 \tag{6.1}$$

where

$$f_{\text{local}}(y), y \; \in \; \{y \mid 0 < y \leq c\}$$

with c being the maximum number of physical cores.

The functions deployed on the cloud provider also compute GFLOPS based on a cloud resource setting. This is formalized in Equation 6.2:

$$f_{\text{provider}}(x) \; = \; m_2 \; * \; x \; + \; t_2 \tag{6.2}$$

where

$$f_{\text{provider}}(x), x \in \{x \mid x \; is \; a \; cloud \; resource \; setting\}.$$

$m1$ and $m2$ are slopes which describe an increase in GFLOPS based on the selected execution environment. The unit for $m1$ and $m2$ are GFLOPS/CPU and GFLOPS/MB, respectively. $t1$ and $t2$ are intercepts with the y-axis. Their unit is GFLOPS. Although the two values for y in CPUs and x in MBs have different units, the outcome of the equations $f(y)$ and $f(x)$ are measured in GFLOPS, an abstract

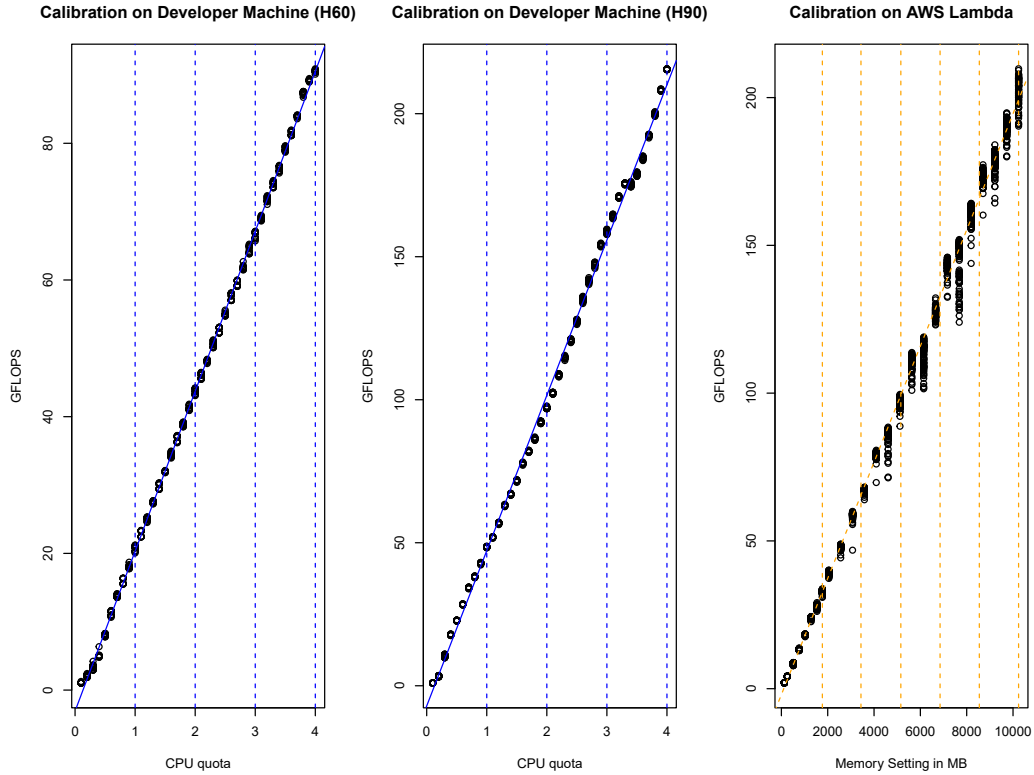


Figure 6.2.: Calibration result of the performed LINPACK benchmarks on a cloud provider platform and locally.

computing value, which allows a direct comparison between the two execution environments.

Figure 6.2 shows exemplary diagrams. To enable an OS independent calibration, a Docker image is prepared for the local execution of the LINPACK benchmark. To get GFLOPS of different CPU shares, we use the capabilities of Complete Fair Scheduler (CFS) of the Linux kernel to limit CPU resources to the executing Docker container. For a solid data basis, we execute the Docker image repeatedly while incrementing the CPU share. In those cases in which the local calibration shows a non-linear distribution, an additional investigation of the machine's kernel and BIOS settings is needed as described in detail in Section 5. In those cases in which the local calibration shows a linear distribution, we use EQ 6.1 to calculate the correct CPU share for the GFLOPS setting we want to consider. Furthermore, we can compute a comparable memory setting for a FaaS provider by using the same GFLOPS setting.

$$y = \frac{m_2 * x + t_2 - t_1}{m_1} \tag{6.3}$$

Our resulting mapping function is presented in Equation 6.3 and computed on the assumption that both machine settings should yield the same result, i.e.

$$f_{\text{local}}(y) = f_{\text{provider}}(x).$$

This equation answers RQ3.2, where we asked how two virtualized environments can be made comparable and mapped to each other.

For an experimental setting in one of our experiments, we got the following regression lines:

- $f(y) = 54.28\,\frac{GFLOPS}{CPU} * y - 7.05\,GFLOPS$
- $f(x) = 0.01965\,\frac{GFLOPS}{MB} * x - 1.995\,GFLOPS$

Assuming that we want to simulate our cloud function for 256 MB and 512 MB memory on our local machine, we can compute the local CPU share using Equation 6.3:

$$y = \frac{0.01965\,\frac{GFLOPS}{MB} * x + 5.055\,GFLOPS}{54.28\,\frac{GFLOPS}{CPU}}$$

This results in 0.186 cores for 256 MB and 0.278 cores for 512 MB. Doubling the memory setting does not result in doubling the CPU resources for the local container and vice versa. This sample data gives a first hint about the obvious fact that the two regression lines have different positive slopes and a direct conversion from one to the other through a scalar factor is not possible.

6.3.2. Execute Cloud Functions Locally

The next step in our subprocess is the local simulation of our function which we want to deploy to the cloud. We overcome shortcomings Kalibera and Jones[114] identified in the evaluation of system research. They categorized experiment dimensions in influencing factors which are random, uncontrolled or controlled by the experimenter. We specify only a set of memory settings in our example, e.g. 256 MB and 512 MB. Therefore, we have only a single influencing factor, which is controlled within the experiment. Other factors like the CPU share are determined by the memory setting and therefore transitively controlled.

To adhere to the dev-prod parity principle, we suggest Docker as a container platform to simulate our functions locally. Concrete execution times from local machines are not directly interpretable but relevant when compared to values from executions in the cloud [4]. To continue with our example, when simulating a function with 256 MB and 512 MB, we assume for this example that the simulation for 512 MB is 1.5 times faster than the simulation for 256 MB. In an ideal, CPU-intensive world, the 512 MB variant should be twice as fast. Hence, the 256 MB solution is more cost effective. If time is a critical factor, e.g. when a function is handling user requests, the 512 MB solution might be preferable. These insights can be drawn without deploying the function to a platform.

Ideally, a user of the simulation tool knows the best, worst and average case for the input of their function as well as the load distribution. Currently a user of our research prototype, explained in more detail in Section 7.1, can specify a single input for the function. Different workloads are not supported. The reason is that a

single input configuration, might it be the worst, best or average case, is sufficient for a prediction of the performance for different resource configurations. The load distribution is relevant for the share of cold starts when operating the cloud function but not for our concept because the execution time of a single function is not influenced by the load distribution when the FaaS platform is constructed in a way that multiple tenants do not act as their noisy neighbors.

6.3.3. Evaluation

The methodological considerations gave an answer to RQ3.2, in particular EQ 6.3. It enables a mapping between a local machine and a FaaS platform. The following evaluation supports our methodology by answering RQ 3.3: *"Does the chosen resource configurations based on the calibration lead to accurate predictions on a provider-hosted FaaS platform in the cloud?"*

6.3.3.1. Experimental Setup

In order to ensure repeatable experiments, we first state the tools and machines we used for our evaluation of the introduced calibration and prediction. As our local experimenter machines we use the two Intel machines calibrated in Section 5: An Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz, model 42 with 4 cores (named H60 in the following) and an Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, model 158 with 4 cores (named H90 in the following). We installed Ubuntu 20.04.2 LTS and Docker to execute the containers on both machines with the help of `SeMoDe`.

We limit the evaluation to AWS Lambda as target platform for our simulations due to its dominance in the market and documentation of its resource scaling behavior. When designing the functionality of SeMoDe in 2018, these were major reasons to focus on AWS Lambda since the linear scaling of resources, see Table 3.4, is predictable and a prerequisite to build repeatable benchmarks and simulation models. All benchmarking requests on AWS presented in this evaluation section were executed on an Intel(R) Xeon(R) Processor @ 2.50 GHz, model 63 and in the availability region *eu-central-1*. Therefore, we have no heterogeneity of CPU architectures in these experiments as discussed in other research [50, 86, 206, 213, 299]. Nevertheless, in Section 6.4, there are also experiments with various Xeon processors at AWS Lambda. Implications of different hardware are discussed there.

6.3.3.2. Calibration Step

Based on the provider documentation and the linear increase on computing resources, the assumption is that the computed GFLOPS grow proportionally to the memory setting on provider side or the local CPU share respectively. Table 6.2 shows that the linear regression is statistically significant. Graphical representations of the local and AWS calibrations are shown in Figure 6.2.

Table 6.2.: Linear regression models for calibration data. The unit of intercepts and slopes is GFLOPS.

	Local (H60)	Local (H90)	AWS
p-value	<2.2e-16	<2.2e-16	<2.2e-16
R^2	0.9995	0.9978	0.9973
Intercept	-3.081	-7.052	-1.995
Slope	23.400	54.284	0.020

The calibration step here follows the methodology already introduced in Section 5. We created a container with the LINPACK source code and executed it by increasing the CPU share in 0.1 steps. Since our machine has 4 cores, we made 40 measurements per run. After 25 runs, we computed a linear regression where the coefficient of determination (R^2) was 0.9995 (H60) and 0.9978 (H90). The intercept of the regression line is not the origin. The intercept is negative, which can be explained by the inherent overhead of all computations. We also computed the regression with an intercept at 0, but this worsens R^2 and the fit of the regression line to the datapoints between 0.3 and 3.6 CPUs. The AWS calibration was executed 100 times for the memory settings 128, 256, 512, 768, 1024, 1280, 1536, *1769*, *1770*[168], 2048, 2560, 3072, 3584, 4096, 4608, 5120, 5632, 6144, 6656, 7168, 7680, 8192, 8704, 9216, 9728, 10240. As for the local calibration we decided to compute a Pearson regression where R^2 was 0.9973. Still an acceptable Pearson regression, but it is obvious when looking at Figure 6.2 that the values for specific memory settings show a wider distribution. As mentioned in the conceptual part, only valid provider values for the specific configuration variables are allowed. In case of AWS the memory setting needs to be a natural number between 128 and 10240 MB as of the time of writing.

We showed a close correlation between memory/CPU shares and GFLOPS for the corresponding environments. Due to the high correlation coefficients, we eliminated the dependent variable GFLOPS and used Equation 6.3 to compute the CPU share for specific memory settings. We are further able to predict the cloud performance of a function when executing it on a local machine during the development process. Comparing absolute values between the cloud and local environment is limited due to the different hardware used, but the trends of the execution time are relevant to enable the proposed simulations locally. Nevertheless, prediction of absolute values is possible. Therefore some execution data for the function for a single memory setting is sufficient as Section 6.3.3.4 explains.

[168] AWS Lambda assigns a second vCPU to the function at 1770 MB so we selected 1769 MB and 1770 MB as settings in order to determine whether this has an impact on the GFLOPS (https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html).

6.3.3.3. Simulating Cloud Function Behavior

In this part, we implemented two functions with different hypotheses. As a literature study showed [240], most publications in the FaaS area focus only on CPU-intensive functions. We followed this approach.

For a first evaluation, we implemented a function to compute the Fibonacci sequence. Hypothesis **H1** is that this function will not profit from multi-core environments and show the same execution behavior for all settings which are equivalent to more than one CPU. The next hypothesis **H2** is that multi-threaded functions will profit proportionally from a resource increase when assigning memory settings which exceed one CPU. We tackle **H2** by implementing a multi-threaded function to search prime numbers in a given range. The Fibonacci function was executed for each memory setting 100 times, the prime number function 5 times.

For Fibonacci, we implemented our function in Node.js and used only a single input value for all tests (n=40) since this eliminates the input as another variable and strengthens the results for execution times and the proposed methodology. The Fibonacci function is widely used as a CPU intensive function, e.g. [182, 271], for microbenchmark experiments.

The vertical lines in Figures 6.2, 6.3 and 6.4 indicate the values where a CPU core is fully utilized and another portion of the next core is added for further executions. In Figure 6.2 the top and middle diagrams show the number of CPUs used by the LINPACK calibration on our local machines, where we are aware of the cpus setting. The bottom diagram shows the AWS LINPACK execution where we computed memory equivalents for fully utilized CPUs. These values were derived from the documentation for the first CPU equivalent and interpolated for CPU equivalents 2-6. For Figures 6.3 and 6.4, we computed the CPU-memory equivalents via Equation 6.3 to have the same dimension on the x-axis to support the interpretation of our results. On H60 for example, one CPU is fully utilized by comparable memory values greater than 1135 MB (2505 MB for H90).

Obviously, the Fibonacci function we deployed does not profit from the multi-core environment as can be seen in Figure 6.3. Especially the execution on H60 (middle of the figure) shows constant execution time after increasing memory and exceeding the first CPU-memory equivalent (vertical line at 1135 MB). In this case e.g. at 2048 MB, the function has access to 1.77 cores, but is only capable of fully utilizing a single one since the function is implemented single-threaded. In a production use case without overbooking and strict resource allocation policies, this would result in wasting CPU resources and adding additional costs for getting the same performance (execution time) compared to other configurations. This problem of overbooking resources without a clarification of the causes can be found in other research (e.g., Fig. 2. in [154] or Fig. 1. in [72]). This observation confirms **H1**.

Jonas and others argued that the FaaS programming model simplifies the deployment and execution of "distributed computing for the 99%" [112, paper title] but the fact that the ability to use multi-core environments for these functions

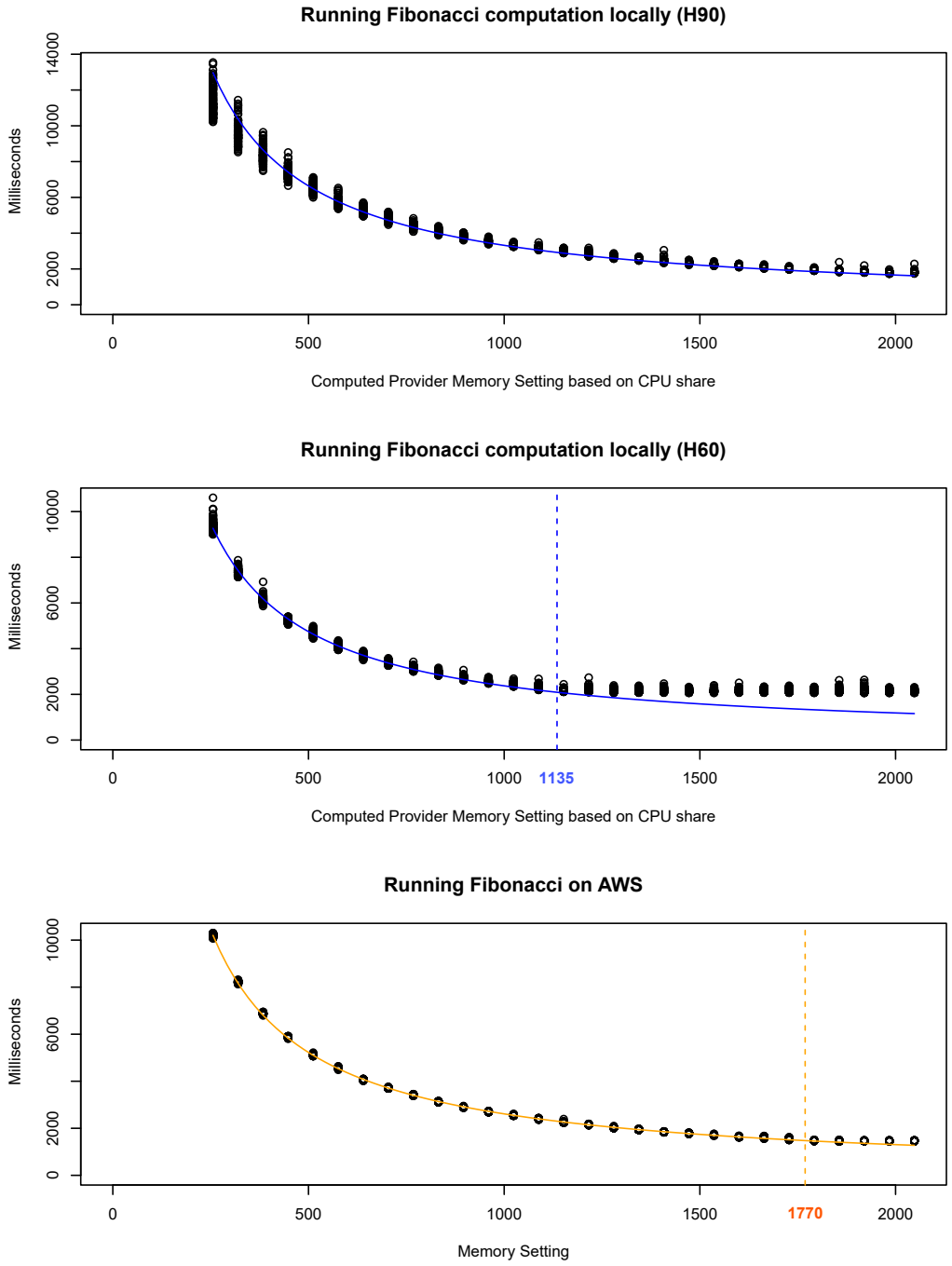


Figure 6.3.: Running Fibonacci cloud functions locally and on AWS.

determines the runtime behavior is often neglected. In addition, awareness of multi-threaded functions is missing. This is even more important when looking at improvements in the resource allocation for cloud functions[169]. Providers in general claim that doubling the resource allocation halves the execution time. This is also the scaling process AWS Lambda advertises on its platform. Such a procedure

[169] AWS Lambda increased their memory and CPU capabilities in December 2020: https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/

is comprehensible and fair but only applicable to highly CPU-intensive functions, implemented in a multi-threaded way without blocking calls to third party services.

This promise - doubling resources results in halving execution time - was used for generating the blue respectively orange curve in Figure 6.3 and 6.4 dynamically for a memory setting of interest (*a*). Based on a grouping of execution times by memory size, we can compute the average (arithmetic mean) execution time (AVG_a) for a memory setting *a*. To get a line assessing execution data visually, memory settings of interest can be chosen (*x*) to compute specific points on the curve.

Equation 6.4 shows the formula for computing the curves.

$$f(x) = \frac{AVG_a * a}{x} \tag{6.4}$$

This gives us the chance to select a memory setting during development and look at performance data to assess the function performance graphically. For an optimization and estimation the curves help in interpreting the results.

As an example, we used 512 MB as our *a* for the plots in Figure 6.3 and 6.4. Therefore, the average execution time (AVG_{512}) is a point on the curve. All values above the curve do not profit proportionally from scaling resources. This would result in spending more resources on the task than necessary. As mentioned before, there might be situations, where doubling the memory setting and therefore the cost is acceptable for a 1.5 decrease in execution time, but these decisions are use case dependent. Vice versa, all executions under the curve profit disproportionately from the resource increase. This is only the case, when a situation as mentioned happens, e.g. 1.5 decrease when doubling resources. Then the first memory setting chosen would profit from a downscale. For CPU intensive functions like the Fibonacci use case this is rarely the case, since the ideal case is halving the time by doubling the resources and our calibration function LINPACK is optimized for such a CPU performance use case. When looking again at Figure 6.3 (middle), we see this over-provisioning starting approximately at 1135 MB on H60 where the execution data is above the optimal curve. The reason therefore is the resource assignment, more than one core, and not the function characteristic.

The second function was implemented in Java. It counts the number of prime numbers within a given range [2, 500'000]. The range, as the input for the Fibonacci use case, is constant for all simulation executions. Each memory setting was executed 5 times. We used the common fork join pool[170] of the JVM to divide the task equally on the assigned cores. Figure 6.4 shows the simulations on H60 and H90 as well as executions in the cloud. We can see that the higher memory settings (e.g. 2048 or 3008 MB) are slightly above the orange curve (*a*=512 MB) which is most likely due to scheduling effects etc. when coordinating various threads. The CPU-memory equivalents on AWS are interpolated based on the range and num-

[170] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinPool.html

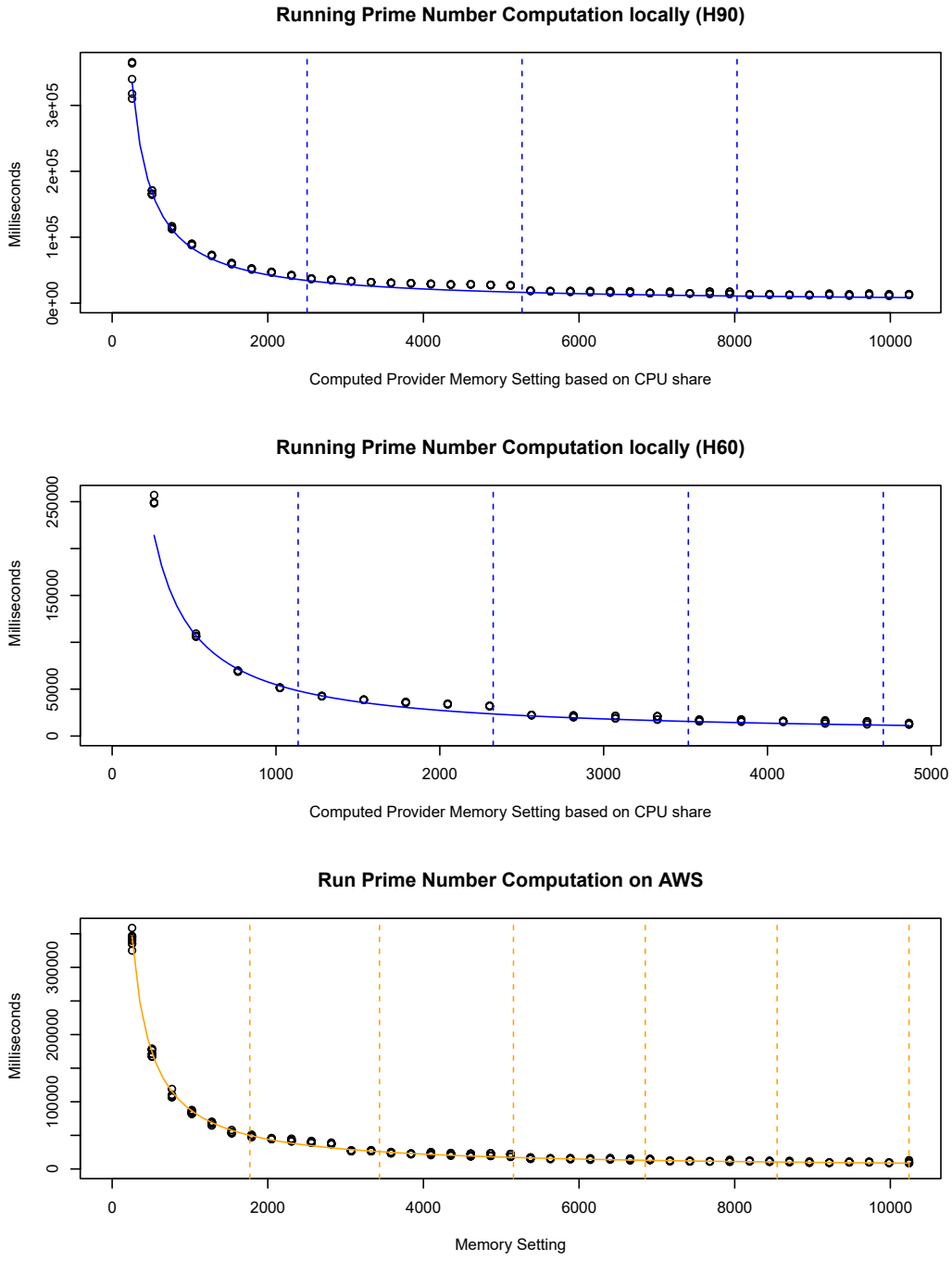


Figure 6.4.: Running prime number cloud functions locally and on AWS.

ber of cores derived from the documentation. As for the Fibonacci use case, the blue curves predict how the function will profit from increasing memory on the provider platform. Also our second hypothesis (**H2**) is confirmed since the AWS Lambda executions as well as the local simulations are closely to the blue respectively orange curves. What is interesting in the simulation runs and also at AWS Lambda is the slight performance degradation when reaching the next complete CPU core, e.g. in the case of H90 when running the prime function with 2304, 5120 or 7936 MB. In this case, scheduling, especially busyness of the CPU and

other system processes consuming CPU time, might be responsible for this phenomenon.

6.3.3.4. Predicting Cloud Function Execution Time

We discussed about CPU-memory setting equivalents a lot. So far, we suggested a calibration step to compute CPU shares based on the memory setting by determining the GFLOPS locally and on AWS. Obviously, these CPU shares differ, so one core on H60 is comparable to 1135 MB whereas H90 has a one core equivalent of 2505 MB on the local machines compared to 1770 MB on AWS.

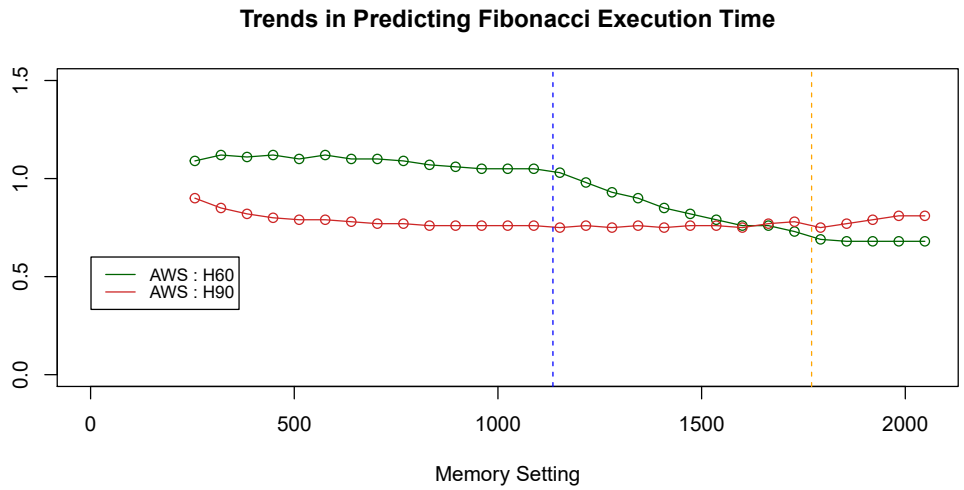


Figure 6.5.: Trends in prediction provider execution time by local execution time for Fibonacci cloud function.

As stated to predict the execution time on a FaaS platform based on simulation data, a few benchmark values are necessary to compute a ratio. This ratio is the reverse answer for RQ3.3 (*Does the chosen resource configurations based on the calibration lead to accurate predictions on a provider-hosted FaaS platform in the cloud?*). In Figures 6.5 and 6.6, we computed ratios for AWS and the local machines for the Fibonacci and prime number use case. Looking at the Fibonacci lines first, the ratio is nearly constant at 1.1 (AWS/H60) until 1135 MB. The other curve is different for the execution data of AWS and H90 with a factor of 0.8 where an increase after 1770 MB is visible. These factors can be used to predict the execution time for functions with a similar characteristic on the platform by using simulation data.

The reason for both increase and decrease is the multi-threading aspect. H60 reaches the one CPU equivalent at 1135 MB. After this, the execution time remains stable for Fibonacci on H60. Since the one CPU equivalent on AWS is reached at 1770 MB, the execution time on AWS decreases until this value and therefore the ratio also decreases (enumerator decreases by constant denominator). After 1770 MB, both environments (H60 and AWS) have the same *issue* and the ratio is constant again.

For the red curve in Figure 6.5 (H90) there is a slight increase of the ratio visible after 1770 MB. Functions running on H90 profit from the increase until 2505 MB (the one CPU equivalent on H90), therefore the enumerator is constant, but the denominator decreases resulting in an increase in the ratio. So for the prediction of execution times from the local to the platform time the inclusion of both, the multi-threading behavior as well as the CPU-memory equivalents, are important.

Figure 6.6.: Trends in prediction provider execution time by local execution time for prime number cloud function.

Figure 6.6 shows the trends for the prime number cloud function. The ratio on H90 is between 0.73 (7936 MB) and 1.08 (2560 MB). In the case of 2560 MB, AWS already profits from a portion of the second CPU, whereas H90 only exceeded the first CPU-equivalent. In the case of 7936 MB, three full CPUs compute the results locally, whereas AWS works with a portion of the fifth CPU based on our interpolation. For H60 and AWS the ratios are between 1.32 and 1.79. The simulation was only possible until 4707 MB (four CPU equivalent on H60 and therefore system limit). As shown in Figure 6.4, at 2560 MB and 2816 MB the executions on AWS Lambda were slower than the optimal values when looking at the optimal curve. These higher values result in a higher ratio. The reasons for this performance decrease can be manifold and will be discussed in further research.

6.3.4. Summary of Achieving Dev-Prod Parity and Local FaaS Simulations

To compare two execution environments with each other, a common measure must be identified first. We chose *program characterization events* [258] in the form of solving a standard problem and use the achieved performance measure, in our case GFLOPS for the LINPACK execution, as a machine independent, common measure. Based on these calibrated values, we map one machine stack to another. Since a single scalar value is not sufficient for this mapping, we compute regres-

sion lines and compare different execution environments based on equal GFLOPS values. This is our answer for achieving dev-prod parity.

The local simulations are then based on this calibrations to compute equivalent settings given a deployment target memory size on the provider or an abstract GFLOPS value on the user side. The evaluation showed that the simulated execution values are relatively comparable. For a prediction of execution time for a cloud function candidate with similar requirements, e.g. CPU intensive functions, a few execution data points for already deployed functions are necessary to compute the quotients used for the prediction.

6.4. Resource Scaling Strategies for Open-Source FaaS Platforms

In the previous section we investigated a method to compare a hardware stack on developer side with public cloud platforms to simulate functions by comparable, in the best case equivalent, resource settings. Another question which now arises is: *"How can resource scaling strategies be applied to on-premise open-source FaaS platforms in a manner that is equivalent to cloud strategies?"*.

To discuss this RQ3.4 in detail, the most famous open-source platform based on GitHub stars was chosen as shown in Table 3.5. We reused some parts of the calibration and simulation approach and applied them to cloud functions running on an on-premise hosted single node OpenFaaS deployment. During experimentation, another interesting aspect, namely multi-tenancy and its effects, occurred which strengthens the importance of a resource aware assignment of computing resources also for open-source platforms.

6.4.1. Motivation

Open-source software is gaining popularity in all areas of life. Especially the Covid-19 pandemic showed efforts in collaboration via software tools and sharing data under one of the several free licenses [209]. OpenStack[171], a top-level open infrastructure project, is the most prominent open source cloud platform [194]. Companies often deploy it on-premise to meet data protection regulations and gain flexibility in the usage and resource allocation to services. This on-premise deployment leads to a major decision for the method of settlement compared to public cloud computing offerings. The IT department has to be structured either as a cost center or profit center [47]. In the latter case, monitoring and metering are important aspects to implement *measured services*, one of the five essential cloud characteristics [192]. In the case of OpenStack, Ceilometer[172] offers a metering service which is the basis for accounting and billing. It monitors the physical and

[171] https://www.openstack.org/
[172] https://docs.openstack.org/ceilometer/latest/

virtual resources [248]. This data can then be used to forecast behavior of services in the future as well as to monitor QoS.

The International Telecommunication Union (ICU) defines QoS as a "set of quality requirements [...] specified in a contract or measured and reported" [104, p. 10]. Typical QoS requirements for software systems are for example throughput, reliability or scalability. For FaaS, a recent study [170] has shown that especially latency, user cost and resource efficiency are considered in literature. A QoS aspect related to resource efficiency is the distribution of execution times which can vary considerably due to overbooking of physical machines or starvation of requests. This can be avoided by suitable resource allocation strategies which we want to focus on in this work. Empirical research on open source platforms [11, 149, 197, 207] did not specify resource constraints in detail. If research address this important aspect, the question is often to understand dynamic resource scheduling strategies for future workloads [102] rather than finding QoS compliant resource settings for deployment of the cloud function on-premise.

Open-source offerings are often investigated by comparing their features to commercial cloud offerings. However, performance benchmarking is rarely executed for open-source tools hosted on-premise nor is it possible to conduct fair cost comparisons. To support developers with varying QoS goals for different cloud functions, an allocation respectively scaling strategy for CPU and memory resources needs to fulfill abstract performance measures to guarantee quality requirements. Resource allocation is the most important aspect to determine the execution time and the demand on the physical machine. For open-source platforms, quality attributes are hard to standardize. These tools are deployed on on-premise hardware which is heterogeneous compared to the mostly homogeneous hardware in computing centers of cloud providers. It makes a difference if, for example, an on-premise hosted platform is deployed on a server with an Intel Xeon Gold processor or on a consumer machine with an Intel i7. Therefore, open source tools cannot guarantee execution time ranges for functions nor abstract performance measures like MIPS [171] or GFLOPS [185] for different function settings in general. Every local on-premise production setup has its own characteristics. Furthermore, private clusters are limited in their capability to run bursty workloads because of the trade-off between utilization and cost efficiency. Also the pricing of functions for on-premise hosted FaaS platforms is often neglected due to the organization of the IT department. In research some experiments like in Das and others [57, p. 610] "assume cost of executing a function in the private cloud to be zero". However, without a pricing scheme for on-premise hosted platforms, a fair monetary and performance comparison to public cloud offerings is not possible. Especially the physical machines, where the cloud function has been deployed to, have a major influence on quality attributes. Since open-source software is unaware of the physical machines used and does not state how hardware should be prepared or even calibrated, it cannot provide the user with information about QoS attributes. These considerations lead to RQ3.4 to enable an informed

decision about a resource-aware deployment of cloud functions on an on-premise hosted open-source FaaS platform.

To answer our research question RQ3.4, we focus on measuring the execution time and limit the consumed resources to implement a QoS compliant strategy when keeping *resource efficiency* in mind. We further suggest a pricing scheme dependent on the resource scaling strategy used.

In Section 6.4.2 we look at related work, especially at comparisons of on-premise hosted tools and equivalent commercial cloud offerings. An understanding of resource scaling strategies for public cloud provider and open-source tools is important to assess the research papers and their contributions. These strategies are listed in our conceptualization in Section 3, Figures 3.4 and 3.5. Secondly, we assess published scaling strategies and experiments. Lastly, scientific improvements in scaling resources are investigated. Section 6.4.3 describes our methodology and answers RQ3.4 followed by our evaluation in Section 6.4.4, where Subsection 6.4.4.4 presents data for the noisy neighbor problem. We end with a discussion of our results and give first insights how a predictable scaling strategy help to implement a profit center structured price model for open-source FaaS platforms.

6.4.2. Related Work

Shared allocation of resources is one of the enablers of cloud offerings. While providers have their expertise in operations, cloud computing customers profit from the pay-as-you-go billing model. Costs for hardware procurement, operations etc. is shared between the users. When thinking about a comparison of self-hosted and cloud services, one important aspect is security [198]. A mixed evaluation is presented in Molnar's and Schechter's work [198] in the early days of cloud computing. Strong data protection and tenancy considerations favor on-premise deployments whereas security investments are more affordable in the cloud due to the allocation of costs. Another costs factor is hosting, investigated in Lorenc and Woda's [160] work where the authors compare a web application hosting on a proxmox VM on-premise cluster and Amazon EC2 as an IaaS solution. According to their evaluation, one in three applications profits from the cloud hosting, whereas the others are more affordable in an on-premise deployment. Since they didn't use additional services at the cloud provider, e.g. databases, they state that the overall costs assessment is use case dependent.

Migration studies like Sáez and others [238] or Rosati and others [235] address the question whether a migration towards the cloud is beneficial w.r.t. performance and cost aspects compared to an on-premise hosting. Another performance comparison was recently done by Eickhoff and others [71] where they compared the latency requirement of Minecraft like games deployed to AWS and Azure cloud as well as on-premise. They found that different applications under different load settings perform best in varying environments. No general statement can be made

to favor cloud over on-premise hosting or vice versa. They further state that a fair comparison is often hard due to the many configuration options in the cloud which are not documented in detail. The predominant reason for this is that cloud providers do not want to share their *secrets* about the technical details of the infrastructure. These black boxes on provider side render an assessment of individual components of a cloud solution based on the used hardware impossible. Tools for a smart comparison of these black boxes are needed [45].

Another problem is that a mapping of the on-premise settings to comparable cloud offerings is hard to achieve which limits the validity of comparisons even more. Additionally, a locally configured environment can be fine-tuned due to the specific needs of an application whereas the cloud offers a fixed number of predefined configurations like the VM types of AWS or Azure cloud.

Empirical research on open source platforms in the FaaS domain, e.g. [11, 149, 197, 207], did not specify resource constraints. The experiments focused on the platform behavior when concurrent users request functions. Since we assume - due to the lack of experiment documentation - that the default settings of the respective platforms are used, a comparison of the included results is limited. Reasons are different resource allocation strategies, contention handling when more requests arrive than instances are up and running, latency and scheduling.

Overall we can summarize that there are comparisons of on-premise deployments and cloud services, but there is a lack of detailed performance comparisons with similar environments based on abstract measures.

6.4.3. Methodology

Before we can adapt our proposed scaling strategy to an open source platform, we have to select the public cloud provider's FaaS offering for our comparison. When selected, we need to understand its scaling algorithm of resources, where Table 3.4 provides guidance. The scaling of resources is unique for every public platform which limits the portability of insights gained for adapting other platforms. The next step in our methodology is to choose an on-premise FaaS offering. Since most platforms support a K8s deployment as can be seen in Table 3.5, we install the open-source platform on an on-premise hosted K8s cluster. Also a deployment to a managed K8s service like Google Kubernetes Engine (GKE) or Amazon Elastic Kubernetes Service (EKS) is possible. The drawback of the latter is that we cannot fine-tune the underlying hardware.

From our point of view, there are three important aspects for a clean experiment. First, we have to use homogeneous hardware for the K8s cluster (in an ideal case identically constructed machines). The next aspect is to check the configuration of each machine like the BIOS or Linux Kernel configurations to guarantee a stable performance with increasing resources. We introduced a way to calibrate machines in Section 5. K8s can also be deployed on a single, calibrated node by

installing a lightweight distribution where K3s[173] shows the most consistent performance across several lightweight distributions compared to K8s [35]. The last step is to provide an additional *QoS layer* to the deployed on-premise platform where a user can only select resource settings which are in relation to the selected scaling strategy of the cloud provider. This QoS layer is our answer to RQ3.4.

A side note here is to consider the first CPU equivalent of the public offering as well as of the on-premise hosting. Only multi-threaded functions profit from a resource increase above this level of resources. When comparing only single-threaded functions, the resource limits should be chosen beneath these limits for both environments.

To verify that the resource scaling strategy of the self-hosted open source platform is comparable to the cloud offering, we suggest to execute a demo function on both environments. We propose a CPU intensive function to have evidence. An assessment can be made if the results comply with the scaling strategy. The goal of this experimental step is not to find exact matches in execution time but rather to compare the trends.

6.4.4. Evaluation

6.4.4.1. Experimental Setup

For our experimental evaluation we chose AWS Lambda as public cloud provider to compare our on-premise results. Its scaling of resources and the price calculation is consistent, comprehensible and the predominant reason for including it in our experiment. We chose OpenFaaS as on-premise platform installed on K3s, a lightweight K8s distribution. As an experiment machine we used an Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, model 158 with 4 cores (named H90 in the following, already used twice). Due to the single node deployment, we circumvent the problem of heterogeneous hardware.

6.4.4.2. Calibration Step

We calibrated our machine following the calibration introduced in Section 5. Figure 6.7 shows the results graphically followed by Table 6.3 with the corresponding statistics.

Table 6.3.: Linear regression models for displayed graphs in Figure 6.7.

	Bare Metal	K3s + OpenFaaS
p-value	<2.2e-16	<2.2e-16
R^2	0.9981	0.9982
Intercept	-10.958	-10.473
Slope	54.920	54.708

[173] https://k3s.io/

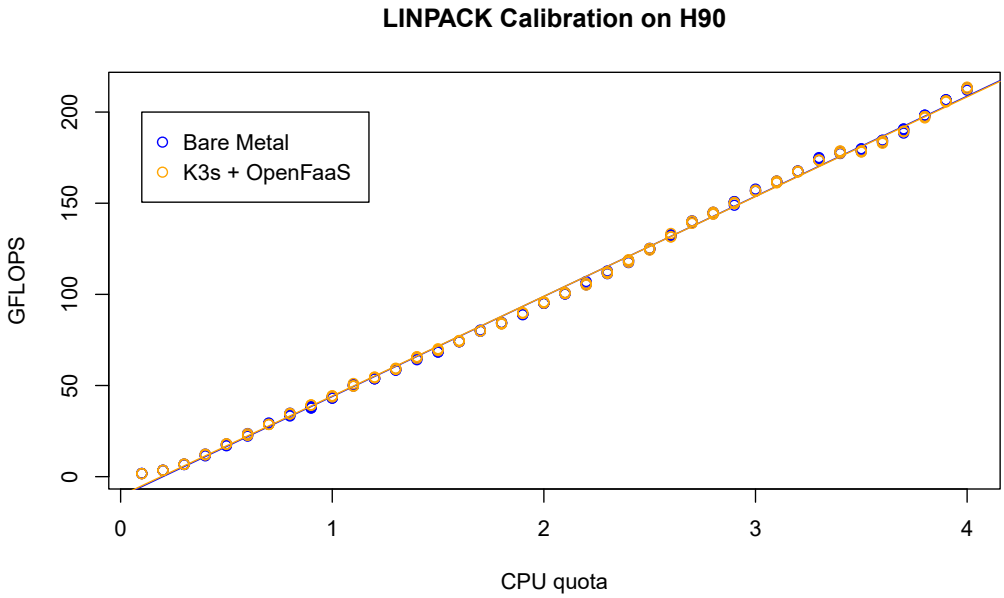


Figure 6.7.: Bare metal and OpenFaaS function performance of LINPACK executed on H90.

We first executed the functionality via Docker containers on bare metal without any other software installed except for the Ubuntu server image. The blue regression line and the dots in Figure 6.7 show the LINPACK results in 0.1 cpus[174] precision. The next step was the installation of K3s and OpenFaaS as documented[175]. Then OpenFaaS functions were deployed with resource constraints comparable to the Docker containers[176]. The result of these executions is plotted in orange in Figure 6.7. The reason for the bare metal and OpenFaaS calibration was to determine the overhead K3s and OpenFaaS introduce on the calibration results as well as to find potential anomalies of OpenFaaS and its scaling behavior. As can be seen from Table 6.3, there is no significant overhead which is in line with other research [35] nor anomalies in the scaling behavior of resources.

From data presented in Section 6.3 we already know the resource scaling strategy of AWS Lambda which is in relation with the documentation AWS provides. We executed LINPACK in 2022 when performing the OpenFaaS research at different memory settings to compare the used infrastructure with the experiments from 2021 when evaluated our simulation approach first. The predominant CPU model at eu-central-1 availability region is still an Intel(R) Xeon(R) Processor @ 2.50 GHz model 63 for the x86_64 architecture and the results for LINPACK at AWS Lambda revealed no changes which is reasonable since a public cloud provider does not change its infrastructure that often due to cost and sustainability reasons.

[174] A Docker command line option to restrict CPU resources: https://docs.docker.com/config/containers/resource_constraints/
[175] https://github.com/johannes-manner/SeMoDe/releases/tag/v1.2
[176] https://docs.openfaas.com/reference/yaml/#function-memorycpu-limits

As has been mentioned before, the one CPU equivalent is important for categorizing functions in single- and multi-threaded ones. From the AWS documentation we know that the one CPU equivalent lies at 1769 MB memory, where we achieve ∼33 GFLOPS computing power. At our local OpenFaaS deployment, the performance at one CPU is about ∼44 GFLOPS. This abstract computing power metric helps to compare the performance of different offerings.

6.4.4.3. Compare OpenFaaS and AWS Lambda Execution Trends

The next step before looking at the execution trends of the two selected platforms is to add a *QoS layer* by suggesting resource settings based on the measured GFLOPS during calibration. This additional layer resolves the problem of having no comparable resource settings when investigating different open-source FaaS platforms. It furthermore allows to transfer the scaling strategy of a public cloud provider to an on-premise hosting w.r.t. our QoS resource allocation.

To implement this QoS layer for the combination of AWS Lambda and OpenFaaS in our experiments, we extended our research prototype `SeMoDe`. For the evaluation in this work we specify our QoS layer with 70 settings ranging from 10 to 80 GFLOPS of abstract computing power for deploying our functions to AWS Lambda and OpenFaaS. Parts of the configurations can be seen in Table 6.4. AWS Lambda allows a continuous increase MB-wise. Locally, on H90, we can also implement the same scaling strategy based on the calibration step. We can continuously assign resources from 0 to the number of CPUs, in our case four. Based on the measured GFLOPS values, we get comparable configurations for the AWS Lambda memory settings as well as the resource limits for OpenFaaS.

Table 6.4.: Resource settings for the suggested QoS layer based on GFLOPS. OpenFaaS configurations are determined by the CPU shares, whereas AWS Lambda configurations are based on the configured memory.

GFLOPS	**OpenFaaS** in CPUs	**AWS Lambda** in MB
10	0.374	610
20	0.557	1119
30	0.740	1628
40	0.923	2137
50	1.105	2646
60	1.289	3155
70	1.471	3664
80	1.654	4173

With these indicated settings, we executed two functions on AWS Lambda as well as on OpenFaaS. The first one is a single-threaded recursive Fibonacci function implemented in JavaScript. The second function searches for prime numbers and is implemented in Java using a ForkJoin Pool to enable a concurrent execution on multiple CPUs. The same functions as in the previous section. For our data evaluation, we excluded the first run on a function instance for each setting.

This is especially necessary for a clean and fair comparison in the Java use case, where additional resources are consumed by the JVM during start up and by the JIT compiler.

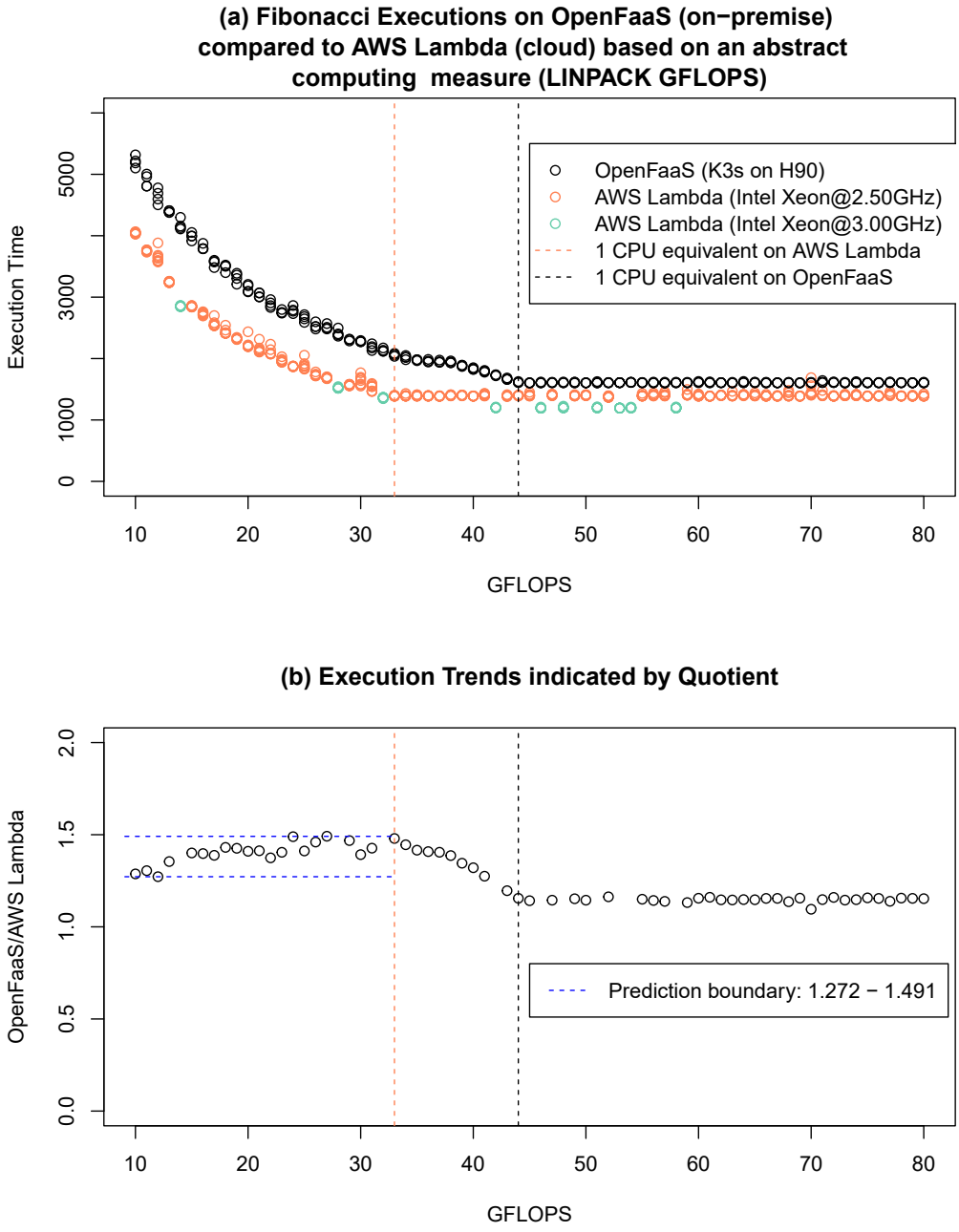


Figure 6.8.: Single-threaded Fibonacci executions on two environments, H90 (on-premise) and AWS Lambda (cloud offering). The function is implemented in JavaScript.

Figures 6.8 and 6.9 show the results of our Fibonacci respectively prime number executions. Locally, we executed the function five times and ten times on AWS Lambda. As already stated in literature [50, 86, 206, 213, 299], AWS uses different hardware for executing functions. By parsing the `/proc/cpuinfo` file, we extracted the CPU model name and the model version. In our experiments two Intel Xeon configurations were present. Since our calibration on AWS Lambda was ex-

clusively executed on 2.50 GHz machines, we excluded the 3.00 GHz executions before computing the trends in the second diagram of each figure. Obviously, as indicated by the executions at 42, 46 or 48 GFLOPS equivalent (the green dots in Figure 6.8(a)), these executions are faster than the executions on 2.50 GHz machines. Including these values would distort the comparison since we do not have calibration data for the 3.00 GHz machines yet. Each figure also includes an orange vertical line at ∼33 GFLOPS indicating the one CPU equivalent on AWS Lambda and another black vertical line at ∼44 GFLOPS indicating the one CPU equivalent on OpenFaaS deployed on H90.

These two CPU equivalents are important for an interpretation of the execution trends when computing the quotients in Diagram 6.8(b). The quotients are calculated by dividing the mean OpenFaaS execution time at a specific GFLOPS setting by the mean AWS Lambda execution time of the same GFLOPS setting. A trichotomy is present. The first segment until the first CPU equivalent on AWS Lambda shows a slight quotient increase when computing a linear regression model ($f(gflops) = 1.259 + 0.00688 * gflops$). We also included a prediction boundary for this specific function and language by stating the minimum (1.272 at 12 GFLOPS) and maximum (1.491 at 27 GFLOPS) quotient for this segment. Within the second segment, the computing power of OpenFaaS still increases whereas the resources assigned at AWS Lambda exceed the one CPU equivalent. The additional resources at AWS Lambda cannot be used by the function resulting in a constant execution time for all GFLOPS settings greater than 33 GFLOPS for the 2.50 GHz configuration. The result is a continuous drop in the quotients until the one CPU equivalent of OpenFaaS is reached at ∼44 GFLOPS (linear regression of the second segment: $g(gflops) = 2.448 - 0.0287 * gflops$). Within the last segment the deployed functions at both platforms consume more resources and therefore also the execution time of OpenFaaS executions remains constant. This behavior is also apparent when looking at the linear regression ($h(gflops) = 1.143 + 7.21 * 10^{-5} * gflops$). The trichotomy in this experiment emphasizes to think about the function characteristics as well as the implementation details of the platform w.r.t. the selected scaling strategy and the resource limits specified.

Figure 6.9 shows the execution points and trends of our multi-threaded prime number search function implemented in Java. The first diagram, 6.9(a), shows the execution time locally as well as on AWS Lambda. The comparable execution times are evident, which are also pointed out by the linear regression (intercept at 1.037 with an negligible slope $8.7/10^6$*gflops) within the legend of Diagram 6.9(b). Compared to Figure 6.8(a), where the local executions are always slower than comparable function instances on AWS Lambda, the programming language and therefore the execution on the machine - interpreted vs. compiled language - influences the distribution of absolute values in our setting with the specified hardware as previous research already showed [182]. The CPU equivalents indicated by the vertical lines have no influence due to the multi-threaded nature

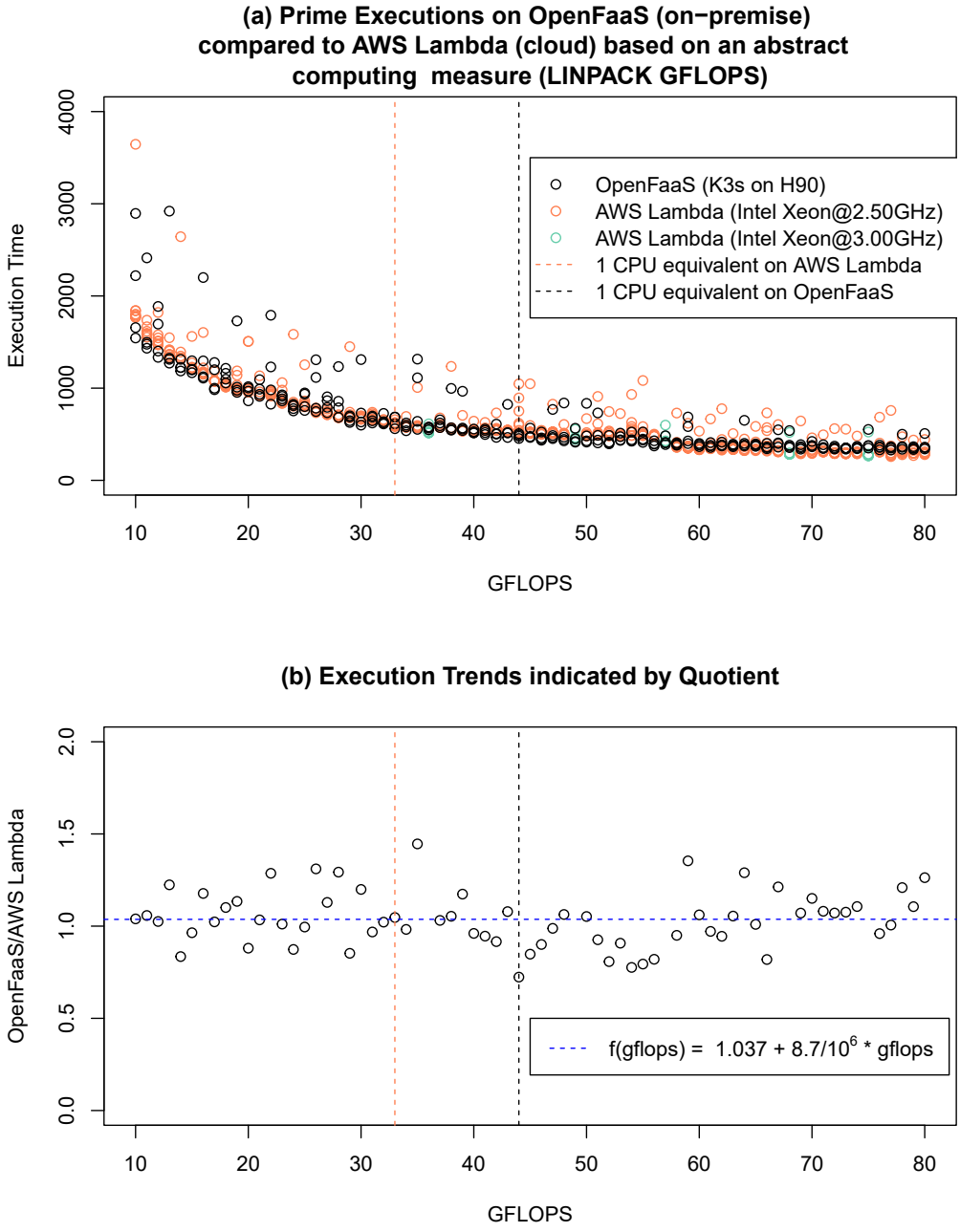


Figure 6.9.: Multi-threaded prime number executions on two environments, H90 (on-premise) and AWS Lambda (cloud offering). The function is implemented in Java.

of the function implementation. Function instances on both environments can make use of the assigned resources also when increasing the resources beyond a single CPU. As already mentioned, this is indicated by the linear regression in Diagram 6.9(b) where no trichotomy is evident.

6.4.4.4. Co-location of Functions: The Noisy Neighbor Problem

In FaaS research, the parallelism level of requests is often discussed due to the promised scaling property of platforms. However, when we think about the scaling of instances, we also have to keep in mind the pressure on resources of our machines. The co-location of workloads on the same machines without a proper separation leads to the possibility of performance losses also known as the noisy neighbor problem. In some scenarios, this can lead to starvation and increased response times. For public providers, these aspects have already been investigated in detail [17] whereas comparable studies for open-source platforms are not available to the best of our knowledge. In the mentioned paper, Barcelona-Pons and García-López [17] also state that there is no noisy neighbor problem present when starting highly parallel workloads on AWS Lambda. Therefore, we did not rerun a parallelism experiment on AWS Lambda. We rather focused on an experiment on our local node with OpenFaaS deployed on top of it.

The results of the two experiments can be seen in Figure 6.10. The first diagram 6.10(a) shows five concurrent users with the resource restrictions of our QoS layer (black dots) compared to the already presented data from diagram 6.9(a) where only one function at a time was running on the machine (orange dots). We repeated the experiments of the five concurrent users 10 times for each setting. Especially noticeable are the dots at 74-80 GFLOPS indicating zero execution time. At 74 GFLOPS, each function instance is configured with 1.544 CPUs. Since our machine has 4 physical and 8 logical cores, K8s only schedules new pods when resources are still available. After deploying four of the five pods, we reached that machine limit. We know from the K8s documentation, that if the resource demand is too high, the pods are in *pending* state for an *indefinitely* amount of time[177]. From this experiment we can state that OpenFaaS using the K8s limits enforces a resource aware scheduling of workloads to our machines. Due to the CPU intensive nature of our prime number function, we faced an increase in latency of up to 54.9% at 1.453 CPUs (69 GFLOPS) compared to the single executions.

In a second experiment 6.10(b), we varied the concurrency level from 1 to 20 concurrent users. Again we made 10 runs at each concurrency level and did not use the resource limits from OpenFaaS/K8s. We used the default behavior implemented. In contrast to the behavior with resource limits enabled there is no function which cannot be deployed. When looking at the pods specifications, we see the *qosClass* attributes of the pods are *BestEffort* where no resource guarantees are made at all. This is also the reason for the execution time of approximately 1270 ms for the first two concurrent users which is counter-intuitive compared to the execution data of diagram 6.10(a) with guaranteed resources/fixed limits. Furthermore, we see a continuous increase in median execution times starting at 1265 ms for a single user until 6702 ms at the concurrency level of 20 users.

[177] https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#specify-a-cpu-request-that-is-too-big-for-your-nodes

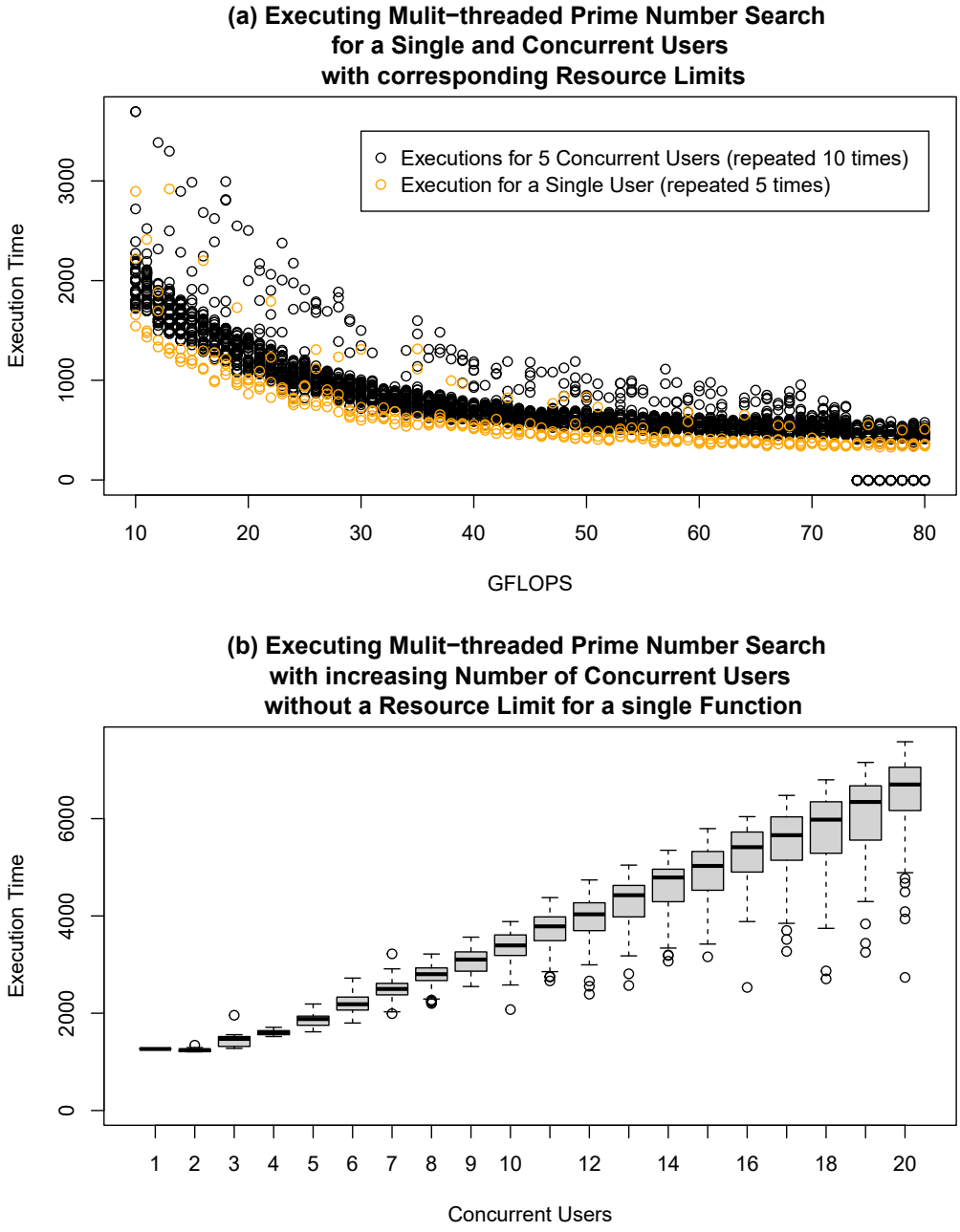


Figure 6.10.: Execution of multiple parallel instances of a multi-threaded prime number search implemented in Java with and without Kubernetes resource limits.

6.4.5. Summary of Implementing a QoS Layer for Open-Source Platforms

Most of the open-source FaaS platforms in our investigation use a K8s abstraction for deployment. Therefore, a resource allocation compliant scaling of resources is possible due to the K8s facilities. The resource limits are comparable to the Docker limits and can be used during deployment to configure pods. From the empirical open-source research we know that resource restrictions to functions are not applied. To adapt the resource scaling of a public FaaS provider, we suggest to execute a benchmark at the cloud provider as well as for the open-source

platform. We executed LINPACK at different resource settings and used the computed GFLOPS as an abstract measure to compare computing environments with each other and provide an answer to RQ3.4. This abstract measure also solves the problem of comparing heterogeneous setups with each other. We especially emphasize the difference between single-threaded and multi-threaded functions since FaaS and its enhanced billing and scaling model leads to situations where a resource increase does not improve performance any more when assigning resources equivalent to more than 1 CPU. Our scaling strategy and the usage of K8s limits guarantee resources on the specific machine which provides a solution for the noisy neighbor problem. Inference between the deployed pods is still present which is a difference to the architecture of AWS Lambda, where each function is performance- and security-wise isolated within a single VM[115].

In the motivation, when stating the problem of fair comparisons between public offerings and open-source alternatives, we cited an article where the authors assumed the cost for on-premise hosted software to be zero. Since this is not the case when looking at hardware procurement and operations, we were interested in suggesting a way to fairly bill an on-premise user to implement a profit center like organization in the IT department. AWS Lambda for example uses a linear pricing model, where doubling the resources means also doubling the price for the same amount of time. In an ideal, CPU intensive world, doubling the resources would result in halving the executing time resulting in a constant price for different settings as shown in Figiela and others, Figure 12 [79]. Since we adapt the same resource scaling strategy in our experiment as AWS Lambda, we suggest to also apply the same pricing scheme as the public cloud provider. The starting price level is dependent on the organization and the overall cost for procurement and operations.

A side aspect in our experiment were the differences in hardware used at AWS Lambda. We have seen in Figure 6.8 that the functions executed on the 3.00 GHz machines took 1198 milliseconds (after the one CPU equivalent) on average, whereas the 2.50 GHz machines took 1400 milliseconds. The former is therefore 14.43% cheaper than the latter when looking at our Fibonacci executions. It is not predictable nor can a user influence which underlying hardware is used but the empirical data from Section 6.4.4 reveal some oversimplified pricing behavior of the public cloud provider.

6.5. Discussion

6.5.1. Discussion of the Simulation Approach

In this section, a simulation approach to find a suitable configuration for a cloud function instance based on the requirements of the user was introduced. We overcome the tedious and costly process of deploying the functions first to a FaaS plat-

form, collecting execution data to understand the function characteristics and adjusting the cloud function configuration.

Different to other simulation research where the focus is often on the platform and the number of instances which are running based on a predefined workload, the proposed simulation approach deals with a single function in isolation. This limited focus is justifiable due to the scale-on-demand and stateless property in FaaS and platform implementations which fire-and-forget function instances on-demand without noisy neighbor effects. Basis for our proposed simulation process is a comparable execution environment as RQ3.2 indicates. Based on the calibration and the assumption that two environments can be made comparable based on an abstract computing measure, we computed linear regression models on the FaaS provider as well as on our experiment machine. A comparison and conversion from one resource setting to another is then possible when equalizing the two linear regressions. The evaluation in Section 6.3.3 confirms RQ3.3 in the sense that the resource configurations based on calibration lead to accurate predictions on a provider-hosted FaaS platform in the cloud. As seen by the presented data, the local predictions are relative to each other when comparing different resource settings. If a user of the simulation model wants to predict the execution duration on a FaaS platform, a few data points from the corresponding FaaS platform are still necessary to compute quotients. The trend curves in Section 6.3.3.4 show that it is possible to predict a range of ratios for best, worst and average case of the expected execution times on the provider when running the function locally. These ratios are different for the used programming language and local setup. The research prototype `SeMoDe` supports developers in choosing a suitable resource configuration. Details on the implementation and the UI are presented in Section 7.1. When reading related work and assessing the experiments included in detail, it was apparent that a lot of researchers did not consider multi-threaded functions and resource configurations which exceed a single CPU. When looking at the included multi-threaded functions and the comparison to single-threaded executions, it is important to consider this aspect and the performance gain when using more than a single CPU equivalent on the provider side. This section also wants to raise awareness that multi-core environments are differently treated in FaaS compared to, for example, PaaS.

Focused on a single functionality, utilization is fully dependent on the function implementation and not on the workload or other functions running within the same application. This aspect is unique and forces developers to think about the resource configurations which exceed a single CPU equivalent to fully utilize the resources they configure for the cloud function.

The methodology and evaluation provided is a first step to support developers to choose a proper configuration. In Section 7.1, advancements with regard to cost and multi-core environments are made in the web UI to provide all information needed for an informed decision about the cloud function configuration.

A last contribution of this section is the inclusion of on-premise hosted open-source platforms into this schema for resource-aware configuration of cloud functions. Based on the included related work in Section 6.4.2, there is no experiment conducted with open-source FaaS platforms where configuration of the executing instance, in particular a K8s pod, is comparable to public cloud offerings. A suggested QoS layer for open source tools, in our case OpenFaaS deployed on a single node K8s solution, enables the usage of resource limits and therefore dev-prod parity and provides an answer to RQ3.4. It furthermore solves also the noisy neighbor problem by limiting the number of deployed instances on a single node within the K8s cluster.

6.5.2. Threats to Validity

Based on the experimental nature of this section, there are a few threats to validity of the methodology and the results to consider. Some of the following aspects can also be worked on in future work:

- **Dev-Prod Parity** - The more similar the simulation environment is to the production environment on a provider platform, the better comparable are the results. In our approach we use a virtualized environment, where we execute all calibrations and simulations in Docker containers. AWS Lambda for example uses an additional VM layer to separate VMs from different tenants on the same physical host [1]. This does not lead to uncomparable environments but it should be kept in mind that the system stacks are different and should be adjusted in future work.

  The same holds for the deployment of the on-premise hosted open-source platform OpenFaaS. Here, the usage of K3s as a tool for using K8s features might influence the separation of cloud function instances where an overhead was present when comparing the execution data of a single function deployed to the multi-tenancy case.

- **Limited Scope** - Our evaluation was executed with a limited scope. We only use CPU intensive functions to evaluate our approach which is quite common in system research in the FaaS area [240]. The benefit of this strategy is better control over settings and functions which aids the interpretation of the results. Therefore, we decided to stick to the CPU-bound functions Fibonacci and prime number search.

  Another aspect of minimal scope is our focus solely on a single cloud function in isolation. A typical usage of FaaS, for example, is storing state in a database where requests and communication over the wire influence the overall application flow.

- **Single Provider** - Our evaluation is limited to a single provider in a single region (eu-central-1). There are other publication (e.g. [50, 206]) which show that different regions can make a difference - even for the same provider.
- **Sample Size** - As for each empirical evaluation the sample size is questionable. Overall since a single calibration run on H60 took round about 6 hours, we only conducted 25 of them, which is sufficient from our point of view since the deviation of the results is minimal. For the evaluation section, especially the prime number run, an experiment with more datapoints or a further statistical evaluation might disclose further insights.
- **QoS Layer** - Our methodology was prototypically tested for the combination AWS Lambda and OpenFaaS. AWS Lambda implements its scaling of resources linearly, which is not the case for other public cloud offerings like Google Cloud Function where a predefined set of resource configurations is available nor for the unpredictable scaling of Microsoft Azure. Therefore, scaling strategies of other providers need to be reviewed to determine whether they comply with their documentation before integrating them into our methodology.

6.6. Future Work

The discussion and presentation of a simulation based approach where the execution on a developer's machine reveals insights in the platform behavior of the function is an important step within the development process and deployment of cloud functions.

Dev-prod parity and the interaction mechanism with other services are especially interesting when using FaaS and influence the performance and the execution behavior. It is vital for cloud function use cases to interact with other services on the corresponding platforms. One direction of future work is therefore to simulate an application consisting of several functions and BaaS services, in particular gateways and databases, to demonstrate that the proposed simulation approach also works for applications and that individual functions are independent from each other performance-wise. A first proof of concept to simulate an application consisting of several services is given in Section 8.

Suggesting resource scaling strategies for open-source FaaS platforms which are comparable to commercial cloud offerings is a first step towards a fair comparison of open-source platforms with each other as well as with cloud platforms. Performance and also cost comparisons are possible with our approach. In the open-source FaaS area, the investigations of Section 6.4 raised three further questions. First we want to work on a pricing scheme for an on-premise hosted FaaS platform and develop a self-updatable price model for FaaS offerings dependent on abstract computing measures. Another aspect we want to work on empirically is the *fairness of billing*. AWS Lambda for example states in the documentation

that pricing is linear but due to different hardware used this is not the case. The hardware executing our function has a major influence on the execution time and consequently determines the price as can be seen by our experiments. Thirdly, we want to assess other possibilities for a stricter performance isolation for open source FaaS platforms. Based on our instance concurrency test, there is a need for on-premise K8s installations to rethink performance isolation. The restriction by limits is already a first and important step towards a more robust and predictable resource allocation but does not provide the performance isolation like separated micro VMs on AWS Lambda.

7. Decision Support and Guidance for Function Configuration

Parts of this chapter have been taken from [174, 180, 182, 185].

In this chapter, RQ4 *(How can developers be supported in making reasonable decisions about their cloud function configurations?)* and RQ5 *(Which factors influence the cold start behavior of a function besides the function configuration?)* will be answered.

Our research prototype `SeMoDe` is the practical realization of the introduced calibration and simulation ideas. It supports developers to select proper configurations for their functions via its web interface. Section 7.1 describes this web interface and includes references to other sections in the thesis which explain the shown functionality and link the paragraphs to their corresponding methodology and evaluation. Aspects which influence the especially important start-up behavior of cloud function instances are discussed in Section 7.2. The initial experiments published in 2018 and their repetition in 2023 are included. These two experiments showed the progression of FaaS.

7.1. Graphical User Guidance For Function Configuration Options

This section describes the process of using `SeMoDe`'s web interface for exploring function configuration options by a local simulation process. To do so, users have to (1) calibrate their local development environment with the desired provider environment (see 7.1.1) by executing the provided calibration function locally and in the cloud; (2) configure the mapping of calibration configurations from the previous step and specify desired simulation scenarios (see 7.1.2); (3) perform simulations locally and investigate the results to decide on a suitable function configuration (see 7.1.3).

7.1.1. Calibration

The configuration of the local calibration is already shown in Section 5.6, Figure 5.2. Figures 7.1, 7.2 and 7.3 show further screenshots for the `setups/{setupName}/calibration` endpoint. These diagrams are a useful tool for developers in

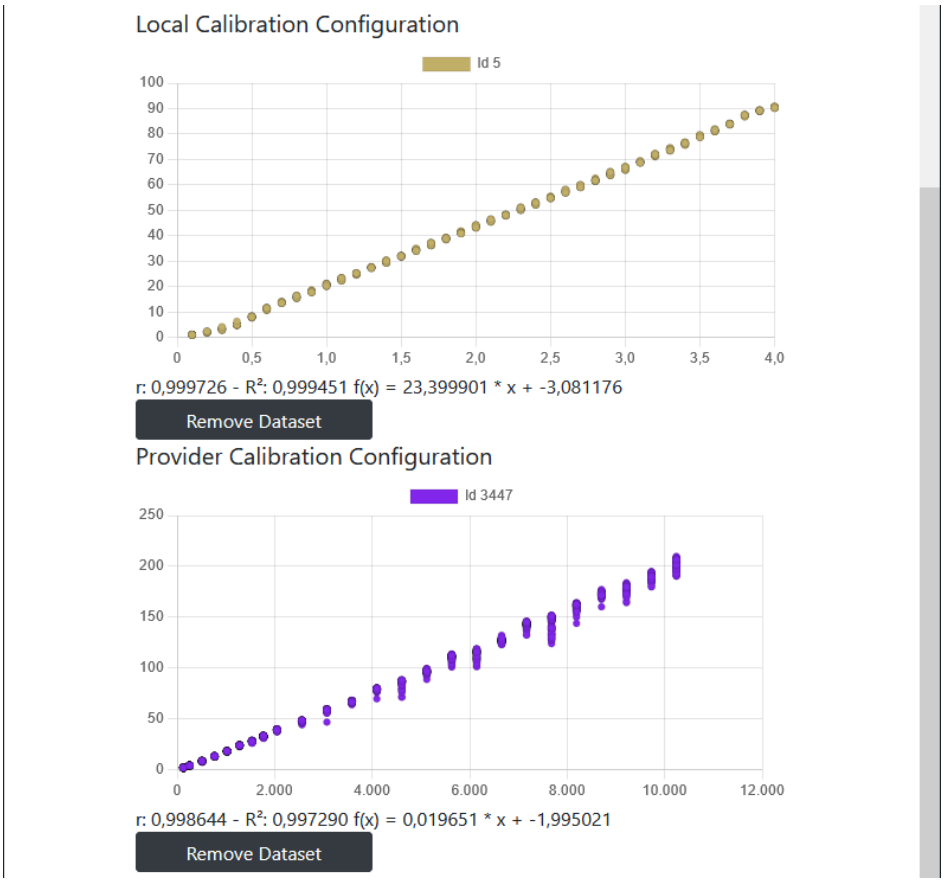


Figure 7.1.: Local and provider calibration graphs together with their linear regression models.

different stages of the simulation experiments to support them to select a proper resource configuration. A calibration of the corresponding public cloud provider - in our case AWS Lambda - is supported by `SeMoDe`'s benchmark feature which was already introduced and explained in detail in Section 4.3, Figures 4.6 and 4.7. In the AWS configuration, the only change compared to the benchmark functionality is the *Bucket name*, that is, where the output of the calibration function should be stored. The mechanism here is different to the benchmarking example since the API Gateway runs into a timeout and cannot get the response directly from the platform. Additionally, the `LogHandler` for AWS does not parse the execution data of the calibration function. Therefore, the result of the calibration on AWS Lambda is fetched by `SeMoDe` from this bucket and analyzed via the `LinpackParser` class.

In Figure 7.1, a local and a provider calibration are rendered based on the execution data. The statistical evaluation based on a Pearson linear regression model is also shown. It is used in the further process of mapping environments to each other. Linpack is currently the only supported option to calibrate the hardware stacks. Other calibration functions can be included by customizing the source code directly respectively the image at DockerHub[178].

[178]https://hub.docker.com/r/jmnnr/linpack

7.1.2. Mapping

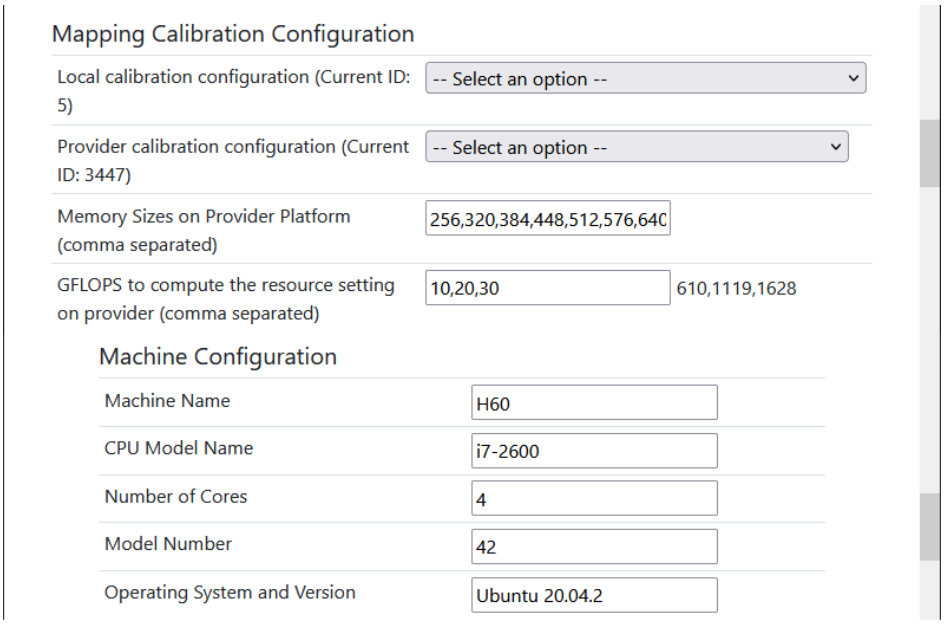


Figure 7.2.: Mapping step for preparing equivalent settings for local simulations.

The next section in the webpage includes a mapping step from the local calibration (and therefore the local machine stack) to the provider calibration. Both calibrations have to be executed to allow a mapping from one application stack to the other based on Equation 6.3. It is initiated via the two dropdown fields which allow a selection of different configurations to be mapped. This is especially interesting when changing BIOS or kernel settings to influence the CPU frequency scaling as discussed in detail in Section 5. The currently selected calibrations are displayed in brackets and their data is used for generating the two diagrams from Figure 7.1. The next input field is for public providers, in the depicted case AWS Lambda, where the memory settings which should be simulated are entered in a comma separated format. The tool then automatically computes the cpus value for the local simulation. If simulations have already been executed or if a user of the tool wants to achieve a specific abstract computing value, the reverse is also possible: A GFLOPS settings is specified by the user and the corresponding provider memory setting is computed as shown in Figure 7.2. In the case shown, memory equivalents for 10, 20 and 30 GFLOPS are required which - based on the mapping information - would result in 610, 1119 and 1628 MB on AWS Lambda. The fields grouped under *Machine Configuration* are for documentation purposes to describe fair and repeatable benchmarks with as much information as possible. The *Number of Cores* field is used for computing the full CPU equivalents and displaying them in the simulation diagrams. From the previous figure we already know that 30 GFLOPS at our local machine H60 exceeds the one CPU equivalent. Therefore, only a multi-threaded function would correctly forecast the runtime behavior for the mapped 30 GFLOPS/1628 MB.

7.1.3. Simulation

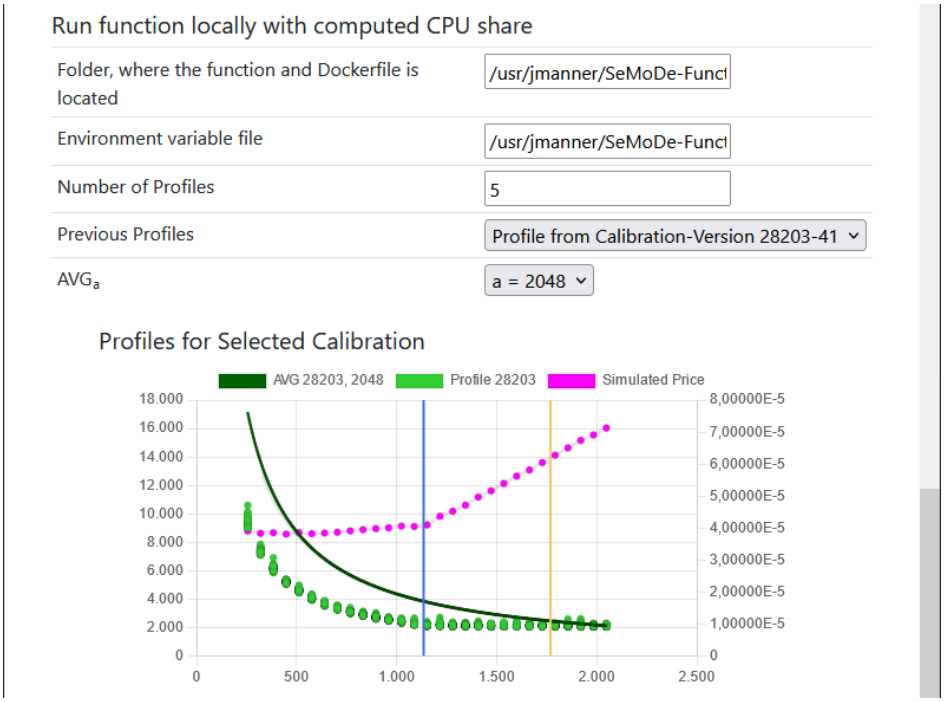


Figure 7.3.: Local simulations to assess the runtime characteristics for a single-threaded Fibonacci cloud function.

Figure 7.3 shows the additional configuration parameters which are necessary to start the local simulation. A user describes the folder where the source code of the function under test is located. Environment variables are set by a local file containing the variables to start the containers. In the example in the screenshot, `SeMoDe` starts the function five times - configurable value - for every `cpus` value based on the configured memory settings. These three input fields are the needed configuration to start a local simulation run. A prerequisite is that the mapping is configured beforehand. The `ContainerExecutor` class configures the containers with the correct `cpus` quota and runs the simulation. As a result of the container execution, the information is parsed, saved in files at `/setups/{setupName}/-calibration/profiles` and stored as `ProfileRecords` in the database.

Via the dropdown menu *Previous Profiles*, a user can then select a profile and can also see the version number of the calibration. Based on the machine configuration from the previous figure, in particular the *Number of Cores* input field, the orange vertical lines are drawn, indicating the full CPU equivalents. Figure 7.3 shows three result sets. The primary y-axis shows execution times in milliseconds, whereas the secondary y-axis predicts the cost of executing one request at AWS Lambda based on the simulated execution time and the valid pricing for the AWS region eu-central-1. The data in light green is the profile of the simulation values with the calibration id 28203, version 41. It is based on the same data as the middle part of Figure 6.3 where we evaluated our methodology with the statistics program R. The curve in dark green is based on Equation 6.4 and depicts the optimal per-

formance/cost function based on AVG_a, in this case the mean execution value for 2048 MB.

To recap, we take a linear cost and resource scaling model as a basis where doubling resources results in a situation of halving the execution time. Since the cost model scales equally, this would result in constant cost for arbitrary settings. As already mentioned in Section 6.3.3.3, memory settings below the dark green line would be slower and save money compared to a, whereas a memory setting above the line would be faster but also more expensive. As shown in Figure 7.3, all other settings would be cheaper. The magenta dots are the calculated price for the simulated execution time for a single function when using the pricing scheme of AWS Lambda. As can be seen, the price is constant until the one CPU equivalent on H60 (blue vertical line) and then increases as shown in other research, e.g. [72]. We know from the discussion of the simulation approach that the function characteristic would not profit from a resource increase beyond the first CPU equivalent which would be 1769 MB at AWS Lambda.

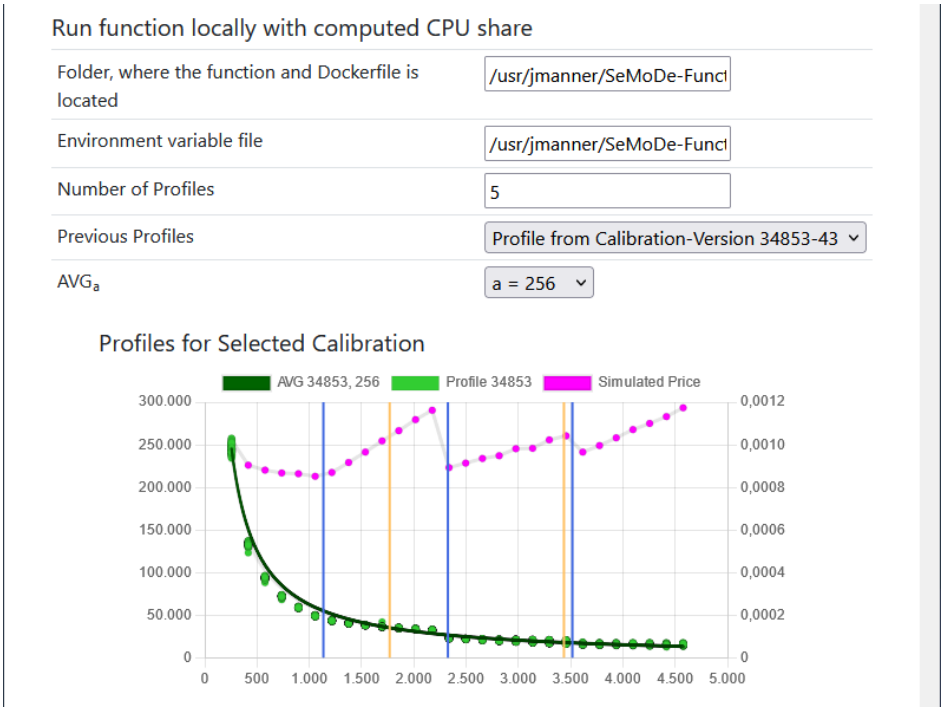


Figure 7.4.: Local simulations to assess the runtime characteristics for a multi-threaded prime number search cloud function.

A simulation for the multi-threaded prime number search in Figure 7.4 reveals a different picture as resources are fully utilized. The color scheme is the same as for the previous screenshot. It is especially noteworthy that the predicted cost increases when the next CPU equivalent for the local machine setup - depicted by the blue vertical lines - is reached. As already mentioned in Section 6.3 possible reasons for this behavior could be scheduling overhead when the CPU becomes busier.

Figures 7.3 and 7.4 are parts of SeMoDe's web interface for implementing a simulation framework. The diagrams and the explanation provided here support developers during the development process to select a proper resource configuration for their functions. This provides an answer for RQ4.

7.2. Guidance for Improving Cold Starts

7.2.1. Motivation

Scale-on demand, one of the FaaS characteristics discussed in the conceptualization of this work in Table 3.3 comes with an inherent problem. The first execution of a cloud function experiences a cold start since the container has to be started prior to the execution which causes a slightly longer runtime. Due to performance reasons, FaaS providers do not shut down a container immediately after its function is done running. Subsequent executions use spawned containers to profit from the *warm* execution environments. Avoidance of idling and scaling on demand are game changers compared to other cloud service models, but entail more cold starts.

So far, cold starts are perceived as a system-level challenge [10, 161]. There are several ideas on how to circumvent the cold start problem. One *solution* is to ping the cloud function on a regular basis [145, 273]. Pinging keeps the function instance(s) warm so that subsequent requests can reuse these instances to profit from an already provisioned execution environment. Unfortunately, this *ping hack*[179] is opposed to the scale to zero principle of FaaS. Another solution are preconfigured function instances which public cloud providers offer with new releases of their platforms. They are ready to serve requests immediately. However, this also comes with additional cost and doesn't fit with FaaS characteristics, i.e. scale to zero and pay-per-use.

Therefore, we are motivated to research factors that limit the cold start overhead and pose the following research question RQ5 *(Which factors influence the cold start behavior of a function besides the function configuration?)*. Based on this question, we formulated influencing factors as hypotheses and used SeMoDe to execute corresponding benchmarks for each hypothesis. The experiments included in this section were partly published in a paper in 2018, three years after the initial release of AWS Lambda. For detailed information about the original work, we refer the interested reader to our paper [182]. The time difference between 2018 and 2023 as well as the lessons learned within these five years, like the checklist for performing experiments in Section 4.2, motivated us to redo some of the experiments and compare the two data sets with each other. Besides the results obtained, the experiments show also the usefulness of the simulation approach proposed in the work at hand.

[179] https://www.jeremydaly.com/15-key-takeaways-from-the-serverless-talk-at-aws-startup-day/

The agenda for this section is as follows: Section 7.2.2 lists hypotheses about cold start influencing factors followed by a discussion of other cold start investigations in the FaaS domain. The aforementioned hypotheses are tested in experiments in Section 7.2.4. The results are presented in the subsequent paragraph and discussed in Section 7.2.6. Some ideas for future work conclude this section as well as the current chapter.

7.2.2. Hypotheses

To facilitate an unbiased evaluation, hypotheses about the implications of different parameters and decisions were formulated prior to the experiment. FaaS users and especially providers make decisions which may influence the cold start behavior of the executed functions. These decisions include the programming language, the deployment package size, the memory setting, number of dependencies, or the workload, therefore influencing the number of instances running in parallel. Prior executions and container shutdown intervals are also of relevance. This list of influential factors is not complete but contains the most important factors:

H1: Programming Language
FaaS platforms offer a large variety of programming languages [161]. JavaScript (JS) for example is supported by all major platforms since it is a perfect fit for small, stateless cloud functions. Also compiled languages like Java and C# come into focus due to the engineering benefits for more complex functions. Because of the environment overhead, our hypothesis is that compiled programming languages impose a significantly higher cold start overhead than interpreted languages like JS. For instance, the execution of a cloud function written in Java needs a running Java Virtual Machine (JVM) which must be set up prior to function execution.

H2: Deployment Package Size
We assume that the cold start overhead increases with the deployment package size. We want to measure the time, which is needed to copy the function image of different sizes to the container, load the image into memory, unpack, and execute it.

H3: Memory/CPU Setting
Our hypothesis is that the cold start overhead decreases with increasing resources because the container can be loaded and set up faster. We assume that this behavior is observable for low memory settings where the CPU is busy, but is negligible for high settings since the CPU is underutilized. This limitation does not weaken the hypothesis because the low memory settings starting at 128 MB are of particular interest since a low memory configuration is often used to implement glue code. Memory and CPU are used in combination since most of the mature platforms offer a linear scaling of CPU power based on the memory setting.

H4: Number of Dependencies
Loading dependencies takes time when spinning up a cloud function. Our hypothesis is that the amount and size of dependencies increases the cold start overhead since they must be loaded prior to the first execution and can be reused in subsequent ones. If we can confirm this hypothesis, a best practice would be the division of required libraries in sublibraries so that the needed subset of functionality is extracted in a new artifact.

H5: Concurrency Level
FaaS gets attention especially due to the scaling property of cloud functions. We hypothesize that the concurrency level, i.e., the number of concurrent requests and therefore started containers, neither influences the cold start overhead nor the execution time of a single function. Functions are started independently of each other in a separate container for every concurrent execution. If 1000 requests arrive simultaneously, we expect that 1000 containers are started by the middleware of the FaaS platform.

H6: Prior Executions
Avoidance of idling is a significant improvement of FaaS compared to PaaS. Achieving this goal comes with the drawback that unused containers are removed from running machines. Subsequent calls to the cloud function require a new container. Hence, we assume that the cold start overhead is independent of prior executions. This hypothesis is of particular interest for the first execution of the cloud function after deployment.

H7: Container Shutdown
Providers might optimize their infrastructure by using learning algorithms for identifying cloud functions which are used frequently. Due to cost effects and user satisfaction, we hypothesize that the duration after which a container shuts down is dependent on the number of previous executions. According to the FaaS paradigm, executions are independent of each other and should not influence the lifespan of a container.

7.2.3. Related Work

An early work on cold starts for Java functions on AWS revealed that small deployment packages take less time than bigger ones [224]. In their work, Puripunpinyo and Samadzadeh only had two different package sizes which induced us to look at this dimension in a more structured way resulting in the formulation of H2. Scaling the number of concurrent functions has been the focus of other experiments, however, they only noted the delays for the 99^{th} percentile of four public cloud providers without considering the resource configurations of the platforms, in particular the memory setting [145, 196]. Another recent experiment deployed Monte Carlo workflows to three major platforms where each function was assigned 256 MB of memory [231]. They documented the memory setting

chosen but did not discuss different languages or other resource settings. Several function states from the first invocation on a provider's platform, over the first invocation on VM and container, to the warm execution were in focus of another experiment [157]. Comparing these four states with each other, they measured up to 15 times overhead when comparing the first invocation on the platform with a warm cloud function instance.

The ping hack was also an inspiration for performing structured experiments. Lloyd and others introduced keep-alive workloads to prevent the FaaS platform from scaling instances down [158]. Another approach suggests a pool strategy [155] where a number of instances are kept for serving a baseline of requests. An inherent problem of all these cold start mitigation strategies [271] is the anticipation of future workloads which are typically unknown for the FaaS provider. When considering a chain of functions, the situation changes and a user can anticipate which functions will be called in the future. The first function in the chain still experiences a cold start but the other functions within the chain can be anticipated. This is the basic idea of an approach implemented by Bermbach and others [24] to reduce the overall number of cold starts for the application.

7.2.4. Experiments

7.2.4.1. Selection of Experiment Dimensions

The experiments in this section evaluate three out of the seven hypotheses. *Programming Language*, *Deployment Package Size* and *Memory/CPU Setting* are investigated. These three hypotheses were already tested in 2018 and repeated in 2023. The reasons for choosing these hypotheses are the ease of testing and the likelihood of getting stable and reproducible results. Back in 2018, it was the first benchmark to focus only on cold starts. Therefore the aim was and still is to make a clear experimental setup and reduce other parameters and external influences to a minimum. Hypotheses with a concurrent notion (i.e. H5-H7) are omitted due to the side effects which are introduced by concurrency in general. The data base produced by our sequential benchmark has a minimum set of external influences and could serve as a data base for follow-up experiments. Therefore, our benchmark is of special interest for real world applications which are only requested once or twice per hour and thus benefit from the scaling to zero property.

We selected Java and JS as programming languages. An important reason for this decision is, that Java is a compiled and JS an interpreted language. This selection emphasizes the differences in programming languages for the evaluation of the programming language hypothesis. Furthermore, Java and JS are widely used in enterprises and the open source community[180]. Additionally, we checked the influence of the assignment of resources with regards to the cold start period. We chose the recursive version of Fibonacci as it has already been done

[180]https://octoverse.github.com/2022/top-programming-languages

in other experiments like in the evaluation of our simulation in Section 6.3.3. Therefore, the algorithm is well suited for our benchmark assessing cold starts. In 2018, we assumed that the hardware used in a data center is identical [145], which would have improved predictability and low variance in function execution time and guarantees stable results. However, since this is not the case as already shown [50, 206, 213], one focus of the second execution of our benchmark is to split the results by machine type. The low memory usage ensures that we can benchmark the function with any of the memory settings provided by FaaS platforms.

Finally, we select the memory and package sizes to test our hypotheses. An *initial* package size is the size of a source package after the build phase. Initial Java packages have approximately 1.5 MB, JS ones are smaller than 1 KB. The following package sizes, which differ from the initial ones, are artificially increased by adding a file to the zip archive or increasing the JS file with a comment. Deployment packages were initially sized 3.125 MB, 6.25 MB, 12.5 MB, 25 MB and 50 MB. For AWS, 50 MB was the upper package size limit for functions at the time of the experiment (June 2018) but has now increased to 75 MB (default in May 2023)[181]. Due to some errors during the deployment of packages bigger than 40 MB, we decided to use the aforementioned settings and reduce the last package size to 40 MB. The next step was to compute linear regressions to get a prediction for other settings as well.

The memory setting was configured on AWS with 128 MB, 256 MB, 512 MB, 1024 MB, 2048 MB and 3008 MB. The memory setting linearly determines the compute power of the container. Every combination of deployment package size, memory, language, and provider resulted in a cloud function. Therefore, we deployed 72 cloud functions on AWS.

Our experimental setup is designed to exclude side effects. Calculating the execution overhead (cold – warm) as logged by the client isolates the perceived cold start overhead. The average execution time of the function (recursive Fibonacci calculation), network latency, and routing within the FaaS platform is assumed to be equal for cold and warm executions and therefore irrelevant for the cold start overhead value. The remaining duration results in an isolation of the additional time consuming steps, which occur during a cold start.

7.2.4.2. Experimental Setup

Due to the specific focus on cold starts, the aim of the experimental setting is to force a cold start closely followed by a warm start on the same container instance. In our setup, a warm start is defined as the reuse of a container. Given that there is only a single cold start per container, having a pair with a single cold and a single warm execution guarantees a sound comparison because calculating the mean over several warm executions is avoided. Such mean calculations

[181] https://docs.aws.amazon.com/lambda/latest/dg/limits.html

could have distorted our results because platforms can optimize the performance of cloud functions after a certain amount of invocations, as we observed during our initial experiments. Tests in 2018 have shown that containers on most platforms were shut down after at most 20 minutes of idling, but there is no guarantee nor documentation when a container is shut down.

A FaaS platform is a black box. The precise execution duration, which is used for billing on the platform, includes the function execution and parts of the start up process. Other parts of the initialization and start up of the container plus other needed infrastructural components are not included. To measure these aspects, we performed a REST-based interaction with the FaaS platform to also log the start and end time on the client side as discussed in Section 4.4.

Logging the time stamps locally enables us to compare the local execution with the platform duration. After storing the starting time stamp, a REST call is executed which sends the request over the network to the API gateway endpoint. This endpoint creates a new event which triggers a container creation or reuse. Finally, the cloud function is executed and the middleware on the platform logs the start and end time of the function execution as well as the precise duration, which is the difference of both time stamps. The result of the computation is transferred to the API gateway endpoint, wrapped in a response, and sent to the caller via the network. The client REST call exits and the local end time stamp is logged on the host machine of the FaaS user. The two local time stamps enable an assessment of the perceived execution duration for the user and as a consequence the difference between cold and warm starts from the user's perspective.

To force pairs of cold and warm executions, we used the *SeMoDe* benchmarking mode *sequential with changing intervals* in 2018. This mode triggers the provided function with a delay between execution start times. Delays vary and are defined in a provided $array$ of $delays$ d in a round robin fashion. The platform response includes a container and platform identifier. These identifiers enable an unambiguous matching between the local REST data and the platform log data. The start time of each execution is generalized in Figure 7.5.

$$start(i,d) = \begin{cases} 0 & \text{if } i = 0 \\ start(i-1,d) + d[i \mod len(d)] & \text{if } i \geq 1 \end{cases}$$

Figure 7.5.: Start time of the i$^{\text{th}}$ execution of the local benchmark invocation.

We set our array d to {1 minute, 29 minutes}. The start time is the time of the local service which calls the API gateway. A representation of the resulting execution sequence can be seen in Figure 4.9. Once again, it should be noted, that the invocation of the cloud functions is sequential.

When performing the experiments in 2023, we decided to update the function description at AWS Lambda after the cold/warm execution pair to force the platform to start a new instance for the next call. The original experiments were executed between 6/25/2018 and 7/1/2018. The repetition of the experiments was

performed between 5/25/2023 and 5/31/2023. Each cloud function was invoked 550 times to get 275 pairs of cold and warm executions. If a cloud function returned 500 as HTTP status code, which indicates a server error, or if another error like an API gateway timeout occurred, we excluded the cold as well as the warm execution. Only pairwise valid data was processed and included in the results. In 2023 we collected the CPU model information. 99.3% of our JS functions and 99.1% of the Java functions were executed on Intel(R) Xeon(R) Processor @ 2.50 GHz. For clarity of the results, other entries executed on different CPU models were removed from the results. An investigation of the execution data of the other machine, the already mentioned 3.0 GHz machine, was not meaningful due to the few data points.

To summarize our setting, the resulting data matrix consists of six dimensions: Programming Language, Deployment Package Size, Memory Setting, Specific Invocation Time, Local Duration, and Platform Duration.

7.2.5. Results

7.2.5.1. Hypotheses Independent Results

Before we confirm or reject the selected hypotheses, we gather general insights from the data in this part. Figures 7.6 and 7.7 and Table 7.1 are based on the same dimension selection. The deployment package size is initial, all valid pairs of each cloud function are used to compute the figures and mean values. For AWS, if not noted otherwise, the cloud function with 256 MB memory is selected. In addition, the repetition of the experiments revealed improvements in the platform and a change in assigning resources.

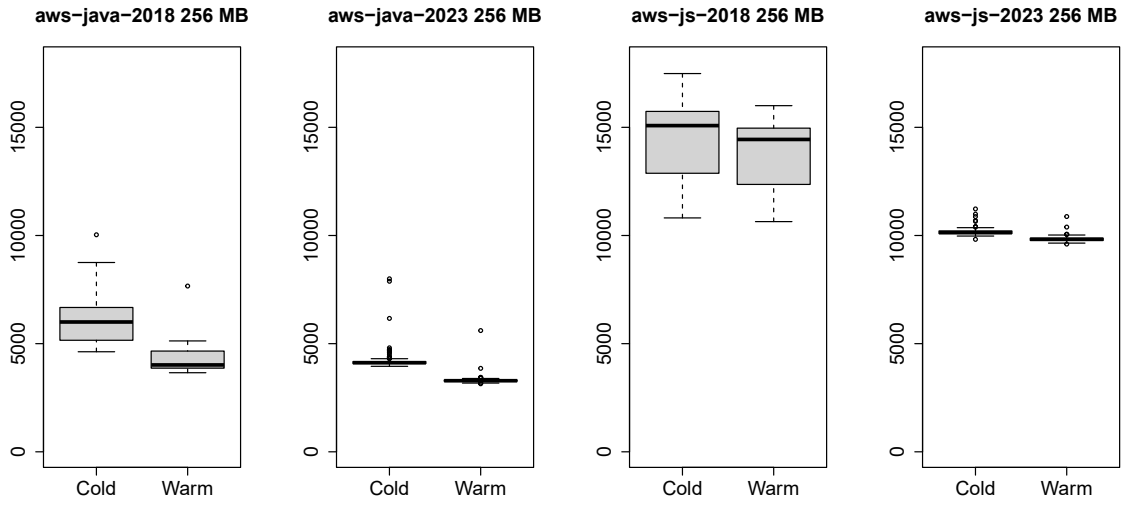


Figure 7.6.: Execution times of cold and warm invocations on client side, 2018 and 2023.

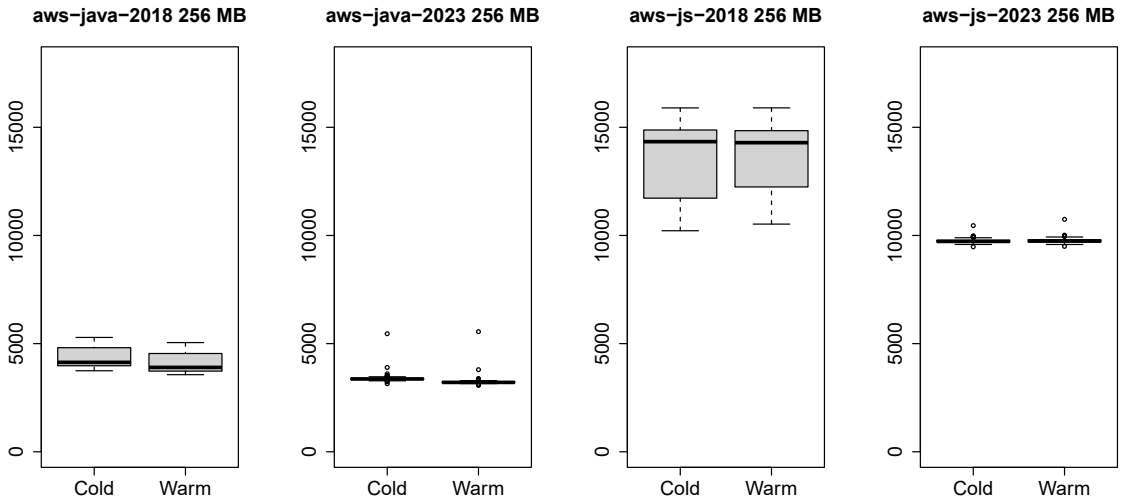


Figure 7.7.: Execution times of cold and warm invocations on provider side, 2018 and 2023.

Figure 7.6 shows boxplots of the function execution duration in milliseconds, measured as the delta of the start and end time of REST calls on the client. The bottom of the box is the 25th percentile and the top is the 75th percentile. The center line of the box is the median equal to the 50th percentile. Upper whisker is the 75th percentile plus the box length multiplied by 1.5. Corresponding to this computation, the lower whisker is the 25th percentile minus the box length multiplied by 1.5. Values that are not between the two whiskers are outliers and depicted as dots. This procedure was also chosen for the generation of Figure 7.7. The values of these boxplots were fetched from the logging services of the respective platforms.

As can be seen in the diagrams, the values for the cold executions compared to corresponding warm ones are consistently higher on the client (Figure 7.6). These values from the client can visually be related to the platform ones (Figure 7.7) because the box plots in the two figures are based on exactly the same raw data and also the y-axis dimension is the same for both figures and the box plots. Due to this visual coherence, some outliers are no longer included in the figures. The raw data, the box plots as printed here and the box plots including the outliers are accessible online [176][182]. Sometimes cold executions are faster than warm ones which is especially evident in the AWS values in Figure 7.7, which show a huge duration intersection of cold and warm executions. The warm and cold values for AWS JS are even seemingly equal.

To get more insights about absolute values, Table 7.1 presents mean values in milliseconds measured on client and platform side. The dimensions of the presented data are programming language, the location where the data is gathered and

[182]https://github.com/johannes-manner/SeMoDe/releases/tag/wosc4

Year	Language	Client/Platform	Cold	σ	Warm	σ	Difference
2018	Java	Client	5961	875	4211	465	1750
2018	Java	Platform	4329	450	4082	434	247
2018	JS	Client	14320	1894	13676	1742	644
2018	JS	Platform	13496	1782	13539	1738	-43
2023	Java	Client	4187	403	3304	166	883
2023	Java	Platform	3381	151	3230	166	151
2023	JS	Client	10163	142	9834	110	329
2023	JS	Platform	9743	84	9757	98	-14

Table 7.1.: Mean values in milliseconds for cold and warm executions on client and platform side in 2018 and 2023.

initial deployment package size, which means that no artificial data is added to the artifact. To assess the quality of service, the two experiment periods can be compared with each other. They show the distribution of values and the standard deviation σ for cold and warm execution periods.

We measured an execution overhead for the cold start of 1,750 ms (cold – warm) on the client as opposed to 247 ms on the platform for Java. JS overheads were smaller with 644 ms on client and even -43 ms on the platform in 2018.

We observed that some cold executions on the platform are faster than the warm executions on the same container. This is the reason why the value for JS is negative in this case. For JS, 63 % of the cold executions were faster than the corresponding warm ones. 16% of AWS cloud functions written in Java were executed faster on the cold start compared to the warm execution on the same container. These start and end times were logged on the platform. On the client side, a cold execution is never faster than its corresponding warm execution, neither for JS nor for Java.

Our conclusion for AWS is, that typical tasks during container start up etc. are not included in the logged value on the platform. Our results strengthen this assumption. Java needs a more resource intensive environment with an initialization of a JVM during the cold start, whereas JS uses only an interpreter to execute the code. We assume that underutilization when executing the cold request, collocation of various cloud functions on the same host, and other reasons influence the performance as well.

There are a few points which are especially interesting when comparing the 2023 gathered data with the measurements from 2018. The distribution of the data is narrower as indicated by the standard deviations in Table 7.1, most prominent for the JS executions. Also the overall execution time decreases by around 30%. The most obvious reason would be a different approach of scaling and assigning resources to the cloud function instances. This assumption can be confirmed when looking at the provider execution times for different memory settings in Figure 7.8. Here the trend of lower execution times as well as scaling strategies is apparent. It is important to remember that the constant execution time for the 2023 lines for 2048 and 3008 MB is explainable by a single-threaded Fibonacci

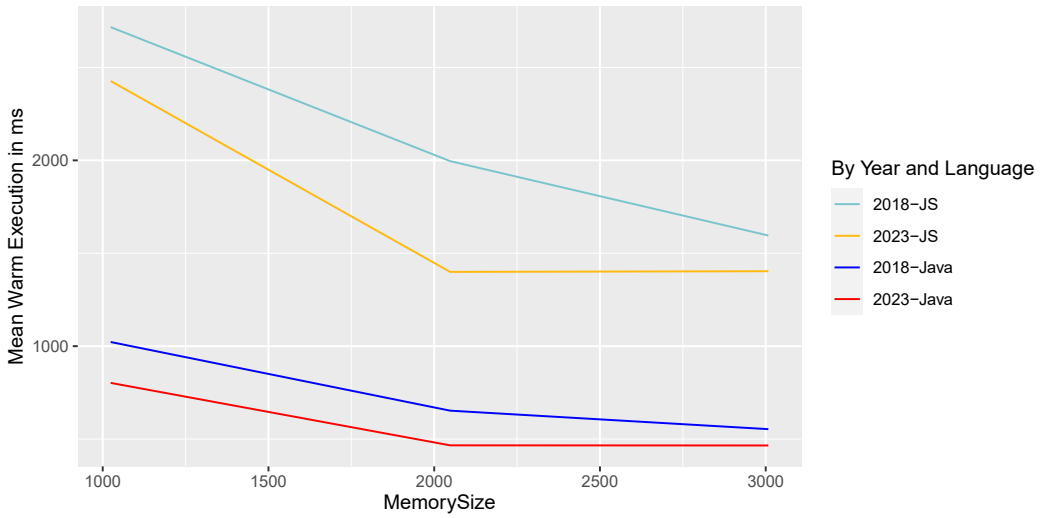


Figure 7.8.: Mean execution time for memory settings 1024, 2048 and 3008 MB in 2018 and 2023 for recursive Fibonacci for the second request to a cloud function instance.

implementation (one CPU equivalent 1769 MB). On the other hand, the strategy of assigning resources in 2018 was different since there is a decrease between the last two memory settings. It is likely that more powerful machines have been used in 2023 since AWS changes their servers typically every four years based on their annual report to investors[183].

Another general insight can be gathered from the overall execution time. The same functionality, recursive Fibonacci, is compared for two different programming languages. This could be seen as a small programming language benchmark on AWS Lambda. For a more thorough investigation of different programming languages, the interested reader is referred to a study which compares 27 languages [216]. The choice of the *right* programming language can lead to better response times and lower cost. Since FaaS has a cost model with a fine-granular pay-as-you-go scheme, Java would be the better choice since it provides the same result in a third of the time and therefore saves a considerable amount of money. As already remarked in Section 3, billing on a millisecond basis makes a big difference to the other established cloud service models, especially IaaS and PaaS, considering the very low execution time.

In order to evaluate our hypotheses, we need to know the total execution overhead of cold starts. This is the reason why for all further analyses we only consider the execution times logged on the client.

[183] https://d18rn0p25nwr6d.cloudfront.net/CIK-0001018724/f965e5c3-fded-45d3-bbdb-f750f156dcc9.pdf

7.2.5.2. Hypotheses Dependent Results

To assess the hypotheses dependent results, we use mean values but more often a correlation metric to make a clear statement as to which degree the measured data is significant.

The correlation coefficient ρ ranges from negatively correlated (-1) to positively correlated (+1). There are different interpretations considering the significance of correlation. We stick to a widely used interpretation [301], where 0 indicates no correlation, an absolute value of 0.2 weak, 0.5 moderate, 0.8 strong and 1.0 perfect correlation. Additionally, we constructed a linear regression model to calculate the slope of the regression line plus the intersection point of the y-axis. This enables us to formulate an equation to compute other configurations than the investigated ones. Especially for the deployment hypothesis H2, this approach can forecast arbitrary package sizes. The resulting linear models and the correlation coefficient ρ are presented in Tables 7.3 and 7.4. The slope of the line is no indicator for correlation, but states, how strongly y is influenced by an increasing or decreasing x.

H1: Hypothesis Programming Language Cold start overheads for different memory settings for Java and JS functions are shown in Table 7.2. Cold start times for Java cloud functions are between two to four times higher than those of respective JS functions. Based on the input we used for executing the Fibonacci function (n=40), the cloud functions implemented in Java are still faster than the JS counterparts when looking at the overall execution times not solely on the overhead cold starts introduce. For other input parameters, this might change. As for the standard deviation, the cold start overhead also shows a narrower distribution especially when looking at the JS overheads for the data in 2023. Here the data even shows a constant overhead of around 320 milliseconds and indicates that the execution environment does not stress the executing machine even with the lowest resource configuration. As remarked, this could be a baseline for further experiments how costly starting a new execution environment is from a client perspective. Based on these ratios, the presented data supports the hypothesis H1, as we noticed that the cold start time was significantly larger for each of the Java functions compared to JS.

		Memory in MB					
Year	**Language**	**128**	**256**	**512**	**1024**	**2048**	**3008**
2018	Java	1887	1649	1240	1119	1271	899
2018	JS	587	644	614	368	589	371
2023	Java	1070	883	816	761	696	627
2023	JS	315	329	310	321	327	317

Table 7.2.: Differences of cold and warm executions on the client side for hypothesis H1 considering programming languages.

H2: Hypothesis Deployment Package Size The next hypothesis is that the deployment package size, in particular the size of the uploaded archive, influences the cold start overhead. The assumption is that transmission from a bucket store to the executing environment increases linearly with increasing package size. As a reminder for interpreting the following data, the resources at AWS Lambda are configured with 256 MB memory and the overhead on the client is calculated by computing the difference between the cold and warm execution of the same cloud function instance.

Year	Language	ρ	Linear Model
2018	Java	0.29	$1510\,ms + 9\,ms/MB$
2018	JS	0.37	$613\,ms + 12\,ms/MB$
2023	Java	-0.14	$882\,ms - 0.45\,ms/MB$
2023	JS	0.74	$359\,ms + 13\,ms/MB$

Table 7.3.: Spearman's correlation coefficient ρ and linear regression model for hypothesis H2 considering the deployment package size.

Table 7.3 shows the statistical evaluation for the deployment package size hypothesis. The correlations in 2018 were weak, but present. This changed in 2023, when we see for the Java use case that there is no correlation and a slightly negative slope. It indicates that the deployment package size in the form of artificially adding a file to the zip which is not used by the function in the Java case does not influence the startup of a cloud function instance. On the other side, the JavaScript cloud functions experienced an overhead when starting new instances. The correlation with a coefficient of 0.74 is strong and the intersect with 359 ms of the linear model is similar to the mean overhead in Table 7.2 with 329 ms for 256 MB. Therefore, we observe a mixed picture for this hypothesis and cannot provide a conclusive answer.

H3: Hypothesis Memory Setting The hypothesis *Memory Setting* states that the cold start overhead decreases with the size of memory. As for the last hypothesis, we calculated the Spearman's correlation coefficient ρ as well as the linear regression model.

Year	Language	ρ	$Linear Model$
2018	Java	-0.59	$1634\,ms - 0.2491\,ms/MB$
2018	JS	-0.20	$606\,ms - 0.0668\,ms/MB$
2023	Java	-0.82	$947\,ms - 0.1185\,ms/MB$
2023	JS	0.16	$319\,ms + 0.0007\,ms/MB$

Table 7.4.: Spearman's correlation coefficient ρ and linear regression model for hypothesis H3 considering the memory setting in 2018 and 2023.

Our hypothesis holds partly true since the values for the correlation coefficient ρ in Table 7.4 were in a mediate correlation range in 2018. The correlation coefficient ρ is negative due to the negative slope but the interpretation is the same as for positive values as discussed above. For Java, we observed a higher correlation and slope. We assume that this is caused by a costlier middleware layer. As Java is a compiled language, the JVM needs to be set up to execute the code. The available CPU and memory influence how fast this can be done. JS is an interpreted language and therefore the execution environment is different as the one for Java, but more resources also had a positive effect on the cold start time in 2018. This effect was not evident in 2023 for JavaScript cloud functions as can be seen from the mean values in Table 7.2. Therefore, the presented data support the hypothesis H3 for Java functions, but not for JavaScript ones.

7.2.6. Discussion

7.2.6.1. Discussion of Results

Our motivation to take the cold starts of cloud functions into consideration was the prevailing strategy in 2018 of using pings to pre-warm cloud function instances. The experimental setup of our benchmark is a REST-based interaction via an API gateway. As noted in the introduction, this *ping hack* is opposing the FaaS principle of scaling to zero. Nowadays, platforms have the option to reserve capacity and already configure a pool of warm cloud functions instances which comes along with higher cost.

Our methodology to assess the cold start from a user point of view is inevitable, because platforms report only a fraction of cold start overhead in their function startup duration. Additionally, they may report different fractions of the provisioning and initialization. Especially for functions written in JS on AWS our results were surprising. We measured that cold starts on the platform were faster than the consecutive warm ones in some cases. This leads to the conclusion that AWS only bills the users for their function executions without the time to set up servers, virtual machines and containers.

The gap between compiled and interpreted languages with a ratio between 2 and 4 was higher than expected. Our explanation is that complex execution environments, like the JVM in case of the compiled language Java, overcharge the already busy CPU at startup. This effect is smaller for higher memory settings but still present. Especially the performance gain for compiled languages is worth mentioning. Cold start overhead of Java functions correlates with $\rho = -0.59$ in 2018 and even -0.82 in 2023. Only the deployment package size hypothesis shows a mixed picture as the correlation is lower and varies between positive and negative values within the same platform.

We measured the mean cold start overhead for different platforms, languages and without artificially increased deployment package sizes. It ranges from 320 ms for JavaScript to 1887 ms for Java with the lowest memory configuration for a clean

deployment package. Based on this investigation, the *ping hack* may not always be necessary. Additionally, scaling also leads to cold starts and the ping hack therefore does not solve the problem at all. The ping hack only ensures that a fixed amount of containers is available but does not pre-provision further containers to anticipate future workloads. Our results, especially the comparison of cold and warm executions on the client side, demonstrate that in some use cases there is no need for this kind of hack. This applies particularly in situations where response times of a few hundreds of milliseconds are reasonable. Because of this wide range of cold start overheads, it is important to assess the impact on specific applications. For applications requiring a fast response or involving user interactions, even small cold start overheads might impose a problem. The three investigated hypotheses already provide a few hints for designing cloud functions and answer *which factors influence the cold start behavior of a function besides the function configuration* (RQ5). Further investigations are needed in this area because cold start is one of the main, inherent issues of FaaS.

We conclude this discussion with two thoughts on the repetition of the experiments in 2023: First of all the hardware might have changed in the five years after the initial experiments. Since our data collection process from 2018 did not include the machine configuration, this is only speculation but suggested by the investors' letter from AWS. The experience within this time period showed that documentation is lacking in experiments - also in our early ones - and hardware heterogeneity influences the execution time and the overall quality of service. When analyzing data without this machine configuration dimension, conclusions might be ambiguous. Secondly, the platforms change their inner implementation over time as can be seen by the standard deviations in Table 7.1 and the distributions visually presented in Figures 7.6 and 7.7. A continuous benchmarking of platforms is beneficial to understand the improvement as well as complain about service degradation as discussed and proposed by Figiela and others [79]. In the current scientific community, there is less reward for repeating experiments. Based on the shown data and insights gained from repeating experiments, new workshop respectively conference formats like the reproducibility track at the European Conference on Information Retrieval[184] could be of interest and motivate researchers to engage in this research area. Such efforts will ultimately increase the quality of experiments and their documentation.

7.2.6.2. Threats to Validity

Based on the characteristics Huppler [103] mentioned in his benchmarking publication, we tried to make the experiments presented here as robust, self-explanatory, and repeatable as possible. But there are some factors that could threaten the validity of our data:

[184] https://ecir2023.org/calls/reproducibility.html?v=3.8

- **Platform Documentation Limitations** - There is only limited information available on how containers are initialized and cloud functions are executed. With the documentation information only, the high variety of different execution times of a cloud function is not fully explainable. Also, additional services like the API gateway on AWS can influence the results.
- **Available Metrics** - The function execution time that is logged and used for billing on the platforms provides only limited information. In AWS, the start up duration of a container is partly included in the logged execution time. This initialization of a container is crucial for the perceived cold starts.
- **Sample Size** - We tested our hypotheses with only 275 cold-warm pairs per function. More data points would better substantiate the findings.
- **Temporal Relevance** - Due to the young and evolving FaaS paradigm, the updates and changes in the platforms limit the relevance of our results to a certain time frame. The repetition of the experiment in 2023 confirmed this threat. The currently up to date data from these experiments will again be outdated by the next major release within the cloud provider's infrastructure.

7.2.7. Future Work

We plan to do the same benchmark setting again for the tested hypotheses and want to integrate additional FaaS platforms as listed in Tables 3.4 and 3.5. The next benchmark will be executed for a longer time period to assess daily differences in the execution time and cold start behavior. Testing further hypotheses, especially the number of dependencies, which is important during the implementation of cloud functions, is scheduled for future work.

Such a follow-up benchmark could serve as a data basis for a concurrency benchmark, which will be executed at the same time to get comparable data points. The concurrency tests are quite important since one of the main use cases is the usage of cloud functions as a reactive component to decouple peak loads in a web application scenario. Peak loads are the main cause for triggering a huge amount of cold starts on the platform.

To understand the different FaaS use cases, further cloud function triggers need to be investigated in respect to their cold start impact. Especially the event triggers of databases are widely used, where, for example, a cloud function is triggered for every inserted entry in a database.

Part IV.

Outlook and Conclusion

8. Simulating Microservices Architecture - an Outlook

So far, the experiments in this work focused on single functions in isolation. Each of these functions was CPU-intensive and invoked via an API gateway. This design was chosen on purpose to ensure clean setups and repeatable experiments in order to draw strong conclusions on the methodology and results. Since this is a simplification of the problem domain and only a small and limited fraction of FaaS use cases, the motivation for this outlook is to use the proposed simulation approach to test more than one function and build a more common use case for FaaS.

Therefore, we present a simulation of a typical FaaS microservices architecture here. In Section 8.1, we describe the use case and corresponding functions and present early results in Section 8.2 to discuss them in Section 8.3. Future work on simulating a microservices architecture with our simulation approach concludes this outlook.

8.1. An Exemplary Use Case

We know from the conceptualization in Section 3 that FaaS is an event driven computing model. As already remarked, cloud functions are integrated with several other backend services and tied to the ecosystem they are running in. Winzinger and Wirtz [288] checked several FaaS projects at GitHub where most of the applications which use another service on the same FaaS platform use a storage solution like DynamoDB.

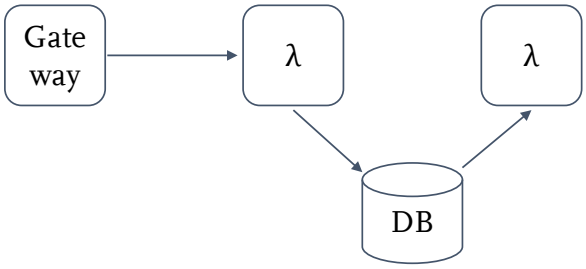


Figure 8.1.: Architecture of a typical microservices FaaS application.

Figure 8.1 shows a small microservices architecture followed by the most common integration scenario based on the mentioned investigation. The first function is triggered by an API Gateway over HTTP. It computes a random number, simu-

lates a fixed blocking IO call by using the programming language's sleep method and stores an entry into a database with a URL of a picture. The second function is triggered by an event which is spun up when new items are inserted in the database. It generates a PDF based on the picture URL from the stored entry and temporarily stores this PDF in a folder.

The chosen provider for the experiment is again AWS with the three services API Gateway, Lambda and DynamoDB which are necessary to construct the application within the AWS cloud in the region eu-central-1. The triggers at AWS work as explained in the more abstract description above. The API Gateway routes the REST call to the corresponding FaaS middleware where a new function is started or an existing instance is used for executing the request. For connecting to the database, the corresponding provider SDK was used. The function then stores the data in a DynamoDB table. The *GeneratePDF* function is then triggered based on the DynamoDB event. It generates a PDF with an image from a free image repository like pixabay[185]. For the local simulation of the microservices architecture, our research prototype `SeMoDe` needed an extension to allow the execution of several functions in sequence. The gateway invocation as well as the DynamoDB event are mocked within the main method of the corresponding function to guarantee proper input values for execution. So locally, there is no trigger executing the second function. To comply with the dev-prod parity principle raised in this work, we used an official image for DynamoDB[186] to start the database locally. Since we assume that Lambda functions and DynamoDB instances are not collocated on the same physical machine within the AWS cloud, we decided to also deploy the local database container on another machine within our university cluster.

The experiments on AWS Lambda were executed on 06/13/2023. Five memory settings were used for each function. To isolate DynamoDB triggers, we furthermore created five tables. Since the executions locally were done in sequence, there was only a single DynamoDB instance.

8.2. Early Results

In the following, early results are presented for local simulations as well as benchmark data collected on AWS. A statistical evaluation is not presented here due to the intended use of the simulation framework. A developer should be able to select a proper function configuration based on the graphical representation of the simulation framework as discussed in Section 7.1. For high resolution of the figures and a better resizing, they were generated with R, but the graphical representation provided in the web UI of our prototype is the same.

Figure 8.2 shows the simulation data for both functions. We executed the functions on H90 where the one CPU equivalent is comparable to 2505 MB which

[185]https://pixabay.com/
[186]https://hub.docker.com/r/amazon/dynamodb-local/

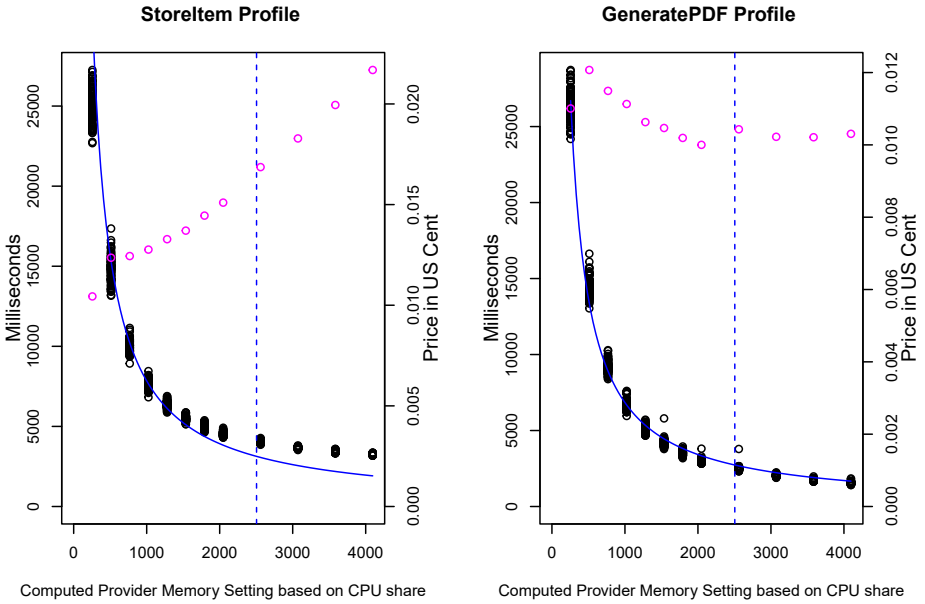


Figure 8.2.: Simulation results for two cloud functions within a microservices architecture executed on H90. Black dots correspond to the primary y-axis and show the simulated execution time in milliseconds. Magenta indicates the price per average function invocation in US cent.

is represented by the vertical dotted blue line. For the ideal cost-performance curve in blue see Equation 6.4. The memory setting for computing this curve is 1024 MB. For the first function, *StoreItem*, which computes a random number, simulates an IO call to a third party service and stores the random number in the database, the black dots indicate lower response times when increasing resources. So for latency related use cases, developers of this cloud function could consider a higher memory setting to satisfy latency requirements. If they do so, they have to be aware that they will spend more money for the same functionality as the rising price, the magenta dots, indicate. Assessing the predicted prices after the one CPU equivalent, it is obvious that the function does not profit from a multi-threaded execution environment. This can be seen from the slopes of the two intervals from 1024-2048 MB and 3072-4096 MB. The simulation for the second function, *GeneratePDF*, is different. The primary y-axis again shows the execution time and the secondary y-axis the simulated price per request. It can be seen that the execution time drops when resources are increased and that this is also the case for multi-core environments. The price lies in a narrow interval between 0.010-0.012 US Cents per invocation. Based on these simulations, a developer could use any memory setting they like to deploy the function but should decide on a higher memory setting to achieve a better performance.

To discuss the soundness of the simulation results, we deployed the same application on AWS. The diagrams in Figure 8.3 contain the execution data for both functions. We used the benchmark facility of `SeMoDe` again to deploy the functions and trigger the first function *StoreItem* via HTTP. The benchmark mode was

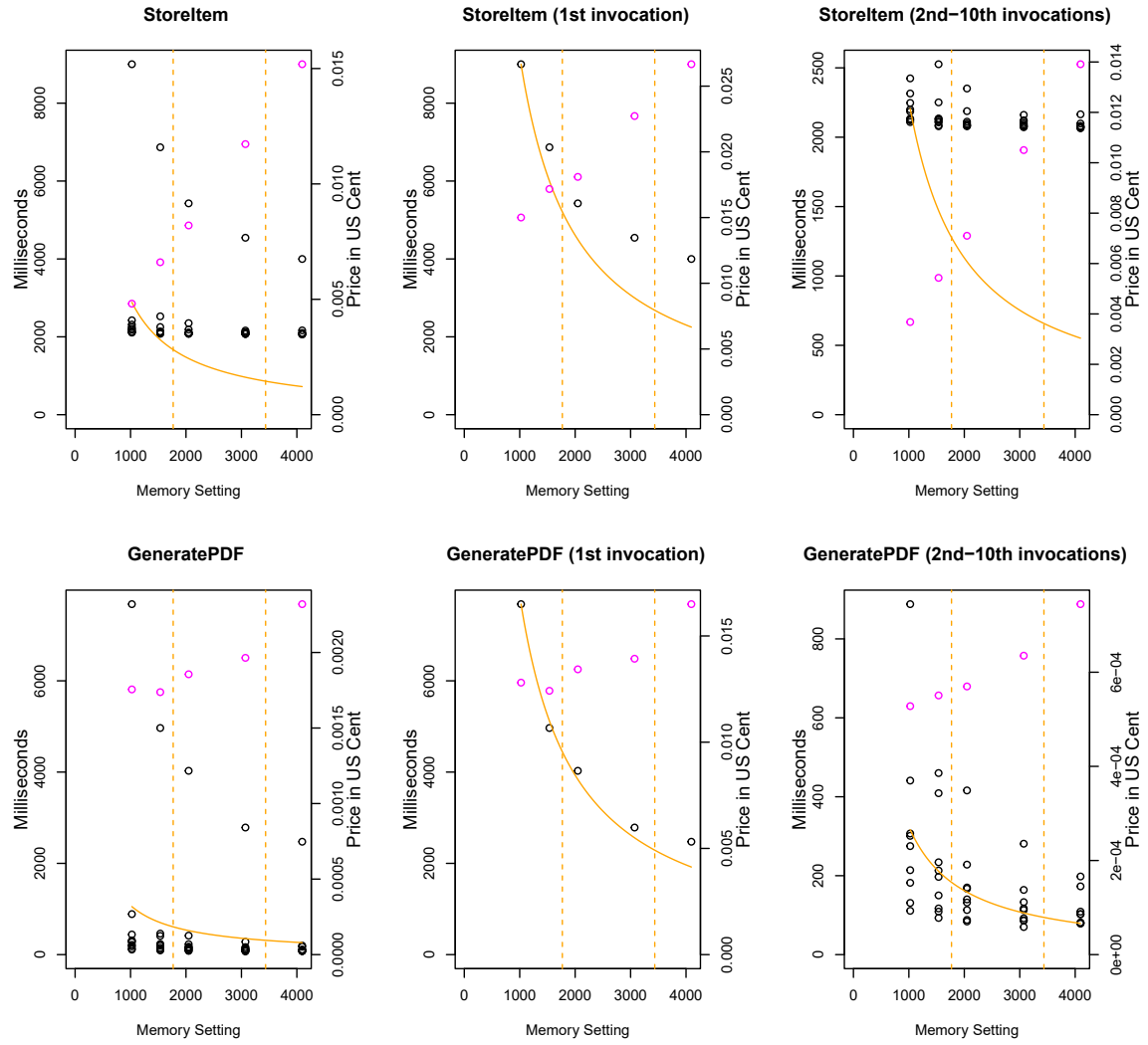


Figure 8.3.: Execution Results for two cloud functions within a microservices architecture at AWS Lambda. Black dots correspond to the primary y-axis and show the execution time on AWS Lambda in milliseconds. Magenta indicates the price per average function invocation in US cent.

set to `sequentialInterval` with two minutes between the requests. Every deployed function was executed ten times. The matrix of diagrams in Figure 8.3 is as follows: The first diagram in every row shows all ten requests of the corresponding function whereas the other two diagrams are a subsets of the first. When we analyzed the data, it was evident that the first invocation had a different execution behavior than the others. Therefore, only the first invocation is shown in the middle column. The reason for this different execution behavior is that this first invocation faced a cold start and all other invocations for every setting for both functions reused an already existing cloud function instance on AWS Lambda. The last diagram in every row shows these warm executions. For the *StoreItem* function, the cold start execution behavior is comparable to our simulation. The execution time drops but the function does not profit from resource increases nor multi-core environments in the same way as indicated by the price trend. When comparing

the first invocation of the second function, similar trends as in our simulation are present. Due to the single data point for every memory setting, these early results need additional confirmation but it seems that the function would profit from a resource increase also beyond the first CPU equivalent.

For the warm executions, the first conclusion of the simulation still holds: The first function would not profit from a resource increase. However, the execution data distribution is different. The trends show a dominant impact of the blocking IO call which was simulated with a sleep interval of two seconds. Loading dependencies, establishing the connection to the database, as well as some of the JIT compilation efforts were already performed during the first invocation. For the second function the picture is similar. The function does profit from a resource increase but shows a moderate price increase for higher settings which is not indicated by the simulation. Here a more detailed investigation of the JIT compilation might be beneficial in order to understand the effects on warm instances in this specific case.

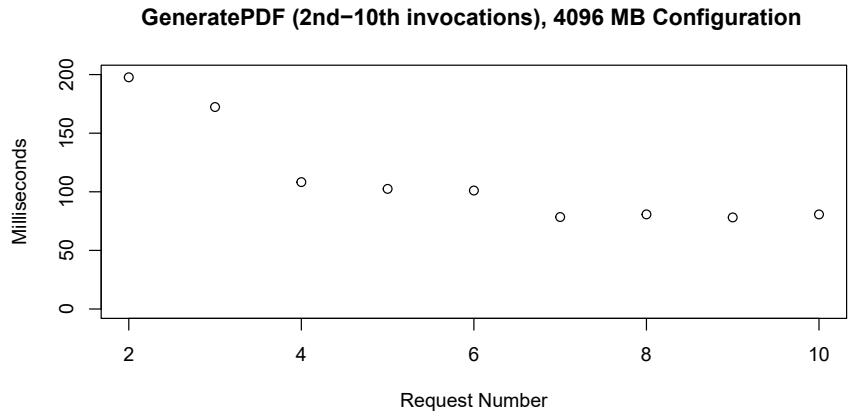


Figure 8.4.: Drill down of the GeneratePDF function from Figure 8.3 for the memory configuration 4096 MB for warm executions.

Figure 8.4 shows the execution data of Figure 8.3 for the warm executions of the second function with 4096 MB memory setting. The x-axis shows the request number in chronological order. It can be seen that later invocations are faster than previous ones until the 7^{th} request. We assume that all hot spots are compiled and the JIT efforts converges. For other memory settings, the distribution over time is similar. Such optimizations, like for the Java cloud function in this example, can also reduce the execution time and cost when the cloud function is called often and the platform reuses instances.

8.3. Discussion

The idea of comparable execution environments to achieve dev-prod parity was already raised a few times in this work. So far the focus has been to make the

cloud function runtime and local deployment setups comparable and focus on a single function in isolation. Since FaaS as an event driven computing model is dependent on a plethora of other services like gateways, databases, notification systems etc., a test and in our case a simulation environment is necessary to be able to perform local integration tests and apply our proposed simulation approach. To achieve dev-prod parity for the complete microservice application, a local equivalent of every third party service is desirable. Since AWS for example has more than 200 services within its ecosystem[187], there is no local equivalent nor OCI compliant image available for every service. If there is no local option, the alternative would be to use configured test instances at the cloud provider and access them via standard protocols. However, the simulation results in these cases cloud be distorted due to the round trip time when calls to database instances are performed. In our case, we compared a local deployment within our university cluster, where both functions and the database instance are running, to the AWS deployment in a single region which is the fairest setup we can get.

A question that often arose in discussions of the simulation approach at conferences and workshops was whether the sole focus on a single function in isolation is representative for FaaS. From a use case perspective, this is not the case. However, the data we presented in previous sections and especially in this brief outlook show that executions of functions are independent of each other. There are two reasons for this claim: The first aspect is the typical FaaS worker architecture explained in Section 3.5 where every function instance is wrapped in a micro VM sandbox. In the experiments in Section 7.2, there is no co-location of several functions or function instances on a single VM. The same is true for *StoreItem* functions which are deployed on unique VMs. Secondly, there is also no evidence in the execution data that the second function is influenced by the triggered events from Dynamo DB. This justifies the focus on a single function in prior sections of this work and also the sequential execution of chained functions in our simulation approach. A limitation of this outlook is our workload which triggers only a single *StoreItem* function at a time which writes to the database. There is no notion of parallel events generated by the database. If we would change the workload to have several events at the same time triggering a function, databases might become a bottleneck.

Another limitation of the current simulation framework is the focus on cold starts only where the prediction of the function behavior is comprehensible but warm executions are currently neglected. For the warm function invocations in the included example, also the programming language and its specialties, in our case Java with JIT compilation has to be assessed in detail. Due to different JIT compiler and garbage collection algorithms, the results may differ.

The last aspect to discuss is the expected workload of the function which determines the number of cold respectively warm starts. As elaborated in detail in Section 7.2, different aspects like the programming language and the memory set-

[187] https://aws.amazon.com/what-is-aws

ting influence the cold start time in general. When combining the insights from this outlook with the cold start investigation, cloud functions with a predictable, constant invocation pattern could benefit from execution environments which optimize the executed code as in the Java case with JIT compilation. For bursty workloads, another choice of programming language and a different resource configuration might be more appropriate.

8.4. Future Work

Further research is necessary in three aspects in order to improve the dev-prod parity aspect for microservices architectures and the applicability of the proposed simulation approach. Firstly, a study on test dummies respectively emulators for third party services would support developers to choose proper local tools for integration testing. A feature comparison of these local tools should be included to provide an understanding about the dev-prod parity property.

In its current form, the research prototype `SeMoDe` only simulates cold starts. As seen in this outlook, the graphical representations give developers guidance and are comparable with the execution data on AWS Lambda but the warm executions are still missing. As remarked in the discussion, in some cases the invocation workload of the function might be known and fine-tuning based on the programming language used and the memory setting can lead to further performance and cost improvements.

Finally and tightly related to the shown JIT influence on the execution behavior, another potential improvement to be implemented in future work would be an integration of profiling support. Tools like FlameGraphs[188] provide an understanding of the resource consumption of a single invocation or a sequence of function calls. These tools can also uncover unknown performance vulnerabilities within the function implementation. It is obviously worth to look at code paths which consume a lot of resources before refactoring the code and reduce the number of lines of code. Since FaaS platforms implement a fine-granular pay-per-use model, every optimization reduces cost.

[188] https://github.com/brendangregg/FlameGraph

9. Conclusion

This section concludes the thesis and discusses the most important competing approaches in Section 9.1. Only those are listed which directly influenced the design decisions made for experiments and the practical realization of the proposed simulation framework. Section 9.2 summarizes the main contributions of this thesis. A vision is drawn for future work on how to make benchmark and simulation approaches more comparable and reproducible.

9.1. Competing Approaches

The overall aim of the work in hand is to propose a simulation framework for cloud functions which provides a prediction of the execution behavior on a cloud platform during the development process of the cloud function. Based on a newly introduced mechanism to achieve dev-prod parity, the simulation framework is unique in its methodological approach of calibrating environments and making them comparable. To the best of the author's knowledge, there is no competing approach which simulates the execution behavior locally under different resource configurations to predict the execution behavior on a FaaS platform. Nevertheless, there are competing approaches for benchmarking and simulations to unravel FaaS platform mysteries. This section only includes the most relevant competing approaches for benchmarking and simulation to stress similarities but also limitations and differences. A more thorough discussion of related work was presented in the SLRs and the corresponding related work sections in every chapter. Furthermore, early ideas from related work on dev-prod parity are also included here which are the foundations of the calibration step in this work.

There are a few important approaches found in the papers discussed in the SLR for benchmarking in Table 4.3 which inspired our work. The identification and documentation of executing hardware, like in Giménez-Alventosa and others [86], is important for the data evaluation to prevent conclusions being influenced by different hardware. Their data evaluation is particularly transparent since each hardware configuration is depicted in a different color. Nevertheless, for some of the evaluations in their paper, the link to the used hardware is also missing which makes the interpretation fragile. Another important aspect for understanding the deployment of cloud functions on public cloud providers is the VM identification [282]. Wang and others present a solution for all major public providers in their work but missed to also gather the hardware information. Inspired by these approaches, we overcame their problems of incomplete raw data by stating

a checklist in Section 4.2 which includes hardware and VM information. Our research prototype SeMoDe is implemented to parse and collect the platform data accordingly.

From a use case and applied function perspective, the research papers from AGH University of Science and Technology are most related to our benchmarking efforts [79, 167, 211] and inspired the design decisions of SeMoDe. They used LINPACK and a CPU intensive function to compare the different offerings of the four public cloud providers AWS Lambda, Azure Functions, Google Cloud Functions and IBM OpenWhisk. Their work was the predominant reason to focus on AWS Lambda due to the consistent scaling of resources. They also inspired the measurement methodology of tying the local request-response to the platform execution log data. However, their benchmarks were executed at a time when AWS Lambda offered only configurations with a single core. As already remarked, resource improvements exceeding a single CPU are often misinterpreted as can be seen in many publications [17, 48, 70, 72, 154, 189]. But there are also publications besides our research papers [180, 185] which discuss multi-threaded functions for public cloud providers [293] and another publication for open-source platforms [300]. Yu and others [293] omitted the cloud function configuration and concluded that in-function parallelization aka multi-threading leads to an increase in execution time compared to the single-threaded variant per se. We know from research included in this work, that this is only the case for configurations less than a single full CPU due to scheduling efforts when coordinating various threads. Therefore, this general conclusion is wrong. Zhang and others [300] stated in their abstract that "the impact of resource allocation on function performance in serverless platform[s] is still not clear". When looking at their multi-threaded function experiments this statement is surprising since functions with higher CPU settings profit proportionally from the resource increase. This is always the case except for functions which do not use more than a single CPU. These insights on resource configuration and multi-threading motivated isolated experiments as well as incorporating CPU equivalents in our simulation approach.

Considering FaaS simulation approaches, *Sizeless* [72] and *SAAF* [50] are most closely related to our research. The approach in Sizeless requires an input data set with function segments for different classes of problems deployed to the cloud platform and executed for different memory settings. Given this data set, arbitrary functions can then be analyzed and a prediction can be made if monitoring data is available for a specific memory setting. The generation of this large data set can be compared to the calibration introduced in this work. However, for Sizeless, additional data is needed from the platform to predict the behavior of the function and to select a proper resource configuration. This is a step which can be omitted in our approach.

The Serverless Application Analytics Framework (SAAF) [50] measures machine metrics like CpuSteals, pagefaults etc. to generate a detailed utilization profile for the function and the corresponding executing hardware. Their motivation was

to incorporate hardware heterogeneity and effects of multi-tenancy at public providers to predict the execution behavior on different hardware for the same or another public cloud provider. Their data evaluation is mainly based on a larger set of experiments and linear regressions to compare the execution behavior of one provider and its machine metrics to another. This allows predictions for similar workloads on different hardware in the cloud which is a dimension we do not consider in our simulation approach since a FaaS platform user has no influence on which underlying machine a function is allocated. Nevertheless, their research on analyzing the function and generating an utilization profile could complement our efforts. Local simulations in the sense of understanding the execution behavior of arbitrary functions was not part of their work.

In 2017, Arif and others already stated that a single, scalar factor is not sufficient to make different environments comparable to each other [4]. In their research, they compared virtualized and bare-metal environments and gained several findings for CPU and IO related metrics, concluding that they are not easily comparable to each other. To achieve dev-prod parity in FaaS, a different approach is necessary. The work on a combined benchmark and simulation approach [111] inspired some aspects of this work. Their use case was on finding proper VM configurations by extending CloudSim [36]. Their idea was to make experiments on a public provider and map the results to another provider based on monitoring data. This can be seen as a first step to make different provider offerings comparable to each other for the data points included in their benchmark. It is already an improvement to the scalar factor idea discussed in the previous paper. Their defined mapping rules then result in an approximation solution for the simulations. But it is only an approximation of the target setting since they only use monitoring data. The restrictions on some target settings motivated us to collect calibration data and compute linear regression models to calculate resource configurations for arbitrary settings. Nevertheless, Johng and others [111] are still capable of running simulations for different influential settings to find a proper VM configuration based on their requirements. This approach is the most closely related one from a conceptual point of view, but deals with applications, not solely functions, and does still not provide proper configurations for arbitrary simulation settings.

9.2. Summary

The characteristics of FaaS are unique in two aspects compared to other *as a Service* models. First of all, FaaS platforms offer a truly pay-as-you-go service. Only the execution time of a cloud function instance in millisecond granularity is used for billing a user. The other distinctive aspect is scaling on demand and especially to zero. Jonas and others argue that FaaS platforms are one way to "Occupy the Cloud: Distributed Computing for the 99%" [112], as they state in their paper's title. It sounds easy and promising, to only deploy source code to a FaaS platform with the provider caring for the complete operations work, but there are two influ-

ential factors which still make this task challenging. Developers need to choose a runtime like *java17* or *nodejs18.x* to run their functions. When the same functionality is implemented with a different language, the execution duration and resource consumption will vary significantly [216]. The second hurdle is the allocation of resources. In order to find the best configuration for a given set of requirements, developers often deploy their functions with different resource settings and analyze the data afterwards. This process is expensive and lengthy and consumes additional time for the analysis and the subsequent settings update.

Furthermore, an often neglected aspect when implementing cloud functions are multi-threaded functions. As could be seen in the course of this work, it is however important to consider multi-threaded functions and their impact on performance when allocating multi-core environments to the cloud function instance. Putting all this together, the paper title of Jonas and others sounds promising to easily implement and deploy auto-scalable functions but neglects a resource-aware allocation of resources. Therefore, this thesis provides methods and tools to run a cloud function in various resource settings on a developer's machine to simulate the expected behavior at FaaS platforms. To reach this aim, we worked on four main contributions. These are summarized briefly in the following:

The first contribution is a conceptualization of FaaS and a disambiguation to Serverless. When researchers, speakers, and practitioners referred to FaaS in early publications, they often used Serverless as a synonym to describe event-driven, stateless cloud functions which scale on demand and offer a fine-grained metering of resources. This terminology usage has been adopted into language and led to a self-reinforcing effect. To overcome this situation and provide proper definitions, we made a pull request on the CNCF glossary for both terms[53,54] which is pending as of the time of writing[189]. We hope that this contribution raises awareness within the scientific community to use proper and precise terms. As a short summary we agree with the majority of publications that FaaS is one form of Serverless in a sense that the provider manages all operational tasks to provide the specific service.

Building upon a proper terminology, the next contribution is a structured investigation of performance research in the FaaS domain. The goal of this SLR was to understand related work in the benchmarking area and distill knowledge for our work. It was evident after the SLR that the scientific community lacks guidelines for documenting experiments which would allow a proper interpretation of results. This issue is confirmed by secondary studies which showed that only three out of 26 experiments are reproducible [137], a majority of 122 papers is not reproducible [114] or only 26% of 315 data projects published raw data [51]. Also other disciplines like digital medicine face this *reproducibility crisis* [261]. To overcome this issue for the FaaS research domain, we formulated a checklist for conducting FaaS benchmarks with a focus on documentation which is enforced when using our research prototype `SeMoDe`. A current movement is trying to

[189] https://github.com/cncf/glossary/pull/2217

provide better access to documentation and research data by promoting the use of ACM badges[190] or encouraging authors to use platforms like Zenodo. Nevertheless, there is still too little reward in documenting experiments in detail and uploading raw data. Another interesting finding during this SLR on FaaS benchmarking was the assignment of multi-core environments to cloud functions and the conclusions drawn in the papers. A lot of authors described an effect of constant execution time for higher resource settings without clarification why this happens [17, 48, 70, 72, 154, 189]. They missed the fact that higher configurations exceed a single core. Only cloud functions implemented in a multi-threaded way profit from a resource increase beyond a single core. An attempted explanation for this blind spot in research may be the engineering and runtime characteristics of cloud functions. As mentioned before, Wolff introduced the term nanoservice in his book on *Microservices* [290] to distinguish a single-scoped cloud function from a self-contained microservice. This already implies that there are various methods running in parallel within a microservice. Assigning more computing resources to a microservice, also beyond a single core, might be beneficial since these different methods within the microservice are scheduled on the available cores and can utilize increasing resources. The utilization profile from a microservice therefore contains some noise which makes a drill-down to specific methods challenging.

Our third contribution is the methodological part of our simulation framework. A prerequisite for our simulation experiments is to make different execution environments comparable. This is in line with one of the Twelve-Factor app principles i.e. dev-prod parity[4]. Based on an abstract computing factor, in our case GFLOPS based on LINPACK, we suggested a new approach to artificially utilize a specific portion of resources to understand the scaling of resources of the physical machine. In the course of this calibration work, we overcome situations where custom CPU frequency scaling algorithms, like `intel_pstate`, distort results. The graphical representation of our research prototype supports developers to configure their systems appropriately even if future hardware improvements produce different kinds of cores on a single chip. Our proposed methodology produces a regression models to compute arbitrary resource settings by equalizing regression models of two calibrated environments. We showed in experiments that this approach can predict the execution time of similar functions on a public cloud provider's platform when some execution data from the target FaaS platform is available. Our research in this regard contributes to a resource-aware allocation of resources for public cloud provider platforms. Open source tools on the other hand are often compared feature-wise but without performance awareness. Since K8s is often used as a higher level of abstraction, we showed in another investigation [180] that our methodology also works for K8s pods when resources are restricted based on K8s limits.

The last contribution is of practical nature. `SeMoDe` assists users to make sensible decisions about their cloud function configuration. Emphasis is put on the

[190] https://www.acm.org/publications/artifacts

different CPU equivalents of the target platform and the local machine to overcome the aforementioned misinterpretation of single respectively multi-threaded functions. Furthermore, our trend curves add the cost dimension to the simulated execution data and allow developers to understand the tradeoff between cost and performance, i.e. execution time, when running the function locally. Furthermore, the research prototype is a tool to conduct scientific experiments and test hypotheses like the ones included in this work which target cold start influencing factors.

Besides future work ideas, which have already been raised in the corresponding sections of the main part of this work, we want to propose another more generic vision for the future of performance aware computing. Abstract computing measures like GFLOPS in our case and ACU for Microsoft Azure[134] can be seen as Key Performance Indicators (KPIs). They could support the comparability of performance research and enable researchers to distill knowledge out of several publications based on a common denominator. To provide a holistic, abstracted view of an execution environment, also other factors like network and memory should be considered in future research. Such an abstracted view on computing resources based on KPIs could support efforts in green computing to make the consumption of energy comparable as well which would ultimately allow the computation of CO_2 budgets for workloads.

To sum up, the thesis suggests a simulation framework for cloud functions. Based on a newly introduced methodology to make different execution environments comparable, we are able to predict the runtime characteristics on a target FaaS platform based on simulations with different resource assignments during the development process. This process frees developers from the burden to deploy their functions first, run it in production with best-effort configurations, analyze the execution data and tweak the function and resource settings afterwards just to start the same process again. We think that this methodology is also applicable to other services and applications to generate profiles under different resource configurations with the caveat of having more noise in the execution data. An example for such a situation is the already mentioned microservice case where several methods are executed in parallel. Overall, the introduction of abstract computing measures, the compliance to documentation guidelines, and reproducibility efforts in general make experiments and publications more comparable and robust to gain information out of data.

Bibliography

[1] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020. (Cited on pages xi, 18, 19, 20, 21, and 151.)

[2] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring serverless functions using statistical learning," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, 2020. (Cited on pages 76 and 121.)

[3] L. F. Albuquerque Jr, F. S. Ferraz, R. F. A. P. Oliveira, and S. M. L. Galdino, "Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS," in *Proceeding of the International Conference on Software Engineering Advances (ICSEA)*, 2017. (Cited on page 4.)

[4] M. M. Arif, W. Shang, and E. Shihab, "Empirical study on the discrepancy between performance testing results from virtual and physical environments," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1490–1518, 2017. (Cited on pages 24, 25, 64, 124, 128, and 187.)

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. (Cited on page 4.)

[6] T. Back and V. Andrikopoulos, "Using a Microbenchmark to Compare Function as a Service Solutions," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2018, pp. 146–160. (Cited on pages 74, 76, 77, and 123.)

[7] O. Balci, "Guidelines for successful simulation studies," in *Proceedings of the Winter Simulation Conference (WSC)*, 1990. (Cited on pages 36 and 37.)

[8] ——, "Quality assessment, verification, and validation of modeling and simulation applications," in *Proceedings of the Winter Simulation Conference (WSC)*, 2004. (Cited on page 36.)

[9] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20. (Cited on pages 41, 50, 51, 52, and 54.)

[10] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: function composition

for serverless computing," in *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! (ACM SIGPLAN)*, 2017. (Cited on pages 77 and 160.)

[11] D. Balla, M. Maliosz, and C. Simon, "Open source FaaS performance aspects," in *Proceedings of the International Conference on Telecommunications and Signal Processing (TSP)*, 2020. (Cited on pages 64, 66, 138, and 140.)

[12] D. Balla, M. Maliosz, C. Simon, and D. Gehberger, "Tuning runtimes in open source FaaS," in *Internet of Vehicles. Technologies and Services Toward Smart Cities*. Springer International Publishing, 2020, pp. 250–266. (Cited on pages 11 and 66.)

[13] D. Balla, M. Maliosz, and C. Simon, "Estimating function completion time distribution in open source FaaS," in *Proceedings of the International Conference on Cloud Networking (CloudNet)*, 2021. (Cited on page 66.)

[14] L. A. Barba, "Praxis of reproducible computational science," *Computing in Science & Engineering*, vol. 21, no. 1, pp. 73–78, 2019. (Cited on page 80.)

[15] ——, "12 ways to fool the masses with irreproducible results," 2021. (Cited on page 30.)

[16] E. Barbierato, M. Gribaudo, M. Iacono, and A. Jakóbik, "Exploiting CloudSim in a multiformalism modeling approach for cloud based systems," *Simulation Modelling Practice and Theory*, vol. 93, pp. 133–147, 2019. (Cited on page 120.)

[17] D. Barcelona-Pons and P. García-López, "Benchmarking parallelism in faas platforms," *Future Generation Computer Systems*, vol. 124, pp. 268–284, 2021. (Cited on pages 5, 9, 13, 60, 61, 76, 78, 81, 147, 186, and 189.)

[18] D. Bardsley, L. Ryan, and J. Howard, "Serverless Performance and Optimization Strategies," in *Proceedings of the International Conference on Smart Cloud (SmartCloud)*, 2018. (Cited on pages 42, 72, and 74.)

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003. (Cited on page 19.)

[20] M. Becker and S. Chakraborty, "Measuring software performance on linux," *arXiv e-Prints*, vol. 1811.01412, 2018. (Cited on page 106.)

[21] J. Beckett, "Bios performance and power tuning guidelines for dell poweredge 12th generation servers," DELL, Tech. Rep., 2012. (Cited on page 103.)

[22] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2005. (Cited on page 19.)

[23] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking*. Springer International Publishing, 2017. (Cited on pages 27, 28, 29, 30, 33, and 103.)

[24] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using Application Knowledge to Reduce Cold Starts in FaaS Services," in *Proceedings of the*

ACM/SIGAPP Symposium on Applied Computing (SAC), 2020. (Cited on page 163.)

[25] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. (Cited on pages xi, 22, 23, and 45.)

[26] J. Bloch, *Effective Java*. Addison Wesley, 2018. (Cited on page 9.)

[27] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 1976. (Cited on page 36.)

[28] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Industry practices and challenges for the evolvability assurance of microservices," *Empirical Software Engineering*, vol. 26, no. 5, 2021. (Cited on page 54.)

[29] D. Bortolini and R. R. Obelheiro, "Investigating performance and cost in function-as-a-service platforms," in *Proceedings of Advances on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2019. (Cited on pages 76, 77, 79, and 81.)

[30] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, 2007. (Cited on page 49.)

[31] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2015. (Cited on pages 45 and 64.)

[32] E. Bugnion, J. Nieh, D. Tsafrir, and M. Martonosi, *Hardware and Software Support for Virtualization*. Morgan & Claypool Publishers, 2017. (Cited on pages xi, 18, and 19.)

[33] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016. (Cited on page 24.)

[34] R. Buyya and M. Murshed, "GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1175–1220, 2002. (Cited on page 120.)

[35] S. Böhm and G. Wirtz, "Profiling lightweight container platforms:microk8s and k3s in comparison to kubernetes," in *Proceedings of the Central European Workshop on Services and their Composition (ZEUS)*, 2021. (Cited on pages 141 and 142.)

[36] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2010. (Cited on pages 73, 120, and 187.)

[37] E. Calore, A. Gabbana, S. Schifano, and R. Tripiccione, "Software and DVFS tuning for performance and energy-efficiency on intel KNL processors," *Journal of Low Power Electronics and Applications*, vol. 8, no. 2, p. 18, 2018. (Cited on page 106.)

[38] L. Carvalho and A. Araujo, "Orama: A benchmark framework for function-as-a-service," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*. SCITEPRESS - Science and Technology Publications, 2022. (Cited on pages 72 and 74.)

[39] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, and L. Zhu, "RADON: rational decomposition and orchestration for serverless computing," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1-2, pp. 77–87, 2019. (Cited on page 71.)

[40] E. Casalicchio and V. Perciballi, "Measuring docker performance," in *Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE)*, 2017. (Cited on pages 119 and 120.)

[41] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless Programming (Function as a Service)," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2017. (Cited on page 4.)

[42] ——, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019. (Cited on pages 41, 50, 51, 52, and 53.)

[43] M. Cha, H. Haddadi, F. Benevenuto, and K. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *Proceedings of the International AAAI Conference on Web and Social Media (ICWSM)*, 2010. (Cited on page 28.)

[44] A. Chan, K.-T. A. Wang, and V. Kumar, "Balloonjvm: Dynamically resizable heap for faas," in *Proceedings of the International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING)*, 2019. (Cited on page 57.)

[45] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, "Smart CloudBench—a framework for evaluating cloud infrastructure performance," *Information Systems Frontiers*, vol. 18, no. 3, pp. 413–428, 2015. (Cited on page 140.)

[46] H. Choi and H. Varian, "Predicting the present with google trends," *Economic record*, vol. 88, pp. 2–9, 2012. (Cited on page 45.)

[47] V. Choudhary and J. Vithayathil, "The impact of cloud computing: Should the IT department be organized as a cost center or a profit center?" *Journal of Management Information Systems*, vol. 30, no. 2, pp. 67–100, 2013. (Cited on page 137.)

[48] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the International Middleware Conference (MIDDLEWARE)*, 2021. (Cited on pages 9, 60, 75, 76, 77, 186, and 189.)

[49] J. Corbet, "Seccomp and sandboxing," Linux Weekly News, 2009, https://lwn.net/Articles/332974/. (Cited on page 23.)

[50] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting performance and cost of serverless computing functions with SAAF," in *Proceedings of the 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 (DASC/PiCom/CBDCom/CyberSciTech)*, 2020. (Cited on pages 27, 77, 95, 103, 122, 129, 144, 152, 164, and 186.)

[51] J. L. Couture, R. E. Blake, G. McDonald, and C. L. Ward, "A funder-imposed data publication requirement seldom inspired data sharing," *PLOS ONE*, vol. 13, no. 7, p. e0199789, 2018. (Cited on pages 31 and 188.)

[52] A. Cuomo, M. Rak, and U. Villano, "Simulation-based performance evaluation of cloud applications," in *Intelligent Distributed Computing VI*. Springer Berlin Heidelberg, 2013, pp. 263–269. (Cited on pages 34, 119, and 120.)

[53] M. Curiel and A. Pont, "Workload Generators for Web-Based Systems: Characteristics, Current Status, and Challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1526–1546, 2018. (Cited on page 29.)

[54] A. Dakkak, C. Li, S. G. de Gonzalo, J. Xiong, and W. mei Hwu, "TrIMS: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2019. (Cited on page 9.)

[55] G. D'Angelo and R. D. Grande, "Guest editors' introduction: Special issue on simulation in (and of) the cloud," *Simulation Modelling Practice and Theory*, vol. 58, pp. 113–114, 2015. (Cited on page 120.)

[56] A. Das, S. Patterson, and M. Wittie, "EdgeBench: Benchmarking Edge Computing Platforms," in *Proceedings of the Workshop on Serverless Computing (WoSC)*, 2018. (Cited on pages 72 and 73.)

[57] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2020. (Cited on page 138.)

[58] R. Debab and W. K. Hidouci, "Containers runtimes war: A comparative study," in *Proceedings of the Future Technologies Conference (FTC)*, 2021. (Cited on pages 24 and 25.)

[59] C. Denninnart and M. A. Salehi, "Efficiency in the serverless cloud computing paradigm: A survey study," *arXiv e-Prints*, vol. 2110.06508, 2021. (Cited on pages 51, 52, and 54.)

[60] M. A. der Landwehr, M. Trott, and C. von Viebahn, "Computer simulation as evaluation tool of information systems: Identifying quality factors of simulation modeling," in *Proceedings of the Conference on Business Informatics (CBI)*, 2020. (Cited on pages 36 and 37.)

[61] C. Dhule and U. Shrawankar, "Impact analysis of hypervisors on the performance of virtualized resources," in *Proceedings of Integrated Intelligence Enable Networks and Computing (IIENC)*, 2021. (Cited on page 20.)

[62] G. Didier and C. Maurice, "Calibration done right: Noiseless flush + flush attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2021, pp. 278–298. (Cited on page 106.)

[63] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, "Understanding modern storage APIs," in *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2022. (Cited on page 106.)

[64] J. Domaschka, M. Leznik, D. Seybold, S. Eismann, J. Grohmann, and S. Kounev, "Buzzy: Towards realistic DBMS benchmarking via tailored, representative, synthetic workloads," in *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2021. (Cited on page 28.)

[65] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979. (Cited on pages 12, 107, 109, and 125.)

[66] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. (Cited on pages 12, 107, and 125.)

[67] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, "Automatically exploring tradeoffs between software output fidelity and energy costs," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 219–236, 2019. (Cited on page 106.)

[68] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pp. 195–216. (Cited on page 44.)

[69] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a service (XaaS) on the cloud: Origins, current and future trends," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2015. (Cited on page 4.)

[70] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2020. (Cited on pages 9, 60, 186, and 189.)

[71] J. Eickhoff, J. Donkervliet, and A. Iosup, "Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games extended technical report," *arXiv e-Prints*, vol. 2112.06963, 2021. (Cited on page 139.)

[72] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless," in *Proceedings of International Middleware Conference (MIDDLEWARE)*, 2021. (Cited on pages 9, 60, 76, 121, 122, 131, 159, 186, and 189.)

[73] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?"

IEEE Software, vol. 38, no. 1, pp. 32–39, 2021. (Cited on pages 42 and 43.)

[74] A. Eivy, "Be Wary of the Economics of "Serverless" Cloud Computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017. (Cited on pages 69 and 117.)

[75] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, "Performance evaluation of container runtimes," in *Proceedings of International Conference on Cloud Computing and Services Science (CLOSER)*, 2020. (Cited on pages xi, 22, 24, and 25.)

[76] F. Fakhfakh, H. H. Kacem, and A. H. Kacem, "Simulation tools for cloud computing: A survey and comparative study," in *Proceedings of the International Conference on Computer and Information Science (ICIS)*, 2017. (Cited on page 120.)

[77] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014. (Cited on page 28.)

[78] D. Fernández-Cerero, A. Fernández-Montes, A. Jakóbik, J. Kołodziej, and M. Toro, "SCORE: Simulator for cloud optimization of resources and energy consumption," *Simulation Modelling Practice and Theory*, vol. 82, pp. 160–173, 2018. (Cited on page 120.)

[79] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, p. e4792, 2018. (Cited on pages 5, 76, 79, 149, 173, and 186.)

[80] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the Cloud: What It Should, Can, and Cannot Be," in *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, 2013. (Cited on pages 28 and 29.)

[81] B. Full, J. Manner, S. Böhm, and G. Wirtz, "MicroStream vs. JPA: An empirical investigation," in *Service-Oriented Computing*. Springer International Publishing, 2022, pp. 99–118. (Cited on pages 31 and 32.)

[82] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "How low can you go? a limbo dance for low-latency network functions," *Journal of Network and Systems Management*, vol. 31, no. 1, 2022. (Cited on page 106.)

[83] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019. (Cited on pages 72 and 74.)

[84] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. (Cited on pages 51 and 52.)

[85] A. U. Gias, A. van Hoorn, L. Zhu, G. Casale, T. F. Düllmann, and M. Wurster, "Performance engineering for microservices and serverless applications: The RADON approach," in *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020. (Cited on pages 72 and 74.)

[86] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS Lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019. (Cited on pages 76, 77, 129, 144, and 185.)

[87] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, and D. Holmes, *Java Concurrency in Practice*. Pearson Education (US), 2006. (Cited on page 9.)

[88] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974. (Cited on page 18.)

[89] C. Gough, I. Steiner, and W. Saunders, "Operating systems," in *Energy Efficient Servers*. Apress, 2015, pp. 173–207. (Cited on page 105.)

[90] H. Govind and H. GonzaleznVelez, "Benchmarking serverless workloads on kubernetes," in *Proceedings of the International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021. (Cited on pages 72 and 73.)

[91] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," *arXiv e-Prints*, vol. 2102.12770, 2021. (Cited on pages 75, 76, and 77.)

[92] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast intra-kernel isolation and security with IskiOS," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021. (Cited on page 106.)

[93] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2019. (Cited on page 47.)

[94] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, 2008. (Cited on page 20.)

[95] R. Hancock, S. Udayashankar, A. J. Mashtizadeh, and S. Al-Kiswany, "Orcbench: A representative serverless benchmark," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2022. (Cited on pages 72 and 74.)

[96] R. Hartauer, J. Manner, and G. Wirtz, "Cloud function lifecycle considerations for portability in function as a service," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2022. (Cited on pages 7, 8, 33, 54, and 94.)

[97] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, pp. 1–29, 2021. (Cited on pages 41, 51, 52, 54, 55, 56, 57, and 66.)

[98] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2019. (Cited on pages 42, 51, 52, and 54.)

[99] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with openLambda," in *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HOTCLOUD)*, 2016. (Cited on page 63.)

[100] M. Horikoshi, L. Meadows, T. Elken, P. Sivakumar, E. Mascarenhas, J. Erwin, D. Durnov, A. Sannikov, T. Hanawa, and T. Boku, "Scaling collectives on large clusters using intel(r) architecture processors and fabric," in *Proceedings of the Workshops of HPC Asia (HPCASIA)*, 2018. (Cited on page 106.)

[101] S. Horovitz, R. Amos, O. Baruch, T. Cohen, T. Oyar, and A. Deri, "FaaStest - machine learning based cost and performance FaaS optimization," in *Economics of Grids, Clouds, Systems, and Services*. Springer International Publishing, 2019, pp. 171–186. (Cited on page 122.)

[102] M. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and Z. Tari, "A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform," in *Service-Oriented Computing*. Springer International Publishing, 2017, pp. 241–255. (Cited on page 138.)

[103] K. Huppler, "The Art of Building a Good Benchmark," in *Proceedings of the Performance Evaluation and Benchmarking (TPCTC)*, 2009. (Cited on pages 29, 30, 31, 32, 33, and 173.)

[104] *ITU-T Rec. X.902*, International Telecommunication Union (ITU) Std., 1995. (Cited on page 138.)

[105] A. Iosup, R. Prodan, and D. Epema, "IaaS Cloud Benchmarking: Approaches, Challenges, and Experience," in *Proceedings of the Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2012. (Cited on pages 26 and 28.)

[106] A. Ismail, "Energy-driven cloud simulation: existing surveys, simulation supports, impacts and challenges," *Cluster Computing*, vol. 23, no. 4, pp. 3039–3055, 2020. (Cited on page 120.)

[107] D. Jackson and G. Clynch, "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions," in *Proceedings of the Workshop on Serverless Computing (WoSC)*, 2018. (Cited on pages 75, 76, and 81.)

[108] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019. (Cited on pages 41, 51, 52, 53, and 54.)

[109] H. Jeon, C. Cho, S. Shin, and S. Yoon, "A CloudSim-extension for simulating distributed functions-as-a-service," in *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2019. (Cited on pages 72, 73, and 121.)

[110] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu, "Lessons learned from migrating complex stateful applications onto serverless platforms," in *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems (APSYS)*, 2021. (Cited on page 9.)

[111] H. Johng, D. Kim, T. Hill, and L. Chung, "Estimating the Performance of Cloud-Based Systems Using Benchmarking and Simulation in a Complementary Manner," in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, pp. 576–591. (Cited on pages 120 and 187.)

[112] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," *arXiv e-Prints*, vol. 1702.04024, 2017. (Cited on pages 42, 78, 123, 131, and 187.)

[113] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, 2019. (Cited on pages 4, 11, 41, 42, 50, 51, 52, 54, 64, and 118.)

[114] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the International Symposium on Memory Management (ISMM)*, 2013. (Cited on pages 30, 128, and 188.)

[115] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *Proceedings of the International Software Metrics Symposium (METRICS)*, 2004. (Cited on page 27.)

[116] K.-D. Kang, G. Park, H. Kim, M. Alian, N. S. Kim, and D. Kim, "NMAP: Power management based on network packet processing mode transition for latency-critical workloads," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021. (Cited on page 106.)

[117] M. Karpowicz, E. Niewiadomska-Szynkiewicz, P. Arabas, and A. Sikora, "Energy and power efficiency in cloud," in *Computer Communications and Networks*. Springer International Publishing, 2016, pp. 97–127. (Cited on page 105.)

[118] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity," in *Proceedings of the International Workshop on Serverless Computing (WoSC)*, 2019. (Cited on page 64.)

[119] S. Kehrer, J. Scheffold, and W. Blochinger, "Serverless skeletons for elastic parallel processing," in *Proceedings of the International Conference on Big Data Intelligence and Computing (DATACOM)*, 2019. (Cited on page 9.)

[120] W. Kiess and M. Mauve, "A survey on real-world implementations of mobile ad-hoc networks," *Ad Hoc Networks*, vol. 5, no. 3, pp. 324–339, 2007. (Cited on page 38.)

[121] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2019. (Cited on pages 75, 76, and 125.)

[122] ——, "Practical cloud workloads for serverless FaaS," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019. (Cited on page 76.)

[123] J. Kim, J. Park, and K. Lee, "Network resource isolation in serverless cloud function service," in *Proceedings of the International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2019. (Cited on page 75.)

[124] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University, Keele,UK and University of Durham, UK, Tech. Rep. EBSE-2007-01, 2007. (Cited on page 42.)

[125] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the linux virtual machine monitor," in *In Proceedings of the Linux Symposium*, 2007. (Cited on page 20.)

[126] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv: Optimizing the operating system for virtual machines," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2014. (Cited on page 21.)

[127] L. Klaver, T. van der Knaap, J. van der Geest, E. Harmsma, B. van der Waaij, and P. Pileggi, "Towards independent run-time cloud monitoring," in *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2021. (Cited on page 54.)

[128] D. Kliazovich, P. Bouvry, and S. U. Khan, "GreenCloud: a packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1263–1283, 2010. (Cited on page 120.)

[129] G. Kochhar and N. Dandapanthula, "Optimal bios settings for hpc with dell poweredge 12th generation servers," Dell Inc., Tech. Rep., 2012. (Cited on page 103.)

[130] S. Kolb, *On the Portability of Applications in Platform as a Service*. Bamberg University Press, 2019. (Cited on pages xi, 49, and 56.)

[131] A. Koschel, S. Klassen, K. Jdiya, M. Schaaf, and I. Astrova, "Cloud computing: Serverless," in *Proceedings of the International Conference on Information, Intelligence, Systems & Applications (IISA)*, 2021. (Cited on pages 50, 51, 52, and 54.)

[132] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking*. Springer International Publishing, 2020. (Cited on page 26.)

[133] S. Kounev, C. Abad, I. T. Foster, N. Herbst, A. Iosup, S. Al-Kiswany, A. A.-E. Hassan, B. Balis, A. Bauer, A. B. Bondi, K. Chard, R. L. Chard, R. Chatley, A. A. Chien, A. J. J. Davis, J. Donkervliet, S. Eismann, E. Elmroth, N. Ferrier, H.-A. Jacobsen, P. Jamshidi, G. Kousiouris, P. Leitner, P. G. Lopez, M. Maggio, M. Malawski, B. Metzler, V. Muthusamy, A. V. Papadopoulos, P. Patros, G. Pierre, O. F. Rana, R. P. Ricci, J. Scheuner, M. Sedaghat, M. Shahrad, P. Shenoy, J. Spillner, D. Taibi, D. Thain, A. Trivedi, A. Uta, V. van Beek,

E. van Eyk, A. van Hoorn, S. Vasani, F. Wamser, G. Wirtz, and V. Yussupov, "Toward a Definition for Serverless Computing," in *Serverless Computing (Dagstuhl Seminar 21201)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, vol. 11, pp. 89–91. (Cited on pages xi, 50, 51, 52, 54, and 56.)

[134] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017. (Cited on pages 24 and 25.)

[135] N. Kratzke and R. Siegfried, "Towards cloud-native simulations – lessons learned from the front-line of cloud computing," *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, p. 154851291989532, 2020. (Cited on pages 36 and 121.)

[136] K. Kritikos and P. Skrzypek, "Simulation-as-a-Service with Serverless Computing," in *Proceedings of the IEEE World Congress on Services (SERVICES)*, 2019. (Cited on pages 72, 74, 118, and 121.)

[137] J. Kuhlenkamp and S. Werner, "Benchmarking FaaS Platforms: Call for Community Participation," in *Proceedings of the Workshop on Serverless Computing (WoSC)*, 2018. (Cited on pages 11, 30, 43, 71, 72, 75, and 188.)

[138] J. Kuhlenkamp, S. Werner, M. C. Borges, K. E. Tal, and S. Tai, "An evaluation of FaaS platforms as a foundation for serverless big data processing," in *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2019. (Cited on pages 12 and 75.)

[139] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, and D. Wenzel, "Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications," in *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, 2020. (Cited on pages 74, 76, 78, and 81.)

[140] M. Kumar, S. S. Sran, L. Kaur, and J. Singh, "Thermal aware learning based CPU governor," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 11, 2022. (Cited on page 106.)

[141] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache OpenWhisk serverless platform," in *Proceedings of the International Middleware Conference (MIDDLEWARE*, 2018. (Cited on pages 75, 76, and 81.)

[142] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the European Conference on Computer Systems (EUROSYS)*, 2020. (Cited on page 21.)

[143] J. Lambert, R. Monahan, and K. Casey, "Accidental choices—how jvm choice and associated build tools affect interpreter performance," *Computers*, vol. 11, no. 6, p. 96, 2022. (Cited on page 81.)

[144] A. Law, *Simulation Modeling and Analysis*. MCGRAW HILL BOOK CO, 2014. (Cited on pages 36 and 37.)

[145] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of Production Serverless Computing Environments," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2018. (Cited on pages 14, 76, 78, 125, 160, 162, and 164.)

[146] W. Lehner and K.-U. Sattler, "Database as a service (DBaaS)," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010. (Cited on page 4.)

[147] P. Leitner and J. Cito, "Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds," *ACM Transactions on Internet Technology*, vol. 16, no. 3, pp. 1–23, 2016. (Cited on pages 24 and 25.)

[148] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019. (Cited on pages 9, 41, 51, 52, and 55.)

[149] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms," in *Proceedings of the International Workshop on Serverless Computing (WoSC)*, 2019. (Cited on pages 54, 64, 66, 138, and 140.)

[150] ——, "Analyzing open-source serverless platforms: Characteristics and performance," *arXiv e-Prints*, vol. 2106.03601, 2021. (Cited on page 66.)

[151] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," *arXiv e-Prints*, vol. 1708.01388, 2017. (Cited on pages 24 and 25.)

[152] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, p. 220, 2022. (Cited on pages 50, 51, 52, and 54.)

[153] D. J. Lilja, *Measuring Computer Performance*. Cambridge University Press, 2000. (Cited on pages 27 and 102.)

[154] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2021. (Cited on pages 9, 60, 76, 78, 122, 131, 186, and 189.)

[155] P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *arXiv e-Prints*, vol. 1903.12221v1, 2019. (Cited on page 163.)

[156] Y. Lin, K. Ye, Y. Li, P. Lin, Y. Tang, and C. Xu, "BBServerless: A bursty traffic benchmark for serverless," in *Proceedings of Cloud Computing, Held as Part of the Services Conference Federation (SCF)*, 2021. (Cited on pages 72 and 73.)

[157] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *Proceedings of the International Conference on Cloud Engineering (IC2E)*, 2018. (Cited on pages 14, 65, 76, 78, 81, and 163.)

[158] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with

Keep-Alive Workloads," in *Proceedings of the Workshop on Serverless Computing (WoSC)*, 2018. (Cited on page 163.)

[159] P. G. Lopez, A. Slominski, M. Behrendt, and B. Metzler, "Serverless predictions: 2021-2030," *arXiv e-Prints*, vol. 2104.03075, 2021. (Cited on page 42.)

[160] P. Lorenc and M. Woda, "IaaS vs. traditional hosting for web applications - cost effectiveness analysis for a local market," in *Advances in Dependability Engineering of Complex Systems*. Springer International Publishing, 2017, pp. 233–243. (Cited on page 139.)

[161] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," in *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017. (Cited on pages 51, 52, 53, 54, 160, and 161.)

[162] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2015. (Cited on pages 35 and 119.)

[163] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, 2011. (Cited on page 17.)

[164] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013. (Cited on page 21.)

[165] N. Mahmoudi and H. Khazaei, "Simfaas: A performance simulator for serverless computing platforms," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2021. (Cited on pages 76, 88, and 121.)

[166] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *Proceedings of the International Conference on Distributed and Event-based Systems (DEBS)*, 2020. (Cited on pages 76, 77, and 80.)

[167] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking Heterogeneous Cloud Functions," in *Proceedings of the Parallel Processing Workshops (EURO-PAR)*, 2017. (Cited on pages 11, 74, 76, 79, 80, 125, 126, and 186.)

[168] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020. (Cited on pages 72 and 74.)

[169] S. Malla and K. Christensen, "HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)," *Internet Technology Letters*, vol. 3, no. 1, p. e137, 2019. (Cited on pages 72 and 74.)

[170] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *arXiv e-Prints*, vol. 2105.11592, 2021. (Cited on page 138.)

[171] ——, "Deadline-aware dynamic resource management in serverless computing environments," in *Proceedings of the 21st International Symposium on*

Cluster, Cloud and Internet Computing (CCGrid), 2021. (Cited on page 138.)

[172] ——, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Computing Surveys*, vol. 54, no. 11s, 2022. (Cited on pages 41, 51, 52, 54, 56, and 80.)

[173] J. Manner, "Towards Performance and Cost Simulation in Function as a Service," in *Proceedings of the Central European Workshop on Services and their Composition (ZEUS)*, 2019. (Cited on pages 4, 7, 8, and 9.)

[174] ——, "SeMoDe – simulation and benchmarking pipeline for function as a service," in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik*. Otto-Friedrich-University, 2021, no. 105. (Cited on pages 3, 7, 8, 11, 13, 43, 57, 69, 101, 117, and 155.)

[175] ——, "A structured literature review approach to defineserverless computing and function as a service," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2023. (Cited on pages 7, 8, and 41.)

[176] ——, "A simulation framework for function as a service: Supporting materials," 10.5281/zenodo.7447912, 2023. (Cited on pages 3, 61, 69, and 167.)

[177] ——, "Structured literature review for a conceptualization of function as a service," 10.5281/zenodo.7671234, 2023. (Cited on pages 43 and 49.)

[178] J. Manner and S. Böhm, "Lecture notes : Concurrency topics in java," in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik*. Otto-Friedrich-University, 2022, no. 106. (Cited on page 9.)

[179] J. Manner and G. Wirtz, "Impact of Application Load in Function as a Service," in *Proceedings of the Symposium and Summer School On Service-Oriented Computing (SUMMERSOC)*, 2019. (Cited on pages 7, 8, 79, 81, 87, 121, and 122.)

[180] ——, "Resource scaling strategies for open-source faas platforms compared to commercial cloud offerings," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2022. (Cited on pages 3, 7, 8, 11, 27, 28, 29, 36, 41, 54, 62, 66, 71, 73, 95, 117, 155, 186, and 189.)

[181] ——, "Why many benchmarks might be compromised," in *Proceedings of the International Conference on Service-Oriented System Engineering (SOSE)*, 2021. (Cited on pages 7, 8, 101, 103, and 117.)

[182] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *Proceedings of the Workshop on Serverless Computing (WoSC)*, 2018. (Cited on pages 3, 7, 8, 12, 14, 15, 69, 75, 81, 125, 131, 145, 155, and 160.)

[183] J. Manner, S. Kolb, and G. Wirtz, "Troubleshooting serverless functions: a combined monitoring and debugging approach," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2-3, pp. 99–104, 2019. (Cited on pages 7, 8, 34, and 119.)

[184] J. Manner, S. Haarmann, S. Kolb, and O. Kopp, Eds., *ZEUS 2020 - European Workshop on Services and their Composition*. CEUR Workshop Proceedings, 2020. (Cited on page 10.)

[185] J. Manner, M. Endreß, S. Böhm, and G. Wirtz, "Optimizing cloud function configuration via local simulations," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2021. (Cited on pages 3, 7, 8, 12, 17, 27, 36, 41, 60, 62, 69, 75, 77, 79, 80, 81, 82, 117, 138, 155, and 186.)

[186] J. Manner, S. Haarmann, S. Kolb, N. Herzberg, and O. Kopp, Eds., *ZEUS 2021 - European Workshop on Services and their Composition*. CEUR Workshop Proceedings, 2021. (Cited on pages 10 and 80.)

[187] J. Manner, D. Lübke, S. Haarmann, S. Kolb, N. Herzberg, and O. Kopp, Eds., *ZEUS 2022 - European Workshop on Services and their Composition*. CEUR Workshop Proceedings, 2022. (Cited on page 10.)

[188] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: a security perspective," *Journal of Cloud Computing*, vol. 11, no. 1, pp. 1–12, 2022. (Cited on pages 51, 52, 54, and 64.)

[189] H. Martins, F. Araujo, and P. R. da Cunha, "Benchmarking serverless computing platforms," *Journal of Grid Computing*, vol. 18, no. 4, pp. 691–709, 2020. (Cited on pages 9, 60, 74, 75, 76, 79, 81, 186, and 189.)

[190] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017. (Cited on pages 72 and 74.)

[191] I. McGregor, "The relationship between simulation and emulation," in *Proceedings of the Winter Simulation Conference (WSC)*, 2002. (Cited on page 38.)

[192] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, Tech. Rep., 2011. (Cited on pages 3, 4, 42, 48, and 137.)

[193] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2014. (Cited on page 23.)

[194] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4things: Integrating IoT with OpenStack in a smart city context," in *Proceedings of the International Conference on Smart Computing Workshops (SMARTCOMP)*, 2014. (Cited on page 137.)

[195] P. Mieden and P. Partarrieu, "Performance analysis of kvm-based microvmsorchestrated by firecracker and qemu," 2019, Security and Network Engineering, University of Amsterdam, The Netherlands. [Online]. Available: https://dreadl0ck.net/papers/Firebench.pdf (Cited on page 21.)

[196] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proceedings of the Workshop on Hot Topics in Cloud Computing (HOTCLOUD)*, 2019. (Cited on page 162.)

[197] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018. (Cited on pages 54, 66, 72, 73, 138, and 140.)

[198] D. A. Molnar and S. E. Schechter, "Self hosting vs. cloud hosting: Accounting for the security impact of hosting in the cloud," in *Proc. of WEIS*, 2010. (Cited on page 139.)

[199] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *Proceedings of the International Conference on Cloud Engineering (IC2E)*, 2015. (Cited on pages 24 and 25.)

[200] M. Moravcik, P. Segec, M. Kontsek, J. Uramova, and J. Papan, "Comparison of LXC and docker technologies," in *Proceedings of the International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2020. (Cited on page 23.)

[201] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri, "Shadow profiling: Hiding instrumentation costs with parallelism," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007. (Cited on pages 34 and 119.)

[202] S. Newmann, *Building Microservices*. O'Reilly UK Ltd., 2016. (Cited on page 44.)

[203] K. L. Ngo, J. Mukherjee, Z. M. Jiang, and M. Litoiu, "Evaluating the scalability and elasticity of function as a service platform," in *Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE)*, 2022. (Cited on pages 41, 51, 52, and 54.)

[204] T. L. Nguyen and A. Lebre, "Virtual machine boot time model," in *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2017. (Cited on page 21.)

[205] F. A. Oliveira, S. Suneja, S. Nadgowda, P. Nagpurkar, and C. Isci, "A cloud-native monitoring and analytics framework," IBM Research Division Thomas J. Watson Research Center, Tech. Rep. RC25669 (WAT1710-006), 2017. (Cited on pages 119 and 120.)

[206] J. O'Loughlin and L. Gillam, "Performance evaluation for cost-efficient public infrastructure cloud use," in *Economics of Grids, Clouds, Systems, and Services*. Springer International Publishing, 2014, pp. 133–145. (Cited on pages 77, 95, 103, 129, 144, 152, and 164.)

[207] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proceedings of the IEEE World Congress on Services (SERVICES)*, 2019. (Cited on pages 54, 56, 138, and 140.)

[208] S. C. Palepu, D. Chahal, M. Ramesh, and R. Singhal, "Benchmarking the data layer across serverless platforms," in *Proceedings of the Workshop on High Performance Serverless Computing (HiPS)*, 2022. (Cited on pages 72 and 74.)

[209] M. Papoutsoglou, G. M. Kapitsaki, D. German, and L. Angelis, "An analysis of open source software licensing questions in stack exchange sites," *arXiv e-Prints*, vol. 2110.00361, 2021. (Cited on page 137.)

[210] M. Patrou, J. M. Baird, K. B. Kent, and M. Dawson, "Software evaluation methodology of node.js parallelism under variabilities in scalable systems,"

in *Proceedings of the Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2020. (Cited on page 105.)

[211] M. Pawlik, K. Figiela, and M. Malawski, "Performance evaluation of parallel cloud functions," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2018. (Cited on pages 76, 79, and 186.)

[212] K. Pawlikowski, H.-D. Jeong, and J.-S. Lee, "On credibility of simulation studies of telecommunication networks," *IEEE Communications Magazine*, vol. 40, no. 1, pp. 132–139, 2002. (Cited on page 37.)

[213] I. Pelle, J. Czentye, J. Doka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on AWS," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2019. (Cited on pages 12, 75, 77, 95, 129, 144, and 164.)

[214] R. Pellegrini, I. Ivkic, and M. Tauber, "Function-as-a-Service Benchmarking Framework," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2019. (Cited on pages 76 and 79.)

[215] N. Pemberton and J. Schleier-Smith, "The serverless data center: Hardware disaggregation meets serverless computing," in *Proceedings of the Workshop on Resource Disaggregation (WORDS)*, 2019. (Cited on page 42.)

[216] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2017. (Cited on pages 169 and 188.)

[217] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*, 2020. (Cited on page 77.)

[218] A. Pi, W. Chen, X. Zhou, and M. Ji, "Profiling distributed systems in lightweight virtualized environments with logs and resource metrics," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2018. (Cited on pages 35, 119, and 120.)

[219] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *Proceedings of the International Conference on Data Science and Data Intensive Systems (DSS)*, 2015. (Cited on page 56.)

[220] ——, "ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers," *Software: Practice and Experience*, vol. 47, no. 4, pp. 505–521, 2016. (Cited on page 120.)

[221] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974. (Cited on pages 18 and 19.)

[222] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 291–304, 2011. (Cited on page 21.)

[223] M. Portnoy, *Virtualization Essentials*. John Wiley and Sons Inc, 2016. (Cited on pages xi, 17, and 19.)

[224] H. Puripunpinyo and M. H. Samadzadeh, "Effect of optimizing Java deployment artifacts on AWS Lambda," in *Proceedings of the Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017. (Cited on page 162.)

[225] D. F. Quaresma, T. E. Pereira, and D. Fireman, "Validation of a simulation model for faas performance benchmarking using predictive validation," *arXiv e-Prints*, vol. 2106.15555, 2021. (Cited on pages 72 and 74.)

[226] U. U. Rahman, K. Bilal, A. Erbad, O. Khalid, and S. U. Khan, "Nutshell—simulation toolkit for modeling data center networks and cloud computing," *IEEE Access*, vol. 7, pp. 19 922–19 942, 2019. (Cited on page 120.)

[227] A. Randal, "The ideal versus the real," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–31, 2020. (Cited on pages 17, 18, and 22.)

[228] K. Razavi and T. Kielmann, "Scalable virtual machine deployment using VM image caches," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. (Cited on page 21.)

[229] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 2010. (Cited on pages 34, 119, and 120.)

[230] A. Reuter, T. Back, and V. Andrikopoulos, "Cost efficiency under mixed serverless and serverful deployments," in *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020. (Cited on page 122.)

[231] S. Ristov, C. Hollaus, and M. Hautz, "Colder than the warm start and warmer than the cold start! experience the spawn start in faas providers," in *Proceedings of the Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED)*, 2022. (Cited on pages 76, 77, and 162.)

[232] D. Rivas, F. Guim, J. Polo, and D. Carrera, "Performance characterization of video analytics workloads in heterogeneous edge infrastructures," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 14, p. e6317, 2023. (Cited on page 105.)

[233] M. Roberts and J. Chapin, *What Is Serverless?* O'Reilly Media, 2017. (Cited on pages 42, 50, 51, 52, and 54.)

[234] S. Robinson, "General concepts of quality for discrete-event simulation," *European Journal of Operational Research*, vol. 138, no. 1, pp. 103–117, 2002. (Cited on pages 36 and 37.)

[235] P. Rosati, F. Fowley, C. Pahl, D. Taibi, and T. Lynn, "Right scaling for right pricing: A case study on total cost of ownership measurement for cloud migration," in *Communications in Computer and Information Science*. Springer International Publishing, 2019, pp. 190–214. (Cited on page 139.)

[236] A. Rumyantsev, P. Zueva, K. Kalinina, and A. Golovin, "Evaluating a single-server queue with asynchronous speed scaling," in *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 157–172. (Cited on page 106.)

[237] M. Sadaqat, M. Sánchez-Gordón, and R. C. Palacios, "Benchmarking serverless computing: Performance and usability," *Journal of Information Technology Research*, vol. 15, no. 1, pp. 1–17, 2022. (Cited on pages 72 and 74.)

[238] S. G. Sáez, V. Andrikopoulos, M. Hahn, D. Karastoyanova, F. Leymann, M. Skouradaki, and K. Vukojevic-Haupt, "Performance and cost evaluation for the migration of a scientific workflow infrastructure to the cloud," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2015. (Cited on page 139.)

[239] K. A. Scarfone, M. P. Souppaya, and P. Hoffman, "Sp 800-125. guide to security for full virtualization technologies," National Institute of Standards & Technology (NIST), Gaithersburg, MD, USA, Tech. Rep., 2011. (Cited on pages 18, 19, and 20.)

[240] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020. (Cited on pages 13, 43, 71, 75, 131, and 151.)

[241] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," *arXiv e-Prints*, vol. 2205.07696, 2022. (Cited on pages 76 and 78.)

[242] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021. (Cited on pages 41, 42, 51, 52, and 54.)

[243] N. Schneidewind *et al.*, "IEEE standard for a software quality metrics methodology," The Institute of Electrical and Electronics Engineers, Tech. Rep. IEEE Std 1061™-1998 (R2009), 2009. (Cited on pages 27 and 28.)

[244] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: A Cautionary Tale," in *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2006. (Cited on pages 28 and 29.)

[245] D. Shadija, M. Rezai, and R. Hill, "Microservices," in *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, 2017. (Cited on page 44.)

[246] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022. (Cited on pages 50, 51, 52, and 54.)

[247] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019. (Cited on page 118.)

[248] A. Sharma and M. O. Joshi, "Openstack ceilometer data analytics & predictions," in *Proceedings of the International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2016. (Cited on page 138.)

[249] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. Pai, "Challenges in computer architecture evaluation," *Computer*, vol. 36, no. 8, pp. 30–36, 2003. (Cited on page 36.)

[250] A. Slominski, V. Muthusamy, and V. Isahagian, "The Future of Computing is Boring (and that is exciting!)," in *Proceedings of the International Conference on Cloud Engineering (IC2E)*, 2019. (Cited on page 4.)

[251] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore CPUs," *Computer*, vol. 43, no. 3, pp. 24–32, 2010. (Cited on page 124.)

[252] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for DevOps organizations," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2021. (Cited on page 47.)

[253] I. Sommerville, *Software Engineering*. Pearson, 2015. (Cited on page 36.)

[254] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, "PanOpticon: A comprehensive benchmarking tool for serverless applications," in *Proceedings of the International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2020. (Cited on pages 72 and 74.)

[255] J. Spillner, "Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation," *arXiv e-Prints*, vol. 1703.07562, 2017. (Cited on pages 12 and 75.)

[256] ——, "Transformation of python applications into function-as-a-service deployments," *arXiv e-Prints*, vol. 1705.08169, 2017. (Cited on page 4.)

[257] ——, "Serverless computing and cloud function-based applications," in *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2019. (Cited on page 41.)

[258] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002. (Cited on pages 12, 124, and 136.)

[259] C. Starner and M. Chessin, "Using emulation to enhance simulation," in *Proceedings of the Winter Simulation Conference (WSC)*, 2010. (Cited on page 38.)

[260] V. Stodden, J. Seiler, and Z. Ma, "An empirical analysis of journal policy effectiveness for computational reproducibility," *Proceedings of the National Academy of Sciences*, vol. 115, no. 11, pp. 2584–2589, 2018. (Cited on page 31.)

[261] A. Stupple, D. Singerman, and L. A. Celi, "The reproducibility crisis in the age of digital medicine," *npj Digital Medicine*, vol. 2, no. 1, 2019. (Cited on page 188.)

[262] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. (Cited on page 23.)

[263] O. Sürer and M. Plumlee, "Calibration using emulation of filtered simulation results," in *Proceedings of the Winter Simulation Conference (WSC)*, 2021. (Cited on pages 38 and 124.)

[264] D. Taibi, N. E. Ioini, C. Pahl, and J. Niederkofler, "Patterns for serverless functions (function-as-a-service): A multivocal literature review," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2020. (Cited on pages 41, 43, and 49.)

[265] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. (Cited on page 44.)

[266] W. Tian, M. Xu, A. Chen, G. Li, X. Wang, and Y. Chen, "Open-source simulators for cloud computing: Comparative study and challenging issues," *Simulation Modelling Practice and Theory*, vol. 58, pp. 239–254, 2015. (Cited on page 120.)

[267] *TPC BENCHMARK^{TM}C*, Transaction Processing Performance Council Std., Rev. 5.11, 2010. (Cited on page 30.)

[268] UEFI-Forum, "Acpi specification, version 6.3," UEFI Forum, Inc., Tech. Rep., 2019. (Cited on pages 103 and 104.)

[269] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005. (Cited on page 17.)

[270] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. (Cited on pages 72 and 74.)

[271] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *Proceedings of the International Conference on Omni-layer Intelligent Systems (COINS)*, 2020. (Cited on pages 12, 75, 131, and 163.)

[272] H. Valter, A. Karlsson, and M. Pericàs, "Energy-efficiency evaluation of openmp loop transformations and runtime constructs," *arXiv e-Prints*, vol. 2209.04317, 2022. (Cited on page 106.)

[273] E. van Eyk and A. Iosup, "Addressing Performance Challenges in Serverless Computing," in *Proceedings of the ICT.OPEN*, 2018. (Cited on pages 14 and 160.)

[274] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures," in *Proceedings of the International Workshop on Serverless Computing (WoSC)*, 2017. (Cited on pages xi, 41, 50, 51, 52, 54, and 56.)

[275] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is More: From PaaS to Present Cloud Computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018. (Cited on pages 4 and 50.)

[276] E. van Eyk, A. Iosup, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, and C. L. Abad, "The SPEC-RG reference

architecture for FaaS: From microservices and containers to serverless platforms," *IEEE Internet Computing*, vol. 23, no. 6, pp. 7–18, 2019. (Cited on pages 50, 54, and 64.)

[277] E. van Eyk, J. Scheuner, S. Eismann, C. L. Abad, and A. Iosup, "Beyond Microbenchmarks: The SPEC-RG Vision for A Comprehensive Serverless Benchmark," in *Proceedings of the Third Workshop on Hot Topics in Cloud Computing Performance (HotCloudPerf)*, 2020. (Cited on pages 72 and 73.)

[278] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 2016. (Cited on page 74.)

[279] J. von Kistowski, N. Herbst, D. Zoller, S. Kounev, and A. Hotho, "Modeling and Extracting Load Intensity Profiles," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2015. (Cited on pages 29, 30, 31, 32, and 33.)

[280] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2021. (Cited on page 58.)

[281] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proceedings of the Fourteenth EuroSys Conference 2019*. Association for Computing Machinery (ACM), 2019. (Cited on page 57.)

[282] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2018, pp. 133–146. (Cited on pages 51, 52, 53, 54, 76, 78, 81, 95, 123, and 185.)

[283] R. Weingärtner, G. B. Bräscher, and C. B. Westphall, "Cloud resource management: A survey on forecasting and profiling models," *Journal of Network and Computer Applications*, vol. 47, pp. 99–106, 2015. (Cited on pages 34 and 119.)

[284] J. Wen, Z. Chen, and X. Liu, "Software engineering for serverless computing," *arXiv e-Prints*, vol. 2207.13263, 2022. (Cited on pages 41, 51, 52, and 54.)

[285] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed andnetworked applications," University of Washington, Tech. Rep. UW-CSE-02-02-01, 2002. (Cited on page 18.)

[286] S. Winzinger and G. Wirtz, "Model-based analysis of serverless applications," in *Proceedings of the International Workshop on Modelling in Software Engineering (MiSE)*, 2019. (Cited on page 35.)

[287] ——, "Applicability of coverage criteria for serverless applications," in *Proceedings of the International Conference on Service Oriented Systems Engineering (SOSE)*, 2020. (Cited on page 119.)

[288] ——, "Data flow testing of serverless functions," in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2021. (Cited on pages 35 and 177.)

[289] ——, "Automatic test case generation for serverless applications," in *Proceedings of the International Conference on Service-Oriented System Engineering (SOSE)*, 2022. (Cited on page 42.)

[290] E. Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley, 2016. (Cited on pages 44 and 189.)

[291] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2020. (Cited on pages 50, 51, 52, 53, and 54.)

[292] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov, "Modeling and Automated Deployment of Serverless Applications using TOSCA," in *Proceedings of the International Conference on Service-Oriented Computing and Applications (SOCA)*, 2018. (Cited on page 44.)

[293] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2020. (Cited on pages 9, 62, 76, 79, 82, and 186.)

[294] V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster, "A systematic mapping study on engineering function-as-a-service platforms and tools," in *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, 2019. (Cited on pages 43, 56, 57, and 71.)

[295] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "From serverful to serverless: A spectrum of patterns for hosting application components." in *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, 2021. (Cited on page 50.)

[296] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "FaaSten your decisions: A classification framework and technology review of function-as-a-service platforms," *Journal of Systems and Software*, vol. 175, p. 110906, 2021. (Cited on pages 43 and 45.)

[297] F. V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé, "Intelligent colocation of HPC workloads," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 125–137, 2021. (Cited on page 106.)

[298] M. Zakarya and L. Gillam, "Modelling resource heterogeneities in cloud simulations and quantifying their accuracy," *Simulation Modelling Practice and Theory*, vol. 94, pp. 43–65, 2019. (Cited on pages 27 and 122.)

[299] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing," in *Proceedings of the Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2019. (Cited on pages 129 and 144.)

[300] Y. Zhang, K. Ye, and C. Xu, "An experimental analysis of function performance with resource allocation on serverless platform," in *Proceedings of Cloud Computing - CLOUD 2021 - 14th International Conference, Held as Part of the Services Conference Federation*, 2021. (Cited on pages 62, 71, 72, 73, 82, and 186.)

[301] K. H. Zou, K. Tuncali, and S. G. Silverman, "Correlation and Simple Linear Regression," *Radiology*, vol. 227, no. 3, pp. 617–628, 2003. (Cited on page 170.)

University of Bamberg Press

Serverless Computing is seen as a game changer in operating large-scale applications. While practitioners and researches often use this term, the concept they actually want to refer to is Function as a Service (FaaS). In this new service model, a user deploys only single functions to cloud platforms where the cloud provider deals with all operational concerns – this creates the notion of serverless computing for the user.

Nonetheless, a few configurations for the cloud function are necessary for most commercial FaaS platforms as they influence the resource assignments like CPU time and memory. Due to these options, there is still an abstracted perception of servers for the FaaS user. The resource assignment and the different strategies to scale resources for public cloud offerings and on-premise hosted open-source platforms determine the runtime characteristics of cloud functions and are in the focus of this work. Compared to cloud offerings like Platform as a Service, two out of the five cloud computing characteristics improved. These two are rapid elasticity and measured service. FaaS is the first computational cloud model to scale functions only on demand. Due to an independent scaling and a strong isolation via virtualized environments, functions can be considered independent of other cloud functions. Therefore, noisy neighbor problems do not occur. The second characteristic, measured service, targets billing. FaaS platforms measure execution time on a millisecond basis and bill users accordingly based on the function configuration. This leads to new performance and cost trade-offs.

Therefore, this thesis proposes a simulation approach to investigate this tradeoff in an early development phase. The alternative would be to deploy functions with varying configurations, analyze the execution data from several FaaS platforms and adjust the configuration. However, this alternative is time-consuming, tedious and costly. To provide a proper simulation, the development and production environment should be as similar as possible. This similarity is also known as dev-prod parity. Based on a new methodology to compare different virtualized environments, users of our simulation framework are able to execute functions on their machines and investigate the runtime characteristics for different function configurations at several cloud platforms without running their functions on the cloud platform at all. A visualization of the local simulations guide the user to choose an appropriate function configuration to resolve the mentioned trade-off dependent on their requirements.

ISBN: 978-3-86309-979-4

9 783863 099794

www.uni-bamberg.de/ubp