TU WIEN Informatics

# Self-Adaptive Serverless Edge Computing for Edge Intelligence

DISSERTATION

zur Erlangung des akademischen Grades

**Doktor der Technischen Wissenschaften**

eingereicht von

**Dipl.-Ing. Philipp Raith, Bsc**
Matrikelnummer 01425076

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Zweitbetreuung: Assistant Prof. Dr. Stefan Nastic

Diese Dissertation haben begutachtet:

Rajiv Ranjan

Ming Zhao

Wien, 27. August 2025

Philipp Raith

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

# Self-Adaptive Serverless Edge Computing for Edge Intelligence

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktor der Technischen Wissenschaften**

by

**Dipl.-Ing. Philipp Raith, Bsc**
Registration Number 01425076

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Second advisor: Assistant Prof. Dr. Stefan Nastic

The dissertation has been reviewed by:

Rajiv Ranjan

Ming Zhao

Vienna, August 27, 2025

Philipp Raith

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Philipp Raith, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. August 2025

Philipp Raith

# Acknowledgements

Everyone's PhD journey is different and takes different turns and twists throughout. One constant that every journey shares is the people, the colleagues, the families, and the friends who support one on this road. Therefore, I am grateful for everyone with whom I had the pleasure to work and follow me on this journey - without them, I would not be where I am today. When I started my studies, I was not sure where this would lead me, but with the immense support from my master's thesis supervisors, Schahram Dustdar and Thomas Rausch, I was able to find a path that I was to follow. Working under Thomas Rausch as a project assistant during my master's studies has helped me find a research area I am passionate about. I also thank my co-supervisor, Stefan Nastic, who has given me structure, guidance, and support throughout my journey. I am grateful for having Rajiv Ranjan and Ming Zhao agree to be my reviewers and provide valuable feedback, as well as Uwe Zdun and Maria Christakis to be my proficiency evaluation committee.

I also want to thank the whole team of the Distributed Systems Group. It was delightful to have worked with Renate Weiss and Christine Kamper - my success is due to your support in navigating the administration jungle. An integral part of my success is also Alexander Knoll - his continuous support has been helping me since day one in conducting my research.

I must thank Alireza Furutanpey for growing together on our journey - I could not have wished for a better PhD buddy. Teaching has also been part of my journey, where I had the pleasure of working and befriending my colleague Andrea Morichetta. At the same time, I thank the students with whom I have worked and contributed to my research.

A crucial part of my journey was also the research internships at Hewlett Packard Labs under the supervision of Dejan Milojicic and Gourav Rattihalli. I am grateful for this opportunity and have grown as a researcher through their guidance and the fruitful discussions with my colleagues and friends Vijay Thurimella and Aditya Dhakal.

Last but not least, I want to thank my parents and family, who have enabled this journey in the first place, and my friends, who provided the best support I could have wished for.

The research presented in this thesis was funded by TU Wien research funds, European Union's Horizon 2020 research and innovation programme under grant agreement No 871403.

# Kurzfassung

Das Edge Intelligence Anwendungsparadigma ist ein Oberbegriff für KI-Anwendungen, die über das Kontinuum von Edge und Cloud hinweg zusammenarbeiten. Kennzeichnend für sie sind strenge Anforderungen in Bezug auf geringe Latenzzeiten, Datenschutz, Personalisierung und komplexe Interaktionen. Das Edge-Cloud-Kontinuum ist in Bezug auf die Ressourcen heterogen, hochdynamisch, da sich die Benutzer ständig bewegen, und die Senkung des Energieverbrauchs ist bei Edge- und Cloud-Implementierungen wichtig. Aus diesem Grund ist eine autonome Anwendungsorchestrierung von entscheidender Bedeutung für die breite Einführung von Edge Intelligence. Die derzeitigen Plattformparadigmen unterstützen diese neuen Anwendungen jedoch nicht vollständig. Daher werden in dieser Arbeit vier Beiträge vorgestellt, die sich mit Plattformen für Edge Intelligence befassen. Wir führen eine auf Edge Intelligence basierende Anwendungsfallstudie durch, um Plattformanforderungen und eine Vision für eine Edge Intelligence as a Service-Plattform abzuleiten. Basierend auf diesen Anforderungen und der notwendigen Unterstützung für ein autonomes Anwendungsmanagement untersuchen wir das Serverless Edge Computing. Serverless Edge Computing ist ein Plattformparadigma, das ein vollständig automatisiertes Management des Anwendungslebenszyklus verspricht, indem es das heterogene Edge-Cloud-Kontinuum abstrahiert und es den Benutzern ermöglicht, ihren Code einfach als kleine, in sich geschlossene Funktionen hochzuladen. Wir geben einen Überblick über den aktuellen Stand der Serverless Edge Computing-Angebote, die von Open Source über kommerzielle Angebote bis hin zur Forschung reichen. Die Analyse der Edge-Intelligenz und die Literaturrecherche zeigen, dass es noch offene Herausforderungen in Bezug auf Mobilität und Energieverbrauch gibt. Daher stellen wir zwei neuartige Ansätze vor, die Serverless Edge Computing-Plattformen erweitern und die Herausforderung mobiler Nutzer und des Energieverbrauchs angehen. Der mobilitätsbewusste Ansatz schlägt eine dezentrale Plattform vor, die sich auf Schwellenwerte stützt, um Funktionen als Reaktion auf mobile Benutzer zu verwalten (z. B. in Smart Cities). Der zweite Ansatz entwickelt einen Container-Scheduler, der sich auf ein neuronales Graphen-Netzwerk stützt, um den Stromverbrauch von Geräten abzuschätzen und den Gesamtenergieverbrauch zu reduzieren. Auf der Grundlage dieser beiden Orchestrierungsstrategien haben wir die Herausforderung entdeckt, sie richtig zu parametrisieren, und schlagen eine selbstanpassende Plattform vor, um dieses Problem zu lösen. Wir stellen ein Test-Framework und einen Serverless-Edge-Simulator vor, die wir zum Aufbau eines selbstanpassenden Plattform-Prototyps vereinen. Dieser Prototyp wird mit realen Überwachungsdaten in

eine Simulation eingespeist, die Einblicke und eine fliegende Optimierung der Orchestrierungsstrategieparameter ermöglicht. Darüber hinaus schlagen wir eine Methode vor, um vorhersagebasierte Orchestrierungsstrategien durch die Erzeugung synthetischer Datensätze zu testen. Der erste selbstanpassende Ansatz zielt auf schwellenwertbasierte Strategien während der Laufzeit ab, während der zweite ein auf Vorhersagemodellen basierender Ansatz ist. Beide Ansätze ermöglichen eine selbstanpassende Serverless Edge Computing-Plattform, die Edge Intelligence-Anwendungen unterstützt.

# Abstract

The Edge Intelligence application paradigm is an umbrella term for AI applications that cooperate across the edge-cloud continuum. Stringent requirements around low latency, privacy, personalization, and complex interactions are characteristic of them. The edge-cloud continuum is heterogeneous in terms of resources, highly dynamic as users constantly move, and reducing energy consumption is important in edge and cloud deployments. Autonomous application orchestration is crucial for the widespread adoption of Edge Intelligence. However, current platform paradigms do not fully support Edge Intelligence. Therefore, this thesis presents four contributions that investigate platforms for Edge Intelligence. We conduct a use case study to derive platform requirements and a vision for an Edge Intelligence as a Service platform. Based on these requirements and the necessary support for autonomous application management, we further investigate Serverless Edge Computing. Serverless Edge Computing is a platform paradigm that promises to deliver a fully automated application lifecycle management by abstracting away the heterogeneous edge-cloud continuum and allowing users to simply upload their code as small self-contained functions. We review the current state of Serverless Edge Computing offerings that range from open source to commercial offerings and research. The analysis of Edge Intelligence and the literature review discern that open challenges around mobility and energy-awareness remain. Therefore, we present two novel approaches, which extend Serverless Edge Computing platforms and tackle the challenge of mobile users and energy consumption. The mobility-aware approach proposes a decentralized platform that relies on thresholds to manage functions in response to mobile users (e.g., in Smart Cities). The second approach builds a container scheduler that relies on a graph neural network to estimate the power consumption of devices and reduce overall energy consumption. Based on these two orchestration strategies, we discovered the challenge of properly parameterizing them and propose a self-adaptive platform to tackle this issue. We present a testing framework and a Serverless Edge simulator that we unify to build a self-adaptive platform prototype. This prototype is fed real-world monitoring data into a simulation that provides insights and on-the-fly optimization of orchestration strategy parameters. In addition, we propose a method to bootstrap prediction-based orchestration strategies by generating synthetic datasets. The former self-adaptive approach targets threshold-based strategies during runtime, while the latter is a prediction model-based approach. Both approaches facilitate a self-adaptive Serverless Edge Computing platform that supports Edge Intelligence applications.

# Contents

# Publications

The research presented in this thesis is partly based on the following peer-reviewed publications and tech reports:

- Philipp Raith and S. Dustdar, "Edge Intelligence as a Service," 2021 IEEE International Conference on Services Computing (SCC), Chicago, IL, USA, 2021, pp. 252-262
- Philipp Raith, T. Rausch, S. Dustdar, F. Rossi, V. Cardellini and R. Ranjan, "Mobility-Aware Serverless Function Adaptations Across the Edge-Cloud Continuum," 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC), Vancouver, WA, USA, 2022, pp. 123-132
- Philipp Raith, T. Rausch, P. Prüller, A. Furutanpey and S. Dustdar, "An End-to-End Framework for Benchmarking Edge-Cloud Cluster Management Techniques," 2022 IEEE International Conference on Cloud Engineering (IC2E), CA, USA, 2022, pp. 22-28
- Philipp Raith, S. Nastic and S. Dustdar, "Serverless Edge Computing—Where We Are and What Lies Ahead," in IEEE Internet Computing, vol. 27, no. 3, pp. 50-64, May-June 2023
- Philipp Raith, T. Rausch, A. Furutanpey, S. Dustdar, "faas-sim: A trace-driven simulation framework for serverless edge computing platforms", Softw Pract Exper. 2023; 53(12): 2327–2361
- Philipp Raith, G. Rattihalli, A. Dhakal, S. R. Chalamalasetti, D. Milojicic, E. Frachtenberg, S. Nastic and S. Dustdar, "Opportunistic Energy-Aware Scheduling for Container Orchestration Platforms Using Graph Neural Networks," 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Philadelphia, PA, USA, 2024, pp. 299-306
- Philipp Raith, S. Nastic and S. Dustdar, "SimuScale: Optimizing Parameters for Autoscaling of Serverless Edge Functions Through Co-Simulation," 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), Shenzhen, China, 2024, pp. 305-315

- Philipp Raith, G. Rattihalli, V. Thurimella, P. Bruel, D. Milojicic, C. Bash, S. Nastic and S. Dustdar, "ConVert: Proactive Server Consolidation and Vertical Scaling in Serverless Computing", *Under Review*
- Philipp Raith, A. Furutanpey, N. Lukic, V. Thurimella, S. Nastic, "EdgeCloudForge: Simulation-Driven Synthetic Dataset Generation for Proactive Serverless Edge Function Autoscaling", *Accepted - IC2E 2025*

CHAPTER 1

# Introduction

## 1.1 Motivation

Edge Intelligence (EI) applications aim to provide the next generation of Artificial Intelligence applications by combining all resources available in the edge-cloud continuum [RD21]. Applications like Mobile Augmented Reality or Personal Assistance are typically user-centered and require low latency and privacy, typically achieved through on-device computation [RRD+22, ZCL+19]. However, AI training and inference can not be handled on all devices due to device resource constraints. Therefore, the computation is split across the edge-cloud continuum [ZCL+19]. To this end, EI developers require platforms that can autonomously manage application deployments while satisfying their requirements. Serverless computing platforms promise to abstract away the infrastructure from their customers by autonomously orchestrating applications. The orchestration includes starting and removing applications to withstand incoming requests and distributing these requests following operational goals. Goals include user and platform-centered requirements, such as increasing resource efficiency or decreasing energy consumption. While this platform paradigm has found significant adoption in Cloud Computing, its extension, Serverless Edge Computing, seems promising for EI applications but has yet to be explored [NRF+22]. Building and optimizing orchestration strategies for Serverless Edge Computing platforms is the main topic of this thesis.

Several challenges have yet to be addressed by Serverless Edge platforms that we will investigate [ATC+21, NRF+22, RND23]. They revolve around serving mobile user requests and energy efficiency, both critical and needing to be addressed [PSP+21, RND23, ATC+21]. In Multi-Access Edge Computing (MEC), service providers can host applications on radio tower base stations. When mobile users connect, they can send requests to these stations. Due to delays, such as cold startups, application instances can not instantaneously spawn in new locations. Platforms need to know exactly where to start new instances in response to incoming workload changes. Therefore, platforms must

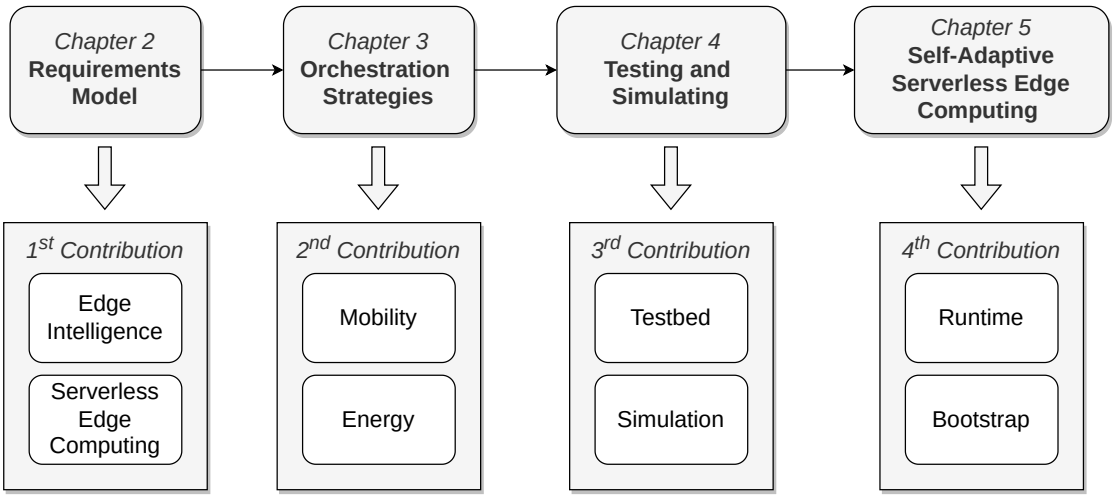



Figure 1.1: Main chapters and contributions of this thesis

be mobility-aware to reduce inter-cluster network traffic and keep end-to-end latency minimal. Besides, Serverless Edge Computing must pay attention to the emerging challenges in cloud data centers. To this end, we want to focus on the issue of sustainable Serverless Computing [PSP+21]. Specifically, reducing energy consumption is critical and aids resource-constrained devices across the edge-cloud continuum. Due to hardware accelerators, compute-intensive applications, such as AI, consume large amounts of energy. Embracing the fine-grained placement strategy of serverless computing to increase resource usage efficiency and placing them in an energy-aware manner can decrease overall energy consumption.

Figure 1.1 depicts an overview of the main chapters of this thesis and the contributions presented in each. The context and ideas of each chapter build on the predecessors, as the thesis explores challenges surrounding Serverless Edge Computing, which finishes with a vision of a self-adaptive platform for Edge Intelligence.

The first stage, Chapter 2, analyzes application platform requirements for Edge Intelligence and investigates open challenges for the promising platform paradigm of Serverless Edge Computing. This requirements model builds the first contribution. The next stage introduces our second contribution in Chapter 3. This stage introduces and evaluates strategies to tackle mobility and energy-related challenges in Serverless Edge Computing using a threshold- and prediction-based approach. We focus on the mobility aspect in an edge-cloud scenario while tackling energy awareness in a cloud environment. Literature exists for both; while mobility is a well-studied problem in edge-cloud systems, energy-awareness in Serverless Computing is just gaining traction [PSP+21]. An important aspect of our orchestration strategies is that the mobility-aware strategy builds on thresholds, while the energy-aware one builds on an AI model. Both are built on parameters (i.e., thresholds or a trained prediction model) that must be set before deploying them. The challenge of finding those parameters stems from the edge-cloud environment. Edge-cloud

systems and EI are dynamic, and the environment and infrastructure in which they operate change over time. For example, an infrastructure for the Industrial Internet of Things (IIoT) will have different resources than one for a Smart City deployment [RLF+20, RRD21]. Moreover, request patterns can vary across Smart Cities and also across applications [RND24, JHA+23] Consequently, orchestration strategies must adapt to counteract this dynamicity. As these directly impact the decisions of the orchestration strategies, they influence the performance and Quality of Service of deployed user applications.

Therefore, this thesis's third and fourth stages tackle the challenge of finding orchestration parameters. The goal is to build a self-adaptive Serverless Edge Computing platform that autonomously tunes parameters at runtime and bootstraps them. To this end, the third contribution, presented in Chapter 4, proposes a simulation [RRFD23] and a testbed framework [RRP+22]. These unified tools offer the necessary features for building a self-adaptive platform. The testbed and simulation framework are combined into a co-simulation framework that sends real-time monitoring data from the infrastructure to the simulation. We use the term co-simulation to describe a system that facilitates simulations that replicate the real world and are similar to symbiotic simulations and digital twins [TPS+22, OMS+18]. The simulation can then implement strategies to optimize the parameters and propagate them back to the platform (i.e., the testbed).

The self-adaptive platform, presented in Chapter 5 as the fourth contribution, revolves around optimizing the parameters. Parameters can include AI models, thresholds, etc., and depend on the orchestration strategies' implementation. The self-adaptive platform must account for heterogeneous infrastructures and applications, which makes a flexible approach crucial. Therefore, we propose a co-simulation-based framework in this work that creates a feedback loop between the real world and the simulation. We can use the simulation to optimize the parameters and apply them in real-time during runtime. In contrast, we can also use the simulation beforehand to generate synthetic datasets to train the models.

## 1.2 Problem Statement

The vast amount of heterogeneous hardware in the edge-cloud continuum requires significant changes to application deployment as manual management is not feasible [RD21]. In recent years, Serverless Computing has shifted the responsibilities towards the platform providers to manage application deployments autonomously. The three main orchestration strategies, scheduling, autoscaling, and request routing, are responsible for this. However, current platforms are unsuited for EI and require significant changes [RND23, ATC+21]. Serverless Edge Computing aims to address these issues and provide a suitable autonomous platform enabling EI [NRF+22]. We identify two key orchestration challenges that Serverless Edge platforms must tackle: mobility and energy. Mobility awareness enables platforms to shift applications along the moving users' trajectories, mainly at the network's edge [ZPX+18]. Energy awareness is essential for battery-powered

devices and large cloud data centers that aim to reduce their overall power consumption [PSP+21]. Before tackling these challenges, we formulate our first research question that revolves around the requirements of EI and Serverless Edge Computing platforms.

**RQ1. What are the requirements that Serverless Edge Computing needs to fulfill for Edge Intelligence?**

Edge Intelligence imposes several challenges on application platforms due to stringent requirements and the vision of a fully coordinated edge-cloud continuum. These requirements revolve around the applications, such as response times, and the platform, such as energy consumption. Additionally, the edge-cloud continuum is heterogeneous regarding available devices, their computational power and available resources, and dispersion across geographical regions, resulting in heterogeneous network conditions. Our approach to answering this question is two-fold. First, we lay out Edge Intelligence using a use case study and determine general EI application characteristics and requirements. This entails the types of data processed, the general application domains they belong to, and the requirements they impose on runtime aspects, such as application performance. Second, we investigate Serverless Edge Computing offerings and focus on their support for applications that facilitate the heterogeneous resources, such as Edge Intelligence. These two analysis pieces formulate a requirements model that stretches from the general view of EI applications to the specific challenges of Serverless Edge Computing, fulfilling the need for next-generation applications across the edge-cloud continuum.

**RQ2. Which orchestration strategies can build the foundation for mobility- and energy-aware Serverless Edge Computing platforms?**

Mobility- and energy-awareness are crucial to the success of Serverless Edge Computing. Therefore, appropriate orchestration strategies must be proposed.

Mobility-aware platforms must determine where applications should be put in response to user movements. This is a common scenario for EI applications, such as Mobile Augmented Reality in Smart Cities, where users move around a city and request object detection based on what they see, which requires low-latency responses [RHS+21]. However, such strategies are limited in how much information they can base decisions on. The heterogeneous infrastructure discrepancy in the edge-cloud continuum causes this. Moving away from large-scale data centers with high-performance networking infrastructures causes significant network performance degradation. Specifically, networks tend to be more unreliable the more we move to the network's edge [RLF+20, RRD+22]. Therefore, Serverless Edge platforms must accordingly limit monitoring when deploying operational strategies across the edge-cloud continuum. Energy awareness is crucial for battery-powered devices and in large-scale clusters where providers try to minimize the energy footprint [PSP+21]. Application schedulers must place application instances to reduce overall energy consumption. While research approaches focus on mobility- or energy-awareness, they lack integration in a Serverless context and most often do not consider the monitoring overhead [RRD+22, PSP+21].

RQ2 aims to build and evaluate mobility- and energy-aware strategies for Serverless

Edge Computing. We propose two sets of orchestration strategies, one focusing on mobility while the other on an energy-aware scheduler. Our orchestration strategies integrate well into real-world systems, such as Kubernetes. Kubernetes is a popular container orchestration service that implements low-level services on which we build. For example, Kubernetes offers an API to create and remove containers; we implement strategies communicating with this API. Our mobility-aware [RRD+22] strategy relies on thresholds, while our energy-aware [RRD+24] one relies on trained prediction models. Both require parameterization to work correctly in their deployed environment. This entails setting thresholds and training a model. Both tasks can be challenging in the face of unknown or changing environments. Only relying on monitoring data constitutes a reactive adaptation process by optimizing the parameters using historical data. This is problematic because they can only adapt after changes have happened, which deteriorates the quality of service. To avoid this issue, we propose a self-adaptive platform that uses a simulation to evaluate different future scenarios. Therefore, we identify and develop the required tools that work in tandem with Kubernetes and facilitate a self-adaptive platform that can tune strategies before deployment and during runtime.

**RQ3. What tools and methods are required to build a self-adaptive Serverless Edge Computing platform?**

Serverless Edge platforms are complex systems that include components to fully enable an automated application lifecycle, from uploading applications to autonomously managing them during runtime. However, due to the heterogeneous and dynamic nature of the edge-cloud continuum, the platform must adapt to the changing environment. This thesis focuses on the main orchestration mechanisms of function scheduling, function autoscaling, and request routing. Each of these mechanisms can be implemented using different optimization techniques. Therefore, tools that facilitate a self-adaptive platform must be able to support a variety of implementations and should not be tied to one. Specifically, orchestration mechanisms often influence each other (i.e., suboptimal scheduling decisions can lead to increasing autoscaling actions, etc.) and should not be optimized apart from each other. Simulations can facilitate such a platform by capturing major components of a Serverless Edge Computing platform as a co-simulation. This research question involves finding tools and methods to facilitate a self-adaptive platform. The tools and methods must support various orchestration strategies and be able to model the interactions between different ones (i.e., autoscaling and scheduling). Moreover, the tools must be compatible with real-world entities as it is important to transfer as much information as possible into the self-adaptation process.

**RQ4. How can we tune orchestration strategy parameters during runtime and bootstrap them ahead of deployment? How can simulation facilitate a self-adaptive Serverless Edge Computing platform?**

In **RQ2**, we have devised two orchestration strategies based on threshold-driven optimization and a prediction model. These two represent a large class of orchestration strategies found in related work [RND23]. For example, in literature, numerous optimization problem formulations, such as Linear Programming (LP), have been explored to solve the

scheduling problem [RSK23, ATCG22, RRD21]. While Rausch et al. [RRD21] optimize the weights of several objective functions used for scheduling, Rastegar et al. [RSK23] formulate an LP problem whose solution represents an optimal application placement. Other approaches can use prediction techniques, such as AI or ML, to implement the orchestration mechanisms. Reinforcement Learning (RL) has been used to implement intelligent agents that interact with the world by deciding where to place applications [RNC19]. On the other hand, AI-based solutions can also use neural networks to predict values of interest to base a decision on [RRD+24, MPV+23, TGX+22]. Tuli et al. [TGX+22] train a network to predict the Quality of Service given an application mapping and can use it to decide which mapping is best.

Based on this, we want to explore which orchestration strategies can benefit from a simulator and how the approach differs between optimization- and prediction-based implementations. For example, in an optimization-based implementation, the simulator can optimize parameters and send the output back to the real-world system. The simulator can generate synthetic datasets for a prediction-based orchestration strategy to create more robust models.

Another challenge is that the simulation must adapt the orchestration strategies to different edge-cloud infrastructures with varying computational and network heterogeneity and client request pattern heterogeneity. The edge-cloud continuum is a heterogeneous fabric of computing resources; therefore, platforms must adapt their orchestration strategies accordingly. Rausch et al. [RRD21] show that for their optimization-based solution, the infrastructure significantly impacts the scheduler's configuration to achieve good results. For example, the scheduler's configuration for a cloud data center infrastructure differs significantly from an edge-cloud infrastructure. The same holds for prediction-based approaches, where the prediction model is trained on an infrastructure different from the real world. This can cause Quality of Service degradation, and the system might require time to learn the new environment.

Therefore, we want to explore a self-adaptive Serverless Edge Computing platform using a simulation to optimize orchestration strategy parameters during runtime and ahead of deployment. Exploring approaches that work before deployment can increase the Quality of Service by optimizing the strategies for new applications and infrastructures ahead of deployment.

## 1.3 Methodology

In order to answer our research questions, we follow a typical systems and operations research methodology based on experimental analysis and quantitative methods. In the following, we outline our methodology to answer the research questions.

1. We devised a comprehensive use case-based analysis of EI application requirements and EI platforms. The requirements revolve around various aspects, such as

performance, security, and application characteristics (e.g., how much data sensors produce). Based on that, we outline a technical vision of such a platform for EI and conclude that Serverless Edge Computing is a promising candidate. As the next step, we investigate current literature and projects around Serverless Edge Computing and introduce criteria that evaluate the maturity of these platforms. Combined, the EI requirements analysis and the criteria for Serverless Edge Computing build a requirements model.

2. We implement and evaluate mobility- and energy-aware orchestration strategies. To this end, we implement two approaches that work in tandem with Kubernetes and evaluate them on a real-world testbed. The evaluation consists of the analysis and discussion of relevant key performance indicators. These include application response time, resource usage, and network traffic. Based on thresholds, the first approach includes all orchestration strategies, scheduling, autoscaling, and request routing, and focuses on an edge-cloud setup using low-latency applications, similar to AI inference. The second is an AI-based energy-aware scheduling approach at the local data center level to minimize energy consumption. In both cases, the empirical analysis of each approach's impact on the testbed and the analysis of the respective methods are crucial. Our mobility- and energy-aware approaches showcase how thresholds and prediction models can be used for decision-making. At the same time, they highlight the need for self-adaptive platforms that find optimal parameters during runtime and pre-train the prediction model for new environments.

3. We build the foundation for our adaptive Serverless Edge Computing platform based on an edge-cloud testing framework [RRP+22] and Serverless Edge Computing simulator [RRFD23]. The simulator builds the core tool for our self-adaptive platform. The testing framework contains a workload generator for developers and a set of EI applications to test their orchestration strategies. The simulator has been developed and tested in multiple works, showing high adaptability towards new requirements, which is important to build a simulation-driven Serverless Edge Computing platform due to variable factors, such as the actual implementation of orchestration strategies or the infrastructure. The goal is to unify both to build a self-adaptive platform that uses the simulation to drive the optimization of the orchestration strategies. To this end, we propose function orchestration strategies that the platform optimizes during runtime and before deployment.

4. We implement a threshold- and a prediction-based orchestration strategy with which we tune our self-adaptive platform. The platform tunes the threshold-based strategy during runtime, while the platform generates synthetic datasets on which we can train the prediction model before deploying the strategy. We evaluate different aspects of our adaptive Serverless Edge Computing platform. The methodology consists of empirical systems research, consisting of running experiments to determine the platform's ability to create robust prediction models and adapt parameters during runtime.

## 1.4 Contributions

This thesis provides contributions to the field of edge-cloud systems with a focus on orchestration applications in Serverless Edge Computing from the viewpoint of EI applications. Novel contributions are made regarding orchestration strategies, testing and simulation frameworks, and a self-adaptive platform capable of optimizing deployed orchestration strategies by tuning threshold parameters and enhancing prediction models through synthetic dataset generation. In the following we introduce four contributions (i.e., **C1**, **C2**, **C3** & **C4**), whereas **C1** maps to *RQ1*, **C2** to *RQ2*, **C3** to *RQ3* and **C4** to *RQ4*.

**C1. Edge Intelligence & Serverless Edge Computing Requirements Model** Chapter 2 analyzes Edge Intelligence based on various use cases and proposes platform requirements. The requirements revolve around the need for autonomous application management that alleviates platform clients. Customers should only upload their application code and hand over the application orchestration to the platform. Serverless Edge Computing, an extension of the cloud paradigm Serverless Computing [NRF+22, RND23], appears as a promising solution to the orchestration challenges of Edge Intelligence. Therefore, we analyze over 45 offerings from academia and public open source and commercial offerings from various perspectives that hinder adoption. These two chapters present a requirements model for Edge Intelligence applications and the Serverless Edge Computing platform paradigm. The requirements and challenges Edge Intelligence and Serverless Edge Computing pose are manifold. However, we address two: mobility- and energy-awareness in **C2.**

**C2. Mobility- and energy-aware orchestration mechanisms for the Edge-Cloud Continuum** Building orchestration mechanisms focusing on mobility and energy is fundamental for Serverless Edge Computing and EI. We split the work into two orchestration strategies. The first focuses on the mobility aspect in edge-cloud scenarios, while the second focuses on energy-awareness in cloud systems.

- **C2.1 Mobility-aware Serverless Edge Computing** We have implemented orchestration mechanisms to mitigate the issue of mobile users that generate workload in different geographical regions [RRD+22]. The approach was built on our novel *pressure* concept, which aggregates metrics (such as CPU usage, response times, etc.) at the cluster level, which a global autoscaler uses to scale applications based on pre-defined thresholds. The metric aggregation significantly reduces the communication between remote edge clusters and the cloud. The pressure metric allows the global scheduler to determine the direction of requests and take scaling actions to address the incoming workload of moving users. Results from our evaluation show an increase in performance and a reduction in network transfer.
- **C2.2 Energy-aware Container Scheduling** Energy plays an important role in resource-constrained environments (such as Orbital Edge Computing [DL19])

and large-scale data centers, which produce large amounts of carbon emissions and increasingly more through the popularity of AI and hardware accelerators. Therefore, we investigated scheduling mechanisms that can reduce the overall energy consumption in an HPC setting [RRD+24]. The container scheduler uses a Graph Neural Network (GNN) to predict the energy consumption of devices. Similarly to simulations, the GNN allows us to perform *what-if* scenarios, which the scheduler facilitates to choose the device with the lowest predicted energy consumption.

To summarize, **C2.1** tackles the issue of placing resources along the user's movement to reduce network overhead and improve end-to-end latency. This is especially prevalent in Smart Cities and focuses on low-latency applications, such as AI inference. **C2.1** introduces a decentralized platform, consisting of a centralized controller and decentralized local schedulers, that benefits from its *pressure* metric that aggregates low-level metrics that can reduce network overhead caused by monitoring, as the central control mechanism relies on this value. **C2.2** investigates energy-aware scheduling at a local cluster level using a GNN. The GNN model presented in this thesis only captures one host, but it can include multiple hosts and possibly the complete cluster at once. Thus, **C2.1** and **C2.2** complement each other as **C2.2** can be integrated into the platform presented in **C2.1**. However, both contributions have limitations. **C2.1** fails to address the challenge of heterogeneous devices when scheduling; this is crucial for the edge-cloud continuum, as application performance can significantly degrade through bad scheduling decisions. **C2.2**'s solution is fine-grained and requires extensive monitoring data. This can introduce additional network overhead and negatively impact co-running applications. Moreover, in our experiments, we conducted preliminary results to gather a dataset that contains the energy consumption across different types of applications. While the model is application-agnostic (i.e., only includes system-level resources), it must constantly learn if new applications or hardware are deployed.

Moreover, contributions **C2.1** and **C2.2** propose novel orchestration strategies that utilize thresholds and predictions to enable mobility- and energy-aware strategies. However, both rely on the challenging task of finding the right parameters. In **C2.1**, the optimal thresholds must be found to guide scaling actions, while in **C2.2**, a GNN must be trained, which relies on historical monitoring data.

Therefore, we aim to build a self-adaptive platform to bootstrap parameters and determine them during runtime. **C3** builds the foundation of this platform, and **C4** introduces optimization-based approaches that find thresholds during runtime and generate synthetic datasets to bootstrap the prediction model.

**C3. Tools for a self-adaptive Serverless Edge Computing platform** This contribution revolves around building tools that enable a self-adaptive platform. Our idea is to use co-simulation to find parameters, which requires an end-to-end framework of monitoring infrastructure, transmitting the data, and feeding it into a compatible

simulation. To this end, we propose a serverless experimentation framework [RRP+22] that can be integrated with our serverless simulator [RRFD23].

- **C3.1 Serverless Experimentation Framework** The framework is based on Galileo [RRPD19], a distributed load testing framework, which we extend by adding orchestration capabilities that can be synchronized with the simulator, and offers a streamlined approach to evaluate orchestration mechanisms [RRP+22]. This includes convenience functions to analyze experiments and help write new orchestration strategies. A unified API used by this framework and the simulator avoids rewriting them to a certain extent and promotes reproducibility on a real-world testbed and the simulator.
- **C3.2 Serverless Edge Simulator** We also present a simulator [RRFD23] based on a realistic domain model using a trace-driven resource model. This means we can inject traces (i.e., monitoring data) from the real world into the simulator. Using this simulator and the testing framework, we can build a self-adaptive Serverless Edge Computing platform that uses simulations.

Contributions **C3.1** and **C3.2** build a foundation with which we explore two approaches for self-adaptive platforms.

**C4. Simulation-driven and self-adaptive Serverless Edge Function Autoscaling** The final stage of this thesis is to explore strategies that build the core of a self-adaptive Serverless Edge Computing platform. Specifically, we focus on simulation-driven approaches and on finding autoscaler thresholds and bootstrapping prediction models for autoscaling using synthesized datasets. To this end, we propose a strategy that finds thresholds during runtime (**C4.1**) and that helps bootstrap prediction models by synthesizing datasets (**C4.2**).

- **C4.1 Self-Adapting Autoscaler Thresholds during Runtime:** We implement a threshold-based autoscaling strategy to find optimal parameters during runtime. As this is a proof-of-concept prototype, the evaluation will focus on a small-scale testbed consisting of tens of nodes. We implement an end-to-end prototype that unites Kubernetes, our testing framework [RRP+22], and the simulation [RRFD23]. The prototype constantly feeds monitoring data into the co-simulation, replicating the real-world state. An optimization approach is deployed to find new threshold values using the simulation. We conduct experiments by sending workload to the testbed, optimizing autoscaling thresholds during runtime, and measuring key performance indicators around performance and resource usage. The evaluation results show low SLO violations and resource usage.
- **C4.2 Bootstrapping Autoscaling Prediction Models through Synthesizing Datasets:** We introduce a prediction-based autoscaling orchestration strategy

and generate synthetic datasets using the simulation. This entails running an optimization process to find simulation scenarios and generate datasets for possible infrastructure settings. The approach is evaluated against the baseline approach of relying on historical data and related work.

To summarize the contributions and provide an overview, **C4.1** introduces a novel strategy to tune autoscaling parameters during runtime. In contrast, **C4.2** focuses on generating synthetic datasets to train models that predict the needed resources to satisfy workloads. Both contributions complement each other, as runtime tuning and bootstrapping models enable a self-adaptive platform with different orchestration strategies. The strength of **C4.1** lies in the fast optimization of parameters and the ability to replicate the real-world platform. **C4.2** shows that simulations can speed up the training process with minimal historical data from the real-world platform. However, both contributions have limitations, especially in their focus on simple autoscaling approaches and assumptions. Specifically, **C4.1** optimizes the parameters of a simple threshold-based autoscaling algorithm that resembles the autoscaling process of the commonly used container orchestration service Kubernetes. However, results indicate that our approach could struggle with more complex orchestration strategies that require more parameters, i.e., our strategy required one parameter per cluster and per function. A limitation of **C4.2** is the assumption of having an accurate workload prediction model. As the autoscaling strategy depends on the predicted workload, underestimating or overestimating the incoming workload can lead to drastic changes in results, and thus, performance, energy, and resource consumption.

**C4** presents the foundation of a self-adaptive platform using simulations, but has focused only on autoscaling. This thesis has explored its applicability in tuning and aiding autoscaling approaches. Consequently, other orchestration strategies, such as scheduling, have yet to be explored using this approach.

## 1.5 Thesis Structure

The thesis is structured as follows: Chapter 2 motivates Edge Intelligence applications and outlines how service platforms can facilitate their adoption across the general public. Serverless Edge Computing is a promising platform paradigm that can facilitate this vision. Therefore, we examine EI based on a use case study and Serverless Edge Computing based on criteria, and evaluate the current state of Serverless Edge platforms in the literature and open source and commercial offerings, which build our requirements model. Based on the previously determined challenges, we focus on mobility- and energy-awareness challenges and propose two novel approaches to tackle them. Chapter 3 proposes a threshold-based mobility-aware orchestration strategy and an AI-based energy-aware scheduling approach. However, finding thresholds and training prediction models is challenging. Therefore, we aim to develop a self-adaptive platform to solve this task. Chapter 4 proposes an end-to-end testing framework for edge-cloud resource management

and a simulator that builds the core for this self-adaptive platform. Chapter 5 implements and evaluates a self-adaptive Serverless Edge Computing platform using co-simulation. The platform can generate synthetic datasets to train prediction-based orchestration strategies ahead of deployment and find thresholds during runtime. Chapter 6 concludes the thesis and provides topics for future work.

CHAPTER 2

# Edge Intelligence & Serverless Edge Computing Requirements Model

This chapter is dedicated to analyzing Edge Intelligence and Serverless Edge Computing. First, Section 2.1 proposes a vision of an Edge Intelligence as a Service platform with its requirements based on a use case study. The requirements revolve around EI applications, their data types, application domains, and elastic requirements that platforms need to address. Second, Section 2.2 introduces various criteria for Serverless Edge Computing and reviews approaches from academia, the open source community, and commercial offerings. The criteria are split into design time and runtime, and focus on emerging applications that profit from the edge-cloud continuum and its heterogeneity (i.e., hardware accelerators), such as EI.

Section 2.1 is based on [RD21], and Section 2.2 is based on [RND23].

## 2.1 Edge Intelligence as a Service

This section outlines our vision of Edge Intelligence by analyzing various use cases and outlining a technical implementation. Besides identifying use cases and their requirements, we take a look at what an Edge Intelligence as a Service platform would consist of.

### 2.1.1 Introduction

We have seen the rise of highly capable AI in recent years, powered by computational capabilities of hardware accelerators, and the consequential research of neural network architectures. AI starts to infiltrate our daily life, covering all important aspects (i.e.,

health, cities, agriculture). Applications range from personalized food recommendations [YHY+17], real-time video analytics [ABB+17], and irrigation scheduling predictions [CTCL19]. Important characteristics are: context-awareness, ultra-low-latency, large amounts of sensor data, and confidentiality. The emerging Edge Intelligence paradigm fulfills all the conditions by extending the currently popular cloud-centric infrastructure to the edge of the network. Edge Intelligence is meant to build a unified platform for *AI on edge* applications with yet unseen capabilities by intelligently leveraging resources in near proximity to users [DZF+20]. To train models with context in mind, we need to deploy applications where the data originates from. Zhou et al. [ZCL+19] define six levels of Edge Intelligence characterizing where training and inference are happening. While *Level 1* considers a central training in cloud and using resources at the edge for inference, *Level 6* imagines all tasks to happen on user devices, which reduces the amount of data offloading to a minimum and guarantees confidentiality of personal data. In contrast to *AI on edge*, *AI for edge* focuses on developing strategies for platforms to intelligently manage applications as intended by users [DZF+20]. Intentions can have different forms and are not limited to guaranteeing service metrics (i.e., response time) but extend to providing relevant data and context-aware execution. Therefore, Edge Intelligence as a Service platforms (EIaaS) need to allow developers not only to specify service metrics but also to add contextual information, such that intelligent strategies transparently manage deployments.

In general, the platform can be envisioned by deploying a sensing and compute fabric [RD19]. The former represents the abundance of deployed sensors (i.e., IoT) while the latter acts on the emitted data. The computing fabric is based on the Edge Computing (EC) paradigm, which pushes centralized resources from the cloud to the edge of the network [Sat17]. By pushing resources closer to the users, we can mitigate the issue of the increasing bandwidth needs and process information and requests immediately. Unfortunately, unleashing the promised potential of Edge Computing is hard and stems from several factors: heterogeneous compute units, offering different capabilities, network distance playing a crucial role to guarantee the stringent latency requirements, yet unknown infrastructure governance, legal policies, context-awareness etc. [SD16]. To overcome the burden of managing edge infrastructures, accessible platforms are necessary to let people make full use of Edge Intelligence, and therefore allow us to enter the new stage in the cyber-human lifecycle [RD19, MG19]. Transparent deployment guarantees access to AI for businesses without dedicated resources for developing custom solutions and creates an opportunity to generate a wealth of diverse and novel AI applications.

Therefore, the goal of this chapter is to envision an end to end platform that pushes the democratization of AI further, by giving non-experts the opportunity to build EI applications. We discuss in detail a motivating use case, centered around a business that produces food and is located in Smart City, Health, Agriculture, and Food Computing domains. The use case showcases elastic properties, data sharing across domain boundaries, and helps us outline requirements for an EIaaS platform. Besides showing requirements, it also depicts the possibilities for businesses to leverage AI in a mean-

ingful way to improve quality for themselves, but also for customers. We identify four main tasks encountered when deploying AI models and highlight for each of them the problems that arise in EI and need to be tackled. Based on the previous investigation of use cases, application requirements, and tasks, we outline the general architecture and user interaction possibilities. The proposed platform builds on the industry-proven MLOps concept [Amaa, Goo], and extends it to enable the lifecycle management of EI applications. Additionally, we discuss possible implementations of this platform and identify important requirements that the underlying orchestration services must fulfill.

The rest of the chapter is structured as follows: in Section 2.1.2 we introduce the necessary background and relevant concepts, accompanied by related work. Section 2.1.3 is dedicated to portraying our use case study by highlighting applications, needed data sources, domains, and elasticity requirements. Upon analysis of possible applications, we describe in Section 2.1.4 the four main tasks in AI and draw attention to problems that we will inevitably encounter in the future. Based on this, we can envision a plausible platform and afterwards give details to important technical problems encountered in Edge Intelligence in Section 2.1.5. We outline limitations and future work in Section 2.1.9 and conclude the work in Section 2.1.10.

### 2.1.2 Background & Related Work

Edge Intelligence uses data produced at the edge of the network by applying AI applications on it. Edge Computing infrastructures help us solve various issues that arise (i.e., privacy concerns) in EI by pushing resources to the edge of the network. Zhou et al. [ZCL+19] state in their survey that the term EI is not yet fully defined, but one common usage is to refer to the execution of AI models at the edge.

Though they define it as the efficient use of data in an edge-cloud cooperative manner, where inference and training can happen across all devices. We also believe in that vision and showcase in our use case that data from different domains can be used to create new and intelligent applications. Zhou et al. [ZCL+19] also introduce a six-level rating that specifies where applications are executed. We briefly highlight the main differences, and advise to read a detailed explanation in [ZCL+19]:

- Cloud Intelligence: Training and inference on the cloud
- *Level 1* Cloud-edge co-inference:

  Training: Cloud - Inference: Edge-Cloud
- *Level 2* In-edge co-inference:

  Training: Cloud - Inference: Edge
- *Level 3* On-device inference:

  Training: Cloud - Inference: Device (no offloading)

- *Level 4* Cloud-edge co-training:
  Training: Edge-Cloud - Inference: Edge-Cloud
- *Level 5* All in-edge:
  Training: Edge Inference: Edge
- *Level 6* All on-device (no offloading):
  Training: Device - Inference: Device

Because EC contains resource-constrained devices, some tasks must be offloaded to other nodes. Other surveys have comprehensively shown open issues we encounter in EI [DZF+20, WHL+20]. Similar studies have been published that investigated the complete food supply chain. Pang et al. [PCHZ15] showcase how IoT can influence food supply chains. They focus on conducting surveys on the importance of IoT applications with experts (i.e., shelf life prediction) and describe in detail the technical realization of those sensor networks, including sensor data fusion. In [RHM+19], the authors present a serverless platform for Edge Intelligence applications. Their focus lies in discussing the technical details but also introduce the problem that arises from context-aware systems. The work is complementary to ours, as they propose a programming model and low level details about the execution platform. Zhang et al. [ZWL+19] present an open framework that focuses on the technical implementation of EI applications at the edge. An important issue is the large size of AI models and the resulting mismatch with resource-constrained devices. They propose a systematic description of EI algorithms, a RESTful API to expose EI applications and outline usage with four use cases. We look at Edge Intelligence from a higher level, by describing a plausible business, how it can benefit from EI, and that not only the computing infrastructure is heterogeneous. Further, our platform focuses on the democratization of AI, which will push development and use to a broader public. The work of Rausch et al. [RD19] resembles our approach but differs in that ours is based on a business case study, which highlights the importance of all these different applications to a single stakeholder.

### 2.1.3 Motivating Use Case

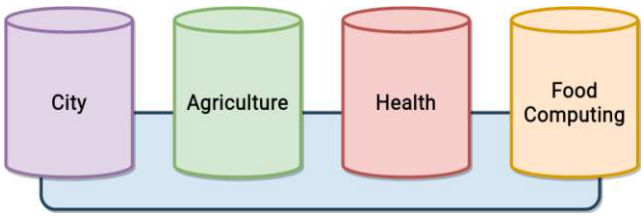



Figure 2.1: Crossing domain boundaries

Our motivating use case presents possible EI applications that can be developed and deployed with our proposed end to end platform. The use cases are settled in the context

of Smart Cities [ABD15], Smart Agriculture [RC18], Smart Retail [Dac17], Smart Health [SPC+14] and Food Computing [MJJ19], depicted in Figure 2.1. We aim to showcase the potential of EI to help a business that produces vegetables. Each application is briefly described by focusing on the necessary data, the possible EI levels, which domains are involved, and which elasticity requirements are important. Table 2.1 summarizes our findings.

Consider an international company, set up in different countries, that (1) operates a commercial agricultural business and (2) sells the products in stores. Whereas products can originate from the local area or foreign countries, and the business is in direct contact with the farmers. The main revenue stems from selling food to individuals in shops with accompanying restaurants, making the business's goal to adapt its offering to the cities. We briefly highlight the business lifecycle, and afterwards showcase useful applications that can be deployed on an EIaaS platform. We consider a simplified process that is split into two parts: farm and shop. Farmers want to efficiently plant seeds, take care of plants, and yield high-quality crops. The shop gets them delivered, places them in-store where customers buy them, either the raw produce or as a dish served in the in-store restaurant. Based on the analysis of sales, businesses and farmers can adopt their strategies with the help of a Recommender System. Figure 2.2 shows each step and indicates the feedback loop. After outlining the business's internal process, we describe for each step plausible applications that could help reduce cost and improve quality.

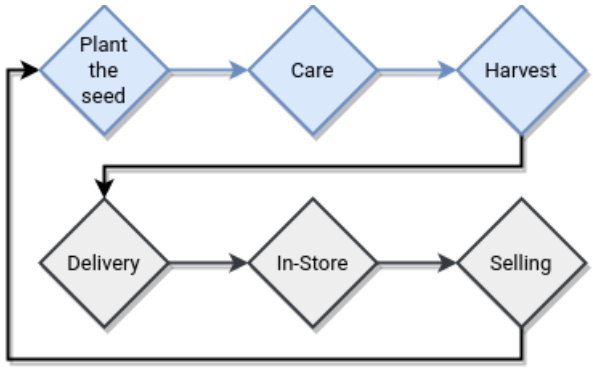



Figure 2.2: Steps in plant farming chain

#### 2.1.3.1 Farm

**Plant** First, farmers can use AI to map the soil to assess chemical characteristics. Aitkenhead et al. [ACT+13] trained a neural network that can predict soil characteristics to estimate fertility. Their approach appears optimal for cheap and rapid field assessment using colour values from digital photographs [ACT+13]. A crowd-sourced labeling process can prove to be efficient, and, therefore, inference and training at the edge are recommended. Further, elasticity requirements will target almost exclusively the

reduction of cost, as analysis does not have real-time requirements and only serves as a cheap assessment without focusing on high accuracy.

**Care** During the growing phase of plants, farmers can take advantage of automatic discrimination between crop and weed based on images [GRN13, ADM+03], plan irrigation with short-term weather forecast [CTCL19], and perform vision-based disease detection [MBW+04, MBO+05]. Crop and weed prediction has to be performed in real-time to allow tractor-mounted spraying equipment to efficiently target weed patches between crop rows. Both crop/weed discrimination and plant disease detection can profit from localized fine-tuning due to the large number of different species [BARGP11]. Climate and soil are vastly different across the globe, directly affecting watering cycles and fertilization, making fine-tuning at the edge a viable solution to guarantee high-accuracy results [CTCL19].

**Harvest** Multiple vision-based systems can be deployed to infer fruit size and quantity and help farmers during harvest to predict the quality [WWV17, HMD+18]. Yield prediction of winter wheat has proven to be sensitive towards location and input time window [HZC+20]. Han et al. [HZC+20] propose a model that predicts the yield of winter wheat in China. Their model uses remote sensing (i.e., vegetation index), climate (i.e., temperature), soil (i.e., soil properties), as well as historic yield data and crop maps. Results show that the local environment is highly relevant for the model, and bigger surroundings lead to worse accuracy. Halstead et al. [HMD+18] have stated that their system is trained with near-maturity peppers and fails to accurately predict the ripeness of juvenile fruits. Their two-stage approach of first detecting the fruits and afterwards estimating the ripeness shows that the second stage may have to be fine-tuned at the edge to guarantee high accuracy by using multiple models.

#### 2.1.3.2 City

**Transportation** After the harvest, produce must be safely transported to the shops and therefore can benefit from crowdsourced road condition assessment, smartphones measure the movement via accelerometer and gyroscope [AKAA17, SvdZD+15]. Not only does it lead to increased traffic safety, but it also avoids any unnecessary spoilage caused by continuous vibrations and high force shocks [PCHZ15]. In road sensing, the quality is important, as trips can be planned ahead and therefore do not require low latency. We think that a localized fine-tuning can yield higher accuracy due to differences in common vehicle characterizations (i.e., suspensions) that may lead to different accelerometer and gyroscope readings on the same road. Further, AR-based helper systems that receive results from nearby traffic cameras can display nearby traffic participants in real time [RHS+21]. The results contain bounding boxes of inferred objects in the camera's view. Locality-awareness is of utmost importance to guarantee relevant results and ultra-low latency, required to continuously provide the AR system with data. Consider that throughout the city, cameras are deployed, and all of them are running object detection

models that emit bounding boxes representing participants. Drivers want and need only those that are in near proximity; the system has to automatically adapt the transferred data based on the user's location. While accurate inference is important, performance must also be considered to make the application usable. Training can be done in the cloud using large datasets, but inference must happen at the edge due to latency requirements.

**In-Store** Upon delivery, produce is put on shelves or prepared as meals. Customers should be engaged and incentivized to buy products during their stay in the local shop. Due to the emergence of Virtual and Augmented Reality [RHS+21], we assume customers are wearing smart glasses to get additional information while strolling through to store.

Smart glasses record the user's view with up to 30 frames per second, which acts as input for an inference pipeline. This pipeline identifies objects, selects products, and classifies them [XHJ+15]. Based on the resulting class of food, additional information of the user's interest is shown. Information may include: detailed nutritional information, possible recipes, ecological impact (i.e., consider the transport of food from foreign countries), etc. The application is context-aware and can use personal data to provide relevant information. Multiple works have shown promising results in recognizing food and dishes [BTP+15, KY13, CNS16, KY15]. Xu et al. [XHJ+15] have added geo-location information to the model to better differentiate between dishes that look the same and stem from different cuisines. This approach seems plausible in the context of EI, where a general food recognition model can be trained in the cloud and fine-tuned accordingly in different restaurants and/or markets. Further, processing the captured video feed of smart wearables must adhere to stringent latency requirements and may even be subject to local policies protecting people's identities (i.e., GDPR). Processing can be done either on-device (i.e., smartphone) or nearby edge devices, and sophisticated anonymization software can act as pre-processing to obey laws [HML19]. Based on personal preferences, applications may recommend food during the customer's visit [ACG+16, BDADFM17]. Smart health wearables may provide applications with contextual data to react accurately to different situations. These applications process highly sensitive data (i.e., allergens, dietary, and vital data) and therefore, training on-device is required. This restriction can build important trust between the user and the application [ORLH17].

**Sale** After users make their choice, smart checkouts using deep learning based recognition software can help shops ease the workload on workers, but need to be accurate and fast at the same time [WTC+16]. This makes us believe that cloud-based training and edge inference are suitable. Context-aware recommender systems can learn user preferences based on previous purchases [CGK+20]. Privacy is important for this application type, and therefore, we recommend that confidential tasks be done on-device.

**Feedback** The last use case revolves around the idea of creating a Recommender System that can help the business and farmers to effectively recommend the next crop. We imagine that this system can build on nearly all previously mentioned data sources. For example, historical sales reports can highlight seasonal patterns, the local fields can

Table 2.1: Possible applications in the supply chain

| Step | Application | Data | EI Level | Domains | Elasticity |
|---|---|---|---|---|---|
| Plant the seed | Soil Mapping [HHZ+16] | Topographic<br>Climatic<br>Vegetative indices | Level 4 | Smart Agriculture | Quality, Context |
| Care | Crop/Weed discrimination [GRN13, ADM+03]<br>Irrigation Scheduling [CTCL19]<br>Disease Detection [MBW+04] | Weather<br>RGB, Spectral images | Level 1 & 5 | Climate<br>Smart Agriculture | Cost, Quality,<br>Privacy |
| Harvest | Fruit Size Estimation [WWV17],<br>Fruit Quantity Estimation [HMD+18] ,<br>Yield Prediction [HZC+20] | RGB(-d) images<br>Remote Sensing<br>Climate, Soil<br>Yield, Crop map | Level 4 | Smart Agriculture,<br>Climate,<br>Food Computing | Quality<br>Performance |
| Transportation | Traffic participant detection [RD19]<br>Road condition assessment [AKAA17, SvdZD+15] | Images, Orientation<br>Angular Velocity | Level 3 & 4 | Smart Traffic | Safety |
| In-Store | Food Classification [XHJ+15]<br>Food Recommendation [YHY+17, BDADFM17]<br>Face anonymization [HML19] | Images, Health<br>Personal preferences | Level 3 & 4 | Food computing,<br>Smart Health | Privacy, Quality,<br>Context, Performance |
| Sale | Intelligent checkout [WTC+16]<br>Personalized recommender [CGK+20] | Images, Location<br>Temporal, Consumer Behavior | Level 2 & 6 | Smart Retail | Privacy, Context<br>Performance |
| Feedback | Produce Recommendation | Sales, Social context<br>Location | Level 4 | Smart Business | Privacy, Quality<br>Context |

be analysed for soil mapping purposes, as well as weather data influences the feasibility of crops. Analysis of personal data (i.e., dietary choices, allergens, preferences, etc.) can foster the system with useful information regarding the overall consumption behaviour of cities, or even districts. Besides data about the citizens of Smart Cities, environmental data of the surroundings can further influence recommendations. In areas with high pollution, trees can help mitigate the issue and produce fresh oxygen. To the best of our knowledge, no work has been done in this direction that combines Smart City and Smart Agriculture to create a feedback loop between the production and consumption of vegetables.

We observe that the applications differ widely in terms of input data, elastic requirements, and EI levels. Data from the domains of Smart City, Agriculture, and Health, as well as Food Computing, were used. We process various types of data: climate, soil, high-dimensional (i.e., images), plant, health, and personal data. While some models may be trained in the cloud and are deployed at the edge for inference, others have to be aware of contextual information and, therefore, are fine-tuned at the edge. Privacy-related concerns make on-device inference and training necessary, as well as the reduction of performance and the conquering of large amounts of data. Elasticity requirements differ in terms of quality, performance, privacy, and should balance cost. Further, we identify issues that relate to the tasks we discuss in the next section: (1) addressability, (2) sources, (3) security, (4) coordination, (5) performance, and (6) context.

### 2.1.4 Tasks

In the following, we categorize operational tasks that our framework will focus on. Our opinionated view on application requirements allows us to define general as well as highly contextual goals.

We consider the four main tasks that appear in Edge Intelligence: sensing, preprocessing,

training, and inference. This classification is based on recent research that identified different layers and tasks [ZWL+19, RD19]. We highlight for each task different aspects and key requirements encountered in our use case study.

#### 2.1.4.1 Sensing

Our use cases show that physical sensors can be deployed anywhere and measure different types of data. Table 2.2 shows a few of them that can be used to achieve the presented applications. For each sensor, we also describe the collected data types and which domain they belong to. Additionally, other studies have investigated the heterogeneity in sensors and emitted data [SAP20, MJL+19, HBDQ+18, YSC+20], clearly showing the need for a unified interface. The heterogeneity stems from different types of velocity, volume, and ownership, as depicted in Table 2.2. Therefore, we think a layer of abstraction is necessary to make this data accessible. The layer should provide ways of selecting, merging, and aggregating data from different sources while being context-aware. Next, we highlight requirements that stem from our motivating use case.

**Addressability** The use cases made it clear that efficient querying for sensor data is necessary and needs to handle context. Nearby traffic participant detection is practically unusable without the knowledge of the users whereabouts. Users must be able to either send information to the system so that it only sends relevant resources or must employ a fine-grained topic subscription to enable these use cases. While in this application users can specify exactly which sensors are needed (i.e., using geospatial queries to select all cameras within a 100m radius), others require the system to understand much more semantically complex queries. Consider personal food recommendations that require highly contextual data to train an accurate model. How should developers approach this daunting task of selecting all sources, containing smartphone sensors, personal health records, etc.? Therefore, being able to easily address, select, merge, and process data from sensors is a very important task. Establishing trust with users (i.e., guaranteeing on-device processing) can be a key enabler for creating a homogeneous platform, providing access to sensors from external providers. For example, imagine we want to create a model that relates temperature and crop growth rate in the area surrounding Vienna. The system must combine temperature sensors with crop sensors in the vicinity. A system that fails to identify relationships between sensors will yield useless models. Therefore, sensors need to be tagged with context such that we can query and group them.

**Sources** Different types of sources have been proposed that can act as sensors. They are not restricted to be of physical nature (i.e., temperature sensor), but can also be virtual or logical [IS03]. All three groups are used throughout our use cases. Table 2.2 displays sensors that were used in the presented applications. Virtual sensors are characterized by gaining information from software. A smart traffic light, using cameras, can emit bounding boxes for detected objects. Other applications can use these bounding boxes, i.e., to display them in an AR headset [RHS+21]. Another plausible and important virtual

Table 2.2: Possible sensors and characteristics

| | Sensor | Data | Velocity | Ownership | Volume |
|---|---|---|---|---|---|
| Smart Agriculture | Crop Location | Tags the location of crops | Slow | Private | Low |
| | Temperature | Temperature of location | Normal | Public | Low |
| | Humidity | Humidity of location | Normal | Public | Low |
| | Camera | Identified objects on-premise | Ultra fast | Private | Low |
| | Spectral Camera | Fruit ripeness detection | Fast | Private | Normal |
| | Camera | Raw video feed | Ultra fast | Private | Very High |
| Smart Health & Food Computing | Smartphone | User location | Fast | Private | Low |
| | Smart glasses | User's Field of View | Ultra fast | Private | Very High |
| | User | Dietary | Slow | Private | Low |
| | User | Allergens | Slow | Private | Low |
| | User | Age | Slow | Private | Low |
| | User | Residence | Slow | Private | Low |
| | BAN | Activity Recognition | Fast | Private | Low |
| Smart City | Camera | Traffic Participants | Ultra fast | Semi-Public | Low |
| | Camera | Raw video feed | Ultra fast | Semi-Public | Very High |
| | Temperature | Outdoor temperature of area | Slow | Public | Low |
| | Barometer | Air pressure | Slow | Public | Low |

sensor is anomaly detection. Anomaly detection can be used in wireless sensor networks, where we want to be notified in case of sensor failure [CFG+19]. Logical ones combine different sensors to emit aggregated data. A personalized recommender may make use of the user's location (measured by a GPS sensor) and take previous shopping behavior into consideration. Therefore, the platform must support not only physical sensors but also be flexible regarding the definition of what constitutes a source. Additionally, we roughly estimate the velocity and volume for each sensor and the ownership. It is important to highlight that sensors are not only heterogeneous in terms of data type, but also other factors increase the complexity of data fusion.

#### 2.1.4.2 Preprocessing

After defining the sources, AI applications typically perform some preprocessing on the data, which builds the first step on the computing infrastructure. Preprocessing is done either for training purposes (i.e., creating batches), inference (i.e., image scaling), or for exploratory analysis (i.e., aggregation). Gravina et al. [GAGF17] argue that separate preprocessing can benefit application development because developers can focus on the functionality while skipping carefully selecting and filtering data from the sensor. Edge Computing provides the capability of preprocessing large amounts of data emitted by sensors in a timely and efficient manner (i.e., by placing applications near the origin). Through the ability of placing applications at the edge, platform providers can guarantee data security and privacy preservation by leveraging light-weight methods to protect data, identity, and location [ZCZ+18]. By combining local processing of sensitive data and pseudo-anonymization, platforms are able obey to local laws and make guarantees about safety for users. This requirement differs across application types, but the presented

applications process highly sensitive and personal data. In the following, we briefly discuss these security concerns.

**Security** Our use case study showcased multiple applications that may act on private data (i.e., personalized food recommendations). Smart Health applications fundamentally require personal data and therefore need to protect the user's privacy. It is important to act only after users give consent and are informed of the data being processed [MBPMS13, ORLH17]. For example, our proposed food recommendation application, which processes data about the user's surroundings, activity, dietary preferences, allergens, body weight, etc., will probably not be adopted by users if they do not trust the platform. A survey in the field of Smart Health, conducted by Li et al. [LWGS16], has shown that: users adopt healthcare wearable through a subjective risk-benefit assessment, whereas the perceived privacy risk is a combination of different factors (i.e., legislative protection), and the benefit is subject to perceived informativeness and functional congruence. Therefore, the platform and applications have to be designed with privacy-related issues in mind to be accepted and adopted by the general public.

#### 2.1.4.3 Training

Training at the edge has several advantages: enabling to learn on raw sensor data (i.e., no aggregation necessary due to possible on-device training), reduction of bandwidth pressure, and privacy preservation. But training is an expensive computational task and can require large amounts of computing resources. While embedded devices, equipped with hardware accelerators, are commercially available (i.e., Nvidia Jetson devices[1]), it remains a challenging task to fully utilize all devices. Difficulties stem from the storage and computationally demanding neural networks, which pose a problem with regard to resource-constrained devices. Further, privacy preservation is important for some use cases, and training must happen locally. Federated Learning tackles these issues and represents a distributed and privacy-preserving model training approach that can even be applied on resource-constrained devices (i.e., smartphones) [MMR+17]. Other works [DZF+20, WHL+20] have already highlighted several issues concerning Federated Learning (i.e., worker selection), and due to our focus on investigating context-awareness, we disregard these issues for a moment and focus on the coordination obstacles that are specific to enabling localized fine-tuned models.

**Coordination** Our use case has shown the potential of training multiple models based on the context of deployment. For example, personalized food recommendation depends heavily on the user and also have high security requirements to protect privacy. Food classification has also shown potential to benefit from context-aware fine tuning due to visual similarity of dishes [XHJ+15]. Further, coordination can also include the task of monitoring AI model performances and, in case of a decline, new training rounds have to be initiated. Model degradation can happen due to new data, which may require

[1] https://developer.nvidia.com/buy-jetson

re-training to perform better [GMCR04]. Therefore, the platform has to intelligently coordinate training not only based on context and possibly training of multiple models, but also has to monitor the performance. Fine-tuning may not only improve accuracy for specific tasks, but also training with unlabeled data can increase robustness of the model and reduce the surface of attack vectors [CRS+19]. Due to the abundance of available information, unlabeled data can be easily retrieved and strengthens the models.

#### 2.1.4.4 Inference

While training AI models requires sophisticated strategies to enable training on resource-constrained devices and worker selection. We argue that this task is not subject to stringent latency requirements that we encounter in inference applications. Our use case has shown applications that require ultra-low latency, and performance is of utmost importance. Besides performance, context-awareness can also introduce new issues. Consider our fine-tuned models that are trained with a specific context in mind. This raises the problem of choosing the right model depending on the users' context. Both issues are addressed in the following and conclude the section.

**Performance** Due to the high heterogeneity in terms of performance, Edge Computing infrastructures require intelligent scheduling mechanisms to reason about placements. Though performance differences are not the only issue, performance degradation caused by multi-tenancy is a serious issue that needs focus when developing an EIaaS platform. In general, multi-tenancy issues can arise in two ways: (1) access to exclusive resources, and (2) performance interference. We identify that some resources may be exclusive to one application, but in other cases, multi-tenancy situations may occur by placing multiple applications on one node. Multi-tenancy can already lead in cloud settings to performance issues [PLM+10], and we encounter the same problem on resource-constrained devices. The platform requires an intelligent scheduler that is aware of those issues and prevents them from happening. A problem that we also face when considering performance and resource-constrained devices is the mismatch between large AI models and resource-constrained devices [ZWL+19]. Techniques are offered to mitigate these issues (i.e., model splitting with computational offloading), but it remains open who will be responsible for this decision. In certain circumstances, some edge devices will be capable of running the full model, avoiding issues that come with offloading. Therefore, the platform (or user) has to decide which method is most suitable.

**Context** As already hinted in our use case study, fine-tuning models for specific contexts is favorable in contrast to having only a single all-knowing model. This stems from the fact that models may not generalize well, and training with unrelated data may lead to a decline in performance, and also use cases require multiple models due to the intrinsic properties of the targeted problem (i.e., similarity in dish pictures). Additionally, personalized models also have to be maintained and associated with their corresponding

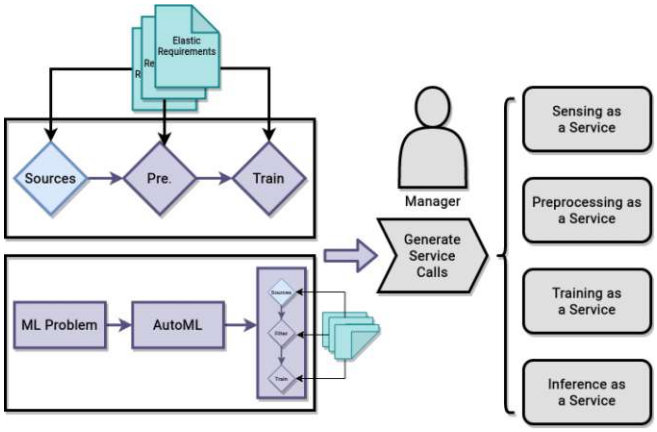



Figure 2.3: End-to-End platform

users. Therefore, users need to be able to specify their current context, such that the platform can choose the right model.

### 2.1.5 Platform

We now present our envisioned EIaaS platform that addresses the previously described challenges. It is opinionated towards Edge Intelligence applications and focuses on providing an accessible interface to push the democratization of AI further. While the proposal is based on the assumption of a working Edge Computing infrastructure, we acknowledge that many problems are yet unsolved and, therefore, showcase ideas from which general-purpose Edge Computing infrastructures can benefit. First, we depict the platform's architecture and the deployment methods on which we would build. Secondly, we present our vision of the platform's interface, how users interact with it, and can influence the application's behavior. Afterwards, we go deeper into technical details and define a plausible scheduler, based on a greedy multi-criteria decision algorithm, enforcing user-defined requirements.

### 2.1.6 Architecture

Operationalizing AI applications is an emerging topic and focuses on automating the lifecycle of AI models. Companies, like Google [Goo] and Amazon [Amaa, LKX+20], have released platforms that offer end-to-end development and lifecycle management for AI. Besides commercial offerings, these systems have also been investigated in open source and academic communities [HMR+19, BBC+17, ZCD+18, RHM+19]. The general approach, MLOps, is based on the DevOps paradigm [EGHS16] that automates the lifecycle of application development (i.e., packaging, testing, deploying, versioning, etc.). In the same manner, AI applications are managed using pipelines, which model the lifecycle of AI models as a directed acyclic graph. In addition to the discussed tasks in AI (i.e., training), continuous monitoring is important to identify concept drifts, which negatively impact a

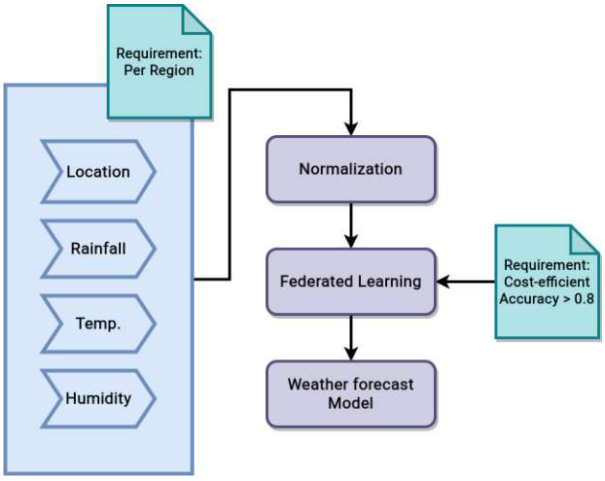



Figure 2.4: Training pipeline

model's accuracy [WHC+16]. MLOps platforms strive to automate the whole AI lifecycle and are therefore suitable to build the base of our platform. Pipelines can be declared using either configuration files [HMR+19] or in a UI [RND23]. The important thing is that the underlying platform receives a description of the pipeline and can compile this into actual service calls. This gives us the flexibility to take user-defined requirements into account.

### 2.1.7 Platform workflow

By assuming the underlying platform is similar to systems such as Google's VertexAI [Goo], KubeFlow [Bis19], or ModelOps [HMR+19], we describe now how users interact with the platform. Specifically, our focus lies in democratizing AI and giving users the ability to add requirements, as well as context. Figure 2.3 depicts a high-level interaction possibility. By leveraging AutoML techniques, users can define the underlying problem through selecting different sources that build the input and output space of the problem. AutoML techniques help democratize the usage and have gained notable traction over the years [RD19, HZC21]. While we think that AutoML can be beneficial, users should also be able to define pipelines manually. This description gets translated into a pipeline, to which users can add elastic requirements. Afterwards, the pipeline is transferred to the manager identity, who takes care of the continuous execution. Pipeline steps are translated into service calls, which are available for each task. Users can add to each step in the pipeline individual elastic requirements that influence the behavior of the application. The requirements add context to the operations and allow the system to reason about when it comes to operational strategies (i.e., placement strategy). Figures 2.4 & 2.5 showcase the definition of two pipelines. Figure 2.4 shows a training pipeline for a short-term weather forecast model [CTCL19]. In the first step, users have to define the data sources, which in this case are physical sensors that combine location and weather data. The associated requirement *Per Region* indicates the platform to group the sensor

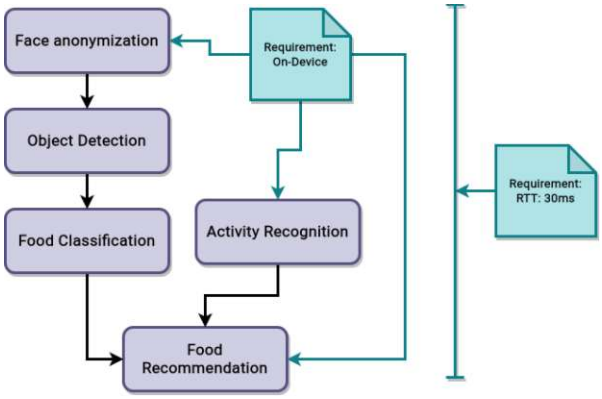



Figure 2.5: Inference pipeline

data and create multiple models. Afterwards, we preprocess the data and can then train on it. The model is either deployed automatically and/or stored for future applications. In our case, we specify the training to reach an accuracy of 80% These elastic requirements are directly related, as federated learning utilizes a worker concept to train data, which consequently has a big impact on the resulting cost. Executing this task will depend on many factors that all affect the cost and effectiveness. For example, the resulting accuracy will depend on the number of training rounds, which determines the cost of the training. The manager has to be aware of this relationship and must balance the execution between these two goals. Figure 2.5 shows a dedicated pipeline for inference consisting of different models that fulfill a certain task. We depict the use case of recommending food to users they see while wearing Smart Glasses. Recent work in optimizing this kind of pipeline [CSM+20] and the re-usability of models makes us believe that the separation is beneficial for all involved parties. Elastic requirements aid the deployment of offloading applications by stating which step has to be executed on-device. Efficient offloading is necessary to make model splitting techniques feasible. This knowledge can further be used of the underlying services to possibly place interdependent functions in near proximity, to avoid network latencies. Requirements include the processing of sensitive data on-device and impose a hard constraint towards the round-trip time (RTT). The RTT is picked according to the assumption of sending 30 pictures per second to create a smooth UX. Crankshaw et al. [CSM+20] present their system that takes tight tail latency constraints into account while considering different hardware accelerators, but it is situated in cloud computing.

### 2.1.8 Technical details

After presenting our vision to develop EI applications, we now turn to describing possible implementation strategies. We consider the necessary technologies to be different from the sensing task, in contrast to the others. Therefore, the platform uses a message-oriented publish/subscribe layer to send sensor data to processing applications. Subscribers can receive messages via topics from various nodes by specifying certain identifiers. Further,

dynamic message brokers have been proposed that can scale in a location-aware way, such that messages get routed over a nearby message broker to reduce latency [RND18]. Complementary to Rausch et al. [RHM+19], we envision the remaining services to be based on Serverless Edge Computing, whereas each step in the pipeline is implemented as a function [JSSS+19]. Therefore, containers include a single stateless application that exposes a single function over the network (usually HTTP). Kubernetes [Kub14], the underlying container orchestration service of Kubeflow, utilizes a greedy multi-criteria decision algorithm to decide which node is the most suitable one for a given function. Currently, the input of this algorithm consists of a given function and a node, which (1) filters infeasible nodes (i.e., node does not have the required accelerator) and (2) calculates a score based on an extensible list of scoring functions. This approach has been proven to work well on different infrastructures and AI applications [RRD21]. The scoring approach offers a highly flexible way of influencing scheduling decisions and can be arbitrarily complex. A limitation of the current Kubernetes ecosystem is that we cannot easily bundle multiple function containers that act equally but have small differences. Users should be able to upload multiple implementations of one function that are equal in functionality. Further, users can add information to each function implementation, such that the service can reason about them and intelligently select the optimal one. Consider our federated learning requirements, in which the user expresses cost-efficiency. While neural network applications greatly benefit from using GPUs, the operation may cost more than using a CPU. Therefore, in some cases, the scheduler may prefer the CPU computing platform. Through the combination of an intelligent manager and the underlying scoring functions, we can create a system that can handle EI applications. For example, locality-awareness is of utmost importance. Consider a platform that manages multiple smart city deployments. Increased usage in one city does not affect the deployed applications in another, and scaling must be able to identify this aspect and make the scheduling process aware of this. This problem can be solved by the following steps: (1) develop a scaler that can differentiate between the two cities, i.e., identify the need for more function instances in one city. (2) integrate an intermediary component to which the scaler and scheduler can talk. The scaler would need to put additional information concerning the location into this component. (3) scoring functions need to retrieve any additional info, and therefore can be location-aware. To the best of our knowledge, Kubernetes currently scales and schedules independently, and scheduling decisions are unaware of scaling reasons. If there is, in fact, a way to pass messages between these two components, our approach would still be plausible by removing the external component. The important takeaway is that the scoring approach is highly extensible to vastly different circumstances that we encounter in EI. Adding to the problem of location-awareness, load balancers have to be distributed and scaled with location in mind. Considering that ultra-low latency requirements can only be satisfied if the route to the cloud can be skipped.

### 2.1.9 Limitations & Future Work

#### 2.1.9.1 Theoretical approach

An important limitation of the proposal in this section is the theoretical approach. While we base our user case, the requirements, and the platform on extensive literature research, we have to mention that this work does not guarantee that the system will work as we described. Though with the current advances in all discussed topics and the existing research we base our system on, we are confident that such systems will exist at some point. The concept of context-aware systems has existed for decades [WHFG92], and with the current development of *Smart-** concepts, Edge Computing, and Edge Intelligence, these ideas may soon become a reality.

#### 2.1.9.2 Centralized vs. Decentralized

Already briefly discussed in Section 2.1.8, service components may need to be decentralized. We consider the load balancer to be the first component that has to be decentralized. Otherwise, we can not fulfill ultra-low latency requirements. It is unclear if other components have to be decentralized too, i.e., scheduler and scaler. Rausch et al. [RHM+19] have investigated the throughput of the Kubernetes scheduler and concluded that a monolithic approach is not feasible. While they do advocate the use of disaggregated schedulers, a decentralized one can also aid scalability, considering geo-distributed deployments. Caveats of decentralization are not only related to the management overhead that comes with such systems, but also security issues that arise. Blockchain technologies can help mitigate these [NPDS20].

#### 2.1.9.3 Governance

Currently, it is not clear who will own Edge Computing infrastructures and how access will be managed [RD19]. This issue is especially concerning for the sensing task. Questions about data ownership will have to be answered, and whether publish/subscribe systems are enough in the face of multiple parties involved (i.e., government, businesses). Therefore, our platform is based on the Sensing as a Service [PZCG14, ZIH+13] concept and leaves the technical implementation open for now, while allowing us to formulate requirements.

### 2.1.10 Summary

Holistic Edge Intelligence as a Service platforms are yet subject to exploration and research. Currently, we are facing the emergence of new paradigms that range from infrastructure (Edge Computing), application development (MLOps), to applications (Edge Intelligence). The paradigms move resources and applications towards the edge of the network and solve problems related to the increasing amount of data produced, the development of ultra-low latency applications, privacy concerns, and democratization. Edge Computing can be seen as a geo-distributed cluster with high heterogeneity, devices range from Raspberry Pi to fully-fledged cloud servers. MLOps promises to open up the

space for developing robust AI applications for the broader public by declaring pipelines and using AutoML techniques. Edge Intelligence is currently being investigated, and its limits are unclear. Our work presents a cross-domain reaching use case study that defines different applications settled in vastly different domains in great detail. We combine Smart City, Health, Agriculture, and Food Computing to showcase possible intersecting use cases that use heterogeneous sensors. Different elastic requirements are outlined, which must be defined for each application, and additionally, we identify their EI level. After explaining the key characteristics of each application, we describe the four main tasks encountered in Edge Intelligence and identify challenges that have to be tackled to make full use of its potential. Our proposed platform builds on state-of-the-art MLOps systems and tackles each identified requirement. We highlight the platform workflow with two different applications and show that, due to the nature of AI pipelines, our system can reach a broad audience and therefore push the democratization of AI forward.

We will investigate Serverless Edge Computing in detail based on our vision for a technical implementation that uses Kubernetes as the underlying orchestration platform. Serverless Edge Computing offers autonomous application orchestration following user goals, such as the ones we have seen in this section. Furthermore, the main focus of this thesis is the investigation of platforms that can build the foundation of an Edge Intelligence as a Service platform and look into the orchestration of functions, rather than on AI applications on the Edge. However, before proposing our own solutions, such as a mobility-aware platform, we analyze current Serverless Edge offerings in the following chapter and discern criteria that act as requirements model.

## 2.2 Serverless Edge Computing - Where we are and what lies ahead

In this section, we are going to investigate the maturity of current Serverless Edge Computing for Edge Intelligence applications. We measure maturity in a multitude of categories that span the complete function lifecycle, from building to running and orchestrating them. This section serves as a literature review and presents a requirement model and analysis of what current platforms are missing. Based on that, we implement concrete orchestration strategies that Serverless Edge platforms need for Edge Intelligence.

### 2.2.1 Introducing Serverless Edge Computing for Edge Intelligence

The edge-cloud continuum is a heterogeneous infrastructure of computing resources ranging from handheld devices to server-grade hardware. The combination of emerging application paradigms, such as Edge Intelligence [DZF+20], with increasingly complex requirements and the heterogeneous infrastructure, poses new challenges for managing, scaling, and deploying applications. Current cloud-centric platforms are not specifically designed for these heterogeneous environments. Serverless Edge Computing promises to efficiently use resources and reduce costs by abstracting the infrastructure and application management (e.g., scaling) away from users.[ATC+21, SKM22] Unfortunately, current

Serverless Edge Computing approaches still face numerous challenges to unlock the full potential of the edge-cloud continuum [ATC+21].

To gain a deeper understanding of serverless platforms, we review the current state of Serverless Edge Computing. This review is important to see where the serverless community, including commercial providers, open source projects, and academia, currently is and which challenges remain. We present a set of criteria that define maturity levels to systematically compare and evaluate the maturity of current serverless approaches to achieve the vision of a unified serverless computing fabric (SCF) for edge-cloud continuum [NRF+22]. The SCF unites storage and compute resources in the edge-cloud continuum and enables new application paradigms [NRF+22].

One of those is Edge Intelligence, which promises to enable new use cases (e.g., Mobile Augmented Reality and autonomous vehicles) by distributing AI inference and training across the continuum.[DZF+20] We identify three main challenges that the SCF must address. (1) EI applications are complex and span across the continuum. They range from short-running inference to long-running and data-intensive training tasks. Developing these applications is challenging, and the SCF must provide proper programming support to simplify the development process. (2) Mission-critical EI applications (e.g., emergency rescue) and long-running and data-intensive tasks (e.g., training) require reliable execution to guarantee service and save costs. It is challenging to implement and enable these applications in the dynamic and heterogeneous environment of the edge-cloud continuum. (3) EI applications run on highly heterogeneous hardware, and workloads can highly fluctuate (e.g., a public event that draws a large crowd). The SCF must address the challenge of providing optimal performance under these conditions. Based on these challenges, our evaluation focuses on evaluating the maturity of: (1) Programming Support, (2) Reliability Engineering, and (3) Performance Engineering.

### 2.2.2 Maturity levels of Serverless Edge Computing

We define maturity levels to evaluate serverless platforms and highlight key aspects for Serverless Edge Computing platforms. Our criteria are split into *design time* and *run time*, see Figure 2.6. The *design time* deals with tasks that happen before deploying the function. The *run time* focuses on challenges that happen after the deployment.

#### 2.2.2.1 Design time criteria

**Application State & Data management** considers design time mechanisms to support developers in writing stateful functions or accessing external storage (e.g., object storage). We introduce a three-level maturity rating system that evaluates if the programming model or platform offers (M1) *any functionality*, such as checkpointing, to solve data-related issues (e.g., statefulness), (M2) incorporates data-management as a *first-class citizen* (e.g., by introducing suitable API abstractions to facilitate Application State & Data Management), and (M3) offers capabilities to handle data management transparently to developers (e.g., *failure handling*).

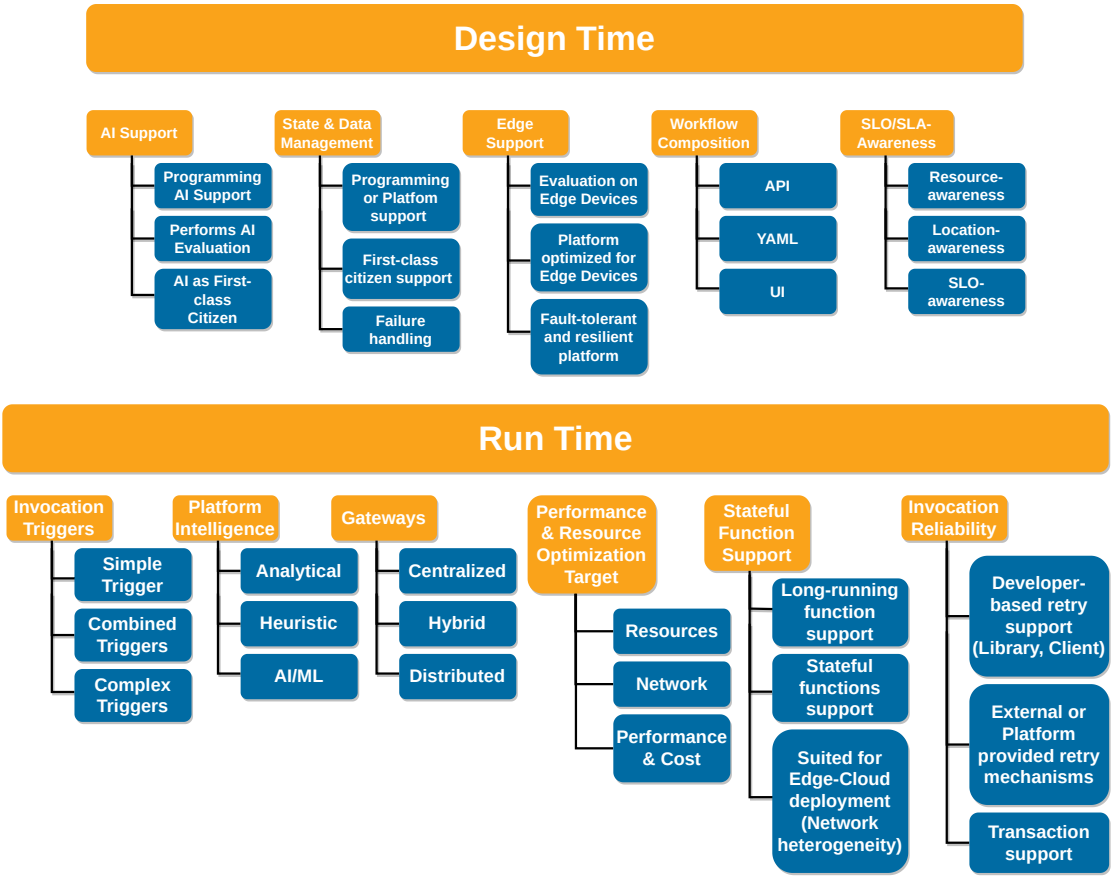



Figure 2.6: Design time and run time maturity levels

**SLO/SLA-awareness** allows developers to specify requirements and constraints during design time. The heterogeneous infrastructure and stringent application requirements make it necessary that developers are able to express requirements.[NPM+21, PNM+22] We define a three-level maturity rating that ranks different types of SLOs and SLAs based on the complexity of the requirement. (M1) specifies the *Resource-awareness*, whether resource consumption is considered and optimized. Approaches that reduce resource consumption (i.e., a lightweight runtime, reduced network traffic, etc.) or offer resource-oriented SLOs fulfill this level. (M2) specifies the *Location-awareness*, whether developers can express location requirements. (M3) specifies the *SLO-awareness*, whether developers can influence the execution of functions. Serverless platforms often allow developers to specify the number of available cores or memory, but we think the SCF has to offer, in addition, more sophisticated features (e.g., latency requirements [RRD+22]).

**AI support** We define three levels to evaluate the maturity of AI support. (M1) specifies the *support of programming languages* that at least offer AI frameworks (e.g., Python). (M2) requires that at least *AI workloads have been considered during the*

*evaluation*. (M3) is treating AI applications as *first-class citizens*. For example, the programming model defines specific abstractions tailored to AI applications.

**Edge Support** shows the maturity of functions to run across the continuum. (M1) shows that real-world edge devices (e.g., Raspberry Pi, embedded AI devices, small-form computers) have been *used during evaluation or development*. (M2) specifies if the platform is *optimized for resource-constrained edge devices*. These optimizations range from low resource footprint, reduced network traffic, energy-awareness, or optimized performance on resource-constrained devices. (M3) specifies that the support for *fault tolerance and reliability in edge-cloud* systems has been explicitly considered.[PB20b]

**Workflow Composition** enables developers to build complex applications consisting of multiple functions. Workflow Composition can be done via UI (i.e., AWS Step Functions[2]) - developers can drag and drop functions and connect inputs and outputs. A UI can reduce the entry barrier by offering a convenient solution. Developers can also compose functions in code (i.e., API). Auxiliary deployment files (i.e., YAML) or domain-specific languages can also compose functions and decouple the definition from the business logic. We introduce the three-level maturity system and rank the approaches by the ease of use: (M1) *API* (in-code), (M2) *auxiliary metadata files*, and (M3) *UI*.

#### 2.2.2.2 Run time criteria

**Invocation Reliability** specifies the fault tolerance and reliability of function invocations. We introduce a three-level maturity system. (M1) specifies that *developers must take care* of failed invocations (e.g., by manual retries) (M2) specifies the support of *retry-based invocation mechanisms*. Platform-based retry mechanisms help mitigate faults during run time and are fundamental to at-least-once semantics [SWC+20]. (M3) specifies *transaction support* which guarantees graceful handling of errors in workflows.[ZCC+20] For example, platforms may offer support to compose workflows that are able to react to failures and execute recovery actions.

**Stateful** Stateful and long-running functions are becoming increasingly relevant.[JPBG19] Serverless platforms were initially designed for short-running tasks (under 15 minutes), and many current serverless offerings still have strict timeouts. Therefore, we introduce a three-level maturity evaluation to determine the level of support for executing stateful functions. (M1) specifies the possibility of performing *long-running functions*. (M2) specifies the explicit *support for stateful functions* and checkpointing. (M3) specifies if the platform has been *tested or implemented with edge-cloud* in mind. Specifically, if the platform addresses heterogeneous network conditions or node capacities.

**Invocation Triggers** are invoked by clients to execute functions. Serverless platforms usually support multiple types: HTTP, queue, and other types of events (e.g., file

[2]https://aws.amazon.com/step-functions/

changes).[SBG+22] Therefore, we introduce a three-level system to evaluate the maturity of triggers. (M1) means that *only one trigger type* is available (i.e., HTTP or queue), (M2) means that *HTTP and queue triggers* are considered, (M3) specifies that the platform allows *complex triggers* composed of multiple event sources (e.g., Amazon EventBridge[3]).

**Edge-Cloud Gateways** Edge-Cloud Gateways serve as entry points for clients to invoke and route requests to function instances. Current cloud-centric platforms deploy *centralized* gateways, which incur high network latency and practically diminish the value of deployed hardware at the edge. We introduce three levels to evaluate the maturity of gateways. (M1) specifies that only a *centralized* gateway is available (i.e., cloud-centric). (M2) is a *hybrid* approach in which the platform deploys multiple gateways across the infrastructure. The gateways can be dynamically placed and scaled, but are commonly deployed at the network's edge (e.g., 5G base stations). (M3) is the *distributed* approach in which each node acts as a gateway and can route requests.

**Platform Intelligence** Platform Intelligence describes how smart or intelligent the typical runtime mechanisms (e.g., scaling, placement, and routing) are: (M1) *analytical*, (M2) *heuristic-based*, and (M3) *AI/ML-driven* approaches. The platform must adapt to the changing infrastructure and know the current state (e.g., capacities, workload) in the edge-cloud continuum. Analytical solutions (e.g., *target value / current value*) are not resource-intensive and do not capture intricate details to optimize functions. *Heuristic-based Platform Intelligence* relies on optimization processes and is able to capture important details and optimize towards multiple goals [BQ21]). *AI/ML-driven Platform Intelligence* offers methods, such as Reinforcement Learning, to continuously learn and optimize the function deployments.

**Performance and resource consumption optimization targets** Different Serverless platforms are designed to be optimized for a specific target of function execution. Such targets can range from low latency to cost efficiency or resource usage. We introduce a three-level maturity system that ranks the *Optimization Target* by platform customer relevancy: (M1) means that the *Optimization Target* targets *Resources* (e.g., number of function instances). Customers usually pay for the execution time and do not care about the used *Resources*. (M2) means that the optimization targets the *Network*. For example, function instances placed far apart can lead to expensive cross-region network traffic and incur high latency. (M3) means the platform can optimize for *Performance or Cost* when executing serverless functions in the edge-cloud continuum.

### 2.2.3 Where are we now?

Our review includes more than 45 commercial, open-source platforms, and academic works. We only include platforms that are actively maintained, meaning they are not

[3] https://aws.amazon.com/eventbridge/

explicitly archived or abandoned[4]. Further, we only include works that have at least evaluated a prototype and thus ignore others that propose concepts or simulation-only evaluations. To structure the review, we group approaches into four categories:

1. *Core Serverless Platforms* includes bare-bones platforms to autonomously manage functions. This category includes platforms like AWS Lambda, OpenFaaS, and prototypes from academia.[PB20a, WS22]

2. *Serverless Platform Extensions* includes research works that improve existing platforms by introducing novel scaling, placement, or routing techniques.[BFRS22, NRdA+20]

3. *Serverless Programming Extensions* includes open source projects such as Zappa [Zap20] or Chalice [AWS16] and research works that introduce new approaches to programming serverless functions [BGJ+21, ZFPS20]

4. *Serverless Function Runtimes* focus on the execution aspect of functions and mostly consist of works that introduce new virtualization or isolation techniques.[GFD22, SP20]

We omit the tables in this thesis and refer readers to view them in [RND23], which show the results of our literature review for the design time and run time, respectively. In the following, we structure our key insights based on the challenges we previously identified for the SCF [NRF+22]. Specifically, we discuss the following topics:

- **Programming Support** is specifically evaluated based on the available support for *AI Support* and *State & Data Management* at design time.
- **Reliability Engineering** is evaluated based on *SLO support* and *Reliable Invocations*.
- **Performance & Infrastructure Engineering** is evaluated based on *Optimization* and *Edge Support*.

#### 2.2.3.1 Programming Support

**AI support** Our review shows that only two open source platforms focus explicitly on AI, and two commercial ones provide AI-focused services. KServe [KSe19], which extends KNative [Kna18], and Nuclio.[Nuc17] KServe focuses on AI inference and supports developers in deploying inference functions by supporting production-ready inference serving engines (e.g., TFServing[5]) Nuclio [Nuc17] focuses on data pipelines, including training. Nuclio offers an optimized function runtime that enables it to offer high

[4]For example, we omit Kubeless, a popular open source platform, because it is currently archived on GitHub and not maintained[Kub16]

[5]https://www.tensorflow.org/tfx/guide/serving

parallelism and a minimized footprint [LKRL19]. *Serverless Platform Extensions* treat AI as a first-class citizen [RRD21] by letting developers specify required AI models, which the placement component considers to reduce network traffic. Only one reviewed work [HKO21] does not support languages with AI frameworks (i.e., they only support Lua). Others have evaluated AI workloads, but only a few treat them as first-class citizens. For example, Wang et al. [WAES21] deploy AI functions on edge hardware and perform workload experiments to evaluate their scaling and placement components. Lordan et al. [LLB21] performed similar experiments to evaluate their platform and resource-constrained devices. Specifically, Google Cloud Functions and AWS Greengrass provide services to ease the development and deployment of AI workloads.

Our findings show that most commercial platforms currently do not offer any hardware accelerator support and open source ones support them only to a certain degree. The only exception to commercial platforms is AWS Greengrass, which lets developers mount and use hardware accelerators in functions. Further, we consider platforms that run on top of Kubernetes to support hardware accelerators because Kubernetes allows mounting GPUs and other hardware accelerators into functions. Interestingly, the work by Hu et al. [HBR+20] investigates hardware accelerators to improve I/O operations instead of compute operations. Werner et al. [WS22] introduce a novel level of abstraction in which the platform autonomously chooses the appropriate accelerator. AI serverless platforms (i.e., KServe, Nuclio) offer hardware accelerator support, and two serverless extensions [NRdA+20, BHQT22] explore options to enable remote GPU sharing and GPUs in edge-cloud scenarios, respectively. We also see that none of the *Serverless Function runtimes* consider hardware accelerators, two *Serverless Platform Extensions*, and only eight *Core Serverless Platforms* support it.

**Application State & Data Management** Application State & Data Management is prevalent across application paradigms, and especially AI applications tend to be data-intensive [RRD21] and stateful.[LGC+21] Moreover, *Serverless Programming Extensions* can help increase the maturity of *State & Data Management* (see Figure 2.6), but only a few platforms focus on it. For example, Karhula et al. [KJS19] focus on enabling seamless checkpointing for function migrations. Kappa [ZFPS20] introduces new Python programming primitives to take checkpoints and manage concurrency. This framework can be used to recover from failures during long-running training tasks. Rausch et al. [RRD21] introduce annotations that help the placement component to reduce the function execution and lower network transfer by shifting the computation close to the data.

Others aim to abstract the storage layer by implementing high level APIs for developers [HBR+20, ZCC+20], extend platforms by introducing actors for stateful computation [BCG+22, BGJ+21] or by implementing an extra layer (e.g., shim) to handle storage operations.[SWC+20]

Our findings show that *Stateful Function Support* support is immature across all categories. Most of the commercial platforms do not offer long-running functions, while open source platforms generally can be configured with long timeouts. AWS Greengrass [Amab] allows

long-running functions that can write and read to a device's storage and reuse variables. Cloudflare[6] offers *Durable Objects*, which give workers, located in CDNs, a unique ID and enable stateful computation. Shillaker et al. [SP20] implement a WebAssembly-based function runtime that implements an efficient local state access approach and enables stateful functions.

#### 2.2.3.2 Reliability Engineering

Dynamicity and heterogeneity are key characteristics of the edge-cloud continuum. Serverless platforms have to address these challenges by providing resilient and fault-tolerant execution. Specifically, we investigate the maturity of platforms in terms of their support for specifying Service Level Objectives (SLOs) and how reliable function invocations are.

**SLO/SLA** The *SLO-awareness* (see Figure 2.6) maturity level is fulfilled by only a few approaches, and no commercial or open source platforms allow users to configure complex SLOs (e.g., latency requirements). However, most of those platforms support the configuration in terms of available CPU cores and memory. Klingler et al. [KTS21] present a set of code annotations that allow developers to fine-tune the function execution. Specifically, these annotations can find the optimal function resource configuration (e.g., available memory) or avoid cold starts. Commercial and open source platforms partially support *Location-awareness*. Amazon and Microsoft offer edge-only environments to execute functions and include custom edge devices that can interact with the cloud platforms (AWS Greengrass and Microsoft's IoT Edge). Others let developers programmatically specify the location of function execution.[SJC+22, HBR+20, RRD21] Hu et al. [HBR+20] implement a DSL that enables developers to specify the place of execution (i.e., Edge or Cloud). Smith et al. [SJC+22] allows clients to specify the location of function execution in the invocation request, while Rausch et al. [RRD21] implement a location- and data-aware placement approach by enabling developers to specify data (i.e., buckets of object storage) needed for function execution. The *Resource-awareness* maturity is fulfilled by reducing network traffic [RRD21, SJC+22] or resource footprint [Ope16, IBM16, PB20a, BBH+22, WKB+19], improving data exchange between function instances [LLG+22, HBR+20], reducing overhead of function runtimes [KSe19, Nuc17, Amab] or explicitly considering resource usage while scaling and placing serverless functions.[LKM+22, WAES21]

**Reliable function invocations** Reliable function invocations are essential due to the dynamic nature of devices and heterogeneous networks in edge-cloud systems. Therefore, we evaluate the maturity of *Invocation Triggers*, *Invocation Reliability*, and *Edge-Cloud Gateways* (see Figure 2.6). First, most of the *Core Serverless Platforms* support HTTP, and all open source and commercial offerings provide additional queuing triggers. HTTP is the most stable and fastest method in public cloud offerings, enabling high-throughput

[6] https://developers.cloudflare.com/workers/learning/using-durable-objects/

Edge Intelligence applications (e.g., inference).[SBG+22] Queuing triggers can guarantee at-least-once execution semantics. Note that open-source platforms typically rely on external queuing systems to enable retry mechanisms (e.g., NATS in OpenFaaS [Ope16], and Kafka in KNative [Kna18]). Using external queue mechanisms enhances the interoperability between platforms. In contrast, commercial platforms offer managed queuing solutions, contributing to the vendor lock-in effect.

Second, approaches offering queuing mechanisms fulfill retry-based *Invocation Reliability*. Only Commercial platforms support mature *Invocation Reliability* [Ama14, Goo16, Mic14] by using managed queueing systems to guarantee retries and allowing workflow transactions that can react to failures. One research paper [WKB+19] is also mature by using a distributed fault-tolerant append-only storage. Our results show that several *Serverless Programming Extensions* focus on fault tolerance.[ZFPS20, ZCC+20, SWC+20] These approaches add fault tolerance by (i) adding a coordinator to orchestrate the transactions [ZFPS20], (ii) a platform independent runtime and library that enables transactions on stateful workflows that are either all committed or reverted [ZCC+20] or (iii) by introducing an extra layer between the platform and the storage to buffer updates and flush them upon successful workflow completion.[SWC+20]

Third, *hybrid Edge-Cloud Gateways* are implemented by many approaches from the research community but lack adoption from open-source projects, such as.[BQ21, BHQT22, LLB21, HKO21] For example, some approaches implement decentralized edge clusters that act as a gateway and router.[BQ21, BHQT22, LLB21, RRD+22] Bermbach et al. [BBH+22] implement a *distributed Edge-Cloud Gateway* by letting each node decide whether to execute the request locally or offload it to the next one till it reaches the cloud. Hu et al. [HBR+20] also implement the *distributed* approach by compiling a user-supplied workflow and specifying for each function in the workflow where the request should be executed (i.e., locally or in the cloud). Amazon's AWS Greengrass and Microsoft Azure IoT allow customers to set up edge clusters, which can act as *hybrid Edge-Cloud Gateways*. For example, Das et al. [DIPW20] dynamically decide whether to process an invocation request on the local AWS Greengrass deployment or offload the task to AWS Lambda.

IBM Cloud Functions has an edge extension that allows developers to pre- and post-process requests at the CDN level, which might be considered a hybrid approach. Our findings show that only a few support the *Workflow Composition* of functions via a UI (most notably commercial offerings) and based on metadata files (i.e., YAML). Most approaches allow the composition via the code API. Hu et al. [HBR+20] show the DSL with which developers can build application graphs and specify which functions can be run in parallel.

#### 2.2.3.3 Performance & infrastructure engineering

**Optimization** Optimization of function execution is an important aspect in Serverless Edge Computing and platforms offer *Performance & Resource Optimization Targets* based

on *Platform Intelligence* (see Figure 2.6). The findings show that commercial approaches do not expose any *Optimization Targets* to users and provide little details on scaling and placement. Open source platforms use analytical approaches and most often re-use scaling methods from the container orchestration service (i.e., default auto scaler from Kubernetes) or black-box performance metrics (i.e., requests per second) to determine the number of required function instances analytically. Further, all open source platforms solely offer *Optimization Targets* for scaling functions and do not offer any smart placement or routing mechanisms, except for round-robin or capacity-based.[Kna18] However, works from academia offer different *Optimization Targets* and can offer mature *Platform Intelligence* solutions. *Core Serverless Platforms* and *Serverless Platform Extensions* allow the specification of latency or deadline requirements using *Optimization-based Platform Intelligence* [BHQT22, WAES21], reduce cost for customers using *Analytical* and *ML-driven Platform Intelligence* [LKM+22, DIPW20] or increase revenue using auctions (i.e., *Optimization-based Platform Intelligence).*[BBH+22] Besides focusing on *Performance & Cost*, our review also includes approaches that focus on *Resources* and *Network*. Specifically, Huang et al. [HLY+22] use Integer Linear Programming to enable function migrations to efficiently use *Compute Resources*, other approaches reduce *Network* traffic by moving function instances towards the data [RRD21] or reduce network latency by dynamically creating computing clusters with nodes close to each other and network-aware routing [BHQT22].

**Edge Support** Commercial platforms offer two types of edge-oriented extensions: custom edge platforms [Amab] that integrate with cloud platforms and function executions at pre-defined CDN nodes.[ATC+21] The former allows the inclusion of custom devices but entails custom management to combine cloud and edge resources.[DIPW20] The latter enables developers to pre- and post-process HTTP requests and responses.[ATC+21] Open source platforms rarely support edge devices and focus on providing resource-optimized platforms (i.e., OpenWhisk Lean [IBM16]). To summarize, commercial and open source platforms offer high *Edge Support* maturity, but their solutions lack the unification of edge and cloud resources, as we envision the SCF.

*Core Serverless Platforms* and *Serverless Platform Extensions* from academia offer high maturity and unify edge and cloud. For example, Wolski et al. [WKB+19] implement a platform that runs on devices ranging from microcontrollers to consumer workstations using a distributed fault-tolerant append-only storage to guarantee function executions across the continuum. Contrary, Akkus et al. [ACR+18] introduce a local message bus deployed per node, making the approach resilient against network failures as functions on the same node can still communicate, and their resource consumption results show a smaller footprint than AWS Greengrass. Other approaches achieve a high maturity level of *Edge Support* by enabling the seamless migration of function instances in edge-cloud scenarios [HLY+22] or by introducing data-aware request routing and data replication that gradually leads to reduced network traffic between regions.[SJC+22]

### 2.2.4 What lies ahead

#### 2.2.4.1 Programming Support

Our review has shown that only a few approaches offer mature *AI Support* and *Application & State Data Management*. Specifically, we see a lack of programming support for first-class citizen abstractions.

**Edge Intelligence** Developing EI applications still remains challenging because of their complex workflows that combine edge and cloud resources to perform training and inference [RD21]. We saw that current platforms do not adequately address this, and new solutions are required. For example, AI model selection and treatment of large AI models should be first-class citizen features in the programming model [WS22].

**Application State & Data Management** Application State & Data Management still poses great challenges due to the edge-cloud continuum's heterogeneous network conditions and geo-distributed infrastructure. Thus, offering distributed cache and storage solutions is not only mandatory but also challenging in terms of implementation and usage. For example, the programming model should alleviate developers by offering appropriate abstractions that simplify access.

#### 2.2.4.2 Reliability Engineering

The review has shown that only a few approaches have reached full maturity in terms of guaranteeing efficient execution. Specifically, support for SLOs and execution guarantees still poses some challenges.

**SLO/SLA-awareness** remains challenging in terms of enabling developers to specify complex SLOs.[PMP+21b, PMP+21a] For example, developers want to specify that the inference model has a certain accuracy [RD21]. Besides, the implementation requires careful design to monitor and manage function executions efficiently. For example, the SCF should continuously learn and build AI models to make informed function scaling, placement, and routing decisions.

**Reliable Invocations** poses challenges in the implementation because each *Edge-Cloud Gateway* approach has its advantages and disadvantages. For example, the *hybrid* approach can introduce additional network overhead by routing through gateway components. While a drawback of the *distributed* approach is the dissemination of knowledge in the system. Further, execution guarantees for mission-critical complex are hard to achieve in such dynamic environments. For example, the SCF can deploy a distributed queuing mechanism that can run on all devices to achieve full maturity.

#### 2.2.4.3 Performance & Infrastructure Engineering

Current approaches lack sophisticated *Optimization* strategies. Specifically, commercial and community-driven open-source solutions do not implement adequate ones that are suitable for the edge-cloud continuum. Besides, the maturity of *Edge Support* is high in approaches from academia but lacks adoption in the open source space.

**Optimization** Optimization based on *ML/AI-driven Platform Intelligence* can continuously adapt to the dynamic edge-cloud environment and make smart scaling and placement decisions. However, we identify two challenging tasks: (1) placement and (2) operationalization. (1) the SCF must consider the implementation and placement of runtime mechanism components (e.g., scaling, placement). Each of those components can be deployed, distributed, and decentralized, requiring coordination and fallback mechanisms. (2) the operationalization of intelligent placement, scaling, and routing is complex and requires re-training due to the dynamic environment of the edge-cloud continuum.

**Edge Support** Edge Support is challenging to offer due to the large variety of devices and range of computational power. To improve this, we suggest that the SCF should abstract the virtualization and isolation layer to enable a resource-efficient and resilient function execution model on all devices and reach full maturity. Containers, WebAssembly, and microVMs each have advantages and disadvantages [LGC+22, GSAN+22, GFD22], and developers should have the option to let the SCF choose the most suitable one dynamically.

### 2.2.5 Related Work

Aslanpour et al. [ATC+21] identify opportunities and open issues. Certain aspects of our work overlap with theirs, but we additionally assess the current state. Cassel et al. [CRdRR+22] and Kjorveziroski et al. [KFT21] review serverless computing for IoT. While the former does not investigate commercial or open-source platforms, the latter neglects stateful aspects. Ioini et al. [IHPT20] focus on the state of commercial and open-source platforms. The section generally focuses on different issues and challenges and presents a novel look at Serverless Edge Computing with our criteria that investigate the programming and execution model.

## 2.3 Summary

In this section, we introduced novel criteria to evaluate the maturity of serverless computing approaches for the edge-cloud continuum. Our work is, in part, a continuation of previous research where we outlined our vision for the SCF [NRF+22]. Our review of over 45 approaches discusses challenges we previously identified and evaluates the maturity of current offerings. We identified issues around data management, the serverless programming model, and AI support. The review shows that current offerings are in

many ways still immature, and we have briefly outlined which steps to take next. The requirements for EI applications Serverless Edge Computing are manifold, as our review has shown. Therefore, we focus on the challenge of satisfying SLOs using orchestration strategies. Specifically, energy- and location/mobility-awareness are lacking, which are crucial for the success of those platforms. Based on that, we focus in the following chapter on proposing mobility- and energy-aware approaches. Mobility-awareness is similar to location-awareness by determining the movement of users, as well as reducing energy consumption, which can drive down cost for platform providers.

CHAPTER 3

# Mobility- and Energy-aware Orchestration Strategies

In this chapter, we are going to address specific challenges around function orchestration that are important for Edge Intelligence, based on our analysis in Section 2.1 and are not fully mature yet, as we have seen in Section 2.2. First, we are proposing a mobility-aware decentralized Serverless Edge Computing platform in Section 3.1 that is based on a complex metric we call *pressure*. Next, we are looking into reducing energy consumption in container-based systems, such as those commonly seen in Serverless Edge Computing platforms. Section 3.2 presents a Graph Neural Network-based scheduler that lowers energy consumption in data center scenarios, which are as critical for Edge Intelligence as edge-oriented infrastructure. The mobility-aware orchestration strategy was first presented in [RRP+22] and the energy-aware one in [RRD+24].

## 3.1 Mobility-Aware Serverless Edge Computing

Serverless Edge Computing and its function-based deployments have emerged as a useful abstraction to manage the complexity of distributed and heterogeneous edge-cloud infrastructure. Current cloud-centric orchestration services and serverless platforms are not suitable for the edge-cloud continuum due to their mobility-unawareness. In this section, we present a mobility-aware framework for edge-cloud systems, where the operational mechanisms of placement, scaling, and routing of serverless functions work in tandem. To that end, we formulate the concept of *pressure* that captures complex system behavior in a single metric. The pressure-based framework handles geo-distributed workload and user mobility by reducing overall function latency and increasing data throughput while efficiently using edge resources. Our contributions include a novel Serverless Edge Computing-based framework revolving around pressure and a real-world proof of concept evaluation. Our approach combines the benefits of a centralized control

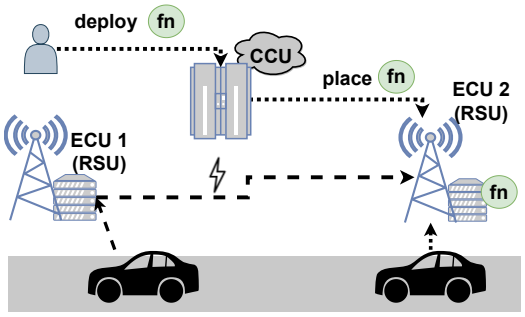



Figure 3.1: Scenario

plane with a decentralized data plane. The results show the efficacy of the platform to address operational goals and make effective and deterministic tradeoffs between system utilization and application performance. The novel concept of pressure shows great extensibility and builds the base for a plethora of future works.

### 3.1.1 Introduction

Serverless functions encapsulate business logic into atomic units of deployment that are autonomously managed by platforms to hide computing infrastructure from application developers. The platform's main responsibilities of scaling, placing, and routing requests need to consider the mobility of users in modern edge-cloud scenarios [ATC+21]. The serverless paradigm can help manage the complexity of operating applications on the edge-cloud continuum, where the geo-distributed and heterogeneous nature of infrastructure presents unique challenges and opportunities [ATC+21]. Applications across the edge-cloud continuum include low latency applications spanning various domains, such as autonomous vehicles [HPS+15], or augmented urban reality [LHOH18]. These use cases share two common characteristics that are particularly challenging to address: geo-distribution and user mobility [TLG16]. Service migration tackles the problem of user mobility and requires platforms to constantly re-allocate resources[OZC18]. Edge-cloud systems are composed of multiple compute units to enable computation in proximity independent from where the requests come [RCLN20]. Thus, users constantly re-connect to the nearest access point, making mobility-aware serverless orchestration strategies necessary for future applications across the edge-cloud continuum. Existing efforts to extend serverless systems (e.g., Kubernetes) to the edge [XSXH18] have only recently begun to address how to adapt cloud-based serverless function orchestration [HSW+21, RCLN20] and cloud-centric solutions are not feasible [OZC18]. Serverless orchestration strategies include scaling and placement of application instances (i.e., function replicas) and the routing of incoming user requests.

Figure 3.1 presents a scenario from the domain of autonomous vehicles in which cars are connected to the closest Road Side Unit (RSU). Specifically, the system is composed of different compute units. Namely, a central Cloud Compute Unit (CCU) and two Edge Compute Units (ECU). In our case, the ECUs are RSUs and offer limited hosting

capabilities. Cars automatically connect to the closest one and offload tasks to the ECU. Tasks include assisted driving, traffic management, and infotainment [LMC+21]. In our case, tasks are processed by stateless functions that are deployed by a developer in the CCU and placed from there across the edge-cloud continuum. The issue is that functions need to be adapted to migrate along the users' trajectory. Specifically, ECU 1 does not host the needed function and needs to re-route the task to ECU 2, where it can be processed. Naturally, if functions do not migrate, the invocation suffers from long network latency trips. Thus, serverless platforms require scaling, placement and routing strategies to migrate functions according to the users' mobility. This entails the consideration of non-negligible network delays, varying workload conditions, resource usage, and user mobility. To that end, we propose a mobility-aware framework based on the novel concept of *pressure* - a metric that measures the force between compute units based on locality and demand. Our framework implements a network-aware offloading strategy and pressure-based scaling and placement that work in tandem. The pressure is highly extensible, and we show this using two pressure formulations that deterministically yield expected results. Additionally, the pressure-based approach significantly reduces communication between compute units, which otherwise might result in a bottleneck on the network layer [GSSS20, MGST22]. Evaluation on a testbed shows that our performance-oriented pressure reduces the 90*th* percentile Round-Trip Time (RTT) by up to 77%, and network latency can be reduced by up to 92% compared to the default Kubernetes Autoscaler. The resource-efficient pressure reduces the average number of running containers by 83% in comparison to the performance-oriented variant while still reducing the RTT by 60% and the network latency by 74%. Inter-network traffic is reduced by 92% and 67%, respectively.

The main contributions of this section are as follows.

- We propose a novel joint scale & placement and routing framework for serverless functions across the edge-cloud continuum.
- We implement two different pressure calculations showing deterministic behavior and good results.
- Our open source proof-of-concept implementation[1] works in tandem with Kubernetes.

The rest of the chapter is organized as follows. Section 3.1.2 discusses related works and highlights issues around them and differences to our work. Afterward, Section 3.1.3 briefly introduces Function-as-a-Service (FaaS) and our system model. Section 3.1.4 introduces the *pressure*, and Section 3.1.4.4 the pressure-based serverless platform, of which we describe our proof-of-concept (PoC) implementation in Section 3.1.5. Section 3.1.6 outlines our methodology for evaluating the proposed system. Section 3.1.7 highlights

[1] https://github.com/phip123/mobility-aware-faas

the key performance indicators, displays the results of our experiments on the testbed, and discusses advantages and disadvantages. Section 3.1.8 concludes our work and gives a perspective for future work.

### 3.1.2 Related Work

The related work is split into three categories. First, we look into works that investigate offloading and service migration solutions. Afterward, we focus on solutions that propose edge-cloud serverless function platforms. Then, we consider works that tackle mobility issues.

#### 3.1.2.1 Mobility-aware offloading and service placement

Labriji et al. [LMC+21] propose a proactive VM service migration scheme in the Internet of Vehicles. Tang et al. [TGW+20] investigate Quality of Service (QoS) requirements in vehicle fleets. Using virtualization, they reduce resource provisioning costs while ensuring QoS requirement satisfaction. Sun et al. [SZML19] formulate a mix-integer non-linear stochastic optimization problem to optimize resource usage and task placement jointly. Maia et al. [MGDVdC20] tackle the joint issue of request routing and service placement using limited look-ahead control and a genetic algorithm. These approaches and more [GCZ20, MMN21, YLZ+20] propose mobility-aware solutions to routing and placement, but evaluate their approaches solely in simulations. While they show promising results, our work differentiates by solving these issues for serverless functions using an integration into Kubernetes.

#### 3.1.2.2 Serverless function orchestration in edge-cloud systems

The authors of [HSW+21] use a distributed Reinforcement Learning (RL) approach to manage applications in Kubernetes using decentralized request dispatchers and a centralized service orchestrator. This design is similar to ours but differs from the approach to minimize data communication between the edge and the cloud. Their approach uses Graph Neural Networks to encode the system's state, while our scale and placement components work on the aggregated complex pressure metric. Rossi et al. [RCLN20] use an RL agent to scale applications and schedule on a geo-distributed system using a network-aware placement heuristic, but do not propose a solution to request routing. Tamiru et al. [TPTE21] extend Kubernetes to manage multiple clusters and re-route traffic across all clusters, but focus neither on low-latency applications nor the characteristics of edge computing. Other papers that evaluated orchestration services lack edge-based load balancing strategies [PWHH19, HSS+19]. Baresi et al. [BM19] propose a serverless platform for edge computing for which they also built a prototype based on the open-source FaaS platform OpenWhisk. Their architecture is based on a decentralized approach containing a self-contained serverless platform per region (e.g., per city). In contrast to our work, they do not explicitly tackle placement, scaling, and routing but resort to measuring the idle memory footprint and performance in a static environment (i.e.,

without scaling). Further, these works have not been evaluated in the context of mobility in edge-cloud systems.

#### 3.1.2.3 Mobility-aware serverless function deployments

Few works have approached the task of implementing and evaluating mobility-aware serverless function solutions. Wang et al.[WGZ+21] propose an RL approach that explicitly tackles user mobility issues but does not jointly investigate placement and routing. Baresi et al. [BHQT22] present NEPTUNE, a network and GPU-aware serverless function orchestration platform that is location-aware and built on top of Kubernetes. They use Mixed Integer programming to perform placement decisions and split the system into separate communities (similar to our compute unit definition). In contrast to our work, communities are not aware of each other. That causes the system to fail if a community is overloaded and has to be reconfigured. Our approach differentiates us by presenting a novel pressure-based framework for serverless functions.

### 3.1.3 Background

Serverless platforms offer convenient deployment of serverless functions by requiring only a containerized stateless application to scale, place, and route requests autonomously. While there are many issues regarding serverless computing (e.g., cold starts [BCF+17]), we specifically focus on these three base function orchestration strategies. We build our platform on Kubernetes, a commonly used orchestration service among open-source serverless frameworks [PKC19] and research [BHQT22, RCLN20, TPTE21]. This paradigm requires that applications offer a single stateless function that can be invoked over the network (i.e., HTTP). While serverless platforms abstract away the details from users, default components are not suited for modern edge-cloud scenarios. Especially when facing mobility, cloud-centric approaches (i.e., scaling and request routing) fail to ensure user requirements. The serverless paradigm promises to abstract the underlying infrastructure away from the platform users, and therefore, deployment should only require minimal deployment information. Therefore, we provide a novel serverless platform to perform dynamic mobility-aware function orchestration strategies. Based on the example in Figure 3.1, we introduce our notation for the system. As mentioned, we envision edge-cloud systems to consist of one or multiple CCUs with unlimited hosting capabilities, but experience non-negligible network delays to users (i.e., cars). ECUs are deployed at the edge, comprised of resource-constrained devices, and in proximity to users. Our approach is dynamic and does not differentiate between these two compute units. Therefore, the platform manages a set of compute units $C$. Each compute unit $c \in C$ contains multiple nodes $n \in N$ and exactly one gateway. The gateway acts as a request router and has function orchestration capabilities, such as the aggregation of fine-grained monitoring data and the local placement of functions. Platform customers deploy functions with resource requirements for CPU and memory ($CPU_f^{req}$ and $MEM_f^{req}$, respectively). A function replica running on node $n$ is denoted as $f_i$, where $i$ is a unique index, and we can look up the node for each $f_i$. Users $u \in U$ generate requests and invoke the deployed

functions. A compute unit $c$ receives requests from user $u$ of the given function $f$ ($R_{u,c,f}$). We also denote requests that are being re-routed from compute unit $x$ to compute unit $y$ with $R_{x,y,f}$. Table 3.1 lists all symbols.

### 3.1.4 Pressure

*Pressure* is an abstract metric composed of multiple low-level metrics and does not impose requirements regarding its definition. Therefore it is versatile and can be formulated toward specific use cases. As our evaluation shows, the pressure can be composed of well-known low-level metrics (i.e., resource utilization). The combination of them positively influences dynamic function orchestration. In the following, we introduce our vision of pressure based on an example scenario.

#### 3.1.4.1 Example scenario

Figure 3.2 depicts a system consisting of three compute units: $ECU1$, $ECU2$, and a cloud ($CCU$). The scenario highlights how the pressure can guide function orchestration strategies and make them mobility-aware. It is based on our introductory example in Section 3.1.1. At the beginning ($t_1$), one function replica is available in $CCU$. Thus, the car is offloading requests to $ECU1$ that are being re-routed to $CCU$, because no function replicas are running on $ECU1$. The network latency negatively impacts different factors (i.e., performance, bandwidth usage), and *ECU1* should process requests. The platform has to adapt the function deployment and place a new function replica in $ECU1$. The pressure captures this mismatch, resulting in a high value between $ECU1$ and $CCU$. At step $t_2$, the system starts a function replica on $ECU1$ and incoming requests can be processed internally (resulting in low pressure). As no requests are being processed in $CCU$, the replica can be shut down. In the next step, $t_3$, the car moves away from $ECU1$ and connects to $ECU2$. $ECU2$ does not host the function and therefore has to re-route them to $ECU1$. In contrast to cloud-centric load balancers, a mobility-aware platform has to consider network latency between compute units when re-routing requests. As depicted in the figure, $ECU2$ is closer to $ECU1$ than $CCU$ (e.g., consider replicas running in $CCU$). In $t_3$, the pressure between $ECU2$ and $ECU1$ is high again, and the platform has to adapt the function deployment dynamically.

This scenario highlights the following important aspects: (1) the platform efficiently places the function from the cloud along the car's path, (2) unused functions are shut down, and (3) the pressure gives direction and enables a joint scale and place algorithm, allowing us to make mobility-aware decisions.

#### 3.1.4.2 Definition

The pressure is calculated for each pair of compute units (i.e., $ECU1$, $ECU2$ and $CCU$) and for each function $f$, where requests were re-directed from one compute unit to another. We denote the pressure as $p_{ECU1,ECU2,f}$, which translates to requests being sent from the origin compute unit (i.e., $ECU1$) to the target compute unit (i.e., $ECU2$). The

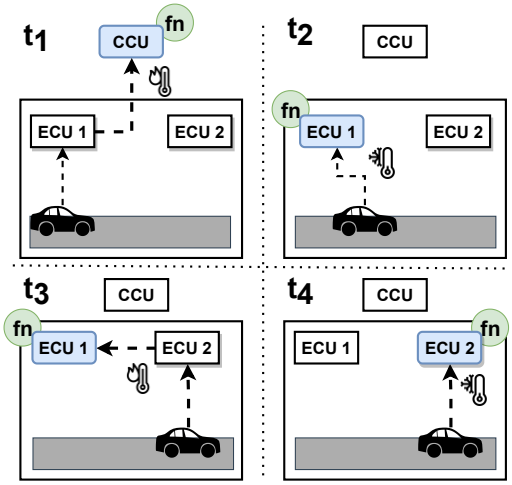



Figure 3.2: Scenario

pressure is directed (i.e., $p_{ECU1,ECU2,f} \neq p_{ECU2,ECU1,f}$). Note that the same compute unit puts pressure on itself (i.e., $p_{ECU1,ECU1,f}$). This is important to adapt compute units independently and ensures requirement satisfaction. Generally, we expect a high

| Variable | Description |
|---|---|
| $C$ | Set of compute units |
| $F$ | Set of deployed functions |
| $N$ | Set of nodes |
| $CPU_f^{req}$ | CPU requirement of function $f$ |
| $MEM_f^{req}$ | Memory requirement of function $f$ |
| $CPU_{f_i}^{\text{rel}}$ | CPU usage of function $f_i$ |
| $f_i$ | Function replica $f$ with index $i$ |
| $U$ | Set of users sending tasks (i.e., a car) |
| $R_{u,c,f}$ | Set of requests from user (or compute unit) $u$ received by $c$ |
| $p_{c_o,c_t,f}$ | The pressure from $c_o$ on $c_t$ for function $f$ |
| $d(n_1, n_2)$ | Network latency between nodes $n_1$ and $n_2$ |
| $t_f^{rtt}$ | Target round-trip time of function $f$ |
| $p_{x,y,f}^{REQ}$ | Request pressure between compute unit $x$ and $y$ |
| $p_{x,y,f}^{RES}$ | Resource pressure between compute unit $x$ and $y$ |
| $p_{x,y,f}^{RTT}$ | Performance pressure between compute unit $x$ and $y$ |
| $p^{\text{perf}}$ | Performance-oriented pressure definition |
| $p^{\text{eff}}$ | Resource efficient pressure definition |
| $w_{f_i,c}$ | Routing weight of replica $f_i$ for gateway in compute unit $c$ |

Table 3.1: Symbols

pressure between two compute units when the latency between them is high (which has to be defined for each function individually) and a relatively large quantity is re-routed, causing performance degradation for many users. On the other hand, if the network latency is negligible, the pressure should be low (i.e., $p_{ECU1,ECU1}$). Additionally, each compute unit should be able to calculate it based on local compute unit metrics and requests. This minimizes communication costs between compute units and the centralized control plane. Before introducing our pressure-based platform, we give details about the pressure formulations used in our evaluation.

#### 3.1.4.3 Pressure formulations

We introduce two different pressure formulations: $p^{\text{eff}}$ and $p^{\text{perf}}$ that are composed of three and two low-level components respectively: (1) the request distribution ($p^{REQ}_{x,y,f}$), (2) the performance ($p^{RTT}_{x,y,f}$) and (3) the resource usage (i.e., average CPU usage) of the target compute unit $y$ ($p^{RES}_{x,y,f}$).

$p^{REQ}_{x,y,f}$ captures the request distribution, the relative amount of requests spawning from $x$ and being processed by $y$. It is formulated as follows, $C_y$ contains all compute units (including $y$) that have sent function requests to $y$ and $R_{x,y,f}$ is the number of requests sent from compute unit $x$ to $y$. We divide the result by the highest ratio to normalize values between $[0, 1]$.

$$R_{y,f} = \sum_{c \in C_y} R_{c,y,f}; u_{y,f} = \frac{R_{y,f}}{|C_y|} \tag{3.1}$$

$$w^{max} = max_{c \in C_y}(\frac{R_{c,y,f}}{u_{y,f}}) \tag{3.2}$$

$$p^{REQ}_{x,y,f} = (\frac{R_{x,y,f}}{u_{y,f}})/(w^{max}) \tag{3.3}$$

$p^{RTT}_{x,y,f}$ anticipates applications that require a specific RTT. As network and performance fluctuate during workloads, we formulate a pressure that models this circumstance. It is based on the RTT and a user-specified requirement ($t^{rtt}_f$). Specifically, we use the 99th percentile of the RTT of function $f$ from $x$ to $y$ ($R^{P99}_{x,y,f}$). The pressure increases when compute unit $x$ sends requests to far away, or overloaded compute units. This allows to fine-tune each function and gives the ability to adapt the importance of performance dynamically. We use a logistic-based function (Eq. 3.4) to act as a cost function to assess the current state.

$$l(d) = \frac{(L-b)}{1 + e^{-k(d - t^{rtt}_f)}} + b \tag{3.4}$$

$$p^{RTT}_{x,y,f} = \frac{l(R^{P99}_{x,y,f})}{L} \tag{3.5}$$

We use the following parameter values:

$$L = 250, b = 1, k = 0.2, t_f^{rtt} = 70 \tag{3.6}$$

Figure 3.3 shows the resulting pressure function $p_{x,y,f}^{RTT}$ which stays in the range $[0, 1]$. In case no traces exist, we set the pressure to 0. The parameters were chosen to penalize (i.e., calculate a high pressure) for requests with an RTT that violates the threshold and favor (i.e., calculate a low pressure) if the RTT is below the threshold. Future works can include investigating which parameters are suitable for different applications.

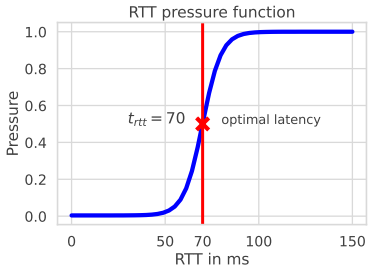



Figure 3.3: $p^{RTT}$ - pressure function

$p_{x,y,f}^{RES}$ is based on the ratio between requested CPU usage and current average CPU usage over all function replicas in the target compute unit $y$ ($CPU_{f,y}^{mean}$). This approach is similar to how Kubernetes' Horizontal Pod Autoscaler (HPA) calculates the ratio for scaling applications and is in contrast to the others not influenced by the origin compute unit $x$. $p_{x,y,f}^{RES}$ can yield higher values than 1. We want to note here that this pressure component can be easily extended to include other resources (e.g., GPU, I/O), but our evaluation application mainly uses CPU.

$$p_{x,y,f}^{RES} = \frac{\text{CPU}_{f,y}^{mean}}{\text{CPU}_f^{req}} \tag{3.7}$$

We calculate each pressure component and multiply them to create the final pressure value. The pressure formulation can be easily adapted and weighted towards different goals and allows generalization across different infrastructures. To demonstrate the adaptability, we use in our experiments two different formulations: a pressure that contains all components (**eff**icient), and one that omits the $p_{x,y,f}^{RES}$ component to scale and place with a focus on **perf**ormance (see equations 3.8 and 3.9, respectively). The presented pressure formulation is our initial evaluation of this concept and can be extended to more sophisticated variants in future work (e.g., weighted sum).

$$p_{x,y,f}^{\text{eff}} = p_{x,y,f}^{REQ} \cdot p_{x,y,f}^{RTT} \cdot p_{x,y,f}^{RES} \tag{3.8}$$

$$p_{x,y,f}^{\text{perf}} = p_{x,y,f}^{REQ} \cdot p_{x,y,f}^{RTT} \tag{3.9}$$

#### 3.1.4.4 Pressure-based serverless platform

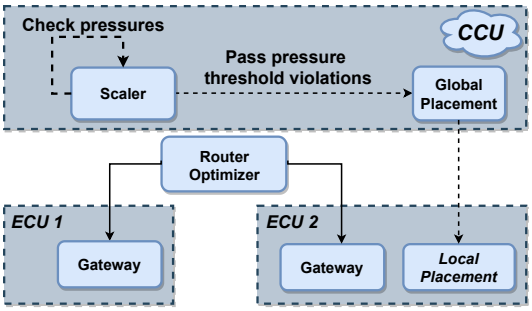



Figure 3.4: Framework

Figure 3.4 shows the components of our pressure-based serverless platform and highlights interactions between them. We employ a joint scale and placement system to dynamically manage the function lifecycle and a network-aware router optimization component. The centralized scale component finds pressure threshold violations. These are passed to the placement component, which consists of a global and local component. The global placement component determines in which compute unit a new function replica should be spawned, and the local one selects the node. The router optimizer frequently updates the gateways in the system.

#### 3.1.4.5 Scaling

The framework uses a reactive threshold-based strategy to scale functions. The component periodically receives the pressure ($p_{c,x,f}$) for each pair of function and compute unit and checks for any threshold violations. In case the pressure is too high (i.e., $p_{c,x,f} > thr_{max}$), it will scale up, and in case the threshold falls below (i.e., $p_{c,x,f} < thr_{min}$) it will trigger a scale down action. The number of replica to scale up or down is currently statically set to one. If the pressure is 0, but replicas are pending to start, we do not take any scale down actions. Additionally, we identify *zero sum actions* to re-use existing resources efficiently. This lets the system identify scaling decisions that would scale down a replica in a compute unit in which we want to schedule a new one. The scaler outputs a list of tuples containing the violating compute unit ($c$) and the corresponding function ($f$), which is passed on to the global placement component.

#### 3.1.4.6 Placement

Our work proposes a two-level decentralized placement approach. The global placement component determines the compute unit, and the local placement one selects a specific node. The *global placement component* has to resolve two kinds of events: scale-up and scale-down. In case of a scale-down event, the pressure $p_{c,x,f}$ was lower than $thr_{min}$, and the current selection strategy chooses the replica with the lowest resource usage (i.e., CPU) in $c$. In the case of a scale-up event, the pressure $p_{c,x,f}$ was higher than

$thr_{max}$. Therefore, the global placement policy receives a list of pressure violations $S$ and the set of compute units $C$ (see Algorithm 1). Each pressure violation in $S$ consists of the violating compute unit $c$ and the function $f$. First, it decides whether $c$ can host another replica (line 6). Otherwise, it selects the next best compute unit using latency and pressure (lines 8-23). First, we sort all compute units in ascending order based on the network latency relative to compute unit $c$ (line 10). We assume that the global component is aware of the underlying network topology. In the next step, we check for each possible target compute unit $x$ if it can host another replica of $f$ and check if the pressure threshold ($thr_{max}$) is violated based on $p_{c,x,f}$. If not, we select $x$ to spawn a new replica. If no compute unit was found, we select the nearest one that can fit another replica (line 20). The pressure allows us to avoid overloading compute units by selecting the nearest one. Afterward, the target compute units and functions (stored in $M^*$) are passed on to the local placement component.

**Algorithm 1** Placement

```
1:  function globalPlacementPolicy(S, C)
2:      Input: S, C: S contains the pressure violations, and C is the set of compute
    units
3:      Output: M*: set containing the target cluster for each function
4:      M* ← ∅
5:      for each (c, f) ∈ S do
6:          if canHost(c, f) then
7:              M* ← M* ∪ {(c, f)}
8:          else
9:              min_c ← null, C* ← C \ {c}
10:             C* ← sortByLatencyAscending(c, C*)
11:             for each x ∈ C* do
12:                 if canHost(x, f) then
13:                     if p_{c,x,f} < thr_max then
14:                         min_c ← x
15:                         break
16:                     end if
17:                 end if
18:             end for
19:             if min_c = null then
20:                 min_c ← firstThatCanHost(c, C*, f)
21:             end if
22:             M* ← M* ∪ {(min_c, f)}
23:         end if
24:     end for
25:     return M*
26: end function
```

The *local placement component* receives the functions and compute units to spawn new replicas and prepares for each pair one replica. This component, situated directly in the compute unit (i.e., on the gateway), can make fine-grained decisions to place the replica on the best-fitting node. We do not propose a sophisticated local placement heuristic in this work and resort to the default Kubernetes scheduler. Our framework intends to focus on the global level leaving fine-grained decisions open for custom solutions. Note that the local placement component has much more information available, and fine-grained telemetry data does not have to be propagated to the global components (as depicted in Figure 3.4). This also increases the system's scalability as the global component does not have to compute the pressure but instead selects the compute unit to place the next replica. The routing optimizer includes the newly spawned replica and routes traffic to it.

#### 3.1.4.7 Router Optimizer

The gateways route, based on a weighted-round-robin technique, and the router optimizer periodically updates the weights based on network latency and resource usage. It calculates the weights for all replicas running in the compute unit and for external compute units hosting the function. For each deployed function $f$, compute unit $c$, and function replica $f_i$, it calculates the weight $w_{f_i,c}$ as follows:

$$cpu_{f_i}^{diff} = 1 - CPU_{f_i}^{rel} \tag{3.10}$$

$$lat_{f_i,c} = max(0.01, l(d(n_{f_i}, n_c))) \tag{3.11}$$

$$w_{f_i,c} = (cpu_{f_i}^{diff} \cdot lat_{f_i,c}) \cdot 100 \tag{3.12}$$

$CPU_{f_i}^{rel}$ denotes the replica's CPU usage relative to the number of available cores and is normalized in the range of $[0, 1]$. In case no CPU usage is available for a replica, we assume the load is 0. Equation 3.11 uses the previously defined logistic-based cost function (see Equation (3.4)). $t_f^{rtt}$ is a parameter that users have to provide. Equation (3.11) uses the dynamically updated mean network latency, obtained via the distance function $d$, between the replica's node ($n_{f_i}$) and the compute unit's gateway node ($n_c$). We avoid weights of 0 and use 0.01 as a fallback value. The final weight for replica $f_i$ ($w_{f_i,c}$) is calculated in equation 3.12. We multiply by 100 because our weighted round-robin implementation expects integers. The presented strategy favors replicas with low CPU usage and low network latency.

$$w_{x,c} = \text{ceil}[w_x^{mean} \cdot lat_{x,c}] \tag{3.13}$$

Equation 3.13 shows the weight calculation to re-route requests from one compute unit $c$ to another compute unit $x$. It takes the average over all weights ($w_x^{mean}$) in compute unit $x$ and weights it with the network latency between the compute units $x$ and $c$. This optimization requires compute units to interact with each other.

### 3.1.5 Proof-of-concept implementation

All components run as Python daemons and cache any events published through Redis. The gateway router is implemented in Go[2] and listens on weight changes via etcd, a distributed key-value storage. The PoC implementation does not deploy multiple local schedulers but uses a centralized unit (default Kubernetes scheduler). It also does not calculate the pressure in a decentralized manner, in contrast to our vision that dictates each compute unit's gateway to calculate it. However, this does not influence our evaluation, and we consider this as future work. Resource usage is pushed from each node via *telemd*[3] through Redis, and the workload is generated by the distributed load testing framework *galileo*[4] [RRPD19]. In the following, we present the RTT and network latency details. Figure 4.4 depicts a request that is forwarded from the *ECU 2*'s gateway to the *ECU 1*'s gateway and finally arrives at a replica, which contains an OpenFaaS watchdog[5]. The request is forwarded to an internal Python-based Flask HTTP server that can process up to four requests in parallel. Based on the automatically set timestamps, we can calculate RTT ($t_{\text{start}} - t_{\text{end}}$), an estimate for the network latency between nodes ($t_1 - t_0$) and the execution time ($t_3 - t_2$). Further, the figure highlights the different network connections: internet, inter-network, and intra-network. We consider the internet to be all external traffic paid by the client. Inter-network is the most expensive for the platform provider and is the traffic between two networks (i.e., compute unit-to-compute unit connection). Intra-network traffic stays in one network and is cheap.

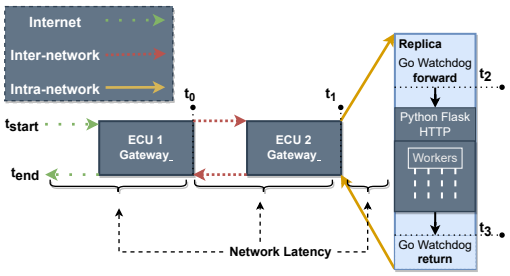



Figure 3.5: Lifecyle of a trace and network types

#### 3.1.5.1 Testbed

The testbed includes a variety of devices, and each is assigned to one compute unit, see Table 4.1 for further information. It is set up using the lightweight Kubernetes distribution K3s[6]. There are three compute units in total: *IoT Box*, *Cloudlet*, and *Cloud*. *IoT Box* and *Cloudlet* are considered to be ECUs and the *Cloud*, a CCU. The edge-based compute units are derived from [RLF+20]. We emulate the network latency between the

[2]https://github.com/edgerun/go-load-balancer/
[3]https://github.com/edgerun/telemd/
[4]https://github.com/edgerun/galileo
[5]https://github.com/openfaas/of-watchdog
[6]https://k3s.io/

Table 3.2: Testbed

| Device | CPU | RAM | Cluster |
|---|---|---|---|
| 1x AsRock | 8x Ryzen @ 2 GHz | 32GB | IoT Box |
| 1x RPI 4 | 4x Cortex @ 1.5 GHz | 1 GB | IoT Box |
| 1x TX2 | 4x Cortex @ 2 Ghz | 8 GB | IoT Box |
| 1x Nano | 4x Cortex @ 1.4 GHz | 4 GB | IoT Box |
| 1x Xeon | 4x Xeon @ 4.6 GHz | 16 GB | Cloudlet |
| 4x Xeon | 4x Xeon @ 3.4 GHz | 8 GB | Cloudlet |
| 4x VM | 4x vCPU @ 2 Ghz | 8 GB | Cloud |
| 2x NUC | 4x i5 @ 2.2 GHz | 16 GB | *Clients* |

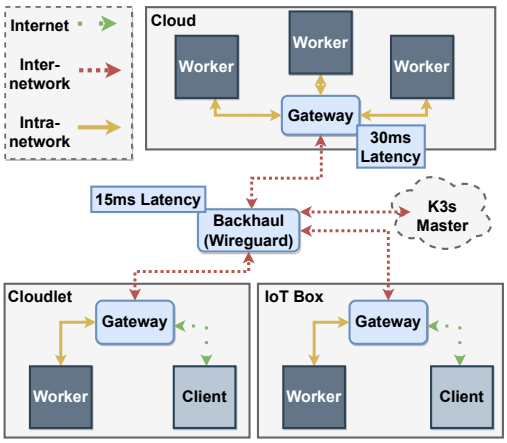



Figure 3.6: Testbed setup

compute units in our system using the Linux network traffic shaping tool *tc*. Network latency is based on a study by Braud et al. [BPKP20]. A request from the edge to the cloud has a network latency of 60ms, and between the ECUs 30ms. The intra-network latency remains untouched, as well as the connection between users and ECUs ($\leq$ 10ms). Figure 3.6 depicts the basic structure of our testbed.

### 3.1.6 Experimental Setting

#### 3.1.6.1 Workload

The generated workload emulates a typical scenario for our use case: cars move around and connect to different compute units over time. This is done by first sending requests to the *IoT Box* and afterward to the *Cloudlet*. Figure 3.7 depicts the requests over time and highlights the origin compute unit (i.e., which compute unit the users are connected with and send the request to). First, users send requests to the *IoT Box* and then move on to the *Cloudlet*. In the end, both compute units receive requests. The experiments last around 9 minutes, processing 6000 requests (up to 35 per second). The deployed function

busy waits 50ms and consumes 100% CPU during execution. The 50ms stem from our preliminary results that investigated the function execution time using a container that utilizes a modern hardware accelerator to execute an AI-based object detection (Google's EdgeTPU[7]). This work disregards the performance heterogeneity of compute units and focuses on evaluating our initial pressure definitions in a mobility-driven scenario.

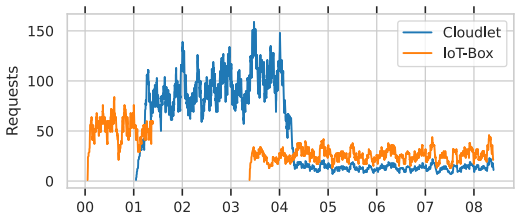



Figure 3.7: Workload over a 5 second rolling window during the 9 minute experiment.

### 3.1.6.2 Comparison strategies

We compare our solution with four different scaling and placement strategies. They are split into two groups, which affects the replica they observe: *central* and *compute unit-level*. The approaches are based on the official Kubernetes algorithms but are re-implemented in Python. This decision was made in order to guarantee equality concerning the monitoring data that is used to evaluate the compute unit state. In the *central* approach, the scaling component observes all replicas, no matter the compute unit they are in. This is the typical compute unit-unaware approach that scales and places without considering the difference in network latency between the compute units. The formulation to get the *target* number of replicas required to run to satisfy the CPU and RTT thresholds is as follows:

$$target = ceil[|\text{replicas}| \cdot (\frac{\text{currentMetric}}{\text{targetMetric}})] \tag{3.14}$$

We denote these approaches as ***HPA*** (CPU) and ***HLPA*** (RTT). In contrast to the compute unit-unaware *central* approach, the *compute unit-based* approach evaluates the *HLPA* per compute unit. This approach is denoted as ***HLCPA*** and scales each compute unit individually, thus unaware of other compute units. Again, we implemented this approach for our testbed in a centralized manner, but in a real world scenario, a scaling component runs in each compute unit. Further, each compute unit runs at least one function instance to avoid cold startup (because the minimum of running replicas is set to 1). It does not re-route across compute units which entails that if the compute unit is full, there is no mechanism to allocate new resources in another compute unit. If the compute unit is full, it would need additional logic to select the next "best" compute unit. During the evaluation, all approaches use the presented network-aware routing strategy.

[7]https://coral.ai/

| Parameter | Value | Description |
|---|---|---|
| $thr_{min}$ | 0.1 | Minimum pressure threshold |
| $thr_{max}^{\text{perf}}$ | 0.5 | Max. pressure threshold for $p^{\text{perf}}$ |
| $thr_{max}^{\text{eff}}$ | 0.3 | Max. pressure threshold for $p^{\text{eff}}$ |
| $t^{rtt}$ | 70 | Target RTT in ms |
| $t_{\text{router}}^{rtt}$ | 35 | Target RTT in ms for router optimization |
| $CPU_f^{req}$ | 1000 | CPU millis requested from eval. function |
| $MEM_f^{req}$ | 512 | MB requested form eval. function |
| $t^{\text{CPU}}$ | 60 | Target CPU usage of HPA |

Table 3.3: Evaluation parameters

This evaluation setup is similar to Baresi et al.[BHQT22], who also propose a serverless platform for edge-cloud systems.

#### 3.1.6.3 Parameters

We repeat each experiment five times and set the scaler's reconciliation interval to 15 seconds while the router optimizer runs every second. In contrast, the *Perf.* experiments run the optimization every 5 seconds to overcome the limitation of only scaling up functions by one replica. The reconciliation interval for the scale/placement components is based on the Kubernetes default. All other parameters are shown in Table 3.3.

### 3.1.7 Results

#### 3.1.7.1 Resource Usage and Scheduling

It is essential to understand when and where replicas were placed to understand the experiments' performance and data throughput aspects. The left side of Figure 3.8 shows the CPU usage per replica over all experiments as boxplots. We observe that the *HLCPA*, *HLPA*, and *Perf.* approaches have the lowest CPU utilization. The low CPU usage can be explained due to the short function execution time (50ms) and the high number of replicas deployed. The *Eff.* approach has on average a 20% CPU usage while the *HPA* approach has the highest one with around 40-50% on average.

Figure 3.8 also depicts the average number of replicas running per second in the right plot. This figure explains the low and high CPU usage for each approach. For example, *HPA* and *Eff.* have less than three replicas running per second, whereas the other approaches use more than 12. That means requests in the former cases are distributed over a few instances, causing them to process requests concurrently. While in the latter, the request distribution leads to a lower invocation rate per replica. It also shows that even though our application uses 100% CPU during execution, the *HPA* approach did not scale often and makes CPU-based scaling not feasible for the evaluated application. A scaling approach based on the queue length of the functions might be a better indicator.

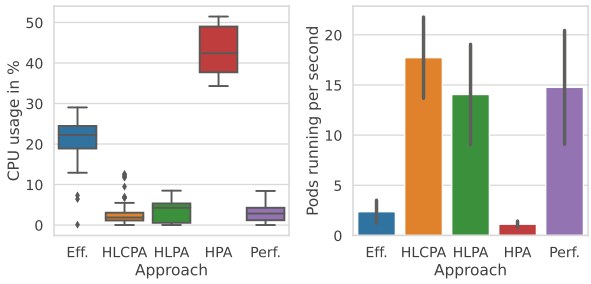



Figure 3.8: CPU usage and mean number of replicas

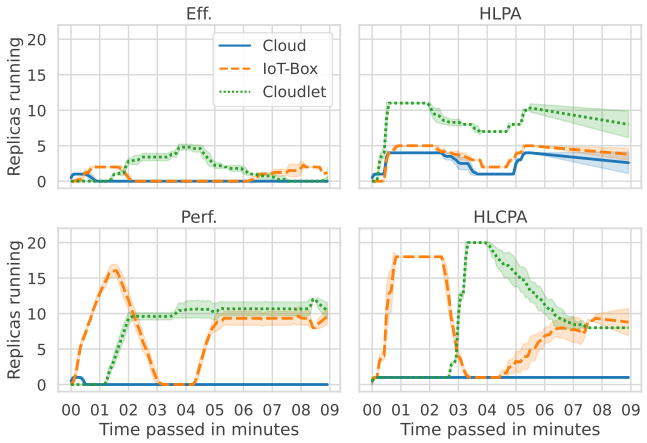



Figure 3.9: Replicas running per cluster

To deepen the analysis with respect to placement, we include Figure 3.9 that shows the number of containers during the experiments. We exclude the *HPA* approach because it scaled up the function only once and placed the container in the *Cloudlet*. The results demonstrate the dynamic aspect of our pressure-based approaches as they remove replicas from the *IoT-Box* while users move to the *Cloudlet* (visible around minute 2). The *HLPA* approach also scales down during this time but randomly selects replicas to remove. This highlights a pressure-based framework's ability to adapt functions regarding user mobility. While *HLPA*, *HLCPA*, and *Perf.* mostly scale up to the maximum number of replicas, the *Eff.* approach keeps the number of containers below five replicas while maintaining good performance.

#### 3.1.7.2 Performance

Figure 3.10 displays on the left side the average 90*th* percentile of RTT and on the right side, the network latency across all experiments. The error bars show the 95% confidence interval in which the true mean lies based on a bootstrap sample of 1000. Note that we exclude a small percentage of outliers (around 30 from 6000 requests) that were caused

by timeouts through the early tear-down of replicas that were still processing requests. While *HPA* had the highest RTT and network latency, the *Eff.* approach still violates the latency by having requests longer than 150ms. Interestingly, the *HLCPA* experiments had trouble maintaining a low RTT. We argue that this circumstance happened because it took the system some time to scale up in the *IoT Box* because resources were slowly released from the *Cloudlet* (see Figure 3.9 from minute 4). In contrast, the *Perf.* approach was able to scale up resources quicker in the *IoT Box* and also has a much lower RTT. The *HLPA* has the best results in terms of RTT and network latency, followed closely by the performance-oriented pressure-based approach (*Perf.*). While *HLPA* had the best performance, the resource usage results showed that it randomly tore down replicas across all clusters. Thus, it does not adapt efficiently to user mobility as pressure-based approaches.

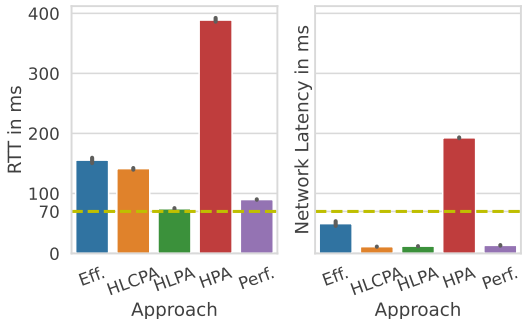



Figure 3.10: P90 RTT and network latency over all experiments

#### 3.1.7.3 Network traffic

In terms of cost, the communication between clusters is the most expensive and therefore is examined in the following. The network latency (see Figure 3.10) of *HPA* is the highest because only the initial replica in the *Cloud* and one in the *Cloudlet* were running, leading to many requests being forwarded to the *Cloud*. *HPA* also has the highest number of inter-network requests (5542). The *Eff.* approach also experienced high network latency caused by the relatively slow resource migration but was able to select clusters deterministically. The average number of inter-network requests is: *Eff.* (1783), *Perf.* (426), *HLPA* (300) and *HPA* (5542). The *HLPA* can only maintain the low network latency because of deploying as many replicas as possible. Due to the testbed's size, it is enough to serve the workload adequately. Our approaches show that mobility-aware scaling and placement reduce network latency and cost. These results also prove the effectiveness of our network-aware routing scheme used throughout the experiments. Even though function replicas are running across the system, inter-network traffic is kept to a minimum.

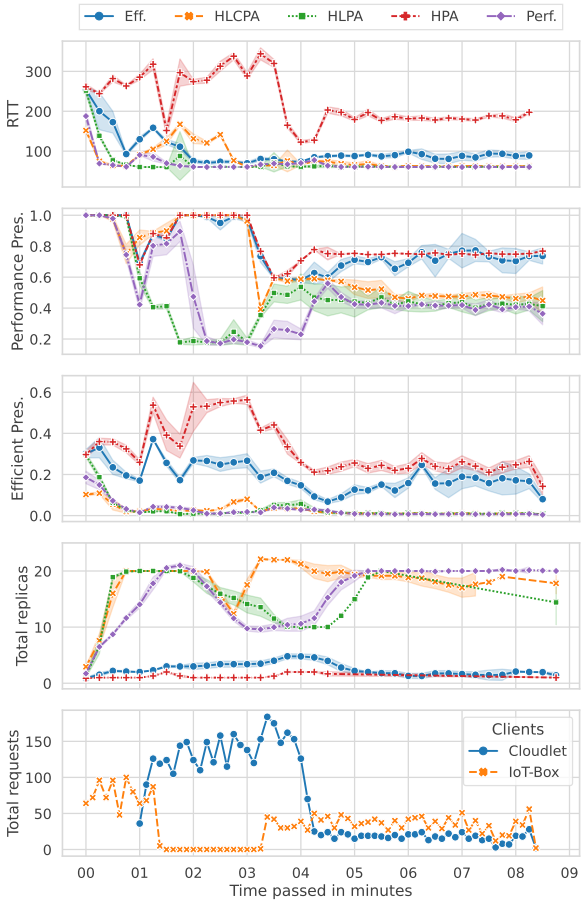



Figure 3.11: Metrics across all experiments.

#### 3.1.7.4 Pressure

In the following, we discuss the different pressure formulations in terms of interpretability and offer a detailed comparison between the approaches. Figure 3.11 shows various metrics across evaluation scenarios and the pressure in relation to other metrics concisely. The figure shows from top to bottom: the RTT, the performance-oriented pressure ($p^{\text{perf}}$), the efficient-based pressure ($p^{\text{eff}}$), total number of replicas, and the request pattern for each approach across the experiments (i.e., the total number of requests sent from that compute unit). Note that the shown values are aggregated over the whole system and contain values across clusters. Figure 3.9 shows that metrics vary heavily between clusters. First, the number of total replicas and the RTT highlight differences between the two approaches (*Eff.* and *Perf.*). The *Eff.* approach tried to keep a balance between total replicas and RTT, while the *Performance* approach focused solely on performance and disregarded resource efficiency. The *Eff.* approach strikes a good balance between the number of running replicas and the performance. It can keep the pressure below the

used maximum threshold of 0.3 and has a low number of replicas running throughout the experiment. This shows that the pressure-based approach can minimize the number of replicas while maintaining good performance. The *Perf.* approach keeps a low number of replicas during the peak workload at around 4 minutes, whereas the *HLCPA* uses up all available resources. The *HLPA* approach reduces the number of replicas, but this happens in a seemingly random fashion. Looking at the *efficient pressure* plot, we can see that the *HPA* and *Eff.* approaches have the highest pressure throughout the experiments, and the *Eff.* approach can quickly reduce it. Other approaches have a very low *efficient pressure* due to the low CPU usage caused by deploying many replicas. We think a different resource pressure component might be better suited (i.e., the number of replicas). On the other hand, the *performance pressure* shows that the *HPA* and *Eff.* approaches violate the RTT constraints, while the other approaches keep it at around 0.5. This shows that we can use the pressure as a valid estimation for the system's performance and state. The reason lies in the RTT pressure component ($p^{RTT}$), as it quickly penalizes requests that take longer than the target. Both pressures show the inability of the default cloud-centric *HPA* approach to adapt the system experiencing high pressure across the experiments successfully. We conclude that each pressure definition can reach its respective goal and serve as a complex metric to evaluate the system's state. Further, results also showed that the dynamic function orchestration is mobility-aware and suitable for scenarios with moving users.

### 3.1.8 Summary of Mobility Awareness

Managing hybrid edge-cloud systems, where mobility, latency, and data throughput play an important role, is complex and, therefore, should be done autonomously by platform providers. Current operational mechanisms of cloud-centric serverless computing lack awareness of geo-distributed setups and the problem of fluctuating workload due to user mobility. We presented a pressure-based joint scaling and placement framework and a network-aware routing strategy. We evaluated a PoC of our framework using two different pressure formulations. They have shown deterministic behavior regarding the final system performance based on simple low-level metrics. The efficiency-oriented pressure reduced resource usage by up to 83%, while the performance-oriented pressure used the available resources to reduce the RTT by up to 92% to cloud-centric solutions. Showing that the pressure can act as a viable proxy that encapsulates cluster state and workload, minimizing communication costs towards a central management unit. We conclude that the pressure can efficiently guide function orchestration strategies and enable mobility-aware scaling and placement of serverless functions. The novel concept of pressure is highly extensible and can be adapted for various application types. Besides sophisticated local placement strategies, future work comprises proactive approaches based on historical pressure data and large-scale simulations.

## 3.2 Energy-aware Container Scheduler

Reducing the energy consumption of data centers is critical to meeting international climate goals and lowering operation costs. Container orchestration platforms can help counteract this trend by optimally placing applications across the infrastructure to increase resource utilization and reduce energy consumption. But platforms in use today are still energy-agnostic and do not offer any insights into energy consumption. In this section, we present a monitoring framework and a new modeling approach for resource usage in data centers. The model captures heterogeneous hardware and software and acts as input for a Graph Neural Network (GNN) to predict power consumption. Based on this model, we derive a set of container scheduling algorithms that opportunistically schedule applications based on the estimated energy impact of incoming containers. Our results show that the GNN-based prediction model is very accurate and achieves an average RMSE (Root Mean Square Error) of 7.5%. We have implemented a custom scheduler to demonstrate the benefits of using our prediction, and our scheduler can decrease energy consumption on average by 6.2% without any code changes for the application and without increasing workload completion time compared to the default Kubernetes scheduler.

### 3.2.1 Introduction

Data centers may be responsible for around 3-13% of global energy consumption by 2030 [AE15]. Therefore, platforms are called to reduce energy consumption and increase sustainability [ABC17, PSP+21]. Especially, container orchestration platforms are prevalent in today's cloud and edge-cloud platforms because they offer autonomous application and resource management for large-scale application deployments [RND23], [RGLT19]. These platforms facilitate the deployment in many application domains, such as AI and HPC [RPGT22, WNL19], but also build the foundation for other platform paradigms, such as serverless computing [RND23, NRF+22]. However, current container orchestration platforms, such as Kubernetes or Apache Mesos, lack autonomous energy-aware management strategies [CPK+19, RHD+23]. Therefore, it is imperative to develop more sustainable containerized platforms [PSP+21]. This study investigates energy-aware resource management in container orchestration platforms to reduce energy consumption. Specifically, we focus on an energy-aware container scheduler using power prediction to make data centers more sustainable, which has not been integrated yet into existing platforms and remains a relatively unstudied problem in serverless computing [ATCG22, CZW+22, PSP+21, RSK23]. Energy estimation of future applications can facilitate this development. To this end, we introduce an end-to-end scheduling approach that includes a monitoring system and a Graph Neural Network to estimate energy consumption, which our scheduling algorithms use. We introduce a novel graph model that allows us to perform *what-if* estimations. For example, we can estimate the energy consumption of a host if we schedule a specific container onto it using historical data. Based on that, we implement three energy-aware container scheduling algorithms that use this prediction in a real-world setup. Therefore, the algorithms opportunistically

estimate the energy impact of applications and base their decisions on this. However, simply estimating the average power consumption of an application to estimate the total energy consumption may also not be accurate, as resource usage varies over time. Our approach solves these challenges using a flexible graph representation. The graph-based resource usage representation allows the combination of arbitrary applications and hosts, accounts for different resource phases throughout the application's lifecycle, and supports heterogeneous systems. However, integrating a power-prediction model into container-based platforms poses several challenges. While tools exist to energy monitoring tools exist, they vary from platform to platform, making energy estimation a universally applicable approach to estimating energy consumption. Our energy estimation builds the foundation of our energy-aware container scheduler, but it can also facilitate other strategies to reduce a data center's power consumption. For example, estimating the power consumption of the hardware accelerators for specific models before deployment can help select more efficient hardware [CZW+22], cooling [MMBD20] and facilitate the development of carbon-aware strategies.

Our contributions in this section are as follows:

- A novel graph model for hosts and a Graph Neural Network to predict the power consumption of hosts in container orchestration platforms. While our custom dataset is comprised of CPU-focused profiling experiments, the graph model can include arbitrary accelerators.
- The integration of the power prediction into Kubernetes which builds on commonly used open-source projects for monitoring resource usage.
- An energy-aware scheduling algorithm based on our power prediction and integration with Kubernetes.

Our results show that the model can account for existing accelerators and has a mean RMSE of 7.5%. On an on-premise cluster, the scheduler evaluation includes hosts with similar hardware specifications but observes different power consumption due to a GPU on one. Moreover, the results show that our energy-aware scheduler can reduce energy consumption by 6.2% while maintaining good performance, demonstrating the viability of our approach in real-world data center management activities.

The remaining work is structured as follows. In Section 3.2.2, we discuss related work, and in Section 3.2.3, we explain our approach to solving the energy prediction for a workload, the required monitoring to support the prediction, and energy-aware scheduler algorithms. In Section 3.2.4, we talk about the evaluation methodology for generating the training dataset and a use case for our prediction in a Kubernetes-based custom scheduler, and in Section 3.2.5, we present and discuss the results of our experiments in multiple scenarios. In Section 3.2.6, we summarize our work's next steps and conclude in Section 3.2.7.

### 3.2.2 Related Work

While energy awareness in cloud applications and resource management has been extensively studied [BSK+22, FLC+21, FLR+22, NMS+22], limited work has been done in exploring energy awareness for container orchestration [TMG+21, RSK23, PSP+21, Sha23, WJM+21]. To this end, we review work spanning from carbon- and energy-aware serverless and container-based systems to power-consumption prediction and the use of GNNs in resource management.

#### 3.2.2.1 Carbon- and Energy-Aware Approaches

We consider carbon-aware approaches related to as we present an AI model to predict power consumption, which can be easily combined with real-time carbon emission intensities (CEI) to build carbon-aware system management tools. Real-time CEI updates are publicly available through different third-party websites, such as WattTime[8] and electricityMaps[9]. Hanfy et al. [HLB+23] present CarbonScaler, a greedy algorithm for carbon-aware scaling that has been integrated into Kubernetes and evaluated using machine learning training and MPI jobs. While our work does not consider carbon emissions, the power consumption estimation we propose can facilitate future carbon-aware strategies. Moreover, our power consumption estimation also differentiates us to other scheduling approaches. By offloading applications to other nodes, Aslanpour et al. [ATCG22] explore energy-aware serverless edge computing. Rastegar et al. [RSK23] minimize CPU utilization and introduce an energy model tied to CPU utilization. In contrast, our power-consumption prediction and the scheduler we built atop can model different resources such as CPU, I/O, and hardware accelerators. Caspart et al. [CZW+22] explore energy consumption of AI workloads on heterogeneous nodes, highlighting the need for models that predict based on resource usage and the high impact of hardware accelerators on power consumption. Our work aims to address this issue by proposing a flexible model for power consumption prediction.

#### 3.2.2.2 Power Consumption Prediction

Power consumption predictions have been explored from various perspectives, such as container-based systems and general-purpose predictions. Ou et al. [OJZR23] present a random-tree-based power consumption prediction in container-based systems. They highlight the importance of container-level predictions, which we implicitly model by estimating the power consumption of the host if a certain container were to run on it. Moreover, our approach follows a fine-grained power consumption prediction model that allows us to account for different resource phases throughout the application's lifecycle. Other studies have presented linear consumption models for heterogeneous servers [ZLQZ13], reviewed energy consumption models for data centers [ABA21], and surveyed different models [DWF15]. Others explore power consumption estimation using

[8] https://www.watttime.org/
[9] https://www.electricitymaps.com/

different resource usage characteristics, such as I/O-based features [ZAL+17, KTIB22] However, the most significant differences with our approach are the use of a graph to model a host, the use of a GNN for predictions, and the inherently flexible definition of heterogeneous hosts with which intrinsic relationships can be captured in contrast to other machine learning methods [WPC+21].

#### 3.2.2.3 GNNs in Resource Management

Next, we discuss related work that uses GNNs for scheduling or resource management strategies. Tulli et al. [TGX+22] use a GNN to predict the Quality of Service (QoS) in scheduling, given a graph of containers and hosts while considering different factors, such as temperature. The main difference between our approach lies in the graph model and the task we solve. We also model hosts at a much finer level. Moreover, our GNN predicts power consumption given an application placement, while theirs predicts QoS. Tam et al. [TLX+23] also use GNNs to predict the performance of applications. Specifically, they model microservice graphs as input for GNNs to predict end-to-end latency.

### 3.2.3 Approach

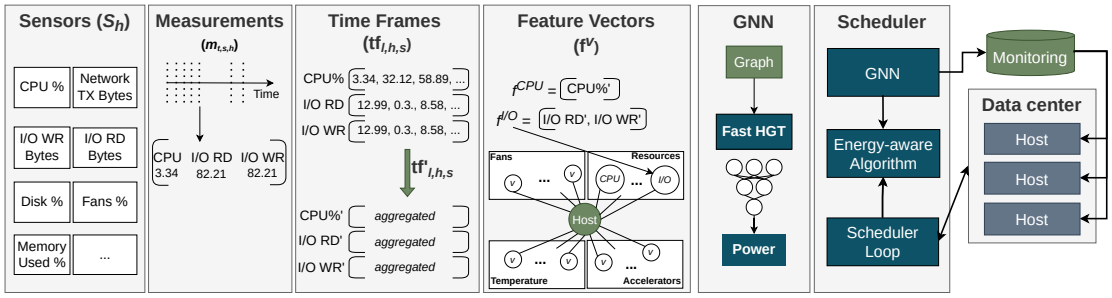



Figure 3.12: GNN-based Energy-Aware Scheduling

Figure 3.12 highlights our approach of building an end-to-end framework for energy-aware scheduling. We start by explaining the monitoring setup and graph model definition.

#### 3.2.3.1 Monitoring & Graph Definition

The monitoring infrastructure captures telemetry data for all hosts $h \in H$ in a data center. Hosts expose a set of sensors $S_h$, each reporting back a single measurement (number) every second, which we denote as $m_{t,s,h}$, where $t$ is the timestamp, $s$ the sensor and $h$ the host. We divide measurements into two types, counter, and gauge, whereas counter measurements can only increase (e.g., CPU time spent) while gauge ones can vary (e.g., RAM usage). Based on this monitoring setup, we now explain how to preprocess the measurements into graphs that act as input for our GNN. We create time frames that group $n$ consecutive measurements per sensor and host. Throughout our work, we set $n$ to 10, *i.e.,* each frame contains ten seconds of measurements. We denote each time frame as $tf_{l,h,s}$, where $l$ is the index of the time frame created by dividing the

original measurements $m_{t,s,h}$ of sensor $s$ on host $h$ by $L$, the number of time frames. The measurements in $tf_{l,h,s}$ are aggregated based on their type. In the case of a counter measurement, we calculate the total amount of work done within a time frame (e.g., the sum of differences between measurements). For gauge measurements, we calculate the arithmetic mean value of all measurements in $tf_{l,h,s}$, resulting in $tf'_{l,h,s}$. Next, we can create $L$ heterogeneous graphs for each host, each capturing the resource usage of one time frame. A simple way of understanding this step is by thinking of each graph containing the aggregated time frames of each sensor of one host. This approach allows the capturing of varying resource usage over time, and granularity, *i.e.*, the size of time frames is flexible.

A heterogeneous graph (HG) is defined as $G = \{V, E\}$. $V$ represents the node set and $E$ the set of edges. Each node and edge belong to a node ($N$) or edge type ($R$). In our implementation, node types represent different resources, each being monitored by one or multiple sensors, and edge types represent semantic or physical connectivity between them, where:

$$\begin{aligned} N = \{&\mathit{Host}, \mathit{CPU}, \mathit{Memory}, \mathit{Disk}, \mathit{Network}, \\ &\mathit{Pressure}, \mathit{GPU}, \mathit{Fan}, \mathit{Temperature}, \mathit{FPGA}, \mathit{SmartNIC}\} \end{aligned} \tag{3.15}$$

$$\begin{aligned} R = \{&\mathit{Host\text{-}CPU}, \mathit{Host\text{-}Memory}, \mathit{Host\text{-}Disk}, \mathit{Host\text{-}Network}, \\ &\mathit{Host\text{-}GPU}, \mathit{Host\text{-}Pressure}, \mathit{Host\text{-}Fan}, \mathit{Host\text{-}Temperature}, \\ &\mathit{Host\text{-}Pressure}, \mathit{Temperature\text{-}Memory}, \mathit{Temperature\text{-}CPU}\} \end{aligned} \tag{3.16}$$

All sensors are measured on a node level, and we exclude container-level metrics as we have seen that generalization suffers from modeling each container as a node in the graph. Each node $v$ in a graph is associated with a feature vector $f^v$ that only contains numerical values and whose length can differ across node types. The feature vector's values can be static (e.g., GPU model) or dynamic (e.g., current GPU frequency), for which we introduced the preprocessing (i.e., $tf'_{l,h,s}$). The unweighted and undirected edges connect each resource type with the *Host* node, and *Temperature* nodes are connected with resource types they report the temperature on (e.g., *CPU*).

#### 3.2.3.2 Application Signature

The graph model also allows us to model the resource usage of applications, which builds the base of our *what-if* estimations. However, only a subset of sensors can be isolated from the host's resource usage. For example, we can measure the CPU usage per application, not the temperature or fan increase it causes. To this end, we use sensors that capture the resource usage at the container level for the following node types: *CPU*, *Memory*, *Disk* and *Network* using cAdvisor, a popular monitoring tool for container-level metrics. We introduce the application signature $as_{a,h} \in \mathit{AppSig}$, ($a$ being the application), that represents the isolated resource usage of an application $a$ on a host $h$, in the form of

graphs. Next, we introduce a function $merge : (G, G) \rightarrow G$ that combines a graph of an application signature and a graph representing a host to one graph. The function adds the values of the counter type measurements together and calculates the arithmetic mean for gauge measurements. The application signatures allow us to perform lightweight *what-if* simulations to estimate the power consumption of a host based on its current system resource usage and an arbitrary application. Because *as* contains a set of graphs, representing the resource usage of an application over time, our model is aware of different resource phases during an application's execution.

#### 3.2.3.3 Graph Neural Network Architecture

The resulting graphs are used as input for our GNN. Its architecture builds on an existing one presented by Hu et al. [HDWS20]. We use the *fast* implementation of the HGT Convolutional layer, which is stacked two times, and then use three global pooling methods (i.e., mean, max, and add) to create a 1-dimensional vector that is passed through four linear layers, which output the final vector containing the power consumption predictions (min, max, avg) for a given graph (time frame).

#### 3.2.3.4 Opportunistic Energy-aware Scheduling Algorithms

We propose three scheduler variants that differ in their decision-making, but all use the GNN for power-consumption prediction. Specifically, we take an opportunistic approach by viewing each container to schedule individually and its impact on energy consumption, in contrast to others that find an optimal placement given a set of containers.

**Base Algorithm (*GNN*)** The base energy-aware scheduling algorithm fetches each host's current state and adds the signature of the container to schedule. To accomplish this, we build graphs for each host and combine them with the container's application signature using the *merge* function. The output of this is a sequence of graphs that represent the estimated resource usage of the container running on the host. Next, we use the GNN to estimate average power consumption in Watts per graph, which we use to calculate the total energy consumption. The last step finds the host with the minimum estimated total energy consumption.

**Application-Aware (*GNN-Aware*)** We also propose an application-aware algorithm that works similarly to the one just described but considers the future resource usage of applications concurrently running. For example, the *base* algorithm fetches the current resource usage of each host and then adds on top the application to schedule. However, applications already running on the host might end soon or run for a long time—the system resource usage varies. It requires the assumption that we have the application signature for each application running, which allows us to estimate the remaining run time and dynamically change the resource usage of each prediction depending on which point of processing they are at.

**Packing (*GNN-Packing)*** The last scheduling algorithm integrates with both of the previously introduced algorithms. The *Packing* strategy modifies the selection algorithm by considering the host with the second lowest estimated total energy consumption. We introduce a threshold with which we determine to take this instead of the lowest. We select it if the second one's energy consumption is within the range of the first one by adding the threshold on top. We set the threshold to 5% for all experiments. We do this to prefer nodes with more applications running to further dampen the power consumption increase.

#### 3.2.3.5 Implementation

After explaining the envisioned prediction and scheduler theoretically, we describe key aspects of our implementation.

**Monitoring** Our monitoring system is based on Prometheus, a time-series database that monitors the infrastructure by following a pull-based monitoring approach. This means that on each host, multiple exporter applications expose a range of metrics via HTTP. Prometheus scrapes each exporter every second and stores it in its database. We deploy cAdvisor[10] to monitor containers, NodeExporter[11] to monitor host metrics and the Nvidia DCGM[12] exporter to monitor GPU usage, as well as two custom exporters for HPE iLO [13] and Xilinc's xbutil [14] tool to monitor FPGAs. We observe 16 dynamic features from cAdvisor, 37 dynamic and static metrics from NodeExporter, and fan and temperature from iLO. While we did not run workload using hardware accelerators, we are still monitoring it and integrate it into our graph during preprocessing. Namely, 22 metrics for GPUs, six for FPGAs and three for SmartNICs. We must include them because it allows the model to learn the individual power consumption of very similar hosts (i.e., same CPU and memory) but have a different idle power due to equipped hardware accelerators.

**Scheduler** The scheduler is implemented in Python and uses a Kubernetes client library to communicate with the API server. We use Python's Multiprocessing library to spawn multiple independent processes that cache monitoring data in memory to instantaneously access telemetry data required for inference and a scheduler loop that repeatedly takes new applications from the scheduler queue. Upon decision-making, the scheduler loop tells Kubernetes which host to start the application. The GNN is implemented in Python using PyTorch v2.0.1 [PGM+19] and uses the library PyTorch Geometric v2.3.1 [FL19] , which offers an implementation of the Heterogeneous Graph Transformer [HDWS20] network.

[10] https://github.com/google/cadvisor
[11] https://github.com/prometheus/node_exporter
[12] https://github.com/NVIDIA/DCGM
[13] https://www.hpe.com/us/en/hpe-integrated-lights-out-ilo.html
[14] https://xilinx.github.io/XRT/master/html/xbutil.html

| Stressor | Resource | *stres-ng* Arguments |
|---|---|---|
| cpu | CPU | cpu (1- 88) |
| memrate | Memory | memrate (1-4), vm-bytes (256M) |
| vm | Memory | vm (1-10), vm-bytes (5%-50%) |
| iomix | Disk I/O | iomix (1-4), iomix-bytes (256M) |

Table 3.4: Stressors and parameters for profiling

### 3.2.4 Evaluation

Our evaluation consists of two parts: training and validating our GNN for power consumption and evaluating the energy-consumption reduction when employing our energy-aware scheduler. The cluster we used consists of three nodes, all equipped with dual-socket AMD 7443 CPUs. Two nodes were configured with lower TDP (Thermal Design Power), while one was configured for full TDP. Each node was also equipped with 256 GB of memory, and one node was also equipped with an Nvidia A100X GPU.

#### 3.2.4.1 Training Dataset

To generate training data, we use different *stress-ng* stressors to stress certain parts of the system. This approach allows us to generate applications with different resource usage phases that mimic the behavior of real-world applications. Benchmark suites like FunctionBench [KL19] offer similar applications (e.g., matrix multiplication, disk I/O), but we want to create a set of applications that focus on different resources so that a wide range of different resource usage is exposed. Table 3.4 shows each hardware resource's stressors, parameters, and value ranges. Each stressor runs for 100 seconds, and in total, there are 72 stress tests for each host. We split the dataset using a 5-fold approach and show the Root Mean Squared Error (RMSE) in the results for each run. The k-fold split is commonly used in machine learning [WY20]. This means we run 5 training-test splits, where one training dataset contains 4/5 of the complete dataset. The profiling data of different hosts is included in training and test sets. In the future, other resources, such as FPGA, should be stressed to train the model on a wider resource usage spectrum.

To train the GNN, we use *Adam Optimizer* with a *learning rate* of *1e-5*, 125 *epochs*, and *batch size* of 5.

#### 3.2.4.2 Scheduler evaluation

Our scheduler evaluation consists of five different approaches: *GNN*, *GNN-Aware*, *GNN-Packing*, *Spreading* and *Packing*. *GNN*, *GNN-Aware*, and *GNN-Packing* are using the algorithms we presented using the power consumption predictions. The *Spreading* scheduling strategy balances containers, i.e., the CPU cores and memory used, as the default Kubernetes scheduler does. The *Packing* strategy always chooses the node with the highest number of CPU cores used. The number of CPU cores requested and used are

| Benchmark | CPU | RAM | Parameters | Duration |
|---|---|---|---|---|
| LavaMD | 4 | 4 GB | -cores 4 -boxes1d 50 | ∼100s |
| LavaMD | 8 | 8 GB | -cores 8 -boxes1d 50 | ∼50s |
| Leukocyte | 4 | 4 GB | 40 40 testfile.avi | ∼30s |
| Leukocyte | 8 | 8 GB | 50 40 testfile.avi | ∼18s |
| srad_v1 | 4 | 4 GB | 1000 0.5 5020 458 4 | ∼12s |
| srad_v1 | 8 | 8 GB | 0.5 5020 458 4 | ∼10s |

Table 3.5: Rodinia applications for scheduler evaluation

pre-determined, and the applications are using the Rodinia Benchmark Suite [CBM+09]. All applications immediately run the computation after starting and report the results (i.e., execution duration) back for post-experiment analysis.

Table 3.5 shows all configurations, the requests, and limits for the number of cores and memory for Kubernetes resource allocation and the average execution duration. We define two scenarios based on a sinusoidal pattern. The short-running one submits 72 containers in 209 seconds, while the longer-running one submits 128 containers in 234 seconds.

### 3.2.5 Results

In this section, we present the results of our evaluation. We start by showing the results of the GNN training and, afterward, the scheduler evaluation results.

#### 3.2.5.1 Graph Neural Network

The mean RMSE of our AI model from the 5-fold evaluation is around 7.5%, and the standard deviation is around 3%. These encouraging results can be improved further by considering a more complex network architecture and the inclusion of additional sensors. The selection of sensors was mainly motivated by the use case of implementing a scheduler based on the ability to perform *what-if* estimations to predict the impact of an application on power consumption.

We also see that the model can differentiate between similar systems observing different static power consumption. For example, one host in the cluster is equipped with an Nvidia A100X that increases the static power consumption by roughly 50 Watts (∼20% increase). The model can differentiate between these because we model the equipped GPU in the input.

In addition, we performed experiments with different GNN architectures, using LSTM-based approaches and different combinations of the presented model architecture (i.e., by varying the number of HGT layers). Our results indicate that there is a performance-accuracy trade-off to be made. Specifically, with an increasingly complex model, the accuracy increases, but the performance suffers (i.e., inference speed decreases). We saw

| Scheduler | Wh | Makespan (s) | EtS (J) | Mean Performance (factor) | Std. Performance (%) | Mean Power Consumption (W) |
|---|---|---|---|---|---|---|
| *GNN* | 89.27±0.3 | 260±0.71 | 105206±331.63 | **1.28±0.021** | 22.2±1.03 | 403±1.55 |
| *GNN-Aware* | 90.05±0.11 | 259±0.71 | 104967±915.7 | 1.29±0.022 | 20.13±0.12 | 403±2.95 |
| *GNN-Packing* | **88.84±0.39** | 260±0.71 | 104250±782.06 | 1.31±0.01 | 20.72±0.45 | **398±3.95** |
| *Packing* | 87.44±0.39 | 261±2.83 | 102039±221.32 | **1.32±0.004** | 18.44±0.43 | 390±2.82 |
| *Spreading* | **90.69±0.35** | 251±2.12 | 106854±806.81 | 1.27±0.033 | 23.26±1.89 | **424±0.69** |

Table 3.6: Results of individual schedulers for the short experiment

| Scheduler | Wh | Makespan (s) | EtS (J) | Mean Performance (factor) | Std. Performance (%) | Mean Power Consumption (W) |
|---|---|---|---|---|---|---|
| *GNN* | 141.69±1.07 | 336±1.41 | 169078±341.53 | 1.37±0.004 | 22.35±0.47 | **502±2.83** |
| *GNN-Aware* | 140.75±0.09 | **332±2.83** | 168130±1623.52 | **1.34±0.008** | 20.98±0.92 | 506±0.78 |
| *GNN-Packing* | **139.17±0.79** | 332±0.71 | 168039±439.82 | 1.36±0.016 | 23.78±0.61 | 504±3.24 |
| *Packing* | 141.14±1.29 | **339±0.71** | 168273±178.19 | **1.39±0.012** | 21.88±0.84 | 495±0.7 |
| *Spreading* | **148.38±2.28** | 335±2.12 | 178960±87.68 | 1.20±0.003 | 18.54±1.56 | **532±3.25** |

Table 3.7: Results of individual schedulers for long experiment

2x changes in terms of time to predict but a marginal increase in accuracy (i.e., a 2% decrease of RMSE). Based on that, we chose the non-LSTM approach.

#### 3.2.5.2 Energy-Aware Scheduler

We discuss the schedulers by showing the summarized results, looking at the power consumption over time, and investigating the overhead our scheduler imposes.

Table 3.6 and Table 3.7 show the results for the short-running and long-running, respectively. The *Packing* scheduler achieved the lowest total energy consumption (Wh) and lowest EtS over all others in the short-running experiment. The *GNN-Packing* algorithm is close and only has slightly higher energy consumption, *i.e.,* by 1.6%. The Energy-to-Solution (EtS) metric shows the workload's total energy consumption [WAP+14]. Whereas the *GNN-Packing* was able to reduce total energy consumption and EtS in the long-running one. In all cases, the *Spreading* scheduler performed worse and has, in comparison to the *GNN-Packing* scheduler, an increase of 6.2% Wh and consumes 5.27% W per second more on average in the long-running one. In terms of makespan, the *Packing* scheduler performed worse, making our GNN-based approaches a viable solution to balance between the two goals. Interestingly, the performance, indicated by the *Mean Performance factor* that shows the increase to the average duration, is similar across the different approaches, even though the *Spreading* spread the workload across nodes. The reason for this is that the workload submits enough jobs to saturate all hosts, which makes the sharp difference in Watts (i.e., a decrease of ∼5%) very good.

**Power Draw over Time** Figure 3.13 shows the power over time for the long-running experiment setup. It is clearly visible that the *Packing* approach maintained a low power consumption. We also see a sharp increase in power consumption for the *Spreading* because it chooses all nodes equally and, therefore, will also pick nodes with higher power consumption, which results in higher energy consumption. Our model can differentiate between nodes and pick the low-energy ones. While *Packing* did not select this node due to the implementation, our *GNN* could predict that the other nodes are more

energy-efficient and select those on purpose. This circumstance, including the fact that *Packing* will have a very high performance degradation, due to the strategy of filling up nodes, proves that our algorithms can strike a good balance between *Packing* and *Spreading*.

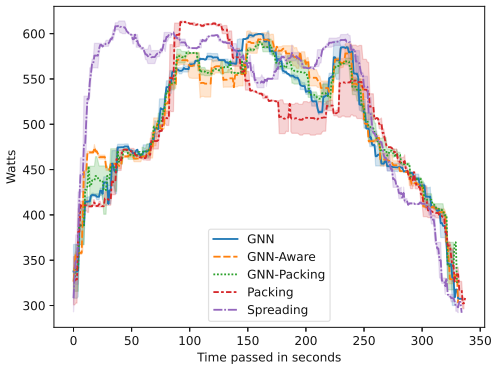



Figure 3.13: Power over Time (long-running experiment)

**Scheduling Overhead** The last aspect of our evaluation is the scheduling overhead caused by using the *GNN*. For this, we measured the time from preprocessing to post-processing the selection algorithm. The mean overhead is around 250ms, which can be improved upon by using GPUs for inference. It also shows the potential of our approach because we were still able to reduce makespan and energy consumption.

### 3.2.6 Future Work and Limitations

An important task to alleviate this proof of concept to a production-ready solution is considering more edge cases than we covered. Specifically, what should the scheduler do in case of the arrival of new applications? If the application has never been run on any host, we must resort to a default scheduling strategy (i.e., *Packing*). Once the application has been run on a node, we can use its isolated monitoring from cAdvisor. This entails implementing continuous learning using MLOps [KKH23] to fine-tune the system during run time. While our model learns the power consumption of the host instead of on a per-application base, we still need to incorporate newly available data. Lastly, to become a viable production system scheduler, scalability and energy efficiency experiments have to be made to guarantee energy savings concerning the scheduling overhead. We also think the GNN can facilitate the creation of a clustering method to identify similar workloads [TPPM23]. This way, the GNN-based scheduler can take historical data of similar workloads for its decision. Moreover, resource interference-aware scheduling methods can be used by taking advantage of this clustering approach by avoiding placing applications from the same cluster together. Reducing energy consumption is not an exclusive goal of cloud data centers, but also plays a vital role in the edge-cloud continuum, where devices might be battery-powered [RCP23]. The ability to support heterogeneous devices is, in this context, even more important. Our initial graph model can be extended with

domain-expert knowledge and models of complex relationships between components to increase its accuracy. While in this work, we only model hardware accelerators but did not run profiling experiments, we plan to run more Rodinia benchmarks [CBM+09, CFL+18] on AMD FPGAs and GPUs to extend the applicability of our model.

### 3.2.7 Summary of Energy-Awareness

This section introduces a novel model to represent hosts and host components in a data center as graphs. Atop the graph, we implement and train a Graph Neural Network to predict the power consumption over a specific time frame. The model's mean RMSE is $\approx 7.5\%$ and can facilitate the development of energy-aware data center management strategies differently. We showcase the feasibility of using our GNN in scheduling containers, where we can reduce energy consumption by an average of 6.2% while also completing the workload faster than a scheduler working like the default Kubernetes scheduler. However, improving energy efficiency is only a partial solution as it has become clear that we should focus on reducing carbon emissions as well. Therefore, deploying only energy-aware strategies might not be enough, and carbon-aware ones are required. Our approach to predicting energy consumption can facilitate building new carbon-aware strategies.

## 3.3 Summary

We have presented two novel orchestration approaches that aid Serverless Edge Computing. One focuses on the mobility of users, the other on energy consumption in data centers. Both approaches have achieved their respective goals, but we have also encountered challenges in both. Namely, finding the right parameters. In Section 3.1, the mobility-aware approach, we had to find thresholds to guide the autoscaler. In Section 3.2, the energy-aware container scheduler, we had to execute profiling experiments to generate training datasets for our GNN.

We recognize that the edge-cloud continuum is heterogeneous and dynamic, meaning that finding parameters is a challenge we will continuously face when using Serverless Edge Computing platforms. Therefore, we want to build a self-adaptive platform that tackles this challenge, and for which we introduce two tools that act as a foundation for our vision in the following Chapter.

### Future Work

Section 3.1 has focused on inference workloads that are short-lived and require infrastructure to enable high-throughput processing. However, Edge Intelligence also entails much longer-running applications, such as model training. Model training introduces challenges around data movement and puts greater emphasis on planning execution ahead of time. Specifically, training requires the application to fetch the model weights to train and the dataset, both of which can range from several hundred megabytes to

several gigabytes. Downloading and uploading that much data poses a challenge in network-constrained environments such as the edge-cloud continuum. Network-aware and data-aware schedulers can avoid network congestion.

Besides, training can take minutes to hours or days, which makes long-term decisions much more important and requires schedulers to plan ahead and consider co-existing functions. Therefore, future work of the contributions presented in Section 5.1 and Section 5.2 is exploring the aid of network-aware schedulers that not only focus on function placement with regards to data movements but also for planning ahead over much longer time horizons than the autoscaler in this work has so far considered.

Similarly, in Section 3.2, we have focused on longer running applications, but focused on cloud computing and the energy consumption. The GNN presented only models a single host; future work would expand on this and create larger graphs that encompass data centers or even the edge-cloud continuum. The use case of the GNN can also be extended to serve as a guidance mechanism for future applications based on their resource usage, i.e., by estimating the energy consumption ahead of deployment during development.

CHAPTER 4

# Testing And Simulating Serverless Edge Computing

The following chapter introduces our testing framework for Serverless Edge platforms in Section 4.1, first presented in [RRP+22]. The framework offers monitoring capabilities and a programming API to manage functions. Section 4.2 presents *faas-sim*, our simulator for Serverless Edge Computing, presented in [RRFD23][1]. The simulator and testing framework share the entity model to a degree, which allows us to replicate the real-world system in the simulation. This facilitates a simulation-driven self-adaptive platform that we evaluate in Chapter 5.

## 4.1 End-to-End Evaluation Framework for Serverless Edge Platforms

This section presents a framework for defining, performing, and analyzing distributed load testing experiments for benchmarking edge-cloud clusters. This end-to-end workflow helps researchers build reproducible environments to evaluate cluster management techniques. Our implementation extends the open-source tool Galileo by adding support for distributed execution on Kubernetes clusters, additional system monitoring instruments, as well as out-of-the-box experiment workloads. We focus on providing tools that run across popular CPU architectures and provide a set of representative workloads, such as edge AI functions. We demonstrate our framework's capabilities in a set of experiments based on use cases commonly found in edge computing systems research.

[1] A basic version of the simulator has been presented in the dissertation of Rausch [Rau21]. Therefore, parts of it have been reused for the paper publication and this thesis. However, the simulator has been substantially extended since the publication of [Rau21], and [RRFD23] was its first peer-reviewed publication. In addition, we explicitly mark parts that were first presented in [Rau21].

Additionally, we show that the resource usage of our system is minimal and that it can run on resource-constrained devices.

### 4.1.1 Introduction

To evaluate edge-cloud cluster management techniques, such as orchestration strategies used for Serverless Edge Computing, researchers often rely on building testbeds that are tailored to evaluate the particular technique [RCLN20, TPTE21, SFEG21, KF22]. In the absence of publicly available edge infrastructure, testbeds are, next to simulations [AJH+21] and emulations [MGG+17], the only way to evaluate systems approaches. However, there is currently no generally accepted way of creating such testbeds or representative benchmark workloads. This makes it hard for other researchers to reproduce results and prevents the community from comparing works easily. Reproducible experiments for edge-cloud clusters aid in evaluating cluster management techniques and include the following steps: (1) setting up the testbed, (2) generating workload patterns, (3) managing application deployments, (4) orchestrating experiments, (5) instrumentation and monitoring, (6) analyzing experiment data. Performing these steps manually is error-prone and impedes reproducibility [JNW+19]. An experiment and analytics framework allows researchers to evaluate their systems approaches while opening up other possibilities crucial to developing and testing cluster management techniques. For example, some cluster management techniques are based on optimizations that rely on historical data [RRD21]. An experiment framework gathers data in a streamlined way, significantly reduces manual steps, and allows scaling in terms of applications and cluster sizes.

Reducing performance degradation caused by the interference of co-locating monitoring applications on the same device is challenging for cloud deployments and more so for edge-cloud clusters [JGJ+18] that are typically heterogeneous and resource-constrained. Hence, we require hardware architecture agnostic, lightweight, and non-intrusive instrumentation tools to measure resource usage and power consumption. Further, trace-driven simulations require real-world data to run realistic simulations [RHM20]. An experiment framework helps record this data and enables the community to perform complex simulations based on real-world data.

Based on the abovementioned observations and requirements, we conclude that such a framework should provide scalable and reproducible experiments, simple configuration of workload and applications, fine-grained monitoring telemetry, storage for analysis data, and standardised analysis techniques.

To this end, we built *Galileo* [RRPD19], a distributed load testing framework for edge computing environments. However, while Galileo allows the definition and execution of experiments, it cannot operationalize those experiments on existing cluster testbeds nor provide a streamlined way to instrument and collect monitoring data from such clusters.

In this section, we present an end-to-end experiment framework based on an extension of Galileo and the combination of different components. We motivate our work by

introducing different use cases typically encountered in evaluating cluster management techniques, demonstrating our framework's efficacy by executing them, and showcasing the results.

Edge Run[2] is a collection of projects aimed at helping developers create, monitor, and test applications for the edge-cloud continuum. Part of the Edge Run project is Galileo[3], a distributed load testing framework [RRPD19]. Galileo can start clients, generate configurable requests, and store trace and telemetry data in a SQL database, which is unfit for high volumes of time-series data. The caveat of Galileo is that it requires a manual setup and does not offer any kind of integration with modern orchestration services such as Kubernetes. In this work, we introduce a framework that extends Galileo, solves these issues, and provides an analytic framework that completes the end-to-end framework. Further, we introduce components mimicking the characteristics of edge-cloud clusters.

Specifically, our contributions are as follows:

- An end-to-end experiment and analytics framework using a container orchestration service based on Galileo.
- An open-source repository of benchmark applications that can be used within our framework.

To evaluate and demonstrate the end-to-end process, we conduct experiments based on representative use cases.

### 4.1.2 Galileo experiments

This section describes the aim of our framework based on the experiments and applications we intend to support and the characteristics of edge-cloud clusters.

#### 4.1.2.1 Aim of the framework

The framework aims to provide tools to perform benchmarks on edge-cloud compute clusters. Benchmarks aim to profile single devices or evaluate cluster management techniques in scenario experiments. The edge-cloud continuum deploys different types of hardware; therefore, all framework components must run on various hardware architectures. Components are tailored to run from resource-constrained devices to off-the-shelf Virtual Machines hosted in the cloud. Particularly, monitoring components should not interfere with running applications and cause minimal overhead. Additionally, the framework uses a container orchestration service to automate the deployment of components and execution of experiments. Lastly, an integrated monitoring strategy is necessary

[2]https://github.com/edgerun
[3]https://github.com/edgerun/galileo

to provide tools for analyzing cluster management techniques. Based on the collected monitoring data, the framework offers features to analyze experiments. It supports commonly observed key performance indicators (i.e., resource usage, CPU consumption, scheduling behavior) out of the box. We present and evaluate in Section 4.1.4.2 use cases that include these indicators. To summarise, our end-to-end framework aims to support users in performing experiments from deployment to analyzing them.

#### 4.1.2.2 Galileo

*Galileo* [RRPD19] was created to develop and evaluate cluster management techniques using Symmetry, a custom cluster management system. Galileo's main tasks are to coordinate clients to generate requests as well as monitor and store monitoring data in a MySQL instance.

Based on the description of our view of a modern end-to-end experiment and analytics framework, *Galileo* requires adaptations and extensions. Specifically, we intend the integration into a modern container orchestration service such as Kubernetes. This allows us to deploy framework components into the cluster, reducing the manual steps required to run experiments. Further, we provide a representative set of ready-to-deploy applications (i.e., AI inference functions) to the community for evaluation purposes and as examples for creating custom applications. Additionally, we add to the SQL-based data storage the time-series database InfluxDBv2[4]. The SQL storage stores experiment metadata (i.e., start, end), and data gathered during runtime is stored in InfluxDB. The fine-grained monitoring tool, *telemd*, has also been extended since its first release and now supports Kubernetes pods as well as the recently released Pressure Stall Information (PSI)[5]. PSI is available for CPU, I/O, and network. It shows the time some or all processes have been waiting for a specific resource. We believe that using these new metrics can lead to many novel cluster management techniques.

#### 4.1.2.3 Experiments

The experimentation and analytic framework offer a streamlined approach for executing and measuring benchmarks. We consider two types of experiments in this section: *scenario* and *profiling*.

*Scenario* experiments are complex scenarios to perform resource management actions (i.e., scaling). These experiments allow users to specify multiple application instances on nodes. Request generation is based on workload profiles that *Galileo* clients execute.

*Profiling* experiments are similar to *scenarios* but offer reduced functionality and concentrate on executing one type of application on one node. Profiling experiments aim to rapidly test new applications to estimate resource usage and performance across devices.

[4]https://www.influxdata.com/
[5]https://www.kernel.org/doc/html/latest/accounting/psi.html

#### 4.1.2.4 Applications

Our framework supports any containerized application that exposes an HTTP endpoint. Though our primary focus while developing it was using OpenFaaS[6] based functions on different hardware. Therefore, we offer ready-to-use functions that support all common architectures (i.e,. amd64, arm64v8 and arm32v7). The functions use the OpenFaaS watchdog[7], which uses a Go-based proxy and calls an internal Python Flask web server to invoke the function. At the time of writing, our function repository[8] primarily consists of AI inference functions. For example, the functions can perform object detection (i.e., gun, human, mask), object classification, and pose estimation.

Custom applications require implementing methods to call the relevant service and a Kubernetes Pod factory for the container image. This guarantees flexibility concerning future applications and requires minimal coding efforts. Section 4.1.3.3 presents details about the individual projects.

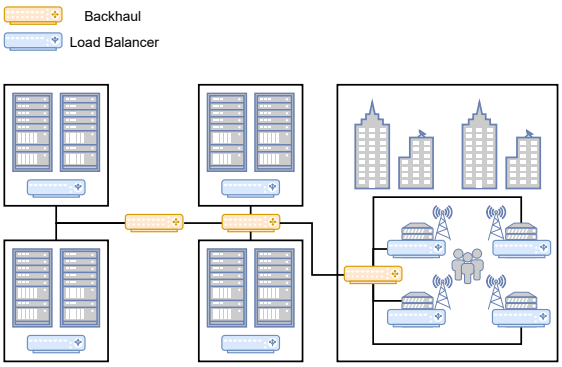



Figure 4.1: Edge-cloud system consisting of multiple clusters

#### 4.1.2.5 Edge-cloud testbeds

To ensure the reproducibility of experiments, we require tools for setting up testbeds. However, our vision of an edge-cloud system consists of multiple heterogeneous interconnected clusters with varying and adverse network conditions. Consequently, we cannot rely on existing work on cloud testbeds characterized by relative homogeneity of resources and static and optimal network conditions. Figure 4.1 depicts such an edge-cloud system with multiple clusters, where each is assigned a load balancer.

Cluster management techniques must consider user mobility since it is an important characteristic of edge-cloud systems [OZC18]. For example, clients move around the city and connect to radio towers, which act as the first entry point. Each load balancer can redirect requests to other clusters. Internet backhauls inter-connect all clusters but

[6]https://docs.openfaas.com/
[7]https://github.com/openfaas/of-watchdog
[8]https://github.com/edgerun/galileo-experiments-functions

introduce significant network latency[BPKP20]. Our architecture is comparable to works presented in [RCLN20] and [TPTE21], which evaluate cluster management techniques in geo-distributed systems using public cloud offerings. As mentioned, we use a container orchestration service to set up the cluster. Thus, we create deployment files that must be applied to the cluster once. This includes clients, monitoring agents, as well as load balancers, enabling users to set up our framework on their testbed. Network latency is emulated via Linux tools and mimics the behavior of geo-distributed edge-cloud clusters. Reproducibility is guaranteed through the usage of Linux tools and an orchestration service.

### 4.1.3 System

This section introduces our experimentation framework by describing the system architecture, explaining the experiment workflow, highlighting important implementation details, and presenting the testbed used to demonstrate our framework.

#### 4.1.3.1 Architecture

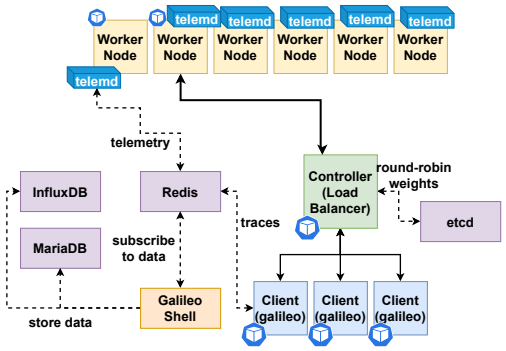



Figure 4.2: A high-level overview of the experiment setup

The architecture facilitates generating requests, hosting applications, distributing messages, and storing data. We use the lightweight edge-oriented K3s[9] Kubernetes distribution to host applications and clients. All nodes join the same cluster; while not intuitive at first, considering we want to create a testbed with multiple clusters, it avoids the overhead of setting up a cluster federation. We use labels to tag controllers, client nodes, and workers, which assign each node cluster. Each cluster has one controller node acting as a load balancer. WireGuard [Don17], a lean VPN, is used to form a homogeneous network and *tc*[10], the Linux network shaping tool, to introduce latency between clusters. Section 4.1.4.3 presents more details about the testbed setup. Figure 4.2 depicts the described system. Workers host applications, while client nodes run *Galileo* and wait for workload generation requests published via Redis. Clients send HTTP requests to the

[9] https://k3s.io/
[10] https://man7.org/linux/man-pages/man8/tc.8.html

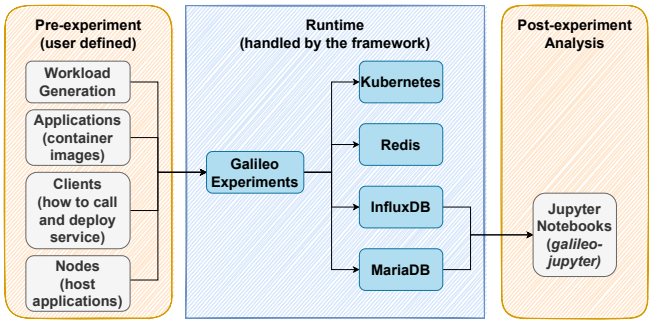



Figure 4.3: Experiment workflow

load balancer, which forwards them to application instances. The controller hosts the load balancer instance. We provide a Go-based load balancer (*go-load-balancer*) which implements a weighted round-robin based load balancing strategy. Though users can choose to use any other L7-layer load balancer (i.e., Traefik[11]). Our experimentation framework sets initial weights for load balancers. Users can modify weights via etcd[12], a key-value storage used by Kubernetes to store state. Each load balancer instance watches the etcd instance for changes and updates its internal weights accordingly. The *galileo shell* is the component that initiates the experiment and subscribes to Redis, and stores data in InfluxDB and MariaDB for post-experiment analysis.

#### 4.1.3.2 Experiment workflow

In the following, we explain how users can define and run experiments. Figure 4.3 splits the experiment into three phases: pre-experiment, runtime, and post-experiment. Users only have to interact during the pre and post-experiment phases, while our framework handles the system during the runtime phase. Users have to supply the following input: workload (i.e., a list of inter-arrival times), applications (i.e., container image), clients (i.e., implement a method to create the HTTP request), and nodes (i.e., the testbed to perform experiments on). In the pre-experiment phase, users can generate workloads. The framework supports two types of workloads: parameterized and profile-based. The parameterized approach requires three arguments: $n$, the number of requests, $ia$, the inter-arrival time, and the number of clients. Clients are *asynchronous* and send $n$ requests with a fixed inter-arrival time $ia$. Consequently, experiments will stop without waiting for responses. The profile-based variant expects an array that contains inter-arrival times, each array representing one client. For example, $[0.5, 1, 0.5]$ will send three requests: one after 0.5 seconds, another one after 1 second, and the last after another 0.5 seconds.

*Profiling* and *scenario* experiments, as described in Section 4.1.3.2, have different parameters. The former takes both types of workload, while the latter only supports the profile-based approach. Additionally, profiling experiments take the application container

[11]https://traefik.io/
[12]https://etcd.io/

image, the number of application instances, the host to profile, and the cluster clients that generate requests. In scenario experiments, users create a mapping between nodes and container images, including the number of instances. Further, scenario experiments can start clients in different clusters, making it possible to define complex scenarios (i.e., migrating users from one cluster to another).

After users start the framework and enter the runtime. The framework sets up load balancers, spawns applications, configures clients, and starts the *galileo shell* to record all telemetry and trace data. Finally, in the post-experiment phase, users can analyze the results using *galileo-jupyter*. We categorize analytics data into three main categories:

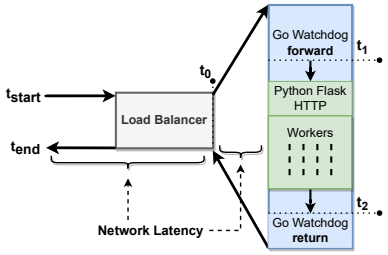



Figure 4.4: A detailed look into traces

metadata, traces, and resource usage. Metadata consists of an experiment ID, the creator, start, and end. Additionally, the framework also saves information about all nodes. This includes information detailed in the *telemd* repository and Kubernetes labels of each node and their allocatable resources. Figure 4.4 provides a detailed description of one trace. We record: the time the trace was created, the client sent it ($t_{start}$), it was forwarded from the load balancer ($t0$), it was received by the function $t_1$, returned $t_2$ and arrived back at the client $t_{end}$. Based on these timestamps, we can calculate and estimate further measures: round-trip-time (RTT, $t_{end} - t_{start}$), network latency between client and load balancer ($t_0 - t_1$), the network latency between load balancer and application instance ($t_1 - t_0$) and the execution time ($t_2 - t_1$). The execution time includes any queuing caused by the static number of worker threads the Python Flask server employs to handle requests. Functions can return the function execution time in the response body. Telemetry data consists of node-based resource usage (i.e., CPU usage) and per container (i.e., CPU time). We refer readers to look up the project's homepage for an exhaustive list of monitored resources[13].

In scenario experiments, users must deploy their own cluster management techniques. They can create and tear down Pods via the Kubernetes API, and the *telemd-kubernetes-adapter* publishes these events, as well as *telemd* will monitor newly created Pods. Further, if our *go-load-balancer* is used, users must update weights manually to include new or deleted applications.

[13]https://github.com/edgerun/telemd/

#### 4.1.3.3 Implementation

Our projects mainly use Python. Only the load balancer, a Kubernetes adapter, and the monitoring agent are written in Go. All projects are open source and available in the Edge Run project. In the following, we want to give a high-level overview of the different repositories and their functionality.

*Galileo* does two things: start clients and record experiment data. It offers the essential working tools but is not automated. Users must deploy an experiment setup manually.

*Galileo Experiments* contains code that invokes *galileo* to start and record experiments. It also includes deployment files for necessary components and describes in detail which other services are required to start an experiment.

*Galileo Experiments Extensions* provides client implementations for our functions and serves as an entry point for experiment execution.

*Galileo Experiments Functions* contains OpenFaaS-based functions that can be deployed and benchmarked using clients implemented in the *Extensions* project.

*Galileo Jupyter* contains gateways that allow users to easily access data recorded during the experiments. Specifically, the *K3sGateway* is implemented to analyze Galileo experiments. The *galileo-jupyter* project internally uses *galileo-db* to access the data storage. Both projects have been extended to support InfluxDB.

The *Request Generator* project provides functions to generate workload profiles that can be used in the experiments.

*Go Load Balancer* It is a weighted round robin Load Balancer that watches etcd for weight changes. It sets HTTP headers to later analyze the path each request has taken to finally arrive at an application instance.

*Telemd* is used as a lightweight push-based fine-grained monitoring agent and supports resource monitoring on the node and application level. Users can set the interval for each resource separately. Published messages follow a specific topic schema that starts with the keyword `telem`, followed by the corresponding node, the metric, and an optional subsystem. Available subsystems depend on the metric. For example, when reporting disk usage, the subsystem might indicate the actual disk from which the metric originates. For example, the topic to monitor disk usage of disk `sda` on host `server` looks like this: `telem/server/disk/sda`. The message of each topic first contains the timestamp of the measurement and then the value. Currently, *Telemd* supports various metrics ranging from CPU statistics, such as usage and clock frequency, to memory, disk and network usage. *Telemd* also supports the collection of metrics on a system- and container-level, including integration with Kubernetes Pods.

The framework automatically deploys the *Telemd Kubernetes Adapter*, which watches Kubernetes Pods using the official Go Kubernetes client. Any Pod lifecycle event is reported via Redis and gets recorded during experiments. This allows users to analyze

scale events and associate telemetry with Pods. The message schema of the *Telemd Kubernetes Adapter* is similar to the one from *Telemd*.

### 4.1.4 Demonstration

This section describes possible use cases to motivate our framework, and we evaluate in Section 4.1.4.2. Afterwards, we present our applications used to evaluate selected use cases to show the results of the framework on our testbed.

#### 4.1.4.1 Use cases

*Profiling* experiments can be used to gather performance measures and resource usage to help developers better understand application implementations and estimate how well a deployment might perform [JNW+19]. Especially in heterogeneous environments, with resource-constrained devices and modern hardware accelerators, it is essential to understand hardware and applications [BJCG21]. Another use case for a *profiling* experiment is the measurement of performance interference due to multi-tenancy [JGJ+18]. The *scenario* experiments differ in configuration capabilities and let users define a range of initial application instances and their location. The framework does not offer any support for cluster management techniques, and users must start schedulers or scalers that interact with the cluster. Nevertheless, during the experiment, any changes to the clusters are recorded, meaning that users can perform cluster management actions during runtime and use our framework to analyze the experiments. For example, an analytics technique is to display when and where application instances have been created and shut down [RCLN20].

#### 4.1.4.2 Applications

We are deploying an OpenFaaS-based function, which serves the Mobilenet neural network [HZC+17] using TFLite in CPU mode. TFLite is a version of TensorFlow tailored towards resource-constrained devices. The function performs object classification on base64-encoded images. The function, as well as the *Galileo experiment* client application, are available on GitHub. You can find the code for the demonstration on Github[14] as well. We also include an evaluation of resource usage of *telemd*, our monitoring agent, the only experiment component running on end-devices (except for *kubelet*).

#### 4.1.4.3 Testbed

Table 4.1 displays a selection of devices in the testbed. *NX*, *TX2* and *Nano* are all Nvidia Jetson devices[15]. There are three clusters in total: *IoT Box*, *Cloudlet*, and *Cloud*. The edge-based clusters are derived from [RLF+20].

[14] https://github.com/edgerun/galileo-experiments-tdis-2022
[15] https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/

Table 4.1: Testbed

| Device | Arch | CPU | Memory | Cluster |
|---|---|---|---|---|
| 1x AsRock | x86 | 8x Ryzen @ 2 GHz | 33GB | IoT Box |
| 1x RPI 4 | arm32v7 | 4x Cortex @ 1.5 GHz | 1 GB | IoT Box |
| 1x NX | arm64v8 | 4x Cortex @ 2 Ghz | 8 GB | IoT Box |
| 1x TX2 | arm64v8 | 4x Cortex @ 2 Ghz | 8 GB | IoT Box |
| 1x Nano | arm64v8 | 4x Cortex @ 1.4 GHz | 4 GB | IoT Box |
| 1x Xeon | x86 | 4x Xeon @ 4.6GHz | 16 GB | Cloudlet |
| 4x VM | x86 | 4x vCPU @ 2Ghz | 8 GB | Cloud |
| 2x NUC | x86 | 4x i5 @ 2.2 GHz | 16 GB | *Clients* |

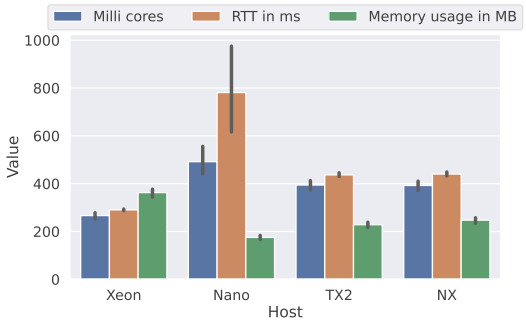



Figure 4.5: Profiling results of the MobileNet function

#### 4.1.4.4 Experiments

**Profiling experiment** The profiling experiment aims to evaluate performance and resource usage across devices. One client sends 100 requests with an inter-arrival time of one second. We execute the experiment on four devices: *Xeon*, *NX*, *TX2* and *Nano*. Figure 4.5 shows the milli cores used by the container, the RTT in ms, and the memory usage in megabytes for each device. The error bars indicate the 95th mean confidence interval. While *NX* and *TX2* have comparable performance, the *Xeon* node has the lowest RTT. The *Nano* performs worse, and outliers are as long as 6 seconds. A milli core value of 1000 equals one fully utilized CPU core. The *Nano* performs again the worst and has the highest CPU usage. Interestingly, the *Xeon* device has the highest memory usage, while the *Nano* has the lowest. It is out of scope to further discuss the results, as we only intend to demonstrate possible use cases for our experimentation and analytics framework. The *galileo-jupyter* project offers many convenience functions (i.e,. `preprocessed_traces`, `preprocessed_telemetry`) that enable developers to plot this chart based on the results (i.e., with Jupyter Notebooks[16]).

[16]https://jupyter.org/

**Scenario** We now turn to the scenario experiment. Specifically, we start three clients spawning 60 requests with an inter-arrival time of 1 second. Two clients are situated in the *IoT-Box* cluster and one in the *Cloudlet* cluster. We also start a Python program that randomly scales up and down in an interval of 5 seconds. With a probability of 10% it does nothing, with 40% it starts one new function instance, and with 50% it tears one instance down. All load balancers distributed requests in a round-robin fashion for this experiment (i.e., from *Cloudlet* to *Cloud*). Figure 4.6 shows how many function replicas have been running across the cluster. This plot illustrates the behavior of cluster management techniques, and analyzing it serves as an essential task during development. Further, our framework enables users to analyze where requests have been generated and where they have been processed. To sum up, our framework can perform experiments that allow requests to be processed across the edge-cloud continuum and offers functionality to analyze them. As before, *galileo-jupyter* exposes convenience functions to support users in developing and testing cluster management techniques (i.e., `get_replica_schedule_statistics`).

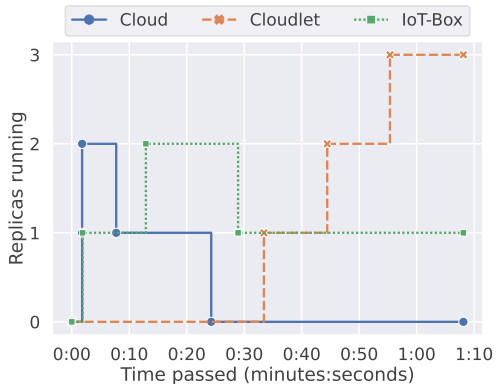



Figure 4.6: Function replicas running across clusters

**Telemd evaluation** To conclude this section, we show our monitoring agent's resource usage while it emits resource metrics every second. Figure 4.7 displays results recorded during a 100-second experiment. The figure shows the relative container CPU usage of *telemd*. 100% CPU usage means that the container (i.e., *telemd*) fully utilized all cores of the device. Figure 4.7 also shows the memory usage in megabytes. Both results show that resource usage is relatively low on all tested devices, ranging from a Raspberry Pi 4 to an Xeon-based PC. Users can set a different monitoring interval for each resource to reduce resource usage further.

### 4.1.5 Related Work

Works have been published that provide applications (i.e., functions) to evaluate systems but do not propose an end-to-end framework [KL19]. We can use these functions with our framework if they expose an HTTP endpoint. Grambow et al. [GPB+21]

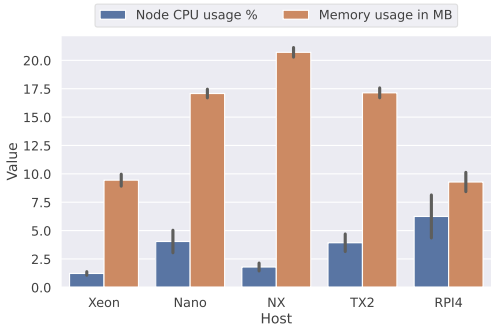



Figure 4.7: Resource usage of *telemd*

present a benchmarking framework for FaaS platforms. They focus on supporting public FaaS offerings while we present a framework for edge-cloud testbeds. Gao et al. [GZGD20] present LinkLab, a scalable IoT testbed that supports experimentation and remote development. In contrast to their work, our framework aims to support users in evaluating cluster management techniques and profiling applications, while theirs focuses on developing and evaluating IoT applications. Yang et al.[YWCW22] present EdgeTB, a hybrid testbed to evaluate distributed machine learning applications at the edge. EdgeTB combines emulated nodes with real ones to create a high-fidelity and large-scale evaluation. In contrast to our work, their framework is for AI training, while ours is flexible regarding applications. Das et al. [DPW18] present a benchmark suite to evaluate commercial edge computing platforms. To summarize, many approaches exist that build benchmarks for FaaS or edge-cloud computing, but we specifically aim to support the development of cluster management techniques.

### 4.1.6 Future Work

The testbed configuration (i.e., labeling the nodes) is not yet fully automatic, and configuring network latency between clusters has to be done manually. For future work, we plan on extending *Ether*, Edge Run's topology synthesizer [RLF+20], to model the real-world testbed as code and perform configurations (i.e., WireGuard) as well as set the network latency between them. Another extension we are working on right now is adding power measurements and using them to train models that can predict energy consumption based on resource usage. Using our framework, we can fully automate and perform different experiments to gather enough data to train our models.

### 4.1.7 Summary

Evaluating edge computing systems is challenging, given both the nascent nature of the field and the heterogeneity of the infrastructure on which such systems operate. Benchmarks and testbeds used to evaluate cluster management techniques are often tailored to the specific technique, making it difficult to reproduce them and therefore compare approaches. We extend *Galileo*, a system for distributed load testing that enables

users to define and run reproducible benchmarks. *Galileo* lacks tools to operationalize experiments; users must manually deploy other tools to scale workers, instrument their testbed, and collect data. We extend *Galileo* to use Kubernetes to deploy runtime components, and extend and create tools to offer an end-to-end experiment and analytics framework for edge-cloud clusters. Therefore, the framework is able to evaluate and serve as a base for our self-adaptive Serverless Edge Computing approach in Section 5.1. The next section deals with the simulation of edge-cloud environments.

## 4.2 Serverless Edge Computing Simulator

This section presents *faas-sim*, a simulation framework tailored to serverless edge computing platforms. In serverless computing, platform operators are tasked with efficiently managing distributed computing infrastructure completely abstracted from application developers. To that end, platform operators and researchers need tools to design, build, and evaluate resource management techniques that efficiently use of infrastructure while optimizing application performance. This challenge is exacerbated in edge computing scenarios, where, compared to cloud computing, there is a lack of reference architectures, design tools, or standardized benchmarks. *faas-sim* bridges this gap by providing (a) a generalized model of serverless systems that builds on the function-as-a-service abstraction, (b) a simulator that uses trace data from real-world edge computing testbeds and representative workloads, and (c) a network topology generator to model and simulate distributed and heterogeneous edge-cloud systems. We present the conceptual design, implementation, and a thorough evaluation of *faas-sim*. By running experiments on both real-world test beds and replicating them using *faas-sim*, we show that the simulator provides accurate results and reasonable simulation performance. We have profiled a wide range of edge computing infrastructure and workloads, focusing on typical edge computing scenarios such as edge AI inference or data processing. Moreover, we present several instances where we have successfully used *faas-sim* to either design, optimize, or evaluate serverless edge computing systems.

### 4.2.1 Introduction

**Context & Background** Serverless edge computing is an emerging distributed computing model that applies principles from serverless cloud computing to edge computing systems [ATC+21]. In traditional serverless computing, serverless platforms hide the underlying infrastructure of massive cloud data centers from application developers while giving platform operators the controls to schedule and scale workloads to optimize operational goals [JSSS+19]. The edge-cloud continuum extends the narrow cloud- or edge-centric view to a hierarchical geo-distributed system, ranging from resource-constrained edge devices to large-scale cloud data centers [RND23]. Applying the operational mechanisms that enable serverless computing in this new environment is highly challenging and subject to active research [XTQ+21, ATC+21, RND23]. Current cloud-centric orchestration services need to be adapted in their architecture [PB20a], which is typically

centralized and can form bottlenecks. While concepts like federated clouds can help connect multiple on-premises data centers and mitigate scalability issues [TPTE21], how current serverless platforms behave in these heterogeneous and geo-distributed scenarios is not well understood. New challenges introduced by the edge-cloud continuum include discovering and forming computing clusters [LLR+21], incorporating new computing device types while aiming for resource efficiency [WKB+19], or dealing with data and computation movement trade-offs [RRD21].

**Motivation** With the growing adoption of serverless edge computing and the complexity inherent in these systems, there is a pressing need for effective evaluation frameworks to assess the performance and help aid the design of serverless edge computing platforms. In cloud computing research, there is a long history of well-understood benchmarks [CST+10], testbeds [DRM+19], and trace data sets [VPK+15] to perform systems evaluations. Evaluating edge computing systems is much more difficult due to the lack of established benchmarks, reference architectures, trace data sets, or real-world testbeds [RLF+20]. Researchers must invest considerable resources to create testbeds and benchmarks often tailored to the system they are evaluating [RRP+22]. This then often leads to evaluations that do not allow generalizable conclusions about the performance of the systems under the wide variety of conditions that the edge-cloud continuum provides. Simulators help to fill this gap by allowing quick iterations over ideas and evaluating their feasibility before deploying them on real-world hardware [MAJ+21]. They allow the creation of synthetic infrastructure according to specific parameters, such as deployment density in geo-distributed settings or heterogeneity of compute nodes [RLF+20], thereby vastly expanding the evaluation parameter space. Moreover, simulations can facilitate use cases such as tuning serverless function orchestration strategies [RRD+22], resource planning, cost prediction, and application performance estimation. As we explore in this section, simulators can even be used in co-simulation, where systems optimize themselves during runtime by evaluating different scenarios using the simulator.

**Challenges** In recent years, numerous simulators have emerged for different system environments, such as cloud computing [JCSY19] IoT [JAA+20, AJH+21] serverless computing [MK21, MAJ+21, RHHP22], and edge computing [SOE17, GVDGB17]. However, existing work does not thoroughly investigate the requirements, use cases, and solution approaches to address the challenges described in the previous subsection. Moreover, many state-of-the-art systems simulators, such as CloudSim [JCSY19] and its many derivatives [GVDGB17, SOE17], build on oversimplified resource models that have several limitations that we analyze in depth in this section. From studying the current state-of-the-art and analyzing simulation requirements, we summarize the challenges for building a modern simulator for serverless edge computing platforms as follows:

(i) To seamlessly transfer simulation designs to real-world systems, the domain model should generalize edge-cloud simulators to include serverless computational models. (ii) The simulator should have strong support to evaluate orchestration strategies as they are essential for governing the overall behavior of function deployments. Specifically,

serverless function orchestration strategies (scaling, placement, and routing) should be first-class entities for users to modify and extend. (iii) Since serverless edge computing is still evolving, the simulation engine should strongly emphasize modularity and extensibility. (iv) High-fidelity simulators are typically slow and challenging to implement but closely resemble real-world conditions. Conversely, low-fidelity simulators are fast and straightforward to implement. Hence, the simulator should be configurable to accommodate the current needs of client programmers and platform designers and create low or high-fidelity simulations. (v) The spatio-temporal context in simulations is highly application-specific, i.e., it requires careful modeling of component interference and user access patterns. Consequently, client programmers and platform designers should be able to integrate custom resource models. (vi) The simulation should support programmable topologies that scale for large clusters to model the heterogeneous resource aggregates.

**Contributions** We have built *faas-sim* to address these challenges and overcome the limitations of state-of-the-art simulators for serverless edge computing platforms. *faas-sim* is a trace-driven stochastic discrete-event simulation engine, enabling the fine-grained evaluation of serverless edge computing platforms on edge-cloud infrastructures. It simulates the deployment, execution, and resource consumption of serverless workloads on different types of computing hardware and comes with a flow-based network simulator based on the network topology synthesizer *Ether* [RLF+20]. Our code framework provides a generalized serverless system model that allows modeling a wide range of function orchestration strategies and platform optimization algorithms, as well as developing reusable benchmarks and experiments. The simulation engine has a flexible performance and resource modeling approach to evaluate common system performance indicators such as resource consumption, function execution time, data throughput, or network usage. *faas-sim* is trace-driven and relies on data from real profiling experiments. It ships with a wide range of profiled workloads and compute nodes that can be used out of the box to bootstrap experiments, but users can provide custom profiling traces to model other workloads and hardware. *faas-sim* is integrated into the Edgerun project [17], offering extensive tooling to support researchers and practitioners performing edge-cloud experiments. It is written in Python, built using the discrete event simulation engine SimPy [Mat08], and integrates seamlessly with modern data science tools. The objective of *faas-sim* and the complementary Edgerun ecosystem is to aid researchers and platform designers in developing and evaluating new serverless edge platforms. For example, the experimentation framework *Galileo* [RRP+22] can run experiments on testbeds and pre-process traces that *faas-sim* expects.

We summarize our contributions as follows:

- *faas-sim*, an open-source[18] serverless edge computing simulator based on the design and performance data of real-world platforms.

[17] https://github.com/edgerun/
[18] https://github.com/edgerun/faas-sim

- *faas-sim* enables researchers and practitioners to design, implement, model, and evaluate serverless platform architectures and orchestration strategies such as function autoscaling, scheduling, or load-balancing algorithms.
- An efficient performance and resource modeling approach that uses profiling data from real-world edge-cloud workloads and compute infrastructure.
- A set of profiling benchmark traces included in *faas-sim* to bootstrap simulations.
- An evaluation on a real-world testbed demonstrating the accuracy of our network simulator.
- A use case-based evaluation showing the capabilities of *faas-sim*.

**Outline** The remainder of the section is structured as follows. Section 4.2.2 introduces the simulation-driven design process of serverless edge computing. Specifically, we introduce two main tasks that serverless edge platforms must implement. Based on use cases, we show how our simulation can facilitate implementation and highlight the resulting simulation challenges. Afterward, Section 4.2.3 describes the conceptual model, architecture of *faas-sim*, and various models for resources, performance, and network. Section 4.2.4 represents the evaluation of our work. It is structured into multiple parts and evaluates *faas-sim*'s ability to model the simulation use cases we present in Section 4.2.2, shows traces that come with *faas-sim*, evaluates the network simulation, and showcases its flexible domain model by implementing the reverse proxy component of OpenFaaS [Ope16]. Afterward, Section 4.2.5 highlights the main similarities and differences between ours and other simulation engines. Section 4.2.6 concludes our work and outlines future tasks.

### 4.2.2 Simulation-driven Serverless Edge Computing Design

In the following, we describe fundamental aspects of our work revolving around serverless edge computing and simulations. We introduce in Section 4.2.2.1 serverless edge computing in general and the tasks of designing platforms and function orchestration strategies which *faas-sim* intends to support. Therefore, we show in Section 4.2.2.2 use cases that facilitate the design process of platforms and address the simulation challenges that arise.

#### 4.2.2.1 Serverless Edge Computing

Serverless edge computing extends serverless computing to manage resources and applications across the edge-cloud continuum [ATC+21]. Serverless computing combines Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [JSSS+19]. FaaS enables developers to split complex applications into multiple simple functions, package and upload them. The FaaS provider is responsible for placing and scaling function replicas and routing requests to running replicas. These are the main orchestration strategies of FaaS platforms to manage function deployments dynamically. Platform clients can

invoke functions through various triggers. A client can send an HTTP request forwarded through a router (e.g., application-layer load balancer) to a function replica. Clients can publish messages in a queue, and brokers are responsible for forwarding them to function replicas. Other event-driven triggers include file and database changes. [JSSS+19]. Function replicas are typically stateless and only have ephemeral storage. However, stateful function support is actively explored and commercially available (e.g., on Microsoft's Azure Functions platform [BGJ+21]). Replicas cannot rely on the assumption that data is locally available and must fetch it from backend services. BaaS enables FaaS and manages various services such as databases, message middleware, routers, and caches. The autonomous orchestration of backend service components are crucial to succeed in the edge-cloud continuum and guarantee operational goals [RND18]. Orchestration strategies are essential for serverless platforms and directly impact operational goals. Each of the orchestration strategies can have different implementations. For example, placement can have various implementations, such as optimization-based [RRD21] or AI-driven [YCYO23]. Scaling can be reactive or proactive, while platforms can route centralized (e.g., a single gateway that all requests go through) or decentralized (e.g., where multiple gateways exist). These options are not exhaustive and should only highlight different ways of adapting function deployments. Interactions between the orchestration strategies are also complex. The system's performance after updating them can lead to unexpected behavior. Serverless edge computing represents a complex system facing many tasks and challenges. Figure 4.8 shows an example of a serverless edge computing platform. Note that requests are routed in a decentralized manner, but orchestration decisions are made centrally in the cloud. This architecture might not be feasible due to the cloud's centralized scaling and placement components. However, a naive decentralization approach can lead to other problems, such as state sharing and consistency problems [RB19]. The figure also emphasizes that serverless platforms must implement orchestration strategies for FaaS and BaaS. The platform must place and scale function replicas and manage the backend services.

Operational goals can range from performance (e.g., response time) to privacy (e.g., execution location) and are crucial for emerging application paradigms (e.g., Edge Intelligence [DZF+20]). Edge Intelligence emphasizes and requires the execution of applications across the edge-cloud continuum. Applications may require ultra-low latency (e.g., $<$ 10 milliseconds), have strict privacy constraints (e.g., health data), and are formulated as complex workflows consisting of multiple applications, each having different requirements (e.g., AI pipelines) [RD21]. Serverless orchestration is not only based on requirements but must consider environmental changes. A key characteristic of the edge-cloud continuum is its dynamic and heterogeneous environment. Therefore, platforms must be resilient, and their orchestration strategies must continuously evolve and cope with changes. These changes stem from the infrastructure side (e.g., nodes fail, new nodes join), from new function requirements (e.g., privacy), or unseen functions. This introduction to serverless edge computing should give a broad overview of the complex interdependencies between the platform architecture and serverless function orchestration. However, we want to give more details about these two aspects of platform design and

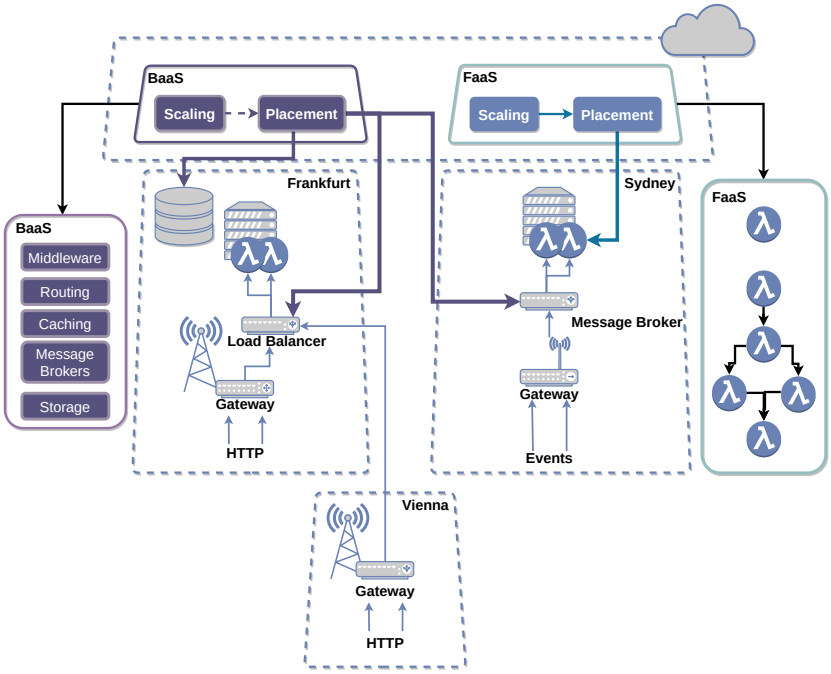
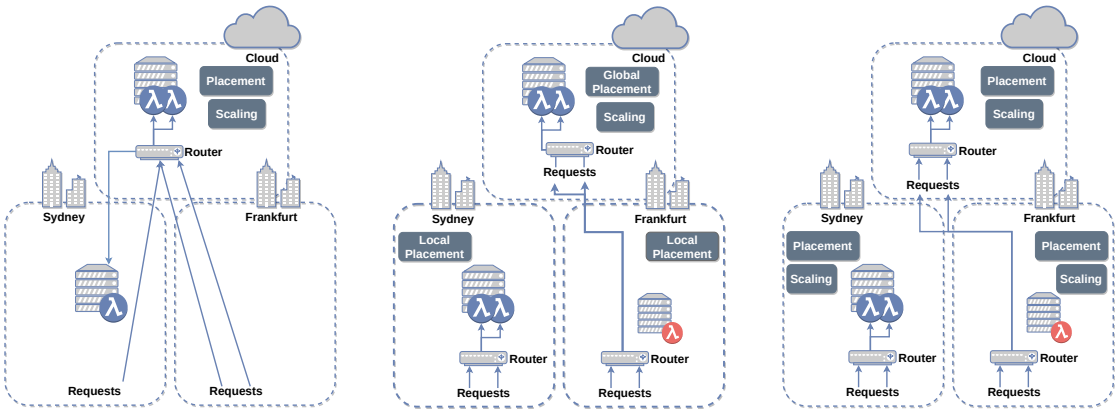



Figure 4.8: Example platform with decentralized gateways and centralized scaling and placement components

how they can benefit from *faas-sim* and the simulation challenges they introduce.

**Platform architecture** In this work, the platform architecture determines the deployment (e.g., location of execution) and the responsibilities of the serverless orchestration components, and which technologies the execution model consists of (e.g., function runtime). Thus, a simulation tool should be able to model these different aspects, from the placement of components to the underlying function runtimes. To highlight the difficulty in selecting an architecture, we briefly describe three platform architectures that are currently deployed or plausible for serverless edge computing platforms. This comparison should emphasize the need for tools that can evaluate different platforms without the burden of implementing each platform on a real-world system. Figure 4.9 depicts these architectures on an infrastructure that consists of three computing clusters exist (i.e., Cloud, Sydney, and Frankfurt). It is plausible that the number of computing clusters is much higher and consists of heterogeneous devices and network conditions. Thus, introducing the simulation challenge to be flexible regarding scenario definition. The first architecture is the typical cloud-centric Centralized Platform Architecture. It deploys a single entry point for users in the cloud, and placement and scaling components are also situated there. In this example, all components reside in the cloud, and requests must travel from the edge to the data center. This architecture has several shortcomings and represents how most commercial and open-source serverless platforms are built (e.g.,




(a) Centralized Platform Architecture

(b) Hybrid Platform Architecture

(c) Decentralized Platform Architecture

Figure 4.9: Three possible Platform Architectures for Serverless (Edge) Computing

OpenFaaS[19], Knative[20], Fission[21], OpenWhisk[22], AWS Lambda[23]). The main shortcoming is that requests must travel to the cloud to be handled, which increases network latency and diminishes the advantages of function instances at the edge. The Hybrid Platform Architecture shows an approach that deploys routing components at the edge and deploys a two-step placement approach [RRD+22]. The main differences from the Central Platform Architecture are decentralized router instances and the decentralized placement approach. The scaling component still resides in the cloud and decides when new function instances are necessary (or when to remove instances). The global placement component decides the computing cluster, and the local placement component must select a node. This architecture scales better because the global placement component can use simple heuristic algorithms while the local placement component can use complex algorithms. The drawback of this approach is that the cloud components can still be a bottleneck, which the design must incorporate. The Decentralized Platform Architecture deploys placement, scaling, and routing components in each computing cluster [BHQT22]. This architecture offers the advantage of decentralized components and high scalability. However, implementing this design can be challenging as it must coordinate placements and routing between computing clusters.

Other variants are possible; for example, the placement components can have distributed implementations that increase the complexity but offer high scalability [RB19]. To explore the performance impact of different platform architectures and underlying infrastructure, simulators are a good tool for getting first results and building confidence in a particular design. To this end, simulations must be customizable and extendable to adapt and

[19] https://www.openfaas.com/
[20] https://knative.dev
[21] https://fission.io/
[22] https://openwhisk.apache.org/
[23] https://aws.amazon.com/lambda/

model different platform designs quickly. Moreover, simulation users should be able to model and configure various system parameters easily, which introduces the challenge of simulation configurability. Simulators should ship with a basic set of system assumptions so researchers can get started with baseline experiments quickly, but also allow high configurability through custom performance models (see Section 4.2.3.2), parameterizable infrastructure topologies (see Section 4.2.3.2), or adding completely custom simulation components.

**Serverless function orchestration strategies** The platform architecture determines where the orchestration components reside and how they interact (e.g., centralized vs. hybrid). However, the three main orchestration strategies (i.e., scaling, placement, and routing) govern the placement of function instances and backend services. We highlight their challenges in the edge-cloud continuum and briefly discuss implementations from the literature. This discussion highlights the complexity of designing function orchestration for serverless edge computing platforms, from which there are simulation use cases that researchers and practitioners alike can benefit from. We categorize the challenges that orchestration strategies face into application and infrastructure. Edge-cloud applications exhibit high amounts of spatio-temporal workloads. Due to systems spanning multiple cities or countries, the workload is generated at different rates in different places. Therefore, orchestration strategies must optimally place function instances in the edge-cloud continuum to process requests efficiently. Emerging application paradigms (e.g., Edge Intelligence) can have stringent requirements that platforms must fulfill. Moreover, these applications also exhibit high heterogeneity in resource usage and complexity. Complex AI pipelines are a prime example in which different hardware accelerators must be taken into account, but dynamic application flows can also change the path of execution (i.e., which functions must be executed) [CSM+20, RD21]. We can split the challenges from the heterogeneous infrastructure into node-level and topology-level. Nodes offer different capabilities and capacities. Capabilities can be hardware accelerators or other physical attachments exclusive to specific devices (e.g., cameras); therefore, orchestration strategies must be aware of correctly placing function instances that require capabilities. Multi-tenancy leads to performance interference, which can negatively impact resource-constrained nodes. Further, some capabilities are exclusive to individual function instances (e.g., hardware accelerators); therefore, orchestration strategies must carefully use those resources. The topology can consist of computing clusters that have different capacities and capabilities. Due to heterogeneous networks, nodes have different network conditions and can leave or enter the topology, causing instability. Simulations help understand the ability of function orchestration strategies to handle these types of heterogeneity. Previous challenges have been extensively covered in literature but are fundamental to orchestration strategies and serverless edge computing [ATC+21, SKM22]. It becomes clear that implementing orchestration strategies in the edge-cloud continuum is complex. Therefore, extensive evaluation of orchestration strategies is crucial for a successful deployment. However, it also introduces several challenges that the simulation must address for a viable solution. First, the simulation must incorporate well-known

components in the core to evaluate orchestration strategies. These components include a scheduler, an autoscaler, and load balancers. Simulation users should be able to extend and implement novel solutions for these components. At the same time, the simulation should use a realistic domain model that maps simulation components to real-world ones. The last challenge revolves around the scenarios and request patterns during simulations. As previously mentioned, spatio-temporal context is important in edge-cloud systems (e.g., people moving around the city [RRD+22]), and simulations must be able to model this aspect.

Our simulation can support the development of serverless edge platforms by allowing rapid evaluation of different designs that propose individual solutions to these challenges. Therefore, we present simulation use cases that can help design and understand platform designs.

#### 4.2.2.2 Simulation use cases

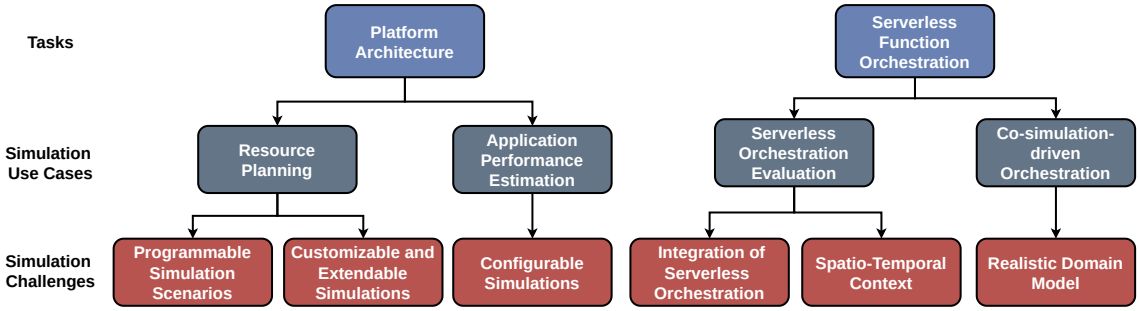



Figure 4.10: Simulation use cases that facilitate implementation of tasks for serverless edge computing platforms.

Before discussing serverless edge computing platform simulations, we want to describe their use cases. Figure 4.10 shows each task's corresponding simulation use case and the simulation challenges it creates.

**Resource planning** Resource planning determines the necessary physical hardware capacities to serve workloads adequately [LLR+21]. The simulation can help understand platform providers and which architectures suit their infrastructure. They also can perform simulations in which the infrastructure varies, which can support future investments in new equipment. In addition, simulations can also determine the highest workload the infrastructure can withstand. The simulation tool must support automated topology and workload creation to perform many simulations to enable platform providers to identify possible interesting infrastructures.

**Application performance estimation** Application performance estimation has various facets and not only includes the estimation of performance-related indicators such as the response time of functions but can also include other factors such as Quality of

Service and cost [MK21]. Customers can get estimates for deploying their functions, while platform providers can use them to see whether different platform architectures impact the cost in advance. The application performance estimation use case requires simulation users to freely adapt the simulation to their specific use cases.

**Serverless orchestration evaluation** It is challenging to operationalize orchestration strategies in dynamic systems like the edge-cloud continuum, and requires continuous re-evaluation and optimization at both design-time and run-time. Orchestration strategies range from threshold-based approaches to heuristics and AI-driven strategies. Different ways of evaluating orchestration strategies exist. Testbeds and emulations help evaluate orchestration strategies on realistic setups that remain on a small scale [RRD+22, RNC19]. They offer realistic results but lack scale and can be expensive. Additionally, results can be hard to reproduce because setups have many variables (e.g., OS version, library versions). While efforts exist to make testbed evaluations reproducible [RRP+22] (e.g., by streamlining the definition, execution, and analysis of experiments), many factors depend on resource configuration. Emulations bridge the gap between real-world behavior and scalability of simulations [MGG+17]. Both approaches rely on the available hardware and its configuration, making it hard to reproduce results. Simulations, while highly dependent on the implementation (i.e., performance, resource models, etc.), offer high reproducibility and can support large-scale scenarios. Researchers and system designers must be able to extend and develop their own serverless function orchestration algorithms into the simulation. A simulation tool should offer baseline orchestration strategies, generic APIs to easily interface with the system (e.g., reasoning over the cluster topology, accessing simulated resource usage, or triggering function deployment), and ways to capture spatio-temporal context to model things like clients moving through the geo-distributed topology.

**Co-simulation driven orchestration** The co-simulation-driven orchestration use case focuses on operationalizing orchestration in real-world platforms. Serverless orchestration strategies rely on thresholds, heuristic-based optimizations, or AI-driven strategies. These orchestration strategies rely on parameter tuning, historical data, and possible optimization processes (e.g., AI training) [ZXR+22, MKB22]. Observing the system for changes is crucial once an orchestration strategy has been deployed. These changes can come from the applications (e.g., new applications or new requirements) or the environment (e.g., network saturation, node failure). In any case, the orchestration strategies must incorporate new circumstances during runtime. If orchestration strategies only interact with the real world and learn using historical data, updated configurations might need to be updated when deployed. One of the key characteristics of the edge-cloud continuum is its dynamic nature, and platforms have to autonomously and quickly adapt. In the best-case scenario, they can proactively update strategies to possible future scenarios. The concept of co-simulation tackles this issue by using an underlying simulation engine to quickly update deployed orchestration techniques by simulating possible future scenarios [TPS+22]. Therefore, simulations must use a realistic domain

model to provide interoperability between the implementation in the simulation and the real-world system.

### 4.2.3 faas-sim

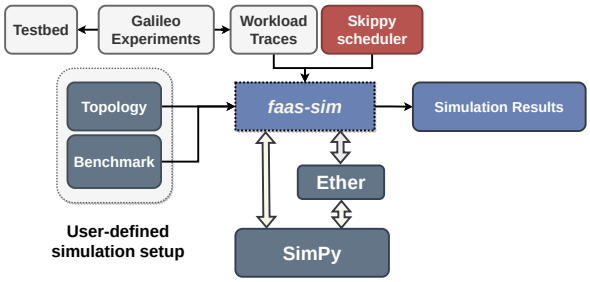



Figure 4.11: *faas-sim* overview

*faas-sim* is a trace-driven stochastic discrete-event simulation framework built on SimPy [Mat08]. It was initially developed to evaluate how well *Skippy* [RRD21], a generic container scheduling system based on the Kubernetes scheduling logic, could make trade-offs between data and computation movement in edge computing infrastructure scenarios. To that end, *faas-sim* primarily simulates serverless workload execution and network data transfer. Figure 4.11 shows an overview of simulation inputs, internal components, and simulation outputs of *faas-sim*.

The default workload scheduling system is Skippy, which is highly customizable and models a wide range of placement strategies but can also be replaced with a completely custom scheduler. To simulate workloads, *faas-sim* uses workload traces from profiling experiments on real-world testbeds to simulate the execution time and resource usage of running serverless workloads on different types of computing hardware. Our *Galileo* experiment framework [RRP+22] can facilitate the execution and pre-processing of experiments on a real-world testbed to generate traces for new functions and devices. For network simulation, it uses a higher-level flow-based network simulation that is part of the *Ether* edge network topology synthesizer [RLF+20].

All functionality *faas-sim* is implemented using Python and SimPy primitives. SimPy's simulation engine is single-threaded and uses a single event queue to model time progression in discrete steps. It leverages Python's generator concept, allowing users to implement simulation processes using the `yield` keyword to emit simulation events and programmatically advance the simulation clock. *faas-sim* uses SimPy's base mechanisms to facilitate the entire simulation and requires no additional extensions. As shown in Figure 4.11, *faas-sim* and *Ether* use SimPy and, therefore, can interact with each other in the process simply by in-memory communication using standard Python. A single Python process simulates the network and the serverless platform. A *Benchmark* is the programmatic execution of a user-defined simulation scenario, which may include workload generation based on specific patterns and allows users to plug in custom workloads

or function simulators. The simulation result is a set of Pandas data frames that include fine-grained workload traces and network simulation results that allow the calculation of system key performance indicators such as average function execution, execution cost, resource usage, or network usage.

*faas-sim* provides pre-defined benchmarks, topologies, workload traces, and scheduling strategies, which users can individually customize. Specifically, *faas-sim* provides (a) traces from several common edge computing devices and representative serverless workloads such as AI-based image recognition, numerical computations, or AI-based speech-to-text; (b) request generators that model real-world workload patterns; (c) common edge computing system topologies and cluster configurations that are synthesized from real-world edge computing use cases [RLF+20]; (d) implementations of the container scheduling strategies presented in previous works by the authors [RRD21],[RRD+22] that can be used as baselines for new strategies.

We outline its conceptual model before explaining the implementation and architecture of *faas-sim* in Section 4.2.3.2.

### 4.2.3.1 Conceptual model

The conceptual model of functions, deployments, and running replicas is shown in Figure 4.12. The API of the conceptual model can be found in a dedicated code repository[24]. It serves as a unified API collection, allowing implementations for different environments, such as *faas-sim*.

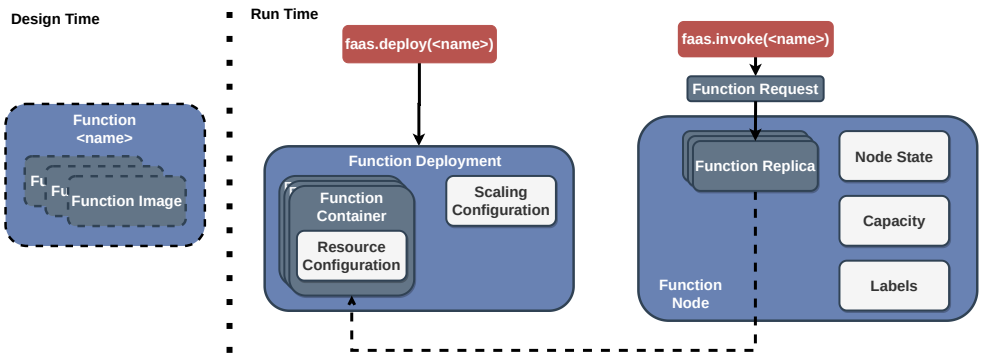



Figure 4.12: Conceptual model of functions and their deployment.

We split the conceptual model of functions into two parts: *Design time* and *Run time*. The *Design Time* includes *Functions* and *Function Images* and gives simulation users the possibility to change the implementation of functions without redefining *Functions*. It decouples the interface from the implementation and enables higher flexibility when defining functions.

[24] https://github.com/edgerun/faas

- A *Function* is the highest level of abstraction and refers to some functionality identified by a name that can be invoked with a *Function Request*. For example, a *Function* could be an object detector named "detect-objects" that takes an image as input and returns the bounding boxes and labels of objects in the picture.
- A *Function* comprises several *Function Image* instances, one for each deployment platform. A *Function Image* is conceptually the code that implements a function on a specific deployment platform. For example, our "detect-objects" function could have one version that uses the GPU and one that uses a TPU (an AI accelerator). The reason for this additional abstraction is the way container platforms like Docker deal with multiple computing architectures. Docker images group different CPU architectures via a manifest to a multi-arch image. A *docker pull* command will pull the correct image based on the node's architecture. However, there is no way to include additional platform aspects such as GPUs or TPUs. If two container images exist for the same function, one that uses the CPU and one that uses the GPU, it may be ambiguous at runtime which image to pull. Instead, we want to allow the placement component to decide which image to deploy for a particular function.
- A *Function Deployment* is an instance of a *Function* with a concrete resource allocation and scaling policy configuration. A deployment consists of multiple *Function Container* instances and said configurations.
- A *Function Container* is the runtime configuration of a *Function Image*. It has a specific resource configuration that declares how many resources are allocated on a node when a particular replica of this *Function Container* is deployed. In our running example, a GPU-based "object-detector" might require less CPU but more VRAM than the CPU-based *Function Image*. A *Function Replica* is a concrete instantiation of a *Function Container*. It represents the actual running function (like a Docker container).
- Each *Function Replica* has a *Function Simulator* object that models the function's behavior for the simulation. Each *Function Replica* has a *Function Replica State* that signals the replica's current life cycle. The life cycles are inspired by Kubernetes and range from being scheduled for placement to running and shutdown.
- A machine capable of hosting function replicas is called *Function Node*. The *Node State* is a generic container for data needed during simulation time, for example, storing the number of concurrent invocations to a particular replica to calculate performance degradation. It also stores information that describes its CPU architecture, number of cores, memory size, and allocatable resources.

#### 4.2.3.2 *faas-sim* architecture

*faas-sim*'s architecture allows the modification and extension to accommodate different serverless edge computing platform designs and execute various use cases. The architecture

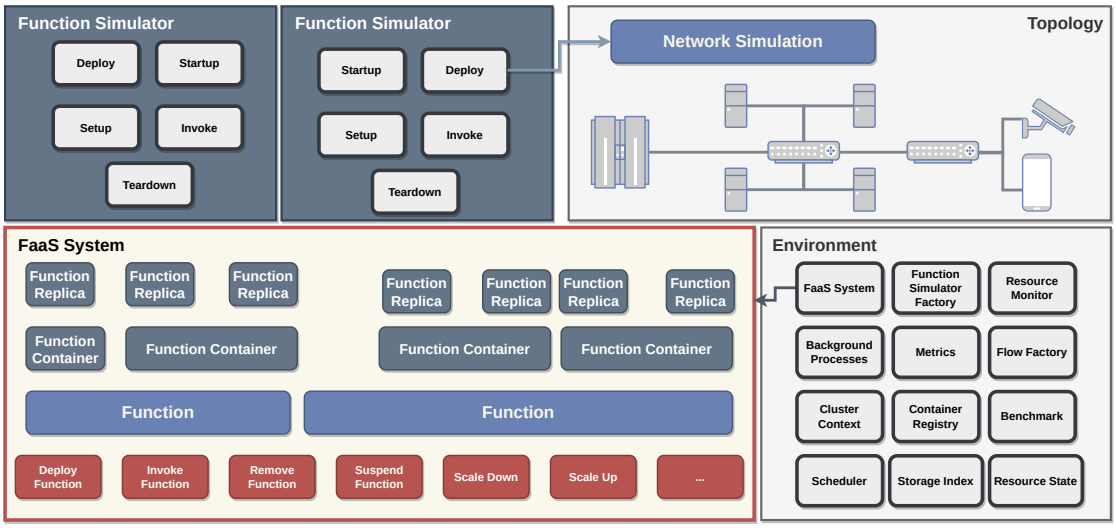



Figure 4.13: Main abstractions of *faas-sim*

mainly consists of three abstractions: *FaaS System*, *Environment*, and *Function Simulator*. Figure 4.13 shows the internals of them, whereas *Environment* contains object references (e.g., to the *FaaS System*), while *FaaS System* and *Function Simulator* show methods to interact with them. The *FaaS System* acts as the central entity that exposes methods to interact with the platform. For example, it allows users to deploy functions, remove, invoke, or scale them up and down (also commonly referred to as *scale out* and *scale in*[RCLN20]). It manages a set of functions and the associated *Function Replica* and acts as the serverless edge computing platform's front end. The *Environment* contains references to essential objects used throughout the simulation. These objects contain the main logic of the simulation and can be configured and accessed via the *Environment*. For example, simulation users can replace the *Scheduler* (i.e., the placement component) by simply pointing it to their implementation. Each *Function Replica* has a *Function Simulator* containing the function's simulation model. Based on that, we structure the remaining section by first outlining the *FaaS System*, followed by the *Environment*. Both represent high-level components that encompass the simulation. Afterward, we detail how the *Function Simulator* works and our approaches to modeling resources, performance, and network as well as faults.

**FaaS System** The *FaaS System* abstraction is the high-level interface a simulation user interacts with. It is inspired by open-source FaaS platforms (i.e., OpenFaaS [Ope16]) and container orchestration services (i.e., Kubernetes [Kub14]). The API of the *FaaS System*[25] presents the frontend of a serverless edge computing platform and lets simulation users deploy, remove, scale, and invoke *Function Deployments*. The following shows an excerpt of available methods:

[25]The source code of this class is available under https://github.com/edgerun/faas. However, *faas-sim* provides an implementation called `DefaultFaaS System`

- `deploy`: makes the function invokable and deploys the minimum number of *Function Replica* instances on the cluster. The number of minimum running instances is configured via *Scaling Configuration*. *FaaS System* creates for each *Function Replica* a new *Function Simulator* using the *Function Simulator Factory*. Each *Function Replica* gets a *Function Node* assigned using the *Scheduler*. The *Cluster Context* is updated accordingly, and the *FaaS System* starts the *Function Replica* life cycle.
- `invoke`: the *Load Balancer* selects the replica and simulates the function invocation by calling the `invoke` method of the *Function Replica's Function Simulator*.
- `remove`: removes the function from the platform and shuts down all running *Function Replica* instances, calling the `tearown` method of each.
- `discover`: returns all running *Function Replica* instances that belong to the *Function Deployment*.
- `scale_down`: removes the specified number of running *Function Replica* instances, with respect to the minimum requirement. The current implementation first picks the most recently deployed *Function Replica* instances, which is Kubernetes' default behavior [RCLN20]. However, simulation users can also pass a list of *Function Replica* to remove. This operation is equivalent to the commonly used *scale in* operation [RCLN20].
- `scale_up`: deploys the specified number of *Function Replica* instances but has to respect the maximum number specified in the *ScalingConfiguration*. As with `scale_down`, the simulation user can also pass a list of *Function Replica* instances to schedule, and the operation is commonly referred to as *scale out* [RCLN20].
- `poll_available_replica`: repeatedly waits and checks for running *Function Replica* instances of the *Function Deployment*.

The API exposes additional lookup methods that we did not include above. The *FaaS System* manages functions using the *Environment* and is responsible for invoking all life cycle phases of the *Function Simulators*. Using the *Environment* allows simulation users to inject their *Load Balancer*, *Scheduler*, and other objects that the *FaaS System* uses.

**Environment** The *Environment* is the central instance that stores object references used throughout the simulation. Figure 4.13 highlights most of the objects currently in the *Environment*. This might change over time as the simulation is continuously updated to implement and provide more features out of the box. However, we explain some of them in detail in the following and show which features they enable.

- The *Cluster Context* keeps track of available *Function Container* images available and resources (i.e., CPU and memory) on each node. This enables the simulation

only to download images unavailable on the node and the *Scheduler* to check if a node has enough resources.

- The *Function Simulator Factory* is a component that simulation users have to provide and determines which *Function Simulator* instance is assigned to a given *Function Replica* upon creation.

- The *Topology* contains all nodes and the network graph and can be created using the external library *Ether*, which also provides the network simulation. *Ether* implements a flow-based simulation that users can transparently change by modifying the *Flow Factory*, which determines the concrete type of network simulation.

- The *Scheduler* has to implement a `schedule` method, which takes a *Function Replica* and decides on which node it should be placed. The default implementation of *FaaS System* uses a SimPy queue to process each requested *Function Replica* sequentially and use the *skippy-scheduler*[RRD21]. However, simulation users can replace the *Scheduler* implementation or modify the *FaaS System* implementation to suit their needs. For example, simulation users can change the default monolithic scheduler architecture to a decentralized one.

- The *Resource State* keeps track of the resources across nodes and *Function Replicas*. This object acts as storage and offers methods such as `put` and `remove`. The simulation user is responsible for using those methods; *faas-sim* does not restrict what kind of resources are stored. The *Resource Monitor* can measure resource usage repeatedly during the experiment by fetching the resource usage for all running function replicas. This approach simulates a pull-based monitoring strategy, in which a central controller is responsible for fetching the latest resource usage from all replicas, which mimics the behavior of real-world monitoring frameworks, such as Prometheus[26]. However, users can also modify the *Resource State* to log every *put* and *remove* action without using the monitor.

- The *Metrics* object is the central logging entity and stores everything in a table-like data structure for export at the end of the experiment (e.g., as CSV file). The class offers some methods tied to the simulation, which *faas-sim* automatically logs (i.e., the scheduling process), but simulation users can log arbitrary data. Simulation users can also implement the *Metrics* class to support other storage mechanisms (e.g., SQL databases) to improve performance and keep memory usage low.

- Simulation users populate the *Container Registry* before starting the simulation with their *Function Containers*. Each *Function Container* has a CPU architecture (i.e., `arm64v8`, `arm32v7`, `amd64`) and size of the image in bytes. The *Container Registry* is a node in the *Topology* which enables the simulation of downloading the *Function Container* onto the node.

[26] https://prometheus.io/

- The *Background Processes* object is a list that simulation users can populate. SimPy has a built-in concept of processes running in the simulation's background. The list of processes is automatically started at the beginning of each simulation and allows users to run processes continuously. For example, the *Resource Monitor*, which periodically fetches the resource usage of each *Function Replica* and node, is a background process. Another critical component in the simulation of serverless edge computing is the *Autoscaler*. *faas-sim* includes a variety of *Autoscalers* that can be included in the list of *Background Processes*. Each *Autoscaler* implements a `run` method that repeatedly observes the platform state and invokes the *FaaS System's* `scale_up` and `scale_down` methods.

- The last concept we include in *faas-sim* is the *Storage Index*, which enables users to model object storage that is popular in serverless computing and is mainly used to store data that functions have processed. Therefore, *faas-sim* can realistically simulate the network transfer between *Function Replicas* and storage nodes.

Having access to all important aspects of the simulation (e.g., *FaaS System*) enables the implementation of *Function Simulators* that act as *Load Balancer*, *Autoscaler*, *Scheduler*, and arbitrary components.

Before describing our evaluation, we want to define the *Simulation* and input parameters (i.e., *Topology* and *Benchmark*) in more detail.

**Trace-driven function simulators** The main goal of *faas-sim* is to determine the function execution time (FET) and resource usage of running workloads on particular nodes. *faas-sim* is trace-driven, which means that users need to provide measurements or estimations from real-world experiments to feed the simulation. We explain this in more detail in Section 4.2.3.2. In this section, we present the *Function Simulator* abstraction, which allows users to simulate various stages of a function that ultimately determine the FET and resource usage.

While we ship a set of functions with *faas-sim*, we also enable simulation users to profile new functions on their hardware using *Galileo* experiments [RRP+22]. In the following, we focus on our profiling approach, but want to emphasize here the flexibility of our simulation. As we mentioned, the simulation does not make any assumptions about resource usage, and users must inject their traces. Further, *faas-sim* also does not have an integrated resource model, and simulation users must implement their own. However, *faas-sim* offers functionality to implement a resource model as we describe in Section 4.2.3.2. Figure 4.14 shows the *Function Replica* life cycle phases and denotes for each one the data we must store to enable the trace-driven simulation. For example, we calculate the resource usage of a single function invocation and use this in the simulation. We also use the duration for each phase to simulate it accurately. The phases, depicted in Figure 4.14, correspond to the phases that the *Function Simulator* offers to implement.

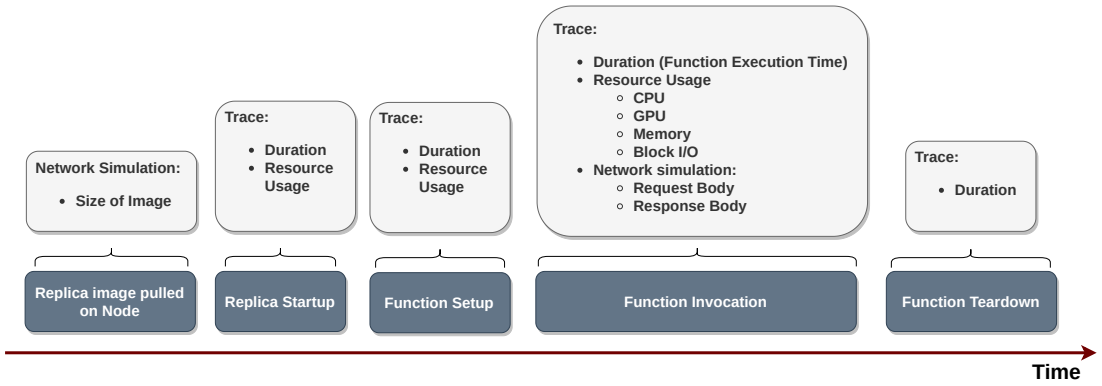



Figure 4.14: Trace-driven analysis process

The *Function Simulator* is the core of *faas-sim* and implements the behavior of a function. Deploying a *Function Replica* consists of the following life cycle phases. The *Function Replica* is *deployed* on the node (e.g., by pulling the container image if it is not present), and *starts* the *Function Replica*. Afterward, the replica can execute some *setup* code and then is ready for *invocation*. In the end, the replica is *torn down*. To simulate this life cycle on a node, a `FunctionSimulator` class must be implemented with the following methods. Each method yields SimPy simulation events, which enables simulation users to implement arbitrary complex simulation models of functions.

- `deploy`: deploys the *Function Replica* on the node (e.g., by pulling the container)
- `startup`: starts the *Function Replica*
- `setup`: executes any setup code of the *Function Replica*
- `invoke`: simulates the invocation of the *Function Replica*
- `teardown`: cleans up and stops the running *Function Replica*

Figure 4.15 shows the life cycle phases and puts the real-world execution and function code next to the simulation code. It also depicts the different states the replica is in during the simulation, from starting to shutting down. The deployed function offers AI inference and initially loads an AI model into memory, which is kept in memory to speed up inference.

Before going into details, we want to highlight three things: (1) in some phases `docker` commands are shown. Typically, the underlying orchestration service executes these commands, and we include them only to simplify the explanation. (2) the simulation is not tied to Docker or container virtualization techniques and can model the behavior of other technologies. (3) this example highlights many aspects of the function execution model, which are all optional,i.e., simulation users can decide which of them they include.

In the *deploy* phase, the `docker pull` command downloads the function image from the container registry. The simulation model simulates a network download using *Ether*, which downloads the Docker image size from the container registry if it is not already on the node.

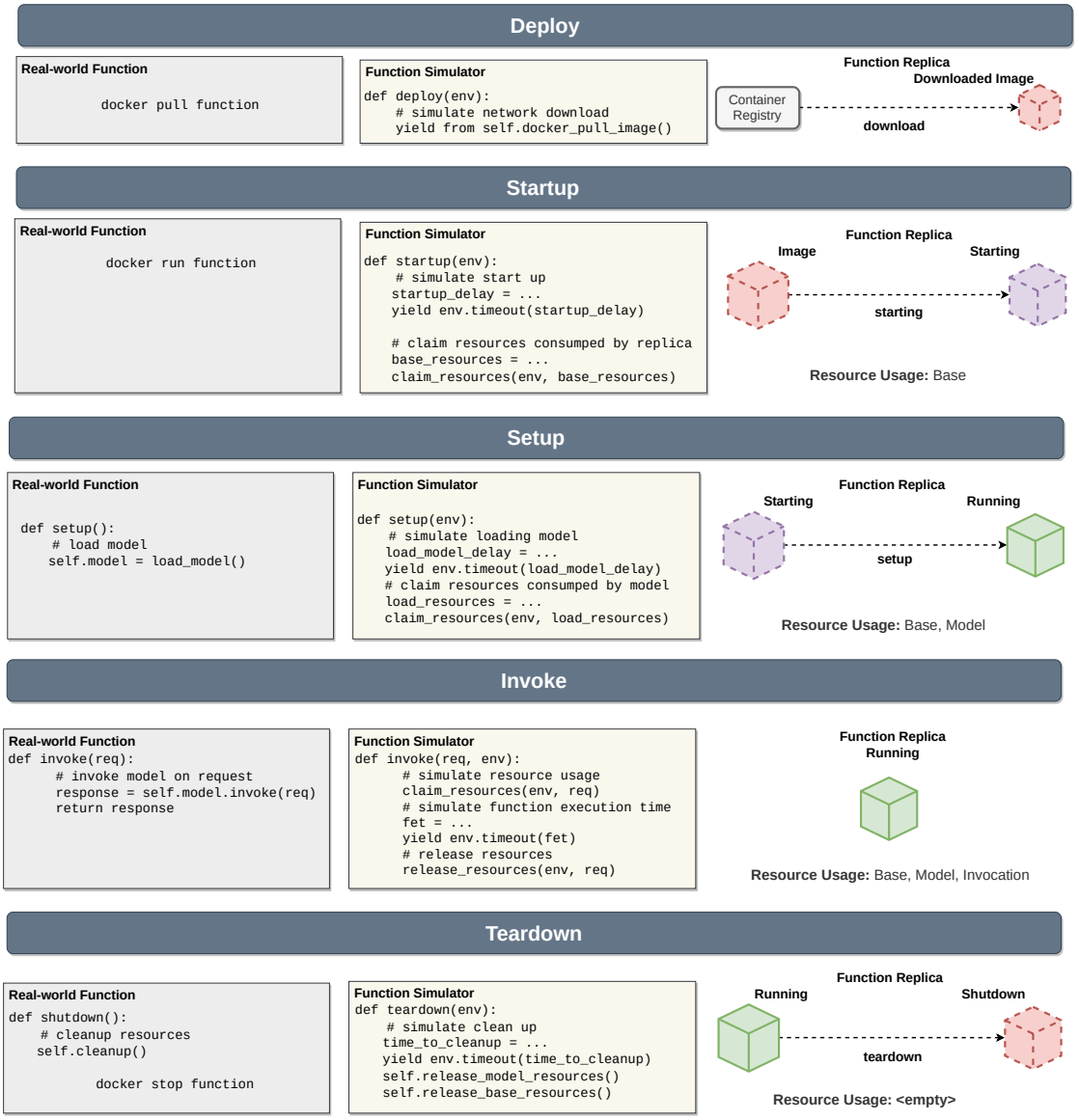



Figure 4.15: Life cycle phases of AI inference function based on OpenFaaS' HTTP *of-watchdog*.

The `yield` statement enables SimPy to track time spent and includes the simulated download time from *Ether*. Afterward, the `docker run` command starts the function, which we simulate using the time taken from the traces. The `env.timeout` call lets

the simulation wait. SimPy has no built-in concept of units, leaving this open to the simulation users. Our traces are in seconds; therefore, `startup_delay` is also in seconds. Next, the simulation includes the container's resources after the startup (e.g., CPU usage, memory). The delay in starting the function and the resource usage come from the trace analysis that the simulation users must provide. The *Function Replica* transitioned in the meantime to the *starting* state and began the *setup* phase. The *Function Replica* only consumes the base resources. As we describe in Section 4.2.3.2, *faas-sim* is not built around a specific resource model but offers a resource state object that acts as storage. For example, we can measure the function's idle CPU usage, store it, and afterward remove it. Further, the *Function Simulator* methods accept (among other arguments not shown for conciseness) an *Environment* (i.e., `env`) argument, which gives access to the resource state. In the *setup* life cycle phase, the container loads an AI model into memory. The simulation models the delay of loading the model and claims resources consumed by the AI model. After the *setup* phase, the replica is *running* and can be *invoked*. In the real world, the function invokes the AI model with the request (e.g., an image). However, in the simulation, the *Function Simulator* first claims resources that are used due to the inference process and simulates the inference by waiting for the time we derived from the traces, and afterward releases the resources. While this example is kept simplistic on purpose, *faas-sim* users can implement sophisticated simulations that model more complex aspects of a function. For example, the execution duration can depend on the input (e.g., small vs. large input), and simultaneously, simulated resource consumption can also rely on the input. To achieve this, a formula (either using profiled traces or ML models) is needed that can estimate the execution time. In contrast, the parameters of this formula can depend on the input size, resource usage based on the input, etc. This allows the modeling of functions that use different resources (e.g., I/O, CPU) based on the input and other factors. The resource modeling approach of *faas-sim* is explained in Section 4.2.3.2, and we later show how it can be used to implement complex multi-tenant scenarios in Section 4.2.3.2 and Section 4.2.4.1.

Eventually, we can stop the function replica. In the real world, we call `docker stop`, and the container receives a signal to shut down (i.e., leads to the invocation of `shutdown`). The *Function Simulator*'s `teardown` method is called, and it waits for a specific time, simulating the clean-up process, and then releases all resources.

The *Function Simulator* is flexible, and simulation users can adapt it to their needs. We show in Section 4.2.4.4 how they can mimic the behavior of OpenFaaS functions.

**Trace-driven performance modeling**[27]

As mentioned, *faas-sim* is trace-driven and relies on the results of basic profiling benchmarks that *faas-sim* then uses to model more complex behavior. Existing simulation systems for edge and cloud computing, such as CloudSim [CRB+11] and its numerous extensions [GVDGB17, SOE17], use a performance model based on discrete values of CPU instructions. The execution duration is calculated using the number of CPU instructions

[27]The trace-driven nature of the simulator has already been introduced in [Rau21].

of a workload and the CPU's instruction rate in instructions per second (IPS) [MK20]. However, it is difficult to accurately measure the number of CPU instructions of a workload due to different CPU architectures; the same holds for the CPU's instruction rate. Further, the CPU speed varies significantly in the edge-cloud continuum [RAD18], and functions do not only rely on the CPU speed but also on other resources, such as I/O. Ultimately, even these approaches have to estimate or profile the number of instructions of a function invocation and the CPU, similar to our profiling experiments. Leading us to a trace-driven approach that can be facilitated using our Galileo framework.

In Section 4.2.3.2, we introduced the *Function Simulator* and showed which traces the simulation can use during the simulation. The *Invoke* step of our example shown in Figure 4.15 has a placeholder for determining the function execution time (FET), which we want to elaborate on. Generally, we simulate FET using an *oracle* that determines the FET based on the workload, the node executing the workload, and the current resource utilization. We model performance and performance degradation by fitting functions on the distribution of workload traces for a particular workload and compute device. During simulation time, the oracle samples from the fitted distributions to determine the FET, and the simulation system records how many requests are being executed in parallel.

While *faas-sim* is not tied to a specific performance model to determine FET and resource usage, we want to show and evaluate the model that *faas-sim* ships with. To that end, we first explain how we model resources of cluster nodes, and then how we use stochastic performance models to simulate performance degradation and multi-tenancy.

**Resource modeling**[28]

Our resource model approach is based on traces gathered during workload and device profiling experiments. Similar to our performance model, which is centered around the execution time of one function invocation, the resource model allows us to describe the resource usage of one function invocation. Based on the traces, we calculate a resource vector representing the utilization of the following resources:

- $u_{cpu}$: the CPU utilization,
- $u_{io}$: the block I/O in bytes/sec,
- $u_{net}$: the network I/O in bytes/sec,
- $u_{gpu}$: the GPU utilization (if available), and
- $u_{ram}$: the RAM usage

Each resource represents the mean usage for one function invocation. We are using *telemd*[29], a lightweight telemetry daemon that continuously collects and publishes data.

[28] Initially conceived in the Master Thesis [Rai21] and later on partly presented in [Rau21].
[29] https://github.com/edgerun/telemd/

*Telemd* enables fine-grained profiling on the container level (i.e., CPU, network & block I/O, and memory). The GPU utilization is measured on the system level, and measures must be taken to avoid interference. Details can be found in [Rai21]. The resource vector can be subsequently *claimed* during *Invoke* step of the function simulator. Adding the resources during each function invocation facilitates estimating performance degradation in multi-tenancy scenarios, as shown in the following section.

**Stochastic performance models**[30]

Stochastic performance modeling embraces the fact that there is variance in the execution duration of a task, even if the conditions appear equivalent. When performance degrades because of concurrent execution of tasks and resource contention, not only does the function execution duration increase, but the variance also increases. Modeling this behavior is particularly relevant for systems that need to balance the load between workers. There are two distinct scenarios: single-tenant performance degradation and multi-tenant performance degradation. In the single-tenant case, only one type of workload is executed on a node, and the performance degradation can be formulated simply as a function of the number of concurrently executing processes. Details on the implementation of the single-tenant scenario can be found in [Rau21]. In the multi-tenant case, multiple workloads may require heterogeneous amounts of resources, which is much harder to model.

**Multi-tenant performance degradation** Besides the single-tenancy case, which is based on fitting simple parameterized distributions, we also explored an ML-based approach to predict performance degradation occurring in multi-tenancy scenarios [Rai21]. Our approach is based on the previously described resource vectors that we assume to have for each deployed function. We integrate this model as follows. At each function invocation, we sample the execution duration based on the stochastic performance model and collect the current resource usage afterward. The resource usage is used as input to invoke our ML model, which returns the factor that worsens the performance. For the final execution duration, we multiply the performance degradation factor by the initially sampled duration. The input is based on the function requests being executed at the moment and base resource usage as well as hardware accelerators (e.g., GPU). The single-column vector has a fixed length of 34. It contains the sum, mean, standard deviation, minimum, maximum, $25^{th}$, $75^{th}$ percentile of CPU, GPU, network, and block I/O usage, mean memory usage, and the number of running containers. The difficulty of this approach lies in training a good model, which we describe in more detail and present results in Section 4.2.4.1.

**Network simulation**[31]

A network simulation is fundamental to simulating interactions between nodes and workloads within a serverless system. Network topology, capacity, and usage significantly

[30] Partly presented in [Rau21] (i.e., single-tenant model).
[31] Partly presented in [Rau21] (i.e., the flow-based network model).

impact the performance of serverless systems, so having a reasonable simulation model for networking is essential. Examples of interactions in our model where the impact of networking is high include:

- downloading a *Function Container* onto a cluster node.
- transferring data from storage nodes to a *Function Replica*.
- transferring data between functions (e.g., request and response data).

Our network simulation is based on the network model of Ether [RLF+20]. Other simulators such as ns-3 [HLR+08] or OPNET [Cha99], precisely model the low-level interaction between network protocols and networking hardware, which differs from our goal. Instead, our simulator implements a high-level network model on top of topologies based around *flows*, representing data transfers between nodes through several links. This way, we trade off fidelity for simulation performance.

The conceptual model of Ether is simple and has three core concepts that are relevant to the network simulation: (a) **node**: a computing or storage device within the network; (b) **link**: a connection between nodes in the network (e.g., a network card, a WiFi access point, or a mobile network uplink), which has an associated *data rate capacity* (bandwidth); (c) **topology**: a graph that models nodes and links as vertices and connections as edges that can hold QoS attributes such as latency.

The model opens a flow *flow* through several connected networks *links*, i.e., a route, when simulating data transfer. This is a very common model for simulating and reasoning over networks [BG96, MR99, Kel97]. Our network simulation implements a simple shortest-path routing through the topology and fair link bandwidth allocation across flows. When simulating edge computing systems, more research is needed to determine the accuracy of different bandwidth-sharing models, such as proportional fairness [MR99]. Further details of the network simulation can be found in [Rau21, RRFD23].

**Limitations** Although the simulation model is straightforward, its accuracy in the basic scenarios we have investigated is comparable to that of packet-level simulators like ns-3, as shown in our evaluation of the network simulation accuracy in Section 4.2.4.3. However, the simplicity of the model comes with some limitations. TCP congestion control using additive-increase/multiplicative-decrease (AIMD) is known to converge to an equal allocation of a contested link between flows [CJ89], our max-min fairness allocation does not model the process of converging to that allocation, which takes time and may affect the result of individual flow simulations. Also, TCP is known to have degrading performance behavior when there are many parallel flows [Mor97], which still needs to be explicitly handled in the current model. Moreover, the goodput of a flow can be negatively affected by TCP packet loss, which is not simulated and likely harder to implement since we do not simulate on the packet level. It is currently unclear to us when and how such behavior would impact the conclusions drawn from simulation results,

given that network conditions are often only one aspect of the systems that *faas-sim* targets. For example, in the evaluation of Skippy [RRD21], an optimizing container scheduler for geo-distributed scenarios, the network simulation played a significant role in identifying network bottlenecks. However, aspects such as the workload execution time or computational resource contention had a similar or higher impact, such that the additional fidelity of modeling TCP loss rate, which Morris [Mor97] showed can range from 1%-5% with 40-140 active TCP flows, would have no significant impact on the overall result of the evaluation. To understand this more and to be able to generalize, we invite the community to contribute baseline scenarios that can provide more evidence for whether this type of fine-grained simulation is needed for edge system evaluations.

**Fault modeling** *Faas-sim* supports out of the box two common classes of faults in distributed systems, specifically in the edge-cloud continuum: network degradation and replica failure. The default network simulation configuration throws an error when the bandwidth of a link is under a certain threshold. This simulates the exhaustion of network connections that can quickly saturate in resource-constrained environments (e.g., a Raspberry Pi 3 has a 100 Mbit/s bandwidth). Moreover, users can implement methods that randomly shut down functions or make nodes unavailable. The flexible *Function Simulators* can provide more fine-grained faults, such as resource exhaustion during invocation. For example, a function invocation might consume too much memory and cause the function and possibly others on the same node to shut down abruptly.

#### 4.2.3.3 User-defined simulation scenarios

A simulation encapsulates the configuration and the runtime state of a simulation. It requires two inputs: a *Topology* and a *Benchmark* which can be built using *request generators*.

**Topology** The simulation takes an *Ether* [RLF+20] topology as an input parameter. This enables users to generate network topologies in code and synthesize plausible infrastructures. As we described in Section 4.2.3.2, *Ether*'s fundamental concepts, which are tightly integrated into *faas-sim*, are nodes, links, cells, and the topology. A node represents a compute or storage device, and users can set the CPU and memory resources and label it to expose any additional capabilities (e.g., hardware accelerators). Links represent connections between nodes, such as access points, routers, and switches. Each link can be configured to have a specific data rate capacity. Cells represent a group of nodes, links, or other cells and help users to compose larger computing clusters. Topologies represent these components' graph representations and connect the nodes, cells, and links.

*faas-sim* provides several *Ether* topologies for common edge computing scenarios out of the box. These include: (a) an Industrial IoT scenario with distributed premises and shared cloud infrastructure, (b) an urban sensing scenario based on data from the Array of Things project [CBSG17] that connects distributed IoT nodes through mobile

Internet, (c) a multi-region cloud scenario with several cloud data centers connected via an Internet backbone. Users can use *Ether* to generate topologies for their scenarios, but the pre-defined scenarios can serve as a baseline. These topologies have been used to successfully evaluate edge computing resource management algorithms, as shown in Section 4.2.4.

**Benchmark** The *Benchmark* is a container for simulation parameters and the workload configuration to model a particular experiment scenario. Such scenarios could be: (1) generate $N$ sequential requests for function $f$; (2) for $t$ number of hours in simulation time, generate requests for functions $f_1$, $f_2$, and $f_3$ based on random workload patterns; or (3) for $t$ number of days in simulation time, create $u$ number of workload generators (representing users in the scenario generating requests) that generate requests for function $f$ in a sine-based workload pattern. *faas-sim* provides several such scenarios as examples but will ultimately have to be provided by the user depending on their use case and evaluation scenarios.

Specifically, the *Benchmark* is a SimPy process implemented in Python that prepares the evaluation environment by preparing the different workload types and then generates requests based on some logic. Simulation users must implement two methods: `setup` and `run`. The `setup` method has access to the *Environment*, and users can initialize objects, such as the container registry (i.e., register available images). The `run` method is invoked to bootstrap the actual simulation scenario and start the workloads. Simulation users can also deploy *Function Deployments* at this stage and use our request generators to produce workload on the deployed functions.

**Request generators** *faas-sim* allows programmable workload generators to produce realistic invocation patterns. These generators are located in the *request-generator* project on GitHub[32]. They are composed of an arrival process and a workload pattern. The arrival process determines the distribution from which the inter-arrival times, i.e., $\Delta t$ between requests, are drawn. The *request-generator* offers as of now: *constant*/*static* and *expovariate* processes. The workload pattern determines the average target requests per second (*rps*) pattern at a given time. Several workload patterns are supported, such as *constant*, *sine*, and *random walk*. Simulation users can combine the arrival process and workload patterns freely. These tools allow the modeling and creation of dynamic workload patterns and can simulate realistic ones. Moreover, *faas-sim* also supports files that contain a list of inter-arrival times. This enables users to export workloads and replay them, increasing the reproducibility of simulations and allowing users to model workloads without our tool. Specifically, simulation users can take existing datasets [tax18] or synthesize their own [KYMS20] and pre-process them into a list of inter-arrival times our simulator supports.

[32] https://github.com/edgerun/request-generator/

### 4.2.4 Evaluation

The evaluation consists of five parts: We showcase a selection of published works that have used *faas-sim* and highlight their results. This shows the ability of *faas-sim* to be used for the use cases shown in Section 4.2.2 and addresses the challenges introduced in Section 4.2.1. In this thesis we select certain use cases and refer readers to [RRFD23] for more.

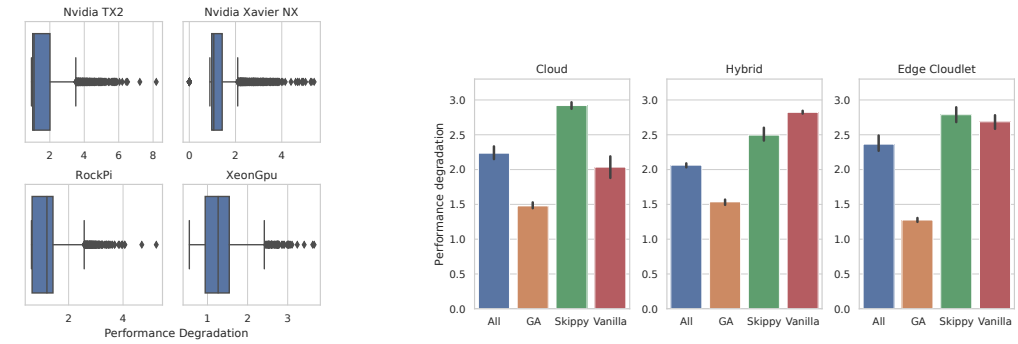
Moreover, based on our resource and performance modeling approaches, we highlight functions profiled included in *faas-sim* that can bootstrap scenarios without additional experiments in Section 4.2.4.2. The high heterogeneity in terms of performance duration of the traces emphasizes the necessity for stochastic models, as presented in Section 4.2.3.2. Section 4.2.4.3 presents our network simulation experiments comparing results from *Ether* with a testbed and ns-3 [RH10]. Section 4.2.4.4 shows the flexibility of our *Function Simulator* approach. Lastly, Section 4.2.4.5 presents experimental results on the resource usage of *faas-sim* for different scenarios.

#### 4.2.4.1 Simulation use cases

This section showcases selected works using *faas-sim* for the use cases outlined in Section 4.2.2.2. We highlight their approaches and results and reflect on the introduced challenges. These works range from ML-based multi-tenancy performance models, serverless orchestration strategies (including the evaluation aspects) and simulation of different infrastructure topologies. This use-case-based evaluation also shows that *faas-sim* tackles the simulation challenges.

**Application performance estimation** In [Rai21], we introduced a novel performance degradation prediction approach that estimates the performance degradation factor based on the current resource usage. This approach models the impact that multi-tenancy has on performance. The ML-based approach was developed by executing various functions (e.g., AI inference, CPU-heavy) on various devices (e.g., Raspberry Pi 4 to an Intel NUC).

Figure 4.16a shows the results of these experiments as boxplots, where the performance degradation is the factor of increase in contrast to the mean execution duration. Due to hardware limitations, not all functions were executed on all devices (e.g., on the Rock Pi only a few can run). Besides the performance degradation, we also measured the resource usage to create a resource vector for each function. The resource vectors are, in turn, used to pre-process the ML model's input. The input is a vector describing the current resource usage based on the function requests being executed at the moment and base resource usage, as well as hardware accelerators (e.g., GPU). The single-column vector has a fixed length of 34. It contains the sum, mean, standard deviation, minimum, maximum, $25^{th}$, $75^{th}$ percentile of CPU, GPU, network, and block I/O usage, mean memory usage, and the number of running containers. During the simulation, we determine which functions are currently being executed on each node and use the associated resource vector we presented in Section 4.2.3.2 to form the input. We used TPOT [OM19], an AutoML tool,




(a) Performance Degradation on edge devices

(b) Impact of scheduling on performance Degradation on edge devices

Figure 4.16: Performance degradation in real-world experiments and in simulation [Rai21].

to find a suitable ML pipeline, and validation results have shown good performance (i.e., the mean absolute error ranged from 0.02 to 0.09).

The model's accuracy shows that our resource model can be used to integrate sophisticated approaches to estimate the impact of multi-tenancy. Thus, it allows users to get reasonable estimates for their application performance through simulations and shows that *faas-sim* can support high configurability simulations by extending the core resource model. A limitation of this work is that it has to be investigated how well it generalizes to new, unseen functions, as the training and evaluation focused on a limited set of functions. Moreover, we trained an individual model for each device, thus increasing the effort to include new devices as they would need to be profiled accordingly.

**Evaluating serverless orchestration approaches** Evaluating serverless orchestration approaches is challenging on real-world testbeds, given the complexity of experiments necessary to draw generalized conclusions about the performance of such approaches. Simulation tools like *faas-sim* that can easily generate variations of computing infrastructure and workload patterns allow much more meaningful evaluations. The simulation should be able to mimic the spatio-temporal context of real-world applications. That is the distance between the request origin and the execution location and the varying request patterns over time. *faas-sim* has been used to evaluate different orchestration approaches, ranging from optimization-driven approaches that avoid performance interference and decrease performance degradation due to multi-tenancy [Rai21] to the evaluation of load balancer placement [Pal22]. Scheduling of data-intensive applications [RRD21]. Specifically, the results of [Rai21] used the performance degradation caused by multi-tenancy as a key performance indicator, as shown in Figure 4.16b. The figure shows different function orchestration approaches (i.e., All, GA, Skippy, and Vanilla) representing different scheduler configurations. Using the performance degradation model, simulation users can estimate the efficacy of their orchestration strategy to avoid this in multi-tenancy situations. In [Pal22], we have evaluated different load balancer placements,

thus focusing on the spatio-temporal context caused by varying request patterns arriving at the load balancer instances. We implemented a scheduling and scaling strategy to spawn load balancer instances dynamically and evaluated the function execution time with different configurations.

To summarize, *faas-sim* can evaluate different serverless orchestration strategies and, in combination with *Ether*'s topologies, considers spatio-temporal context. This is normally very difficult given the many parameters of the infrastructure and the environment that would have to be tuned to draw generalizable conclusions from experiments on real-world testbeds. Therefore, updating the models is advisable when adding new devices or functions.

**Resource planning** Resource planning helps platform designers and providers estimate their system's ability to handle different applications on a given infrastructure. It lets them plan before buying expensive hardware by simulating scenarios with varying hardware configurations. *faas-sim* has been used to conduct simulations on various infrastructure configurations that ranged from the cloud, industrial IoT, and Smart City infrastructures [RRD21], and on a range of infrastructures with different compositions of available hardware ranging from small factor computers to cloud-like servers [Rai21]. In both works, *faas-sim* was able to estimate the impact different hardware configurations had on the platform, and through the programmable scenario definition, various topologies were automatically created. Specifically in [Rai21], the authors extended the simulation and Ether by generating random topologies based on probabilities (i.e., users can specify how many nodes have an Intel Xeon or Intel i5 CPU equipped, etc.). Moreover, in [RRD21], we presented an initial evaluation of the Skippy scheduling system that is also used by default in *faas-sim* (see Section 4.2.3). Skippy introduces several soft constraints (scheduling functions) that are designed to help the scheduler make better container placement decisions in edge computing scenarios. Using *faas-sim*, we were able to show that the placements that Skippy makes lead to more scalable application deployments across different scenarios.

#### 4.2.4.2 *faas-sim* traces

We presented in Section 4.2.3.2 and Section 4.2.3.2 *faas-sim*'s resource and performance model, respectively. In the following, we want to highlight traces included in *faas-sim* and ready to use. The traces are split into performance (i.e., mean FET per invocation) and resources (i.e., resource usage per invocation). Table 4.2 shows our testbed, which contains a variety of typical edge devices. It includes modern embedded AI devices (i.e., Nvidia's Jetson series[33]), to resource-constrained Single Board Computers (i.e., Raspberry Pi, Rock Pi), and a PC with an Nvidia GPU (i.e., Xeon (GPU)). We collect traces from the following functions.

[33]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

| Device | Arch | CPU | RAM | Accelerator | Storage |
|---|---|---|---|---|---|
| Xeon (GPU) | x86 | 4 x Core Xeon E-2224 @ 3.44 GHz | 8 GB | Turing GPU - 6 GB | SSD |
| Intel NUC | x86 | 4 x Intel i5 @ 2.2 GHz | 16 GB | N/A | NVME |
| RPi 3 | arm32 | 4 x Cortex-A53 @ 1.4 GHz | 1 GB | N/A | SD Card |
| RPi 4 | arm32 | 4 x Cortex-A72 @ 1.5 GHz | 1 GB | N/A | SD Card |
| RockPi | aarch64 | 2 x Cortex-A72, 4 x Cortex-A53 | 2 GB | N/A | SD Card |
| Coral DevBoard | aarch64 | 4 x Cortex-A53 | 1 GB | Google Edge TPU | eMMC |
| Jetson TX2 | aarch64 | 4 x Cortex-A57 @ 2 Ghz | 8 GB | 256-core Pascal GPU | eMMC |
| Jetson Nano | aarch64 | 4 x Cortex-A57 @ 1.43 GHz | 4 GB | 128-core Maxwell GPU | SD Card |
| Jetson NX | aarch64 | 6 x Nvidia Carmel @ 1.9 GHz | 8 GB | 384-core Volta GPU<br>48 tensor cores | SD Card |

Table 4.2: Device type specifications

| Function | Jetson NX | Jetson Nano | Jetson TX2 | Xeon (GPU) | Intel NUC | RockPi 4 | RPi 4 |
|---|---|---|---|---|---|---|---|
| Fio | 13.77 | 19.64 | 4.31 | 1.14 | 1.12 | 21.99 | 27.81 |
| Mobilenet Inf. TFlite | 0.33 | 0.45 | 0.33 | 0.28 | 0.28 | 0.52 | 1.28 |
| Python Pi | 0.83 | 0.75 | 0.55 | 71.59 | 0.25 | 0.92 | 23.59 |
| Resnet50 Inf. CPU | 0.51 | 0.93 | 0.74 | 0.17 | 0.16 | 1.38 | 2.91 |
| Resnet50 Inf. GPU | 0.39 | 0.73 | 0.39 | 0.13 | × | × | × |
| Resnet50 Preprocessing | 6.08 | 7.95 | 6.39 | 2.66 | 2.53 | 7.65 | 19.5 |
| Resnet50 Train CPU | × | × | × | × | 197.45 | × | × |
| Resnet50 Train GPU | 142.0 | 847.17 | 228.12 | 32.13 | × | × | × |
| Speech Inf. GPU | 1.65 | 4.54 | 3.31 | 0.75 | × | × | × |
| Speech Inf. TFlite | 2.68 | 3.89 | 3.44 | 1.08 | 1.06 | 3.82 | 6.75 |
| TF GPU | 1.17 | 0.62 | 1.89 | 0.37 | × | × | × |

Table 4.3: Function traces included in *faas-sim*. Shows the mean FET in seconds over 100 requests.

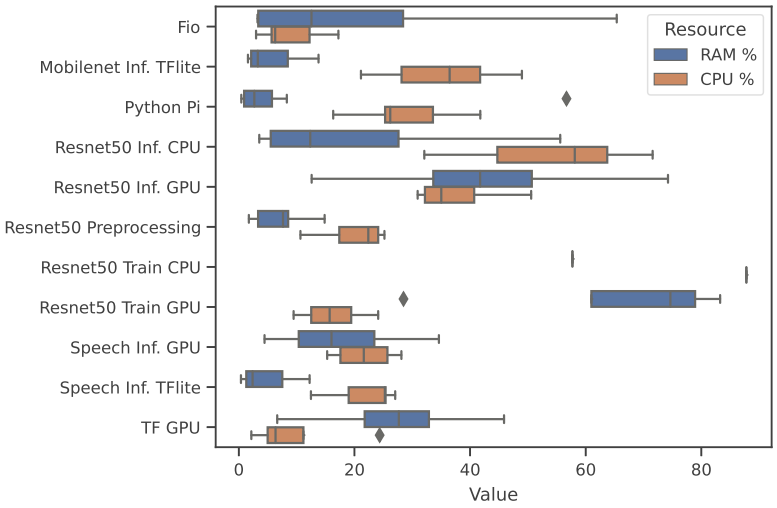



Figure 4.17: Mean CPU and RAM usage in % per function invocation across all devices [Rai21].

- AI-based object classification inference and training (i.e., Resnet50 [HZRS15], Mobilenet [HZC+17])
- AI-based speech-to-text inference (i.e., DeepSpeech [HCC+14])
- Matrix multiplication using Tensorflow [ABC+16]
- A Python-based function that calculates Pi
- I/O heavy workload (i.e., Fio[34])

TensorFlow offers two implementations: one that targets inference on resource-constrained devices (i.e., TFlite[35]) and the regular TF2 distribution[36]. We perform inference and training using TF2 on GPUs and inference using TFLite. Table 4.3 shows the average FET in seconds over 100 requests for each device. Note that × means the function has not been profiled on that particular device. This can have two reasons: the execution mode is not supported (i.e., because the device does not have a hardware accelerator), or the execution failed (e.g., due to low resources). The average CPU and RAM usage per request across all devices is shown in Figure 4.22. This figure shows that resource usage vastly differs across devices. Note that the *Resnet50 Train CPU* has only been executed on the Intel NUC.

These traces enable our performance and resource modeling approaches.

#### 4.2.4.3 Network simulation evaluation

As we described in Section 4.2.3.2, *Ether*, the network model underlying *faas-sim*, uses a flow-based model to simulate data transfer between nodes, as opposed to a packet-level simulation like ns-3. This trades off fidelity (the ability to simulate network QoS like packet loss) against performance (running simulations faster).

To that end, we demonstrate our networking simulation a) produces similar results to ns-3 when simulating node-to-node transfer of data that is characteristic of data-intensive serverless edge computing workloads [RRD21] (at least in simple scenarios), and b) accurately replicates real-world geo-distributed network scenarios such as the cloud testbed used in the evaluation of EMMA [RND18].

We should note that these experiments do not yet provide conclusive evidence that *any* edge computing networking scenario can be accurately modeled with *Ether* and *faas-sim* or that the reduced fidelity provides good enough results for all scenarios. As we argued in Section 4.2.3.2, and as we show in the evaluation, the extra fidelity may not be necessary to draw meaningful conclusions for the systems that *faas-sim* targets. That said, more evidence is needed to generalize this statement. Specifically, more experiments

[34] https://fio.readthedocs.io/en/latest/fio_doc.html
[35] https://www.tensorflow.org/lite
[36] https://www.tensorflow.org/

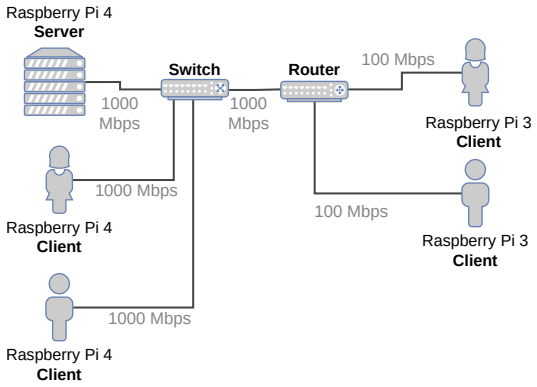



Figure 4.18: Testbed used for basic node-to-node data transfer experiments.

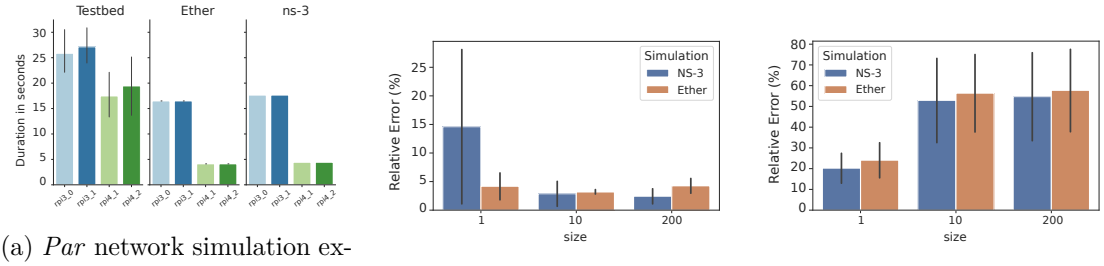
are needed to show how simulation accuracy behaves in scenarios where many flows compete for scarce networking resources for prolonged periods of time.

**Comparison to packet-level simulation** We first want to understand whether our flow-based network simulator can accurately replicate basic scenarios of node-to-node data transfer, which is typical in serverless edge computing systems like the ones presented in [RHM+19, RRD21]. This includes transmitting individual images for ML inference, video files over Internet uplinks, or pulling container images from worker nodes.

**Methodology** To that end, we run these basic scenarios on a real-world testbed with an emulated network topology. Then, we replicate the scenarios first on the baseline packet-level simulator ns-3 and compare it to running the experiments on our simulator.

Figure 4.18 shows the testbed setup and also indicates which devices act as servers and clients. Our testbed consists of three Raspberry Pi 4, two Raspberry Pi 3, an unmanaged network switch, and one EdgeRouter X. We model the network topology in ns-3 and *Ether* and execute the experiments in each simulator and on the testbed. Transferring data via HTTP is one of the main use cases in our simulation, as described in Section 4.2.3.2, so our experiments build on these technologies. Specifically, the scenarios consist of an Apache HTTP server and multiple clients downloading files using the HTTP client *curl*. The file size varies between experiment runs and is based on real-world observations:

- **1MB**: the median size of a graphical image from a desktop web page (as of February 2023) [htt].
- **10MB**: 90th percentile size of a video on the Internet [htt].
- **200MB**: represents a *Docker* container image that contains a serverless function workload.




(a) *Par* network simulation experiments with 200 Megabyte file

(b) Relative to the ground truth in the *Seq* experiments.

(c) Relative to the ground truth in the *Par* experiments.

Figure 4.19: Results of basic data transfer scenarios on testbed, ns-3, and *Ether*.

We perform two types of experiments we label *seq* and *par*. The *seq* experiment tests one client downloading a file from one server (i.e., one flow over one link). The *par* experiment tests the network behavior of multiple clients downloading the file in parallel from the same server (i.e., multiple flows over one link). We perform each experiment run with one or multiple clients downloading one file from the same server and repeat this procedure 400 times. However, due to ns-3's deterministic behavior, we only executed each experiment once.

**Results** Figure 4.19b and Figure 4.19c show the relative error between the average download duration measured on the testbed and from the simulations. Specifically, we collect the duration of each download from each client, and the figures aggregate the errors across all four clients (i.e., two Raspberry Pi 3 and two Raspberry Pi 4). Figure 4.19b shows that the relative error of the *Seq* experiments is relatively low. ns-3's variance with the 1 Megabyte file is high because it could simulate the Raspberry Pi 3 client accurately (i.e., 1.06% error), while the simulation error of the Raspberry Pi 4 was 28%. In contrast, the experiments with 10 Megabyte file size are reversed, and the Raspberry Pi 4 simulation was more accurate. However, the error of *Ether* stays under 7% across file sizes.

Figure 4.19c shows the relative error of the *Par* experiments which shows that the relative error can be high with increasing file sizes. However, both simulators show similar errors, whereas NS-3 is more accurate than *Ether*. We think this high error rate in the parallel experiments is mainly caused due to the testbed's components (i.e., Raspberry Pis) and the fast network congestion. Both simulators do not model or consider the performance or hardware and, therefore, can not accurately reflect the low performance of the resource-constrained devices. Figure 4.19a shows that both simulators produce similar and steady results, but the durations measured on the testbed vary greatly.

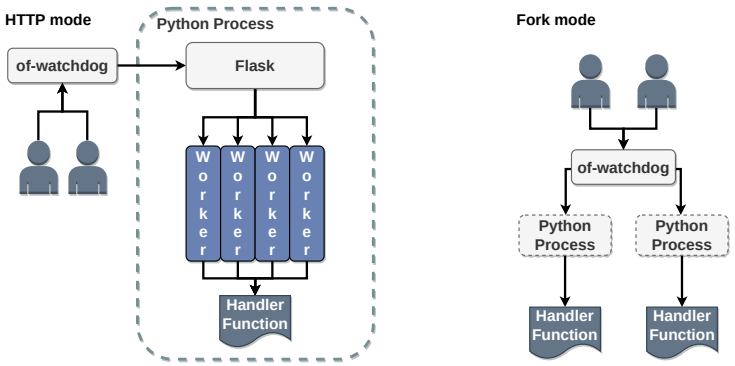



Figure 4.20: *of-watchdog* execution modes

#### 4.2.4.4 Flexible platform design

In the following, we show *Function Simulators* implemented in *faas-sim*. Specifically, they model OpenFaaS' reverse proxy (*of-watchdog*[37]), which acts as the base to implement functions in OpenFaaS. To implement a function in OpenFaaS, users implement a `handle` function that is called through a Go-based reverse proxy (i.e., *of-watchdog*). The execution mode of the reverse proxy determines how it invokes the `handle` function. We focus on two execution modes and assume the function is implemented in Python. Figure 4.20 shows the *HTTP* and *Fork* execution modes.

The *HTTP* execution mode starts a Flask HTTP server and redirects each call to this server. The HTTP server internally uses a queueing mechanism with a pre-defined set of workers that call the *Handler Function* [BMS20]. The *Fork* execution mode forks for each incoming request a new Python process that executes the *Handler Function*. The *HTTP* execution mode can cache expensive resources shared among the workers. The *Fork* mode is computationally more expensive and incurs higher latency because it starts a new Python process for each request. The following shows how these two modes are implemented in *faas-sim* as *Function Simulators*. The classes are `HTTPWatchdog` and `ForkingWatchdog` and extend the `Watchdog` class. The three (simplified) classes are shown in Figure 4.21. The `HTTPWatchdog` initializes during *setup* a SimPy queue and allows up to a user-defined number of concurrent requests. The *invocation* waits for a free worker, claims resources, executes the function, releases the resources, and frees the queue. Simulation users must implement the methods that model the function execution and resource usage. The `ForkingWatchdog` invokes the user's supplied methods. It is up to the simulation users to simulate the creation of the Python process accurately and also consider that function invocations may fail if there are too many because of resource exhaustion. However, this example shows the flexibility of *faas-sim*'s core simulation component, the *Function Simulator*, and its ability to mimic real-world behavior realistically.

[37] https://github.com/openfaas/of-watchdog


Figure 4.21: Classes implementing OpenFaaS' watchdogs.

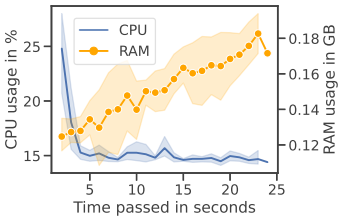
#### 4.2.4.5 *faas-sim* resource usage

Lastly, we want to demonstrate that *faas-sim* is sufficiently performant to simulate long-running scenarios even on developer machines. Moreover, understanding the resource usage of *faas-sim* helps when using *faas-sim* as a co-simulator in a system to make runtime decisions.

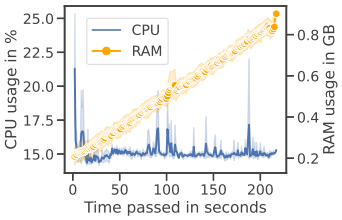
To that end, we execute a series of experiments to evaluate the resource usage of the simulator in terms of CPU, memory, and execution run time. Specifically, we model a Smart City topology with 15 edge clusters and one cloud cluster. Each computing cluster has one client (i.e., Intel NUC) and multiple devices (i.e., Xeon, Jetson TX2, Nano, and NX). This allows us to observe the resource usage over time. Each client sends up to 100 requests per second with a constant workload, and we deploy 30 function replicas of the aforementioned *Resnet50 Inference CPU* function. However, the number of requests each client sends differs *100*, *1000*, and *2500*. The total requests sent across the system are 1500, 15000, and 37500, respectively. We simulate these scenarios using the traces (i.e., FET and resource usage) shown before and deactivate the *Resource Monitor* for these experiments as our system does not use the resource metrics gathered during the simulation. The simulations are started natively (i.e., using `Python`), and the test host has 32GB RAM and an i7 7700K@4.2 GHz with four cores (and eight threads).

Each scenario is executed five times, and Figure 4.22 shows the CPU and memory usage of the host. Before conducting the experiments, we measured the baseline consumption of the host and only show the isolated memory usage of the simulation. The CPU results contain other processes, but due to Python's single-threaded execution model, the simulation constantly uses 100% of one CPU core (i.e., 13% relative to all available threads).

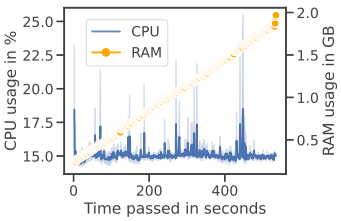
Memory usage increases over time and can reach up to 2GB during the simulation, which lasts over 8 minutes. While this seems problematic, we show in the following that the resource usage is constant with a few adjustments. Figure 4.23 shows the scenario in which each client sends 2500 requests. Contrary to the other experiment runs, we deactivated the logging framework (i.e., *Metrics*) and saw that the simulation's memory usage is much lower and grows more slowly. We also provide an optional logger implementation that continuously writes the internal data structures to disk and removes




(a) 100 requests per client.




(b) 1000 requests per client.




(c) 2500 requests per client.

Figure 4.22: CPU and RAM usage over three different experiments

the old ones, thus avoiding the memory increase observed in the results.

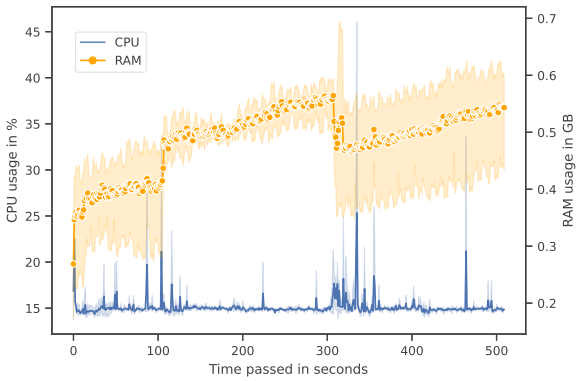



Figure 4.23: CPU and RAM usage with 2500 requests per client but no logging.

### 4.2.5 Related Work

#### 4.2.5.1 Specialized high fidelity simulators

Some simulators were implemented exclusively for evaluating a new orchestration strategy [LMFK23]. Similarly, particular simulations aim at near-identical outcomes as the ground truth for specific system components. For example, Eismann et al. introduce *Sizeless* for predicting memory-related configurations to cut client costs and improve

provider resource efficiency [EBG+21]. Later, Lin et al. propose formally modeling FaaS applications to aid the implementation of an accurate cost model [LMFK23].

The downside is that specialized simulators are costly and more dependent on their intended platforms and applications. Worse, in work where the simulator is not the contribution and is only a necessary side product, it may not generalize well outside its intended scope. Consequently, we provide default implementations that simulation users must populate with realistic traces to cover various applications and platforms. Since the components are interchangeable, it is possible to incorporate arbitrary methods for performance and resource modeling.

Importantly, this design decision allows *faas-sim* to retain relevancy as the state-of-the-art evolves and promotes a streamlined research directive.

#### 4.2.5.2 General edge-cloud simulations

Sphere [FCFMJ+20] is an extension of the SCORE [FCFMJ+18] simulator. Sphere aims to enable the implementation of topology- and orchestration models for edge computing and fast-paced simulation. They evaluate different platform architectures and cover many aspects of the edge-cloud continuum. However, a drawback they state to solve in future work is that their code needs to be more easily extendible. In contrast, *faas-sim* simulation users are encouraged to extend and modify the simulation. Wang et al. [WLL+21] present their open-source simulation platform SimEdgeIntel. The authors want to offer a simulation tool with a low entry barrier to implement new resource management strategies for Edge Intelligence due to the cross-platform and cross-language support. Their system model focuses on device-to-device (D2D) communication and considers network handovers in the simulation. Alwasel et al. [AJH+21] present IotSim-Osmosis, an extension of CloudSim to model and simulate IoT applications across the edge-cloud continuum. They envision a multi-layered architecture comprising IoT, edge, cloud, and software-defined wide area networks (SD-WAN). Their simulation supports a variety of communication protocols. The tool also supports custom policies to control components (i.e., task scheduling and routing). IotSim-Edge [JAA+20] also extends CloudSim and provides entities to model different edge-cloud aspects, such as mobility, energy consumption and IoT protocols (e.g., CoAP) and network protocols (e.g., 4G). The simulation scenarios heavily focus on IoT applications (e.g., stream processing) and offer a GUI to create scenarios. IotSim-SDWAN [AJH+20] focuses on providing accurate network simulations for edge-cloud systems. Specifically, they enable software-defined wide area networks (SDWAN) modeling and introduce a system model to replicate edge-cloud networks realistically.

The simulators above aim to simulate different application scenarios and their concepts. However, *faas-sim* can integrate concepts of them, such as focusing on IoT protocols. Therefore, the focus of this work is dedicated serverless simulations.

#### 4.2.5.3 Serverless simulations

To compare existing serverless simulators, Table 4.4 summarizes whether and to what extent they support the six use cases identified in Section 4.2.2.2. A ✓, ∼, and × indicate full, partial, or no coverage, respectively. We consider a feature partially covered when the work only provides rudimentary support (e.g., no support for fine-grained customization) or when it seems technically possible to cover the feature through manual extension.

*SimLess* by Ristov et al. [RHHP22] introduces abstractions for performing numerous experiments in various conditions without requiring extensive parameter configurations. Although SimLess considers heterogeneous environments and supports performance modeling for client programmers, it does not provide an interface to implement execution models for function invocation. Additionally, it does not support resource planning and serverless orchestration strategies to facilitate research on serverless platforms.

Mastenbroek et al. [MAJ+21] re-design and extend *OpenDC* [IAVB+17] to support serverless computing. The low-level nature of OpenDC naturally provides interchangeable interfaces but incurs higher complexity. SimLess and OpenDC contrast with each other. While SimLess focuses on client programmers and trades reduced configuration complexity with less flexibility, OpenDC focuses on platform providers with powerful modeling tools for highly configurable simulations of data centers. Additionally, the former focuses on client programmers, and the latter focuses on platform providers with powerful modeling tools for highly configurable simulation of data centers. FaaS-sim provides a middle ground between SimLess and OpenDC to support client programmers and platform providers by providing interchangeable high-level abstractions.

Jeon. et al., introduce *DFaasCloud* [JCSY19] as an extension of CloudSim [CRB+11]. We consider their work a proof-of-concept since their serverless support is rudimentary, and simulating complex workloads is infeasible. Moreover, the official repository did not see any contributions after its initial release. *SimFaaS* by Mahmoudi and Khazae [MK21] aids client programmers and platform providers estimate costs. Nevertheless, other than extensively covering cost prediction, it has several limitations. First, it explicitly only supports the scale-pre-request pattern, i.e., simulations with resource-based or concurrency value scaling are impossible. Second, focus on existing public cloud providers. In contrast, *faas-sim* is not limited to existing cloud platform providers by supporting simulations for hybrid edge-cloud environments and allowing simulation users to implement arbitrary serverless function orchestration strategies.

Lastly, we considered the work by Bhardwaj et al. [BB22]. and Manner et al. [MEBW21]. The former introduces *KubeKlone* [BB22], a simulation-based framework based on uqSim [ZGD19] focusing on a simulator to train AI-based operations for microservice deployments. The latter introduces a simulation tool for client programmers to find suitable configurations by quickly permuting various parameters. Nevertheless, they do not associate any open-source repository with their work.

To summarize, the main differences between *faas-sim* and the simulators presented in Section 4.2.5.2 and Section 4.2.5.3 are:

Table 4.4: Comparing simulators supporting serverless abstractions

| | SimLess | OpenDC 2.0 | DFaaSCloud | SimFaaS | *faas-sim* |
|---|---|---|---|---|---|
| Application Performance Estimation | ✓ | ✓ | ✓ | ✓ | ✓ |
| Performance Modeling | ∼ | ✓ | ∼ | ∼ | ✓ |
| Resource Planning | × | ✓ | ✓ | × | ✓ |
| First-class Serverless Orchestration | × | ∼ | ∼ | × | ✓ |
| Co-Simulation Driven Orchestration | × | ∼ | × | × | ✓ |
| Edge-Cloud Continuum | × | × | ✓ | × | ✓ |
| Simulation Configurability | Medium | High | Low | Low | Low-High |
| Topology Generation | × | UI | JSON, UI | × | Code[RLF+20] |

- *Simulation focus*: our system model resembles serverless computing and allows users to evaluate serverless edge platforms. The previous works do not always replicate a specific platform architecture (e.g., general edge-cloud simulations).
- *System models*: *faas-sim* aims to provide a playground for researchers and practitioners, allowing them to introduce new models based on our foundation and does not expect a specific resource model.
- *Application simulation*: to the best of our knowledge, our *Function Simulator* approach is a novel way of modeling applications and gives simulation users great flexibility. Others only allow users to specify the number of instructions required (e.g., CloudSim-based simulations) or only need the application duration (e.g., SimFaaS [MK21]).
- *Serverless Function Orchestration Strategies*, including request routing, scaling, and scheduling of function instances, are only modeled as first-class citizens in *faas-sim*. Inevitably, this also impacts the ability of the simulators to be used in co-simulation-driven orchestration strategies. Specifically, SimFaaS and SimLess do not allow simulation users to specify the function orchestration implementations, DFaaSCloud allows custom function schedulers but does not outline customization of the scaling component, and OpenDC 2.0 appears to only allow users to select the function scheduler and request routing implementations. However, not the autoscaler implementation [SK22].
- *Co-simulation-driven Orchestration* we consider OpenDC 2.0 and DFaaSCloud only partially fulfilling this criterion, as they allow us to inject custom orchestration components to some extent. Contrary to that, *faas-sim* not only enables users to customize the three major orchestration strategies, but it also uses the *skippy-scheduler* internally that works the same way the default Kubernetes scheduler does, thus allowing users to easily optimize scheduling approaches and apply them in the real world.
- *Simulation Configurability* is based on the parameters that simulation users can tweak. For example, SimFaaS and DFaaSCloud offer only high-level parameters that describe the general characteristics of the deployed functions (e.g., arrival rate,

execution duration, etc.). SimLess takes a different approach by observing the execution of functions on public cloud platforms and using the measurements to feed the simulation. OpenDC 2.0 has a high level of configurability as users design the data centers from the ground up and can tweak any aspect, which holds for *faas-sim* as well as the topology creation considers hardware characteristics but leaves it up to the simulation users to incorporate these details in the simulation. Therefore, *faas-sim* also allows performing simulations without much configuration, similar to DFaaSCloud and SimFaaS. Moreover, OpenDC 2.0 tries to model data centers on a very fine-grained level while *faas-sim* tries to model serverless edge platforms.

- *faas-sim* is the only one with first-class citizen support for generating topologies based on code. OpenDC 2.0 gives users an interface where they can manually create the layout of a data center, while DFaaSCloud offers an existing GUI and accepts JSON as input. SimFaaS and SimLess do not allow any topology modeling.

### 4.2.6 Summary of Serverless Edge Computing Simulator

Serverless edge computing is an emerging platform paradigm that extends the promising serverless computing paradigm. These paradigms abstract the infrastructure from developers and promise cost-efficient deployment through autonomous management. However, serverless computing platforms are cloud-centric and have yet to adapt to the emerging edge-cloud continuum. Combining all resources and enabling function execution across the continuum requires novel platform ideas that move away from the cloud-centric design and distributed and decentralized control and data plane. We identified two main tasks that each serverless edge computing platform has to implement: platform architecture and serverless function orchestration strategies. The edge-cloud continuum deepens the complexity of implementing them and differentiates them from cloud-centric platforms. To this end, we presented *faas-sim*, an open-source trace-driven discrete-event Python simulator that can support researchers and practitioners in developing and evaluating serverless edge computing systems. Our evaluation demonstrated that *faas-sim* is usable in a wide range of scenarios out of the box while allowing users to extend and modify it to their use case. We have shown that our networking simulation provides similar results to state-of-the-art packet-level simulators such as ns-3 and in an evaluation using a cloud testbed setup. However, these results do not necessarily generalize to more complex scenarios or imply that any scenario can be accurately modeled. More experiments are needed to show the accuracy of the network simulation in complex scenarios. Our trace-driven resource modeling approach for simulating workloads provides a flexible and accurate framework for modeling various workloads and heterogeneous computing platforms. While we have evaluated it using the herein presented functions, our single-tenancy and multi-tenancy approaches should be re-evaluated when adding new devices and functions or including new simulation aspects. *faas-sim* has been used successfully in several publications to evaluate different systems aspects, from scheduling performance comparisons to simulating large-scale distributed edge computing infrastructure to using

*faas-sim* as a co-simulator to optimize system parameters at runtime. *faas-sim* is part of the *Edgerun* project, an ecosystem of tools to help advance the research of edge-cloud system challenges. The simulator users can use these tools to automatically integrate their testbeds and profiling experiments to gather trace data for their simulation scenarios. In future work, we want to explore embedding *faas-sim* into the control loop of *operating* serverless systems. With a high-performance trace-driven simulator, we can incrementally build a digital twin of the real infrastructure and use trace data from real workload execution to refine the simulation model at run-time. The simulator can then be used as an engine to evaluate placement or scaling decisions stochastically.

## 4.3 Summary

This chapter has presented our end-to-end framework for testing Serverless Edge platforms in Section 4.1 and our simulator *faas-sim* in Section 4.2. Both work on similar domain models that allow us to transfer the real-world state into the simulation and unify them to build a self-adaptive platform. In the following chapter, we will demonstrate how we can utilize optimization methods to bootstrap orchestration parameters and tune them during runtime.

CHAPTER 5

# Self-Adaptive Serverless Edge Computing using Simulation

This chapter explores methods for building a self-adaptive platform using simulations. Section 5.1 presents our approach to tuning autoscaler thresholds during runtime using an optimization method. The approach uses our previously presented testing framework and simulation tool, which we combine and use to evaluate our runtime tuning approach on a real-world edge-cloud testbed. Section 5.2 explores a simulation-driven approach of generating synthetic datasets to train a prediction model for autoscaling, for which we introduce our novel domain model that captures the heterogeneity of the edge-cloud space.

Section 5.1 was first presented in [RND24], and Section 5.2 was first presented in a paper that has been accepted at IC2E 2025[1].

## 5.1 Simulation-based Runtime Autoscaler Parameters Adaptation

Serverless Edge Computing is growing in popularity, and while commercial providers are starting to offer edge-oriented products, much research is still being done on orchestrating functions (e.g., autoscaling). These approaches range from threshold- to AI-based strategies and support various Service Level Objectives (SLOs), such as Round-Trip-Time (RTT) and resource usage. However, the Quality of Service (QoS) continuously deteriorates due to the dynamic edge-cloud continuum and static parameterization of orchestration strategy parameters. Platforms must adapt the orchestration parameters

[1]Philipp Raith, A. Furutanpey, N. Lukic, V. Thurimella, S. Nastic, "EdgeCloudForge: Simulation-Driven Synthetic Dataset Generation For Proactive Serverless Edge Function Autoscaling", Accepted - IC2E 2025

during runtime to counteract this drift that causes SLO violations. To this end, we introduce the Orchestration Parameter Optimization Problem (OPOP), which aims to find parameters for orchestration strategies to minimize SLO violations. We propose a novel self-adaptive Simulation-based Scaling (*SimuScale*) approach that uses co-simulation to solve OPOP for autoscalers during runtime. *SimuScale* uses live monitoring data to feed the simulation and perform parameter optimization. Our Proof of Concept is integrated with Kubernetes and evaluated on a real-world edge-cloud testbed. While this work focuses on a threshold-based autoscaler, it can be extended to optimize other orchestration components (e.g., schedulers). Our experimental results show that *SimuScale* finds parameters that decrease RTT SLO violations between 15% and 40%. *SimuScale* also can reduce resource usage by 29.87% while maintaining the target 95*th* RTT percentile. Moreover, it can reduce variance caused by different request patterns, making orchestration strategies more resilient in realistic scenarios.

### 5.1.1 Introduction

Serverless Computing promises to prevent customers from manually managing function deployments by autonomously orchestrating function instances in response to incoming requests. Serverless Edge Computing [NRF+22, GND17] can be a key enabler for emerging application paradigms, such as Edge Intelligence [DZF+20], that require all resources in the edge-cloud continuum to work together [ZCL+19, RD21]. Serverless platforms build on orchestration strategies to autoscale function instances, schedule them, and route requests from clients. Orchestration strategies adapt functions to satisfy operational goals, such as service level objectives (SLOs), which include user- and platform-oriented ones. For example, users want low latency response times to reduce cost and satisfy function requirements. Platform providers try to satisfy customer SLOs and avoid penalties. At the same time, they aim to achieve high resource efficiency, making it challenging to balance between customer and platform provider goals.

Research in orchestrating deployments has been extensively discussed and evaluated in the literature. Autoscaling has evolved over the years, from grid computing to cloud computing, and nowadays moves towards the edge [ATC+21, RND23]. However, an issue that remains is finding appropriate parameters and drifting orchestration strategies. Proper parameterization of orchestration strategies, in general, is challenging. Autoscaling strategies can use various optimizations, ranging from heuristic- , threshold-, to AI-based, etc. [RND23]. Which can focus on different aspects, such as reducing data movement [RRD21], guaranteeing response times [LKM+22], or reducing resource usage [RRD+22]. However, the optimal performance of those strategies is bound to the proper parameterization, which depends on the environment (e.g., infrastructure, deployed functions, client request patterns, etc.) [SG19, RRD21].

Straesser et al. [SGvK+22] present several challenges for autoscalers in production, and one is finding optimal parameters, which can become increasingly complex with different approaches. Al et al. [AHSS13] show the impact of the varying CPU threshold for autoscalers on cost and performance. Calzarossa et al.[CMT20] conclude that the

performance of autoscalers depends on the configurations, which in turn is affected by the request patterns, giving reason to adapt configurations during runtime. Especially in Serverless Edge platforms spanning multiple geo-distributed clusters, request patterns can vary from one to another. For example, the Shanghai Telecom dataset [GWZ+20, WGZ+21, LZMW22], which contains real-world mobile user request patterns, shows a high variance of concurrent users over time, causing *Orchestration Strategy Drift*. Therefore, an open challenge is putting orchestration strategies into production and finding parameters to satisfy operational requirements during runtime to avoid *Orchestration Strategy Drift*.

These drifts are analogous to the concept of *Concept Drift*, established in Machine learning operations (MLOps) [KKH23]. Concept Drifts occur when the input changes over time and AI models need to be retrained. A common indicator is a decreasing model performance. In the same way, we observe increasing SLO violations due to outdated orchestration parameters as request patterns and environments change. To find appropriate parameters and prevent this drift, we introduce the *Orchestration Parameter Optimization Problem* (*OPOP*), which tries to minimize SLO violations by updating orchestration parameters during runtime.

To this end, we introduce a self-adaptive Simulation-based Scaling approach (*SimuScale*) that can efficiently tune orchestration strategy parameters during runtime. It uses a trace-driven open-source simulator *faas-sim* [RRFD23] to run the optimization process of finding autoscaling parameters and solving *OPOP*. *SimuScale* is implemented as Proof-of-Concept (PoC) that works in tandem with Kubernetes. The Co-Simulation allows us to evaluate new orchestration strategy parameters efficiently, thus enabling parameter optimization. It uses live monitoring data to replicate the state of the real-world system as closely as possible and can then use optimization techniques to find appropriate parameters. We present two approaches: a random parameter search and a gradient descent-based one.

The contributions of this section are as follows:

1. Based on the issue of *Orchestration Strategy Drifts*, we introduce our model to solve *OPOP*. This aims to minimize SLO violations by selecting optimal parameters for orchestration strategies.

2. We propose *SimuScale*[2], a self-adaptive approach to automatically adapting threshold-based autoscalers during runtime by using co-simulation. *SimuScale* works in tandem with Kubernetes, a container orchestration platform commonly serving as a base for open-source Serverless platforms. Because we use a simulation, we evaluate the impact of the time horizon we simulate ahead. Results show that a short time horizon (*i.e.,* 30 seconds), results in lower SLO violations for high-intensity request patterns.

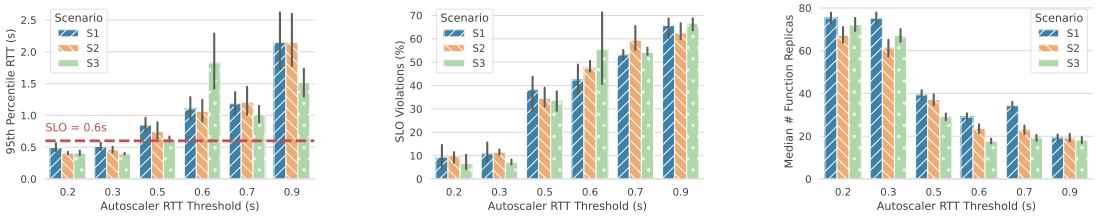
[2]https://github.com/edgerun/simu-scale

3. A random and a gradient descent approach are used to solve *OPOP* and integrated into *SimuScale*. We evaluate our approach on a real-world testbed that demonstrates *SimuScale's* ability to reduce SLO violations by tuning the autoscaler's parameters. Experiments show that *SimuScale* can reduce SLO violations between 15% to 40% while overcoming the challenge of dynamic request patterns.

Our evaluation on a real-world testbed shows that the co-simulation approach can facilitate self-adaptive platforms. The results indicate that our platform prototype can reduce SLO violations and resource usage effectively.

The remaining work is structured as follows. Section 5.1.2 introduces Orchestration Drifts in more depth and shows preliminary results that underline the issue of static autoscaler parameters. Section 5.1.3 introduces our model to solve *OPOP* that builds on three components: a Serverless Edge Platform, the Parameter Optimization, and the Co-Simulation. Section 5.1.4 presents details of our PoC implementation and the experiment setup. Section 5.1.5 shows and discusses the results obtained from our empirical evaluation of *SimuScale*. Section 5.1.6 presents related work, and and Section 5.1.7 introduces future work and concludes this work.

### 5.1.2 Motivation




(a) 95*th* Percentile RTT in seconds

(b) SLO Violations in % (Target: 0.6*s*)

(c) Median of running Function Replicas

Figure 5.1: Autoscaler Drift across different Request Patterns from Shanghai Telecom Dataset [GWZ+20, WGZ+21, LZMW22].

This section introduces the issue of drifting orchestration strategies and highlights its impact on functions based on results from preliminary experiments.

#### 5.1.2.1 Orchestration Strategy Drift

The parameters of an orchestration strategy depend on the implementation. They are crucial as they dictate the strategy's behavior and must be appropriately set to perform well in real-world systems [RRD21, SGvK+22]. For example, an autoscaler that tries to satisfy a certain round-trip-time (RTT) might neglect resource usage. Different request patterns make this even more difficult and render seemingly *obvious* parameters unsuitable, like setting the threshold to the target RTT. We showcase this behavior

in Section 5.1.2.2 but focus for now on the overall problem of drifting orchestration strategies and the impact on platform performance.

For that, we quantify a platform's performance through Service Level Objectives (SLOs). SLOs are measurable targets of Key Performance Indicators (KPI) that quantify deployed functions' Quality of Service (QoS). KPIs include the state of the function (e.g., availability, latency, cost) or the state of the infrastructure (resource or network usage). For example, an SLO might specify that the function's RTT should have a 95*th* percentile of 0.6s. A violation occurs any time the RTT is higher than 0.6s, resulting in penalties for the platform provider. Customers usually impose them, but we also consider platform-oriented ones, such as resource usage. The platform's task is to balance these goals and avoid SLO violations.

However, serverless systems are highly dynamic and not static. For example, request patterns can differ significantly across compute clusters, even within a single city. The Shanghai Telecom dataset, based on real-world mobile user request patterns, shows this [GWZ+20, WGZ+21, LZMW22]. Therefore, setting orchestration strategy parameters statically is not enough as systems change over time, causing more SLO violations to happen [SGvK+22, AHSS13]. We call this issue *Orchestration Strategy Drift* and investigate its impact on our testbed by running request patterns from the Shanghai Telecom dataset using different autoscaler parameters. Specifically, we look into the negative impact static parameters can have under varying request patterns on performance and resource usage.

#### 5.1.2.2 Drifts in Practice

In the following, we motivate and showcase the *Orchestration Strategy Drift* in practice on our testbed. We focus on the impact of different request patterns and extract three patterns from the Shanghai Telecom dataset [GWZ+20, WGZ+21, LZMW22]. The dataset contains mobile user connections of over 2000 base stations in Shanghai. We introduce three scenarios (*S1*, *S2*, *S3*) that combine request patterns of different base stations. *S1* being the one with the highest number of requests and *S3* the one with the lowest. We expect the RTT's 95*th* percentile of the deployed function to be around 0.6s. This includes network latency and any queuing delay in the system. Further details of the autoscaler and the experiment setup, including the testbed, can be found in Section 5.1.4.

As seen in Figure 5.1, we set the autoscaler's RTT threshold parameter in the range of 200ms to 900ms by using the 95*th* percentile. The results show that the SLO violations differ between scenarios, and in our case, no setting could have a steady RTT across all scenarios. We also observe that setting the RTT threshold to 0.6*s* (the SLO) has a significant variance between scenarios, making it challenging to find the right setting. Moreover, Figure 5.1c shows the resource usage (i.e., running replicas) per threshold setting. While lower RTT thresholds imply better performance, they also lead to higher resource usage. These experiments focus on a threshold-based autoscaler; other papers explore similar topics and indicate that parameterization is not easy and heavily depends

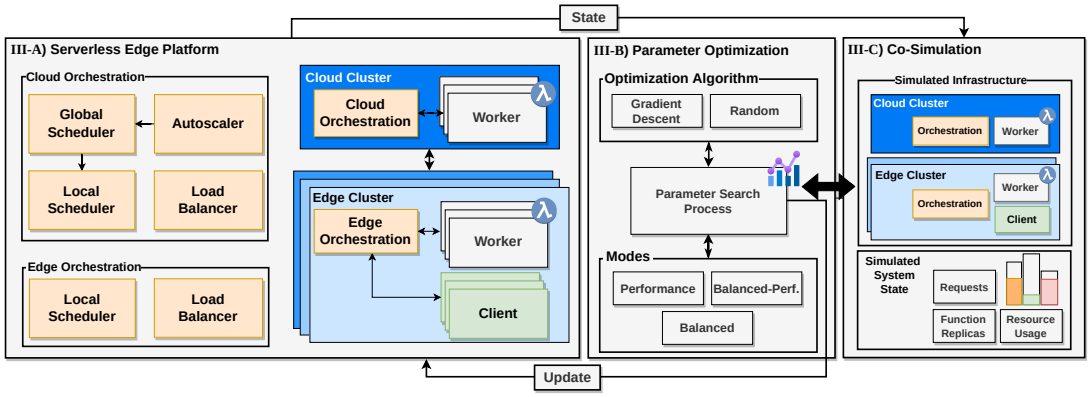



Figure 5.2: *SimuScale* - A high-level system overview

on the environment [RRD21, SGvK+22]. Based on those findings, we introduce our model to overcome *OPOP* and present *SimuScale*, a Simulation-based Scaling approach that we implement on a real-world system.

### 5.1.3 SimuScale - Orchestration Parameter Optimization

Serverless Computing platforms, responsible for avoiding SLO violations through function scaling, scheduling, and request routing, grapple with the complexity of maintaining optimal orchestration parameters over time. We previously identified this issue as *OPOP* and introduce our model to solve it, which is an unconstrained optimization framework. Its core objective is to find orchestration strategy parameters that minimize the SLO violations. The SLO violations measure the discrepancy between the observed KPIs and the desired SLO targets. Therefore, the goal is to minimize a function $f(\mathbf{x})$, $\mathbb{R}^n \to \mathbb{R}$, where the input $\mathbf{x}$ represents the orchestration parameters and $f$ estimates the SLO violations resulting from those parameters for a given scenario. We leave the decision variable $\mathbf{x}$ unconstrained to emphasize the general applicability towards various orchestration strategies. $f$ uses a function $h$ to estimate the KPIs based on $\mathbf{x}$ which are put into a cost function $v$ to calculate the SLO violations (i.e., the discrepancy between the target SLO and the observed KPIs).

Figure 5.2 depicts the individual components of our model to solve *OPOP* — *SimuScale*. The three high-level components contain the real-world Serverless Edge Platform (Section 5.1.3.1), the Parameter Optimization (Section 5.1.3.2), and the Co-Simulation (Section 5.1.3.3). The parameter optimization runs at intervals and uses co-simulation to optimize the parameters and minimize SLO violations. The Co-Simulation receives the latest platform state to mimic the real world as closely as possible. This includes the orchestration components, clients, and worker nodes on which function instances run. The Co-Simulation represents the implementation of the function $h$ to estimate KPIs, while the Parameter Optimization uses different *modes* that describe the implementation of the cost function $v$. The Co-Simulation enables us to perform *what-if* scenarios considering

the platform's state, the request pattern, and parameters. We take a snapshot of the current state, transform the simulation so that it can be used, and execute scenarios based on the known request pattern. The optimization uses Co-Simulation to find suitable parameters, minimize the SLO violations, and send them to the platform, where the orchestration components are updated. Next, we lay out the details of each component by starting with the Serverless Edge Platform.

#### 5.1.3.1 Serverless Edge Platform

The Serverless Edge platform is geo-distributed and consists of multiple clusters. It is based on a decentralized orchestration architecture [RRD+22] and includes worker nodes that host function replicas and clients that spawn requests. Orchestration strategies include schedulers, autoscalers, and load balancers that route client requests to the function replicas. It consists of a decentralized autoscaler implementation in which each cluster runs one instance. Autoscaler instances monitor their cluster and scale in or out by sending messages to the global scheduler. The global scheduler determines which cluster should start or remove function replicas. A local scheduler selects a worker node to start a new function replica. These build the fundamental components of our envisioned Serverless Edge Platform. However, we want to emphasize that *SimuScale* is not constrained to this particular architecture. For example, *SimuScale* is not concerned with the implementation details of schedulers, autoscalers, or load balancers. The implementation of $h$ is responsible for capturing the platform's logic, including orchestration and compute clusters, to estimate the KPIs. To this end, we present our Parameter Optimization approach that uses Co-Simulation to run an optimization algorithm and updates the real-world components afterward, as seen in Figure 5.2.

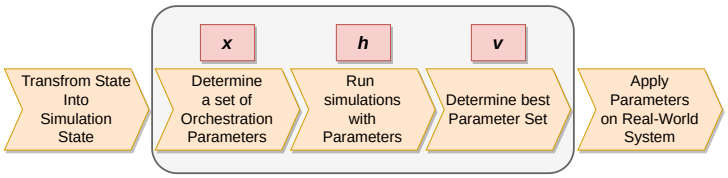



Figure 5.3: Parameter Optimization using Co-Simulation

#### 5.1.3.2 Parameter Optimization

We introduce the Parameter Optimization process to solve *OPOP*. As shown in Figure 5.2, the Parameter Optimization consists of three parts: the Optimization Algorithm, The Parameter Search Process, and the Modes. The Optimization Algorithm is used to find parameters that minimize the SLO violations. The Parameter Search Process takes the Optimization Algorithm and a Mode as input and uses the Co-Simulation to find parameters that reduce SLO violations, effectively solving *OPOP*. The Modes can be viewed as implementations of the cost function $v$ and the Co-Simulation as an implementation of $h$.

A basic approach to finding parameters is depicted in Figure 5.3. First, we transform the real-world state into the simulation state. Then, we generate parameters based on the current ones. Next, we run simulations for each parameter set, calculate the SLO violations for each setting, and choose the parameter set with the lowest violations. We apply these parameters to the real world. This builds the base implementation of our model to solve *OPOP*. The orchestration parameters represent the input, $\mathbf{x}$, the Co-Simulation represents the implementation of $h$, while the step of determining the optimal parameter set is the implementation of $v$. However, we need an optimization approach to find parameters that minimize SLO violations, which consists of the optimization algorithm and the cost function.

We use two optimization algorithms: a random parameter search and one based on the Gradient Descent (*GD*) [Rob51] algorithm. The *Random* approach creates $n$ parameter sets and starts $n$ simulations in parallel. Each parameter set is generated randomly by applying a percentage adjustment to each parameter. Specifically, we draw a random percentage number from a uniform distribution $[-15, 15]$ and apply the adjustment to the current orchestration parameter value. The *GD* approach approximates the gradient of the cost function by adding and subtracting an epsilon value $e$ to each parameter $x \in \mathbf{x}$, assuming $\mathbf{x}$ is of length $m$. We approximate the gradient $\frac{\partial f}{\partial x_i}$ using the central approximation method.

And then apply the update, with a learning rate of $\alpha$:

$$x_i = x_i - \alpha \cdot \frac{\partial f}{\partial x_i}$$

In every iteration, we perform these steps for each $x_i \in \mathbf{x}$.

These two approaches differ in complexity in finding suitable parameters. For example, Gradient Descent has two stopping criteria and simulates the system sequentially to determine the parameters. The random approach only iterates once and is, therefore, easy to parallelize, whereas we can only parallelize each iteration of the Gradient Descent. Besides the approaches to perform the optimizations, we also need to determine the *best* parameter settings. For example, how can we determine which configuration is the most suited? To answer this, we introduce *modes* that dictate how results are judged. Specifically, this addresses implementing the function $v$. $v$ acts as an objective function and calculates the SLO violation cost based on the observed KPIs.

We introduce three modes that vary in the metrics they use and the goal they follow. The first one, *Performance*, only takes the RTT into account by using the $95th$ percentile to calculate the difference to the configured SLO target ($SLO_{RTT}$). This *mode* focuses on performance and, therefore, will have increased resource usage while keeping performance SLO violations down. The other two modes use, in addition, a second objective function that takes the resource usage into account by calculating the difference to a target resource usage ($SLO_{CPU}$). Resource usage is the average number of CPU cores used,

and the goal is a percentage of the target resource usage. Based on that, we introduce a cost function that combines performance and resource usage by building the weighted sum of the deviation from the target:

$$v_{\text{combined}} = \frac{RTT_{P95}}{SLO_{RTT}} * w_p + \frac{CPU_{mean}}{SLO_{CPU}} * w_r$$

This allows us to balance the importance of each error and introduce the remaining two modes: *Balanced* and *Balanced-Perf.*. *Balanced* sets $w_r = 1$ and $w_p = 0.5$, while *Balanced-Perf.* reverses the weight settings.

#### 5.1.3.3 Co-Simulation

The Co-Simulation estimates the platform's behavior (i.e., KPIs) based on the orchestration parameters. For this, we transform the state of the real-world platform in the Co-Simulation and run different simulation scenarios. It allows us to simulate different scenarios ahead of time and use the KPIs to calculate the SLO violations, as shown in Section 5.1.3.2. In the following, we describe the transformation of the real world into the simulation state. The mapping includes orchestration strategies, worker nodes, clients, and network connections between clusters and individual nodes, as shown in Figure 5.2. The co-simulation uses the monitoring messages from *telemd* and *telemd kubernetes adapter* that were introduced in Section 4.1. The simulated system state contains the latest information about function replicas, requests (i.e., completed function invocations), and resource usage. Specifically, the simulation starts with a *Topology* that resembles the real-world infrastructure. The information about finished requests contains the start and end timestamps from the client and the actual function execution (i.e., the duration without any queuing), which load balancers were used, and the function replica that processed the request. Function replica state includes the worker node they were placed on and the state (e.g., running, deleted, etc.). It is also important to transform the function replicas currently being scheduled (i.e., pending) and the resource usage, such as CPU and memory usage. The simulated infrastructure and system state can be extended to incorporate other aspects. *SimuScale* constantly observes the latest state to perform the transformation every time the Parameter Optimization is executed.

#### 5.1.3.4 Challenges

An issue prevalent across many orchestration implementations is the challenge of setting the initial parameters. The impact of those can be crucial to the overall performance of functions and the resulting resource usage. A challenge for *SimuScale* is the ability to self-adapt from seemingly wrong settings, which can be seen as convergence to the optimal value and is important for *SimuScale's* ability to handle worst-case scenarios. Therefore, validating whether *SimuScale* can handle this is essential.

Another challenge is to set the right *Simulation Time Horizon*. That is, how much time should each scenario simulate? For example, if we set the *Simulation Time Horizon* too short, we might miss critical changes in the request pattern. On the other hand,

each simulated scenario might take longer with an increasing *Simulation Time Horizon*, thus resulting in fewer updates. Another aspect to consider is when putting *SimuScale* into production; tools must be used to estimate future request patterns, which probably worsen with longer time horizons. Therefore, finding the right *Simulation Time Horizon* setting is critical.

Moreover, we investigate what impact *SimuScale* has on SLO violations and resource usage compared to the static deployment of orchestration strategies. Especially when considering different request patterns and whether the modes are deterministic regarding their goal. We are addressing all these questions and challenges, and our real-world implementation of *SimuScale* in the following.

### 5.1.4 Implementation & Evaluation

We evaluate *SimuScale*[3] using the two optimization approaches: Gradient Descent (*GD*) and *Random* on a real-world testbed. We compare it to the *static* case, where parameters are only set once initially. This serves as a baseline and replicates the behavior expected nowadays on serverless platforms. The following outlines our implementation of *SimuScale* and details of the experiment setup.

#### 5.1.4.1 Serverless Edge Platform

Our PoC Serverless Edge platform is built on Kubernetes, a container orchestration service, and is split into multiple clusters - from cloud to edge. Users can send requests to the load balancers, which forward them to running function replica instances, which the autoscalers and schedulers manage. *telemd* pushes every second monitoring data via Redis' publish/subscribe feature. The orchestration components subscribe to these metrics and are all implemented in Python, each using an in-memory cache for the system state. The system state includes resource usage, information about requests, running application instances, etc.

The orchestration components run as individual containers in the cluster for which they are responsible.

**Orchestration Strategies** To comply with our model presented in Section 5.1.3.1, we must implement an autoscaler, global and local schedulers, and load balancers for the decentralized orchestration architecture. The autoscaler implementation mimics the official Kubernetes Horizontal Pod Autoscaler[4], which is commonly used in open-source serverless platforms, such as KNative[5]. We denote this strategy as *HPA*. It is based on the round-trip-time ($RTT$) of requests per cluster. Based on the current metric value, the *HPA* estimates how many replicas are needed to satisfy the threshold value. The

[3] https://github.com/edgerun/simu-scale
[4] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[5] https://knative.dev/docs/

threshold value is a parameter that we denote as $thr_{RTT}$. It uses the following formula to calculate the desired number of function replicas:

$$desiredReplicas = currentReplicas \times \frac{RTT_{P95}}{thr_{RTT}}$$

$RTT_{P95}$ is the $95th$ percentile of the observed RTT. Other parameters include *reconcile interval*, *lookback*, and a *percentile*. The *reconcile* interval determines the time the autoscaler waits before executing again, the *lookback* determines how far we look into the past when calculating the current value, *i.e.,* $RTT_{P95}$. The threshold ($thr_{RTT}$) is the parameter we optimize in each experiment. HPA and open source platforms that re-use it default to a *reconcile interval* of 15 seconds [JBP+23, Ope16]. We set the *percentile* to 95 and the *lookback* to 10 seconds across all experiments.

If the autoscaler decides the platform should create or remove function replicas, it sends a message to the global scheduler. The global scheduler implements a simple network-aware strategy by first prioritizing clusters from which the scale action stems. If the origin cluster is out of resources, it chooses the nearest cluster based on network latency. The local scheduler is implemented like the default Kubernetes scheduler, filling up all worker nodes while balancing out used resources. Both schedulers are kept simple on purpose to focus on the autoscaler and the optimization of its parameters.

#### 5.1.4.2 Co-Simulation

We use the open-source trace-driven Serverless Edge Computing platform simulator, *faas-sim* [RRFD23], to replicate the real-world in the cluster. However, to limit the scope of this work, we make the following assumptions. (1) we know the workload pattern, (2) we know the network configuration between nodes and clusters, (3) we do not simulate any node failure that can happen in the real world, and (4) we assume that all function images are already stored on each node. These assumptions would otherwise require solutions outside this work's scope, as we want to focus on the efficacy of the co-simulation's ability to adapt to real-world orchestration components. Specifically, we need a simulator that can replicate the behavior of our components in the simulation. We use a Python API to implement the core logic of all orchestration components once and reuse it in the simulation and the real-world[6]. The orchestration strategy implementations can be found on Github[7]. Besides avoiding code duplication and guaranteeing parity in terms of logic, the simulator uses real-world traces to feed the simulation. In contrast to the popular CloudSim [CRB+11], it does not rely on the knowledge of how many instructions an application takes but rather on real-world measurements. In our case, we conduct profiling experiments to determine the execution time of the deployed function and fit a logarithmic distribution from which we sample [RRP+22, RRFD23]. The profiling experiments invoke the function that we use for the evaluation on each host and allow

[6] https://github.com/edgerun/faas
[7] https://github.com/edgerun/faas-optimizations/

us to fit a logarithmic distribution from which we sample in the simulation. The real advantage of this approach lies in the support for future applications that can be observed in a black-box manner without counting executed instructions. A challenge of such a system is to communicate the state efficiently. The Co-Simulation runs in a central location and always keeps the global state in-memory. Therefore, powerful machines can be used to execute the optimization.

#### 5.1.4.3 Parameter optimization

We implement the Parameter Optimization through a *Random* and *GD* approach to minimize the SLO violations of $f$. The input $\mathbf{x}$ of $f$ contains the target duration ($thr_{RTT}$) of each autoscaler instance. The *Random* approach generates $n$ new autoscaler parameters that are each evaluated through a simulation. In contrast to the simulations in *GD*, we can execute all $n$ simulations at the same time. Specifically, we generate *5* parameter sets, and each autoscaler parameter is randomly set in the range of $\pm 15\%$ of the original value. The *GD* settings were chosen based on a set of preliminary experiments. As mentioned, we must avoid long-running optimizations to avoid updating the autoscaler based on outdated data. Therefore, we set the number of iterations to *3*, the learning rate to *1e-1*, and the epsilon value to *0.2*. These settings must be adjusted to the underlying simulation and have proven feasible for our setup. Besides the optimization approaches, we also implement three *modes*, already introduced in Section 5.1.3.2: *Performance*, *Balanced*, and *Balanced-Perf*. *Performance* only considers performance SLO violations, *i.e.,* using the RTT, while the other two incorporate performance and resource aspects. To this end, we set weights for *Balanced* and *Balanced-Perf.* that determine their focus. The former favors resource usage (i.e., $w_r = 1$, $w_p = 0.5$) and the latter reverses them (i.e., $w_r = 0.5$, $w_p = 1$). The weights are set according to their goal. For all modes $SLO_{RTT} = 0.6$ and $SLO_{CPU} = 50\%$. That means all modes have 0.6s as $95th$ percentile target and should not use more than 50% of available CPU cores. We evaluate all approaches and modes, resulting in six combinations.

#### 5.1.4.4 Experiment Setup

Besides implementing *SimuScale*, we also highlight details of the remaining experiment setup. This ranges from the testbed specification to the function we use to test *SimuScale* and the request patterns synthesized from a real-world dataset.

**Testbed** The experiments run on a real-world testbed that consists of three compute clusters using an end-to-end evaluation framework for edge-cloud resource management [RRP+22]. We denote the edge clusters as *EC1* and *EC2*, while the one mimicking a cloud is called *CC*. Our testbed contains 15 VMs with different sizes to mimic a heterogeneous environment, similar to related works [TDFS21, RCLN20]. Each VM has between 2 to 8 cores at 2.1GHz and 3 to 16GB of memory — we set the platform to spawn at most 90 function instances. We use the Linux network traffic shaping tool *tc* to emulate network latency between the clusters. Between cloud and edge clusters, we

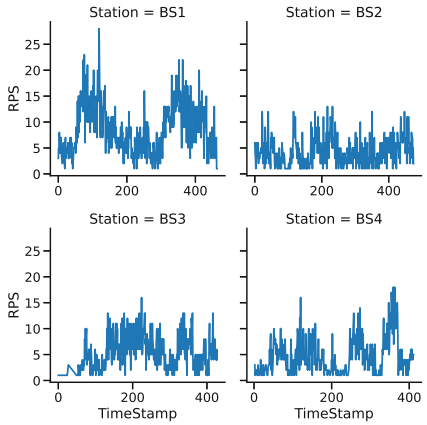



Figure 5.4: Base station request patterns

inject a network latency of 60ms, and between edge clusters, 30ms. The clients connect to the Edge Clusters and send the requests to the respective load balancer instances in each cluster.

**Function** The deployed function is CPU-bound and calculates a specified number of digits of $\pi$. This enables us to mimic short-running CPU-focused functions and vary the execution duration. Due to system delay, we expect the $95th$ percentile to be around 600ms. As we do not focus on the most optimal function placement in this work, we expect the cloud cluster to host function replicas and incur network overhead. The function is chosen to mimic a low-latency AI inference service, such as is typical for Edge Intelligence [RRD+22].

**Request Patterns** An important part of our experiments is the use of request patterns that exhibit different degrees of requests over time. The request patterns are based on the Shanghai Telecom dataset [GWZ+20, WGZ+21, LZMW22], and we extract them from four base stations. We chose the base stations based on the number of connected users and their patterns. Specifically, we have taken the profiles of the following base stations and assigned each an ID from which on we use to refer to them: *31.218201/121.487151* (BS1), *35.379598/116.072359* (BS2), *31.129955/121.336848* (BS3), *31.395915/121.363062* (BS4). We synthesize request patterns for our experiments by extracting each base station's general pattern over 48 hours and trimming it to 10 minutes of actual requests. The patterns per base station are shown in Figure 5.4. Base station A has the highest requests ($\sim$ 4000), while the others have each around $\sim$ 2000. The requests per second are based on the number of users connected at each point, each sending three requests. The interarrival times are drawn from a Poisson distribution. Based on these profiles, we create three scenarios that assign a request pattern to an edge cluster in our testbed. The scenarios and request pattern mappings are shown in Table 5.1.

| Scenario | EC1 | EC2 |
|---|---|---|
| S1 | BS1 | BS2 |
| S2 | BS3 | BS4 |
| S3 | BS2 | BS4 |

Table 5.1: Request pattern to Edge Cluster mapping.

### 5.1.5 Results

#### 5.1.5.1 Key Performance Indicators

In the following, we focus on three KPIs: SLO Violations in %, 95*th* percentile of the Round Trip Time in seconds, and the median number of running replicas over the experiment. The SLO violations are calculated by grouping the requests into windows (*i.e.,* 1 second) and calculating the 95*th* percentile. A violation occurs if the value exceeds the SLO (*i.e.,* 0.6s). Based on that, we count how often an SLO violation occurred relative to the complete experiment.

#### 5.1.5.2 Simulation Time Horizons

First, we look into the impact the *Simulation Time Horizon* has on the results. For this, we run shortened experiments with the *Balanced-Perf.* mode using four duration settings that we simulate ahead: *30*, *60*, *120* and *240*. The results of these experiments decide which setting we use in the remaining experiments. Figure 5.5 shows the median SLO Violations in (%) across the duration settings and scenarios. We observe that shorter *Simulation Time Horizons* positively impact the SLO violations. Specifically, the two scenarios with more requests (*i.e., S1* and *S2*) had the lowest median SLO Violations with the duration set to *30* seconds. However, a positive trend exists when looking at *S3*, where a longer time horizon lowers the median number of SLO violations and reduces the variance. We believe these results confirm our assumption of shorter *Simulation Time Horizons* allowing the system to update more often, making it more resilient against bursty workloads. While longer *Simulation Time Horizons* allows the platform to find more suitable parameters in case the workload is relatively stable. We select the *30* seconds *Simulation Time Horizon* as our default for the remaining experiments.

#### 5.1.5.3 Static vs. Co-Simulation

Next, we look into the results when setting the initial autoscaler RTT threshold to 0.6s (*i.e.,* $thr_{RTT} = 0.6$). The results determine whether *SimuScale* can reduce the SLO violations and its impact on resource usage. Figure 5.6 shows that *SimuScale* can reduce the SLO Violations by up to 15% on average across all scenarios when using the *Balanced-Perf.* mode and *GD* optimization in comparison to *Static*. Moreover, the minimal exhibited variance shows that *SimuScale* is effective across scenarios. However, an unexpected result is that the *GD* approach using the *Performance* mode was not

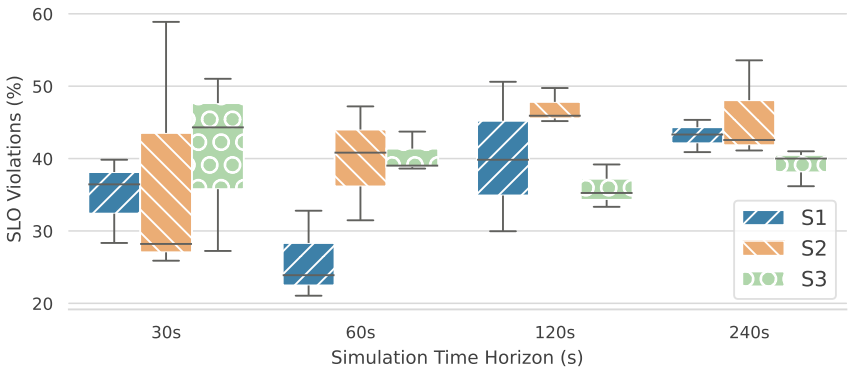



Figure 5.5: SLO Violations under different Sim Time Horizons

able to reduce the SLO violations as much as the *Random*. The resource usage in Figure 5.7 shows that these two had the highest number of replicas. While this should imply a better performance, it might be less beneficial in edge-cloud systems to spawn that many function replicas due to network latency. We suspect that the *Performance* settings spawned many replicas at the wrong time. Therefore, overall RTT increased in these cases. However, we can also see in Figure 5.8 that the RTT for *Performance* and *GD - Balanced-Perf.* were lower than the *Static* setting. Based on that, aggressive modes, such as *Performance*, can harm the overall SLO compliance in contrast to those more reluctant to increase resource usage in favor of performance. The *Balanced* modes, while using the least amount of replicas, performed worse in terms of performance. However, looking at the SLO Violations, it becomes clear that using *GD* for optimization reduces variance. We conclude that *SimuScale* can effectively reduce SLO violations while balancing performance and resource usage, *i.e., Balanced-Perf.* only used on average 30% more replicas but keeping the resource usage below the $SLO_{CPU}$ of 50% while reducing the SLO violations on average by 15%.

#### 5.1.5.4 Impact of Thresholds

The following section covers the other two initial autoscaler RTT thresholds we set. Specifically, we set $thr_{RTT} = 0.3$ and $thr_{RTT} = 0.9$. The first setting allows us to observe the platform's performance if the threshold is set much lower than the SLO, and the second is when the initial threshold is higher. To examine *SimuScale's* ability to adapt the threshold, we want to show it can reduce resource usage while keeping the RTT low and update the threshold setting to reduce SLO violations substantially.

First, we observe the results when setting $thr_{RTT}$ to $0.3s$. Figure 5.9 shows the median $95th$ percentile of the observed RTT over all scenarios and experiments. Such a low threshold positively impacts the performance of the *Static* autoscaler. It achieved a $95th$ percentile of under *0.6s* without updating the threshold. At the same time, the *GD* approach using the *Balanced-Perf* mode achieved the same with some outliers. The other

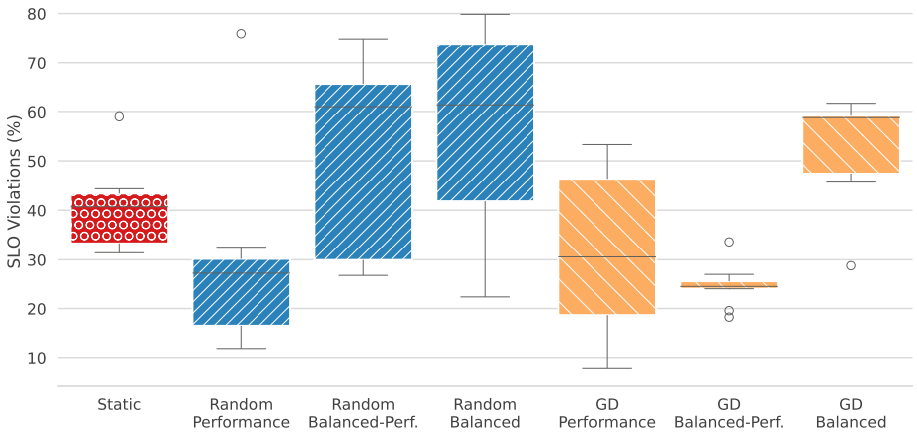



Figure 5.6: SLO Violations - $thr_{RTT} = 0.6s$

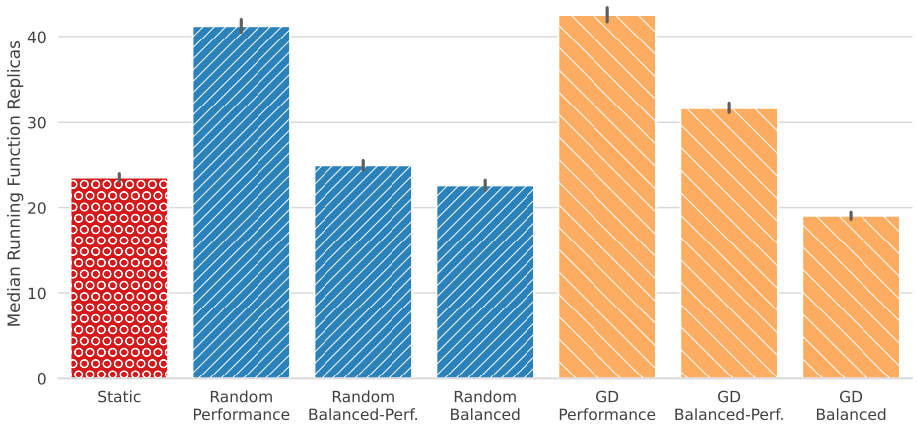



Figure 5.7: Resource Usage - $thr_{RTT} = 0.6s$

approaches diverged more from the SLO. Before looking more into the *Balanced-Perf* results, the *GD - Balanced* approach was able to have a similar SLO violation count in the *S1* scenario. In this case, we suspect the high amount of traffic caused the optimization to stay at a low threshold, while in other scenarios, it resulted in increasing thresholds.

Moreover, the *GD - Balanced-Perf.* approach was able to have only 8% SLO violations in *S3*, while the *Static* approach had an average of 6%. Therefore, *SimuScale* was also able to handle a lower threshold setting and, as Figure 5.10 shows, was able to reduce the resource usage significantly in comparison to the *Static* approach. Especially in the just described scenario, where *SimuScale* only had 2% more SLO violations, it could do so with a 29.87% resource usage reduction. The resource usage decreased overall in all approaches and modes, which explains the rising SLO violations and high RTT.

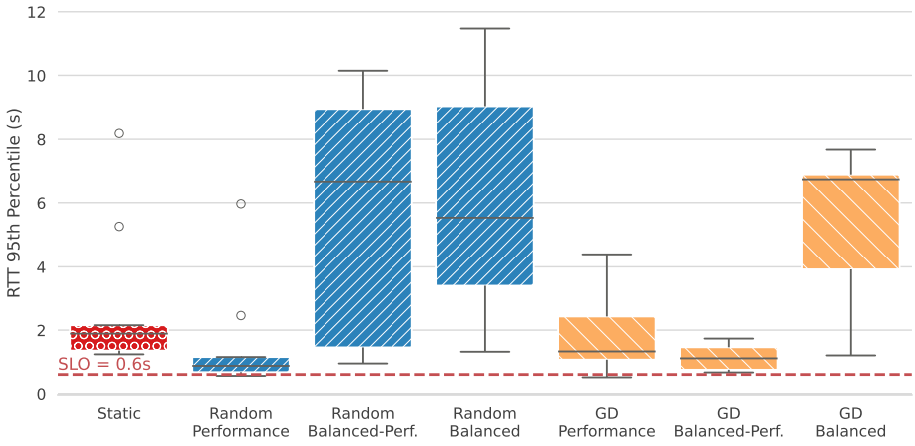



Figure 5.8: RTT P95 - $thr_{RTT} = 0.6s$

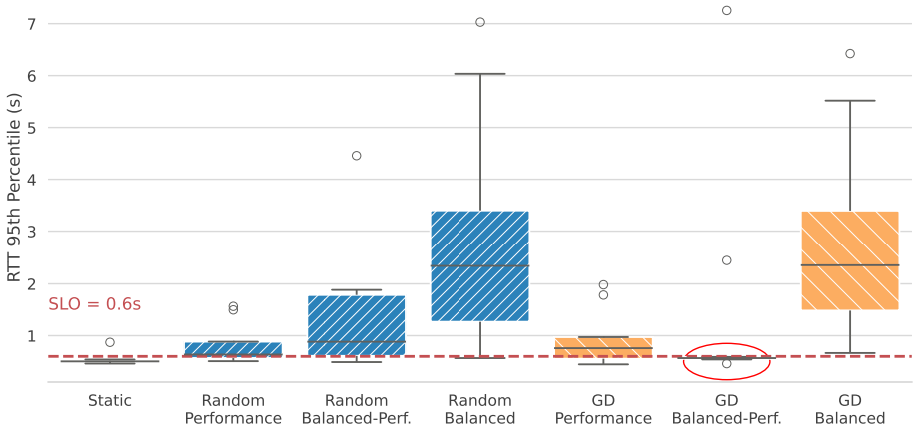



Figure 5.9: RTT P95 - $thr_{RTT} = 0.3s$

While *SimuScale* can handle lower thresholds by having a reduced resource and maintaining a similar performance, it also can adapt to threshold settings that are too high. We set the parameter $thr_{RTT} = 0.9s$ in this case. The *Static* variant will perform worse with many SLO violations. However, the focus of these results should emphasize the versatility of *SimuScale*. Figure 5.11 shows the SLO violations for these experiments. While *SimuScale's Random* approach and the *Balanced* mode failed to adapt quickly enough, it was able to significantly reduce SLO violations using the *GD* approach with *Performance* and *Balanced-Perf.* modes. Across all scenarios, the *Performance* mode reduced median SLO violations compared to the *Static* approach by around 40%. This indicates that the *Performance* mode was too aggressive for other initial threshold settings but appropriate for too high values. Besides reducing the SLO violations, Figure 5.12 shows the difference of the 95*th* RTT percentile relative to the *Static* experiments. Here

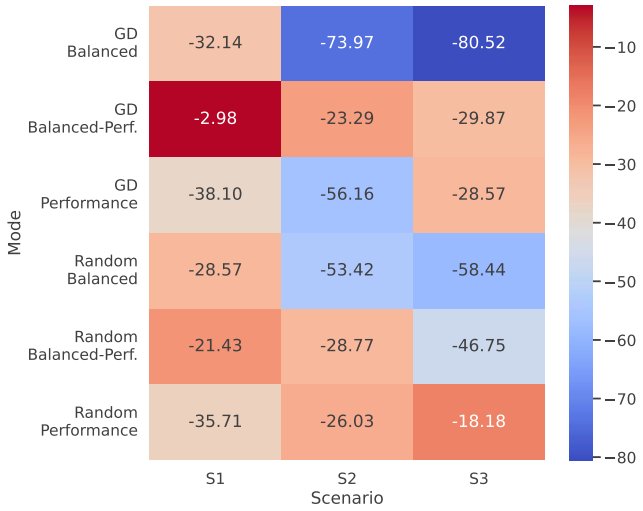



Figure 5.10: Resource Usage Decrease (%) to *Static* ($thr_{RTT} = 0.3s$)

we can see that *Performance* reduced the RTT by up to 84% in the case of *S1*.

Based on these results, we conclude that *SimuScale* can reduce resource usage and SLO violations. While the *GD* approach using *Balanced-Perf.* was able to perform better or similar than the *Static* deployment, the other variants require further fine-tuning. Especially the *Balanced* setting focused too much on resource usage, negatively impacting SLO violations and RTT. Also, the *Performance* mode behaved unexpectedly in some cases and was only able to outperform *Balanced-Perf.* when setting $thr_{RTT}$ to 0.9s. Moreover, while the *Random* approach yielded good results, it overall had a higher variance than *GD*. This was especially visible in the last set of experiments, where we set the initial threshold to 0.9s.

### 5.1.6 Related Work

#### 5.1.6.1 Orchestration Strategy Parameters

Haidari et al. [AHSS13] investigated the impact of different thresholds for different request patterns. Their results indicate that the optimal threshold varies under different loads to minimize resource usage and response times. This underlines our work as we have shown a similar behavior in Section 5.1.2. Similarly, Taherizadeh and Grobelnik [TG20] explore different settings for the HPA that they conduct under different request patterns. They investigate how the reconciliation interval impacts autoscaling decisions, resource usage, and performance. Interestingly, they also looked into changing the policy when removing application instances by only removing one for a specific set of request patterns. Our work can complement this finding by extending the optimization process to tweak parameters and trying different policies that change the overall behavior. Rossi et al. [RCPN23] use Reinforcement Learning agents to update thresholds for autoscaling

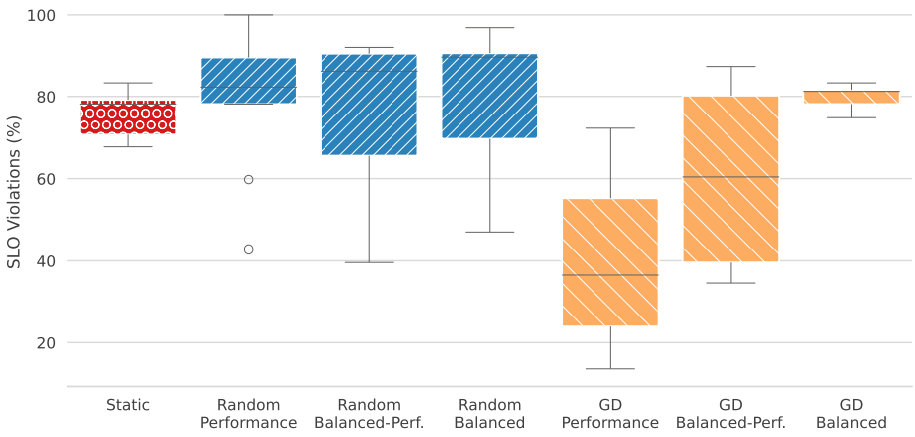



Figure 5.11: SLO Violations - $thr_{RTT} = 0.9s$

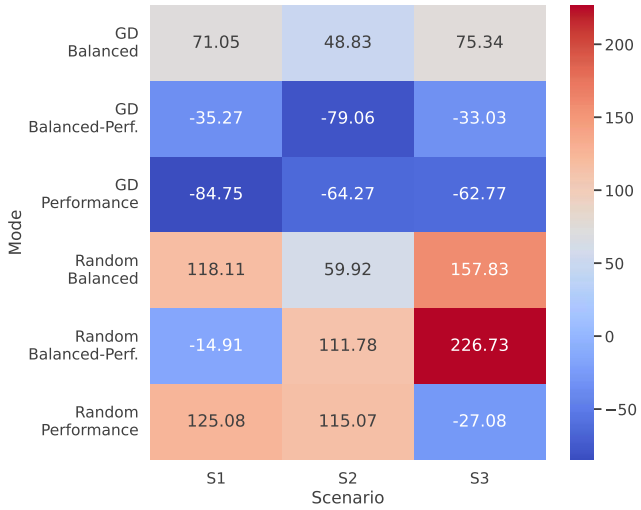



Figure 5.12: RTT P95 Difference (%) to *Static* ($thr_{RTT} = 0.9s$)

during runtime. The main differences are that their agent learns and interacts with the real world and that we base our scaling decisions on the RTT, whereas they explore a CPU-based autoscaler. Therefore, our simulation-based platform can complement their work by using it to speed up the training of the agents.

#### 5.1.6.2 Simulations for Resource Management

Simulations of real-world systems allow decision makers to assess *what-if* scenarios that can guide the selection process [RKK+19]. In the context of orchestration systems (e.g., microservices, functions, containers, etc.), this can help providers efficiently adapt system components in dynamic environments [TPS+22]. Specifically, the adaptation of autoscaler or scheduler components profits from simulation-based adaptations in multiple ways.

Depending on the domain and use case, simulations unite multiple simulations together in a single cohesive simulation unit that can reproduce the behavior of a real complex system. These simulations are called coupled simulations (co-simulations) [GTB+18] and have been successfully used in improving container scheduling performance [TPS+22, TC22]. Like co-simulations, symbiotic simulations use simulations to direct real-world systems [OMS+18, OOW20]. Onggo et al. [OMS+18] identified three symbiotic systems combining real systems and simulations, and differ in the interactions between the two. For example, in the *Decision Support Symbiotic Simulation System*, decision makers (e.g., humans) use the simulation's output to direct the real system. In contrast, in the *Control Symbiotic Simulation System*, simulation directly influences the physical system.

In our work, we implement a PoC that integrates into Kubernetes and mixes the *Control Symbiotic Simulation System* approach and co-simulations to solve *OPOP*. The use of simulations for resource management during runtime has been explored in some other works. Frey et al. [FFH13] use a simulation-based Genetic Algorithm to find optimal architecture configurations for cloud systems. This work complements our work, as we focus on the platform's parameters while they are searching for an optimal architecture. Simulations can be used similarly in that they mimic the real world, and their output is used to influence the orchestration components [TPS+22, TC22, OOW20]. Tuli et al. [TCJ23] propose CILP, which uses the simulation as a teacher to generate the ground truth with which their AI models learn. This work showcases that simulations can be used to optimize orchestration strategies (i.e., AI-based). While they use it as a surrogate to generate data, our simulation interacts directly with the real-world system by sending autoscaler parameters. *SimuScale* differs from others due to its runtime being able to adapt the autoscaler's parameters in response to request pattern changes and has proven to trade off resource usage and performance based on high-level parameters. Rausch et al. [RRD21] investigate how to use simulations for tweaking scheduler parameters, but in contrast to our system, they do not evaluate in a real-world setting. However, it showcases that simulations are feasible for resource management and, specifically, tweaking parameters.

### 5.1.7 Summary of Runtime Tuning

Serverless Edge Computing presents a promising platform paradigm for the heterogeneous edge-cloud continuum by autonomously managing functions under the SLOs' guarantee. Many research works explore different strategies to autoscale, schedule, or route requests, but in production systems, the Orchestration Strategy Drift arises, which renders initial orchestration parameters unsuitable, causing SLO violations.

Based on that, we propose *SimuScale*, a self-adaptive Simulation-based Scaling approach for Serverless Edge Computing. It uses a co-simulation to guide the optimization algorithm in finding optimal autoscaler parameters. Results show that shorter simulation time horizons favor bursty workloads, while longer horizons can perform better in scenarios with less variation. Our evaluation on a real-world testbed has shown that *SimuScale* can improve the results overall by reducing SLO violations and variance across different scenarios. Especially in instances where threshold parameters are set lower and higher than the SLO, it can reduce resource usage by up to 29.87% while maintaining the same RTT and SLO violations by up to 40% and the RTT by up to 84.75%, respectively. However, the results also show that more investigation is necessary to determine appropriate settings for the optimization process. For example, some optimization goals did not yield the expected results and, therefore, require a detailed look at defining these goals.

Future work aims at extending *SimuScale* to support other autoscaling strategies or update parameters of schedulers. An interesting avenue to make our work usable for AI-based models is using simulation-generated data for training [TPS+22]. This can be useful to extend datasets and enhance the rate at which AI models can be updated in new environments. Finally, we want to investigate the uncertainties that are inherently present in such environments [NCTD15] and how they can be integrated into the co-simulations to achieve more accurate predictions.

## 5.2 Simulation-driven Dataset Generation for Bootstrapping Autoscaling Prediction Models

Serverless edge computing promises autonomous function management across the heterogeneous edge-cloud continuum. Specifically, autoscaling of functions increases resource efficiency by creating and destroying instances on demand. However, reactive function instance creation can cause high end-to-end latency due to requests waiting for processing. Researchers and practitioners use proactive autoscaling mechanisms to overcome this by predicting the required number of function instances in advance. Proactive approaches pose challenges because prediction models must be fine-tuned for new environments and updated over time. Therefore, we propose a simulation-driven framework, EdgeCloudForge, that generates datasets to train prediction models for autoscaling in edge-cloud environments ahead of their deployment, thus preparing them in advance for new environments. We introduce the concept of Edge-Cloud Domain Space to describe different infrastructure attributes. EdgeCloudForge uses a two-step method based on initial sampling and Bayesian optimization to navigate this domain space. Simulation results show that our approach can reduce SLO violations by up to 79% across different request patterns compared to the reactive approach and is able to outperform approaches from related work by up to 91%. EdgeCloudForge also shows the capability of training models that can be used across different infrastructures.

### 5.2.1 Introduction

Serverless Edge Computing allows customers to upload functions and let the platforms autonomously orchestrate them. Autoscaling is an orchestration strategy that decides when to create or remove function instances in response to workload changes at runtime. However, they need to consider the cold startup time of function instances [JHA+25, LSd+24]. The cold startup time includes downloading missing dependencies (e.g., container images) and the starting time. Based on environmental factors (e.g., network conditions), this can take several seconds during which no requests can be handled [RRD21]. Reactive approaches, typical for public or open source serverless offerings [RND23], act too late and cause Service Level Objective (SLO) violations, especially when handling sudden bursts of incoming requests, common for serverless workloads [JHA+25].

Therefore, proactive approaches address this challenge by estimating the number of instances before they are needed, based on the workload [AIEB19]. However, these models must be trained for their environment. Finding appropriate parameters (i.e., training datasets) for autoscalers presents a challenge prevalent across autoscaler implementations [SGvK+22]. A common approach in commercial platforms is to rely on historical data between one day and 14 days of historical application data [Goo25, Ama25, Mic25]. Besides, deploying a model in a new environment can be challenging for proactive approaches, as their models are trained in other environments and need time to adapt. However, this means that subpar autoscaling and a slow adoption rate for changing workloads are expected at that time. Moreover, Serverless Edge Computing platforms are

heterogeneous, and the infrastructure changes over time through hardware failures or new hardware updates [RND23, ATC+21]. It is challenging to train a model for autoscaling in minimal time and make it robust for Serverless Edge Computing.

To this end, we propose EdgeCloudForge, which offers synthetic dataset generation to train the model ahead of time and mitigate this issue. Similar approaches have been applied to the cloud domain [AIEB19], but lack the inclusion of heterogeneous systems, such as edge-cloud clusters. We present a novel model to capture heterogeneous aspects of edge-cloud infrastructures to generate plausible simulation scenarios. The model is called Edge-Cloud Domain Space (ECDS) and consists of multiple dimensions: cluster, device, network, and workload. These dimensions describe the space in which a serverless edge platform should operate. A guided process is necessary to navigate this space to generate synthetic datasets. Our framework implements this in a two-step approach that samples random scenarios and then uses Bayesian optimization to explore the ECDS. The framework is co-simulation-driven by mimicking the real-world system to speed up the training phase of autoscaling models through intelligent synthetic dataset generation.

Our main contributions are as follows:

1. We show that historical data from cloud and edge-cloud platforms have different impacts on a prediction model's ability to estimate the number of required function instances. While the predictive models trained on cloud-based traces yield low SLO violations, edge-cloud patterns vary much more across a month, yielding higher SLO violations than the reactive approach.

2. We formulate a novel model to capture the heterogeneous edge-cloud landscape. Edge-Cloud Domain Space (ECDS) defines domain spaces in which serverless edge platforms can be deployed.

3. We implement our framework, EdgeCloudForge[8], that generates synthetic datasets using co-simulation and Bayesian optimization to explore the Edge-Cloud Domain Space. Simulation-based results show that we can decrease the average SLO violations across multiple request patterns of two Deep Neural Network inference functions by 79% to the reactive policy. Moreover, we compare our approach to related work [AIEB19, AIM+21] and see that *EdgeCloudForge* trains more robust models regarding varying workload conditions. The framework can also synthesize datasets when facing heterogeneous infrastructure settings, reducing the burden of re-training the model for each infrastructure.

The remainder of the chapter is structured as follows. Section 5.2.2 shows simulation results highlighting models underperforming with historical data across cloud and edge-cloud workload patterns. Section 5.2.3 introduces our framework and the implementation, Section 5.2.4 presents the evaluation results. Section 5.2.5 presents related work on

[8] https://github.com/edgerun/edgecloudforge

synthetic dataset generation and co-simulation frameworks for platforms, and Section 5.2.6 concludes the chapter.

### 5.2.2 Motivation

We demonstrate that limited access to historical data leads to subpar autoscaling decisions, which motivates our work to synthesize data to improve the model's performance. We show this by training multiple models with varying data periods, from one day to three weeks. The model predicts the number of needed function instances to handle the incoming workload. Our framework, EdgeCloudForge, can significantly speed up the training process through co-simulation and an optimization approach to achieve the same or reduced SLO violations with only one day of historical data, instead of waiting for up to 21 days.

All experiments use the HeROsim simulator [LdB+24]. We focus on simulating Deep Neural Network (DNN) inference functions that the HeROsim provides out of the box, and extract request patterns from the real-world serverless platform datasets from Huawei [JHA+25] and Globus Compute [BPC+24].

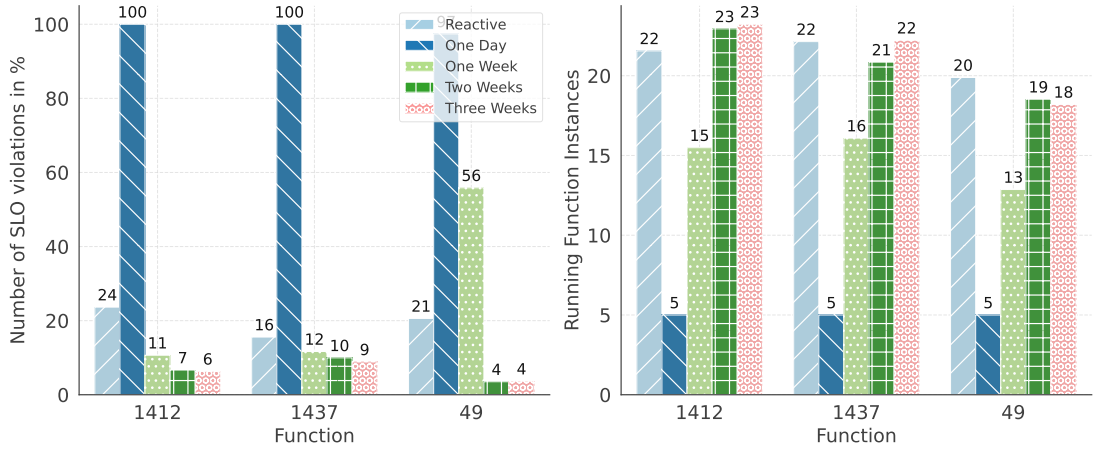
#### 5.2.2.1 Request Pattern Selection

In the following, we describe our method of selecting request patterns from a dataset with different similarities across weeks. We use the Huawei Serverless Cloud dataset [JHA+25], which contains the requests per minute across 31 days over multiple functions and is split into five regions. We only include functions that were called over 90% of the time, have an average above ten requests per minute, and were invoked in Region 1. This is because the dataset contains many functions that were constantly called less than ten times per minute. While resources must be allocated for those, they require fewer scaling decisions than the ones we choose. We select three functions that exhibit high, moderate, and low similarity across weeks to show that 14 to 21 days of historical data can be enough for some functions but not all of them, and that it is essential to consider all cases.

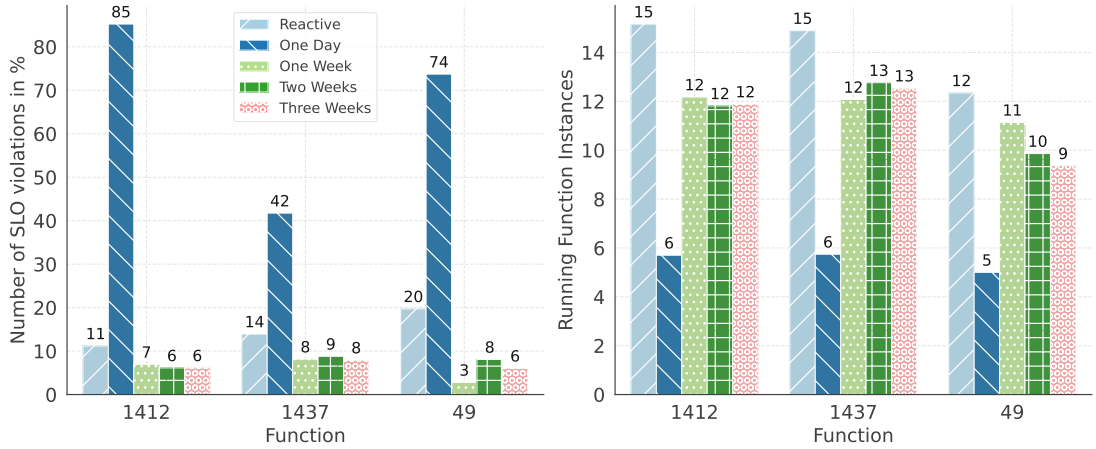
Therefore, we first normalize the request patterns using z-score normalization and then use Dynamic Time Warping (DTW) to calculate the pair-wise distance of all weeks. Then, we calculate the difference between the minimum and maximum distances between weeks to identify very similar or dissimilar functions. We choose three functions (49, 1412, 1437) with the lowest and highest difference, and also take one based on the median distance. We do this to evaluate if models struggle with workloads, observing significant differences across weeks. We also use the Globus Compute dataset [BPC+24], which contains traces from an edge-cloud serverless platform. However, none of the functions were continuously called for at least a month, which led us to extract the request patterns of individual endpoints. Endpoints act as a gateway and comprise invocations of multiple functions.

In the following, we first evaluate prediction models trained on the cloud request patterns and then assess them on the request patterns from the edge-cloud platform.

#### 5.2.2.2 Cloud Request Pattern Experiments




(a) *dnn2* SLO violations and average running function instances.




(b) *dnn1* SLO violations and average running function instances.

Figure 5.13: Comparing reactive and proactive models trained on a cloud dataset with different numbers of days, using the fourth week as a validation request pattern. Results show that larger historical data can reduce SLO violations, while datasets with too few samples (i.e., a day) can have a negative impact.

We simulate these patterns using a reactive autoscaling approach based on Knative's autoscaling policy that estimates the number of instances needed to reach certain requests per instance. The reactive policy allows us to collect historical data, which we use to train four models with different amounts of historical data, ranging from one day to

three weeks. Because we train a model with up to three weeks of data, we always use the last week of each workload pattern as the validation workload. The models receive the average request per second of the next minute as input. An assumption that we make in this chapter is that we have an accurate model to predict the incoming workload. However, in our evaluation, all proactive approaches have access to this accurate model, focusing on the ability of these models to estimate the number of function instances.

The scenarios simulate two serverless functions that execute low-latency Deep Neural Network inference and are included in HeROSim [LSd+24]. Figure 5.13a shows the SLO violations and average number of running function instances of function *dnn2* when using the request patterns from the Huawei serverless dataset [JHA+25] of the functions with IDs 49, 1412, 1437, Figure 5.13b shows the same metrics but simulates the function *dnn1*. The SLO violations are based on the requests that did not fulfill the latency SLO set for each function.

Figure 5.13a shows that the models can autoscale properly and reduce the number of SLO violations compared to the reactive approach, while maintaining the same number of running instances.

Especially the models trained using two and three weeks of historical data have maintained low SLO violations. Figure 5.13b shows the results of the experiments that simulate the function *dnn1* which has a shorter execution duration in comparison to *dnn2*, thus, less number of instances are required. We see a similar trend: proactive models can reduce SLO violations. The results show that having one day of historical data is insufficient, and at least 1 week is needed to train a model that can reduce SLO violations. We conclude that cloud-based request patterns seem consistent over a month to train a good-performing model quickly.

#### 5.2.2.3 Edge-Cloud Request Experiments

Figure 5.14 shows the results of the motivational experiments based on the request patterns extracted from the Globus Compute dataset [BPC+24]. The patterns originate from a single endpoint of the months February (*02*) and March (*03*). While the cloud-based historical data has proven to yield low SLO violations, the edge-cloud patterns vary much more across the month. For example, using the model trained on three weeks' worth of data achieves a similar number of SLO violations as the reactive approach. The models with even less historical data achieve worse SLO violations than the reactive approach.

These results showcase that while generally using one week or more can be enough to train the model in case of the cloud patterns, it is not enough for the patterns from the edge-cloud platform. To overcome this challenge of (1) preparing a model with minimal historical data and (2) making it scale appropriately for all functions, we introduce EdgeCloudForge to generate synthetic datasets.

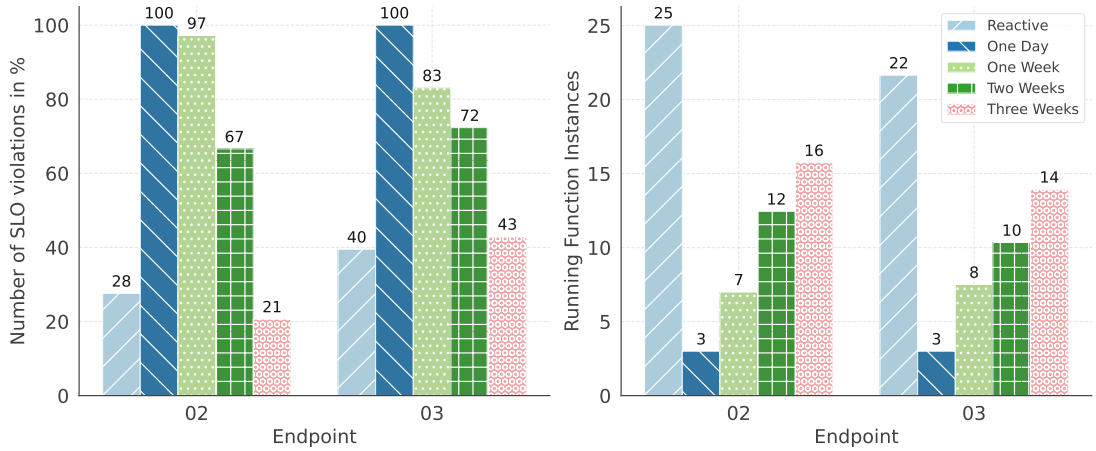



Figure 5.14: *dnn2* SLO violations and average running function instances when training on edge-cloud request patterns. The prediction models were not able to outperform the reactive approach in terms of SLO violations.

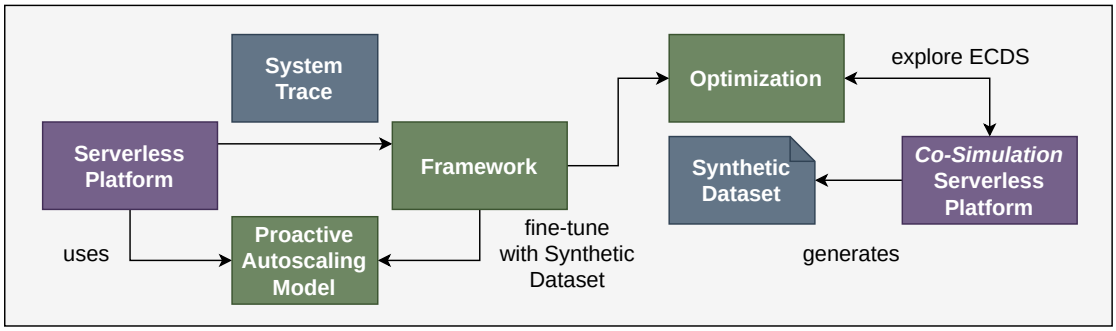



Figure 5.15: EdgeCloudForge Framework

### 5.2.3 Simulation-driven Synthetic Dataset Generation Framework

#### 5.2.3.1 Overview

Figure 5.15 shows the proposed EdgeCloudForge framework, its interaction with the real-world serverless platform, and its internal components. The serverless platform uses a proactive autoscaling model that relies on proper training. The framework is triggered by receiving a trace from the serverless platform. A trace contains recent monitoring data, including the execution times of functions, container image sizes, request patterns of functions, and the cluster configuration, such as devices and network conditions. Our implementation does not entail a real-world integration and is solely based on the HeROsim simulator [LdB+24], which provides function-related traces out of the box. EdgeCloudForge uses an optimization module that executes different simulation scenarios to generate synthetic datasets. These datasets are used to fine-tune the proactive autoscaling model to increase robustness over future request patterns and decrease SLO

violations. Therefore, we now introduce the Edge-Cloud Domain Space, which the framework uses to generate simulation scenarios.

#### 5.2.3.2 Edge-Cloud Domain Space

The Edge-Cloud Domain Space (ECDS) defines the conditions in which an edge-cloud platform is meant to operate. The conditions are imposed by the infrastructure and deployed functions, and include aspects around the infrastructure's compute capabilities (i.e., available devices), network conditions (i.e., network bandwidth), and function-related characteristics such as the request pattern (i.e., average request per second). It is loosely based on the Operation Design Domain (ODD) paradigm, which mainly revolves around robotic applications and their operating space [GLCCT22]. For example, these operating spaces specify the road type for which autonomous driving is deemed to be safe.

ECDS describes the space in which a serverless edge-cloud platform should optimally manage functions, whereas the users must specify the definition of optimal. In this work, we focus on reducing SLO violations of functions and resource usage. We define ECDS as a tuple of multiple dimensions that span the space:

$$S_{ECDS} = \langle \{W_i\}_{i=1}^{m}, \{N_j\}_{j=1}^{p}, C, B \rangle$$

Where $W$ represents the intensity dimension of the functions' workload, $N$ is the configuration dimension of the node types, $B$ is the dimension of the network configuration, and $C$ is the dimension of the cluster configuration. Specifically, $W_i$ describes the range of requests per second a function $i$ can be invoked, $N_p$ the proportion of device type $p$ in the cluster, $B$ the network bandwidth, and $C$ the size of the cluster. Moreover, $S_{ECDS}$ includes a dimension $W_i$ for each function $i$ and a dimension $N_p$ for each type of device $p$. Each dimension consists of four parameters: the average expected value, the boundaries (minimum and maximum), and the step size used to generate samples. This space allows us to define the environment in which the platform will operate. For example, we can describe the function $i$ to have an average $W_i^{avg}$ of 150 RPS, a minimum $W_i^{min}$ of 25 RPS, and a maximum $W_i^{max}$ of 300 RPS. We use the step size $W_i^{step}$ to enumerate all possible values for a dimension and create plausible scenarios. $S_{ECDS}$ describes the possible values of each dimension, and we formulate $\vec{s}$ as an instance of $S_{ECDS}$. ECDS allows us to generate plausible scenarios and lets us navigate the space to generate synthetic datasets.

#### 5.2.3.3 Framework

The framework consists of the initial and optimization stages. The initial stage converts a request trace $T$ of the real-world serverless platform into an Edge-Cloud Domain Space $S_{ECDS}$. That means $T$ contains historical data of the request patterns for each deployed function to create workload intensity dimensions $W$, cluster information to create $C$, $B$, and $N$. The framework can take any historical data as input, but we always take one day in this work. We generate initial simulation samples $\vec{s}$ before exploring the space using

Bayesian optimization for each function $f$. These simulation scenarios are created by randomly sampling from $S_{ECDS}$ using Latin Hypercube Sampling. For example, a sample $\vec{s}$ can contain a cluster size of 10 devices, which consists of 40% of device type A, 60% of device type B, a workload intensity of 150 RPS, and a network bandwidth of 100 Mbit/s. The framework converts $\vec{s}$ into an input for the simulation (in our case, HeROSim) and executes it. In the current implementation, the workload uses Poisson-based arrival times with an average request rate from the sample and runs for at least 350 seconds with a slight ramp-up. We chose the Poisson-based arrival times to synthesize data (i.e., prediction samples) that map the requests per second to the number of required function instances satisfying the SLO. Thus, we incrementally learn the distribution of required function instances over various requests per second. We use the reactive policy to execute these scenarios, where each result is prepared in a data set $\mathcal{D}$ that we collect as an initial dataset.

**Algorithm 2** Synthetic Dataset Generation

**Input:**
- Initial datasets $\mathcal{D} = \{X, y\}$ of function $f$
- Param bounds $\mathcal{B}$ based on $S_{ECDS}$
- Resource weight $\omega$, iterations $N$, workers $P$
- Saturation threshold $\tau = 0.95$

Initialize $\mathcal{BO}$ with $\mathcal{B}$, $\mathcal{X}_{\text{sat}} \leftarrow \emptyset$
**for** $i \leftarrow 1$ to $N$ **do**
  Generate $P$ candidates $\mathbf{x}_j \sim \mathcal{BO}$
  PruneCandidates($\mathbf{x}_j$)
  // Parallel evaluate:
  **for** each $\mathbf{x}'_j$ in parallel **do** // $\mathbf{x}'_j$ remaining candidates
    $(\phi_j, \phi^j_p, \phi^j_{res}, \text{sat}_j, \mathbf{X}^j_{\text{new}}, \mathbf{y}^j_{\text{new}}) \leftarrow$ EvaluateParameters($\mathbf{x}'_j, \mathcal{D}, \omega, \tau$)
  **end for**
  Update $\mathcal{X}_{\text{sat}}$ with $\{\mathbf{x}'_j | \text{sat}_j = \text{True}\}$
  Update $\mathcal{BO}$ with $\{\phi_j\}$
  **for** each function $f$ **do**
    $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{X}^j_{\text{new}}, \mathbf{y}^j_{\text{new}})$
  **end for**
**end for**
**return** $\mathcal{D}$

Next, the optimization-driven synthetic dataset generation begins. Algorithm 2 outlines the steps of this stage. It takes the initial datasets for a function $f$, $D$, the parameter bounds defined through $S_{ECDS}$, the resource weight $\omega$, the number of iterations $N$ and the number workers $P$ as well as a saturation threshold $\tau$, which helps pruning candidates that overload the infrastructure with requests. $\mathcal{B}$ is the parameter space used to generate candidates for exploration, and we use Latin Hypercube Sampling again in our Bayesian

**Algorithm 3** EvaluateParameters

**Input:** $\mathbf{x}$, $\mathcal{D}$, $\omega$, $\tau$
**Output:** $\phi_{\text{total}}$, $\phi_p$, $\phi_{res}$, saturated, $\mathbf{X}$, $\mathbf{y}$
$\mathcal{S}_r \leftarrow \text{ReactiveSim}(\mathbf{x})$
Extract $\mathbf{X}_{\text{new}}, \mathbf{y}_{\text{new}} \leftarrow \mathcal{S}_r$
$\mathcal{M} \leftarrow \text{Train}(\mathcal{D} \cup (\mathbf{X}_{\text{new}}, \mathbf{y}_{\text{new}}))$
$\mathcal{S}_p \leftarrow \text{ProactiveSim}(\mathbf{x}, \{\mathcal{M}\})$
Calculate penalties:

$$\phi_p \leftarrow \text{SLO violations in } \mathcal{S}_p$$

$$\phi_{res} \leftarrow \sqrt{\underbrace{|\mathcal{S}_r^{\text{res}} - \mathcal{S}_p^{\text{res}}|}_{\Delta_{\text{abs}}} \cdot \underbrace{\frac{\Delta_{\text{abs}}}{\mathcal{S}_r^{\text{res}}}}_{\Delta_{\text{rel}}}}$$

$$\phi_{\text{total}} \leftarrow \phi_p + \omega\phi_{res}$$

Check saturation:
saturated $\leftarrow (\mathcal{S}_r^{\text{util}} > \tau)$
**return** $\phi_{\text{total}}$, $\phi_p$, $\phi_{res}$, saturated, $\mathbf{X}_{\text{new}}, \mathbf{y}_{\text{new}}$

**Algorithm 4** Saturation-Aware Pruning

**Maintain:** Saturated configurations set $\mathcal{X}_{\text{sat}}$
Define similarity threshold $\tau_{\text{sim}} = 0.95$
**procedure** PruneCandidates($\mathbf{x}_j$)
    **for** each $\mathbf{x}_{\text{sat}} \in \mathcal{X}_{\text{sat}}$ **do**
        $\phi_{\text{rat}} \leftarrow \frac{1}{n}\sum_{k=1}^{n} \frac{|\mathbf{x}_j^k - \mathbf{x}_{\text{sat}}^k|}{|\mathbf{x}_{\text{sat}}^k|}$
        **if** $\phi_{\text{rat}} > \tau_{sim}$ **then**
            Reject $\mathbf{x}_j$ with penalty $\phi \leftarrow -\infty$
        **end if**
    **end for**
**end procedure**

Optimization ($\mathcal{BO}$). $\omega$ is a weight used when calculating the objective value - we not only consider SLO violations when exploring the space, but also want to reduce resource usage. We explain the details later. The optimization is looped $N$ times, and $P$ candidates are evaluated in parallel.

In each iteration, we sample candidates $x_j$ using $\mathcal{BO}$ and then prune candidates having parameters that overload the infrastructure, thus, are not worth investigating. Algorithm 4 shows this procedure that compares the values between the given parameters and the recorded saturated parameter values. We calculate the ratio between them and use the average similarity over all parameters to decide whether to reject them. If the two sets are on average too similar (i.e., larger than $\tau_{sim}$), the optimization rejects this candidate immediately. The intuition behind this pruning is that we can sample workload intensities from $\mathcal{B}$ that the given infrastructure cannot adequately process (i.e., more functions would have been needed, but all resources were exhausted). Consequently, these candidates yield high SLO violations, thus getting more explored by the optimizer, but do not generate meaningful data. Future work could exploit these results and report them back to users, providing them with information on which scenarios were infeasible. Then, we evaluate the candidates as shown in Algorithm 3 in parallel, which entails the execution of the simulation using a reactive policy, training a model by combining $\mathcal{D}$ with the new features from the reactive simulation. Afterward, we execute the same scenario with the proactive policy and calculate penalties, which act as objective values for the optimization. $\phi_p$ contains the number of SLO violations, $\phi_{res}$ represents the resource usage penalty based on the ratio between used resources during the reactive and proactive approach. Specifically, $\phi_{res}$ combines absolute and relative resource efficiency using a geometric mean approach. No penalty is applied when the proactive approach yields better resource usage (i.e., fewer function instances). However, if the proactive approach uses more resources than the reactive one, we apply a penalty proportional to the available resources. $\phi_{total}$ combines the two penalties by weighting $\phi_{res}$ with $\omega$, allowing us to vary the importance of resource efficiency. Maximizing the penalty allows us to explore the space of parameters the model is not yet adapted to and collect all new synthetic datasets. We mark the parameters as saturated if the utilization exceeds the given threshold. The utilization is based on the amount of resources used. After evaluating all candidates, we update $\mathcal{X}_{sat}$ with all marked saturated results. Then we update the optimizer and combine the previous datasets with the new ones. The optimization is stopped once the number of iterations has been reached. The output of *EdgeCloudForge* is a synthesized dataset that can be used to train predictive models for one function $f$.

### 5.2.4 Evaluation

We first outline details of our setup, including the implementation of the reactive and proactive autoscaling models, and then present two evaluation scenarios.

#### 5.2.4.1 Experiment Setup

**Implementation** The proactive autoscaling model uses the predicted average workload for the next minute and outputs the number of function instances. This work assumes that an accurate workload forecasting model is available, such as presented for serverless workloads by Joosen et al. [JHA+23]. The accurate workload forecasting model is used for all proactive approaches, including the ones we compare ours with, which emphasizes the resulting performance of the function instance prediction under the same circumstances. Our evaluation uses a polynomial regression model ($\delta = 3$) using a Standard feature scaler and the Stochastic Gradient Descent algorithm, showing promising results in preliminary experiments. The training data is created by calculating the average of completed requests in five-second time windows. Furthermore, the training data is based on the behavior of a reactive autoscaling policy. In our case, the standard Knative Autoscaling policy tries to create as many replicas as needed to reach a particular target concurrency level per function instance. The concurrency level represents the number of requests in the queue of a function instance. We have implemented all components in Python and used the HeROSim simulator [LdB+24], which includes function traces for the evaluation. We simulate the functions *dnn1* and *dnn2* from HeROSim. Both stem from an intrusion detection system application and have been profiled on edge devices, such as a Raspberry Pi, an Nvidia Jetson board, and an edge FPGA device [LSd+24]. Moreover, in these experiments, we set $N$, the number of optimization iterations, to 10 and $P$, the candidates per iteration, to 5, and the number of initial samples to 5. For each scenario, we generate new synthetic datasets based on the first day of the chosen request pattern.

**Comparison** The evaluation focuses on comparing our framework to the reactive approach (*Reactive*), the predictive model trained on three weeks of historical data (*Pred-3W*), and the approaches presented in [AIEB19] (*SLOPred*) and [AIB+20] (BAAS). The *SLOPred* approach revolves around using simulation to generate synthetic datasets and use them to train a prediction model for autoscaling. The prediction model uses the response time SLO of an application and the request per second prediction as inputs. In contrast to our approach, a single model is trained for all applications. They linearly increase the load of an application in a step-wise fashion in the simulation to generate synthetic datasets. BAAS presents a burst-aware autoscaling algorithm that uses a container prediction model, trained on a similar synthetic dataset as presented in [AIEB19]. We implement this approach by first creating a simple queue-based simulation to determine the response times of a single function instance with increasing workload. Then we can apply the trace-driven simulation approach presented in [AIEB19] to test the number of containers needed to satisfy a given number of requests per second. This generates a dataset which maps a number of requests and the response time SLO to the number of function instances, which is used to train a decision tree regressor, as presented in [AIEB19, AIB+20]. Due to our search-based approach and using a full trace-driven simulation, our approach takes at least one hour in to generate the synthetic dataset. Therefore, the approaches of [AIEB19] and [AIB+20] are much faster. However,

*EdgeCloudForge's* goal is to generate the datasets in advance, before deploying functions, and, thus, is not optimized for fast synthetic dataset generation. This approach contrasts with others, such as SimuScale [RND24], that aim to use simulations during runtime. Our approach, *EdgeCloudForge*, uses our per-function function instance prediction model and is trained on our ECDS-based datasets. Table 5.2 presents a summary of all approaches.

Table 5.2: Summary of Autoscaling Approaches

| **Approach** | **Description** |
|---|---|
| Reactive | Reactive approach scaling based on current system state |
| *Pred-3W* | Predictive model trained on three weeks of historical data |
| *SLOPred* | SLO-based prediction model from [AIEB19] using step-wise load simulation for training data [AIEB19] |
| *BAAS* | Original burst-aware algorithm [AIB+20] trained on step-wise simulation data [AIEB19] |
| *EdgeCloudForge* | Our function instance prediction model trained on our ECDS-based datasets |

We compare the SLO violations, i.e., when a request's response time is longer than 15x the raw execution time, and the number of running function instances, which should be minimized. In addition, other performance metrics are used, such as the queuing or execution times in seconds or energy consumption in kilowatt-hour (kWh).

**Scenarios** Two types of experiments are conducted, the first one focusing on the ability of our framework to generate synthetic datasets for the two functions (i.e., *dnn1* and *dnn2*) using two distinct sets of request patterns. The first set contains three function request patterns (i.e., 49, 1412 & 1437) from the Huawei serverless dataset [JHA+25], representing a serverless cloud platform. The second one is on the Globus Compute Dataset, representing a serverless edge platform with request patterns. However, we extract an endpoint's request pattern instead of using a function's requests. Endpoint request patterns comprise the invocations of multiple functions and serve as a gateway. This is because none of the functions were called consistently for a month, a prerequisite for our scenario. The endpoint's UUID is *fc1d7a40-6293-758c-2158-b7986a2089d6*, and we extract the request pattern of February (02) and March (03) 2023. In all cases, we evaluate the last week of the month and shorten it to 20 minutes while keeping the patterns intact. Section 5.2.4.2 shows the results of the Huawei Cloud dataset and Section 5.2.4.3 the results of the Globus Edge-Cloud dataset. We evaluate in Section 5.2.4.4 the scenario in which *EdgeCloudForge* generates synthetic datasets based on an $S_{ECDS}$ comprising various network and cluster configurations, and then is evaluated on three different infrastructures to test the robustness of the trained model when facing unseen infrastructures. This scenario investigates the impact of varying cluster configurations in terms of device

| | $W_{49}$ | $W_{1412}$ | $W_{1437}$ | $W_{02}$ | $C$ | $B$ |
|---|---|---|---|---|---|---|
| *avg* | 134 | 167 | 157 | 225 | 10 | 100 |
| *min* | 20 | 12 | 2 | 3 | 10 | 100 |
| *max* | 625 | 368 | 574 | 630 | 10 | 100 |
| *step* | 0.05 | 0.05 | 0.05 | 0.05 | 0 | 0 |

(a) Parameter Space $S_{ECDS}$ for optimizing provactive model with three request patterns from the Huawei dataset (i.e., $W_{49}$, $W_{1412}$ & $W_{1437}$) and one from the Globus Compute Dataset (i.e., $W_{02}$). Simulations were executed using both functions, *dnn*1 and *dnn*2.

| | $C$ | $B$ | $W_{49}$ | $N_{rpi}$ | $N_{xavier}$ |
|---|---|---|---|---|---|
| *avg* | 10 | 100 | 20 | 0.5 | 0.3 |
| *min* | 5 | 100 | 134 | 0.1 | 0.1 |
| *max* | 30 | 500 | 625 | 0.9 | 0.4 |
| *step* | 5 | 100 | 0.05 | 0.1 | 0.05 |

(b) Parameter Space $S_{ECDS}$ for optimizing provactive model against varying infrastructure settings. Simulations only used function *dnn*2 and were evaluated with patterns $W_{49}$ and $W_{02}$

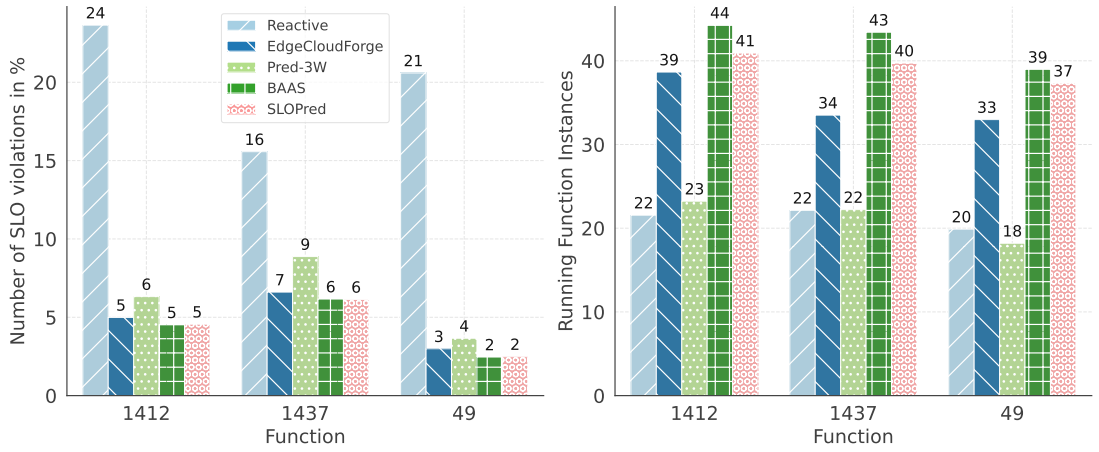
Table 5.3: Parameter spaces for different optimization scenarios

proportions, cluster size, and network bandwidth. Table 5.3 shows the $S_{ECDS}$ for both experiments.

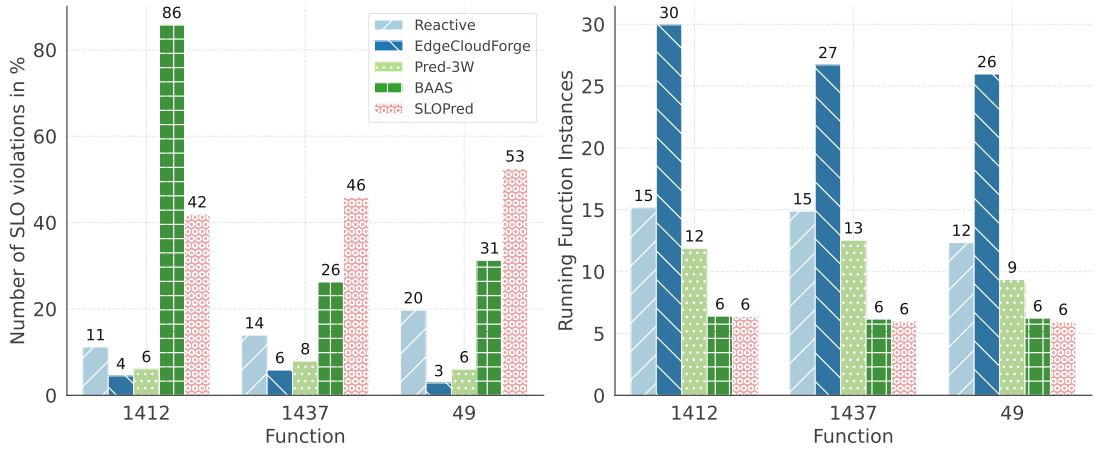
#### 5.2.4.2 Cloud Function Autoscaling

The following compares the *Reactive*, *Pred-3W*, *BAAS*, *SLOPred* and *EdgeCloudForge* approaches when executing *dnn1* & *dnn2* with the request patterns of the functions *49*, *1412* and *1473*. Those are the three functions shown in the Section 5.2.2. These functions exhibit high, medium, and low similarity across the month.

Figure 5.16 shows the SLO violations and average number of running function instances of the simulated functions *dnn*2 and *dnn*1 across the approaches. We see that across all request patterns, *EdgeCloudForge* produced models that reduced SLO violations compared to the reactive approach. Specifically, it reduced the SLO violations by up to 79% in the case of the request pattern *1412* when executing *dnn2*. In comparison to the other proactive approaches, *EdgeCloudForge* yielded similar results. However, in *EdgeCloudForge's* case, the SLO violation reduction is tied to many average running instances. Especially when comparing this to the *Pred-3W* model. We investigated this further and determined that this high resource usage happened in response to drastic workload changes. For example, when a peak was imminent, our model created more instances than *Pred-3W* and then gradually scaled down again. This also explains the differences in SLO violations, as our approach absorbed the bursts of requests. When comparing the number of running instances in times of low load, both approaches have a similar instance count. Figure 5.16b shows that *BAAS* and *SLOPred* were not able




(a) Huawei Request Pattern evaluation results of *dnn2*'s SLO violations and average number of running function instances.




(b) Huawei Request Pattern evaluation results of *dnn1*'s SLO violations and average number of running function instances.

Figure 5.16: Comparing reactive and proactive models trained on a generated synthetic and historical dataset. Results show that *EdgeCloudForge's* models can yield good results across request patterns.

to accurately estimate the number of running instances, which led to a high amount of SLO violations. Contrarily, Figure 5.16a shows that both approaches yielded very low SLO violations but high resource usage. We attribute this circumstance to the approach *BAAS* and *SLOPred* use to generate synthetic datasets. Specifically, *dnn1* has a lower measured execution time; supposedly, fewer instances are needed to process more requests. However, this theoretical approach does not capture all aspects of the system and, thus, underestimates the number of instances. This also shows that a simulation-based approach like *EdgeCloudForge* can circumvent this issue through more

extensive simulations that capture more environmental aspects, such as cold start-up times.

Table 5.4 and Table 5.5 show further performance metrics in detail, ranging from the average queuing time of a request (*QTime*), the average function execution time (*ExecTime*), the average computation time (*CompTime*), and the energy consumed. HeROSim provides all these metrics at the end of the simulation and spans all requests. While the energy is similar across approaches, the QTime can differ. We highlight the lowest QTime for each function. The results show that *BAAS* had the lowest queueing times when simulating *dnn2*, which also reflects the low amount of SLO violations. In the case of *dnn1*, *EdgeCloudForge* yielded the lowest average queuing time.

The results show that our framework can generate datasets that adequately prepare the model. Our model had a relatively high average number of running function instances regarding resource usage for *dnn*1, which we attribute to the reaction to bursts of requests. Moreover, *EdgeCloudForge's* models kept the resource usage lower than *BAAS* and *SLOPred* when simulating *dnn*2. *BAAS* and *SLOPred* were able to yield low SLO violations in case of *dnn2* but had a high number of violations when simulating *dnn1*. We believe this is due to the theoretical model approach of generating synthetic datasets. The *Pred-3W* model performed well across functions and shows that three weeks of historical data can be enough to autoscale functions. Combining long-running historical data and our synthetic datasets sets an avenue for future work.

Table 5.4: Performance metrics for proactive approach for function *dnn2* on Huawei request patterns. *EdgeCloudForge*'s QTime is close to the best one from *BAAS*. While using on average fewer function instances as shown in Figure 5.16.

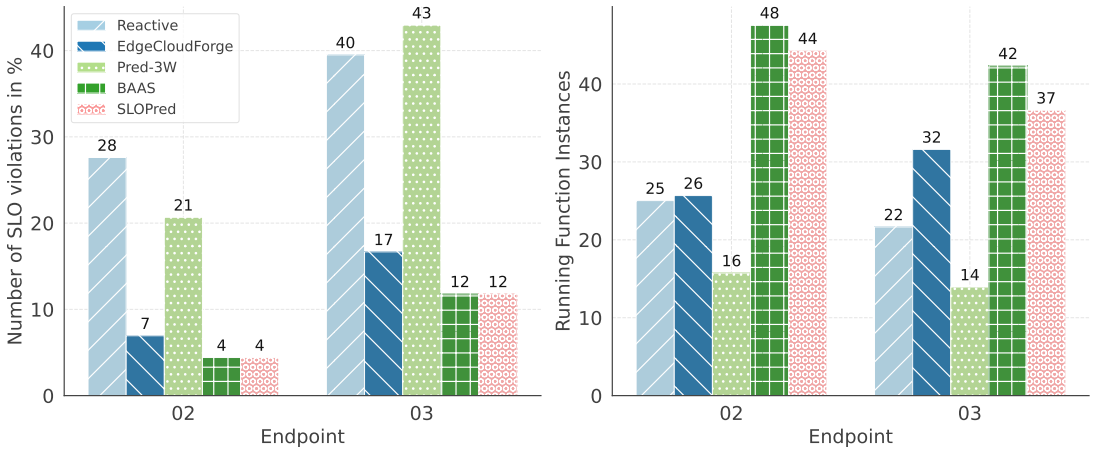
| Func | Proactive | QTime (s) | ExecTime (s) | CompTime (s) | Energy (kWh) |
|---|---|---|---|---|---|
| 1412 | BAAS | **1.112** | 0.040 | 0.061 | 0.036 |
| *1412* | *EdgeCloudForge* | *1.117* | *0.032* | *0.053* | *0.034* |
| 1412 | Pred-3W | 1.234 | 0.025 | 0.046 | 0.035 |
| 1412 | Reactive | 3.304 | 0.035 | 0.056 | 0.036 |
| 1412 | SLOPred | 1.125 | 0.031 | 0.052 | 0.036 |
| 1437 | BAAS | **1.580** | 0.035 | 0.056 | 0.034 |
| *1437* | *EdgeCloudForge* | *1.587* | *0.030* | *0.051* | *0.033* |
| 1437 | Pred-3W | 1.728 | 0.025 | 0.046 | 0.033 |
| 1437 | Reactive | 3.918 | 0.040 | 0.061 | 0.033 |
| 1437 | SLOPred | 1.628 | 0.029 | 0.050 | 0.032 |
| 49 | BAAS | **0.532** | 0.031 | 0.052 | 0.035 |
| *49* | *EdgeCloudForge* | *0.632* | *0.028* | *0.049* | *0.035* |
| 49 | Pred-3W | 0.768 | 0.025 | 0.046 | 0.035 |
| 49 | Reactive | 2.718 | 0.036 | 0.057 | 0.036 |
| 49 | SLOPred | 0.555 | 0.032 | 0.053 | 0.036 |

Table 5.5: Performance metrics for proactive approach for function *dnn1* on Huawei request patterns. *EdgeCloudForge* achieved the shortest QTime for all patterns.

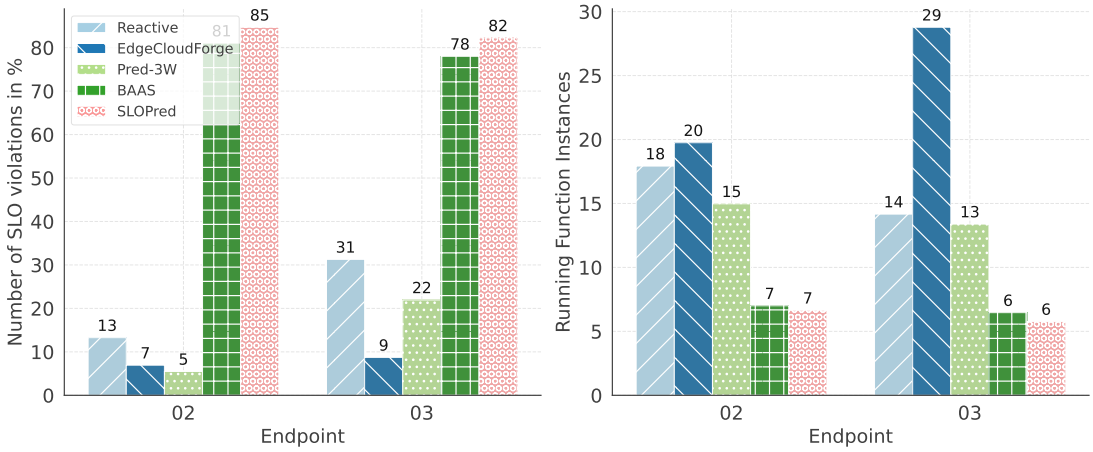
| Func | Proactive | QTime (s) | ExecTime (s) | CompTime (s) | Energy (kWh) |
|---|---|---|---|---|---|
| 1412 | BAAS | 6.833 | 0.005 | 0.026 | 0.023 |
| *1412* | *EdgeCloudForge* | ***0.900*** | *0.002* | *0.024* | *0.021* |
| 1412 | Pred-3W | 1.102 | 0.001 | 0.022 | 0.021 |
| 1412 | Reactive | 2.072 | 0.002 | 0.022 | 0.021 |
| 1412 | SLOPred | 4.096 | 0.003 | 0.024 | 0.022 |
| 1437 | BAAS | 4.115 | 0.001 | 0.022 | 0.020 |
| *1437* | *EdgeCloudForge* | ***1.190*** | *0.006* | *0.026* | *0.022* |
| 1437 | Pred-3W | 1.441 | 0.001 | 0.022 | 0.020 |
| 1437 | Reactive | 2.689 | 0.002 | 0.023 | 0.021 |
| 1437 | SLOPred | 4.938 | 0.002 | 0.023 | 0.021 |
| 49 | BAAS | 3.189 | 0.001 | 0.022 | 0.021 |
| *49* | *EdgeCloudForge* | ***0.571*** | *0.003* | *0.023* | *0.022* |
| 49 | Pred-3W | 0.789 | 0.001 | 0.022 | 0.021 |
| 49 | Reactive | 1.436 | 0.003 | 0.023 | 0.022 |
| 49 | SLOPred | 4.030 | 0.002 | 0.023 | 0.021 |

#### 5.2.4.3 Edge-Cloud Function Autoscaling

The following results focus on the request patterns extracted from the Globus Compute dataset [BPC+24]. Moreover, in these experiments, *EdgeCloudForge* only generated a dataset based on the endpoint *02*, and we used the same model for endpoint *03*. This way, we show that EdgeCloudForge's models can be used across different environments. Figure 5.17 shows the SLO violations and number of running instances across approaches for *dnn2* and *dnn1*. While *Pred-3W* had low SLO violations in the Huawei request patterns, this does not hold for the Globus Compute request patterns. Specifically, when simulating *dnn2*, the model could not properly scale up and had SLO violations of up to 43%. In contrast, *EdgeCloudForge*, *SLOPred*, and *BAAS* were able to scale up accordingly and yield low SLO violations. *EdgeCloudForge* also had low SLO violations and a low number of instances. This further highlights the impact of request patterns, especially since three weeks might not be enough training data. The experiments simulating *dnn1* show similar results as before, with *BAAS* and *SLOPred* performing worse. *EdgeCloudForge* has the lowest SLO violations overall, albeit also having the highest resource usage. Table 5.6 and Table 5.7 show further performance metrics across approaches. While *EdgeCloudForge* did not achieve the lowest queuing time for each scenario, it was able to achieve overall low times.




(a) Globus Compute Endpoint Request Pattern evaluation results of *dnn2*'s SLO violations and average number of running function instances.




(b) Globus Compute Endpoint Request Pattern evaluation results of *dnn1*'s SLO violations and average number of running function instances.

Figure 5.17: Comparing reactive and proactive models trained on a generated synthetic and historical dataset. Results show that the synthetic dataset, based on one day of historical data, yields a more robust model regarding different workload patterns.

#### 5.2.4.4 Varying Cluster Configurations

In the following experiments, we focus on the *dnn2* function and select the request pattern of the Huawei function *49* and the Globus Endpoint pattern of February (*02*). We vary all dimensions of $S_{ECDS}$ during the optimization, and details on the $S_{ECDS}$ are shown in Table 5.3b. We set up three different infrastructure scenarios for the evaluation. The Small one has 10 nodes, with 100 Mbit/s, 20% Raspberry Pis, and 80% Jetson Xavier devices. The Medium scenario consists of 20 nodes, with 250 Mbit/s, and 80%

Table 5.6: Performance metrics for proactive approach for function *dnn2* on Globus Endpoints.

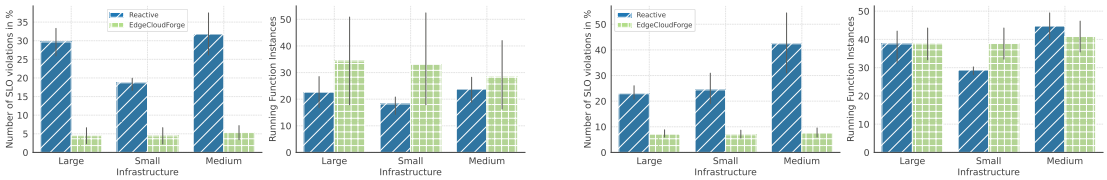
| Endpoint | Proactive | QTime (s) | ExecTime (s) | CompTime (s) | Energy (kWh) |
|---|---|---|---|---|---|
| 02 | BAAS | 0.993 | 0.032 | 0.053 | 0.032 |
| *02* | *EdgeCloudForge* | *1.287* | *0.027* | *0.048* | *0.033* |
| 02 | Pred-3W | 2.494 | 0.025 | 0.046 | 0.032 |
| 02 | Reactive | 4.294 | 0.041 | 0.062 | 0.034 |
| 02 | SLOPred | **0.992** | 0.031 | 0.052 | 0.032 |
| 03 | BAAS | **2.444** | 0.038 | 0.058 | 0.030 |
| *03* | *EdgeCloudForge* | *2.978* | *0.038* | *0.059* | *0.030* |
| 03 | Pred-3W | 8.633 | 0.025 | 0.046 | 0.029 |
| 03 | Reactive | 8.969 | 0.041 | 0.062 | 0.031 |
| 03 | SLOPred | 2.444 | 0.036 | 0.057 | 0.030 |

Table 5.7: Performance metrics for proactive approach for function *dnn1* on Globus Endpoints.

| Endpoint | Proactive | QTime (s) | ExecTime (s) | CompTime (s) | Energy (kWh) |
|---|---|---|---|---|---|
| 02 | BAAS | 5.126 | 0.003 | 0.024 | 0.021 |
| *02* | *EdgeCloudForge* | *1.077* | *0.003* | *0.024* | *0.021* |
| 02 | Pred-3W | **0.883** | 0.002 | 0.023 | 0.021 |
| 02 | Reactive | 3.564 | 0.003 | 0.023 | 0.022 |
| 02 | SLOPred | 5.440 | 0.002 | 0.024 | 0.021 |
| 03 | BAAS | 9.734 | 0.003 | 0.024 | 0.021 |
| *03* | *EdgeCloudForge* | ***2.068*** | *0.004* | *0.024* | *0.021* |
| 03 | Pred-3W | 2.583 | 0.001 | 0.022 | 0.020 |
| 03 | Reactive | 5.131 | 0.003 | 0.024 | 0.020 |
| 03 | SLOPred | 8.317 | 0.001 | 0.023 | 0.020 |

Raspberry Pis and 20% Jetson Xavier devices. The Large scenario has 30 nodes with 500 Mbit/s, and both devices are included equally.

We compare the results of the optimization framework against the reactive approach aggregated over all four weeks, shown in Figure 5.18. Results show that *EdgeCloudForge* was able to generate datasets that allowed the proactive approach to reduce SLO violations across infrastructures. However, we see differences in terms of resource usage. Figure 5.18a shows that *EdgeCloudForge* required at times up to 50 function instances, drastically increasing the usage in comparison to the *Reactive* approach. While the resource usage was sometimes high, we also saw a low deviation in terms of SLO violations, leading us to believe that a more pronounced model could keep SLO violations and resource low. Figure 5.18b, focusing on the Globus pattern, indicates that the *Reactive* approach could not absorb the requests even though it has a high average of running instances. This indicates once more that this approach is not suitable for serverless users. To summarize,




(a) Huawei request pattern *49*. *EdgeCloudForge's* models lower SLO violations across infrastructures while increasing resource usage.

(b) Globus Endpoint request pattern *02*. *EdgeCloudForge's* models lower SLO violations and maintain a similar resource usage.

Figure 5.18: Results across heterogeneous infrastructures for two request patterns. *EdgeCloudForge's* models consistently lower SLO violations and outperform the reactive approach.

*EdgeCloudForge* generated datasets based on the ECDS that allowed us to train a model that can be used across infrastructures and outperform the *Reactive* approach.

### 5.2.5 Related Work

Trace-driven modeling has emerged as a powerful technique for predicting system behavior and optimizing resource management in edge-cloud environments. Abdullah et al. [AIEB19] show how to use trace-driven modeling to improve the performance of a proactive autoscaler. Ray and Banerjee [RB20] present a trace-driven framework for modeling and verifying mobility-aware service allocation and migration policies in mobile edge computing. Given the scarcity of large, realistic datasets in emerging network environments, several researchers have used synthetic data generation to fill this gap. In this context, Pandey et al. [PTR+23] propose a resource-efficient synthetic data generator that uses multi-head attention and BiLSTM layers to create realistic mobile traffic and user behavior datasets, essential for evaluating MEC performance in 5G networks. Enhancing this approach, 5GT-GAN [PTI+23] integrates autoregressive techniques within a GAN framework to preserve both static and temporal features of mobile Internet traffic. Moreover, discrete-event simulation methods by Chan et al.[CRP22] and advanced synthesis frameworks like Meta-Sim by Kar et al. [KPL+19] further illustrate how synthetic data can bridge the gap when real-world production data is scarce. Furthermore, Kannan [Kan22] introduces a JSON schema-driven framework tailored to generate realistic sensor data in edge analytics of IoT. Building on synthetic data generation techniques, predictive autoscaling methods have been introduced to optimize resource allocation further. Abdullah et al. [AIM+21] propose a predictive autoscaling approach that learns a reactive autoscaling policy by executing experiments before deploying the model in the homogeneous system. The caveat of this approach is the tight connection between the learned policy and infrastructure. In response, Vu et al. [VTK22] develop a hybrid autoscaling framework that combines both vertical and horizontal scaling in Kubernetes, leveraging machine learning–based workload forecasting and burst detection to balance quality-of-service requirements with resource efficiency

better. Similarly, Abdullah et al. [AIB+20] present a burst-aware predictive autoscaling method designed explicitly for containerized microservices by detecting workload spikes using advanced forecasting techniques. Recent work has also highlighted the potential of co-simulation frameworks to improve scheduling decisions in complex edge-cloud environments. The works of Tuli et al. [TPS+22][TCJ23] illustrate how co-simulation can facilitate container scheduling by using the simulator as a training ground for developing robust scheduling models. In contrast, our approach highlights the need for co-simulation to adapt strategies ahead of deployment and for different environments and workload patterns. The need for this arises from the heterogeneous environment of the edge-cloud continuum, which can render proactive approaches useless without proper fine-tuning.

### 5.2.6 Summary of Bootstrapping Parameters

Serverless Edge Computing promises autonomous and elastic management of functions in response to incoming client workloads. However, cold startup time and changing edge-cloud infrastructure render reactive approaches less effective and favor proactive approaches. Proactive approaches that rely on trained models must overcome a specific period before producing sufficient estimations. Synthetic dataset generation can speed up this process using a co-simulation-based Bayesian exploration of the Edge-Cloud Domain Space (ECDS). The ECDS dictates under which conditions serverless edge platforms should operate and enables us to generate scenarios based on historical data (e.g., one day). Our framework, EdgeCloudForge, explores the domain space and generates synthetic datasets, which it uses to train a prediction model. EdgeCloudForge only requires one day of monitoring to generate scenarios and synthesize datasets. Our simulation results show that the synthetic datasets can train our model to reduce SLO violations over the reactive approach and approaches from related work [AIEB19, AIB+20] across different real-world serverless edge-cloud workloads. Results also indicate that the ECDS can capture heterogeneous infrastructures and synthesize datasets so that the model can reduce SLO violations under more uncertain conditions related to the infrastructure. Under heterogeneous conditions, our model trained on such a dataset clearly outperformed the reactive approach. However, we believe that our autoscaling can be improved by explicitly considering heterogeneous hardware and function configurations [PN25]. Therefore, future work includes a heterogeneity-aware autoscaling algorithm and exploring a workload prediction model that can be integrated into EdgeCloudForge. Future work also includes a more detailed analysis of the impact the resource weight has on the datasets and testing different prediction models when synthesizing datasets. Finally, we want to investigate uncertainties inherently present in such heterogeneous environments [NCTD15] and how they can be integrated to achieve more accurate predictions.

## 5.3 Summary

This chapter has shown how simulations can build the foundation of a self-adaptive platform. We first introduced in Section 5.1 the Orchestration Parameter Optimization

Problem that aims at finding parameters to minimize SLO violations and tackle Orchestration Strategy Drifts. We implemented *SimuScale*, which solves the problem and tunes thresholds during runtime to reduce resource usage and optimize performance. While our solution already transfers real-world state into the simulation at runtime, a future challenge lies in keeping the simulation on par with the real world. This entails tracking the execution times of functions to update the simulation model. The second approach, shown in Section 5.2, focused on generating synthetic datasets that aided the autoscaler to reduce SLO violations. Specifically, we proposed the framework *EdgeCloudForge*, which uses Bayesian optimization to find simulation scenarios that aim to synthesize datasets that improve the robustness of the prediction model. To this end, we formulated the Edge-Cloud Domain Space, which defines the boundaries of possible scenarios and which builds our search space. *EdgeCloudForge* has shown that it can train prediction models that can be used across different infrastructures and workload patterns based on a single day of historical data.

### Future Work

In the thesis, we focused on the optimization of parameters for Serverless Edge Computing; however, the ideas can be applied to other application domains and use cases as well. Specifically, data stream processing applications, such as those popular in the Internet of Things domain [Yan17], must place data stream processors across the edge-cloud continuum; therefore, they can benefit from our contributions. For example, data stream processing platforms can leverage the simulations to perform *what-if* scenarios and evaluate the impact of different stream processing operator placements before deciding which is best. This process is similar to the one we have used in *SimuScale*, where we determined optimal parameters for autoscaling.

Platform providers can also benefit from simulations when designing and building infrastructures. The simulations aid decision-making by enabling fast design cycles to test the processing power of the infrastructure and can help find the optimal set of devices for the intended applications. At the same time, simulations can also help upgrade existing infrastructure. If new applications are deployed, simulations can find the most cost-efficient way to support them by finding the minimum number of new devices needed to satisfy user goals. The Edge-Cloud Domain Space presented in Section 5.2 can help navigate the scenario space when searching for new hardware.

# CHAPTER 6

# Conclusion

This chapter concludes the thesis. Section 6.1 presents a summary that covers the topics of this thesis and our contributions. Section 6.2 discusses our research questions, and Section 6.3 gives a brief outlook on future research directions and open issues.

## 6.1 Summary

Edge Intelligence is a paradigm that unites resources from edge to cloud to facilitate the next generation of pervasive AI applications. These applications range from low-latency inference tasks on edge devices to long-running training tasks on cloud infrastructure. Therefore, deploying them is complex and needs to happen automatically. Serverless Computing, focusing on fine-grained functions, is built on the promise that platform providers handle the complete application deployment life cycle. Customers only provide their applications (i.e., in the form of functions) and upload them. Platforms must route client requests to running function instances, which are created in response to the workload by the autoscaler. Serverless Edge Computing is the natural extension that offers the same autonomous application management across the edge-cloud continuum. However, this platform type has several challenges, such as mobility and energy-awareness.

In this thesis, Chapter 2 introduced various requirements for platforms that support Edge Intelligence as a Service, and a requirements model for Serverless Edge Computing platforms. Specifically, Section 2.1 first laid out how a possible Edge Intelligence as a Service platform can be built, including use cases and potential platforms. We envision such a platform autonomously orchestrating EI applications from training to inference between the edge and the cloud. Serverless Edge Computing can suit those needs, but faces several challenges that have yet to be addressed. Section 2.2 introduced a requirements model using multiple criteria and analyzed current public offerings and works from research. It also presented challenges for future iterations of Serverless Edge Computing platforms.

Based on this analysis, we tackled two challenges: mobile users and energy consumption. We proposed both approaches in Chapter 3, where one approach relied on thresholds and the other on a prediction model. Subsequently, we identified the challenge of finding the proper parameters for those orchestration strategies and proposed a self-adaptive platform to overcome this challenge.

A self-adaptive platform should optimize these parameters when first deploying the strategies (i.e., bootstrapping parameters) or during runtime. To this end, Chapter 4 introduces a testing framework for Serverless Edge Computing and a Serverless Edge Computing simulator. The testing framework includes tools for monitoring and evaluating orchestration strategies. The simulator is based on a realistic domain design that allows us to replicate the real-world state in the simulation.

Our proposal is a co-simulation-driven approach that feeds real-world monitoring data into the simulation, which can then execute scenarios based on the current real-world state.

In Chapter 5, we unite the testing framework and simulation tool to build the self-adaptive platform. We propose two strategies that tune threshold parameters during runtime and help bootstrap prediction-based approaches by generating synthetic datasets.

## 6.2 Research Questions

In Section 1.2 we posited our research questions that aim at (1) deriving requirements for Edge Intelligence and Serverless Edge Computing platforms, (2) identifying orchestration strategies for mobility- and energy-aware Serverless Edge Computing, (3) building tools for a self-adaptive Serverless Edge Computing platform, and (4) developing and evaluating methods that use simulation to facilitate a self-adaptive platform. The following discusses these questions based on the contributions presented in this thesis.

**RQ1. What are the requirements that Serverless Edge Computing needs to fulfill for Edge Intelligence?**

Edge Intelligence applications introduce various tasks and requirements that edge-cloud platforms must address. The tasks range from sensing, preprocessing, training, and inference. Each task introduces different requirements, such as addressability, security, coordination, and performance, that must be met for a holistic Edge Intelligence platform. One aspect that we also highlight is the required Elasticity of applications. Elastic requirements range from security to performance and are proposed by customers and implemented by platforms. Based on these platform requirements, we determined that the emerging Serverless Edge Computing paradigm can be a promising candidate. To that end, we deepened our requirements model for EI by proposing a set of criteria that Serverless Edge Computing platforms must fulfill, especially to meet the requirements of AI edge-cloud applications. The criteria are split into design time and runtime to cover requirements from building to deploying and running serverless edge applications. The design time focuses on aspects that involve developers and how they interact with the

platform. For example, how they can specify SLOs for their applications is crucial for EI. Run-time criteria focus on what platforms offer and how they are implemented. An example is Platform Intelligence, which is based on the underlying strategy of reaching goals, such as SLOs. We also introduced maturity levels for the proposed criteria and evaluated multiple works from academia, open-source projects, and commercial offerings. To answer the research question, EI applications pose several requirements that require autonomous application management, which Serverless Edge Computing can offer. However, based on our requirements model, many offerings have not yet fully matured. The requirements are manifold, while EI applications pose general requirements around the execution of applications, such as performance or security. The requirements for Serverless Edge Computing address gaps in current implementations. We split them into design and runtime to address the needs for developers to express their needs (e.g., SLOs) and the runtime that focuses on implementing those.

Especially challenges around location-awareness to support mobile users and increasing energy consumption across the edge-cloud continuum overlap EI and Serverless Edge Computing. Therefore, we introduced two novel approaches to address these challenges.

**RQ2. Which orchestration strategies can build the foundation for mobility- and energy-aware Serverless Edge Computing platforms?**

Our mobility-aware approach [RRD+22] introduced a novel decentralized serverless edge platform and implemented a request router, a scheduler, and a scaling system consisting of global and local autoscalers. The local autoscalers aggregate high-frequency monitoring data into a complex pressure value that gives direction between clusters, while the global autoscaler uses this information from all autoscalers to determine where users are moving and where new function instances are needed. This information is passed on to the scheduler. Because pressure aggregates monitoring data, clusters are not required to send all data to the cloud. Therefore, it avoids large amounts of network overhead. Our system results in increased performance and decreased network overhead of function calls by effectively shifting function instances across clusters based on the users. However, the challenge of this approach is finding suitable thresholds to scale in or out. Besides, our approach also requires other parameters (such as SLOs) that clients must set properly. While our threshold-based system can build the foundation of mobility-aware orchestration strategies, it introduces new challenges in finding parameters.

Next, we introduced a novel graph-based approach to model infrastructure devices, including the individual components and their resource usage. This graph is used as input for a GNN that predicts the state's power consumption. Therefore, with appropriate profiling data, we can perform *what-if* scenarios and determine the energy consumption across devices to select the one with the lowest impact. We evaluated this container scheduler in a data-center scenario and tested the GNN for accuracy, which had low error values. Like our threshold-based system, the training data (the parameters) are crucial for a well-performing model and accurate estimations. However, gathering training data is time-consuming and depends on the prediction model. Some models rely on profiling data that can be generated in an isolated environment, but others rely on real-world historical

data. The former introduces overhead to set up and run these profiling experiments, while the latter has to wait for real-world data to be generated.

To answer the research questions, we have successfully implemented and evaluated a decentralized threshold-based autoscaling strategy to tackle the mobility challenge and used a prediction-based approach to implement an energy-aware scheduler. While our methods are suitable, we discovered the issue of finding the proper parameters for threshold- and prediction model-based systems.

**RQ3. What tools and methods are required to build a self-adaptive Serverless Edge Computing platform?**

Serverless Edge platforms are complex systems with components that enable a fully automated application lifecycle, from uploading applications to autonomously managing them during runtime. This thesis focused on the main orchestration mechanisms, scheduling, autoscaling, and request routing. Chapter 3 has shown how thresholds and prediction models can build mobility- and energy-aware orchestration strategies. We have derived from these strategies that finding parameters is challenging and critical for good performance. Therefore, we look at what we need to do to support threshold-based and prediction-based systems, which builds the requirements for the tools and methods we can deploy in a self-adaptive system.

First, we look at bootstrapping parameters in the case of prediction-based strategies. To train a power prediction model, we need to profile devices a priori to train models that can capture the power consumption model of the devices. While our approach was made to be independent of specific applications, i.e., by relying on system-level metrics, it still needs a spectrum of metrics to achieve good performance. We have also shown in Section 5.2 how limited historical data can achieve sub-optimal prediction performance for autoscaling. A requirement for tools to facilitate bootstrapping prediction model parameters is the ability to generate training datasets that cover a broad spectrum of possible scenarios in advance.

Second, based on our threshold-based approach, it can be challenging to determine their effectiveness in satisfying operational goals even with a small number of parameters. Specifically, we have shown in our work in Section 5.1 that thresholds are not intuitive to set even in simple scenarios. However, instead of tuning these thresholds before deploying them, we want to explore the tuning of those during runtime. The tools of a self-adaptive platform must be able to mimic the real world at runtime and offer the ability to find parameters at the same time.

To summarize, tools are required to monitor and profile the infrastructure, generate synthetic datasets, and be fed in real time to facilitate optimization at runtime.

Therefore, we propose two tools and methods to build our system's foundation. The first tool is our end-to-end platform orchestration evaluation framework that works in tandem and provides workload generators and monitoring features. The second tool is our serverless edge-cloud simulator *faas-sim* that is based on a realistic domain model

(i.e., entities exist in correspondence with real-world deployed ones). The simulator allows us to inject monitoring data and replicate the real-world state, thus implementing a co-simulation. The two methods we propose and evaluate in Chapter 5 are optimization-driven and revolve around exploring different scenarios to generate synthetic datasets and update thresholds using the co-simulation during runtime.

To answer the research question, the concrete methods for self-adaptive systems depend on the orchestration strategy's implementation. Therefore, they must be flexible and build on a domain model akin to the real world. We determined that they should be used to bootstrap and tune them during runtime, and that a lack of historical data can hinder prediction models. Furthermore, testing thresholds at runtime can yield suboptimal results. Therefore, we deem simulations a promising candidate that can explore scenarios in advance to generate synthetic datasets and enable optimization-driven approaches to fine-tune parameters.

**RQ4. How can co-simulation facilitate a self-adaptive Serverless Edge Computing platform?**

We have devised in **RQ3** that co-simulation can help fine-tune parameters at runtime and bootstrap them. In Chapter 5, we have shown our approaches to implement two novel strategies that rely on co-simulation and address self-adaptation of parameters during runtime and to bootstrap them.

First, Section 5.1 evaluated a method to tune parameters during runtime using co-simulation that builds on live monitoring data to rebuild the infrastructure in the simulation and execute potential future scenarios. We developed a threshold-based autoscaling method and introduced different co-simulation-driven optimization strategies to balance resource usage and performance. Furthermore, we have unified the programming interfaces of our tools that we presented in Chapter 4. This allowed us to program the orchestration strategies once, deploy them on the real-world testbed, and reuse them in the simulation, further strengthening the congruence between the two. Furthermore, due to the simulator's realistic domain design, we could mimic important aspects, such as the infrastructure and function instances, in the simulation. Contrary to bootstrapping, where we made a few assumptions about the target environment, which led to a large simulation space, we saw that aspects like infrastructure are given when tuning during runtime. However, workload prediction is necessary to enable this, which limits our work due to the assumption of having the workload at hand during optimization. Through our experiments on a testbed that fed live monitoring data into the simulation, our tools could work in this environment and update the thresholds in real time based on the simulated scenarios. However, we have also determined that the simulation speed is crucial for timeliness and how far ahead we should simulate.

Second, Section 5.2 addressed the challenge of bootstrapping parameters. To this end, we proposed a prediction-based autoscaler that we tuned with a range of historical data and introduced our concept of edge-cloud continuum domain space that can capture the possible heterogeneity of edge-cloud scenarios, including infrastructure and workloads. We

implemented an approach to show that combining co-simulation with scenario exploration can generate synthetic datasets. This implementation has also shown the challenges of such an approach; generating plausible scenarios is a complex task with almost infinite possible solutions. However, our Bayesian-based guided search generated synthetic datasets in our experiments that let the model outperform the trained historical data model.

To answer the research question, co-simulation can facilitate self-adaptive Serverless Edge Computing platforms in generating synthetic datasets before deployment and using optimization to search for new parameters during runtime. The former has revealed the challenge of generating plausible simulation scenarios to train robust prediction models. At the same time, the latter has emphasized the challenge of synchronizing the simulation with the real-world state. The contributions of Chapter 5 have addressed both and shown that co-simulation is a promising avenue for future self-adaptive platforms.

## 6.3 Limitations & Future Work

In this thesis, we investigated how orchestration strategies for Serverless Edge Computing can handle the challenges of mobility and energy consumption. We identified the challenge of proper parameterization [SGvK+22] of our strategies and proposed a self-adaptive platform to handle this. While we were able to develop methods to tune or bootstrap parameters, there are limitations to the presented contributions, as well as future work remains open that builds upon the ideas of this thesis. The following presents these limitations and a roadmap for future work. The roadmap includes steps to optimize the simulation to accommodate large-scale scenarios, deepen the knowledge of self-adaptive optimization using the Edge-Cloud Domain Space, and add advanced support for Serverless Edge Computing platforms, culminating in an effort to tackle the challenges that come with the integration into real-world edge-cloud deployments.

### 6.3.1 Analytical and prediction-based simulations

A limitation of our work is the implementation of our simulation, which can be slow and resource-intensive at times. Specifically, our simulator builds on top of SimPy [Mat08], a Python-based simulation engine that internally uses an event-driven system. In our evaluation, we saw the downsides of using this engine, such as slowed-down performance in more complex scenarios. Nevertheless, resource-efficient and fast-paced simulations are crucial for future work that adopts our idea of supporting orchestration strategies with simulations.

Based on that, we also investigated other simulation approaches, such as analytical and prediction-based approaches. Specifically, in Section 3.2, we have developed a Graph Neural Network to predict the energy consumption of a system based on resource usage, thus simulating the power draw given a set of applications. This has enabled us to perform *what-if* simulations that investigate power consumption if specific applications

run on a given system. In our other work, exploring server consolidation for Serverless Computing, we have introduced an analytical autoscaling model to estimate the number of servers needed to host a specific number of function instances. In both cases, the overhead of executing the simulation is reduced, but it introduces other challenges. The prediction-based approach is challenged to acquire enough training data to yield accurate estimations. The analytical model requires a deep understanding of the underlying system to formulate appropriate mathematical models.

To this end, the first step on our roadmap for future work would be integrating multiple types of simulations into the self-adaptive platform to improve accuracy, execution speed, versatility, and scalability.

### 6.3.2 Simulation Space Exploration

In Chapter 5, we introduced the Edge-Cloud Domain Space, which covers different configurations of the edge-cloud continuum. This configuration space facilitates the generation of plausible simulation scenarios for edge-cloud clusters. We used this space to explore a search-based approach to generate synthetic datasets for possible infrastructure settings. However, a limitation of our work is that the space only included the most relevant characteristics, such as workload intensity, infrastructure configuration, and serverless functions. Therefore, two avenues can be explored to further enrich the exploration of space. First, we could extend the domain space used as input to generate plausible simulation scenarios. For example, an issue prevalent in the edge-cloud continuum that we have not addressed in our design space is the failure of devices. Specifically, edge devices are prone to fail due to network outages, hardware failures, or other factors. Therefore, extending our domain space definition to include this characteristic can make the self-adaptive platform aware of such failures and optimize against them.

Second, multiple objective functions should be tested when exploring the space. In our approach, presented in Section 5.2, we focused on SLO violations by navigating the space to find configurations for which our platform performed worse to generate more training data. However, we could also use the search to recommend cluster configurations using other objectives, such as energy consumption. That way, platform providers can use the platform to find infrastructure settings predicted to have high energy consumption, thus optimizing the platform for those situations.

Therefore, the second step on our roadmap is performing large-scale simulations to explore more diverse characteristics of the edge-cloud continuum and deepen our exploration of the Edge-Cloud Domain Space.

### 6.3.3 Extended Orchestration Strategy Support

Our self-adaptive approaches have focused on supporting autoscaling (i.e., Section 5.1 & Section 5.2) but did not consider function scheduling or request load balancing. While our mobility-aware approach presented in Section 3.1 has implemented all three strategies, the evaluation focused on an individual short-lived function whose execution took the same

amount of time across different devices. This is a crucial limitation as Edge Intelligence applications also encompass long-running applications, such as model training. Based on that, the following summarizes three possible avenues for future work that build and extend our work:

1. Develop a heterogeneity-aware scheduling mechanism. This tackles a fundamental issue in edge-cloud computing and fills the gap in our mobility-aware approach.

2. Devise an autoscaling and scheduling approach that is suitable for short-lived and long-lived functions. Both are crucial for Edge Intelligence applications (i.e., inference and model training).

3. Extend the self-adaptive approaches to support the newly developed orchestration strategies, including load balancing.

### 6.3.4 Extended Self-Adaptive Platform

Our work has focused on optimizing parameters of function orchestration strategies that manage individual functions, such as autoscaling. However, a Serverless Edge Computing platform also consists of other components, such as Backend as a Service (BaaS) [NRF+22, RND23]. BaaS refers to components that offer essential backend services to Serverless Edge Computing. These services include storage, messaging middleware, and others [NRF+22]. Storage commonly refers to object storage solutions that serverless functions often use to upload or download files. Messaging middleware is crucial to enable real-time and low-latency message exchange between clients and functions. These components enable many specialized applications and paradigms; a self-adaptive platform can also optimize these mechanisms. Specifically, we can simulate incoming workloads using co-simulation and determine which objects are most often retrieved (i.e., AI models for inference) and where they are downloaded (i.e., from which region). The self-adaptive platform can use this knowledge to proactively move such objects to reduce network overhead by downloading them from remote locations. Similarly, messaging middleware can be placed in regions of frequent access to lower network latency.

Besides extending support for BaaS, we think function compositions or workflows can profit from a self-adaptive platform. Serverless functions are often composed into complex applications in which functions can call each other upon completion. For example, Edge Intelligence's inference pipeline consists of multiple AI models to perform more complex tasks, such as video monitoring [CSM+20]. While this thesis focuses on orchestrating individual functions, a self-adaptive platform can similarly aid workflows.

This extended platform builds the fourth step on our roadmap and lays the groundwork for the final step of our vision.

### 6.3.5 Self-Adaptive Platform for Large-Scale Infrastructure

Our self-adaptive approach has so far been evaluated only on a small-scale testbed and with smaller-sized simulated infrastructures. This is in part owed to slow-performing simulations, which have already been mentioned as a limitation and introduced as future work. However, besides speeding up simulation execution time, it is also critical to think about large-scale deployments and how to integrate the simulation with them. Specifically, the monitoring necessary for our approach can, at a large scale, introduce non-negligible network overhead on the infrastructure. This is a crucial step in realizing the idea of our self-adaptive platform for real-world systems. Two avenues for future work can tackle this issue:

- Dynamic monitoring frequency: Instead of having a static monitoring interval, the platform could dynamically set the monitoring interval across the infrastructure. Consider a geo-distributed multi-cluster platform; at times, it might not be necessary to gather from all parts high-frequency monitoring updates due to idle resources. This could save network bandwidth without negatively impacting the self-adaptation process.
- Hierarchical simulations: our implementation in Section 5.1 currently simulates the complete infrastructure for each optimization round. However, in large-scale clusters, this could become a bottleneck and reduce the ability of the platform to react quickly. Therefore, splitting the simulation across the infrastructure could leverage distributed compute resources to enable faster simulation execution, improve the optimization outcomes, and support the edge-cloud continuum.

This concludes our roadmap for future work; the final outcome would be an efficient self-adaptive platform that can be deployed on real-world large-scale infrastructures.

# Overview of Generative AI Tools Used

The following tools were used for the creation of this thesis: Grammarly[1] and Writefull[2]. Both tools have been used for editing and proofreading (e.g., grammar checking, etc.).

[1] https://www.grammarly.com
[2] https://www.writefull.com/

# Übersicht verwendeter Hilfsmittel

Die folgenden Tools wurden für die Erstellung dieser Arbeit verwendet: Grammarly[3] und Writefull[4].

Beide Tools wurden für die Bearbeitung und das Korrekturlesen (z. B. für die Grammatikprüfung usw.) verwendet.

[3]https://www.grammarly.com
[4]https://www.writefull.com/

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography


[ABA21] Kazi Main Uddin Ahmed, Math HJ Bollen, and Manuel Alvarez. A review of data centers energy consumption and reliability modeling. *IEEE Access*, 9:152536–152563, 2021.

[ABB+17] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 265–283, Savannah, GA, November 2016. USENIX Association.

[ABC17] Maria Avgerinou, Paolo Bertoldi, and Luca Castellazzi. Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency. *Energies*, 10(10), 2017.

[ABD15] Vito Albino, Umberto Berardi, and Rosa Maria Dangelico. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of urban technology*, 22(1):3–21, 2015.

[ACG+16] Giuseppe Agapito, Barbara Calabrese, Pietro Hiram Guzzi, Mario Cannataro, Mariadelina Simeoni, Ilaria Caré, Theodora Lamprinoudi, Giorgio Fuiano, and Arturo Pujia. Dietos: A recommender system for adaptive diet monitoring and personalized food suggestion. In *2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–8. IEEE, 2016.

[ACR+18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: To-

wards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[ACT+13] MJ Aitkenhead, M Coull, W Towers, G Hudson, and HIJ Black. Prediction of soil characteristics and colour using data from the national soils inventory of scotland. *Geoderma*, 200:99–107, 2013.

[ADM+03] MJ Aitkenhead, IA Dalgetty, CE Mullins, Allan James Stuart McDonald, and Norval James Colin Strachan. Weed and crop discrimination using image analysis and artificial intelligence methods. *Computers and electronics in Agriculture*, 39(3):157–171, 2003.

[AE15] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[AHSS13] Fahd Al-Haidari, M Sqalli, and Khaled Salah. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 256–261. IEEE, 2013.

[AIB+20] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 15(3):1448–1460, 2020.

[AIEB19] Muhammad Abdullah, Waheed Iqbal, Abdelkarim Erradi, and Faisal Bukhari. Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 119–126, 2019.

[AIM+21] Muhammad Abdullah, Waheed Iqbal, Arif Mahmood, Faisal Bukhari, and Abdelkarim Erradi. Predictive autoscaling of microservices hosted in fog microdata center. *IEEE Systems Journal*, 15(1):1275–1286, 2021.

[AJH+20] Khaled Alwasel, Devki Nandan Jha, Eduardo Hernandez, Deepak Puthal, Mutaz Barika, Blesson Varghese, Saurabh Kumar Garg, Philip James, Albert Zomaya, Graham Morgan, and Rajiv Ranjan. Iotsim-sdwan: A simulation framework for interconnecting distributed datacenters over software-defined wide area network (sd-wan). *Journal of Parallel and Distributed Computing*, 143:17–35, 2020.

[AJH+21] Khaled Alwasel, Devki Nandan Jha, Fawzy Habeeb, Umit Demirbaga, Omer Rana, Thar Baker, Scharam Dustdar, Massimo Villari, Philip James, Ellis Solaiman, and Rajiv Ranjan. Iotsim-osmosis: A framework

for modeling and simulating iot applications over an edge-cloud continuum. *Journal of Systems Architecture*, 116:101956, 2021.

[AKAA17] Azza Allouch, Anis Koubâa, Tarek Abbes, and Adel Ammar. Roadsense: Smartphone application to estimate road conditions using accelerometer and gyroscope. *IEEE Sensors Journal*, 17(13):4231–4238, 2017.

[Amaa] Amazon. Amazon sagemaker. `https://aws.amazon.com/sagemaker/`. Accessed August 26, 2025.

[Amab] Amazon. Aws iot greengrass. `https://aws.amazon.com/greengrass/`. Accessed August 26, 2025.

[Ama14] Amazon. Aws lambda, 2014. `https://aws.amazon.com/lambda/`. Accessed August 26, 2025.

[Ama25] Amazon Web Services. How predictive scaling works, 2025. EC2 Auto Scaling User Guide.

[ATC+21] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: Vision and challenges. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery.

[ATCG22] Mohammad Sadegh Aslanpour, Adel N Toosi, Muhammad Aamir Cheema, and Raj Gaire. Energy-aware resource scheduling for serverless edge computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 190–199. IEEE, 2022.

[AWS16] AWS. Chalice, 2016. `https://github.com/aws/chalice`. Accessed August 26, 2025.

[BARGP11] Xavier P Burgos-Artizzu, Angela Ribeiro, Maria Guijarro, and Gonzalo Pajares. Real-time image processing for crop/weed discrimination in maize fields. *Computers and Electronics in Agriculture*, 75(2):337–346, 2011.

[BB22] Ayush Bhardwaj and Theophilus A Benson. Kubeklone: A digital twin for simulating edge and cloud microservices. Asia-Pacific Workshop on Networking (APNet 2022), page 7. Association for Computing Machinery, 2022.

[BBC+17] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[BBH+22] David Bermbach, Jonathan Bader, Jonathan Hasenburg, Tobias Pfandzelter, and Lauritz Thamsen. Auctionwhisk: Using an auction-inspired approach for function placement in serverless fog platforms. *Software: Practice and Experience*, 52(5):1143–1169, 2022.

[BCF+17] Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pages 89–103. Springer, 2017.

[BCG+22] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[BDADFM17] Devis Bianchini, Valeria De Antonellis, Nicola De Franceschi, and Michele Melchiori. Prefer: A prescription-based food recommender system. *Computer Standards & Interfaces*, 54:64–75, 2017.

[BFRS22] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 674–679. IEEE, 2022.

[BG96] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ, USA, second edition, 1996.

[BGJ+21] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[BHQT22] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. Neptune: Network- and gpu-aware management of serverless functions at the edge. In *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '22, page 144–155, New York, NY, USA, 2022. Association for Computing Machinery.

[Bis19] Ekaba Bisong. Kubeflow and kubeflow pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 671–685. Springer, 2019.

[BJCG21] Stephan Patrick Baller, Anshul Jindal, Mohak Chadha, and Michael Gerndt. Deepedgebench: Benchmarking deep neural networks on edge devices. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 20–30. IEEE, 2021.

[BM19] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *Proc. of IEEE ICFC'19*, pages 1–10, 2019.

[BMS20] David Balla, Markosz Maliosz, and Csaba Simon. Open source faas performance aspects. 2020 43rd International Conference on Telecommunications and Signal Processing (TSP), pages 358–364. IEEE, 2020.

[BPC+24] André Bauer, Haochen Pan, Ryan Chard, Yadu Babuji, Josh Bryan, Devesh Tiwari, Ian Foster, and Kyle Chard. The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems*, 153:558–574, 2024.

[BPKP20] Tristan Braud, ZHOU Pengyuan, Jussi Kangasharju, and HUI Pan. Multipath computation offloading for mobile augmented reality. In *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–10. IEEE, 2020.

[BQ21] Luciano Baresi and Giovanni Quattrocchi. Paps: A serverless platform for edge computing infrastructures. *Frontiers in Sustainable Cities*, 3:690660, 2021.

[BSK+22] Salil Bharany, Sandeep Sharma, Osamah Ibrahim Khalaf, Ghaida Muttashar Abdulsahib, Abeer S. Al Humaimeedy, Theyazn H. H. Aldhyani, Mashael Maashi, and Hasan Alkahtani. A systematic survey on energy-efficient techniques in sustainable cloud computing. *Sustainability*, 14(10), 2022.

[BTP+15] Vinay Bettadapura, Edison Thomaz, Aman Parnami, Gregory D Abowd, and Irfan Essa. Leveraging context to support automated food recognition in restaurants. In *2015 IEEE Winter Conference on Applications of Computer Vision*, pages 580–587. IEEE, 2015.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, page 44–54, USA, 2009. IEEE Computer Society.

[CBSG17] Charles E. Catlett, Peter H. Beckman, Rajesh Sankaran, and Kate Kusiak Galvin. Array of things: A scientific research instrument in the public way: Platform design and early lessons learned. In *Proceedings of the*

*2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, SCOPE '17, page 26–33, New York, NY, USA, 2017. Association for Computing Machinery.

[CFG+19] Francesco Cauteruccio, Giancarlo Fortino, Antonio Guerrieri, Antonio Liotta, Decebal Constantin Mocanu, Cristian Perra, Giorgio Terracina, and Maria Torres Vega. Short-long term anomaly detection in wireless sensor networks based on machine learning and multi-parameterized edit distance. *Information Fusion*, 52:13–30, 2019.

[CFL+18] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE, 2018.

[CGK+20] Thomas Chatzidimitris, Damianos Gavalas, Vlasios Kasapakis, Charalampos Konstantopoulos, Damianos Kypriadis, Grammati Pantziou, and Christos Zaroliagis. A location history-aware recommender system for smart retail environments. *Personal and Ubiquitous Computing*, pages 1–12, 2020.

[Cha99] Xinjie Chang. Network simulations with opnet. Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1, page 307–314, New York, NY, USA, 1999. IEEE, Association for Computing Machinery.

[CJ89] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1–14, 1989.

[CMT20] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Evaluation of cloud autoscaling strategies under different incoming workload patterns. *Concurrency and Computation: Practice and Experience*, 32(17):e5667, 2020.

[CNS16] Gianluigi Ciocca, Paolo Napoletano, and Raimondo Schettini. Food recognition: a new dataset, experiments, and results. *IEEE journal of biomedical and health informatics*, 21(3):588–598, 2016.

[CPK+19] Pawel Czarnul, Jerzy Proficz, Adam Krzywaniak, et al. Energy-aware high-performance computing: survey of state-of-the-art tools, techniques, and environments. *Scientific Programming*, 2019, 2019.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource

provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[CRdRR+22] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Systems*, 128:299–316, 2022.

[CRP22] K. C. Chan, Marsel Rabaev, and Handy Pratama. Generation of synthetic manufacturing datasets for machine learning using discrete-event simulation. *Production & Manufacturing Research*, 10(1):337–353, 12 2022. Copyright - © 2022 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group. This work is licensed under the Creative Commons Attribution License http://creativecommons.org/licenses/by/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2024-12-18.

[CRS+19] Yair Carmon, Aditi Raghunathan, Ludwig Schmidt, Percy Liang, and John C Duchi. Unlabeled data improves adversarial robustness. *arXiv preprint arXiv:1905.13736*, 2019.

[CSM+20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. Proceedings of the 1st ACM Symposium on Cloud Computing, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[CTCL19] Jingjing Cao, Junwei Tan, Yuanlai Cui, and Yufeng Luo. Irrigation scheduling of paddy rice using short-term weather forecast data. *Agricultural water management*, 213:714–723, 2019.

[CZW+22] René Caspart, Sebastian Ziegler, Arvid Weyrauch, Holger Obermaier, Simon Raffeiner, Leon Pascal Schuhmacher, Jan Scholtyssek, Darya Trofimova, Marco Nolden, Ines Reinartz, et al. Precise energy consumption measurements of heterogeneous artificial intelligence workloads. In *International Conference on High Performance Computing*, pages 108–121. Springer, 2022.

[Dac17] Scott G Dacko. Enabling smart retail settings via mobile augmented reality shopping apps. *Technological Forecasting and Social Change*, 124:243–256, 2017.

[DIPW20] Anirban Das, Shigeru Imai, Stacy Patterson, and Mike P Wittie. Performance optimization for edge-cloud serverless platforms via dynamic task placement. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 41–50. IEEE, 2020.

[DL19] Bradley Denby and Brandon Lucia. Orbital edge computing: Machine inference in space. *IEEE Computer Architecture Letters*, 18(1):59–62, 2019.

[Don17] Jason A Donenfeld. Wireguard: next generation kernel network tunnel. In *NDSS*, pages 1–12, 2017.

[DPW18] Anirban Das, Stacy Patterson, and Mike Wittie. Edgebench: Benchmarking edge computing platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 175–180. IEEE, 2018.

[DRM+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 1–14, Renton, WA, July 2019. USENIX Association.

[DWF15] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications surveys & tutorials*, 18(1):732–794, 2015.

[DZF+20] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.

[EBG+21] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. Proceedings of the 22nd International Middleware Conference, page 248–259, New York, NY, USA, 2021. Association for Computing Machinery.

[EGHS16] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[FCFMJ+18] Damián Fernández-Cerero, Alejandro Fernández-Montes, Agnieszka Jakóbik, Joanna Kołodziej, and Miguel Toro. Score: Simulator for cloud

optimization of resources and energy consumption. *Simulation Modelling Practice and Theory*, 82:160–173, 2018.

[FCFMJ+20] Damián Fernández-Cerero, Alejandro Fernández-Montes, F. Javier Ortega, Agnieszka Jakóbik, and Adrian Widlak. Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory*, 101:101966, 2020. Modeling and Simulation of Fog Computing.

[FFH13] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *2013 35th international conference on software engineering (ICSE)*, pages 512–521. IEEE, 2013.

[FL19] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[FLC+21] Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E. Papka. Deep reinforcement agent for scheduling in hpc. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 807–816, 2021.

[FLR+22] Yuping Fan, Zhiling Lan, Paul Rich, William Allcock, and Michael E. Papka. Hybrid workload scheduling on hpc systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 470–480, 2022.

[GAGF17] Raffaele Gravina, Parastoo Alinia, Hassan Ghasemzadeh, and Giancarlo Fortino. Multi-sensor fusion in body sensor networks: State-of-the-art and research challenges. *Information Fusion*, 35:68–80, 2017.

[GCZ20] Shuxin Ge, Meng Cheng, and Xiaobo Zhou. Interference aware service migration in vehicular fog computing. *IEEE Access*, 8:84272–84281, 2020.

[GFD22] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE, 2022.

[GLCCT22] Alfredo García, David Llopis-Castelló, and Francisco Javier Camacho-Torregrosa. From the vehicle-based concept of operational design domain to the road-based concept of operational road section. *Frontiers in Built Environment*, 8, 2022.

[GMCR04] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Brazilian symposium on artificial intelligence*, pages 286–295. Springer, 2004.

[GND17] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017)*, page Article No. 28. ACM, 2017.

[Goo] Google. Vertex ai. `https://cloud.google.com/vertex-ai`. Accessed August 26, 2025.

[Goo16] Google. Cloud functions, 2016. `https://cloud.google.com/functions`. Accessed August 26, 2025.

[Goo25] Google Cloud. Scaling based on predictions, 2025. `https://cloud.google.com/compute/docs/autoscaler/predictive-autoscaling`. Accessed August 26, 2025.

[GPB+21] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. Befaas: An application-centric benchmarking framework for faas platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–8. IEEE, 2021.

[GRN13] Wei Guo, Uday K Rage, and Seishi Ninomiya. Illumination invariant segmentation of vegetation for time series wheat images based on decision tree model. *Computers and electronics in agriculture*, 96:58–66, 2013.

[GSAN+22] Tom Goethals, Merlijn Sebrechts, Mays Al-Naday, Bruno Volckaert, and Filip De Turck. A functional and performance benchmark of lightweight virtualization platforms for edge computing. In *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 60–68. IEEE, 2022.

[GSSS20] Dhruv Garg, Prathik Shirolkar, Anshu Shukla, and Yogesh Simmhan. Torquedb: Distributed querying of time-series data from edge-local storage. In *European Conference on Parallel Processing*, pages 281–295. Springer, 2020.

[GTB+18] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: a survey. *ACM Computing Surveys (CSUR)*, 51(3):1–33, 2018.

[GVDGB17] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[GWZ+20] Yan Guo, Shangguang Wang, Ao Zhou, Jinliang Xu, Jie Yuan, and Ching-Hsien Hsu. User allocation-aware edge cloud placement in mobile edge computing. *Software: Practice and Experience*, 50(5):489–502, 2020.

[GZGD20] Yi Gao, Jiadong Zhang, Gaoyang Guan, and Wei Dong. Linklab: A scalable and heterogeneous testbed for remotely developing and experimenting iot applications. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 176–188. IEEE, 2020.

[HBDQ+18] Hadi Habibzadeh, Andrew Boggio-Dandry, Zhou Qin, Tolga Soyata, Burak Kantarci, and Hussein T Mouftah. Soft sensing in smart cities: Handling 3vs using recommender systems, machine intelligence, and data analytics. *IEEE Communications Magazine*, 56(2):78–86, 2018.

[HBR+20] Justin Hu, Ariana Bruno, Brian Ritchken, Brendon Jackson, Mateo Espinosa, Aditya Shah, and Christina Delimitrou. Hivemind: A scalable and serverless coordination control platform for uav swarms. *arXiv preprint arXiv:2002.01419*, 2020.

[HCC+14] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[HDWS20] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, WWW '20, page 2704–2710, New York, NY, USA, 2020. Association for Computing Machinery.

[HHZ+16] Brandon Heung, Hung Chak Ho, Jin Zhang, Anders Knudby, Chuck E Bulmer, and Margaret G Schmidt. An overview and comparison of machine-learning techniques for classification purposes in digital soil mapping. *Geoderma*, 265:62–77, 2016.

[HKO21] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. $\mu$actor: Stateful serverless at the edge. In *Proceedings of the 1st Workshop on Serverless mobile networking for 6G Communications*, pages 1–6, 2021.

[HLB+23] Walid A Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. *arXiv preprint arXiv:2302.08681*, 2023.

[HLR+08] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. SIGCOMM'08 demonstration, page 527. Association for Computing Machinery, 2008.

[HLY+22] Yaodong Huang, Zelin Lin, Tingting Yao, Xiaojun Shang, Laizhong Cui, and Joshua Zhexue Huang. Mobility-aware seamless virtual function migration in deviceless edge computing environments. In *2022 IEEE 42nd*

*International Conference on Distributed Computing Systems (ICDCS)*, pages 447–457. IEEE, 2022.

[HMD+18] Michael Halstead, Christopher McCool, Simon Denman, Tristan Perez, and Clinton Fookes. Fruit quantity and ripeness estimation using a robotic vision system. *IEEE robotics and automation LETTERS*, 3(4):2995–3002, 2018.

[HML19] Håkon Hukkelås, Rudolf Mester, and Frank Lindseth. Deepprivacy: A generative adversarial network for face anonymization. In *International Symposium on Visual Computing*, pages 565–578. Springer, 2019.

[HMR+19] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. Modelops: Cloud-based lifecycle management for reliable and trusted ai. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 113–120. IEEE, 2019.

[HPS+15] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[HSS+19] David Haja, Mark Szalay, Balazs Sonkoly, Gergely Pongracz, and Laszlo Toka. Sharpening kubernetes for the edge. In *Proc. of the ACM SIGCOMM'19*, pages 136–137, 2019.

[HSW+21] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor CM Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. IEEE INFOCOM 2021-IEEE Conference on Computer Communications, pages 1–10. IEEE, 2021.

[htt] HTTP Archive: State of the Web. `https://httparchive.org/reports/state-of-the-web`. Accessed August 26, 2025.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZC+20] Jichong Han, Zhao Zhang, Juan Cao, Yuchuan Luo, Liangliang Zhang, Ziyue Li, and Jing Zhang. Prediction of winter wheat yield based on multi-source data and machine learning in china. *Remote Sensing*, 12(2):236, 2020.

[HZC21] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015. arXiv https://arxiv.org/abs/1512.03385.

[IAVB+17] Alexandru Iosup, Georgios Andreadis, Vincent Van Beek, Matthijs Bijman, Erwin Van Eyk, Mihai Neacsu, Leon Overweel, Sacheendra Talluri, Laurens Versluis, and Maaike Visser. The opendc vision: Towards collaborative datacenter simulation and exploration for everybody. 2017 16th International Symposium on Parallel and Distributed Computing (ISPDC), pages 85–94. IEEE, 2017.

[IBM16] IBM. Openwhisk, 2016. https://openwhisk.apache.org/ Accessed February 28, 2023.

[IHPT20] Nabil El Ioini, David Hästbacka, Claus Pahl, and Davide Taibi. Platforms for serverless at the edge: a review. In *European Conference on Service-Oriented and Cloud Computing*, pages 29–40. Springer, 2020.

[IS03] Jadwiga Indulska and Peter Sutton. Location management in pervasive systems. In *Conferences in Research and Practice in Information Technology Series*, volume 34, pages 143–151. Citeseer, 2003.

[JAA+20] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Zomaya, Schahram Dustdar, and Rajiv Ranjan. Iotsim-edge: A simulation framework for modeling the behavior of internet of things and edge computing environments. *Software: Practice and Experience*, 50(6):844–867, 2020.

[JBP+23] Byeonghui Jeong, Seungyeon Baek, Sihyun Park, Jueun Jeon, and Young-Sik Jeong. Stable and efficient resource management using deep neural network on cloud computing. *Neurocomputing*, 521:99–112, 2023.

[JCSY19] Hongseok Jeon, Chunglae Cho, Seungjae Shin, and Seunghyun Yoon. A cloudsim-extension for simulating distributed functions-as-a-service. 2019 20th International Conference on parallel and distributed computing, applications and technologies (PDCAT), pages 386–391. IEEE, 2019.

[JGJ+18] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. A holistic evaluation of docker containers for interfering microservices. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 33–40. IEEE, 2018.

[JHA+23] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23,

page 443–458, New York, NY, USA, 2023. Association for Computing Machinery.

[JHA+25] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 938–953, New York, NY, USA, 2025. Association for Computing Machinery.

[JNW+19] Devki Nandan Jha, Michael Nee, Zhenyu Wen, Albert Zomaya, and Rajiv Ranjan. Smartdbo: smart docker benchmarking orchestrator for web-application. In *The World Wide Web Conference*, pages 3555–3559, 2019.

[JPBG19] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[Kan22] Subarmaniam Kannan. Synthetic time series data generation for edge analytics. *F1000Research*, 11:67, 2022.

[Kel97] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8(1):33–37, 1997.

[KF22] Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *The Journal of Supercomputing*, pages 1–28, 2022.

[KFT21] Vojdan Kjorveziroski, S Filiposka, and V Trajkovic. Iot serverless computing at the edge: Open issues and research direction. *Computers*, 10:130, 2021.

[KJS19] Pekka Karhula, Jan Janak, and Henning Schulzrinne. Checkpointing and migration of iot edge functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, pages 60–65, 2019.

[KKH23] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access*, 2023.

[KL19] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[Kna18] Knative. Knative, 2018. `https://knative.dev/docs/`. Accessed August 26, 2025.

[KPL+19] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate synthetic datasets. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4551–4560, 2019.

[KSe19] KServe. Kserve, 2019. `https://kserve.github.io/website/`. Accessed August 26, 2025.

[KTIB22] Tahseen Khan, Wenhong Tian, Shashikant Ilager, and Rajkumar Buyya. Workload forecasting and energy state estimation in cloud data centres: Ml-centric approach. *Future Generation Computer Systems*, 128:320–332, 2022.

[KTS21] Raffael Klingler, Nemanja Trifunovic, and Josef Spillner. Beyond@ cloud-function: Powerful code annotations to capture serverless runtime patterns. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, pages 23–28, 2021.

[Kub14] Kubernetes. Kubernetes, 2014. `https://kubernetes.io/`. Accessed August 26, 2025.

[Kub16] Kubeless. Kubeless, 2016. `https://github.com/vmware-archive/kubeless`. Accessed August 26, 2025.

[KY13] Yoshiyuki Kawano and Keiji Yanai. Real-time mobile food recognition system. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–7, 2013.

[KY15] Yoshiyuki Kawano and Keiji Yanai. Foodcam: A real-time food recognition system on a smartphone. *Multimedia Tools and Applications*, 74(14):5263–5287, 2015.

[KYMS20] Oleg Kolosov, Gala Yadgar, Sumit Maheshwari, and Emina Soljanin. Benchmarking in the dark: On the absence of comprehensive edge datasets. 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association, June 2020.

[LdB+24] Vincent Lannurien, Laurent d'Orazio, Olivier Barais, Stéphane Paquelet, and Jalil Boukhobza. Herosim: An allocation and scheduling simulator

for evaluating serverless orchestration policies. *IEEE Internet Computing*, 28(6):8–16, 2024.

[LGC+21] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 2021.

[LGC+22] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, 2022.

[LHOH18] Qiang Liu, Siqi Huang, Johnson Opadere, and Tao Han. An edge network orchestrator for mobile augmented reality. In *Proc. of IEEE INFOCOM'18*, pages 756–764, 2018.

[LKM+22] Xue Li, Peng Kang, Jordan Molone, Wei Wang, and Palden Lama. Kneescale: Efficient resource scaling for serverless computing at the edge. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 180–189. IEEE, 2022.

[LKRL19] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th international workshop on serverless computing*, pages 37–42, 2019.

[LKX+20] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 731–737, 2020.

[LLB21] Francesc Lordan, Daniele Lezzi, and Rosa M Badia. Colony: Parallel functions as a service on the cloud-edge continuum. In *European Conference on Parallel Processing*, pages 269–284. Springer, 2021.

[LLG+22] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, 2022.

[LLR+21] Tero Lähderanta, Teemu Leppänen, Leena Ruha, Lauri Lovén, Erkki Harjula, Mika Ylianttila, Jukka Riekki, and Mikko J. Sillanpää. Edge computing server placement with capacitated location allocation. *Journal of Parallel and Distributed Computing*, 153:130–149, 2021.

[LMC+21] Ibtissam Labriji, Francesca Meneghello, Davide Cecchinato, Stefania Sesia, Eric Perraud, Emilio Calvanese Strinati, and Michele Rossi. Mobility aware and dynamic migration of mec services for the internet of vehicles. *IEEE Transactions on Network and Service Management*, 18(1):570–584, 2021.

[LMFK23] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. Fine-grained performance and cost modeling and optimization for faas applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):180–194, 2023.

[LSd+24] Vincent Lannurien, Camélia Slimani, Laurent d'Orazio, Olivier Barais, Stéphane Paquelet, and Jalil Boukhobza. Herocache: Storage-aware scheduling in heterogeneous serverless edge – the case of ids. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 587–597, 2024.

[LWGS16] He Li, Jing Wu, Yiwen Gao, and Yao Shi. Examining individuals' adoption of healthcare wearable devices: An empirical study from privacy calculus perspective. *International journal of medical informatics*, 88:8–17, 2016.

[LZMW22] Yuanzhe Li, Ao Zhou, Xiao Ma, and Shangguang Wang. Profit-aware edge server placement. *IEEE Internet of Things Journal*, 9(1):55–67, 2022.

[MAJ+21] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent Van Beek, and Alexandru Iosup. Opendc 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pages 455–464. IEEE, 2021.

[Mat08] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009):1–33, 2008.

[MBO+05] Dimitrios Moshou, Cedric Bravo, Roberto Oberti, Jon West, Luigi Bodria, Alastair McCartney, and Herman Ramon. Plant disease detection based on data fusion of hyper-spectral and multi-spectral fluorescence imaging using kohonen maps. *Real-Time Imaging*, 11(2):75–83, 2005.

[MBPMS13] Antoni Martínez-Ballesté, Pablo A Pérez-Martínez, and Agusti Solanas. The pursuit of citizens' privacy: a privacy-aware smart city is possible. *IEEE Communications Magazine*, 51(6):136–141, 2013.

[MBW+04] Dimitrios Moshou, Cédric Bravo, Jonathan West, Stijn Wahlen, Alastair McCartney, and Herman Ramon. Automatic detection of 'yellow rust'in

wheat using reflectance measurements and neural networks. *Computers and electronics in agriculture*, 44(3):173–188, 2004.

[MEBW21] Johannes Manner, Martin Endreß, Sebastian Böhm, and Guido Wirtz. Optimizing cloud function configuration via local simulations. 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pages 168–178. IEEE, 2021.

[MG19] Gabriel Axel Montes and Ben Goertzel. Distributed, decentralized, and democratized artificial intelligence. *Technological Forecasting and Social Change*, 141:354–358, 2019.

[MGDVdC20] Adyson Magalhães Maia, Yacine Ghamri-Doudane, Dario Vieira, and Miguel Franklin de Castro. Dynamic service placement and load distribution in edge computing. In *Proc. of IEEE CNSM'20*, pages 1–9. IEEE, 2020.

[MGG+17] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. 2017 IEEE Fog World Congress, pages 1–6. IEEE, 2017.

[MGST22] Anna-Valentini Michailidou, Anastasios Gounaris, Moysis Symeonides, and Demetris Trihinas. Equality: Quality-aware intensive analytics on the edge. *Information Systems*, 105:101953, 2022.

[Mic14] Microsoft. Azure functions, 2014. `https://docs.microsoft.com/en-us/azure/azure-functions/`. Accessed August 26, 2025.

[Mic25] Microsoft Azure. Use predictive autoscale to scale out before load demands in virtual machine scale sets, 2025. `https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-predictive`. Accessed August 26, 2025.

[MJJ19] Weiqing Min, Shuqiang Jiang, and Ramesh Jain. Food recommendation: Framework, existing solutions, and challenges. *IEEE Transactions on Multimedia*, 22(10):2659–2671, 2019.

[MJL+19] Weiqing Min, Shuqiang Jiang, Linhu Liu, Yong Rui, and Ramesh Jain. A survey on food computing. *ACM Computing Surveys (CSUR)*, 52(5):1–36, 2019.

[MK20] Andras Markus and Attila Kertesz. A survey and taxonomy of simulation environments modelling fog computing. *Simulation Modelling Practice and Theory*, 101:102042, 2020.

[MK21] Nima Mahmoudi and Hamzeh Khazaei. Simfaas: A performance simulator for serverless computing platforms. In Markus Helfert, Donald Ferguson, and Claus Pahl, editors, *Proceedings of the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021, Online Streaming, April 28-30, 2021*, pages 23–33. SCITEPRESS, 2021.

[MKB22] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 2022.

[MMBD20] SeyedMorteza MirhoseiniNejad, Hosein Moazamigoodarzi, Ghada Badawy, and Douglas G Down. Joint data center cooling and workload management: A thermal-aware approach. *Future Generation Computer Systems*, 104:174–186, 2020.

[MMN21] Erfan Farhangi Maleki, Lena Mashayekhy, and Seyed Morteza Nabavinejad. Mobility-aware computation offloading in edge computing using machine learning. *IEEE Transactions on Mobile Computing*, 2021.

[MMR+17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[Mor97] R. Morris. Tcp behavior with many flows. Proceedings 1997 International Conference on Network Protocols, pages 205–211. IEEE, 1997.

[MPV+23] Andrea Morichetta, Thomas Pusztai, Deepak Vij, Víctor Casamayor Pujol, Philipp Raith, Ying Xiong, Stefan Nastic, Schahram Dustdar, and Zhaobo Zhang. Demystifying deep learning in predictive monitoring for cloud-native slos. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 1–11, 2023.

[MR99] Laurent Massoulié and James Roberts. Bandwidth sharing: objectives and algorithms. IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), pages 1395–1403 vol.3. IEEE, 1999.

[NCTD15] Stefan Nastic, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Governing elastic iot cloud systems under uncertainty. In *Proceedings of the IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, pages 131–138. IEEE Computer Society, 2015.

[NMS+22] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatele. Resource utilization aware job scheduling to mitigate performance variability. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 335–345, 2022.

[NPDS20] Dinh C Nguyen, Pubudu N Pathirana, Ming Ding, and Aruna Seneviratne. Blockchain for 5g and beyond networks: A state of the art survey. *Journal of Network and Computer Applications*, page 102693, 2020.

[NPM+21] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216. IEEE, 2021.

[NRdA+20] Diana M Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020.

[NRF+22] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12, 2022.

[Nuc17] Nuclio. Nuclio, 2017. `https://nuclio.io/`. Accessed August 26, 2025.

[OJZR23] Dongyang Ou, Congfeng Jiang, Meilian Zheng, and Yongjian Ren. Container power consumption prediction based on gbrt-pl for edge servers in smart city. *IEEE Internet of Things Journal*, 2023.

[OM19] Randal S. Olson and Jason H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. *Automated Machine Learning: Methods, Systems, Challenges*, pages 151–160, 2019.

[OMS+18] Bhakti Stephan Onggo, Navonil Mustafee, Andi Smart, Angel A Juan, and Owen Molloy. Symbiotic simulation system: Hybrid systems model meets big data analytics. In *2018 Winter Simulation Conference (WSC)*, pages 1358–1369. IEEE, 2018.

[OOW20] David Oakley, Bhakti Stephan Onggo, and Dave Worthington. Symbiotic simulation for the operational management of inpatient beds: model development and validation using $\delta$-method. *Health care management science*, 23:153–169, 2020.

[Ope16] OpenFaaS. Openfaas, 2016. `https://www.openfaas.com/`. Accessed August 26, 2025.

[ORLH17] Yvonne O'Connor, Wendy Rowan, Laura Lynch, and Ciara Heavin. Privacy by design: informed consent and internet of things for smart health. *Procedia computer science*, 113:653–658, 2017.

[OZC18] Tao Ouyang, Zhi Zhou, and Xu Chen. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE Journal on Selected Areas in Communications*, 36(10):2333–2345, 2018.

[Pal22] Jacob Palecek. Improving serverless edge computing for network bound workloads, 2022. Master Thesis. TU Wien.

[PB20a] Tobias Pfandzelter and David Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. IEEE, 2020.

[PB20b] Victor Prokhorenko and M Ali Babar. Architectural resilience in cloud, fog and edge systems: A survey. *IEEE Access*, 8:28078–28095, 2020.

[PCHZ15] Zhibo Pang, Qiang Chen, Weili Han, and Lirong Zheng. Value-centric design of the internet-of-things solution for food supply chain: Value creation, sensor portfolio and information fusion. *Information Systems Frontiers*, 17(2):289–319, 2015.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[PKC19] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642, pages 206–211. IEEE, 2019.

[PLM+10] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 51–58. IEEE, 2010.

[PMP+21a] Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Stefan Nastic, Xiaoning Ding, Deepak Vij, and Ying Xiong. A novel middleware for efficiently implementing complex cloud-native

slos. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 410–420. IEEE, 2021.

[PMP+21b] Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Stefan Nastic, Xiaoning Ding, Deepak Vij, and Ying Xiong. Slo script: A novel language for implementing complex cloud-native elasticity-driven slos. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 21–31. IEEE, 2021.

[PN25] Thomas Pusztai and Stefan Nastic. Chunkfunc: Dynamic slo-aware configuration of serverless functions. *IEEE Transactions on Parallel and Distributed Systems*, 36(6):1237–1252, 2025.

[PNM+22] Thomas Pusztai, Stefan Nastic, Andrea Morichetta, Víctor Casamayor Pujol, Philipp Raith, Schahram Dustdar, Deepak Vij, Ying Xiong, and Zhaobo Zhang. Polaris scheduler: Slo- and topology-aware microservices scheduling at the edge. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 61–70, 2022.

[PSP+21] Panos Patros, Josef Spillner, Alessandro V. Papadopoulos, Blesson Varghese, Omer Rana, and Schahram Dustdar. Toward sustainable serverless computing. *IEEE Internet Computing*, 25(6):42–50, 2021.

[PTI+23] Chandrasen Pandey, Vaibhav Tiwari, Agbotiname Lucky Imoize, Chun-Ta Li, Cheng-Chi Lee, and Diptendu Sinha Roy. 5gt-gan: Enhancing data augmentation for 5g-enabled mobile edge computing in smart cities. *IEEE Access*, 11:120983–120996, 2023.

[PTR+23] Chandrasen Pandey, Vaibhav Tiwari, Rajkumar Singh Rathore, Rutvij H Jhaveri, Diptendu Sinha Roy, and Shitharth Selvarajan. Resource-efficient synthetic data generation for performance evaluation in mobile edge computing over 5g networks. *IEEE Open Journal of the Communications Society*, 4:1866–1878, 2023.

[PWHH19] Stephen Pasteris, Shiqiang Wang, Mark Herbster, and Ting He. Service placement with provable guarantees in heterogeneous edge computing systems. In *Proc. of IEEE INFOCOM'19*, pages 514–522, 2019.

[PZCG14] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. *Transactions on emerging telecommunications technologies*, 25(1):81–93, 2014.

[RAD18] Thomas Rausch, Cosmin Avasalcai, and Schahram Dustdar. Portable energy-aware cluster-based edge computers. 2018 IEEE/ACM Symposium on Edge Computing (SEC), pages 260–272. IEEE, 2018.

[Rai21] P. A. Raith. Container scheduling on heterogeneous clusters using machine learning-based workload characterization, 2021. Master Thesis, TU Wien.

[Rau21] Thomas Rausch. *A distributed compute fabric for edge intelligence*. PhD thesis, TU Wien, 2021.

[RB19] Maria A. Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719, 2019.

[RB20] Kaustabha Ray and Ansuman Banerjee. Trace-driven modeling and verification of a mobility-aware service allocation and migration policy for mobile edge computing. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 310–317, 2020.

[RC18] David Christian Rose and Jason Chilvers. Agriculture 4.0: Broadening responsible innovation in an era of smart farming. *Frontiers in Sustainable Food Systems*, 2:87, 2018.

[RCLN20] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174, 2020.

[RCP23] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Serverless functions in the cloud-edge continuum: Challenges and opportunities. In *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 321–328. IEEE, 2023.

[RCPN23] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. *IEEE Transactions on Cloud Computing*, 11(2):1807–1821, 2023.

[RD19] Thomas Rausch and Schahram Dustdar. Edge intelligence: The convergence of humans, things, and ai. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 86–96. IEEE, 2019.

[RD21] Philipp Raith and Schahram Dustdar. Edge intelligence as a service. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 252–262, 2021.

[RGLT19] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. *IEEE International Conference on Cloud Computing, CLOUD*, 2019-July:33–40, 7 2019.

[RH10] George F. Riley and Thomas R. Henderson. The ns-3 network simulator. *Modeling and Tools for Network Simulation*, pages 15–34, 2010.

[RHD+23] Gourav Rattihalli, Ninad Hogade, Aditya Dhakal, Eitan Frachtenberg, Rolando Pablo Hong Enriquex, Pedro Bruel, Alok Mishra, and Dejan Milojicic. Fine-grained heterogeneous execution framework with energy aware scheduling. *IEEE International Conference on Cloud Computing*, 2023.

[RHHP22] Sashko Ristov, Mika Hautz, Christian Hollaus, and Radu Prodan. Simless: Simulate serverless workflows and their twins and siblings in federated faas. Proceedings of the 13th Symposium on Cloud Computing, page 323–339, New York, NY, USA, 2022. Association for Computing Machinery.

[RHM+19] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19), Renton, WA, July 2019. USENIX Association.

[RHM20] Thomas Rausch, Waldemar Hummer, and Vinod Muthusamy. Pipesim: Trace-driven simulation of large-scale ai operations platforms. *arXiv preprint arXiv:2006.12587*, 2020.

[RHS+21] Thomas Rausch, Waldemar Hummer, Christian Stippel, Silvio Vasiljevic, Carmine Elvezio, Schahram Dustdar, and Katharina Krösl. Towards a platform for smart city-scale cognitive assistance applications. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 330–335, 2021.

[RKK+19] Kasim Rehman, Orthodoxos Kipouridis, Stamatis Karnouskos, Oliver Frendo, Helge Dickel, Jonas Lipps, and Nemrude Verzano. A cloud-based development environment using hla and kubernetes for the co-simulation of a corporate electric vehicle fleet. In *2019 IEEE/SICE International Symposium on System Integration (SII)*, pages 47–54. IEEE, 2019.

[RLF+20] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association, June 2020.

[RNC19] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 329–338. IEEE, 2019.

[RND18] Thomas Rausch, Stefan Nastic, and Schahram Dustdar. Emma: Distributed qos-aware mqtt middleware for edge computing applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–197. IEEE, 2018.

[RND23] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.

[RND24] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Simuscale: Optimizing parameters for autoscaling of serverless edge functions through co-simulation. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 305–315, 2024.

[Rob51] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[RPGT22] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 46–60, 2022.

[RRD21] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021.

[RRD+22] Philipp Raith, Thomas Rausch, Schahram Dustdar, Fabiana Rossi, Valeria Cardellini, and Rajiv Ranjan. Mobility-aware serverless function adaptations across the edge-cloud continuum. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 123–132. IEEE, 2022.

[RRD+24] Philipp Raith, Gourav Rattihalli, Aditya Dhakal, Sai Rahul Chalamalasetti, Dejan Milojicic, Eitan Frachtenberg, Stefan Nastic, and Schahram Dustdar. Opportunistic energy-aware scheduling for container orchestration platforms using graph neural networks. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 299–306, 2024.

[RRFD23] Philipp Raith, Thomas Rausch, Alireza Furutanpey, and Schahram Dustdar. faas-sim: A trace-driven simulation framework for serverless edge computing platforms. *Software: Practice and Experience*, 53(12):2327–2361, 2023.

[RRP+22] Philipp Raith, Thomas Rausch, Paul Prüller, Alireza Furutanpey, and Schahram Dustdar. An end-to-end framework for benchmarking edge-cloud cluster management techniques. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–28, 2022.

[RRPD19] Thomas Rausch, Philipp Raith, Padmanabhan Pillai, and Schahram Dustdar. A system for operating energy-aware cloudlets. In *Proceedings*

*of the 4th ACM/IEEE Symposium on Edge Computing*, pages 307–309, 2019.

[RSK23] Seyed Hamed Rastegar, Hossein Shafiei, and Ahmad Khonsari. Enex: An energy-aware execution scheduler for serverless computing. *IEEE Transactions on Industrial Informatics*, 2023.

[SAP20] Eleni Symeonaki, Konstantinos Arvanitis, and Dimitrios Piromalis. A context-aware middleware cloud approach for integrating precision farming facilities into the iot toward agriculture 4.0. *Applied Sciences*, 10(3):813, 2020.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[SBG+22] Joel Scheuner, Marcus Bertilsson, Oskar Gronqvist, Henrik Tao, Henrik Lagergren, Jan-Philipp Steghöfer, and Philipp Leitner. Triggerbench: A performance benchmark for serverless function triggers (short paper). In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2022.

[SD16] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[SFEG21] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch Glitho. Auto-adaptive fault prediction system for edge cloud environments in the presence of concept drift. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 217–223. IEEE, 2021.

[SG19] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th international workshop on serverless computing*, pages 19–24, 2019.

[SGvK+22] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. Why is it not solved yet? challenges for production-ready autoscaling. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 105–115, New York, NY, USA, 2022. Association for Computing Machinery.

[Sha23] Prateek Sharma. Challenges and opportunities in sustainable serverless computing. *SIGENERGY Energy Inform. Rev.*, 3(3):53–58, October 2023.

[SJC+22] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 17–25. IEEE, 2022.

[SK22] Dilshad Hassan Sallo and Gabor Kecskemeti. Towards generating realistic trace for simulating functions-as-a-service. In Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci, editors, *Euro-Par 2021: Parallel Processing Workshops*, volume vol 13098, pages 428–439, Cham, 2022. Springer International Publishing.

[SKM22] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 54(11s):1–32, 2022.

[SOE17] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. EdgeCloudSim: An environment for performance evaluation of edge computing systems. 2017 Second International Conference on Fog and Mobile Edge Computing, pages 39–44. IEEE, 2017.

[SP20] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[SPC+14] Agusti Solanas, Constantinos Patsakis, Mauro Conti, Ioannis S Vlachos, Victoria Ramos, Francisco Falcone, Octavian Postolache, Pablo A Pérez-Martínez, Roberto Di Pietro, Despina N Perrea, et al. Smart health: A context-aware health paradigm within smart cities. *IEEE Communications Magazine*, 52(8):74–81, 2014.

[SvdZD+15] Fatjon Seraj, Berend Jan van der Zwaag, Arta Dilo, Tamara Luarasi, and Paul Havinga. Roads: A road pavement monitoring system for anomaly detection using smart phones. In *Big data analytics in the social and ubiquitous context*, pages 128–146. Springer, 2015.

[SWC+20] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[SZML19] Xiaoke Sun, Junhui Zhao, Xiaoting Ma, and Qiuping Li. Enhancing the user experience in vehicular edge computing networks: An adaptive resource allocation approach. *IEEE Access*, 7:161074–161087, 2019.

[tax18] 2018 yellow taxi trip data, 2018. Available at `https://data.cityofnewyork.us/Transportation/2018-Yellow-Taxi-Trip-Data/t29m-gskq`.

[TC22] Shreshth Tuli and Giuliano Casale. Optimizing the performance of fog computing environments using ai and co-simulation. In *Companion of the*

*2022 ACM/SPEC International Conference on Performance Engineering*, pages 25–28, 2022.

[TCJ23] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. Cilp: Co-simulation-based imitation learner for dynamic resource provisioning in cloud computing environments. *IEEE Transactions on Network and Service Management*, 20(4):4448–4460, 2023.

[TDFS21] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.

[TG20] Salman Taherizadeh and Marko Grobelnik. Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software*, 140:102734, 2020.

[TGW+20] Guoming Tang, Deke Guo, Kui Wu, Fang Liu, and Yudong Qin. Qos guaranteed edge cloud resource provisioning for vehicle fleets. *IEEE Transactions on Vehicular Technology*, 69(6):5889–5900, 2020.

[TGX+22] Shreshth Tuli, Sukhpal Singh Gill, Minxian Xu, Peter Garraghan, Rami Bahsoon, Schahram Dustdar, Rizos Sakellariou, Omer Rana, Rajkumar Buyya, Giuliano Casale, et al. Hunter: Ai based holistic resource management for sustainable cloud computing. *Journal of Systems and Software*, 184:111124, 2022.

[TLG16] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *Proc. of IEEE INFOCOM'16*, 2016.

[TLX+23] Da Sun Handason Tam, Yang Liu, Huanle Xu, Siyue Xie, and Wing Cheong Lau. Pert-gnn: Latency prediction for microservice-based cloud-native applications via graph neural networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 2155–2165, New York, NY, USA, 2023. Association for Computing Machinery.

[TMG+21] Achilleas Tzenetopoulos, Charalampos Marantos, Giannos Gavrielides, Sotirios Xydis, and Dimitrios Soudris. Fade: Faas-inspired application decomposition and energy-aware function placement on the edge. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, pages 7–10, 2021.

[TPPM23] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. Graph clustering with graph neural networks. *Journal of Machine Learning Research*, 24(127):1–21, 2023.

[TPS+22] Shreshth Tuli, Shivananda R. Poojara, Satish N. Srirama, Giuliano Casale, and Nicholas R. Jennings. Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):101–116, 2022.

[TPTE21] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10, 2021.

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[VTK22] Dinh-Dai Vu, Minh-Ngoc Tran, and Younghan Kim. Predictive hybrid autoscaling for containerized applications. *IEEE Access*, 10:109768–109778, 2022.

[WAES21] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. Lass: running latency sensitive serverless computations at the edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–251, 2021.

[WAP+14] Torsten Wilde, Axel Auweter, Michael K. Patterson, Hayk Shoukourian, Herbert Huber, Arndt Bode, Detlef Labrenz, and Carlo Cavazzoni. Dwpe, a new data center energy-efficiency metric bridging the gap between infrastructure and workload. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 893–901, 2014.

[WGZ+21] Shangguang Wang, Yan Guo, Ning Zhang, Peng Yang, Ao Zhou, and Xuemin Shen. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951, 2021.

[WHC+16] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994, 2016.

[WHFG92] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[WHL+20] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of edge computing and deep learning:

A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[WJM+21] Daniel C. Wilson, Siddhartha Jana, Aniruddha Marathe, Stephanie Brink, Christopher M. Cantalupo, Diana R. Guttman, Brad Geltz, Lowren H. Lawson, Asma H. Al-rawi, Ali Mohammad, Fuat Keceli, Federico Ardanaz, Jonathan M. Eastep, and Ayse K. Coskun. Introducing application awareness into a unified power management stack. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 320–329, 2021.

[WKB+19] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: Portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 236–249, 2019.

[WLL+21] Chenyang Wang, Ruibin Li, Wenkai Li, Chao Qiu, and Xiaofei Wang. Simedgeintel: A open-source simulation platform for resource management in edge intelligence. *Journal of Systems Architecture*, 115:102016, 2021.

[WNL19] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[WPC+21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[WS22] Sebastian Werner and Trever Schirmer. Hardless: A generalized serverless compute architecture for hardware processing accelerators. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 79–84. IEEE, 2022.

[WTC+16] Bing-Fei Wu, Wan-Ju Tseng, Yung-Shin Chen, Shih-Jhe Yao, and Po-Ju Chang. An intelligent self-checkout system for smart retail. In *2016 International Conference on System Science and Engineering (ICSSE)*, pages 1–4. IEEE, 2016.

[WWV17] Zhenglin Wang, Kerry B Walsh, and Brijesh Verma. On-tree mango fruit size estimation using rgb-d images. *Sensors*, 17(12):2738, 2017.

[WY20] Tzu-Tsung Wong and Po-Yang Yeh. Reliable accuracy estimates from k-fold cross validation. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1586–1594, 2020.

[XHJ+15] Ruihan Xu, Luis Herranz, Shuqiang Jiang, Shuang Wang, Xinhang Song, and Ramesh Jain. Geolocalized modeling for dish recognition. *IEEE transactions on multimedia*, 17(8):1187–1199, 2015.

[XSXH18] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *Proc. of IEEE/ACM SEC'18*, pages 373–377, 2018.

[XTQ+21] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F. Richard Yu, and Tao Huang. When serverless computing meets edge computing: Architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5):126–133, 2021.

[Yan17] Shusen Yang. Iot stream processing and analytics in the fog. *IEEE Communications Magazine*, 55(8):21–27, 2017.

[YCYO23] Xuyi Yao, Ningjiang Chen, Xuemei Yuan, and Pingjie Ou. Performance optimization of serverless edge computing function offloading based on deep reinforcement learning. *Future Generation Computer Systems*, 139:74–86, 2023.

[YHY+17] Longqi Yang, Cheng-Kang Hsieh, Hongjian Yang, John P Pollak, Nicola Dell, Serge Belongie, Curtis Cole, and Deborah Estrin. Yum-me: a personalized nutrient-based meal recommender system. *ACM Transactions on Information Systems (TOIS)*, 36(1):1–31, 2017.

[YLZ+20] Quan Yuan, Jinglin Li, Haibo Zhou, Tao Lin, Guiyang Luo, and Xuemin Shen. A joint service migration and mobility optimization approach for vehicular edge computing. *IEEE Transactions on Vehicular Technology*, 69(8):9041–9052, 2020.

[YSC+20] Xing Yang, Lei Shu, Jianing Chen, Mohamed Amine Ferrag, Jun Wu, Edmond Nurellari, and Kai Huang. A survey on smart agriculture: Development modes, technologies, and security and privacy challenges. *IEEE/CAA Journal of Automatica Sinica*, 8(2):273–302, 2020.

[YWCW22] Lei Yang, Fulin Wen, Jiannong Cao, and Zhenyu Wang. Edgetb: A hybrid testbed for distributed machine learning at the edge with high fidelity. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2540–2553, 2022.

[ZAL+17] Zhou Zhou, Jemal H Abawajy, Fangmin Li, Zhigang Hu, Morshed U Chowdhury, Abdulhameed Alelaiwi, and Keqin Li. Fine-grained energy consumption model of servers based on task characteristics in cloud data center. *IEEE access*, 6:27080–27090, 2017.

[Zap20] Zappa. zappa, 2020. `https://github.com/zappa/Zappa`. Accessed August 26, 2025.

[ZCC+20] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

[ZCD+18] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[ZCL+19] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[ZCZ+18] Jiale Zhang, Bing Chen, Yanchao Zhao, Xiang Cheng, and Feng Hu. Data security and privacy-preserving in edge computing paradigm: Survey and open issues. *IEEE Access*, 6:18209–18237, 2018.

[ZFPS20] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 328–343, 2020.

[ZGD19] Yanqi Zhang, Yu Gan, and Christina Delimitrou. uqsim: Scalable and validated simulation of cloud microservices. *arXiv preprint arXiv:1911.02122*, 2019.

[ZIH+13] Jia Zhang, Bob Iannucci, Mark Hennessy, Kaushik Gopal, Sean Xiao, Sumeet Kumar, David Pfeffer, Basmah Aljedia, Yuan Ren, Martin Griss, et al. Sensor data as a service–a federated platform for mobile data-centric service development and sharing. In *2013 IEEE International Conference on Services Computing*, pages 446–453. IEEE, 2013.

[ZLQZ13] Xiao Zhang, Jian-Jun Lu, Xiao Qin, and Xiao-Nan Zhao. A high-level energy consumption model for heterogeneous data centers. *Simulation Modelling Practice and Theory*, 39:41–55, 2013.

[ZPX+18] Chao Zhu, Giancarlo Pastor, Yu Xiao, Yong Li, and Antti Ylae-Jaeaeski. Fog following me: Latency and quality balanced task allocation in vehicular fog computing. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2018.

[ZWL+19] Xingzhou Zhang, Yifan Wang, Sidi Lu, Liangkai Liu, Weisong Shi, et al. Openei: An open framework for edge intelligence. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1840–1851. IEEE, 2019.

[ZXR+22] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Computing Surveys*, 54(10s), sep 2022.