Informatics

A Distributed Compute Fabric for Edge Intelligence

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Thomas Rausch, BSc
Matrikelnummer 00726439

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar

Diese Dissertation haben begutachtet:

Weisong Shi

Ming Zhao

Wien, 5. Mai 2021

Thomas Rausch

TU WIEN Informatics

A Distributed Compute Fabric for Edge Intelligence

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Thomas Rausch, BSc
Registration Number 00726439

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar

The dissertation has been reviewed by:

Weisong Shi

Ming Zhao

Vienna, 5th May, 2021

Thomas Rausch

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Thomas Rausch, 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.

Wien, 5. Mai 2021

Thomas Rausch

Acknowledgements

This thesis is the culmination of over fours years of research, tinkering, personal growth, and wonderful collaborations. I have lots of things to be grateful for, have been immensely privileged to be where I am today, and I owe thanks to numerous people whose involvements in this thesis I want to acknowledge. First I want to thank my supervisor Schahram Dustdar, who gave me the opportunity to pursue my PhD in his group. He has given me the two things I needed most to grow as a researcher and person: the freedom to pursue my own path, and the trust that this path will lead to great outcomes. I owe thanks to Waldemar Hummer, whose advice has helped me get through the many stages of the PhD. Waldemar has been a teacher, mentor, colleague, and friend, and I'm extremely grateful to him. His tenacity lead to an internship at IBM Research, where we, together with Vinod Muthusamy, developed key ideas of this thesis. I also want to thank Mahadev Satyanarayanan from Carnegie Mellon University, and Padmanabhan Pillai from Intel Labs, who made possible my research stay at Satya's edge computing lab at CMU. I owe thanks to my two thesis reviewers Ming Zhao and Weisong Shi, who I've also enjoyed stimulating conversations with at Edge Computing conferences. I also want to thank Wolfgang Kastner and Uwe Zdun for agreeing to be in my proficiency evaluation committe, giving me valuable feedback in the earlier stages of my research. I've had many colleagues at the Distributed Systems Group who have helped me directly or indirectly with my thesis. Renate, Christine, and Margret have been immensely helpful with administrative issues, and I'm very grateful for their patience. Alexander Knoll, who has become my friend, was always eager to help with acquiring hardware for testbeds, build compute clusters, and provide cute videos of cats. Moreover, I wish to thank all my students who have contributed to the research presented in this thesis, and who helped me grow as a teacher: Andreas Bruckner, Manuel Geier, Cynthia Marcelino, Jacob Palecek, Philipp Raith, Alexander Rashed, David Schneiderbauer, and Silvio Vasiljevic. Finally, I'm immensely grateful for Katharina Krösl, who has been a fixed point in the tumultuous past decade of my life. Her never ending patience, tolerance, and helpful character have significantly contributed to my life and my career. Through the many fruitful discussions, we have found exciting topics at the intersection of our research, that have opened up new avenues I hope to continue to pursue together.

The research presented in this thesis was funded by TU Wien research funds, as well as the Austrian Federal Ministry of Science through the Austrian infrastructure program (HRSM 2016) as part of the CPS/IoT Ecosystem project.

Kurzfassung

Edge Intelligence ist ein post-Cloud Computing Paradigma, und die Konsequenz des letzten Jahrzehnts an Entwicklungen in den Bereichen der Künstlicher Intelligenz (KI), Internet der Dinge (IoT), und Human Augmentation. An der Schnittstelle dieser Domänen entstehen Anwendungsszenarien mit herausfordernden Anforderungen, etwa den Echtzeitzugriff auf Sensordaten aus der Umgebung, KI-Modell-Inferenz mit geringer Latenz, oder den sicheren Zugriff auf Daten aus privaten Edge Netzwerken um KI-Modelle zu trainieren. Diese Anforderungen stehen in klarem Widerspruch zum zentralisierten Cloud Computing Paradigma, und haben weitreichende Auswirkungen auf das Design der unterstützenden Computersysteme. Edge Intelligence erfordert neuartige Systeme die explizit für die Charakteristika von KI Anwendungen und Edge Computing Umgebungen entwickelt sind. Diese Systeme verweben mit den entsprechenden Abstraktionen, Cloud und Edge Ressourcen zu einem neuartigen Rechnerverbund, dem "Distributed Compute Fabric". Die Ziele der vorliegenden Arbeit sind die Untersuchung der mit diesen neuen Systemen verbundenen Herausforderungen, und die Entwicklung und Evaluierung von Prototypen um die Anwendbarkeit zu demonstrieren.

Um das Konzept der intelligenten Edge Netzwerke, oder "Edge Intelligence", zu untermauern, analysieren wir zuerst aktuelle Trends in Human Augmentation, Edge Computing und KI. Wir erarbeiten Szenarien in denen dezentrale IT-Ressourcen einen Rechnerverbund bilden, der für die verteilte Ausführung von Anwendungen dienlich sein kann. Das Analogon zum Cloud Server Computer ist der "multi-purpose Edge Computer Cluster", welches die Infrastruktureinheit für Edge Intelligence bildet. Anhand von Experimenten mit einem Prototypen den wir entwickelt haben, verdeutlichen wir die Herausforderungen an Orchestrierungsmechanismen in solchen Systemen. Wir entwickeln neue Evaluierungsmethodologien für Edge Computing. Insbesondere entwickeln wir anhand einer Analyse neuartiger Edge Systeme ein Framework um synthetische aber plausible Cluster und Infrastruktur-Konfigurationen zu generieren, die als Input für einen Simulator verwendet werden. Um verteilte Infrastruktureinheiten zu einem Verbund zu verweben, entwickeln wir zwei orthogonale Systeme: eine elastisch skalierende Message-Oriented Middleware, und eine Serverless Edge Computing Plattform. Von einem zentralen Deployment in der Cloud, diffundieren Nachrichten-Broker nach Bedarf und Ressourcenverfügbarkeit in Edge Netzwerke. Das System optimiert kontinuierlich Kommunikationslatenzen in dem es die Distanz zwischen Clients und Broker überwacht und das Netzwerk entsprechend neu konfiguriert. Unsere Serverless Plattform erweitert existierende Systeme mit einem neuartigen Scheduler, der Zielkonflikte zwischen Datentransfer und Codemobilität zur Laufzeit auflöst, und spezielle Anforderungen von AI Anwendungen analysiert und entsprechend zu spezialisierten IT-Ressourcen, wie etwa GPUs, zordnet. Weiters präsentieren wir eine Methode für das automatische Feineinstellen von Scheduler-Parameter, um Betriebsziele der Infrastruktur zu optimieren.

Abstract

Edge intelligence is a post-cloud computing paradigm, and a consequence of the past decade of developments in Artificial Intelligence (AI), Internet of Things (IoT), and human augmentation. At the intersection of these domains, new applications have emerged that require real-time access to sensor data from the environment, low-latency AI model inferencing, or access to data isolated in edge networks for training AI models, all while operating in highly dynamic and heterogeneous computing environments. These requirements have profound implications on the scale and design of supporting computing platforms that are clearly at odds with the centralized nature of cloud computing. Instead, edge intelligence necessitates a new operational layer that is designed for the characteristics of AI and edge computing systems. This layer weaves both cloud and federated edge resources together using appropriate platform abstractions to form a distributed compute fabric. The main goals of this thesis are to examine the associated challenges, and to provide evidence for the efficacy of the idea. To further develop the concept of Edge Intelligence, we first discuss emerging technology trends at the intersection of edge computing and AI. We then examine scenarios where distributed computing resources can be federated and served as a utility, and argue that multi-purpose edge computer clusters will be a fundamental infrastructural component. Through experiments on prototypes we have built, we highlight the challenges faced by operational mechanisms such as load balancers in these environments. We extend the body of evaluation methods for edge computing, which is, compared to cloud computing research, still underdeveloped. Most notably, we present a toolkit to generate synthetic infrastructure configurations and network topologies that are grounded in our examination of existing edge systems, and serve as input for a trace-driven simulator we have built. To create the distributed computing fabric, we develop two orthogonal systems: an elastic message-oriented middleware, and a serverless edge computing platform. From a static centralized deployment in the cloud, we bootstrap a network of brokers that diffuse to the edge based on resource availability, and the number of clients and their proximity to edge resources. The system continuously optimizes communication latencies by monitoring client–broker proximity, and reconfiguring connections as necessary. Our serverless platform builds on existing container orchestration systems. The core is a custom scheduler that can make tradeoffs between data and computation movement, and is aware of workload heterogeneity and device capabilities such as GPUs. Furthermore, we demonstrate a method to automatically fine-tune scheduler parameters and optimize high-level operational goals.

Contents

Publications

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

- Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021
- Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *3rd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'20, 2020
- Thomas Rausch, Waldermar Hummer, and Vinod Muthusamy. An experimentation and analytics framework for large-scale AI operations platforms. In *2020 USENIX Conference on Operational Machine Learning*, OpML'20, 2020
- Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, and Kaoutar El Maghraoui. Modelops: Cloud-based lifecycle management for reliable and trusted AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019
- Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'19, 2019
- Thomas Rausch and Schahram Dustdar. Edge intelligence: The convergence of humans, things, and AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019
- 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'18, pages 191–197. IEEE, 2018
- Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic message-oriented middleware for the internet of things. *IEEE Cloud Computing*, 5(2):17–25, 2018

- 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*, SEC'19, pages 307–309, 2019
- Thomas Rausch, Cosmin Avasalcai, and Schahram Dustdar. Portable energy-aware cluster-based edge computers. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC'18, pages 260–272, 2018
- S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017

Furthermore, selected topics were investigated together with students, reported in the following Master's theses under my supervision:

- Andreas Bruckner. Self-adaptive distributed MQTT middleware for edge computing applications. Master's thesis, TU Wien, 2021
- Cynthia Marcelino. Locality-aware data management for serverless edge computing. Master's thesis, TU Wien, 2021
- Philipp Raith. Container scheduling on heterogeneous clusters using machine learning-based workload characterization. Master's thesis, TU Wien, 2021
- Alexander Rashed. Optimized container scheduling for serverless edge computing. Master's thesis, TU Wien, 2020
- David Schneiderbauer. Predictive analytics for smart homes using serverless edge computing. Master's thesis, TU Wien, 2019
- Manuel Geier. Enabling mobility and message delivery guarantees in distributed MQTT networks. Master's thesis, TU Wien, 2019

CHAPTER 1

Introduction

1.1 Motivation

The development of **artificial intelligence (AI)** has taken spectacular leaps over the past decade. Even in problem areas long thought to be unattainable without human reasoning and intuition, AI has achieved superhuman capabilities, as impressively demonstrated by Jeopardy winning IBM Watson [FBCC$^+$10] or Google DeepMind's AlphaGo [SHM$^+$16] and AlphaZero [SHS$^+$18]. The increase in data availability, computing power, and specialized AI hardware, as well as the advancement of machine learning (ML) techniques, has moved us into the fast-lane towards a society that is shaped by AI in all its aspects. Although researchers and thinkers seem divided into AI optimists and pessimists, one thing seems clear. The optimal strategy for us humans, for whatever future awaits us with the development of AI, is that we continue to foster a close partnership with AI. This partnership seems particularly important as the number of Internet connected sensors, devices, and autonomous agents that can sense and manipulate our physical surroundings continues to grow. Cisco reports that the number of networked devices per person will grow from 2.4 in 2018, to 3.6 by 2023, resulting in 28 billion Internet connected devices [Cis21]. Moreover, they estimated that by 2021, 850 ZB of data will be generated by all people, machines, and things [Cis19]. With the appropriate human–computer interfaces, and a cyber-physical fabric that permeates our environment, together with the unprecedented amount of computing power and digitally persisted knowledge, there is an opportunity for human cognition to further evolve beyond its current limitations.

Edge computing [GLME$^+$15, SD16, Sat17] will play a fundamental role in this development. The premise of edge computing is that the centralized model of cloud computing is impractical to support emerging Internet of Things (IoT), smart city, and mobile computing applications. First, there physical and architectural limitations to what cloud-based solutions can deliver in terms of latency and bandwidth [SBCD09, Sat17]. For example, in [RHS$^+$21] we showed that, even in simple cases such as detecting objects in a video feed using deep neural networks, compared to edge computing solutions, current cloud-based solutions incur between 400 and 800 milliseconds of additional latency which is impractical for real-time applications. In [RRD21] we showed that, moving data for training ML models from the edge to the cloud causes serious network scalability issues, and decreases throughput by an order of magnitude. Second, there are legitimate concerns about privacy and trust [SD16], particularly as we become more interconnected with, and dependent on, personalized AI. Moreover, as we move towards artificial general intelligence (AGI), previously isolated AI programs will become increasingly interconnected and begin to collaboratively solve increasingly complex tasks [MG18]. While we can bootstrap the development of these systems using cloud-based solutions, a decentralized model where programs make use of geographically dispersed edge computing resources is likely to follow. Computational intelligence and data will gradually be pushed from the centralized data centers closer to the edge of the network.

Edge intelligence is the fusion of edge computing and artificial intelligence [ZCL+19, ZWL+19, RD19]. It is the ability of edge computing systems to: support AI-based applications, provide efficient access to data and computation at the edge to train and serve ML models, and use AI to optimize their own operation at runtime. In this new paradigm that sits at the intersection of human augmentation, IoT, AI, and edge computing, new families of applications emerge that require real-time, location-based data about environments at different levels of fidelity; and appropriate computing resources to both process and store data close to where they are generated. We hypothesize that such applications will be supported by a computing model much like modern enterprise applications are supported by cloud computing systems today, where a platform provider takes care of efficiently operating applications on the underlying utility computing infrastructure. Researchers have put forward several candidates for this infrastructure [PDT18], such as geo-distributed cloudlets [SBCD09], single-board computer (SBC) clusters [EPR+17, BJP+20], or even clusters of mobile devices [HAHZ15]. Although there is currently no consensus on what the dominant infrastructure architecture will be, any one of them will be characterized by extreme heterogeneity in terms of computing capabilities, and the physical proximity to data producers and consumers. This has profound implications on the scale and design of supporting computing platforms, and necessitates a new operational layer that is expressly designed for the characteristics of edge intelligence. This layer weaves both cloud and federated edge resources together, and provides appropriate platform abstractions, to form a **distributed compute fabric**.

1.2 Problem statement

A **distributed compute fabric for edge intelligence** is an edge computing system that (1) federates geographically dispersed and computationally diverse computing infrastructure to form a single transparent *computing substrate*, (2) provides a computing platform tailored to edge AI workloads, which abstracts the increased complexity of this infrastructure by elevating concepts from edge computing and AI into the system's operational mechanisms (such as data access, scheduling, or autoscaling), and (3) provides a sensing and communication middleware to enable real-time data exchange and IoT integration. The main goal of this thesis is to examine the challenges of building, operating, and evaluating such a distributed compute fabric for edge intelligence, and to provide evidence for the efficacy of the idea. Moreover, we want to examine to which extent existing systems abstractions from cloud computing can be re-used or extended to solve the associated challenges.

1.2.1 Research questions

The research presented in this thesis is driven by three overarching research questions:

RQ1. Which are appropriate infrastructure architectures that enable a distributed compute fabric for edge intelligence?

Cloud computing architecture, and the computing infrastructure used for cloud computing, is extremely well understood. Server computers and high-performance networking appliances are consolidated into massive data centers, abstracted by hardware virtualization using Hypervisors, and then managed and made accessible as a utility by virtual infrastructure management software [EPM13]. Edge resources are much more diverse and provide a variety of computational platforms, such as GPUs or AI accelerators, to support the equally diverse range of use cases [SGL19]. At the present time, there is no consensus on what edge computing infrastructure is precisely. The following three examples illustrate the disparity:

1. In Mobile Edge Computing (MEC) scenarios, telecommunication companies operate small data centers in logical proximity to LTE/5G base stations, thereby allowing providers to deliver cloud services directly at the *last mile* [MB17].

2. In a smart city scenario, a network of sensing nodes with limited compute capabilities (such as an Array of Things [CBSG17] or similar product), may be complemented by an equally distributed, but lower density network of cloudlets [SBCD09], which are small-scale compute clusters with diverse computing capabilities.

3. Many envision a more generic, but strictly three-tiered architecture where: tier one is the cloud acting as a centralized coordinator and configurator, tier two is the *Fog* [SSBL16] (a metaphor for a cloud closer to the ground) that provides limited compute and storage closer to tier one, and tier one is made up of IoT sensors and actuators, or edge devices such as mobile phones [SGL19].

We hypothesize that, to enable edge intelligence systems at scale, there will not be a one-size-fits-all model. Instead, geographically dispersed and heterogeneous edge computers will be federated with cloud resources to form a multi-purpose *computing substrate*. Novel middleware platforms and operational mechanisms will weave edge and cloud infrastructure to a single distributed compute fabric. However, it is currently unclear whether (a) a utility computing model like cloud computing works for edge computing applications, and (b) which infrastructure architecture would be the best candidate for implementing such a model to support a wide range of use cases.

RQ2. Which aspects of edge intelligence systems should be abstracted as first-class citizens into platforms that manage such systems?

The characteristics of edge intelligence systems significantly increase the complexity of mechanisms required to manage these systems, when compared to cloud computing. Consider key operational mechanisms such as virtualization, auto scaling, and load balancing. These concepts are so fundamental to the cloud computing model that most cloud platforms provide them as native platform functions. This is possible because cloud computing provides concepts such as virtual machine specifications, SLAs, data center regions, etc., as platform abstractions. These abstractions in turn allow platform operators to develop complex operational mechanisms, such as auto scaling, VM consolidation, or energy-aware machine provisioning, that work transparently to the platform user.

In edge computing, concepts such as device proximity, data locality, or energy consumption play a critical role in the effectiveness of these mechanisms, and may therefore have a similar level of significance. Elevating them to first-class citizens may allow us to develop reusable operational mechanisms that work for variety of edge applications, rather than something ad-hoc tailored to a specific application. Examples derived from the presented scenarios include: elastically expanding a broker network to the edge [RND18], data-locality-aware resource allocation [RRD21], or energy-aware load balancing [RAD18].

RQ3. What are appropriate methods for developing and evaluating edge computing systems to allow more generalizable conclusions?

In cloud computing, data sets like Google's Borg cluster traces [RTG+12] are used extensively by researchers to evaluate cloud operations mechanisms, because they provide real cluster configurations (e.g., density and type of compute hardware), and real workload traces (e.g., VM scheduling requests). Moreover, systems like CloudLab [RET14] allow free and dynamic provisioning of compute resources to build cloud computing testbeds.

Evaluating edge computing systems is much more difficult due to the lack of established benchmarks, reference architectures, trace data sets, or even real-world applications. We attribute this to RQ1, and the subsequent lack of available test beds. Consequently, edge computing systems researchers often have to rely on simulators such as iFogSim [GVDGB17], or EdgeCloudSim [SOE17], or emulation techniques such as EmuFog [MGG+17]. The benchmarks executing with these tools are then often tuned to the specific approach, because there are no available trace data or cluster configurations from real edge computing systems. This lack of trace and profiling data of real edge computing hardware is problematic for edge computing research that relies on simulators. Overall, we observe that there are serious gaps in available methodologies to develop and validate operational mechanisms for edge computing systems.

1.2.2 Challenges and open problems

There are many open challenges to realize the type of system we have described. In this section, we outline the key challenges which we aim to address with our contributions.

Edge intelligence requirements

Edge intelligence is a nascent field of research, and the community is still building consensus on the fundamentals. Around 2019, several papers appeared that discussed applications, and different but orthogonal aspects of edge intelligence. For example, Zhou et al. [ZCL+19] focused on deep learning and other ML algorithms tailored to edge computing, as well as fundamental architectures for ML training and inferencing at the edge. Zhang et al. [ZWL+19] presented the design of OpenEI, a framework for enabling edge intelligence applications. It introduces a layer for selecting and accessing models over REST APIs, as well as an abstraction layer over ML frameworks. So while there has been some conceptual groundwork, concrete requirements for platforms that manage edge AI workflows, as well as the supporting infrastructure for edge intelligence, are still underdeveloped and require further investigation.

Building a distributed compute fabric

Infrastructure for edge intelligence As we described in RQ1, there is currently no consensus on what the supporting computing infrastructure for edge intelligence will look like exactly. Several proposals have been put forth that are either tailored to specific use cases like wearable augmented reality (AR) [HCH+14] or smart city monitoring [CBSG17]; or simply propose smaller, but more geo-distributed data centers [MB17]. This disparity makes it very difficult to develop generalizable operational mechanisms, such as autoscaling, load balancing, or scheduling, for edge systems. Since developing such mechanisms is one of the major goals of this thesis, a better understanding of the underlying infrastructure and its runtime behavior, as well as architecture models for edge intelligence is necessary.

Sensing and communication middleware Components across the entire edge intelligence stack, from IoT sensors, actuators, edge computing resources, to running applications, require low-latency event-driven communication [HKM+17, PDT18]. Currently, device communication in IoT and smart city environments is facilitated by cloud-based message-oriented middleware (MOM) based on the publish-subscribe model [Bar15, Sta19]. Using cloud-based MOM requires routing messages through the public Internet, which incurs significant latencies [KKY+10], inter-network traffic, and causes privacy issues. Edge intelligence instead necessitates the use of edge resources for message brokering, while also allowing centralized configuration and management. It is particularly challenging to maintain low latencies in edge environments where both clients and edge resources are mobile, and regularly leave and enter the network [RDR18, RND18]. Two key objectives are therefore: (1) elastically expanding and retracting brokers from the cloud to the edge, and (2) re-configuring client–broker connections at runtime to maintain low response times. For (1), a promising approach can be novel biology-inspired computing paradigms such as Osmotic Computing [VFD+16], where services *diffuse* to the edge or back to the cloud in order to meet some operational goal. It is different from cloud-based autoscaling in MOM that does not consider proximity of clients and brokers [GSGKK15]. For (2), existing IoT message brokers have configuration approaches that can bridge edge brokers but are typically static [Sca14, Gar16]. Systems such as PADRES [JCL+10] implement overlay network techniques, but it is unclear how well these work in dynamic edge environments under client and broker mobility.

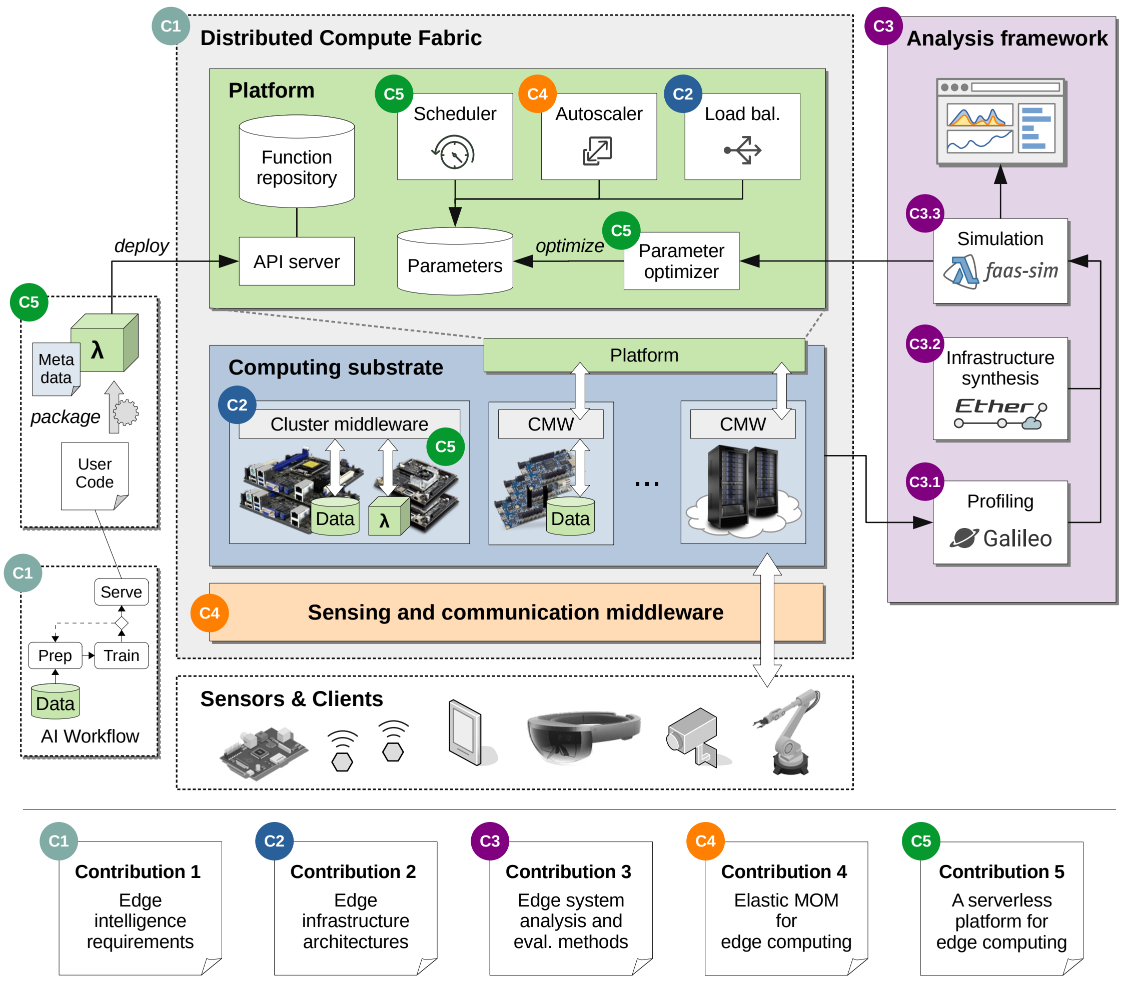
A computing platform for edge intelligence Another key component of our distributed compute fabric is an application runtime environment that is tailored to edge AI workloads. The complexity and scalability of cloud computing applications has lead to sophisticated new system abstractions that we can borrow from, which simplify application deployment and operation. In the serverless model for example, application developers define application code as *cloud functions*, and the events that trigger their execution [JSSS+19]. Developers do not have to know anything about the underlying infrastructure. The serverless platform is responsible for resource allocation (scheduling), autoscaling, discovery, and executing function code [HFG+18]. Analogous to the idea of cloud functions, we hypothesize that *edge functions* can serve both AI model training and inferencing, while enabling complete transparency of edge infrastructure. This is particularly challenging given the geo-distributed and heterogeneous nature of the infrastructure [SGL19]. Two key challenges are: (1) providing the correct abstractions for developing and running edge AI functions on heterogeneous infrastructure, and (2) resource allocation strategies that can make tradeoffs between data and computation movement and generalize to different infrastructure scenarios. For (1), research has shown that serverless can be a good abstraction for serving AI models [IMS18], but has also revealed several limitations for data-centric computing [HFG+18] in general. Existing platforms cannot reason over the dataflow of functions, or the type of workload they are executing and for what computing platform it is optimized for. For (2), there is a wealth of research on cluster scheduling [SKAEMW13] or service placement [ASDL19] algorithms, but it is unclear how well these work in serverless edge computing environments. As we described in RQ1, edge infrastructure may be very different depending on the use case, so schedulers must generalize across different infrastructure scenarios.

Evaluating edge computing systems

A key challenge in building the type of edge computing systems that we described, are robust evaluations that allow generalizable conclusions. As we described in RQ3, given the lack of real-world edge infrastructure, edge computing researchers often rely on simulators [GVDGB17, SOE17], or emulation techniques [MGG+17]. However, the benchmarks executed with these tools have several issues. First, the lack of trace datasets for edge systems requires the simulators to make certain assumptions when simulating workload execution on edge infrastructure. We have found tentative evidence that these assumptions, for example for energy models [CRB+11], may have limited applicability for edge computing infrastructure [RAD18]. Second, simulating or emulating a system requires a description of the system under test, i.e., the nodes in a cluster, their hardware specifications, and the network topology that connects them. Most tools provide ways to either create topologies manually [GVDGB17], which typically results in small topologies with few nodes, or are created randomly based on graph properties such as degree distributions [MLMB01, MM08], which may not produce representative topologies.

1.3 Contributions

This thesis has five main contributions that address the presented open challenges. We now explain the relationship between the respective thesis chapters in which the contributions are presented, and the previously published material. The following figure shows an overview of how the contributions are connected across the different systems.



C1. Edge intelligence: applications and requirements analysis

This contribution builds on [RD19], where we examined technological trends at the intersection of human augmentation, Internet of Things (IoT), smart cities, and AI systems; and discussed how these trends are leading us to a new paradigm of *Edge Intelligence* systems. In the paper, we develop a theory about the key elements to technologically realize edge intelligence systems. In Chapter 2 we present a revised version of the argument, which lays out the foundations for the problem statement presented in Section 1.2. The parts focusing on AI workflows, edge AI applications, and their operationalization, were previously published in [HMR$^+$19, RHM$^+$19, RHM20b]. We thereby contribute to the existing body of knowledge on Edge Intelligence [ZWL$^+$19, ZCL$^+$19] by focusing in particular on general computer systems support for edge intelligence, and deriving requirements for the underlying edge infrastructure.

C2. Edge infrastructure architectures

In Chapter 3 we present a comprehensive discussion of different candidates for edge infrastructure, that is a synthesis of the research presented in [RD19, RRD21, RLF$^+$20]. To better understand the emerging edge infrastructure architectures, we have built several testbeds that comprise a selection of edge computers from the computing continuum. We identified a concrete gap in this continuum, which we addressed in [RAD18]. We presented a prototype for a portable cluster-based edge computer, and performed several experiments that highlight the challenges that systems using such computers face. In [RLF$^+$20], we examined different distribution models of edge infrastructure, and gathered concrete numbers on existing or emerging deployments. The contribution is a significant step towards a better understanding of edge computing infrastructure, and the challenges in enabling the system abstractions we have become accustomed to from cloud computing.

C3. Edge system analysis and evaluation methods

Over the course of the research presented in this thesis, we have developed various evaluation methodologies tailored to edge computing systems. In Chapter 4 we present the synthesis of the results as a separate contribution in three parts.

C3.1. Profiling edge computers The cluster-based edge computer we presented in [RAD18] was the first system for which we created comprehensive profiling experiments. In subsequent research [RRPD19] we systematized the process and the tooling into an experimentation and analytics framework called *Galileo*, that allows researchers to (a) describe and perform reproducible profiling experiments, (b) gather and store system monitoring data in a common data model, and (c) enable ad-hoc and statistical data analysis. We have published Galileo as open source software [Rau20b], and used it to gather profiling data of a broad range of edge devices and workloads, that we have published as open data sets [RR20]. Furthermore, we have used profiling data to drive trace-driven simulations in [RRD21, Ras20, Rai21].

C3.2. Generating plausible edge system topologies Edge system simulators or emulators require as input a description of the infrastructure and the network topology under test. In [RLF$^+$20] we presented *Ether*, a tool that generates such synthetic infrastructure configurations and network topologies, that are based on references scenarios derived from real-world edge computing systems. The scenarios are parameterizable to vary characteristics such as cluster size, geographic dispersion, device heterogeneity, and network topology. We demonstrate how our tool can generate synthetic infrastructure configurations for the reference scenarios, and how these configurations can be used to evaluate different aspects of edge computing systems. We have since used the tool in the evaluations of [RRD21, Rai21, Bru21, Pal21].

C3.3. Trace-driven simulation framework Some problems of existing simulation models for edge computing were uncovered in [RAD18], and by [Rai21]. Key ideas of trace-driven simulation for AI systems were initially developed in the context of [RHM20b] and then extended for edge computing in the evaluation of [RRD21]. We developed a standalone simulation framework for edge computing [RRR20] that extends existing simulation models with, (i) a high-level flow-based network simulation, (ii) improved modeling of edge system energy consumption, and (iii) improved modeling of resource contention and workload interference. For our work, the simulator serves two purposes: first, it allows us to perform scalability and sensitivity analyses, and second, by gathering system traces at runtime, we can use the simulator to run optimization algorithms that tune parameters of the system's orchestration mechanisms.

C4. Elastic message-oriented middleware for edge computing

In Chapter 5 we present our contribution to the sensing and communication middleware layer of a distributed compute fabric. In [RDR18] we presented a system design for a messaging middleware based on the principles of osmotic computing. The system bootstraps a broker network from the cloud that dynamically diffuses additional brokers to the edge as necessary. In the thesis we extend the ideas of the paper by introducing concrete mathematical models for the diffusion process. In [RND18] we presented a prototype of this system that implements algorithms to orchestrate a broker–client network based on proximity and network QoS. We showed that our system can significantly reduce end-to-end latencies even in the face of high client mobility and dynamic resource availability. The initial prototype revealed several limitations, which were later investigated in [Gei19], and [Bru21]. First, subscriber mobility would lead to message loss, which we addressed by introducing a transactional re-configuration mechanism. Second, our network QoS monitoring protocol puts significant stress on the network, which we addressed by extending the Vivaldi virtual network coordinate system. The two principles we developed that apply to edge computing systems in general are (i) **proximity** as first-class-citizen in the orchestration model, and techniques to monitor proximity efficiently, and (ii) **elastic diffusion**, i.e., the idea of auto-scaling based on proximity, rather than classic system metrics such as resource utilization.

C5. A serverless platform for edge computing

In Chapter 6 we present our contribution to the design and implementation of key mechanisms to realize a computing platform for edge intelligence. The platform builds on the principle of serverless computing, but tailored to edge AI workloads and the operational challenges of edge infrastructure. In [NRS$^+$17] we introduced the concept of a serverless edge computing platform, and its principal components. To investigate existing platforms and compare cloud and edge-based solutions, in [Sch19] we developed a smart home data analytics application using the serverless model, which was implemented in both a cloud-based, and an edge-based platform. The evaluation revealed the importance of co-designing data and computation management, i.e., making the tradeoff between moving function code to the edge, or moving data to the cloud. We presented the design and fundamentals of a serverless edge computing platform in [RHM$^+$19], where we focused specifically on supporting data-intensive applications, such as running AI workflows. We introduced a high-level programming model that elevates concepts from AI workflows as first class citizens. We then developed a prototype of this platform in [RRD21], the core of which is a custom function scheduler that makes tradeoffs between code and data movement, and considers the heterogeneity of workloads and compute resources. Moreover, we developed a method to automatically fine-tune scheduler parameters to different edge infrastructure scenarios. The evaluation revealed several limitations, which are the subject of subsequent research in [Rai21, Pal21, Mar21].

CHAPTER 2

Edge Intelligence

This chapter outlines our vision of *Edge Intelligence* based on current development trends and their associated challenges for building the supporting computer systems. To solve the challenges we identify, we propose a distribute computing fabric for edge intelligence, that federates geographically dispersed and heterogeneous edge computing resources, provides a sensing and communication layer, and employs advanced operational mechanisms to deploy and manage applications on this federated infrastructure. The presented analysis and the challenges we elicit are the foundation for the remainder of this thesis.

We begin by analyzing the current development trends of human augmentation in Section 2.1 and edge AI in Section 2.2. We proceed to give a detailed discussion of the critical system components for a distributed compute fabric we have identified, in Section 2.3: **sensing and communication middleware**, a **computing substrate**, and **intelligent operational mechanisms**. In Section 2.4, we examine the numerous existing computing models and how they apply to such a system. The summary in Section 2.5 concludes the chapter.

2.1 The convergence of humans, things, and AI

Edge AI and Human Augmentation are currently considered to be two of the major emerging technology trends [Pan18], which we attribute to three key technological developments. First, the increasing number of Internet connected sensors and smart things embedded in our surroundings, i.e., the Internet of Things (IoT), and wearable smart devices [Cis21], generate a wealth of opportunities to create many types of applications that can enhance our everyday experience and quality of life. Second, the increase in data availability [Cis19], consolidated computing power, as well as the advancement of machine learning (ML) techniques have fostered the development of AI supported applications that continue to transform virtually every aspect of our daily life. Third, the growing friction between cloud-based AI solutions, and the need of many applications to analyze

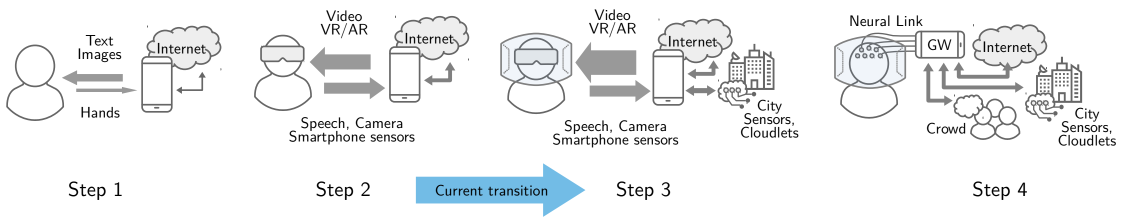


Figure 2.1: The currently observable evolution of the cyber-human

high volume and high velocity data streams in near real-time [SSX+15, NRS+17], has pushed hardware developers to create miniaturized AI accelerators that promise to bring AI to the extreme edge. In this section we discuss these developments in more detail.

2.1.1 The cyber-human evolution

We take a step back and discuss the evolutionary steps of the cyber-human, as illustrated in Figure 2.1. The Internet allows us to store and access information at immense scales, and smartphones have become our main gateway to this world. For many, the smartphone is already an extension of their self, representing the first step in the cyber-human evolution. Typing in a search term into your smartphone and parsing the information it displays takes us a long time compared to the bandwidth available to a superhuman AI. Similarly, we control our smart things via manual gateways, such as the apps on our phone, at a very low bandwidth. With the development of increasingly sophisticated smart devices such as mixed reality smart glasses, combined with augmented reality (AR) and AI techniques, step two has nudged humans and things closer together [HCH+14]. AI applications help us, or our cars, to recognize objects and overlay our field of vision with contextual information. As edge devices become more intelligent, and more and more sensor data in our surroundings can be aggregated, processed and served via edge resource at high bandwidth, step three will fully realize a transparently immersive experience. However, for the seamless augmentation of human cognition, the bandwidth requirements will be even higher. Imagine a world in which we perceive and interact with the cyber-physical surroundings via neural-interface connected gateways and AI assisted control mechanisms. The requirements of step four in the cyber-human evolution are fundamentally challenging our way of building systems.

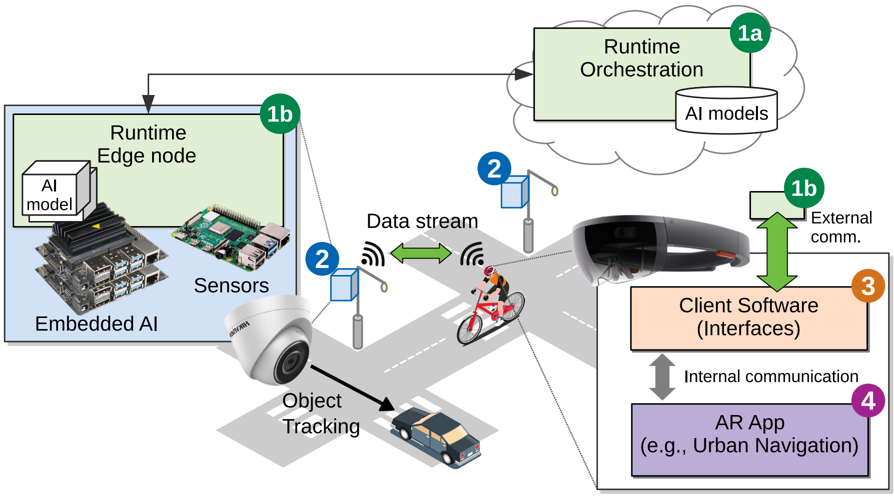
An important aspect to highlight in this evolution is the trend towards artificial general intelligence (AGI), i.e., the development of AI that can solve intellectual tasks that a human being can [PG07]. From a system's perspective, it is likely that AGI will not be achieved by a single method, but rather a context-sensitive ensemble of different task-specific AI agents that permeate our environment via an underlying computing substrate [HJO+10]. Some projects that work towards AGI are based on the idea of an interconnected and self-governing network of AI solution agents that can cooperate to solve increasingly more complex problems as the network grows [MG18]. In such a system, edge computing will play a fundamental enabling role. Compared to the cloud, the edge can provide trust, application-specific hardware in close proximity to data and users, and thereby handle the immense bandwidth and low-latency requirements.

2.1.2 How Edge Intelligence can increase the fidelity of our perception of reality

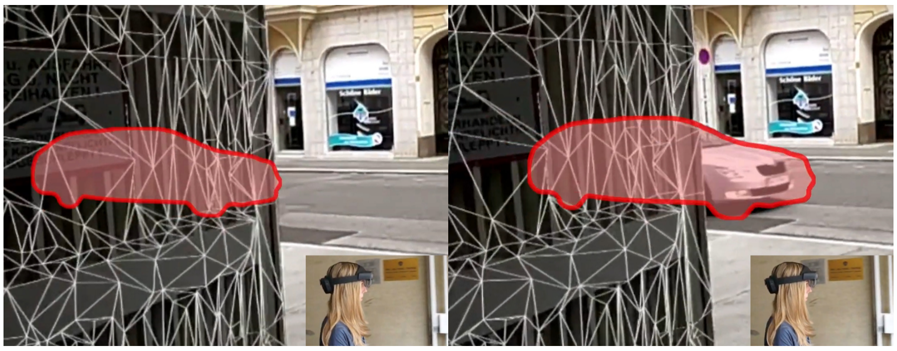
Augmenting human cognition with computation is an idea that dates back decades and has been explored by science-fiction writers and researchers alike. In 2004, Satyanarayanan described a wearable cognitive assistance use case that he said at the time was "clearly science fiction" [Sat04], but was made a reality almost ten years later [HCH+14]. This development is the transition between step two and three in our cyber-human evolution timeline. Today, products such as Google Lens[1] attempt to bring these systems to the masses by integrating cloud-based visual analytics AI systems with personal edge devices. Going a step further, consider a high-fidelity personalized visual discovery assistant. You use your smartphone to visually discover things that your assistant knows are important to you. By pointing your camera along a shopping street, the application recognizes streets, buildings and objects, and displays an AR overlay with retailers that have offers that you may be interested in, restaurants that have food you like, or points of interest you may want to visit, combined with real-time information such as the number of guests currently in a restaurant or shop. With current state-of-the art technology, we would realize such applications via a direct link between the camera app and the cloud.

There are however use cases where momentary information is important, such as high-speed manufacturing lines, or virtual assistants of fast-paced interactions, like navigating through a smart city, where cloud-based analytics become infeasible. There are numerous other wearable cognitive assistance applications that require such momentary information [Sat17]. These scenarios focus mostly on analyzing the video stream of a camera, and annotating it with contextual information. Future applications will move beyond isolated sensing, but synthesize data from many sources that cannot be streamed into the cloud. Consider an urban cognitive assistance application that guides you through a public space shared with other people and autonomous agents such as self-driving cars and mobile robots. Suppose each agent broadcasts its sensor data (gyroscope, accelerometer, location, etc.). Edge resources aggregate sensor data sources in proximity to create a hyper-reality overlay to give you a personalized experience and guiding you through the space. Figure 2.2 illustrates this. As fidelity of this overlay depends on available proximate resources and capabilities [LH18], an approach is needed for a seamless edge/cloud AI system that can transparently trade off accuracy and computation, where accuracy increases with computational power. Suppose these applications could be served by a continuum of edge resources and the cloud. If models are served via edge resources, fidelity of recognition can be much higher, because models can be trained for specific domains and then deployed in proximity of where they are required. Data can then be streamed instead of chunked (e.g., like Google Lens sends individual images to the cloud), which can further increase fidelity and responsiveness of an AR system.

[1]`https://lens.google.com/`



(a) A system for urban cognitive assistance, where augmented reality wearables (4) connect via local gateways (3) to proximate sensors and compute resources (2), managed by an edge computing platform (1).



(b) Silhouette of an approaching car behind a wall that is displayed on the user's wearable device.

Figure 2.2: Example of augmenting human cognition with edge intelligence, which we described in [RHS$^+$21].

Step four in the cyber-human evolution will confront researchers with a multitude of challenges over the next decade. Engineering and managing a system to allow the pervasive and seamless integration of all these applications at scale into our everyday lives requires a new way of thinking about infrastructure and processes, beyond our current understanding in pervasive computing, in particular as AI agents become more involved and applications co-evolve with ML models. Deep neural networks can be pre-trained on massive datasets in the cloud, localized data can be used to refine models the edge using transfer learning, and applications can then be deployed on edge resources optimized for inferencing. We discuss this in depth in Section 2.2.2.

2.1.3 The AI-enhanced smart city

Smart cities provide a number of exciting opportunities for edge computing [DNŠ17]. In fact, we believe that edge intelligence will be the key to fully connect the cyber-physical city with its cyber-human inhabitants. Smart public spaces understand situations by interpreting activity. Many dispersed cameras can be used to create data analytics models (like crowd behavior [ZWT12], flooding or fire spreading models, or ecosystem/plant monitoring [Bec18]). An AI could use weather data together with camera streams to determine road driving conditions, and put that information in context with historical data to assess the immediate risk of accidents. Autonomous vehicles can in turn use this information to adapt their driving behavior. In the case of accidents, a smart city system would react by immediately informing authorities, dispatching first responders, and re-routing traffic. Air quality sensors distributed throughout the city can be used to track development and spread of air pollution. A number of similar use cases with different sensory and analytics requirements could be conceived. What these use cases illustrate is the common need for real-time, location-based data about urban environments and activity at different levels of fidelity. Moreover, appropriate edge resources are necessary to both process and store data close to where they are generated [SVDI16]. Sending data to the cloud to do inferencing on ML models deployed as web services is not feasible in many cases, in particular as the required perception fidelity increases. Researchers and practitioners will therefore be challenged to reconcile smart-city scale sensing capabilities and compute infrastructure with scalable deployment of AI applications to the edge.

2.2 Operationalizing edge intelligence

We have discussed several applications that emerge from the convergence of humans, things, and AI, and have shown that AI services play a significant role in edge intelligence. Delivering AI services involves complicated workflows where data scientists and software engineers collaborate to create and deploy ML models underlying the application [HMR$^+$19, HDB17, BBC$^+$17]. Data is curated and explored, feature engineering and training is performed to create ML models, which are subsequently deployed, monitored, and updated during their lifetime. As edge intelligence transitions from individual and isolated prototypes into interconnected production-grade deployments, the ad-hoc way of building and deploying ML models and AI applications will become impractical. Instead, edge AI operations platforms will fully automate the end-to-end process and provide a closed feedback loop between runtime metrics of deployed models and the workflows that create them [HMR$^+$19]. Such platforms are already a reality for cloud-based AI workflows as we will show. In the case of edge AI, however, the process becomes much more involved. Heterogeneous computing resources, data locality, scalability issues, etc., have to be considered when automating workflows. In this section, we first summarize existing knowledge on AI operationalization. We will then analyze different operationalization requirements for edge intelligence, identify key mechanisms that can facilitate these scenarios, and then discuss various architectural possibilities and arising challenges.

2.2.1 AI lifecycle management: operations for AI

The idea of operationalizing AI has become pervasive throughout both industry and research [Gar17]. Software engineering researchers and practitioners have, over the past decades, developed a variety of DevOps methods for managing the software lifecycle. Continuous integration (CI) and delivery (CD) have become de-facto standards in modern software development processes. Advanced methods such as A/B testing, or continuous experimentation are also employed more and more. Only recently have such lifecycle management techniques and tools for AI applications appeared.

AI pipelines are workflows, i.e., graph-structured compositions of tasks, that create or operate on machine learning models [HMR+19]. Similar to a CI pipeline that is triggered when new code changes arrives and then automatically compiles, tests, lints, and deploys software artifacts; an AI pipeline defines a sequence of steps to manage the AI application lifecycle, from data preprocessing, to model training, to custom steps such as model compression or robustness checking. Transfer learning techniques allows creating base models from aggregated data, which are then refined using personal or domain-specific data, creating a hierarchical workflow. Some examples of such workflows are shown in Figure 2.3.

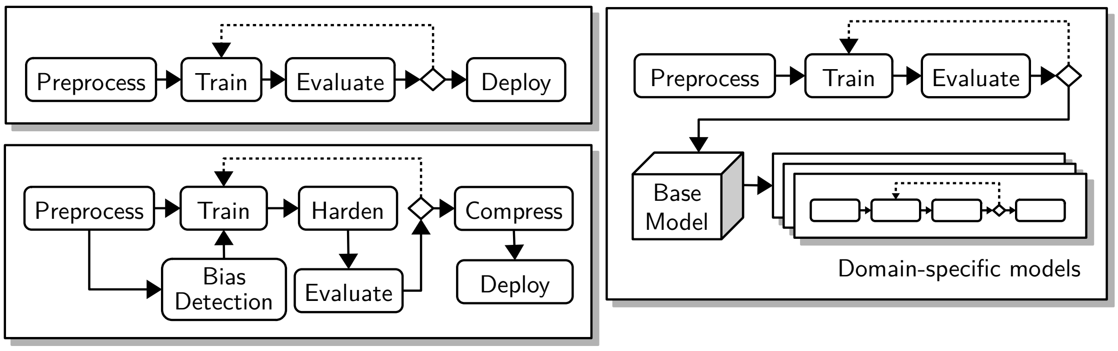


Figure 2.3: AI pipelines with different levels of complexity

Operationalized AI pipelines integrate the entire lifecycle of an AI model, from training, to deployment, to runtime monitoring [SVK+17, BBC+17, HMR+19]. To manage risks and prevent models from becoming stale, pipelines are triggered automatically by monitoring runtime performance indicators of deployed models. Retraining triggers can be as simple as a weekly schedule, or a combination of rules, such as detecting concept drift [GŽB+14] and monitoring the amount of data available for training. Figure 2.4 illustrates this.

Several systems for cloud-based AI lifecycle management have emerged. Uber's Michelangelo [HDB17], IBM Watson Machine Learning [IBM18], ModelOps [HMR+19], TensorFlow Extended [BBC+17], or ModelHub [MLDD17], all provide tools and platforms to define and automate AI pipelines, share data and models, and monitor runtime performance of models, by leveraging cloud services and infrastructure.

We can see that most efforts for AI lifecycle management are focused on both training

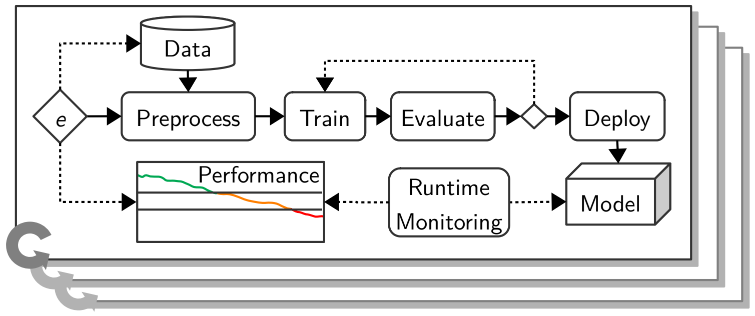


Figure 2.4: Operationalized AI pipeline with a rule-based trigger e that monitors available data and runtime performance metrics to form an automated retraining loop

and deployment in the cloud, and largely neglect edge computing system characteristics. In particular, it is unclear how platforms should handle training data that cannot leave the edge, how to reconcile hybrid edge/cloud infrastructure for executing pipelines, how to scale deployment of a large number of ML models to the edge, and how to achieve scalable runtime monitoring of live models.

2.2.2 Edge AI

AI services that leverage edge computing resources and data source will play a fundamental role in enabling the applications we discussed in Section 2.1. To better understand the specific challenges that confront edge AI systems, we examine specific use cases is more detail. The presented examples are adapted from real-world use cases that currently use cloud-based AI platforms to deliver the applications. We introduce plausible new requirements to each use case that blend characteristics from AI operationalization and edge computing, to highlight the challenges of developing and operating edge AI applications. Selected applications are used in Chapter 5 and Chapter 6 as evaluation scenarios.

Personal assistants Cognitive mobile personal assistants continuously monitor health data via bio sensors, and can predict and raise alerts for critical situation like critically low blood sugar levels [Cor17]. Key concerns in this use case are prediction accuracy, inferencing latency, and data privacy, which can be improved by integrating patients' edge devices into the workflow. To optimize accuracy, the training process is split up into two phases: First, the service provider trains a base model on a representative, anonymized [AP08] sample of the entire population. This is a resource intensive and long-running process that is performed in a cloud environment. Second, in order to account for patient-specific patterns in the data, the individual models need to be adjusted and fine-tuned for each (type of) patient. However, these patient-specific data should be kept private and there should be no way to associate the data points directly with a patient's identity. The base model is transmitted to the edge device and refined using transfer learning techniques and data collected at runtime at the edge. This refined model is then served on the patient's device, thereby enabling low-latency and privacy-aware inferencing.

Field technicians Field service technicians often travel to on-site locations, including engineers maintaining power facilities, mechanics fixing industrial equipment, and technicians for an ISP. Mobile devices augmented with AI capabilities help to identify faulty parts, recommend diagnosis paths, or log and validate the technician's actions [Cor18]. Key concerns in this use case are reliability, inferencing latency, bandwidth, and trust. Due to limited network connectivity, AI models need to run on the these edge devices, such as visual recognition models to classify photos taken in the field. Device equipped with AI accelerators could be used for video stream analytics where momentary information is relevant, for example in high-speed manufacturing lines. Being able to predict and preload AI models on devices with limited memory can be useful when managing fleets of devices. Context-aware policies can help facilitate a trusted AI workflow. For example, security constraints related to device ownership and registration may require that data gathered in the field only be transferred when connected to the corporate network.

Predictive analytics for smart homes Through the IoT, smart homes become cyber-physical systems, where IoT devices can sense and act on the environment. Combined with AI techniques, actuators can be powered by AI models trained on the gathered sensor data. One such use case we presented in [Sch19] uses an AI pipeline to improve the power efficiency of household water heating system connected to a photovoltaic system (PVS). Sensors attached to the water heating system record the water usage and water temperature, occupancy sensors record the number of people in the house, and a weather station records data. The historic data are stored in a local data store. A model is trained to predict future water consumption behavior, and to optimize the power usage costs by using the PVS to heat water when it is most appropriate. Local edge devices are used for the entire end-to-end life cycle, i.e., to store and process data, train the model, as well as running the predictor.

Smart urban spaces Smart public spaces understand situations by interpreting activity. For example, consider a smart city application that recognize problematic situations such as traffic accidents or crowd behavior [ZWT12]. Any type of decision-making will be highly locality specific, yet abstract patterns may be similar across locations. Dispersed edge and IoT devices produce data can provide locality-specific insights, such as cameras or sensor nodes dispersed through the city. Public cloud-based ML facilities, that are cost-efficient and can be provisioned on-demand, can be used to train a base model, that is then refined using transfer learning at on-premises cloudlets that have access to data from local sensors and devices. These cloudlets can use distributed learning to avoid aggregating all data in proximity, and then serve models for low-latency inferencing.

Augmented urban reality Suppose a scenario in which edge resources serve ML models to support an application for detailed visual discovery of points of interest in a city via AR. An AR overlay (either via a smartphone or HMD) provides contextual information about objects or things that are in the camera stream. We have outlined other such use cases for augmented urban reality in [RHS+21]. Ideally, a resource hosts ML models (e.g., for object recognition), for the points of interest in its proximity. Locality and context awareness play a key role in making this work. The resource understands that, in its proximity there exist, for example, an art museum, a botanical garden, or a shopping mall, which each have their own ML models for the given domain. Edge resources can also be used to train models. This is particularly relevant for scenarios where the data to train or refine domain-specific models is proprietary or privacy-sensitive, and may not leave the edge network. These scenarios demonstrate the complexity of edge AI workflow management.

Edge AI workflows

Systems like IBM AI OpenScale[2] make use of cloud-based technologies to enable end-to-end AI operationalization in the cloud. Cluster computing systems such as Apache Spark, and clustered deep learning infrastructure operate on data persisted in cloud-based object storages [BBD+17], and models are deployed as web services on a cloud-based hosting platform from which they can be easily monitored. Companies are driving efforts to integrate edge resources into this process [MS18]. Google Cloud IoT Edge[3], Microsoft Azure IoT Edge[4], or Amazon AWS Greengrass[5] leverage edge resources to serve ML models, but data preprocessing, model training, and message brokering is still mainly performed by the cloud. To fully realize end-to-end edge intelligence workflows we need to make full use of edge resources not only for serving models, but for all steps within the AI lifecycle. With this in mind, we identify five different AI lifecycle workflows, from training to serving.

Table 2.1 lists data and model characteristics for the basic workflows, specific ML mechanisms that are the key enablers, and some example use cases. Which workflow will apply to a given problem depends on many considerations: What are we training models for? How large are the models? How often are models re-trained? How much data is involved? How long does it take, and is it important that training is fast? Some simple curve fitting models for on-the-fly optimization can be executed in a matter of seconds, whereas training image classifiers may take several hours or even days. How fast does inferencing have to be? Can we tolerate latencies from sending data to the cloud? Is the data used for training or inferencing sensitive?

Cloud to Cloud This is the status quo as supported by cloud-based AI platforms such as Microsoft Azure ML, Google Cloud prediction API, or IBM AI OpenScale. Models

[2]https://www.ibm.com/cloud/ai-openscale
[3]https://cloud.google.com/iot-edge/
[4]https://azure.microsoft.com/en-us/services/iot-edge/
[5]https://aws.amazon.com/greengrass/

Table 2.1: Overview of AI operations workflows

	Data characteristics	Model characteristics	Enabling technologies	Example use cases
C2C	- Training data is centralized - Massive data sets	- Models are large - Huge number of inferencing requests need to be load balanced	- Scalable learning infrastructure [BBD$^+$17] - Data warehousing	- Image search - Recommender systems
C2E	- Training data is centralized - Inferencing data may be sensitive	- Inferencing may need to happen in near-real time - Large number of model deployments - Models run on specialized hardware	- Model compression [HMD15] - Latency/accuracy tradeoff [HZC$^+$17a] - Distributed inferencing [DCM$^+$12] - Transfer learning [PY$^+$10]	- Surveillance systems - Self driving cars - Fieldwork assistants
E2C	- Training data is distributed - Training data may be sensitive	- Models can be centralized - Huge number of inferencing requests need to be load balanced	- Decentralized/federated learning [MMRyA16, HRM$^+$18]	- Volunteer computing - Novel smart city use cases
E2E	- Training data is distributed - Training and inferencing data may be sensitive	- Inferencing may need to be near-real time	- Decentralized/federated learning - Distributed inferencing	- Industrial IoT (e.g., predictive maintenance) - Privacy-aware personal assistants - Novel IoT use cases

are trained in centralized training clusters using data aggregated in cloud-based storage silos. Models are then served on cloud resources (e.g., as web services in VMs).

Cloud to Edge This workflow integrates edge resources as deployment targets. It is useful for use cases where training models requires a lot of computational power, and there are massive amounts of data involved (e.g., training classifiers on ImageNet), but inferencing must be fast (e.g, real-time object detection), or inferencing data may not leave the edge (e.g., patient data). Google, Microsoft, Amazon and others are driving effort with their enterprise-grade edge AI platforms, which are still largely in alpha or beta stages of development.

Edge to Cloud This workflow makes use of edge resources to train data, and serves models in the cloud. The workflow may be useful for application where training data is massively distributed (such as in mobile computing scenarios [MMRyA16]), training data may not travel to the cloud, or training needs to be performed close to data sources. Yet, serving models requires either massive scalability, or decentralized inferencing is impractical.

Edge to Edge In this workflow, both training and serving happens in the edge. For reasons similar to the previous two workflows. Industry 4.0 use cases illustrate this, where sensitive data may not leave the companies premises, and inferencing needs to happen in near real-time at the edge.

Complex Hierarchical Models It is likely that future scenarios will call for a creative mix of the above mentioned workflows. Some use cases may require base models to be trained on aggregate data in the cloud, then deployed and fine-tuned with data at the edge (e.g., demand forecasting for a retail chain where a model is built on data across all locations, and fine-tuned for specific stores; or personalized diabetes assistants, where anonymized data across patient groups are used to train a base model, that is then refined using the individual patient's data). There are experimental systems that use federated learning to train model fragments on-device, aggregate the anonymized models in the cloud, and then fine tune on personalized data, such as Google's smart mobile keyboard app [HRM+18]. Localization and context play a big role here, and a hierarchical edge/cloud architectures can help facilitate this as we will show.

The role of edge computing

It is clear why edge computing is an appealing model for these advanced AI applications. A hierarchical hybrid cloud and edge computing architecture, as shown in Figure 2.5, can help facilitate complex edge AI operations scenarios for the applications we presented. Anonymized data can be aggregated in the cloud to create common base models. Base models are deployed to edge resources, where context-specific and unfiltered data can be used to refine the models in a privacy-preserving manner. Specialized hardware resources deployed at the edge (see Section 3.3), combined with their proximity to clients, can enable highly efficient low-latency access to AI services or sensor data from the environment.

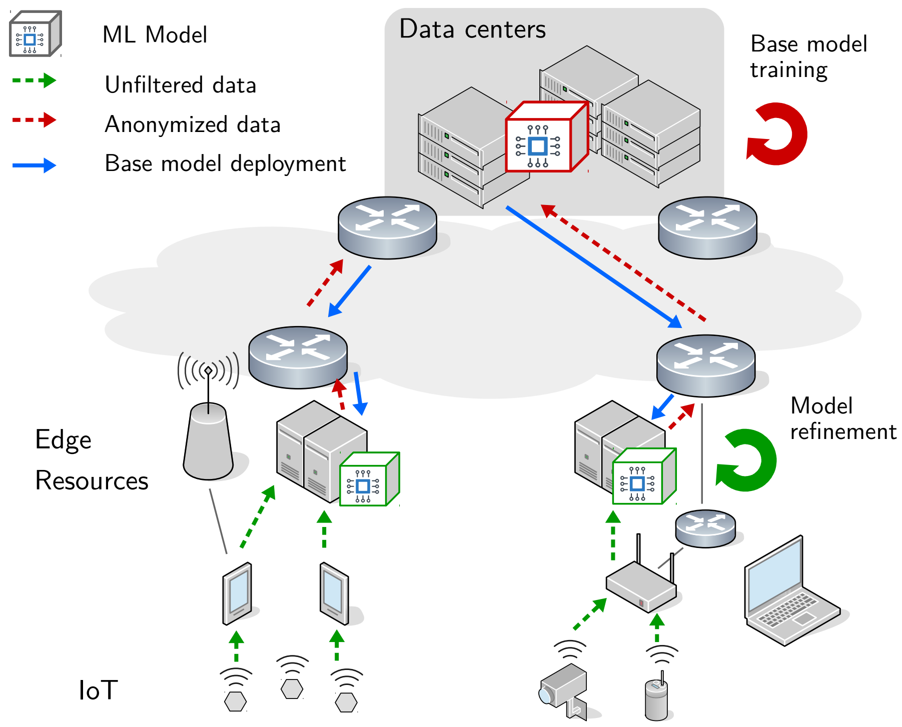


Figure 2.5: Edge AI scenario that leverages hierarchical edge and cloud infrastructure

As promising as edge computing is, there are numerous challenges in realizing such an end-to-end architecture. Deploying and managing AI workflow tasks to the edge in a generalized way may be difficult (compared to, for example, packaging the ML model as a web service and deploying it to a managed Platform-as-a-Service (PaaS) offering), as edge resources are very heterogeneous and cannot yet be used in a classic utility computing manner. How do we allow a "write once run anywhere" approach for heterogeneous resources and workloads? Some training and inferencing tasks may benefit from AI accelerators or GPU, requiring specific computing platforms and precise resource provisioning strategies. Moreover, we may not know up front whether an edge computer has sufficient resources to perform the task. Should the task run on a Raspberry Pi or other SBC that is very close to the data, or rather a more powerful server computer in the cloud which requires moving the data? How will data be stored and accessed from edge resources to make it available in a safe and privacy-aware way for training and inferencing? How can applications communicate efficiently in this highly dynamic and distributed computing environment? We take a closer look at some of these questions, and discuss possible solutions and open challenges.

Our proposal is a distribute computing fabric for edge intelligence, that federates dispersed and heterogeneous edge computing resources, and employs advanced operational mechanisms to deploy and manage applications on this federate infrastructure. We now present the key elements of this system in more detail.

2.3 A distributed compute fabric for edge intelligence

We identify the following three key components to implement the system we envision: **sensing and communication middleware**, a **computing substrate**, and **intelligent operational mechanisms**, which are the driver for the remaining thesis. We discuss in more detail each component and the associated challenges.

2.3.1 Sensing and communication middleware

Implementing use cases such as smart public spaces on smart-city scale requires many different sensing capabilities [DNŠ17]. Having individual companies develop and deploy specific sensors for specific use cases seems inefficient in the long term. Instead, city planners will have a high stake in providing application developers with a sensing infrastructure to make use of smart city data. It seems much more likely that an initial set of smart city and IoT use cases will bootstrap a general set of requirements for sensor capabilities, which will then trigger a deployment at increasingly larger scale. Similar to a smartphone that has a wide array of sensors installed giving developers lots of opportunities to create novel and creative applications. Not all applications use all sensors, in fact, some usage behavior may not require specific sensors at all. They are there nonetheless, because they provide potential utility for future apps. It is reasonable to assume that a similar model will work well for smart-city scale edge intelligence. Once a family of edge intelligence application emerges, and we understand the broader range of requirements, companies can start building arrays of sensors to support applications with sensing capabilities, hardware and potential utility. In fact, there already projects, such as the Array of Things[6], that can work as substrate for providing Sensors Data as a Service (SDaaS) [ZIH+13].

We foresee several challenges for edge computing in this model. Given the large number and dynamic and mobile nature of both publishers and subscribers of sensor data [MJ10], and the stringent QoS requirements of edge intelligence use cases, we will be forced to rethink centralized messaging services such as Amazon AWS IoT or Microsoft Azure IoT Hub [RND18]. Novel messaging systems, such as osmotic message-oriented middleware [RDR18], can help facilitate transparent access to this geographically dispersed network of sensors and actuators, even under highly dynamic node behavior. In fact, in a survey on wide-scale publish–subscribe middleware, Bellavista et al. [BCR14] note this is a necessity: "there will be a growing number of application scenarios where the availability of middleware supporting large-scale PUB/SUB event distribution with QoS will be fundamental". Furthermore, it is unlikely that sensing infrastructure will be

[6]`https://arrayofthings.github.io/`

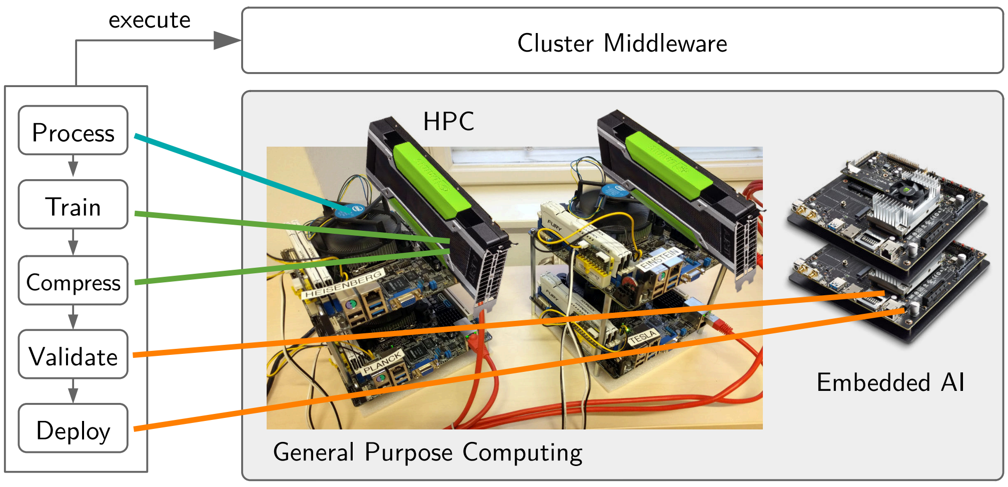


Figure 2.6: Capability-aware execution of an AI pipeline on a multi-purpose edge computer

static and managed by a single central authority. Consider the integration of cognitive assistance devices with self-driving cars and smart traffic systems [RHS+21]. Applications require access to public sensor data, such as statically deployed arrays of sensors; dynamic sensor data from cars; as well as personal sensor data, such as location and movement of pedestrians. How can these different static and dynamic data sources be reconciled to let AI applications make full use of them? We present our solution to the problem in Chapter 5. Moreover, while AI inferencing mostly works on real-time data in an event-based way, the majority of training techniques require batch access to labeled data. Storing and providing scalable access to these highly dispersed data is a fundamental challenge for edge intelligence, which we later show in Chapter 6.

2.3.2 Computing substrate

In contrast to cloud data centers, where the predominant infrastructure unit is the server computer that consolidates large amounts of CPU, RAM, and disk space, edge resources will be much more diverse and provide a variety of computational platforms to support the equally diverse range of use cases [SGL19, PDT18]. Chapter 3 is dedicated to examine this in greater detail. General purpose computing will be complemented by specialized high-performance computing (HPC) and AI optimized hardware, federated to create a powerful, high-density multi-purpose compute units. Together with cloud resources, hierarchies of connected clustered edge computers will form the backbone infrastructure.

Multi-purpose edge computers

At smart city scale, it is impossible to manually map software services to the respective platforms. To make efficient use of these platforms, hardware needs to be grouped into cohesive units of multi-purpose edge computers. An existing example is the Array of

Things (AoT) project, where nodes are deployed throughout the city, and comprise several SBCs with varying compute capabilities. When executing workloads, a scheduling system needs to understand the capabilities of the underlying resource, as well as the effects of scheduling specific workloads to specific platforms on the operational efficiency. For example, as illustrated in Figure 2.6 scheduling an ML model inferencing task will be both faster and more energy efficient on an embedded AI hardware than on a general-purpose computing platform. In mobile scenarios that require portability and reliability, energy efficiency is an additional constraint to consider [RAD18]. Ideally, a cluster middleware learns at runtime how to optimally provision resources for given workloads, learning the tradeoffs between energy efficiency, latency, and accuracy. These self-learning and self-adaptive clusters will form the infrastructural unit of the edge computing fabric. We discuss this in more detail in Section 3.1.

Edge system topologies

Compute resources that are placed in edge networks and pooled together to form a diverse and distributed compute fabric, can form various higher-level topologies. We discuss existing and emerging edge system topologies in greater depth in Section 3.3. The discussion shows the large variety of network topologies that can exist in edge systems, compared to data centers. Federating dispersed Internet-connected networks entails a large variance in bandwidth between nodes, which further complicates resource management and workload scheduling. It is important that generalizable middleware can deal with these networks, without having to make assumptions about the underlying topologies.

Virtualization

Applications rarely live in isolation, and a utility-based edge computing fabric has to deal with challenges that come with multitenancy, which is not as straightforward at the edge as it is in the cloud [LWB16]. Powerful server computers in data centers allow us to easily host multiple VMs as the unit of isolation. Although researchers have argued that Cloudlets will simply be VM hosts at the network edge [SBCD09], we cannot necessarily assume that all edge resources can provide this level of virtualization. VMs require a lot of resources and not all edge computers are powerful enough to provide fully-fledged VM-based virtualization. Because this introduces a break in the edge/cloud continuum, there are legitimate doubts whether VMs will work as the predominant form of isolation for edge computing. Moreover, it is important that developers can trust the “write once run anywhere” principle. Container-based deployment strategies have been recognized as a more viable solution, as they provide lightweight resource virtualization and isolation [Mor17]. More recently, Unikernels have been gaining traction as an alternative way of developing applications as completely isolated machine images that can run directly on top of a bare-metal hypervisor [MMR$^+$13]. However, such Unikernels rely on library operating systems, and are not yet as well understood as containers. Regardless of which technology will dominate, it is clear that multitenancy in resource-constrained environments (compared to a cloud data center) is challenging, as the efficient use of resources becomes more difficult.

2.3.3 Intelligent operational mechanisms

As we have shown, edge computing introduces new challenges for operations researchers. There is evidence that methods from cloud operations research, for example for energy efficiency, may only have limited applicability for edge computers, some of which we demonstrate in Chapter 3. Also, it is clear that maintaining a network of highly dispersed multi-purpose edge computer clusters introduces additional management and coordination complexity. We discuss some of the most pressing operations issues for edge intelligence. In this thesis, we focus on

Proximity & mobility awareness

In edge intelligence scenarios, both software agents and hardware resources can be dynamic and mobile. Allowing for mobility requires a good understanding of proximity, s.t., communication latencies between nodes can be optimized, and privacy policies can be enforced. Mobility can be particularly challenging for messaging systems that provide message delivery guarantees [MJ10, RND18]. Proximity awareness is therefore a fundamental enabling mechanism of mobility in edge computing [RDR18]. In networks, proximity can be distinguished into logical and physical proximity, and both play an important role in edge computing. For example, physical proximity matters when low-range network techniques such as BLE are involved, and logical proximity matters when minimizing round-trip time. Classic techniques as used by Content Delivery Networks (CDN), such as using anycast DNS and latency monitoring, are challenged in highly dynamic environments. Also, simple monitoring techniques, e.g., via network latency, can produce undesirable strain on the network and become difficult to manage, as we will demonstrate in Chapter 5. Besides network latency or logical distance, the application's response time should also be factored into proximity. For example, a message broker in close proximity with a that exhibits high response time because of congested queues can behave as if it were much farther a way. We believe that new methods synthesized from, for example, advanced techniques of interest management, as well as runtime application monitoring, will be necessary for enabling proximity and mobility awareness for edge intelligence at scale. Other network metrics to determine proximity should also be considered. For example, as we show in Chapter 6, a useful measure of proximity is the available bandwidth between data producers and consumers, to determine data and computation movement tradeoffs.

Intelligent workload scheduling

The highly heterogeneous nature and geographic dispersion of edge resources poses significant challenges for workload scheduling. Multitenancy in combination with a limited number of constrained resources further exacerbates the problem. In agent-based scenarios, intelligent eviction or suspend/resume strategies that respect the requirements of AI agents will be necessary, and call for sophisticated management mechanisms. Furthermore, latency and energy-aware scheduling techniques become more difficult in hierarchical architectures, where the underlying platform's operational optimizations must be treated as a black box (see Section 3.2). Effectively scheduling workloads to the edge will require many hard and soft constraints. Performance evaluations of state-of-the-art workload scheduling approaches (such as the Kubernetes container scheduler), exhibit clear scalability limitations when confronted with a large number of constraints [Den16]. Moreover, it is challenging to generalize scheduling parameters given the variety of existing edge infrastructure scenarios (see Section 3.3 and Section 6.4).

AI applications introduce additional challenges for scheduling, and we argue that some aspects of these applications should be considered as first-class citizens, s.t., platforms can reason about the application internals. As we discussed in Section 2.3.2, the capabilities of edge computers are much richer and diverse than those of server computers in classical cloud computing. However, edge resources are also more constrained, both in terms of computing power and storage capacity. From an operations' perspective, delivering AI at the edge has to include additional goals and constraints, in particular because proximity to data producers and consumers plays such a critical role in edge computing, both to maintain privacy and trust, enable low-latency data processing, and reduce inter-network traffic. These challenges are similar to those of large-scale IoT-cloud deployments, for which different provisioning systems have been developed [VSI$^+$15, NTD16]. However, these provisioning frameworks do not consider the co-evolution and dependency of AI applications and the underlying ML models. In particular, they assume a classic compile-test-deploy workflow of software, whereas an AI operations pipeline is highly customizable and its steps have different computational needs

If we take into consideration that the distributed compute fabric will be used for both training and serving AI models, we can summarize the operational requirements of workload schedulers as follows: (1) trade off data and computation movement: decide at runtime whether to move the data stored in edge networks to the cloud, or deploy application code to the edge, (2) consider proximity to data stores: deploy workflow tasks in a way that minimizes inter-network traffic and latency, (3) make efficient use of edge resources: GPUs or AI accelerators should be used if workloads can benefit from them (4) consider resource contention: edge resources may be much more constrained than data center servers, and the performance impact of scheduling multiple workloads to a resource may significantly impair application performance.

Proximity-based autoscaling and load-balancing

Two fundamental operational mechanisms of elastic systems are autoscaling, and subsequent load-balancing between replicas. In cloud computing, traditional autoscaling strategies are reactive scaling based on black-box system metrics, such as reaching CPU utilization thresholds [LBMAL14]. The performance of elastic applications in centralized datacenters is less sensitive to replica placement. Whereas in edge computing, where resources and clients may be dispersed in edge networks, autoscaling mechanisms should instead take into account where application requests are coming from, and autoscale based on demand, and place replicas close to that demand. Subsequently, requests should be routed to the application replicas in proximity. Classic round-robin load-balancing techniques may therefore be insufficient.

2.3.4 Privacy and trust

If personal data spaces turn out to be the dominant form of achieving privacy-aware data distribution, new methods for accessing data for training and inferencing purposes are needed. As people continue to gather personal information into their own space, they will be required to give third parties selective access to their data via complex access rules, or provide only anonymized views on their data via privacy-aware routing techniques. Privacy rules should be enacted in a context-aware way [LRD19a]. Only when people own their own data, the medium they are stored on, and can manage strict access controls, will they be in full control of their personal and sensitive data. As of today, we still put our trust in cloud providers to handle our data, even highly sensitive data such as health related records. Particularly problematic are the increasing number of surveillance cameras throughout public spaces. Although they provide exciting opportunities for visual analytics, situational awareness, or other AI applications, they are fundamentally a privacy concern. Researchers will be challenged to devise methods that reconcile data availability for AI agents, and guarantee data safety and the data owner's privacy. Security is another important factor. Consider a system where autonomous AI agents roam through the fabric to complete specific tasks [LRD21]. It is absolutely crucial that AI agents are safe from outside tampering and isolated from interference. We recognize that privacy and security mechanisms are a prime concern in edge intelligence systems, however they are out of scope of this thesis.

2.4 The role of serverless computing

Cloud computing models have evolved over the past decade to higher and higher levels of abstraction. In the original Infrastructure-as-a-Service (IaaS) model, users have access to virtualized server hardware through VMs, but need to maintain the leased VMs [EPM13]. In Platform-as-a-Service (PaaS) models, users only provide platform-specific application code, and the platform provider takes care of deploying and managing the application on their infrastructure. While PaaS provides more convenience for developers, the platforms are as such very narrow in their applicability. A more general-purpose way of computing without managing infrastructure is serverless computing [JSSS+19]. Compared to PaaS, serverless computing is not an integrated environment for a specific platform (like hosting web applications developed in a particular web framework), but rather a more general way of deploying and managing arbitrary code in the cloud. A major enabler for serverless computing is operating-system-level virtualization technology, where the deployment unit is a lightweight container (an isolated user space instance sharing the same operating system with other containers) rather than VMs. There are various manifestations of serverless computing. In a Function-as-a-Service (FaaS) offering, users upload their code as functions, and defines rules for triggering the function execution. The serverless platform takes the role of the dispatcher, and executes functions in response to events. Another manifestation of serverless computing is Container-as-a-Service (CaaS), that can be seen as a PaaS for containerized application deployments. Kubernetes is a popular container orchestration system, that is essentially a CaaS platform. There are many benefits to serverless computing. First, it provides complete infrastructure transparency. Application developers and operators do not know about the underlying infrastructure, while still benefiting from the scalability that autoscaling and load-balancing mechanisms provide. Second, the billing unit can be very fine-grained, allowing cloud providers to bill individual function invocations [Eiv17]. Third, platform provides can manage their infrastructure more efficiently, because containers. A drawback to hardware virtualization is the weaker isolation properties of containers.

A distributed compute fabric has the properties of PaaS offering. We want users to deploy and configure their code centrally, which has many administrative advantages, but code execution should be decentralized. At the same time, the infrastructure should be transparent to the user, and platform providers should take care of scaling the application. In [Sch19], we analyzed the characteristics of edge AI applications, and compared them in different programming models. These are summarized in Table 2.2.

Table 2.2: Characteristics of programming models for edge AI applications

Characteristic	Serverless cloud	Serverless edge	Local monolith
Internet usage	high: all data transmitted	low: code download	none
Responsiveness	low: WAN latency	high: LAN latency	high: no network
Runtime costs	high: traffic and execution	low if using on-premises resources	medium: powerful server required
Deployment	by cloud provider	mostly platform provider	self-managed
Code mobility	transparent execution in the cloud	transparent execution across cloud and edge	no mobility
Data privacy	low: data in the cloud	high: sensitive data stored in network	high: all data stored locally

Based on the discussion, we argue that serverless edge computing is the appropriate computing model. Edge computing introduces new challenges for serverless computing, which has lead to increased research interest in serverless edge computing [GND17, RHM$^+$19, BM19, ATC$^+$21]. There is an obvious tension between infrastructure transparency, which gives users the illusion of a *serverless* system, and the distributed and heterogeneous nature of the computing substrate. We discuss serverless computing and its role for edge intelligence in more detail in Chapter 6. Because edge intelligence emphasizes AI applications, we also argue that the computing model should elevate concepts from AI workflows we discussed in Section 2.2.2, as first-class citizens. In Section 6.3.2 we present a programming model that allows flexible general-purpose code execution, but simplifies the development of AI workflow functions.

2.5 Summary

Edge intelligence is a post-cloud-computing paradigm, in which AI applications leverage computational resources and data at the edge of the network, to push intelligence from the cloud closer to devices and end users. Its emergence can be attributed to the increasing interconnectedness of humans and Internet connected things, the latency and bandwidth requirements of IoT applications, the increased availability of data produced by edge devices, and the fast-paced development of edge AI methods and hardware. Edge computing is one of the key enablers of this new paradigm.

We outlined the role of the intelligent edge in the cyber-human evolution, and the ability of edge computing to enable the seamless augmentation of human cognition. We presented challenges that will confront edge AI systems for several years to come. Specifically, we have shown how edge computing exacerbates the complexity inherent to AI applications and ML workflows, and that new methods are necessary to leverage hierarchical edge/cloud architectures for the AI lifecycle. To enable the intelligent edge, we propose a distributed compute fabric, that comprises (1) a computing substrate that federates dispersed and heterogeneous resources from across the computing continuum, (2) sensing and communication middleware that self-adapts to dynamic edge computing environments, and (3) intelligent operational mechanisms that manage the complexity of operating applications on distributed edge infrastructure. The remainder of the thesis is dedicated to these three systems aspects, and how to evaluate them.

CHAPTER 3

Edge infrastructure architectures

The computing infrastructure for cloud computing is well established. Powerful server computers are consolidated into massive data centers interconnected with high-performance networking appliances. Cloud computing models, such as IaaS or PaaS, work well on top of these systems because they can build on assumptions about the capabilities of infrastructure and the network topology. Implementing generalizable models for edge computing is much more difficult, given the broad spectrum of available computing hardware for the equally broad range of edge computing use cases. A distributed compute fabric for edge intelligence will federate highly specialized hardware platforms, such as high-density small-form factor computers and embedded AI hardware, into cohesive multi-purpose edge computers. Together with cloud resources, hierarchies of connected clustered edge computers create a computing continuum. In this chapter, we solidify this vague idea with concrete technology proposals. The insights are particularly important for the design of the systems presented in Chapter 5 and Chapter 6.

We begin in Section 3.1 by examining the computing continuum, and present a critical discussion of existing computing platforms for edge computing. Through this examination, we identify a gap in existing state-of-the-art for supporting forward-deployed application scenarios, which we address with a system design and prototype for a portable high-density edge computer cluster, presented in Section 3.3. In Section 3.3 we study emerging application scenarios that leverage edge infrastructure in the entire compute continuum, with the goal to better understand the characteristics of edge infrastructure topologies, and elicit requirements for a distributed compute fabric.

3.1 The computing continuum

To better understand the challenges associated with developing edge computing platforms, in this section, we examine existing computing hardware, and the edge computing scenarios they are used in.

3.1.1 Computing platforms

Edge data centers that use server computers and are placed at the edge of the network are considered candidate infrastructure for many edge computing scenarios. In an early definition by Satyanarayanan et al. [SBCD09], a cloudlet is "a trusted, resource-rich computer or cluster of computers that is well-connected to the Internet and is available for use by nearby mobile devices." Since then, many other definitions have been put forth [LES+14, CRB+11, VSDTD12], but the consensus is that cloudlets aim to bring services otherwise hosted in the cloud closer to the edge of the network to improve Quality of Service for end users. Because these cloudlets use the same underlying infrastructure as data centers, namely racks and server computers, many tools and techniques from cloud computing are directly applicable. For example, cloudlets can serve as hosts for VM virtualization commonly used for IaaS platforms. However, because they are smaller than classic data centers, and may only contain a few computers, they can be much more easily deployed on companies' premises, or in public buildings throughout a city, forming essentially a type of CDN for computation. To serve backends for emerging wearable AR applications, which typically involve running AI workloads on video feeds at high framerate, cloudlets may also be equipped with specialized resources such as GPUs [SXP+13]. Edge data centers are also a key idea of mobile edge computing (MEC), where mobile operators place servers at base stations of cellular networks [BWFS14]. This allows operators to provide highly-responsive cloud services for mobile users based on their proximity to base stations, or optimize the battery lifetime of mobile devices by offloading compute tasks to these cloudlets.

Small form factor (SFF) computers such as Intel's Next Unit of Computing (NUC) platform with built-in CPUs, are used for edge computing scenarios due to their size, low energy footprint, and good performance. In terms of computing performance, they are comparable to a desktop PC, but can be significantly smaller. They are popular for compact desktop workstations, but are extremely versatile and have consequently found uses in many other areas. For example, the Ubuntu Orange Box is a portable compute cluster consisting of ten NUCs with Intel i5-3428U CPUs [VN], that can be used in a variety of offloading scenarios [CDOK17]. Their size also makes them viable for the use on drones. For example, NUCs have been used to enable UAV-based aerial cells that serve as data relays between mobile users and a backhaul network [BPB+19], or for performing computer vision tasks for autonomous landing of UAVs [CPSC16]. We later present our own SFF edge computer based on compact high-density server hardware, in Section 3.2.

Single Board Computers (SBCs) such as ARM-based Raspberry Pis are often associated with edge computing, as they can, e.g., function as edge gateways in IoT scenarios to read sensor data, perform data pre-processing or message relaying. Because they are very compact, cheap, and extremely energy efficient, they are applied in many applications, ranging from IIoT, smart city, to smart home scenarios. For example, in [CWC+18], SBCs serve as communication hub and data pre-processing gateway in an IoT-based manufacturing scenario. In the Array of Things project [CBSG17], which we describe in Section 3.3, sensor nodes deployed throughout a city are equipped with two SBCs, one serving as sensor processing device, the other as communication gateway. SBCs are attractive for this use case, as sensor nodes have to be compact enough to be mounted on traffic lights or lamp poles. Road Side Units (RSUs) that creat the vehicular network infrastructure (see Section 3.3), have similar characteristics and requirements. Another example is a smart home data-analytics application that we built, where an SBC is used to aggregate data from IoT sensors, trains a prediction model, and then serves the model for inferencing [Sch19]. Clusters of commodity SBCs have found a variety of applications [JBP+18]. A clear tradeoff with SBC is computing performance. Experiments we performed have shown that typical commodity SBCs perform roughly an order of magnitude worse than, e.g., a typical Intel CPU of an SFF computer. Increasingly they are being used as a platform for hardware AI accelerators such as Google's Edge TPU [Cas19], that can significantly increase the performance for application-specific workloads.

Mobile devices are much more difficult to pin down than the categories described so far. In pervasive computing, mobile devices often include smartphones, wearables AR devices, smart watches, or health monitoring devices [SGL19]. These are considered client devices, i.e., a user's gateway to pervasive computing. Mobile devices are mostly the source of offloading tasks to nearby edge resources [MB17], but have also been considered as a cluster resource to perform distributed computing [HAHZ15]. More recently, personal mobile devices have been used as part of federated learning workflows, for example to train Google's predictive keyboard app Gboard [HRM+18]. User devices perform training of partial models using their private keyboard logs, and send the trained models back to the cloud. This is similar to the hierarchical workflow we described in Section 2.2.2. Drones, small robots, and vehicle on-board compute elements could also be considered as part of the mobile device spectrum. In [SGL19] and related works, these devices are collectively referred to as the "Mobile and IoT device tier", in the context of a three-tiered device-edge-cloud hierarchy.

Modular AI capabilities AI workloads benefit significantly from specialized hardware. GPU clusters dominate the cloud-based ML landscape, and while the same type of GPUs could be used for edge data centers, they are impractical for many edge resources due to their size and power requirements. A popular example is the NVIDIA Tesla K80 that has a TDP of 300W. Instead, a new family of AI accelerators have emerged that promise to fully enable AI for resource constrained edge devices. The NVIDIA Jetson platform [NVI] provides small GPU computing devices with CUDA support, which can be attached to, for example, SBCs. These devices enable the acceleration of AI applications at the edge that require computer vision or deep learning workloads. Google has devised the Edge TPU[1], an application-specific integrated circuit (ASIC) that can enable, for example, high-fidelity, real-time vision applications running on an SBC. A number of other

[1]https://cloud.google.com/edge-tpu/

prominent examples include Microsoft BrainWave, which uses field-programmable gate arrays (FPGAs) [Bur17]; Intel Neural Compute Stick [Int18]; or Baidu Kunlun [Duc18]. However, as the variety of AI accelerators increases, so does the heterogeneity of the edge fabric, thereby introducing additional complexity for platform abstractions. Moreover, because ASICs are as such designed for specific algorithms, hardware vendors also control what algorithms and platforms we can use, and potential vendor lock-ins pose a significant threat to the democratization of AI. While we see pluggable AI capabilities for edge computers as the way forward in enabling a rich edge intelligence fabric, it is clear that this requires the continued fostering of open standards and robust platform abstractions.

3.1.2 Edge computer clusters

A goal of our approach is to create a seemingly uniform computing substrate, that is, in fact, made up of heterogeneous *edge computers*. We consider edge computers as the counterpart to server computers, expressly designed for the use in edge computing environments. We now discuss various aspects of clustering these edge computers, and how they form cohesive computing units.

Cluster-based edge resources Clustered edge resources of all shapes and sizes have been discussed and evaluated [PDT18]. These proposals range from Sun's Modular Datacenters that come in shipping containers[2]; over Canonical's (discontinued) Orange Box [VN], a cluster of Intel NUCs; to a cluster of single-board computers (SBC), such as Raspberry Pis, federated to form a micro datacenter [EPR$^+$17]. However, none of these come with an integrated power-management runtime, or other mechanisms for native energy-awareness. Although cloudlets are seen in general as either single-node or cluster-based systems, most existing evaluations of cloudlets in research consider only a single node, typically a commodity desktop PC [SBCD09, LES$^+$14]. We argue that this size of hardware (i.e., commodity computers) is useful for many use cases, as it is a compromise between the two previously presented extremes. However, we also argue that single-node systems are challenged to provide reliable services in the face of varying and unpredictable request arrival rates. A cluster of high-density compute nodes, for example Mini-ITX motherboards with server CPUs, can provide a great deal of computational power, while still being compact and portable.

General-purpose edge computers Most definitions consider a cloudlet to be static infrastructure, well-connected to the Internet, deployed statically at a specific location. Recently, researchers have shown that such cloudlets may be impractical for use cases that require on-premises decision-making for military field personnel in resource-constrained tactical environments [LES$^+$14]. To bridge this gap, they propose *tactical cloudlets*, i.e., portable cloudlets deployed on, e.g., vehicles that support computation offload in such resource-constrained environments. We argue that there are many more use cases to be considered that require portable cloudlets, or, more generally, general-purpose edge

[2]`https://docs.oracle.com/cd/E19115-01/mod.dc.s20/`

computers. We consider edge computers a specialized type of server computer, that is designed for the use in edge environments. For example, portable compute clusters could be deployed on emergency vehicles to allow complex decision-making and task-planning analytics applications in emergency response scenarios [LMFJ+04]. Other field-based work, such as archaeological dig sites, field experiments in geology, oceanography, etc., could all benefit from edge computing, but require portable and reliable edge computers. Companies are even exploring space-to-cloud analytics for remote regions using satellites[3].

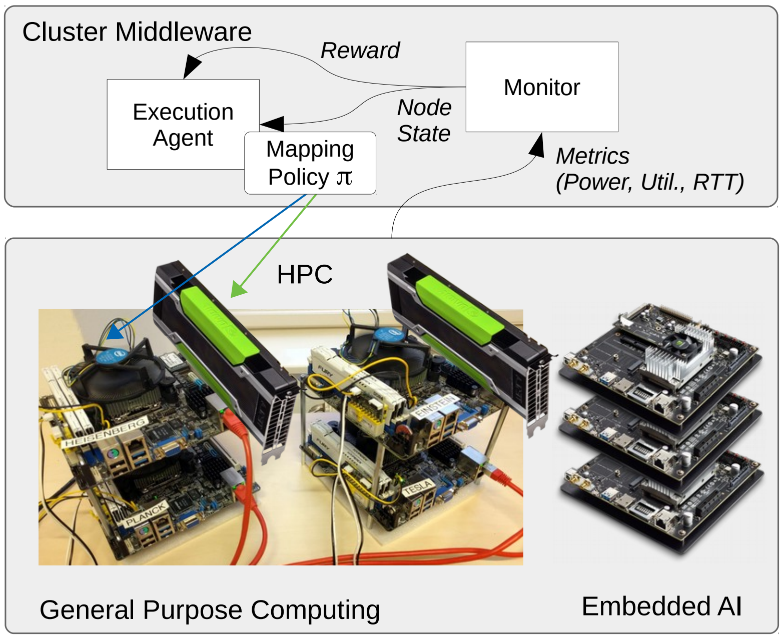


Figure 3.1: Multi-purpose edge computer with self-adaptive middleware that learns how to efficiently schedule workloads

Energy-aware portable edge computer clusters Infrastructure required for such scenarios faces numerous challenges. In particular, these edge computers work in energy-constrained environments, and may therefore have to be partially powered by secondary energy sources such as batteries. As we have discussed, single-node systems are challenged to enable scalability and reliability required for dealing with unpredictable workloads at the Edge. While statically scaling out the infrastructure would provide better service reliability, it also significantly increases energy consumption, thereby diminishing portability. However, by integrating power-management approaches for server clusters, such as vary-on/vary-off (VOVO) algorithms [EKR03], cluster-based edge computers could become the ideal architecture to address these challenges. However, these and other energy-efficiency mechanisms have been developed largely in the context of large-scale data-center infrastructure [MSS12, KKH+09, BB10], and typically depend on a model

[3]https://spire.com/

of energy consumption based on proxy metrics such as CPU utilization [WXZL11]. As we discuss in Section 3.2.4, our data indicates that these models are inaccurate for edge computers with modern, high-density, general-purpose hardware. With future developments and the integration of specialized hardware into these edge computers, such as GPUs or single board AI modules [Fra16], simple models will become infeasible for novel energy-efficiency mechanisms that control these heterogeneous infrastructures. Suppose, for example, a mechanism that optimizes energy consumption based on machine learning techniques such as reinforcement learning (which is also being explored in related cloud-operations research [WXZL11, FLP14]). A learning algorithm will continuously attempt different load balancing configurations given different workload types to minimize energy consumption. To facilitate such mechanisms in a real-world deployment, given the complexity of energy consumption of novel architectures, edge computers have to be natively *energy-aware*, i.e., they need access to energy consumption data in real-time to accurately determine the impact of different workloads on the energy consumption characteristics, learn from past observations, and continuously adapts to changes in the environment that may have an impact on energy consumption. Furthermore, control mechanisms need to be able to control the power state of nodes, i.e, turn them on and off.

3.2 Portable energy-aware edge clusters

A significant number of edge application scenarios can be characterized by the dynamic and resource-constrained environment in which supporting edge computers have to operate. For example, mobile applications are used by emergency response teams for on-premises decision making [LMFJ+04], or by military field personnel for image or speech recognition [LES+14]. Handheld devices stream data and offload compute-intensive tasks to nearby edge computers. Designing and operating edge computing infrastructure for and in these environments is challenging, as edge computers have to be portable to fit on, e.g., an emergency vehicle or a drone; deal with unpredictable client load; provide sufficient performance to host, e.g., data analytics or machine learning applications; and, at the same time, be energy-efficient to be powered by secondary power supplies such as batteries.

In this section, we present a design, prototype, and evaluation of a portable energy-aware, cluster-based edge computer that aims to address these challenges. It is portable because it is compact, and consumes energy at a scale that could be served by medium-sized batteries. It is energy-aware because it provides a power-management runtime to access energy consumption data in real-time, and control the power state of its nodes. Finally, it is cluster-based because it comprises multiple physical nodes to provide reliable and scalable computing. Furthermore, we present an experimental analysis of the energy and resource-consumption characteristics of our prototype in the context of a data analytics application. The results show the feasibility of our prototype for the presented scenarios, but also reveal the intricacies of power-management approaches already built into modern CPUs. We show that different load balancing policies and cluster configurations have a significant impact on energy consumption and system responsiveness. Our insights lay the groundwork for future research on energy-consumption optimization approaches for cluster-based edge computers.

3.2.1 System architecture

We propose an architecture for a general-purpose energy-aware, cluster-based edge computer. Later we present a reference implementation. In our architecture, a cluster-based edge computer is a closed system of n physical general-purpose compute nodes, and a power-management runtime that enables energy-awareness. We focus explicitly on enabling energy-awareness as opposed to general deployment and management of applications, as solutions already exist for these aspects [LES+14, SBCD09]. Figure 3.2 shows a simplified view of our proposed architecture.

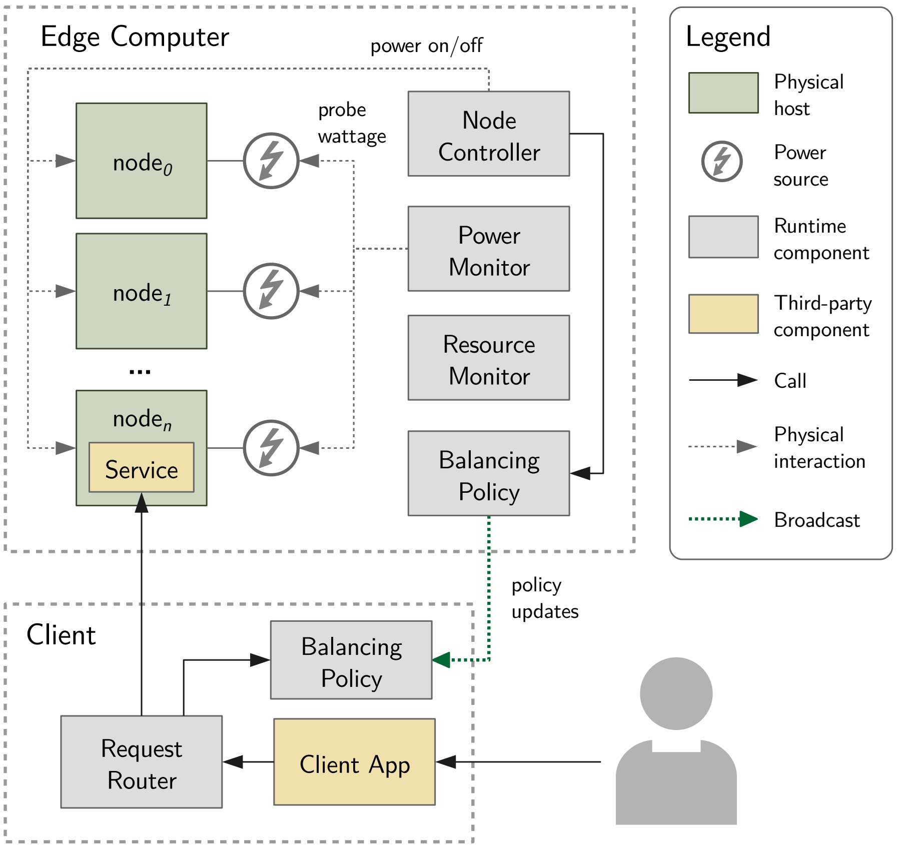


Figure 3.2: Architecture for an energy-aware cluster-based edge computer

Nodes host third-party services that are deployed via, e.g., container-based virtualization. The power-management runtime comprises components to monitor and control the nodes, and to provide energy-aware optimization mechanisms for task scheduling or load balancing. These runtime components are hosted on a separate auxiliary device that is part of the edge computer, but independent of power-management mechanisms and much smaller in scale than the compute nodes to minimize the impact on the overall power consumption (in our prototype we use two SBC, see Section 3.2.3). Between each node's power source, a power sensor is placed that allows the power monitor component to probe the current wattage of each node. A client device hosts the client side of the runtime: a request router that forwards requests to an actual physical node hosting the service, as well as third-party client application.

Power-management runtime

The power-management runtime comprises the following core components:

Node controller the node controller decides at runtime, based on values received from the power monitor and resource monitor, to modify the power state of nodes, i.e., powering them on or off. The node controller relies on an optimization strategy implementation that aims to minimize energy consumption while maintaining responsiveness of the system. When the controller changes the power state of a node, it informs other runtime components about these changes.

Power monitor the power monitor probes the nodes' power sensors, and provides an API for other runtime components to access node energy consumption data in real-time.

Resource monitor the resource monitor tracks the resource utilization of nodes, and also provides an API to access the data in real-time. Specifically, it provides data on the CPU frequency (which may be dynamically adjusted by the underlying hardware), CPU utilization, memory usage, or I/O bandwidth of nodes.

Balancing policy the balancing policy aims to optimize energy consumption and resource utilization by distributing request load across physical hosts. It uses the power and resource monitoring data to intelligently balance service requests among physical hosts. Because request dispatching actually happens on the client, the runtime broadcasts changes in the balancing policy to the client-side balancer. For example, when the node controller changes the power state of a node, the balancing strategy is informed and adapts accordingly (e.g., by removing a turned off node from the pool), and then forwards the information to the client.

Client

The client hosts the third-party client application and the client-side components of our power-management runtime. A client device could be an edge device such as a smartphone or an SBC.

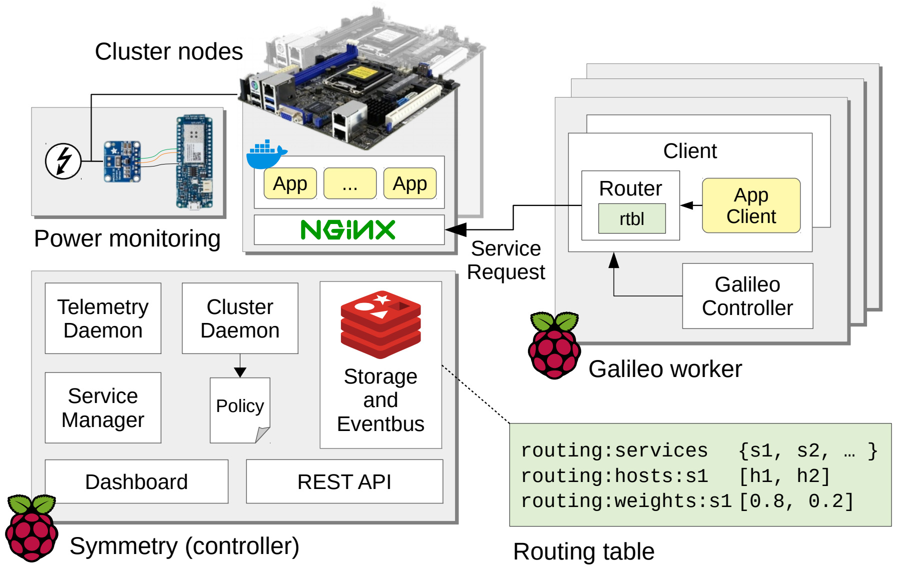
Request router In traditional cloud-based clusters, load balancing is typically done in the cloud via, e.g., L7 switches or reverse proxies hosted on powerful machines that forward requests to the actual services depending on some strategy [GB14]. We cannot rely on data-center scale routing hardware in a small portable edge cluster, and dedicating a node of the cluster to serve as reverse proxy would require this node to be online continuously, and therefore significantly increase the overall energy consumption. Furthermore, for high-performance applications, a load balancer that has the same computational resources as the nodes it manages would quickly create a bottleneck. Instead, in our architecture, we offload request dispatching to the client. The request router on the client device is a proxy for an actual service request and is responsible for dispatching a request from the client app to a node that hosts the service. The runtime broadcasts changes in the balancing policy to the client-side request router. This includes the network addresses or specific service endpoints of currently active nodes. The requests are routed based on the current active balancing policy dictated by the edge computer, e.g., the percentage of requests that should be routed to a specific node or service endpoint. This way, the energy cost of load balancing is decentralized among clients. A drawback of this approach is the increased management complexity of balancing policies.

Balancing policy The request router relies on a balancing policy defined by the power-management runtime. For each request, the request router queries the balancing policy for a node to send the service call to. A balancing strategy could simply be a round-robin or weighted distribution among all currently powered nodes. Implementing and evaluating more complex strategies is out of scope of this paper, but part of our future work.

3.2.2 Symmetry: a reference implementation

We describe *Symmetry* – an end-to-end implementation of the system architecture we presented, that addresses the described issues. Symmetry takes the role of the cluster controller, and is designed to run on SBCs like a Raspberry Pi. It features service management on top of Docker, runtime monitoring of black-box metrics, power draw, and application latency, client-side load balancing, and dynamic cluster reconfiguration mechanisms. We provide load balancing and cluster reconfiguration policies such as round-robin or reactive autoscaling, but also give developers tools and APIs to build their own. Figure 3.3 shows all the components in an example deployment on our hardware prototype presented in Section 3.2.3 The figure also shows how Galileo clients (see Section 4.1), used for distributed load testing experiments, integrate with Symmetry.

Systems that solve similar problems in data-center scale clusters [GHBRK12] are not applicable to the domain of portable energy-aware cloudlets for reasons we have outlined in a previous publication [RAD18]. First, the operational scale impedes the use of components typical in cloud architectures, such as dedicated L4 or L7 load balancers. Second, the models used for energy management in data centers often build on assumptions that do not hold for smaller scale hardware that already has very effective built-in energy



```
routing:services   {s1, s2, … }
routing:hosts:s1   [h1, h2]
routing:weights:s1 [0.8, 0.2]
```

Figure 3.3: The Symmetry system components and their interactions

management. Third, resource management systems for cloud clusters, such as OpenStack or Kubernetes, are often very resource intensive in and of themselves and require powerful servers to operate.

Symmetry comprises a set of lightweight Python components that together make up the system control software. Cluster nodes are server computers that host services, and require no software other than Docker and an SSH server. The system is designed to manage compute clusters with around 2–20 nodes, such as a cluster prototype we have presented, or an Ubuntu Orange Box, and manage stateless services such as image recognition models, database query serving, or similar applications.

Command line interface and REST API

Operators interact with Symmetry via a CLI to deploy services to the cluster or activate specific load-balancing policies. REST APIs provide ways to control the cluster state and request routing, and enable the modular development of additional operational logic. A runtime dashboard provides insights into the current cluster utilization, power draw, and application performance.

Symmetry core

The core platform component of Symmetry is a Redis instance running on the cluster controller. All layers of Symmetry are integrated via this Redis instance, which facilitates both data storage (such as service metadata) as well as inter-process communication using the eventbus architecture of PyMQ[4]. Due to its lightweight design and highly optimized I/O functionality, Redis performs extremely well in this scenario.

Service management

Applications are hosted on cluster nodes as HTTP services in Docker containers. Symmetry starts on each cluster node an NGINX instance to internally route requests to the correct container and to monitor application performance. Services are described via YAML files that specify necessary metadata. The CLI command `symmetry deploy my-service.yml` then deploys the service to each node, starts the necessary containers, registers the service endpoints, and updates the node's NGINX config accordingly.

Telemetry daemon

The telemetry daemon aggregates runtime metrics from various sources and publishes them as time series data into a pub/sub topic that encodes the node and the metric, for example, `telemetry:node1:cpu`. It implements pull-style monitoring by connecting to the cluster nodes via SSH and executing commands for measuring CPU utilization, CPU core frequency, or parsing the NGINX logs. For power data it connects to an Arduino that provides access to readings from integrated current sensors. If a node shuts down, it informs other components via the eventbus.

Cluster daemon

The cluster daemon enacts the load balancing and cluster reconfiguration policy. One policy that Symmetry provides out-of-the-box is *Reactive Autoscaling*, which activates or suspends a node if a system metric exceeds a given threshold for a specified amount of time. For example, our default implementation activates an additional node if the average CPU utilization is above 85% for more than 10 seconds, and suspends it if drops below 25%.

Client-side request routing

Making clusters appear as a single system is typically achieved via dedicated L4 or L7 load balancers, through which all service requests are routed. Instead, we take a client-side request routing approach that uses simple weighted-random load balancing, where weights are updated dynamically by the load balancing policy in a way that meets some operational goal. A routing table specifies for each service how much of the workload should be directed to a given node. Updates to the routing table are propagated to the clients via Redis pub/sub. This way, request routing is decentralized, but simplified such that the client components necessary to call services are kept simple.

[4]`https://github.com/thrau/pymq`

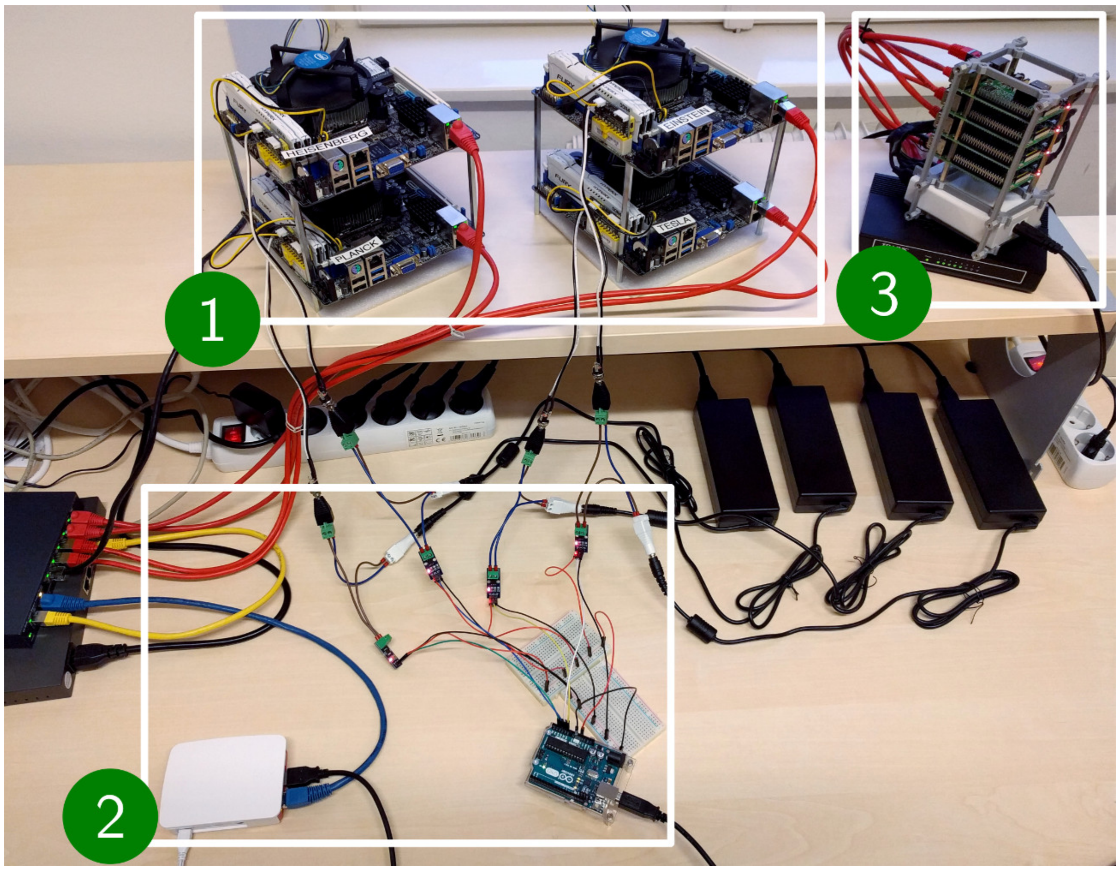


Figure 3.4: Energy-aware cluster-based edge computer prototype

3.2.3 Hardware prototype

Based on our architecture, we have developed a prototype for an energy-aware portable cluster-based edge computer. The prototype comprises four compute nodes, networking and power infrastructure, a power and resource monitoring system, and a small cluster of client devices. Figure 3.4 shows our prototype infrastructure. It shows (1) the cluster compute nodes, (2) the monitoring infrastructure (with the Raspberry in the bottom left corner in a white case, and the Arduino in the bottom right), and (3) Raspberry Pis that serve as clients. Next, we describe each component in more detail.

Compute nodes

The overall design goal of our prototype is to enable high-density computing, i.e., to have good compromise between compactness that enables portability, and performance to host typical cloud-based services. We use Mini-ITX form-factor hardware, with server capabilities in mind, to build a high-density cluster. Each node is a mid-priced but powerful server computer made up of the following components:

- Motherboard: ASUS P10S-I Mini-ITX[5]
- CPU: Intel Xeon E3-1230 (4 cores, 8 threads)[6]
- RAM: 2x16GB Kingston HyperX Fury DDR4
- SSD: Intel SSD 600p 128GB M.2.[7]
- Power supplies: picoPSU-90 12V

Overall, each node takes up roughly 17x17x10cm of space.

To host applications, the compute nodes run a common server operating system, namely CentOS 7[8], with a standard configuration. Furthermore, the nodes run Docker CE[9] as application deployment platform. We later present energy consumption data to show that this hardware configuration could be powered by commodity battery packs.

Power-management runtime

There are many ways to implement the power-management runtime described in Section 3.2.1. Specifically, there are a variety of ways to monitor energy consumption. Although we do not require real-time access to the energy consumption data for the evaluation of this prototype, we wanted a portable and cost-efficient solution to show the feasibility of our design, in particular because the power-management runtime is an integral part of the system and should therefore be portable and small scale. We deploy the power-management runtime on a Raspberry Pi Model 3 B running Raspbian 9. To monitor energy consumption, we developed a compact and cost-efficient monitoring infrastructure using an Arduino Uno, and four ACS712[10] Hall-effect-based linear current senors. We later revised the prototype to use digital current sensors which are less prone to noise. Specifically we use modified Adafruit INA219 [11] DC current sensors. It is important to note that we measure between the picoPSU power supply and the AC adapter, because we intentionally do not want to include the power dissipation of the adapter (which we found in experiments to vary greatly between 10–25%). This is also the reason why we do not use consumer-grade power meters that typically sit between the power socket and the AC adapter. Reading the sensors is done via the analog or digital inputs and a simple Arduino program. The Raspberry provides the Power Monitor, Resource Monitor, and Node Controller component. The Power Monitor

[5]https://www.asus.com/Commercial-Servers-Workstations/P10S-I/
[6]https://ark.intel.com/products/52271/Intel-Xeon-Processor-E3-1230-8M-Cache-3_20-GHz
[7]https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/600p-series/600p-128gb-m-2-80mm-3d1.html
[8]https://www.centos.org/
[9]https://www.docker.com/
[10]https://www.allegromicro.com/en/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS712.aspxx
[11]https://www.adafruit.com/product/904

accesses the power readings via the Arduino at a specified sample interval, and calculates from the raw readings the effective power in Watts. The Resource Monitor uses standard Linux facilities to read resource utilization data from the nodes such as `/proc/stat` or `/proc/cpuinfo` which gives details about the current frequency and idle state of CPU cores. We later extracted the monitoring tooling into the Galileo experimentation and analytics framework (see Section 4.1.

3.2.4 Performance characteristics

To further drive the design of control mechanisms for edge computer clusters, we first want to understand their performance and energy consumption characteristics. To that end, we run several experiments to gather data on the general energy consumption characteristics of the cluster, as well as concrete performance data when running a data analytics application in different cluster configurations. We use the power-management runtime prototype to collect data on the current energy consumption and resource utilization of each node, and to test different balancing policies during real-world application workloads.

Profiling experiments

We examine the system's energy consumption and responsiveness under varying client load and different load balancing policies. The main goal is to examine whether the overall system's energy consumption can be improved with specific load balancing policies given otherwise identical experiment parameters. Furthermore, we are interested in the tradeoff between energy consumption and system responsiveness.

Application scenario For the evaluation, we consider a typical video analytics application [SSX+15] in a forward deployed scenario [LES+14]. For our application we use the Apache MXNet deep learning library [CLL+15] with pre-trained models to perform image recognition tasks. We use Docker to deploy instances of an MXnet Model Server[12], which exposes an MXnet classifier as a web service via an NGINX[13] web server. We then use a simple Python HTTP client application that we developed to send images to the exposed endpoint and await classification results. The clients receive from the power-management runtime the network addresses of endpoints and a weighted load balancing policy. We use the pre-trained SqueezeNet [IHM+16] model, a small-footprint model that has been used in other evaluations of image recognition applications [IMS18]. The workload is representative of many of the use cases we described in Section 2.2.

Emulating clients To generate workload, we set up a small Raspberry Pi cluster where each node hosts a client application and a tool for generating load. We describe this in more detail in Section 4.1. The client cluster is connected via a separate switch to the LAN of the cluster, and is powered by a separate power source. The Raspberry Pis are Model 3 B Rev 1.2 running Raspbain 9.

[12]https://github.com/awslabs/mxnet-model-server
[13]https://www.nginx.com/

Table 3.1: load balancing policies for experiments

Exp.	n_1	n_2	n_3	n_4
1	100%	-	-	-
2	90%	10%	-	-
3	80%	20%	-	-
4	70%	30%	-	-
5	60%	40%	-	-
6	50%	50%	-	-
7	33%	33%	33%	-
8	25%	25%	25%	25%

Experiment setup We stress nodes by sending image classification requests from our Raspberry Pi clients. A client is a simple Python application that spawns several worker processes that send a randomly selected image (pre-loaded into memory) to a node and wait for a response. To make sure the work required for each request is the same, we pre-load five similar images, each with dimensions around 300x220 pixels, and a file size of around 150kB. In exploratory experiments we found that with these parameters, four Raspberry Pis each running eight worker processes to send requests can easily saturate a node's request capacity without slowing down the clients. For each request, we measure the round-trip time (RTT) in milliseconds with which we quantify the responsiveness of the system. To control the request rate we use a load generator that generates a pyramid arrival pattern, starting from 1 request per second (r/sec), to a peak of 300 r/sec, and down to 1 again. The load is adjusted every 20 seconds, and the experiment runs for 20 minutes.

In total, we run eight experiments with which we examine the system's performance and energy consumption under different weighted random load balancing policies. The first experiment sends all requests to n_1. The second experiment balances load between two nodes n_1, n_2 in a ratio of $\frac{n_1}{n_2} = \frac{.9}{.1}$, the third $\frac{n_1}{n_2} = \frac{.8}{.2}$, and so forth, until $\frac{.5}{.5}$ (round-robin scheduling). In the last two experiments we perform round-robin scheduling with three and four nodes respectively. Table 3.1 summarizes this. The columns indicate the percentage of requests that are routed to the respective node, or whether the node is kept offline.

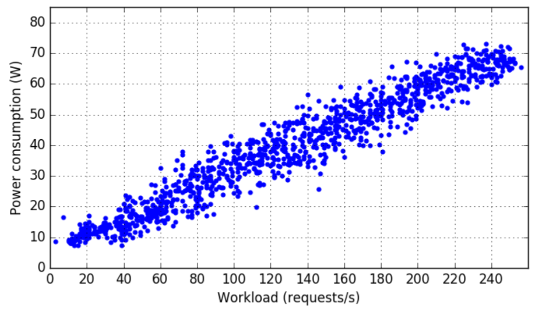
Experiment results

We calculate for each experiment the total energy required for each node, and report the CPU utilization, CPU frequency (dynamically adjusted by the CPU), and statistics on the RTT. Table 3.2 shows the results. We performed the workload experiments six times with similar results. Instead of aggregating the results, which would hide nuances of the experiment runs, we report a representative sample. Column 1 shows the index of the experiment. Column 2 shows the energy consumed in Wh over the entire course of the experiment for each node and in total. To get a more representative result, we list for offline nodes a value that corresponds to the average offline energy consumption over that period of time across all nodes. Column 3 shows the CPU utilization of each online node over time during the experiment. Column 4 shows the corresponding CPU frequency, where the value is the sum over all cores in MHz (8 cores with hyper-threading, where each core has a maximum frequency of 3700 MHz). Column 5 shows the request RTT in milliseconds (mean, 90th percentile, and 99th percentile) over time during the experiment. Note the y-axis scale changes from experiment 5.

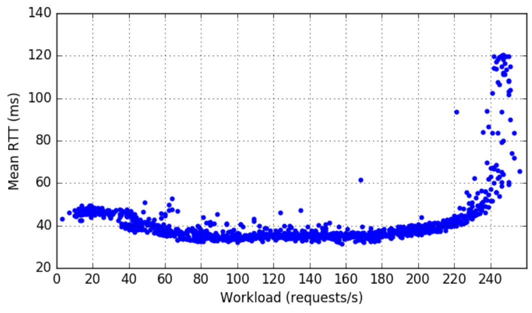
Energy consumption Although experiment 1 shows the lowest energy consumption, it should be noted that a single node was not able to handle the peak load of 300 r/sec, but instead was capped at 250 r/sec, which should be taken into consideration when interpreting the results. Looking at the sustained peak of CPU utilization and RTT after minute 8 shows that the node was busy working a congested request queue, and was able to pick up with the reduced request rate at minute 13. Experiment 2 exhibits similar but less extreme characteristics. In terms of energy consumption, overall we can see that there are only slight variations between the different load balancing policies in experiments 2 through 6. We also observe that adding a new node in experiment 7 does not increase the energy consumption significantly. Looking at the data, we can see that the CPU load is balanced across nodes, which has a cumulative effect on the energy consumption reduction of individual nodes, and this reduction is higher than the cost of adding a node. However, there are diminishing returns: at some point the energy cost of powering on additional nodes will be larger than the benefits of balancing load further, as made evident by experiment 8 (4 nodes round-robin scheduling), where the energy consumption is significantly increased compared to the other experiments.

Responsiveness Overall we can see that, unsurprisingly, the RTT and system responsiveness is generally improved when balancing load between nodes. However, our results reveal interesting insights on the effect of dynamic frequency adjustment on the overall system responsiveness. For example, the first minutes of experiment 6–8 show that, when load is balanced in a way that nodes become underutilized, the reduced CPU frequency will lead to a worse responsiveness, despite more nodes being used than in experiment 6. Figure 3.5 shows this behavior more generally.

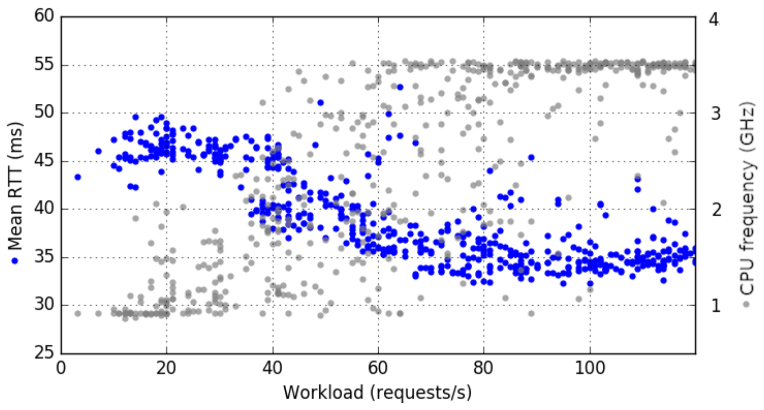
Power consumption vs CPU utilization Assumptions often made when building simulation models for energy-efficiency algorithms in cloud computing are: a) that CPU utilization maps directly to energy consumption [WXZL11, BB12], b) that there is a linear relationship between CPU utilization and energy consumption [KKH$^+$09, MSS12, PDCB15], and c) a low peak-to-idle energy consumption ratio [EKR03, WXZL11, BB12]. These assumptions may be valid for data-center scale servers [KKH$^+$09], but our results indicate that these assumptions will lead to inaccurate results when applied to edge computers. Figure 3.6 shows a scatter plot of the CPU utilization and energy consumption measurements of experiment 3 and node 1. First, we observe that the relationship is more complicated. Until a CPU utilization of 12%, the energy consumption only rises



(a) Power draw against workload

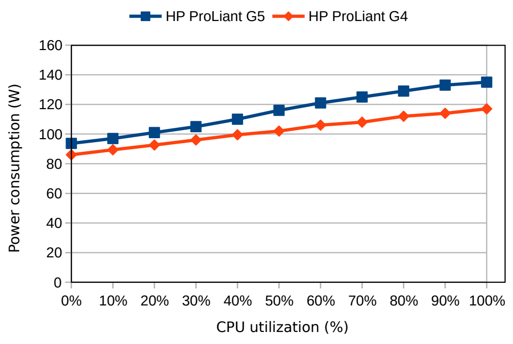


(b) Mean application RTT against workload



(c) The effect of CPU frequency scaling on application responsiveness

Figure 3.5: Results showing the tradeoff between responsiveness and energy consumption



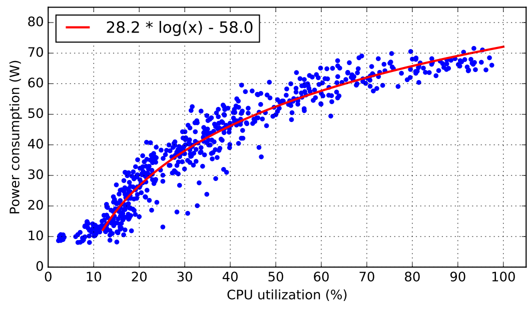
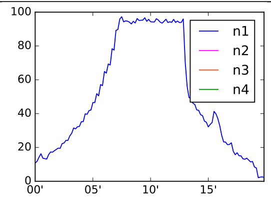
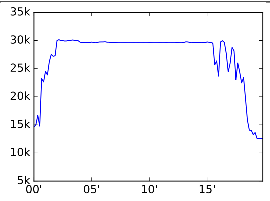
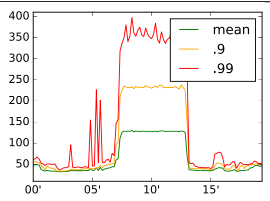
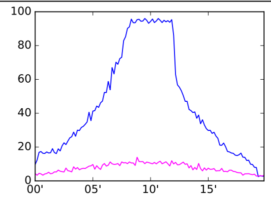
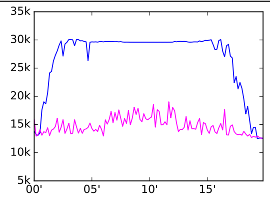
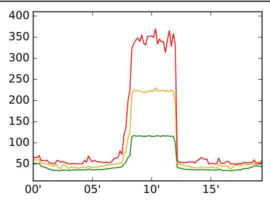
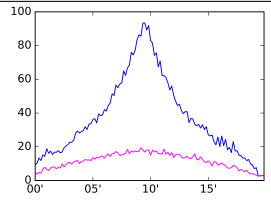
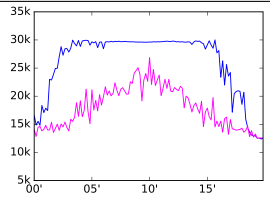
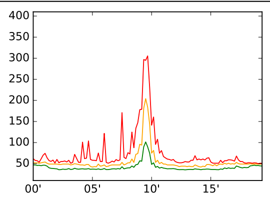
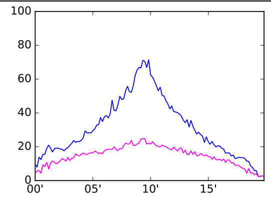
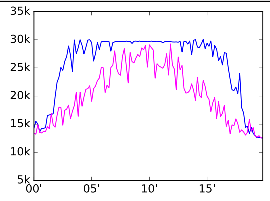
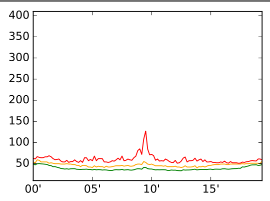
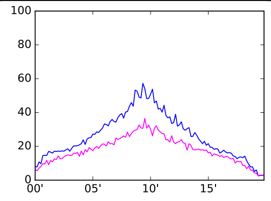
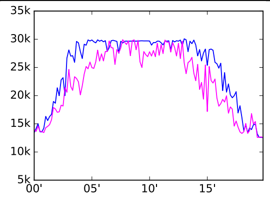
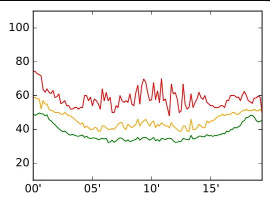
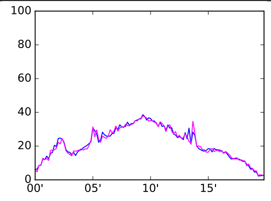
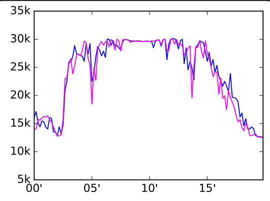
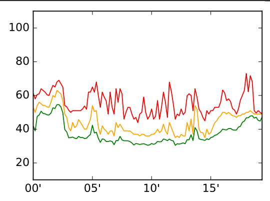
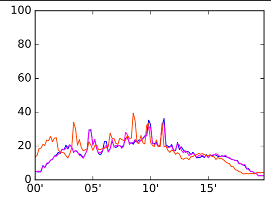
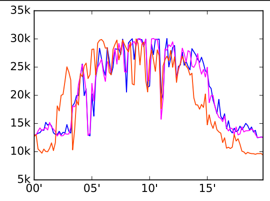
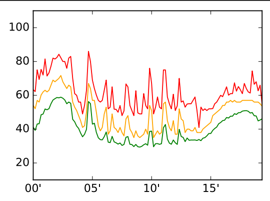
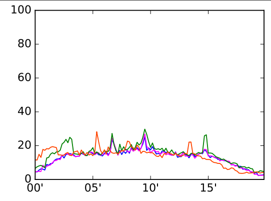
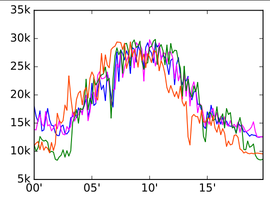
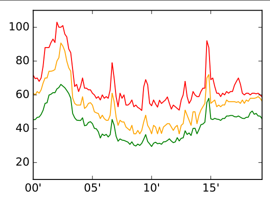


Figure 3.6: Relation between CPU utilization and energy consumption according to [CRB+11] (right), compared to our results of running an image classification service on our high-density edge computing cluster (left)

Table 3.2: Results overview of load experiments in different cluster configurations

#	Energy (Wh)	CPU Utilization (%)	CPU Frequency $\sum MHz$	RTT (ms)
1	n_1 15.063 n_2 0.644 n_3 0.644 n_4 0.644 $\sum$ **16.995**			
2	n_1 14.071 n_2 4.039 n_3 0.644 n_4 0.644 $\sum$ **19.398**			
3	n_1 12.775 n_2 5.084 n_3 0.644 n_4 0.644 $\sum$ **19.147**			
4	n_1 11.491 n_2 6.534 n_3 0.644 n_4 0.644 $\sum$ **19.313**			
5	n_1 10.165 n_2 7.922 n_3 0.644 n_4 0.644 $\sum$ **19.375**			
6	n_1 8.797 n_2 9.326 n_3 0.644 n_4 0.644 $\sum$ **19.411**			
7	n_1 6.477 n_2 6.924 n_3 5.345 n_4 0.644 $\sum$ **19.391**			
8	n_1 5.423 n_2 5.856 n_3 4.454 n_4 5.794 $\sum$ **21.527**			

very slightly. The relation then exhibits logarithmic growth. Second, we observe that for some regions of utilization, the energy consumption variance is very high. For example, around a utilization of 16%, the energy consumption varies between 9W and 30W, and at 90% it only varies between 63W and 70W. Third, we observe that the peak-to-idle energy consumption ratio (at around $\frac{70W}{10W} = 7$) is much higher in our prototype compared to what is assumed for data-center scale hardware (e.g., 1.36 or 1.45 [BB12]).

3.2.5 Discussion of results

An active node in our cluster draws power at a rate between 10–80W, depending on the resource utilization. Even when shut down, a node draws power at a rate of 0.5–2W. This large margin in energy consumption between idle and fully utilized nodes is due to the power management mechanisms of modern CPUs, in particular voltage and dynamic frequency adjustment. What this means for, e.g., simulation environments, is that using CPU utilization as a proxy for energy consumption will lead to inaccurate results. It is crucial to account for adjusted CPU frequencies based on the current utilization. Even then, estimates of energy consumption based on performance indicators alone may not be accurate. Because energy consumption is a complex process dependent on many different factors that are difficult to accurately measure in their entirety, different type of workloads may yield very different energy consumption patterns [GOM18], which makes it necessary for power-management mechanisms to have real-time access to energy consumption data. Further investigation is needed to understand these factors, and how for example, integrating specialized hardware affects the overall energy consumption behavior. Overall, our results show that our prototype requires energy at a scale that can be managed with commodity portable energy supplies. For example, to get a rough idea, a typical 20 Ah lead-acid battery at 12V corresponds to 240 Wh. In our 20-minute experiment the cluster processed around 190 000 requests, and consumed roughly 20 Wh, with a power draw of about 80W if we consider the additional infrastructure energy consumption (switch and router, which averages at 4.1W). Even if we account for Peukert's law and the resulting diminished effective capacity, a single battery of this type could power the system for over an hour in the experiment conditions. This indicates that the general scale of the cluster is feasible for the application scenarios we presented.

Cloud operations research has traditionally assumed high peak-to-idle energy consumption ratios of compute nodes [EKR03, WXZL11, BB12] Our initial intuition was therefore that the idle energy consumption of nodes would be so high, that the optimal configuration would be to fully utilize each node until the application responsiveness drops below a specific threshold (e.g., a maximum RTT value). Our data show the opposite, which we largely attribute to the power-management mechanisms of high-density compute hardware, i.e., dynamic frequency adjustment, and the relationship between CPU frequency and energy consumption. When load is balanced among cores, the CPU utilization is low, and therefore the frequency is scaled down. There is a delicate tradeoff between CPU frequency, energy consumption, and responsiveness, which greatly increases complexity for energy-aware load balancing or scheduling techniques, and is likely to have different characteristics for different types of applications and workloads. Energy-efficiency systems for clusters need to understand the behavior of the underlying hardware's power-management algorithms, and cooperate with these lower-level mechanisms to provide optimal balancing of workloads. These are particularly important observations for future work on building clusters of hardware that already makes use of energy efficiency techniques (such as dynamic frequency scaling), but further investigation is needed to fully understand this relationship with respect to different workload types and node configurations.

3.3 Edge infrastructure scenarios

In edge computing, there is not a single well established infrastructure model [PDT18]. The infrastructure for cloud computing is the data center, where servers are organized into racks, and interconnected using high-performance networking, such as fiber optic 100GbE. Whereas in edge computing, the infrastructure used for computation, data storage, and networking, will depend greatly on the use case and the deployment environment [SGL19]. This complicates the development of generalizable computing models and platforms that work to hide the underlying computing infrastructure from the application. In this section, we outline several edge computing scenarios, and examine their infrastructure and network topology. The goal is to identify commonalities, and to better understand the challenges of building a computing substrate we described in Section 2.3.

3.3.1 Urban sensing

Urban sensing is a smart city concept, to provide citizens and governmental parties with environmental and contextual urban data. More and more cities [SSS$^+$17, CZVZ14, Nun05] deploy IoT nodes with cameras and sensing capabilities into urban areas to enable a variety of applications. Data can be used for urban monitoring applications; to create data analytics models such as crowd behavior, flooding models, or accident risk prediction [Bec18]; or to enhance human cognition (see Section 2.1). An example is the Array of Things (AoT), a networked urban sensing project initiated in Chicago, which aims to provide access to real-time data on urban environments [CBSG17]. Each node is equipped with two Single Board Computer (SBC) devices, sensors, and a camera. Nodes are deployed throughout Chicago's 77 neighborhood areas, e.g., mounted on lamp poles or mobile base stations, and networked with, e.g., cellular networks, Wi-Fi, or wired uplinks. The initial deployment consisted of 40 nodes, and has been increased as of Spring 2019 to about 200 nodes. One use case we described in Section 2.2.2 shows how edge computing resources can be used to offload compute-intensive tasks such as analyzing video feeds. It is reasonable to assume that future version of smart city sensor nodes may also contain embedded AI hardware (see Section 3.1), effectively turning nodes into multi-purpose edge computers we described in Section 2.3.2. With the appropriate middleware, many such nodes could then be used as a cluster. Locality plays an important role in resource management techniques for such a cluster, as the density of nodes per district, and their utilization, depends on the city's structure and urban dynamics. We examine the density of deployments in more detail in Section 4.2.

3.3.2 Industry 4.0

Edge computing is considered a key component of realizing Industry 4.0 concepts such as smart manufacturing or the Industrial IoT (IIoT) [CWC$^+$18]. Consider an international manufacturing company that manages several factories. To facilitate IT/OT convergence, factory floors are equipped with edge computing hardware, IIoT gateways, SBCs as sensor aggregators for soft real-time analytics and preprocessing, embedded AI hardware for low-latency video processing (e.g. for high-speed manufacturing lines), and small form-factor computers as on-site workstations. These are plausible extensions to the prototypes presented in [CWC$^+$18], and the general trend towards using embedded AI hardware for using sensor and video analytics in predictive maintenance scenarios [YMLL17]. On-premises edge data centers (or cloudlets) with varying resource capabilities provide additional IT infrastructure for various compute loads and services. These can be potentially provider-managed, i.e., an on-premises cloudlet that is billed in a pay-as-you-go fashion. Premises are interconnected through the public internet, backed by public cloud services. The infrastructure forms a federated edge-cloud environment that includes resources from across the computing continuum.

3.3.3 Telco-operated mobile edge clouds

Multi-access Edge Computing (MEC) allows third parties to deploy applications on a Mobile Network Operator's (MNO) edge DCs. MEC standardization efforts driven by ETSI include a MEC reference architecture [ETS19]. However, MEC topologies highly depend on the operator's dimensioning decisions, and different candidate locations for edge DCs exist. Installing compute infrastructure directly at base stations would offer significant latency benefits, but may be hindered by space limitations, especially in urban areas. Another option is to deploy edge DCs at the MNO's central offices (CO) [PAA$^+$16], which aggregate traffic from base stations over high-capacity and low-latency links. This performs well in terms of latency and has advantages from a DC "real estate" perspective. A third scenario is to push MEC to core network DCs, co-locating it with the Packet Gateway (P-GW). This is technically the simplest option for legacy 4G networks [HAW17], at the cost of increased latency and reduced traffic offloading gains. Data of real-world MEC deployments is often proprietary and therefore hard to obtain. However, plausible figures per operator include 2-3 core DCs, hundreds of COs, and tens of thousands of base stations country-wide [Tel19]. Current literature corroborates these numbers. For example, Basta et al. [BBH$^+$17] assumed 3 and 4 P-GWs (and thus core DC locations) for Germany and the US, respectively, and Peterson et al. [PAA$^+$16] report that, as of 2016, AT&T was operating 4700 COs. Base station approximate locations and densities can be obtained from crowdsourced LTE coverage maps [opec, opeb, Aus]. For example, in Section 4.2 we show that, as of 2020, there were around 1747 mobile base stations in Vienna, Austria, and the per-district deployment density followed a log-normal distribution. The resources available in edge DCs will also vary. For example, in the techno-economic analysis of [5G-19, Section 2.3.3.2], a small edge DC was assumed to have a capacity of 576 CPUs, 1512 GB RAM, ~40 TB storage, and ~1000 Gbps network capacity, adding a 5 ms latency.

3.3.4 Vehicular networks

Services for connected vehicles, including safety, infotainment, and advanced driving assistance, are key drivers for the automotive industry, and can particularly benefit from edge computing. A typical architecture for vehicle-to-everything (V2X) services includes roadside units (RSU) communicating with vehicles' onboard units using wireless technology, and connected with an optical or wireless (e.g., cellular) backhaul to aggregation points, centralized DCs and the Internet. RSUs are similar to the urban sensing nodes we described in the Urban Sensing scenario, and are candidates for co-location with edge computing hosts for low-latency vehicular services. RSU deployment follows service requirements (coverage, message dissemination latency, reliability). The InterCor project [intb] recommends [DPVW+17] that an RSU should be present at every highway entrance and at least every 6 km. The 5G Automotive Association (5GAA) estimates [5GA19] that the inter-RSU distance will range from 300 m to 1 km, depending on the road type. Due to physical constraints (e.g., an RSU may be mounted at a traffic light in urban environments), the capacity of an edge host deployed at an RSU may be limited, down to that of an SBC or small form-factor node. However, in the Smart Highway testbed (Antwerp, Belgium) [MBLN+19], some RSUs (less than 1 km apart) are equipped with server-class edge compute nodes, connected to the cloud via fiber optic links.

3.3.5 Edge cloud federation

It is common for multi-national companies to implement a hybrid cloud strategy for their IT infrastructure [Wei16], where on-premises IT resources are federated with public cloud providers across several cloud regions. An example are CDNs to provide low latency access to web resources at the edge. Public data sets from production cloud systems, such as the Google Borg cluster traces [VPK+15], provide insights about cluster configurations in public clouds. The median size of a cluster at the time was reported to be around 10 000 nodes, and clusters were reported to be heterogeneous in terms of RAM, CPU, and networking, but have no specialized compute capabilities. Bandwidth within a data center and cross-region vary greatly depending on the provider, but results of a recent benchmark on cross-region traffic of AWS [Cut18] suggest 10 GBit/s within a region, and 1 GBit/s cross-region. On-premises clouds are connected via public internet, where bandwidth can range from 100 MBit/s to 10 GBit/s depending on the service provider. Latencies between Internet backbone regions are fairly static and could be drawn from public data sets. Making generalizable statements about companies' IT resources is difficult, but we can assume that they will range from small on-premises commodity servers connected through the public internet, to multiple racks that connected directly to an exchange through fiber optics.

3.4 Summary

In this chapter we examined the spectrum of computing platforms used throughout different edge computing scenarios. We complemented the state-of-the-art with an architecture and prototype for a portable energy-aware edge cluster, with which we revealed several challenges related to applying models from cloud computing to high-density edge computers. We also showed different scenarios in which dispersed edge computing resources could be pooled together to a computing substrate.

The edge computing scenarios we examined highlight the tension between infrastructure abstractions and resource management precision. Successful cloud computing models, such as serverless computing, are effective at hiding the underlying computing infrastructure from the application developer or operator. It seems this conflicts with edge computing, where services may have to be placed on very specific devices, for example ones connected to a specific sensor or actuator. Moreover, as the variety of AI accelerators increases, so does the heterogeneity of the edge fabric, thereby introducing additional complexity for platform abstractions. The examination revealed some concrete challenges for edge computing platforms. It is particularly important that developers can trust the “write once, deploy anywhere” philosophy. A programming model, such as a Function as a Service (FaaS) style model, needs to hide the heterogeneity of the underlying edge resources, but at the same time provide methods to make placement constraints explicit. Not only is this crucial for efficient development of AI applications, it is also a critical requirement for transparent resource provisioning, and to enable a seamless continuum of edge resources, where coordination middleware can transparently make tradeoffs between application performance, resource utilization, and associated costs. These are critical insight which drove the research presented in Chapter 6.

When building middleware, computing platforms, and operational mechanisms, researchers and developers can often make assumptions about the systems they operate in. In cloud computing, many operational mechanisms build on the assumption that servers have similar computing power, and that they operate in the same network with similar bandwidth. On a highly heterogeneous and geographically dispersed computing substrate, such assumptions may be invalid. Building generalizable operational mechanisms, such as load balancing, scheduling, or autoscaling, is much more difficult, and their performance is subject to many more variables related to the underlying infrastructure. Fine-tuning these operational algorithms to the specific scenario is a key challenge. By studying the hardware platforms, and the infrastructure scenarios we presented in this chapter, we were able to extract commonalities and build a high-level framework for evaluating middleware, computing platforms, and operational mechanisms, that have to operate on top of a dispersed computing substrate. This framework is the topic of the next chapter.

CHAPTER 4

Edge system analysis and evaluation methods

It is challenging to evaluate edge computing systems, such as compute platforms [RHM$^+$19], communication middleware [RND18], or resource allocation methods [SNSD17], in a way that allows generalizable conclusions. We attribute this to the inherent complexity and heterogeneity of such systems, and a scarcity of available reference architectures for edge infrastructure, standardized benchmarks, and data on real-world deployments. Evaluating cloud computing systems in simulated environments has become comparatively straight forward due to the large body of architectural models, benchmarks [HBS$^+$15], and trace data sets from production-grade cloud platform deployments [VPK$^+$15], which facilitate well grounded systems evaluations. In contrast, edge computing researchers currently rely on either highly application-specific testbeds [HCH$^+$14, MKN$^+$17], or abstract system models such as P2P or hierarchical three-tiered topologies [SNSD17, XLD$^+$19, TLG16]. Moreover, the lack of data on real-world edge computing deployments forces developers of simulation tools to resort to overly simplistic simulation models.

In this chapter we present our three contributions to the body of evaluation methodologies for edge computing systems. First, in Section 4.1 we present Galileo, a system that streamlines profiling and benchmarking of real-world workloads and hardware, and provides tools for collecting and analyzing system data. Second, in Section 4.2 we present Ether, a toolkit for generating plausible edge computing infrastructure configurations and topologies, based on our insights from Section 3.3. Third, in Section 4.3 we present a trace-driven simulation framework for distributed container-based systems, that is integrated with both Galileo for building its underlying stochastic simulation model, as well as Ether to generate random edge system topologies. We then discuss related work in Section 4.4. Finally, Section 4.5 concludes the chapter, where we also outline how the presented methods can support the development of operational strategies for distributed systems in general.

4.1 Profiling edge computer systems

We first present *Galileo* – an experimentation and analytics framework for empirical performance analysis of distributed systems. The main purpose of Galileo is to simplify the development, execution, and analysis of distributed system profiling experiments. We have used Galileo to (a) create trace data sets for our trace-driven simulation framework (Section 4.3), (b) perform experiments that compare the impact of different load-balancing strategies on energy consumption (Section 3.2.4), (c) analyse resource usage of different workloads for ML-based workload characterization ([Rai21]). We then present a set of edge computer devices and workloads we have profiled so far.

4.1.1 Galileo – An experimentation and analytics framework

Galileo's main components are a database for collecting and storing profiling experiment data, a distributed load testing runtime, an interactive shell to interact with the load testing environment, an experiment editor UI, a dashboard for ad-hoc analysis, and data science notebooks for statistical analysis of experiment results. Figure 4.1 illustrates the component interactions. We describe each component in more detail.

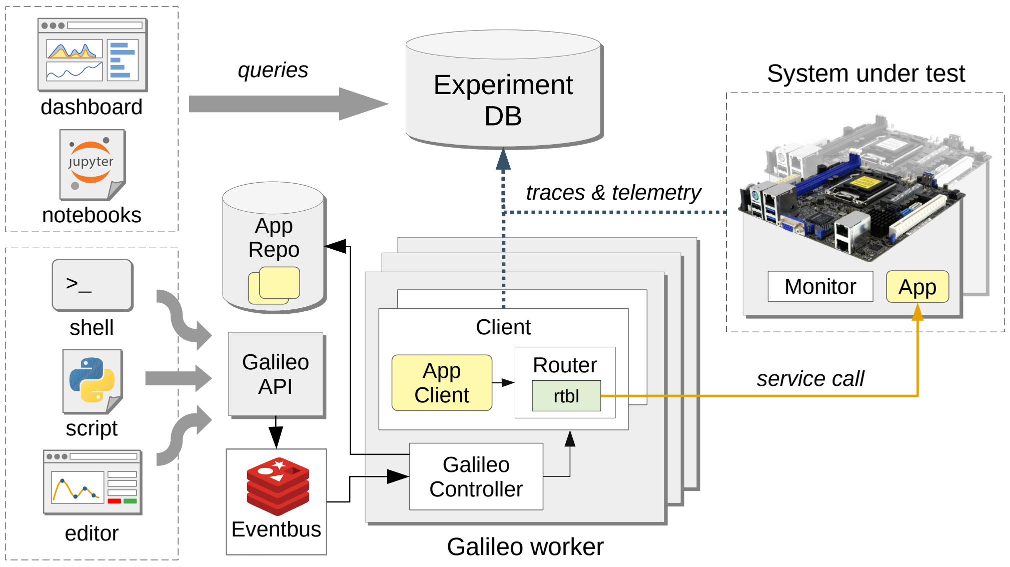


Figure 4.1: The Galileo system components and their interactions

Experiment DB

Galileo stores all recorded experiment data and associated metadata in a simple schema. System monitoring data collected from the system under test are stored as time series, where each record holds the hostname, the metric type (e.g., CPU frequency), the optional subsystem (e.g., a particular disk or network device), and the value. Similarly, we store application-level monitoring data from Galileo workers issuing service requests with additional data about each request, such as the client that created the request, the server response, and so on. Metadata about the experiment or the experiment procedure can be stored as *events*, which are simply timestamp-name-value triples. We found that the schema works well for both populating existing data visualization dashboards, for example Grafana shown in Figure 4.3, as well as facilitating statistical analyses, for example to create the simulation models described in Section 4.3.1.

Galileo workers and clients

The workforce of Galileo are physical machines that host the Galileo worker runtime. Within the runtime, a Galileo worker manages clients that generate parameterized requests to a particular service. Clients can be created and destroyed dynamically at runtime, and are scheduled to available Galileo workers automatically. Clients are simply factories of service requests. By default, clients send empty HTTP GET requests to the URL in the respective routing table entry. The requests can be parameterized to either use a different HTTP verb or pass specific request parameters. Custom clients can be deployed into the Galileo app repository, which can implement arbitrary workloads beyond HTTP, as long as they are implemented in Python. Galileo workers fetch the necessary client code on demand from the repository. This allows workload generators to package resources that can serve as request payload, for example, images or other data assets. For example, for the experiments in Section 3.2.4, the client contained several images of varying sizes that randomly selected as payload for the request to the image classifier service.

Interactive shell

Most load testing tools provide static declarative configurations for creating experiments. Instead, we provide an interactive shell that allows users to interact with the load testing environment at runtime, increment on existing experiments, and then replay the experiment through shell scripts. Through shell commands, the user can interactively scale up/down the number of emulated clients, change the load pattern in which clients are generating requests, or modify how requests are routed to the system under test. Moreover, the scripts enable the development of reusable and parameterizable benchmarks, as well as higher-level tools, such as the experiment editor shown in Figure 4.2.

Listing 4.1 demonstrates how the interactive shell can be used to create a repeatable experiment. In this example, we want to examine how a load balancer performs when host resources are used by other tenants the load balancer does not know about. We first set the routing table records where requests to a particular service should be routed to by the clients. The system provides a weighted round-robin routing scheme, so that, in this case, we can direct more load to host2. Then, we create clients that will send HTTP requests to the services. The variables ‘clients‘ and ‘noise‘ now allow interacting with all clients at once. The ‘request‘ method tells the clients to start generating requests in the given manner, either given a specific interarrival distribution (in this example an exponential distribution with $\lambda = 40$), or a limited number of requests as fast as possible. The shell is notified once all requests have been fulfilled by the clients, allowing flow control structures like ‘wait‘. Because the shell builds on the Python REPL interpreter, users can make use of all Python language features to develop complex experiment definitions.

```
rtbl.set('load_balancer', ['http://gateway.local'], [1])
rtbl.set('noisy_neighbor', ['http://host1.local:8081', 'http://host2.
    local:8081'], [1, 5])

clients = galileo.spawn('load_balancer', n=10, parameters={...})
noise = galileo.spawn('noisy_neighbor', n=5)

noise.request(ia=expovariate(l=40))
clients.request(n=1000).wait()

noise_clients.close()
clients.close()
```

Listing 4.1: Example code that demonstrates the Galileo shell

Experiment editor

A user can interact with the experiment platform either via the shell we just described, or the experiment editor shown in Figure 4.2. Through the editor, users can create custom workload configurations to describe complex request arrival profiles, and configure the system under test.

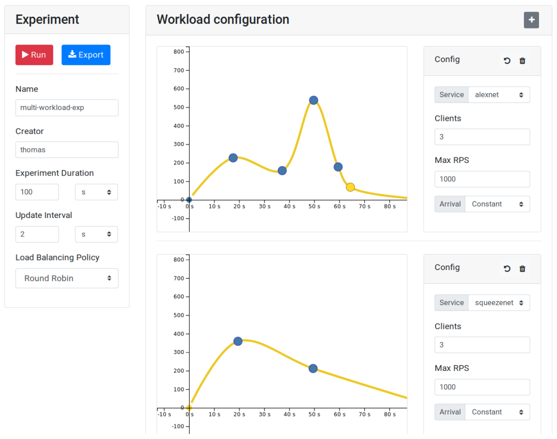
Analytics dashboard

Finally, the analytics dashboard allows ad-hoc exploration of experiment results. It visualizes system metrics for each node, aggregated energy consumption, and client trace data such as round-trip time and request queuing delay. Figure 4.3 shows the analytics UI running experiments on the cluster presented in Figure 3.4.

4.1.2 Profiling and benchmarking workloads

We have collected several workloads representative for edge intelligence applications, packaged as serverless functions. Table 4.1 lists the various functions, a short description, the programming language the function is using, the dispatching mechanism, and the platform the function can use. We provide a brief explanation of the workloads.

The workloads emphasize AI and data processing, but also include block device and network I/O stressors for testing various systems aspects. This is particularly important for the simulation presented in Section 4.3 and the workload characterization we develop in [Rai21]. The first three functions represent an entire AI workflow (see Section 2.2.1), from preprocessing to serving. ResNet [HZRS16] is a popular convolutional neural network (CNN) architecture that can be used for a variety of computer vision tasks, such as image classification or depth estimation. MobileNet [HZC^{+}17b] is a CNN specifically


Experiment

Run | Export

Name: multi-workload-exp

Creator: thomas

Experiment Duration: 100 s

Update Interval: 2 s

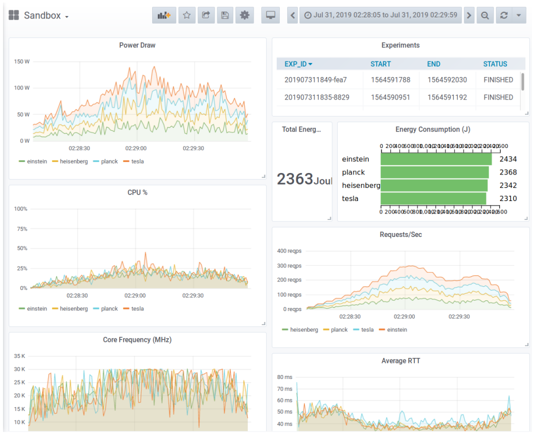
Load Balancing Policy: Round Robin

Workload configuration

Config — Service: alexnet; Clients: 3; Max RPS: 1000; Arrival: Constant

Config — Service: squeezenet; Clients: 3; Max RPS: 1000; Arrival: Constant

Figure 4.2: Galileo experiment editor



Experiments

EXP_ID	START	END	STATUS
201907311849-fea7	1564591788	1564592030	FINISHED
201907311835-8829	1564590951	1564591192	FINISHED

Total Energ... 2363 Joul

Energy Consumption (J)

einstein	2434
planck	2368
heisenberg	2342
tesla	2310

Figure 4.3: Galileo analytics dashboard

designed for mobile devices, and runs on the popular ML platform TensorFlow Lite[1], and the TPU platform. The Speech-to-text function also uses a TensorFlow CNN to perform speech detection on a CPU or GPU. For pure GPU workload, we use basic matrix multiplication with TensorFlow that performs the operations on the GPU. For CPU-based workloads we have a simple function that calculates Pi to a specific amount of digits, as well as a CPU-bound AI function. Specifically, we use the SMT solver CVC4 [BCD+11] to check the satisfiability of SMT formulas. For benchmarking block device usage, we use Linux' flexible I/O tester (FIO)[2] to generate random read/write workload. For network usage we simply use Linux' curl data transfer tool to download data from an HTTP or FTP server [3].

Table 4.1: Serverless functions used for profiling experiments

Function	Description	Language	Dispatching	Platform
resnet50-preprocessing	Scale & resize images	Python 3	Queue	CPU
resnet50-training	Fine-tuning of ResNet50	Python 3	Queue	CPU, GPU
resnet50-inference	Object classification with ResNet50	Python 3	Queue	CPU, GPU
mobilenet-inference	Object classification with MobileNet	Python 3	Queue	CPU, TPU
speech-inference	Speech-to-text transcription	Python 3	Queue	CPU, GPU
tf-gpu	Matrix multiplication	Python 3	Fork	GPU
python-pi	Pi calculation	Python 3	Fork	CPU
smt	Satisfiability Modulo Theories (SMT) solver	C	Fork	CPU
fio	Stress block I/O with random read/writes	Bash	Fork	CPU
curl	Stress network I/O by downloading files	Bash	Fork	CPU

Workloads are packaged as serverless functions (see Section 2.4 and Section 6.2), which can have different dispatching mechanism. The dispatchers are responsible for triggering

[1]https://www.tensorflow.org/lite
[2]https://linux.die.net/man/1/fio
[3]https://linux.die.net/man/1/curl

the function execution based on incoming events. Some dispatchers spawn a single runtime instance for the function, and use queue-based scheduling for invocation. In other words, events that trigger function executions are stored in a queue, and the dispatcher a FIFO ordering to invoke the function. This is useful for functions where the runtime instance uses a lot of resources, or takes a long time to start. This is the case, for example, for AI-based workloads that have to load complex deep neural networks into the VRAM or onto a TPU. Fork-based dispatchers simply fork a new process for each function execution. The distinction is relevant for modelling the impact of performance degradation. Queue-based dispatchers suffer less from resource contention, as invocations are processed sequentially. Whereas fork-based dispatchers are not subject to queueing delay, but may cause significant resource contention. We explore this behavior in more detail in [Rai21].

4.1.3 Edge cluster testbeds

We have built several testbeds using different edge devices from the computing continuum we described in Section 3.1. The specific devices we used are listed in Table 4.2. We emphasized SFF computers, SBCs, and embedded AI hardware, while omitting data center computers and mobile devices. Data center hardware is fairly well understood in cloud computing research, and mobile devices have been studied extensively by the ubiquitous computing community.

Table 4.2: Edge testbed devices

Device	Arch	CPU	RAM	Accelerator	Storage
Small form factor (SFF)					
Xeon	x86	4 x Xeon E-2224 @ 3.44 GHz	8 GB	N/A	HDD
Xeon +GPU	x86	4 x Xeon E-2224 @ 3.44 GHz	8 GB	Turing GPU - 6 GB	HDD
Intel NUC	x86	4 x Intel i5 @ 2.2 GHz	16 GB	N/A	NVME
Single board computer (SBC)					
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
Embedded AI					
Coral DevBoard	aarch64	4 x Cortex-A53	1 GB	Google Edge TPU	eMMC
NVIDIA TX2	aarch64	4 x Cortex-A57 @ 2 Ghz	8 GB	256-core Pascal GPU	eMMC
NVIDIA Nano	aarch64	4 x Cortex-A57 @ 1.43 GHz	4 GB	128-core Maxwell GPU	SD Card
NVIDIA Xavier NX	aarch64	6 x Nvidia Carmel @ 1.9 GHz	8 GB	384-core Volta GPU 48 tensor cores	SD Card

Figure 4.4 shows four cohesive testbeds we have built using the devices. Testbeds have in their network one or two SBC clusters that act as Galileo clients. Each testbed is managed by an instance of Symmetry (see Section 3.2.2) and uses Galileo to perform profiling experiments. The testbeds have been used to evaluate several research prototypes

developed throughout the thesis, as well as gathering profiling data as input for our trace driven simulation framework (see Section 4.3).

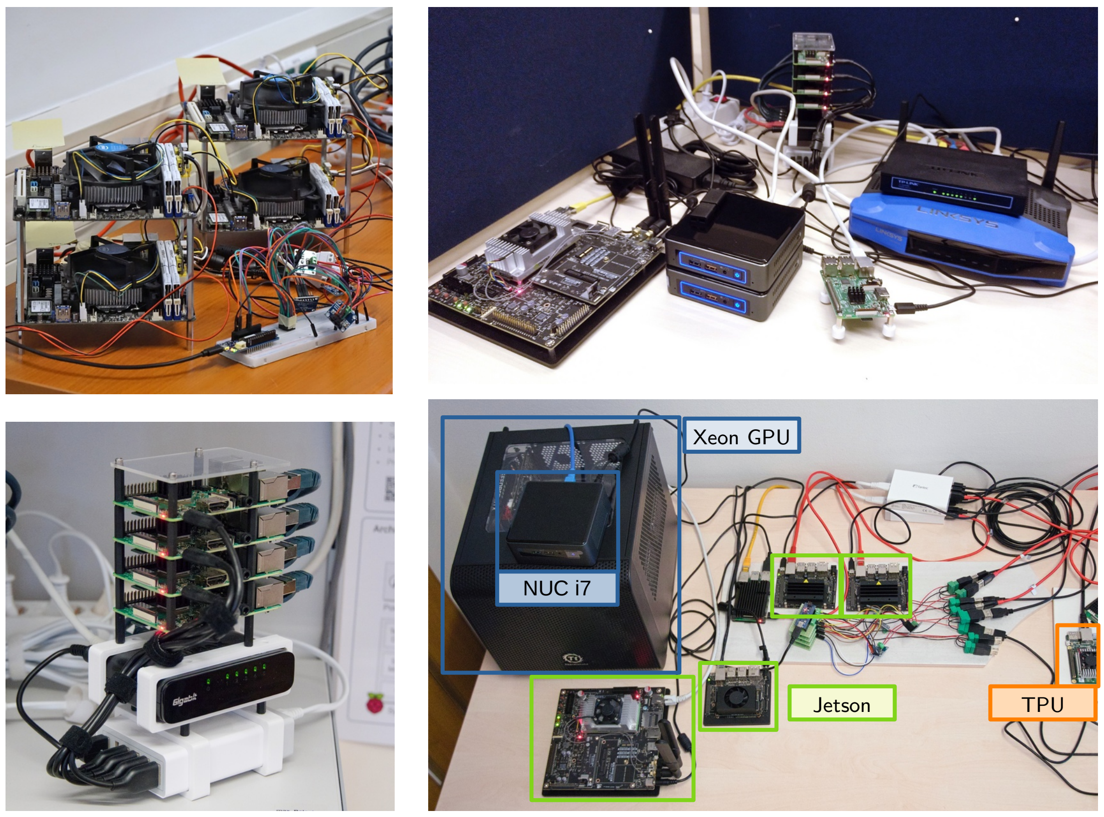


Figure 4.4: Four different testbeds we have built. Top-left: Cluster of Xeon SFF computers (used in Section 3.2.1) Top-right: Heterogeneous compute cluster (used in Section 6.5). Bottom-left: Cluster of commodity SBCs. Bottom-right: Edge AI compute cluster with power monitoring (used in [Rai21]).

4.2 Synthesizing plausible edge infrastructure configurations

This section presents a framework for synthesizing infrastructure configurations for evaluating edge computing systems under different conditions. Evaluating cloud computing systems in simulated environments has become fairly straight forward due to the large body of reference architectures, benchmarks, and trace data sets from production-grade cloud platform deployments. There are a number of tools to simulate or emulate edge systems [MK20], and while they typically provide ways of modeling infrastructure and network topologies, they lack reusable building blocks common to edge scenarios. Consequently, most edge computing systems evaluations to this date rely on either highly application-specific testbeds, or abstract scenarios and abstract infrastructure configurations. From the scenarios we analyzed in Section 3.3 we elicit common concepts for our framework. The scenarios serve as input to synthesize plausible infrastructure configurations, that are parameterizable in cluster density, device heterogeneity, and network topology. We demonstrate how our tool can generate synthetic infrastructure configurations for the reference scenarios, and how these configurations can be used to evaluate aspects of edge computing systems.

4.2.1 Overview

We propose a framework for synthesizing plausible edge infrastructure configurations from reference scenarios, for evaluating edge computing systems under different conditions and parameters. This is particularly important to evaluate the mechanisms we present in Chapter 5 and Chapter 6. We examine the commonalities of emerging application scenarios that leverage edge infrastructure in the entire compute continuum, to elevate common concepts into our framework domain. The result is a set of primitives and building blocks for generating edge infrastructure configurations and topologies. Based on data we have gathered for the scenarios, we identify a parameter space for these building blocks, to give researchers the ability to vary different aspects of the generated topology, such that the cluster topologies remain plausible extensions of the original scenarios grounded in empirical data. We demonstrate the applicability of our open source tool, *Ether* [Rau20a], by synthesizing configurations for selected scenarios, and showing how the synthetic configurations can be used as input for a simulator to evaluate systems aspects such as dataflow in a serverless edge platform [RHM+19].

Our goal is to make it easy to generate infrastructure configurations and network topologies that can be used as input for simulations for evaluating edge computing systems. *Ether*, the tool we have developed, provides out-of-the box parameterized versions of the scenarios we have defined. It also provides low-level primitives to create high-level building blocks that can be synthesized to generate custom cluster configurations. Figure 4.5 shows an example of how framework components form a topology for the IIoT scenario.

4.2.2 Conceptual model

The conceptual model of our system is simple, aims to be useful for network or systems simulations, and integrates well with other tools. It comprises the following components.

Node: A node is a compute or storage device, and represents an instance of one of the devices listed in Section 3.1. It is characterized by its architecture, generic system capacities (RAM, CPU, storage), and capabilities (presence of GPUs, or other hardware accelerators).

Link: A link is anything that facilitates connections between nodes in the network, and can represent network components such as a network card attached to a node, a Wi-Fi access point, or a mobile network base station where bandwidth is allocated across connected nodes. A link has an associated data rate capacity used as parameter for network simulators.

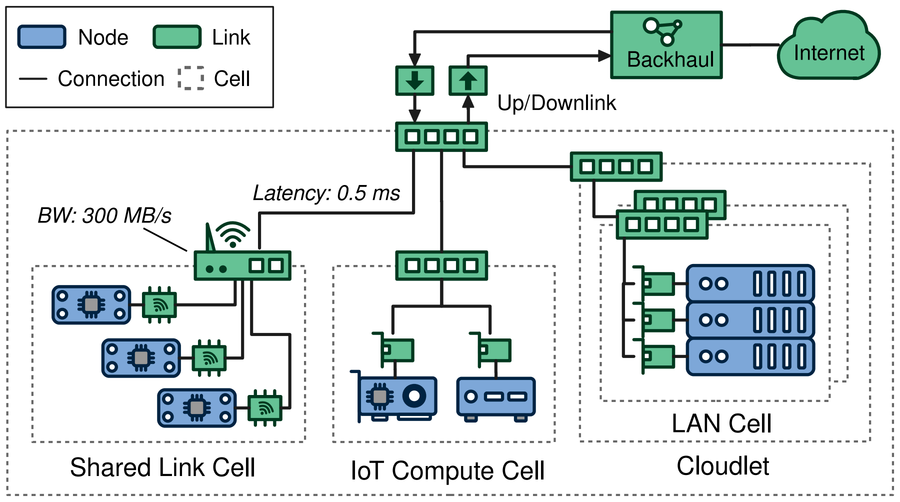


Figure 4.5: Example of how concepts, primitives, and cell archetypes create a topology for the IIoT scenario

Topology: A topology organizes nodes and links as vertices into a graph structure. Edges between nodes and links represent connections, and can hold network-QoS attributes such as latency or jitter. To connect cells across regions, our tool provides a static backhaul graph built from publicly available data on Internet latencies [inta].

Cell: A cell is a composition of nodes, links or other cells, i.e., an annotated community within the topology. It can represent an edge network that has an up- and a downlink connecting the cell to a backhaul. Cells can be composed of other cells, and connected through links (e.g., switches or routers).

4.2.3 Building blocks

Most network topology generators provide basic network infrastructure as components. A core contribution of our tool are parameterized cell archetypes as high-level building blocks, that are drawn from our reference scenarios. For example, a cell could be: a RSU node, with two SBCs for sensing and communication, a small mobile base station, and an external connection; or several cloudlets with an average of n nodes connected through a backhaul.

Primitives

Our framework provides the following primitives to compose more complex cells and topologies.

Host: The simplest cell is a network host, i.e., a node connected to a network via a link (e.g, the host's network card). We provide the devices outlined in Section 3.1 as primitives, which can be composed together to form more complex cells. The node's compute capacities and capabilities are configurable or can be synthesized based on statistical distributions.

LAN cell: A LAN cell is a common and simple way of connecting hosts via Point-to-Point connections. The synthesizer creates a new helper node that connects all nodes of the cell, analogous to a switch. It also creates up/down links to connect the cells to the Internet or other cells.

Shared link cell: A shared link cell connects specified hosts to a common shared link, whose bandwidth is shared across hosts. This can be used to model a WLAN, a mobile network base station, or VMs sharing the VM-host's network.

Geo-spatial cell: This cell puts nodes and cells into a geo-spatial context, which is useful when evaluating geographically distributed edge systems. For example, RSUs may be uniformly distributed along a road at a fixed distance, whereas USNs and mobile base stations are distributed according to a log-normal distribution across city districts (see Figure 4.6). These parameters are relevant for, e.g., capacity planning.

Backhaul connection: Edge networks are typically connected to the Internet through a backhaul network. Common variants are: (i) direct fiber uplink to an internet exchange with symmetric up/down bandwidth. (ii) business ISP with asymmetric up/down bandwidth in the order of 10–200 MBit/s up, and 100–1000 MBit/s down; (iii) mobile network uplink (e.g., via LTE modem).

Cell archetypes

The primitives of our tool allow custom composition of complex cells. Out-of-the-box, our tool provides pre-made parametric cells that are drawn from the scenarios:

Cloudlets: A basic building block for modelling small edge data centers, composed of several racks that can vary in density. Each rack is itself a cell composed of several servers that connect to a switch and may have hardware accelerators such as GPUs. While these may be data-center concepts in terms of terminology, we can use the same concept to build portable multi-purpose cluster-based cloudlets [RAD18], or systems like the Ubuntu Orange Box [VN].

IoT compute cells: Allows generating small IoT compute boxes, such as urban sensing nodes (USNs) like the AoT or Huawei PoleStar; or RSUs. This building block is similar to a small cloudlet, but much more heterogeneous. Typically, they have an IoT gateway connecting sensors or cameras to a processing device with potentially some hardware accelerators, maybe a small storage device, and a communication node connecting the box to a backhaul.

Parameterized cell synthesis

Cells can either be defined statically, or synthesized dynamically from parameters to create randomized cells. Parameterized synthesis is what allows us to generate graphs

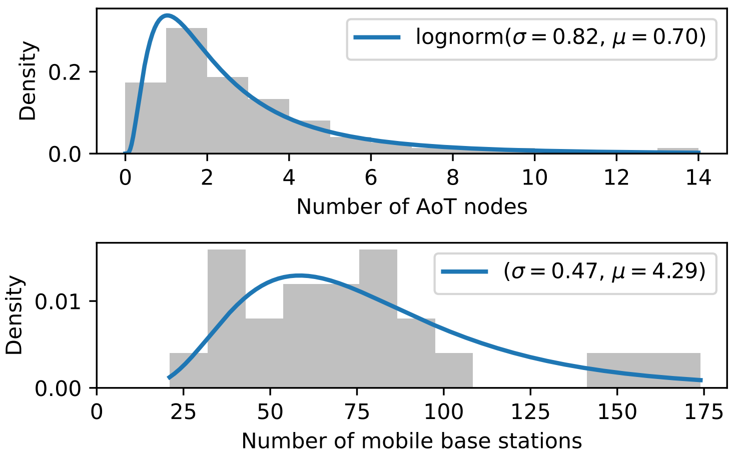


Figure 4.6: Example how geographic cell density functions can be modelled as log-normal distributions. Density of Array of Things nodes per neighborhood in Chicago [CBSG17] (top), Density of mobile base stations per district in Vienna, Austria [Aus] (bottom)

that are similar across scenarios, but different enough to test different possible variations of a system. Such parameters include:

Size: The size n specifies the number of nodes or cells in the final cell's topology. In other words the cell is made up of n other cells that are created by a specified procedure.

Density: If a cell is composed of other synthesized cells, we can provide a density function from which n is sampled for each new cell being created. For example, Figure 4.6 shows parameters for the density of cells in the Chicago city AoT deployment and mobile base stations in Vienna.

Entropy: borrows from information entropy to describe the heterogeneity of cells, i.e., the number and density of different compute device types, and how much nodes differ in compute capacity. A low entropy value means that the cell is fairly homogeneous (e.g., 0 for a server rack with one type of server), whereas high entropy describes heterogeneous cells. This parameter is useful for evaluating, e.g., resource allocation mechanisms under different levels of cluster heterogeneity.

Data distribution: Many edge computing systems evaluations deal with data aspects such as distributed storage, caching, etc. An important parameter for these evaluations is the distribution of data across storage nodes, and cells. Like cell density, we can define distribution functions to randomly vary the availability of storage nodes across edge networks.

Connecting cells to the internet

Many systems evaluate network latencies incurring from cross-regional Internet traffic. Compared to application and use-case specific networks, the core Internet has a fairly static layout, with well-studied link latencies [inta], which we can leverage. Our tool provides a static fixture of the core Internet, to which application-specific topologies can be linked. Figure 4.7 illustrates the concept. For example, in our IIoT scenario, we could link the premises' up- and downlinks to the respective Internet region in which the premises are located. That way, user-generated topologies receive coarse-grained latency values out-of-the-box.

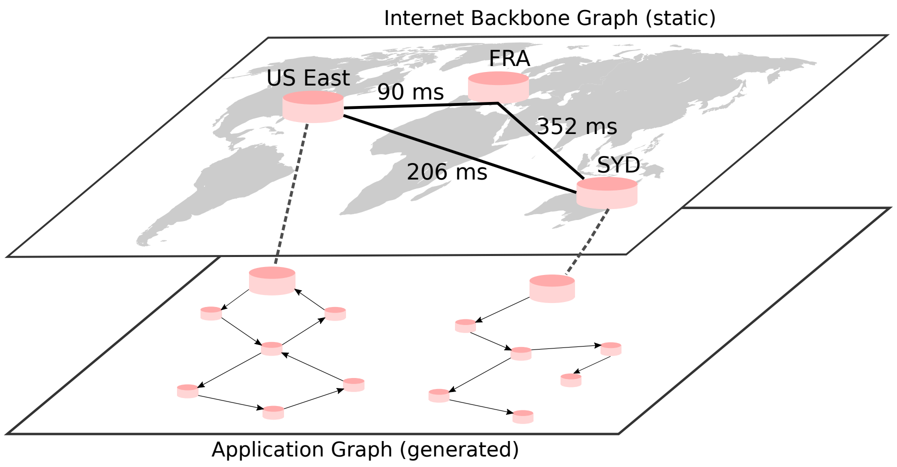

Internet Backbone Graph (static)
US East
FRA
90 ms
352 ms
206 ms
SYD
Application Graph (generated)

Figure 4.7: Mapping generated topologies to the higher-level Internet topology fixture. Latencies from wondernetwork.com

4.2.4 Demonstration

We demonstrate how our tool *Ether* can generate topologies for the reference scenarios from Section 3.3, and how topologies can be used to analyze system behavior such as network flows. We report on two selected scenarios.

Generating configurations

Our tool provides a pre-made parameterized version of **Scenario 1** (urban sensing), but we give an example of how a topology can be generated for it using our tool. We extend the scenario and model it as follows. Each AoT node is an IoT Compute Cell with two SBCs, and connected to a mobile base station in the neighborhood (i.e., they form a shared link cell). For the number of AoT nodes in each of the 77 neighborhoods we use the geographic cell density from Figure 4.6. We assume that each neighborhood has a compute box attached to the mobile base station with an Intel NUC as storage node, and two embedded GPU devices per AoT node camera for, e.g., video processing tasks. The cells are connected via mobile network to the Internet (in terms of Internet topology, as explained in Section 4.2.3, we connect them to the Chicago Internet Exchange). Furthermore, the city provides a fiber connected cloudlet with two racks and five servers per rack. The Python code to generate this topology is given in Listing 4.2. We make use of Python lambdas for parameterized synthesis, and API methods to materialize cells into a topology.

```python
aot_node = IoTComputeBox(nodes=[nodes.rpi3, nodes.rpi3])
neighborhood = lambda size: SharedLinkCell(
    nodes=[
        [aot_node] * size,
        IoTComputeBox([nodes.nuc]+([nodes.tx2] * size*2))
    ],
    shared_bandwidth=800, # selected arbitrarily
    backhaul=MobileConnection('US East'))
city = GeoCell(
    77, nodes=[neighborhood], density=lognorm(0.82, 2.))
cloudlet = Cloudlet(
    5, 2, backhaul=FiberToExchange('US East'))

topology = Topology()
topology.add(city)
topology.add(cloudlet)
```

Listing 4.2: Example code to implement the urban sensing scenario using ether

Using topologies in simulations

As part of the research presented in [RHM+19], we have built a tool to simulate serverless and containerized platforms [RRR20], which we use to evaluate different placement strategies in different infrastructure scenarios. We use *Ether* to generate several topologies as input for our simulation. The simulator implements a high-level network model on top of topologies based around *flows*, which represent data transfers between nodes through several links. We implemented a simple shortest-path routing through the topology and fair allocation of link bandwidth across flows.

The following example is an instance of **Scenario 2** (Industrial IoT) based on a prototype presented in [CWC+18]. We assume ten factory locations, each having 4 SBCs as IoT gateways, and a compute box with 1 Intel NUC and 1 Jetson TX2 board; as well as an on-prem cloudlet with 5 servers, as represented in Figure 4.5. The SBCs are connected via a shared 300 MBit/s Wi-Fi link to an AP that has a 10 GBit/s link to the edge resources, and a 1 GBit/s link to the on-premises cloud. Premises are connected via 250/500 MBit/s Internet up/downlinks.

Figure 4.8 shows the data traffic of a particular placement in the above configuration. (1) Shows the up/downlink connecting the factory premises to the backhaul. (2) Shows the shared link cell that contains the four SBC devices connected to a shared Wi-Fi link.

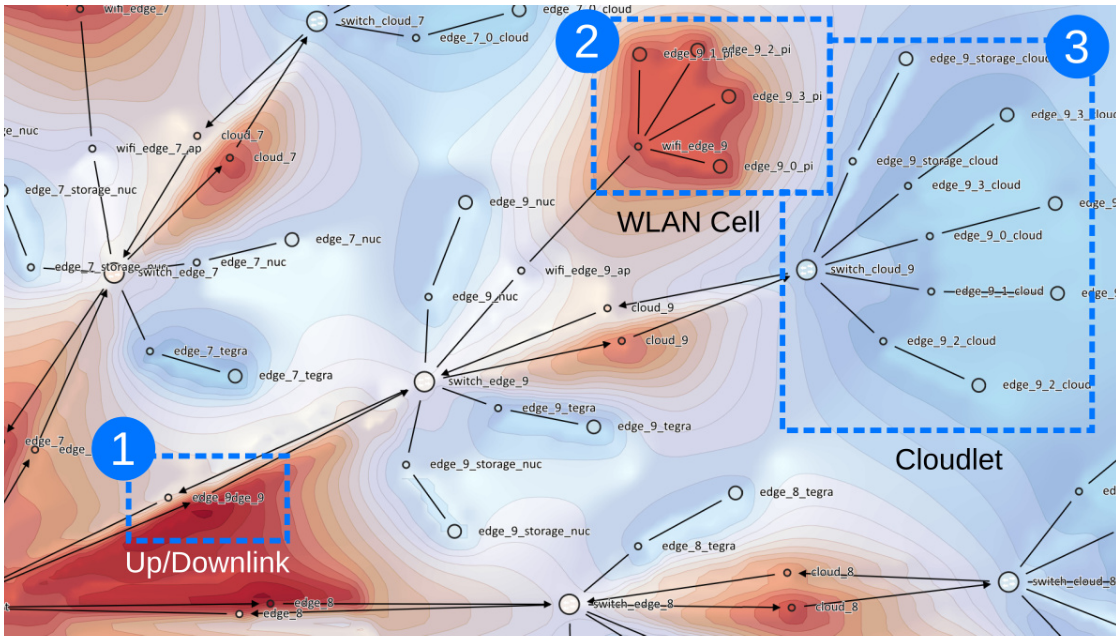


Figure 4.8: Visualization of a generated IIoT scenario topology and simulation data from our simulator, using a *topographic attribute map* [PSK+20]. The elevation field shows link capacity utilization caused by data transfer between serverless functions described in Chapter 6.

The elevation field shows that the 300 MBit/s bandwidth is exhausted. (3) Shows the cloudlet which, given the 10 GBit/s internal LAN, has the lowest utilization.

4.3 Trace-driven simulation framework

Part of our system is a discrete event simulator built with SimPy [Mat08] to simulate the basic behavior of distributed container-based runtime environments. It serves two purposes: (1) it allows experiments in different large-scale scenarios that would not produce meaningful results on our small-scale testbeds, and (2) it can be used to calculate goal functions for optimizing parameters of operational strategies, a technique we demonstrate in Section 6.4. The code is open source [RRR20].

The simulator features: a trace-driven execution simulation model that relies on profiling data rather than abstract system models, a high-level network simulation using the network model of Ether presented in Section 4.2, concepts from container-based execution environments as first-class citizens, and different workload profiles to generate random workload. The default version also comes with orchestration components that are implemented based on state-of-the-art serverless platforms, such as scheduling, autoscaling, and load balancing, which can be extended or replaced.

4.3.1 Simulation model

Network simulation

We use the network model of Ether presented in Section 4.2, which is more high-level than packet-level simulators such as ns-3 [HLR+08] or OPNET [Cha99]. These simulators model very precisely the low-level interaction between network protocols and networking hardware, which is not our goal. Instead, we trade off accuracy for simulation performance. Our network simulation builds on the network model presented in Section 4.2.2. Simulating data transfer involves opening a *flow* through several connected network *links*, i.e., a route. A flow can be thought of as a stream of data between two compute nodes that uses the bandwidth of links. Each link has a certain amount of bandwidth, and we implement fair allocation of bandwidth across flows.

When a data transfer between node n_1 and n_2 is simulated, a route is computed with a shortest path algorithm. The route contains all links along that path between n_1 and n_2. All existing flows that use links along the route are interrupted, and their bandwidth is re-allocated according to the fair bandwidth allocation scheme. A flow will only allocate as much bandwidth on links as the lowest bandwidth a link across the route (in other words, the bottleneck) can provide. For calculating the data transfer duration of the flow, we use a simple model:

$$\text{round trip time} * 1.5 + \left(\frac{\text{bytes to transfer}}{\text{goodput}}\right) \tag{4.1}$$

In other words, the time to establish the TCP connection (which is bound by the link latency, and multiplied by 1.5 to reflect the TCP handshake), plus the ideal time to transfer the given amount of bytes. The round trip time is calculated by summing up the latency values of all connections between links along the route (see Section 4.2). The goodput data rate, i.e., the application-level throughput of a communication, is estimated from the allocatable bandwidth across the route in bytes per second, multiplied by a magic number of 0.97 that roughly captures the TCP protocol overhead. The function *maxAllocatablePerFlow* is described in Algorithm 4.1.

$$\text{goodput(flow)} = 0.97 * \min_{\text{link} \in \text{route}} (\mathit{maxAllocatablePerFlow}(\text{link})) \tag{4.2}$$

We plan to add features for degradation functions to simulate the degrading TCP behavior with many flows [Mor97], as well as goodput reduction through packet loss.

Code execution simulation

The core simulator models the basic behavior of container-based serverless systems, such as Docker Swarm or Kubernetes (see Section 6.2.4). Code is packaged into container images, i.e., uniquely identified deployment units, and executed on cluster nodes by instantiating a container with the respective image. Running a container involves pulling

Algorithm 4.1: Calculate the max allocatable bandwidth per flow on a link

```
   Result: Allocatable bandwidth per flow on a link
1  Function maxAllocatablePerFlow:
       Input: link
2      flows ← flows that use this link
3      fairPerFlow ← $\frac{\text{link bandwidth}}{\text{count(flows)}}$
4      reserved ← flows whose currently allocated bandwidth < fairPerFlow
5      allocatable ← link bandwidth - $\sum_{f \in \text{reserved}}$(bandwidth allocated for $f$)
6      numCompetingFlows ← count(flows) - count(reserved)
7      if numCompetingFlows > 0 then
8          allocatablePerFlow = allocatable / numCompetingFlows
9      else
10         allocatablePerFlow = allocatable
11     end
12     return max(fairPerFlow, allocatablePerFlow)
```

the container image if it is not available on the node, starting up the container, and then running the code inside the container. Our simulator implements the basic system components of container orchestrators, such as an API gateway, scheduler, autoscaler, and load balancer. However, they can be replaced to test custom operational mechanisms. The default implementation uses the components from Skippy that we describe in Chapter 6.

The code execution model is centered around services or *functions* that live inside containers and can be invoked once they are started. To simulate code execution on a node, a `FunctionSimulator` class must be implemented, that abstracts the basic lifecycle of a function with the following methods. Each method produces SimPy simulation events, such as simulating network traffic, or requesting a node resources.

- `deploy`: deploys the code on the node (e.g., by pulling the container)
- `startup`: starts the container
- `setup`: starts the code inside the container
- `invoke`: process a request to the running container
- `teardown`: cleans up and stops the running container

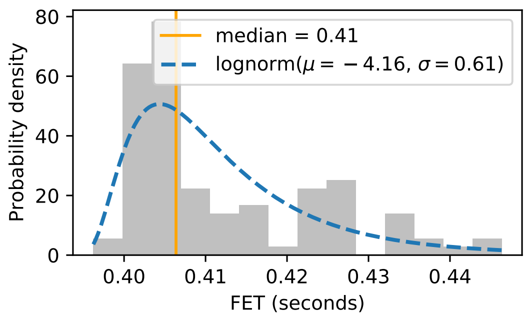
For simulating `deploy`, we synthesize our network simulation and profiling data. That is, pulling a container image is simulated using the network simulation, and the container startup time is estimated from profiling data. For pulling the container image, we first populate a virtual container registry with container image data, i.e, image names, CPU architectures, and their compressed image sizes. We run a network flow between the node pulling the image and the docker registry (which was previously placed in the network topology), and simulate the network download as described earlier. The simulation keeps track of the node's state, for example if the container image has been pulled before, in which case we can skip the operation. This mimics the basic behavior of the Docker pull command. A perfect simulation of a Docker pull command would also consider the layers of an image (Docker images use a layered file system), the availability of layers on a host, and the time it takes to decompress layers. This is out of scope of our current implementation. A limitation that should be highlighted is that, because the simulation model reflects the container deployment model, our simulator currently only supports simulating platforms that deploy function code via containers. Some platforms, such as OpenWhisk, deploy function code through platform-layer mechanisms, which would require additional simulation code.

The remaining methods, i.e., `startup`, `setup`, `invoke`, and `teardown`, have to be implemented for a specific function, or can be implemented in a generic way to simply sample from traces associated with executing the respective function on the node being simulated. Our base implementation uses a repository of traces for workloads that we have profiled on the devices in our testbeds (see Section 4.1). When simulating the execution of a function on a node, we look up function traces for that particular node in our repository, and sample from a statistical distribution that represents the execution duration. During simulation time, each node has an associated state that can holds any runtime information associated with the node. For simulating resource contention and workload interference, for example, we store all currently running functions and their starting times in the node state. We can then modify the sampled execution duration with workload interference factor we describe next.

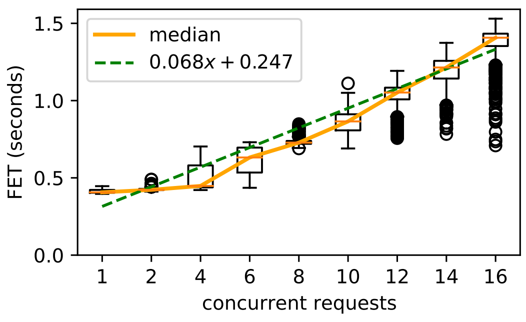
4.3.2 Performance modelling

A goal of the simulator is to determine the Function Execution Time (FET) of a function on a node to know how long the node is busy executing that function. Many system simulators, such as CloudSim [CRB$^+$11], use a simple model that calculates the execution time from the number of instructions of the function, and the processor's instruction rate in instructions per second (IPS) [MK20]. There are several issues with this approach. First, the speed of a given CPU depends on many factors. Depending on the computer architecture, not every instruction takes an equal amount of time to execute, meaning that the IPS is not static for the CPU, but depends on the instruction mix, i.e., the task being executed. Moreover, IPS is a function that depends on the CPU's clock speed, which, as we have shown in Section 3.2.4, can be highly dynamic in edge computing hardware. Other factors include cache sizes, or I/O access density. Second, it is difficult to determine the number of instructions of a task, or the real IPS for a CPU. The effort of determining these data is equivalent to running a full set of profiling experiments, meaning there is no benefit over a trace-driven model.

Instead, we model execution times as random variables whose distributions we determine from traces, and then scale during runtime given certain system parameters to account for,



(a) Distribution with single request $n = 100$



(b) Degradation with increasing requests

Figure 4.9: Example function execution times (FET) of running an AI function

e.g, degrading performance or workload interference. We describe the effect of resource contention on the FET as *degradation functions*, whose output can be used to adjust the FET values from the ideal case, to reflect the current system conditions. We first describe the simple single tenant case where only one type of function exists in the system, and proceed to the general multitenant case.

Single-tenant scenario

Modelling FET and the degradation function for a single service is relatively straight forward. Figure 4.9 shows data from a profiling experiment, where a function that solves Satisfiability Modulo Theories (SMT) formulas, was executed on a node from our edge cluster (see Section 3.2.3). Figure 4.9a shows the density histogram and the fitted log-normal distributions parameters from the ideal case, i.e., where no other tasks are running on the system concurrently. Figure 4.9b shows how the execution duration degrades because of resource contention, and a linear regression estimation of the medians. In Section 3.2.4, specifically Figure 3.5a, we demonstrated that linear models work well when the input parameters are based on the number of concurrent requests.

Based on these observations, a reasonable simulation model for the FET could be defined as follows. We fit a linear function to estimate the median FET based on the number of concurrent request, and, during simulation time, add some log-normal distributed noise. So, we define the FET of function f as a function $\hat{t}_f$ of r, where r is the number of concurrent requests of function f. In a multi-node system we would define $\hat{t}_{f,n}$ for each node n in the system that f can be executed on. We omit this for brevity. For the AI function example from Figure 4.9, the parameters could be defined as

$$\hat{t}_f(r) = 0.068 \cdot r + 0.247 + X \tag{4.3}$$

where $X \sim \text{Lognormal}(-4.16, 0.61)$ and adjusted for the current median location. This yields a reasonable simulation model for the FET of function f that is based on traces from profiling experiments. In previously published research, we have found that this approach yields good results [RHM20b]. The degradation function gives particularly accurate results, since it is fine-tuned to one specific workload. However, this model only works for single-tenant scenarios.

Multitenancy scenario

Much more common are multitenancy scenarios, where multiple functions $f_1, f_2, \dots, f_n$ execute on the same node and share its resources. Here, it is particularly important to model the performance degradation through workload interference, i.e., modelling how concurrent executions of f_1 affect the FET of f_2. Although the features of container platforms, such as Docker that uses the Linux kernel feature cgroups, are supposed to isolate containers from the effect of co-located containers (also known as noisy neighbors), we, and others [JGJ+18, KGL+20], found that there can still be significant degredation through interference, especially for CPU and IO resources.

Building on the single tenant case, we could define $\hat{t}$ for every function f as a multi-dimensional function, whose parameters are the number of concurrent calls of *all* existing functions

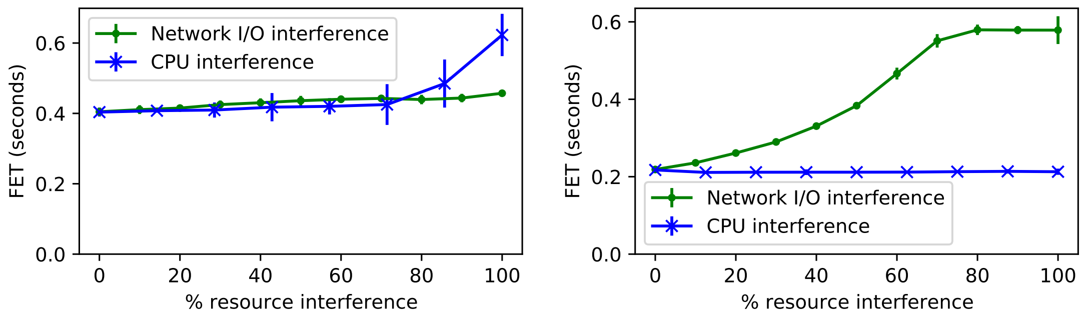
$$\text{FET}_{f_1} : \hat{t}_{f_1}(r_{f_1}, r_{f_2}, \dots, r_{f_n}) \tag{4.4}$$

where r_{f_i} are the number of the concurrent requests to function f_i. This is clearly infeasible when the number of existing functions in the system increases, as the number of possible input parameter combinations explodes, and it would no longer be practical to run the necessary profiling experiments for creating trace data.

Resource modelling and workload characterization

It is clear that we need more generic input parameters that do not depend on profiling the full combinatorial space. For the use in the simulation, it is important that the input parameters of the function (1) are expressive enough to capture the behavior, (2) can be determined from trace data, and (3) can be computed at simulation time. Simulators like CloudSim use system level metrics, such as the MIPS of a processor, to determine execution times. This would be equivalent to using the current CPU utilization as input parameter for $\hat{t}_f$.

However, the effect of resource contention on performance degradation depends on the type of resource being shared, and the type of workload being executed. For example, parallel I/O tasks may experience a heavier slowdown than parallel CPU tasks. Figure 4.10 shows data from our profiling experiments that demonstrate this. Each figure shows the FET for one function, given workload interference of particular resources. The x-axis represents how much of either CPU or Network I/O is used by other background processes. Figure 4.10a shows the execution time of the AI function we described earlier. Figure 4.10b shows the FET of a function that downloads 20 MB of data (e.g., for a data preprocessing step in an ML pipeline). The first function is drastically affected by CPU resource contention, whereas the second function is not at all. These results



(a) AI function (CPU bound) (b) Data fetching function (network bound)

Figure 4.10: Degrading Function execution times (FET) of different functions and workload interference

are not surprising, but demonstrate the problems of modelling FET as a function of one-dimensional parameters such as CPU utilization alone.

Instead, we consider the utilization of five resources, namely:

- 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 any), and
- u_{ram} the RAM usage.

For the approach to be feasible, we also need to: (1) characterize functions in terms of their resource usage, (2) describe $\hat{t}_f$ as a function of these parameters, (3) compute the resource utilization values of a node at simulation time.

Details of the approach are described in [Rai21], here we provide only a brief summary. For (1) we perform several profiling experiments using combinations the workloads described in Section 4.1 with an increasing number of concurrent requests. We record for each function the resource usage and the FET. Then, we compute a workload characterization vector $\mathbf{u}$ for each function that describes the mean resource utilization over the entire function execution duration. For (2) we, build a regression model for each node that predicts a resource degradation factor based on values of $\mathbf{u}$. The factor is applied to the FET in the best case, where 0 means no degradation, and 1 means a doubling of the FET. The regression model takes converts the vectors $\mathbf{u}$ for all concurrently running functions, and converts them into an input vector that contains for each resource u the sum, mean, median, and standard deviation. We train the regressor using the profiling data we gathered for (1). For (3), we simply keep track of all concurrently executing function on a node, and use the determined resource utilization vectors $\mathbf{u}$ for computing the regressor input in the same way.

4.3.3 Energy consumption modelling

CloudSim, and all its derivatives [MK20] such as iFogSim, use a simple power model as a function of a scalar utilization value $\hat{P}(u)$, where u is the CPU utilization between 0 and 1. Each node type n in the system has its own power model. We omit this for brevity. By default, CloudSim provides a linear, square, cubic, and square root based function. The power draw is scaled between a fixed P_{max} and P_{idle}, i.e., the maximum power draw, and the power draw of the system in idle mode. For example, the linear power model is described simply as:

$$\hat{P}(u) = P_{idle} + u \cdot (P_{max} - P_{idle}) \tag{4.5}$$

However, as we have shown in Section 3.2.4, such models are too simple to correctly explain the energy consumption behavior of edge clusters. Instead, we model P as a function of the resources described earlier.

$$\hat{P}(u_{cpu}, u_{io}, u_{net}, u_{gpu}) \tag{4.6}$$

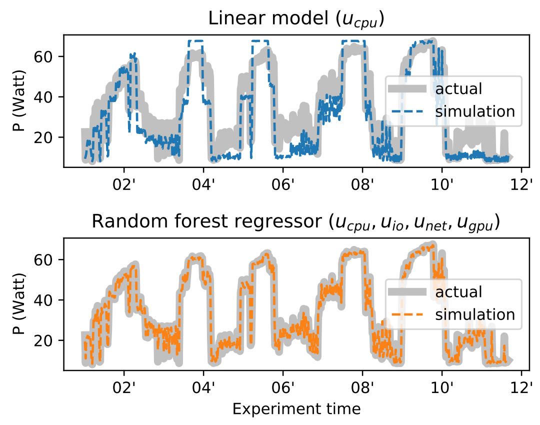
The function $\hat{P}$ is learned via multiple regression for each node, using data from our profiling experiments. We found that random forest regressors produce very good results. The utilization values during simulation time are computed as explained earlier.

To demonstrate the difference in power models, we test four models against each other. Figure 4.11 shows the result two of the models, in experiments we performed where we mixed several workloads of different intensity. The figure on the left shows results from a Xeon CPU node from our cluster described in Section 3.2.3. The figure on the right shows results from a Raspberry Pi 4. In both cases, we run a with semi-random combination of functions that use different resources over a few minutes. The figures show the actually measured power draw during the experiment, and the power draw estimated by the linear CPU-based model and the random forest regressor.

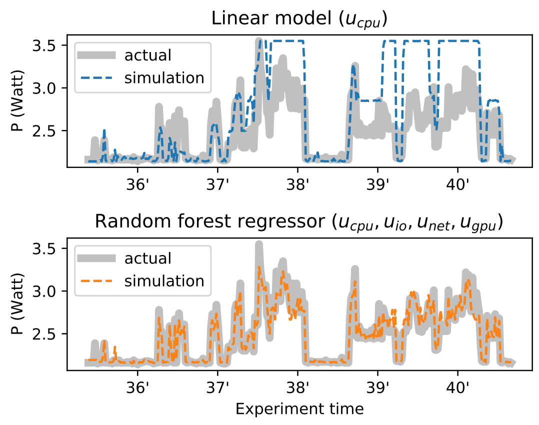
We also fitted a quadratic model using CPU utilization (like CloudSim provides), and a model using multiple linear regression on the same parameters the random forest (RF) model receives. We used the models to estimate the power draw over the entire experiment and calculated the root mean squared error (RMSE) values for each one. The error values are shown in Figure 4.12. The values do not generalize, and are specific to the experiments. The difference between models may vary for other workload compositions.

4.3.4 Synthetic workload generators

We present our approach to generate synthetic workload at simulation time. Synthetic workload generators in cloud computing often use static arrival rates or rely on traces from real systems [KRM06]. Since real-world trace data on edge computing systems

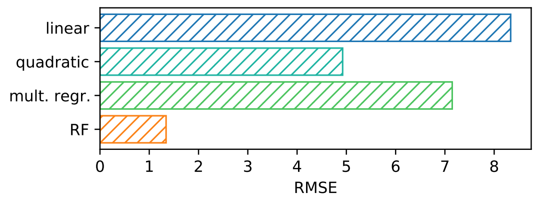


(a) Experiment on a Xeon node

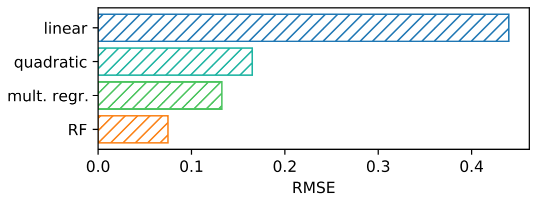


(b) Experiment on a Raspberry Pi 4

Figure 4.11: Predicted and actual power draw over time during experiments comparing linear and random forest regression power models.



(a) Experiment on Xeon node



(b) Experiment on Raspberry Pi 4

Figure 4.12: Error metrics of different power models during the experiments

is scarce or non-existent, we rely on random workload generators. In our simulator, workload generators are defined by composing definitions of an arrival process and a workload pattern. The arrival process is described by the distribution of interarrivals, i.e., Δt between requests. For example, Internet traffic has been found to follow Poisson arrival processes [Fel00], where interarrivals are described by an exponential distribution. The workload pattern describes how the request rate changes over time t. This is useful to model systems where, for example, the number of users changes given depending on the time of day or weekday. Figure 4.13 illustrates the concept and shows several examples on how different arrival processes and workload patterns can be composed.

The first row shows how to achieve a randomized sinusoidal request pattern, for example for the use case described in Section 5.1.1 or the evaluation in Section 3.2.4. For the interarrival distribution we use an exponential distribution. The probability density function (PDF) of an exponential distribution is $\lambda e^{-\lambda x}$, where $\frac{1}{\lambda}$ is the mean. The workload pattern over follows a sine wave, and the value for $\sin(t)$ is used as λ to scale the interarrival distribution. At simulation time we therefore sample from the distribution

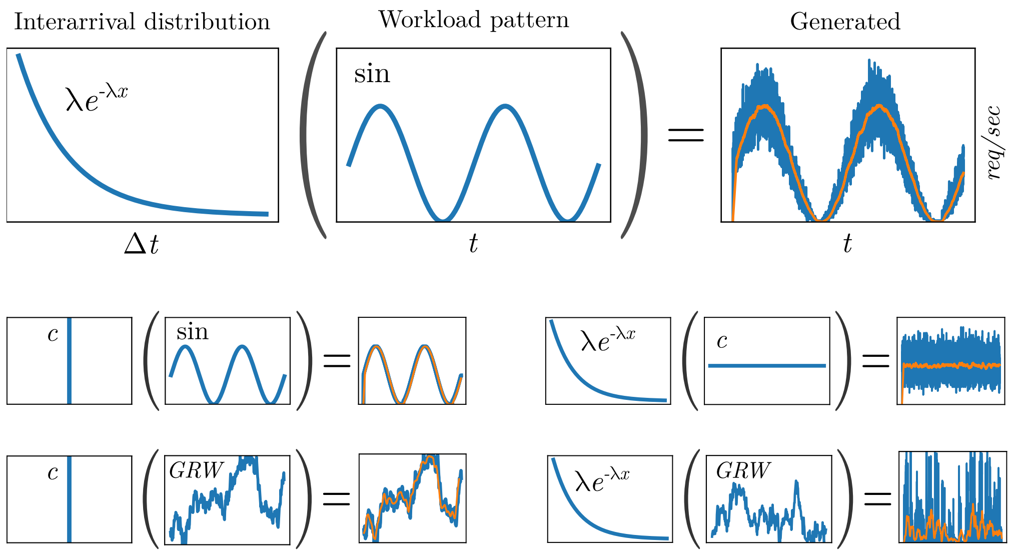


Figure 4.13: Workload generator patterns

$\sin(t)e^{-\sin(t)x}$ to receive the wait time until the next request. The orange line shows a moving average of the requests per second, which should roughly match the workload pattern. The second row shows how a constant interarrival distribution can be used to model exactly the workload pattern, and how a constant workload profile can be used to model a static workload pattern with randomized interarrivals. The last row shows Gaussian random walks (GRW), where each value represents a random sample from a Normal distribution, that is then used as value for μ in the next random sample. The request profile can be parameterized with a σ value that affects the fluctuation over time.

So far we have used these workload generators for evaluations in [RRD21, Ras20, Rai21]. A detailed evaluation of the workload generators is out of scope of this thesis.

4.4 Related Work

In cloud computing research, existing benchmarks [HBS+15], and trace data sets from production-grade cloud platform deployments [VPK+15], facilitate well grounded systems evaluations. Some simulators, such as CloudSim [CRB+11], use profiling data from data-center server computers. Evaluating edge computing systems is much more challenging, given there are no standardized benchmarks, or even a consensus on the type of infrastructure that edge computing systems will use. Only recently have researchers begun to profile edge devices for use-case specific workloads, such as image classification [MPD19], or video analytics [WFC+19]. With our system Galileo, these profiling experiments can be streamlined and developed in a reproducible way. Moreover, our datasets on profiling data of our edge clusters significantly extend the existing body of data.

There are a number of tools to simulate edge systems [PVCM20, MK20]. To evaluate the types of middleware that we are developing in this thesis, we found that these tools lack three things: fine-grained modelling of execution times for heterogeneous workloads and devices, automatic generation of system infrastructure configurations and network topologies. We have already shown issues of existing tools with respect to performance modelling in Section 4.3.1.

For defining topologies, existing edge computing simulators offer in general two ways: They either let users import topologies that were created up-front using random generators like BRITE [MGG+17, MLMB01]; or provide APIs for constructing topologies programmatically [GVDGB17], through configuration files [SOE17], or graph formats [FCFMO+19]. Our framework can be used to generate plausible infrastructure configurations grounded in existing scenarios as input for these tools.

Existing approaches to generate randomized network topologies rely on abstract models such as random geometric graphs [Wax88, ER59], or hierarchical structures and power-law degree distributions [MLMB01]. Network research has been investigating Internet topologies for decades. There are several approaches to procedural topology generation, such as [MLMB01, TGJ+02, CDZ97]. These approaches aim to be generic, and focus on Internet-scale topologies. There is no distinction of host device types, capabilities, or reusable building blocks (such as edge data centers [SBCD09], or portable edge computers [RAD18]). Our approach is different in that it provides high-level concepts from existing edge infrastructure and hybrid edge cloud scenarios.

4.5 Summary

Evaluating edge computing systems, such as compute platforms or resource allocation algorithms, in a generalized way is challenging. Although existing tools facilitate simulation or emulation of such systems, we have found that (1) there is a lack of real-world testbeds and profiling data for edge computing systems, (2) simulators make very simplistic system model assumptions which can result in inaccurate results, and (3) it is difficult to describe an underlying edge infrastructure and vary its parameters. In this chapter, we first presented Galileo, an experimentation and analytics environment to streamline systems experiments, that can generate input data for trace-driven simulation framework that we built. Our simulation framework builds on empirical data rather than simplified system models, allowing more flexibility in the types of systems being simulated. In the previous chapter in Section 3.3 presented edge infrastructure scenarios that encompass many edge system characteristics. We abstracted these characteristics into a framework and a tool, and showed how it can generate different infrastructure configurations for these scenarios, and how the building blocks we provide can help edge systems researchers evaluate systems under different scenarios and conditions.

Together, we believe that these tools can facilitate a closed development loop for operational strategies in general (see Figure 4.14). Operational strategies, such as schedulers, autoscalers, or load balancers, control the real system, which reports monitoring and trace data into a common data format. The traces are used as input for a simulator, that can

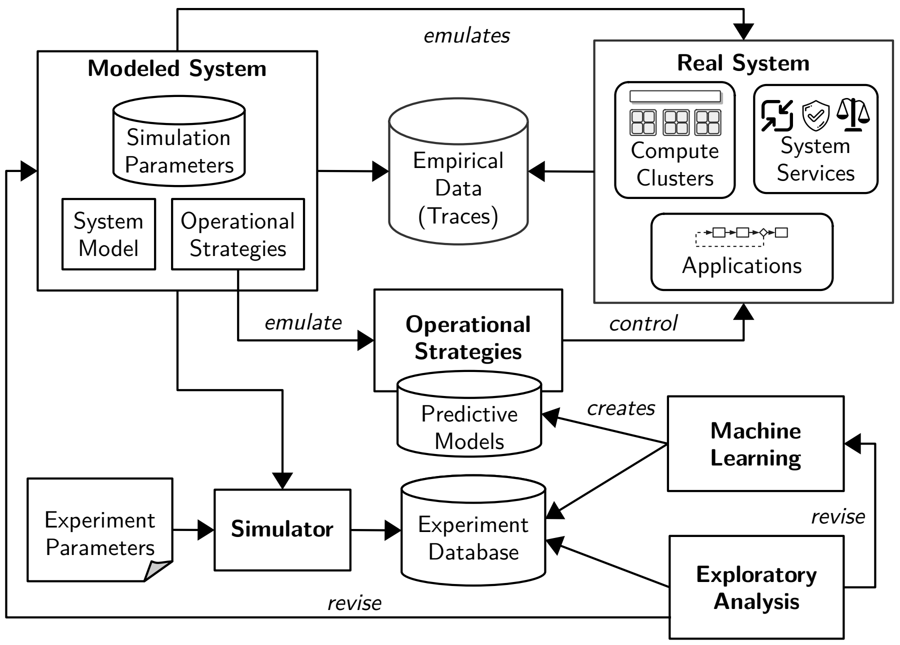


Figure 4.14: Closed development loop

examine operational strategies in given different system parameters. Once an experiment is executed by the simulator, operators can drill into the results using the exploratory analysis dashboard of Galileo. The statistical analysis tool analyzes the results of the experiment database and is used to create predictive models, which themselves feed into the operational strategies, to close the feedback loop into the real system. Moreover, with the tight integration between the trace-driven simulator and the real system, one can use the simulator to tune operational strategies at runtime, we demonstrate this in Chapter 6.

CHAPTER 5

Elastic message-oriented middleware for edge computing

In Chapter 2 we argued that sensing and communication middleware is a crucial component of edge intelligence systems. Efficient device-to-device communication, publishing and accessing sensor data, and integrating loosely-coupled software components, are all important aspects of edge intelligence that can be solved with message-oriented middleware (MOM). Current industry solutions involve cloud-based MOM based on the publish–subscribe (pub/sub) model, where data centers host massive transparently scaling brokers that can serve billions of messages per second [Bar15]. The physical distance between clients and brokers, however, is a serious limitation for many IoT and smart city applications that we have also outlined in Chapter 2. Rather than routing all messages to the cloud, brokers could instead be deployed on edge resources to reduce end-to-end latencies between devices in close proximity. However, the geographic dispersion of edge resources, and their unpredictable availability make it challenging to provision and manage them in a centralized manner. Client mobility further complicates matters, especially with respect to message delivery and QoS guarantees.

This chapter presents EMMA, an elastic message-oriented middleware for edge computing applications, that addresses these challenges. First, in Section 5.1, we motivate the need for edge-enabled MOM in more detail, and elicit design goals. We then present our overall system architecture in Section 5.2. The fundamental mechanisms underlying our system are (1) the orchestration of the broker–client network (Section 5.3), (2) proximity awareness (Section 5.4), and (3) elastic diffusion, i.e., scaling the broker network to the edge dynamically at runtime (Section 5.5). In an empirical evaluation using a real-world testbed, we show that EMMA can significantly reduce end-to-end latencies caused by message routing, even in the face of changing network topologies and client mobility. In Section 5.7 we discuss the rich body of related work on message-oriented middleware. Finally, in Section 5.8, we conclude the chapter with a summary.

5.1 Motivation

Message-oriented middleware has undergone several architectural paradigm shifts over the past decades. The scalability issues of using centralized servers for application-layer message dissemination were discussed intensely during the peer-to-peer era, and many solutions based on completely decentralized peer-to-peer architectures were developed [EFGK03]. But the forces of monopolization and the widespread adoption of cloud computing have caused many commercial solutions to return to a centralized approach, where scalability is largely achieved by consolidating resources into massive data centers. Amazon's AWS IoT [Bar15], or Microsoft's Azure IoT Hub [Sta19], for example, are massive transparently scaling centralized pub/sub systems that can handle billions of messages per second, even in the face of varying load, with a pay-as-you-go cost model. However, despite advances in cloud computing and clustering techniques [GSGKK15], IoT and edge computing scenarios have introduced new challenges, in particular for centralized systems [HKM+17]. IoT applications with ultra-low latency, network resilience, or stringent data privacy requirements cannot be implemented by cloud-based solutions alone, but demand decentralization [SSX+15]. Instead of routing all messages to the cloud, brokers can be deployed on geographically dispersed edge resources to reduce end-to-end latencies between devices in close proximity, and enable network resilience and privacy constraints, illustrated in Figure 5.1. Yet, system engineers have grown accustomed to the convenience and benefits of centralized deployment and management.

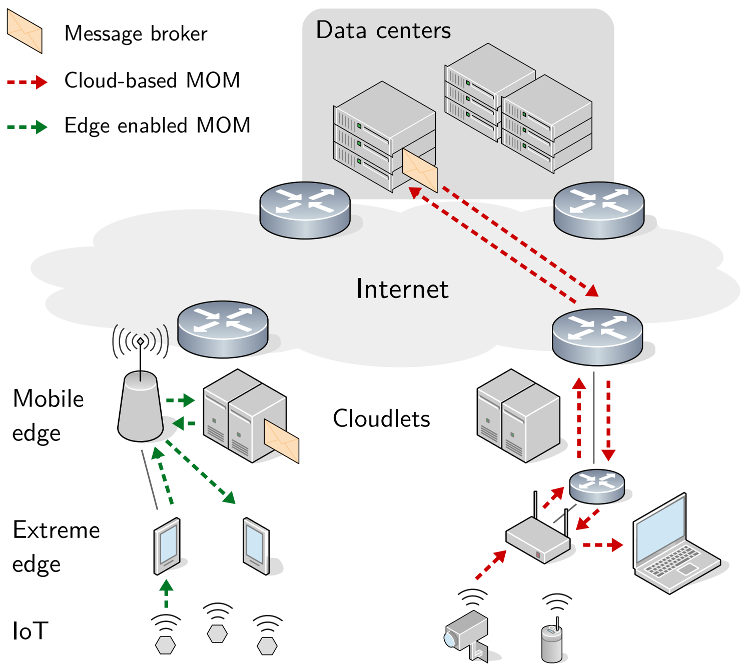


Figure 5.1: Device-to-device communication can be significantly improved when edge resources are used for brokering messages in multi-tier network topologies

Osmotic computing has been proposed as a new computing paradigm to address this dilemma [VFD+16]. It borrows from the biological principles of osmosis, where solvent molecules seamlessly diffuse into regions of higher solute concentration. Although osmotic computing was originally conceived for the dynamic management of microservices across federated edge/cloud infrastructure, it has enjoyed success in other problem areas such as stream processing for IoT [NND+17] or deep learning [MJS+17]. We argue that osmotic computing principles can be applied to seamlessly integrate cloud-based MOM into edge computing. Osmotic computing can abstract how we think about elasticity of MOM towards the edge, i.e., dynamically moving or provisioning message brokers from the cloud the edge based on current demands. In particular, we propose an architecture based on two diffusion models: broker and client diffusion. From a static centralized deployment in the cloud, we bootstrap a network of brokers that diffuse into the edge based on resource availability, and the number of clients and their proximity to edge resources. Clients in proximity diffuse to the edge depending on broker proximity and the potential to optimize QoS between the clients. We take a closer look at an application scenario to elicit concrete requirements for MOM based on osmotic computing principles.

5.1.1 Application scenario

We already summarized several application scenarios in Chapter 2 that drive the development of edge computing solutions. Modern IoT scenarios highlight particularly the need for MOM that leverages resources at the edge. To drill down into the requirements, let us consider the following scenario from the mobile health (mHealth) domain, adapted from Lorincz et al. [LMFJ+04] and Rizzo et al. [RRF+20]. In a major disaster situation, effective coordination and prompt reaction of emergency teams is paramount to save people's lives. To support on-premises decision-making, first responders attach wearable biosensors to patients, which continuously publish medical signs. Emergency technicians carry mobile devices with software to visualize patient data, trigger alerts, and coordinate team efforts. Cloud-based IoT platforms may be impractical or unreliable in this case, as such health-critical mobile systems must be resilient towards network partitions, and require near-real time data processing and message exchange. Instead, resources in proximity, such as portable edge computers we presented in Section 3.2, deployed in emergency vehicles must be used to provide necessary services. From there, data can be processed, forwarded to hospitals for further analytics and acute patient and handed off to the cloud for long-term storage. MOM plays a critical role in this type of scenario. It has to facilitate low-latency communication between devices in proximity, e.g., to provide immediate alerts to changing medical conditions, while still being able to disseminate messages to locations on different levels of geographic dispersion. However, the mobility and unpredictability of both clients and resources that could act as brokers require a completely dynamic operational mechanism. In cloud-based MOM, operational configuration, such as broker addresses or publish/subscribe topics, are typically best defined at deployment time. In contrast, edge-based MOM needs to react to spontaneously emerging network structures and device congregations, such as in our disaster scenario. This also means that clients need to be able to quickly locate the closest broker, and QoS optimizations need to consider that proximity between clients and brokers may change during runtime, leading to dynamic routing as end and edge devices change location.

5.1.2 Design goals

As we have illustrated, the IoT and edge computing architectures provide a great deal of challenges when building messaging middleware. Although the Message Queue Telemetry Transport (MQTT) protocol has proliferated as a standard pub/sub platform for IoT applications [AFGM+15, LCC+15, HKM+17], state-of-the-art MOM solutions based on MQTT fall short of addressing these issues. The dispersion of edge resources adds to the overall complexity of system management mechanisms [Sat17]. Based on our motivating scenarios, and analyses of edge computing characteristics in Chapter 3, we have identified several key design goals that drive the design of our architecture and control mechanisms.

Stringent latency requirements

Optimizing end-to-end latencies for clients is one of the key goals of our middleware. Most of the latency in messaging middleware is caused by packet routing, lost packet retransmission processing, and physical propagation delays of network links. Routing messages to the cloud and then disseminating them from there is wasteful, especially if device-to-device communication happens in close proximity. For these situations, brokers deployed at the edge can provide ultra-low latency communication. At the same time, subscribers may be located at geographically dispersed locations, e.g., a data analytics services in the cloud, meaning that messages must be forwarded to the cloud if necessary.

Mobile brokers and endpoints

Mobility is a fundamental element of IoT [AFGM+15, LCC+15] and edge computing and should be considered as such by osmotic middleware by design. In our scenario, not only clients, but also brokers can be mobile, consequently the network topology is highly dynamic and subject to high churn. Clients as well as edge resources are mobile and may unexpectedly leave or enter the system. Proximity between nodes may change any time during runtime, and the system should be equipped with mechanisms to maintain optimized QoS, even in the face of client or broker mobility. Providing message delivery guarantees in the face of mobility is another prominent challenge.

Horizontal scaling

Resources at the edge are a key component in enabling the previous two goals. However, because these resources are highly dynamic, we cannot predict or rely on their availability. We therefore also cannot provide static operational configurations. Instead, a central configured deployment should be able to organically *diffuse* into edge regions where and when necessary. A key mechanism required to facilitate this is proximity detection.

Monitoring and management

Looking at the previous goals, it becomes clear that proximity is a fundamental concept in IoT and edge computing, and needs to become a first-class-citizen not only in messaging middleware, but edge computing algorithms and systems in general. This would appear to imply that location of brokers and endpoints must be determined in real-time and their physical relationship to each other must be deduced in terms of the local topology. As we will see, proximity can instead be determined by calculating effective network latency. Additionally, routing of messages between elements must balance reliability and performance considerations. Moreover, scalability of proximity monitoring is a key challenge to avoid excessive strain on the network and resource constrained devices.

IoT heterogeneity

A plethora of messaging protocols and client hardware for the IoT exists that may be difficult to replace or update [AFGM$^+$15]. Existing infrastructure should seamlessly integrate with a distributed messaging middleware. Furthermore, middleware control mechanisms need to be completely transparent to end devices.

Message delivery agreements

When it comes to message delivery guarantees, there are always tradeoffs between the level of consistency and the end-to-end latency a system can provide, between transmission reliability and bandwidth utilization, and between transmission delay and processing requirements. We recognize that even a single application can have several types of consistency demands. Take for example an analytics tool in our mHealth application. For some subscribers, it may be critical to receive every single data point in the correct order and without duplicates. For others, duplicate data, or even losing some messages, may not be a problem, but the requirement is ultra-low latency delivery. The MQTT protocol addresses this in part via its “quality of service” concept, which defines the type of delivery guarantee a broker provides: at least once, at most once, and exactly once delivery.

In a single-broker system, implementing these types of guarantees is relatively straight forward, as MQTT shows. However, in our distributed broker system, the type of consistency that the system must provide significantly affects how brokers and clients have to be coordinated during diffusion. For example, when a subscriber to a specific topic is migrated from one broker to another, messages received by the target broker during the reconnection process may not reach the migrating client, as we demonstrate in Section 5.6. This may be problematic for some applications, but a coordination mechanism that provides this type of consistency guarantees can be costly in terms of resource consumption and latency penalties.

5.2 System architecture

Several engineering challenges arise when implementing the message-oriented middleware design we have put forth.

- The ad-hoc distribution of brokers to resource constrained devices requires dynamic and efficient management of distributed subscription tables.
- Clients should be unaware of the dynamic broker network and should be transparently connected to brokers via gateways.
- To determine proximity, the network QoS between nodes needs to be monitored and reported efficiently.
- If proximity between clients and brokers changes, e.g., because a broker leaves or enters the network, client–broker connections have to be reconfigured.
- When brokers provide similar QoS to a set of clients, load needs to be distributed among these brokers.

In this section, we present the architecture of EMMA, a distributed QoS-aware MQTT middleware for edge computing that tackles these challenges, built using design principles from osmotic computing. EMMA consists of five core components: brokers, gateways, the controller, the monitoring plane, and the control plane, illustrated in Figure 5.2. MQTT clients connect to gateways which act as reverse proxies for dynamically connecting clients to brokers. Brokers implement the MQTT server protocol and message forwarding to other brokers. The controller acts as a registry, monitoring hub (via the monitoring plane) and system orchestrator (via the control plane). We explain each component in more detail.

5.2.1 Messaging protocol

EMMA is built around MQTT, a pub/sub protocol that has gained newly found attention in the advent of IoT [AFGM$^+$15, HKM$^+$17]. Due to its lightweight design and minimal network overhead, MQTT is especially well suited for low-bandwidth and low-power environments [AFGM$^+$15]. Many other pub/sub solutions in research rely on their own protocols and models [STM15, BCR14, TS17]. We designed EMMA to act as a transparently distributed MQTT broker, enabling existing IoT software that uses MQTT to be seamlessly used.

5.2.2 Broker network

Brokers implement the pub/sub protocol, and facilitate message dissemination to clients and within the broker network. Because brokers are treated as ephemeral, i.e., they can be spawned or destroyed at any time, their control mechanisms have to be entirely dynamic.

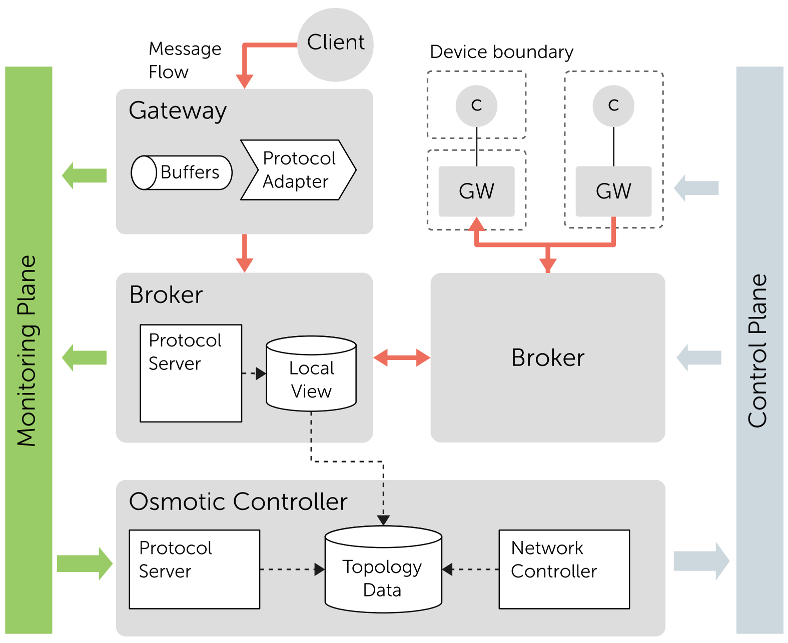


Figure 5.2: The architectural components of EMMA. The osmotic controller orchestrates a network of brokers and gateways via a monitoring and control plane. The gateways transparently connect clients to the system.

This particularly concerns addressing and message routing. To that end, *bridging tables* are synchronized between brokers via the controller. Bridging tables specify which brokers have at least one subscriber to a specific topic. However, maintaining a global view of the network topology at each broker would be impractical. Instead, the cloud maintains a global view, whereas brokers only maintain a local view of the topology and routing tables necessary to operate within their contextual boundary. Brokers also monitor and provide access to key performance indicators, such as resource consumption, message throughput, average routing delay, etc., to provide coordination and load balancing algorithms with necessary data.

5.2.3 Gateway network

Gateways are an integral part of the coordination fabric and provide clients transparent access to the broker network. They serve two main purposes: (1) to seamlessly integrate existing IoT clients into the network, and (2) to enable mobility of clients and brokers. Gateways act as reverse proxies intercept protocol control packets from clients to understand their context (e.g., connection information, Message Delivery Agreements (MDAs) and topic subscriptions) and allow the transparent reconnection of clients to other brokers when adapting to varying network QoS or topology changes. They serve as points in the network to estimate proximity between clients and brokers. Gateways can be deployed on different physical nodes than the clients themselves, and can also handle multiple client connections. Gateways also do simple data processing such as filtering or protocol translation via protocol adapters. Because they are lightweight, gateways can typically be easily hosted by resource-constrained IoT devices. A similar concept was proposed by Luzuriaga et al. [LCC+15], where intermediary buffers decouple message producers and MQTT publisher clients.

5.2.4 Osmotic controller

The controller maintains a full view of the network topology and is responsible for bootstrapping and orchestrating the system. New brokers and gateways register with the controller when they enter the network, and begin continuously reporting data via the monitoring plane. The controller acts as a registry and discovery service for edge resources that can host additional brokers. Brokers are not dependent on the controller to process and forward messages because they maintain a local view of the network topology and routing tables. However, the osmotic controller is required to (1) diffuse clients, i.e., enacting QoS optimization decisions by instructing gateways to connect to other brokers, and (2) diffusing brokers, i.e., autoscaling brokers to edge resources.

The controller maintains the state of the network as a graph data structure. A network monitor component uses data from the monitoring protocol to maintain a graph data structure that reflects the state of the network.Formally, the network $N = (B, C, E)$ is a bipartite graph containing broker nodes B, client nodes C, and connections between clients and brokers as edges E (which we call a *link*). In this context, a client c represents a gateway.

The reconfiguration engine of the controller detects proximity between gateways and brokers based on network latency, and instructs gateways to reconnect to different brokers to optimize QoS. Furthermore, it balances load between brokers that provide similar QoS to clients. The controller also maintains a reduced subscription table, i.e., stores whether at least one client is subscribed to a specific topic at a specific broker.

5.2.5 Monitoring plane

Determining the network topology and monitoring its QoS is a key element for edge-enabled message-oriented middleware. It is particularly important for determining proximity between clients and brokers, as proximity is calculated from network metrics such as latency, routing hops, or physical measurements such as signal strength. Both gateways and brokers need to be able to measure and report on their surrounding network state. We discuss proximity detection, and QoS and demand monitoring in more detail in Section 5.4.

5.2.6 Control plane

A main function of the osmotic controller is to provision brokers to edge resources, and coordinate client diffusion. To that end, the controller issues control commands via the control plane to brokers and gateways. Brokers and gateways provide control endpoints that accept coordination commands, such as an instruction to reconnect to a different broker. We discuss provisioning and network reconfiguration in more detail in section 5.5.3.

5.3 Broker network orchestration

The reconfiguration engine of the controller detects proximity between gateways and brokers based on network latency and instructs gateways to reconnect to different brokers to optimize QoS. It also maintains a reduced subscription table, i.e., stores whether at least one client is subscribed to a specific topic at a specific broker. Internally, the controller uses an event bus architecture that allows it to react to system events such as a brokers or gateways entering the network. A reconfiguration engine is scheduled to run at a fixed interval which examines and reconfigures the network to optimize QoS for clients, and balance load between brokers.

5.3.1 Dynamic topic bridging

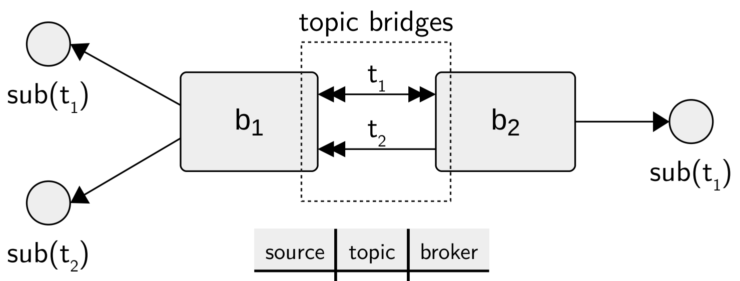
Because we assume dynamic availability of brokers and mobility of clients, we cannot rely on static bridging configurations, e.g., like Mosquitto does [Gar16]. We therefore extend the standard MQTT protocol with dynamic topic bridging. When a client publishes a message into a given topic, brokers first broadcast the message to all connected subscribers according to the MQTT protocol, and then forward the message to other brokers that have at least one subscriber to the respective topic. Brokers inform the controller about changes of their local subscription tables when necessary (e.g., if a client subscribes to a new topic), and the controller propagates relevant changes to the other brokers.

When a client publishes a message into a given topic, brokers first broadcast the message to all subscribers according to the MQTT protocol, and then forward the message to other brokers. Because we generally do not know where, when, and into which topic messages will be published, brokers are required to know about all topic subscriptions of other brokers.

Figure 5.3 illustrates a bridging scenario and the corresponding bridging table maintained by the controller.

Bridging tables are a form of high-level or application-specific routing table that facilitate the topic bridging approach. To maintain bridging tables at the controller, we use the Redis[1] key-value database. Brokers inform the controller about changes of their local subscribers only when necessary. Figure 5.4 shows the end-to-end subscription operation, beginning with the client issuing an MQTT `SUBSCRIBE` packet. The gateway stores the control packet and forwards it immediately to the broker. Stored control packets are used to replay subscription requests during a reconnect to a broker. The broker updates

[1]`https://redis.io`



source	topic	broker
b_1	t_1	b_2
b_2	t_1	b_1
b_2	t_2	b_1

Figure 5.3: Bridging tables

its subscription tables, and, if this is a new subscription, informs the controller that the broker now has at least one subscriber to this topic. In a subsequent step, the controller updates its bridging tables and propagates the changes to all other brokers. We use Redis' keyspace notification feature to propagate bridging tables to a broker only when they are relevant to the respective broker. Brokers maintain a locally cached view of the bridging tables to facilitate high throughput access.

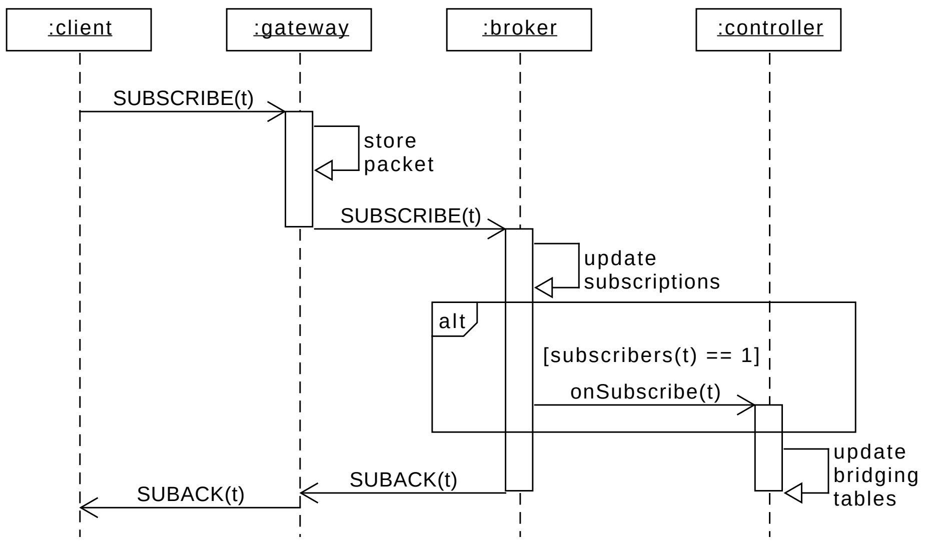


Figure 5.4: Interaction between EMMA components during an MQTT subscribe operation

Table 5.1: Packets of the migration protocol

Name	Description	Bytes
RECONNREQ	Request a gateway to reconnect to a broker	47
RECONNACK	Gateway acknowledges the reconnect	47

5.3.2 Orchestrating client connections

As described earlier, a gateway's main purpose is to hide EMMA from the actual clients, and facilitate the transparent connection to brokers in proximity. Instead of connecting to a broker directly, MQTT client traffic is tunneled through a local gateway. When a client sends an MQTT `CONNECT` packet to a gateway, the gateway initially queries the controller for a broker to connect to. The gateway stores all MQTT control packets sent by the client related to establishing a broker connection and subscriptions (`CONNECT`, `SUBSCRIBE`, `UNSUBSCRIBE`). The controller may decide at runtime to migrate the connected clients to a different broker using a migration protocol To that end, we implemented a simple migration protocol (see Table 5.1), which was in subsequent research extend to perform migrations in a transactional way [Gei19]. The controller sends a `RECONNREQ` packet to the gateway via the monitoring protocol containing the host and port of the target broker. The gateway asynchronously opens a connection to the new broker, disconnects from the old broker once the new connection is established, informs the controller by sending a `RECONNACK` packet, and then places the stored control packets into the send-buffer dequeue. To avoid packet loss during reconnection of clients to new brokers, a gateway maintains, for each client–broker tunnel, two buffers that buffer incoming messages from the client and broker respectively. Any messages sent from the client during a reconnection process are buffered into a dequeue. Once the connection to a new broker is established, the recorded MQTT control packets are placed into the head of the dequeue, the old connection is closed, and the buffer, which includes the control packets, is flushed. Figure 5.5 shows sequence diagrams of the connection and reconnection procedure.

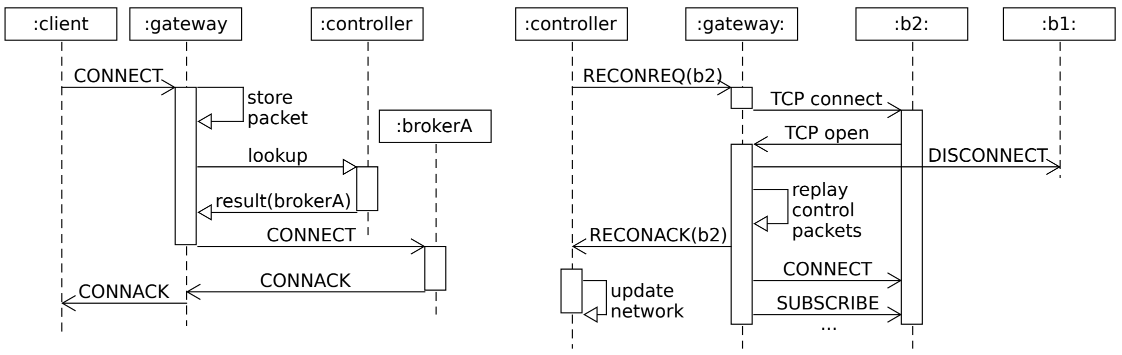


Figure 5.5: Connection (left) and reconnection (right) procedures

5.4 Proximity awareness

Proximity is a fundamental concept in edge computing systems. To build proximity aware systems, we need (1) a good quantification of the proximity between brokers and resources, (2) a scalable way to continuously monitor this proximity, and (3) a way to monitor demand of clients. In other distributed systems, such as Content Distribution Networks (CDN), closeness between nodes is often defined and determined by geospatial location, routing hops, or round-trip times (RTT). For (1), we argue that, because the main goal is to optimize end-to-end latency between clients, using the effective network latency is the best way of determining proximity between clients and resources. In Chapter 7 we discuss other manifestations of proximity. Additionally, for determining proximity between clients and brokers, brokers can add their average message buffering delay to the effective network latency to calculate a more precise RTT value. After all, a "close" broker on a congested resource can behave as if it were physically much farther away. However, in contrast to CDN, where proximity is determined on a per-request basis via DNS probing, osmotic MOM needs to continuously monitor the effective network latency between all nodes, which introduces serious scalability challenges. Simple monitoring solutions can cause high network overhead that becomes difficult to manage, as we demonstrate in detail in Section 5.6.3. For (2), a scalable solution would be a synthesis of predictive network location approaches[DCKM04, WSS05] with concepts such as interest management, where the frequency of monitoring is concentrated to relevant regions. For (3), a combination of throughput monitored from brokers and gateways can be used to estimate demand for a specific region, topic or content type.

5.4.1 Network QoS monitoring protocol

To reason about the network and possible latency optimizations at runtime, network QoS needs to be continuously monitored and analyzed. To that end, we implement a basic distributed network monitoring protocol. In this thesis, we develop a simple implementation to create a baseline. Using virtual network coordinate systems and interest management is explored in [Bru21]. We implement an asynchronous pull-type monitoring, where the controller issues a request and initializes a measurement process. For sending and receiving protocol packets, each component exposes a UDP port addressable by the controller. Messages of the protocol are encoded in a lightweight binary format, summarized in table Table 5.2.

Table 5.2: Packets of the monitoring protocol

Name	Description	Bytes
QOSREQ	Network QoS measurement request	13
QOSRESP	Network QoS measurement response	9
PINGREQ	Ping request	5
PINGRESP	Ping response	5

The general process of the protocol is illustrated in the sequence diagram of Figure 5.6, and described as follows. The controller sends a `QOSREQ` packet containing an ID, and the measurement target (a broker) to a gateway. The gateway then sends 10 `PINGREQ` packets to the broker in an interval of 250 ms. Each `PINGREQ` packet contains an ID, which is returned by the broker in a `PINGRESP` packet. The gateway records the sent and received timestamps for each packet and sends back the average latency as a `QOSRESP` packet to the controller.

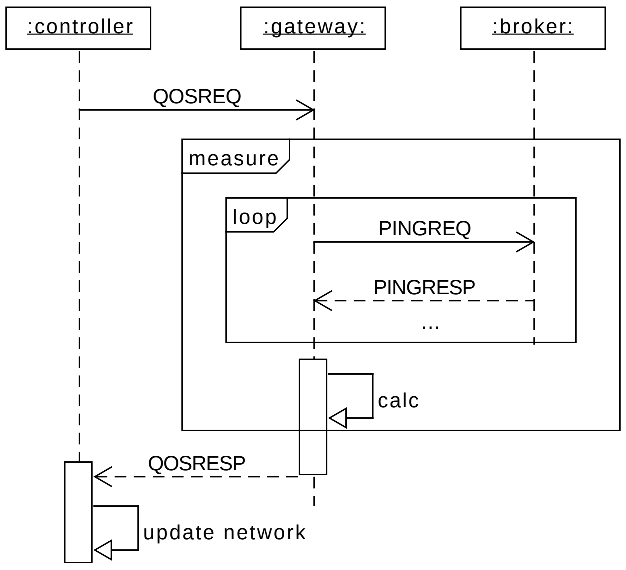


Figure 5.6: Protocol interaction of network QoS monitoring

5.4.2 Latency groups

When brokers provide similar QoS to clients, it is important to (1) avoid migrating clients due to slight variability of latency, and (2) balance load between those brokers. To that end, we stratify brokers into latency groups, based on the premise that connecting to any of the brokers in a group is acceptable in terms of QoS. Figure 5.7 illustrates the concept. Latency groups are defined as millisecond-intervals I on a log-scale. By default, we set $I = \{[0,2),[2,5),[5,10),[10,20),[20,50),[50,100),[100,200),[200,500),[500,1000),[1000,)\}$. That means, for example, if a broker b provides a client c a latency of 4, it would fall into the interval $I_2 = [2,5)$ and therefore into group 2. For balancing load in a group we implemented a simple strategy based on the amount of clients connected to a broker.

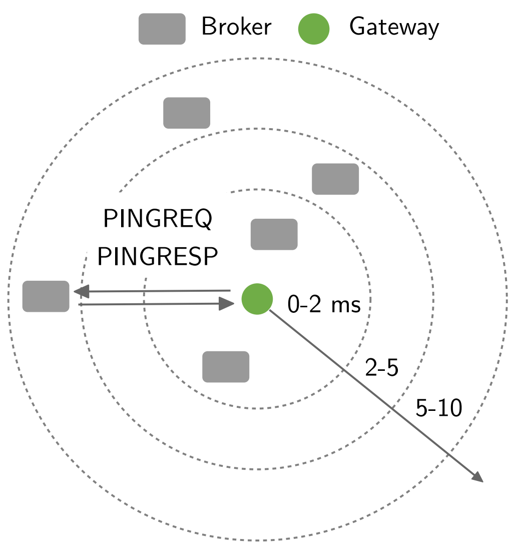


Figure 5.7: Brokers are placed into latency groups illustrated as concentric circles around gateways

5.5 Elastic diffusion

As we have established, cloud-based message brokering cannot satisfy the stringent QoS requirements of IoT applications. This particularly concerns device-to-device communication in close proximity. Instead, brokers at the edge should facilitate low-latency communication for devices in close proximity, and forward messages to the cloud for further dissemination. This already works well for simple configurations and static infrastructure [Gar16]. However, as our motivational scenario illustrated, edge computing applications require a dynamic system that can adapt to changing network QoS and topology, while still allowing centralized configuration and management. During runtime, when clients at the edge require low-latency communication, the broker network needs to be expanded by provisioning brokers to edge resources dynamically. The routing topology needs to be updated, and clients in close proximity need to be migrated to the new broker. When client communication stops, these brokers can be discarded, and the network may shrink again. We call this process *elastic diffusion*. In this section, we first describe the key principles, and then proceed to explain individual mechanisms necessary to achieve elastic diffusion.

5.5.1 Principle

Elastic diffusion is inspired from the physical process of *diffusion* (from the Latin word *diffundere*, meaning “to spread out”), and the cloud computing concept of *elastic autoscaling*. Figure 5.8 illustrates the concept. Starting from a single configuration in the cloud, i.e., an osmotic controller and a root broker, the system grows and shrinks the broker network during runtime to adapt to changing network topology, client mobility and demand, and resource availability. Brokers diffuse to edge resources when there is potential to optimize client QoS. Similarly, clients diffuse to edge brokers to reduce end-to-end latencies, and to balance load between brokers. A key mechanism to facilitate this is a concept we call osmotic pressure.

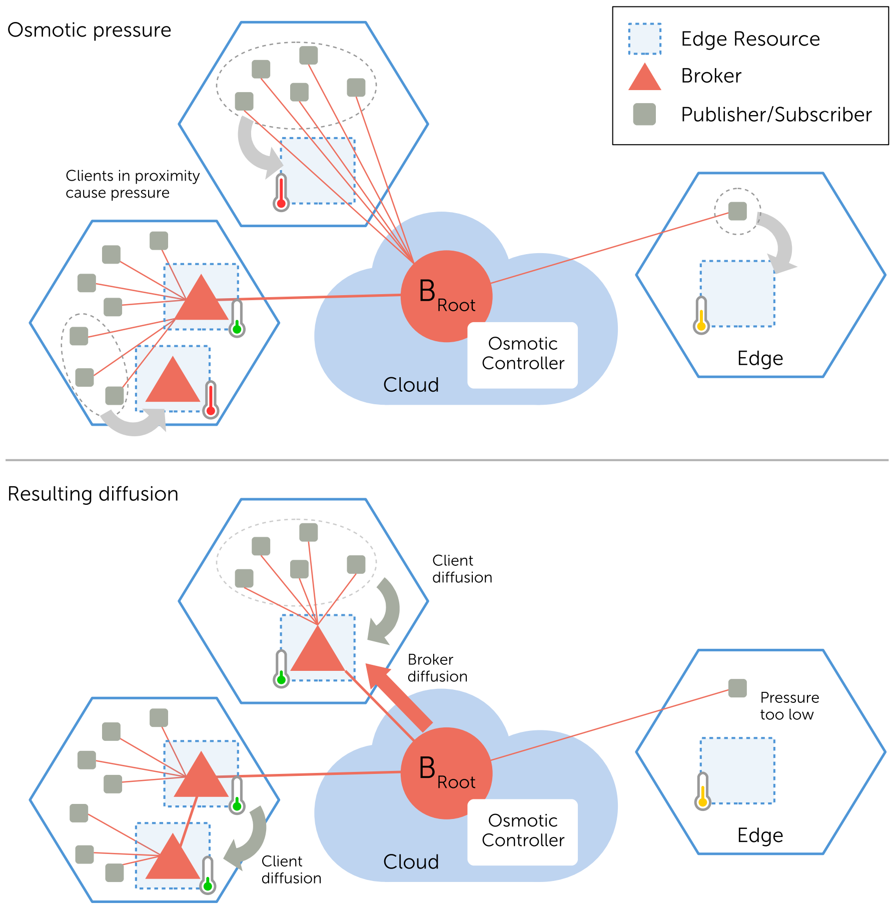


Figure 5.8: Osmotic pressure facilitates controlled diffusion of brokers and clients to the edge. Clients exert osmotic pressure on local resources depending on their proximity and demand. The osmotic controller aims to balance osmotic pressure throughout the system.

Osmotic pressure

To facilitate elastic diffusion, we first need an aggregated quantification of proximity and demand. To that end, we define a metric inspired by a core concept of osmosis and diffusion: osmotic pressure. Similar to the biological process, in our system, osmotic pressure causes brokers to diffuse onto edge resources. Clients in close proximity to edge resources cause a "pulling" effect on those resources. When the pressure exceeds or drops below a given threshold θ, the system deploys or removes brokers from these resources. Similarly, brokers deployed on edge resource cause a "pulling" effect on clients in close proximity, which causes a reconfiguration of client-broker connections to optimize QoS for those clients. For edge resources, we calculate osmotic pressure based on the number of clients and their proximity to the resources. Osmotic pressure is high when many clients exist in close proximity to a resource. The exerted pressure can increase with the message throughput of these clients, i.e., if local demand is high.

Broker diffusion

Like in osmosis, where solute molecules exert osmotic pressure, clients in close proximity of edge resources exert osmotic pressure on these resources, causing an imbalance in the network. To balance the osmotic pressure, the osmotic controller reacts by deploying a broker to the resource. We assume that edge resources provide some form of container-based virtualization, e.g., Docker, which allows the controller to deploy new brokers as containers. The new broker is integrated into the network and the controller reconfigures the routing overlay, i.e., dynamically creates topic bridges to forward messages to the cloud or other brokers if necessary. When the osmotic pressure drops, e.g., because clients move to other locations or leave the network, brokers may be discarded by the system.

Client diffusion

Clients' connections are migrated into edge regions based on their proximity to brokers deployed at the edge. If brokers are deployed at the edge due to osmotic pressure, clients will diffuse there. Because osmotic pressure is calculated using both proximity and broker performance indicators, we get a twofold effect: clients will experience better end-to-end latencies as link usage is minimized, and load is balanced between brokers that provide similar QoS to those clients. When brokers are discarded because of low osmotic pressure, or because of a broker outage, clients diffuse back to a broker closer to the cloud.

5.5.2 Calculating pressure

Osmotic pressure gives us a way to abstract scaling procedures. However, the effectiveness of these procedures is contingent on a meaningful calculation of osmotic pressure p. We present two concrete proposals. One proposal is based solely on the number of clients and their proximity, the other additionally takes into account the distance to other brokers. Distance is considered a proxy for QoS. Considering additional metrics such as message throughput to gauge local demand is left as future work.

Local distance-based calculation

We calculate the pressure p_n for each node n (resources or brokers) based on the number of clients and their distance to the node. The pressure is inversely proportional to the distance, in other words: many clients in close proximity cause a high pressure. $d(a, b)$ denotes distance between two nodes network nodes.

$$p_n = \sum_{c \in C} \frac{1}{d(c,n)} \tag{5.1}$$

We relate the local pressure p_n to global values by defining $\hat{p}_n = \frac{p_n}{p_{\text{root}}}$ Where p_{root} is the pressure of the root broker. That way, the pressure value for n is normalized to the number of clients and their overall dispersion in the system.

Global difference-based calculation

The previous calculation does not consider the current network configuration. We find it useful to adjust the pressure of a node depending on the current connection states of other clients. In other words, the pressure on workers should be high, if clients in proximity to the worker are connected to brokers that are far away, but should be low if clients are connected to brokers that are closer than the worker. To that end, we need to distinguish pressure p for resources (worker nodes $n \in N$) that can host brokers, and active brokers $b \in B$.

For resources, the pressure is calculated as follows, where b_c denotes the broker the client is currently connected to:

$$p_n = \log\left(1 + \sum_{c \in C} \max(0, d(c, b_c) - d(c,n))\right) \tag{5.2}$$

Intuitively, the sum calculates the overall reduction in distance for clients if a broker were to spawn on node n, and is therefore a heuristic for the potential QoS improvement. Hence, the pressure for workers is used for making *scale-out* decisions. If the client is already connected to a broker that is closer than the node, the difference would be negative, so we ignore it. The logarithm is applied to reflect the logarithmic scale of the latency groups we defined in Section 5.4.2. We add 1 to guard against log(0) cases. Figure 5.9 illustrates an example. The notation is the same as in Figure 5.8.

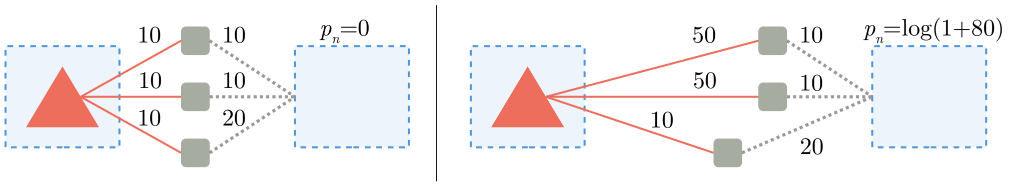


Figure 5.9: Example of how pressure changes based on the distance of clients to other brokers. Values above lines indicate distances (or QoS) between nodes.

For brokers, the calculation is similar.

$$p_b = \log\left(1 + \sum_{c \in C_b} \max(0, d(c, \text{nextBroker}(c)) - d(c, b_c))\right) \tag{5.3}$$

However, because the value is used for *scale-down* decisions, we use for the difference calculation the function nextBroker(c), which refers to the broker the client would be connected to if the current broker b_c is removed. Intuitively, the value is a heuristic for the potential QoS degradation if the broker is removed. Since we are using a greedy heuristic for our client–broker connection management (see Section 5.5.3), where clients are always connected to the closest broker, nextBroker(c) would return the second closest broker. Additionally, we only consider the clients that are connected to the broker, i.e., $C_b = \{c \in C | c_b = b\}$.

5.5.3 Network reconfiguration and QoS optimization

The network is reconfigured by first examining the current QoS of the network, selecting potential broker candidates for each client in a way that will optimize QoS for those clients, and then migrating clients to their designated brokers. The reconfiguration engine of the controller is scheduled to run at a fixed interval of 15 seconds. Figure 5.10 shows the state of a network before and after a reconfiguration. Values of links indicate the proximity (latency in milliseconds). Arrows indicate message flow. Clients c_1 and c_2 are migrated from broker b_1 to b_2. A topic bridge between the two brokers is created automatically through the subsequent reconnection and subscription procedures described earlier.

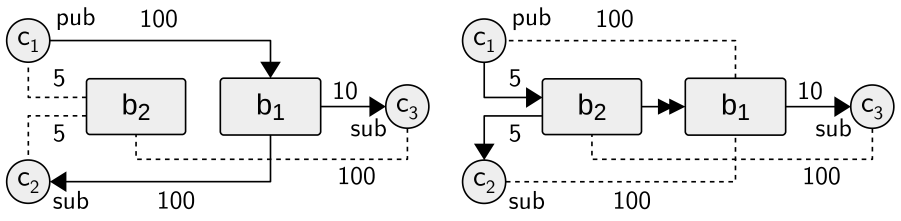


Figure 5.10: Network before and after reconfiguration

The basic algorithm for reconfiguring the network is outlined in Algorithm 5.1. The network $N = (B, C, E)$ is a bipartite graph of brokers B and clients C as described in Section 5.2.4. To avoid migrations when no significant load balancing would occur, we introduce the migration threshold θ, which defines the minimum percentage of connections that have to change within a group of candidate brokers in order to trigger a reconnect. By default, we set $\theta = 0.1$.

5.6 Evaluation

To show the efficacy of our approach, we implement a prototype of the EMMA system and evaluate it in a real-world testbed. We run an experimental scenario that emulates the scenarios described in Section 5.1 to show that the system can (1) deal with clients and brokers unexpectedly entering and leaving the network, (2) dynamically bridge topics

Algorithm 5.1: Broker–client network reconfiguration

```
   Input: A network $N = (B, C, E)$, a migration threshold $\theta$
 1 foreach $c \in C$ do
 2     $b_c \leftarrow$ currentBroker($c$)
 3     $B'_c \leftarrow$ brokersInLowestLatencyGroup($N$, $c$)
 4     if $b_c = b'_c$ then
 5         continue
 6     end
 7     if $b_c \in B'_c$ then
 8         $\delta = \theta \cdot \sum_{b \in B'_c}$ connCnt($b$) if connCnt($b'_c$) $+\delta \geq$ connCnt($b_c$) then
 9             continue
10         end
11     end
12     migrate($c$, $b'_c$)
13 end
```

only when required, (3) reconfigure connections to optimize latency for clients in close proximity, and (4) balance load between brokers that provide similar QoS to clients.

The evaluation of different osmotic pressure models presented in Section 5.5.2, is out of scope of this thesis, but further explored in [Bru21].

5.6.1 Experiment setup

We now present the details of our evaluation environment, the experimental scenario, and deployment details.

Testbed

Our evaluation environment consists of multiple Amazon EC2 virtual machines that span three different AWS data centers. This creates a realistic testbed for testing the system under geographic dispersion and the incurring network latencies. We use eu-central (Frankfurt) for our controller, an initial broker, and a single client. Regions eu-west (Ireland), and us-east (Ohio), host multiple gateways and clients, and we reserve VMs for brokers to be spawned during runtime. Figure 5.11 illustrates the setup and shows latencies between regions. Latencies within regions are in the sub-millisecond range.

We use the following EC2 instances types for the components. The controller runs on a t2.large (2 vCPUs, 4 GiB RAM) instance. For broker nodes, we use t2.medium instances (2 vCPUs, 2 GiB RAM). Gateways and clients share t2.micro instances (1 vCPUs, 1 GiB RAM).

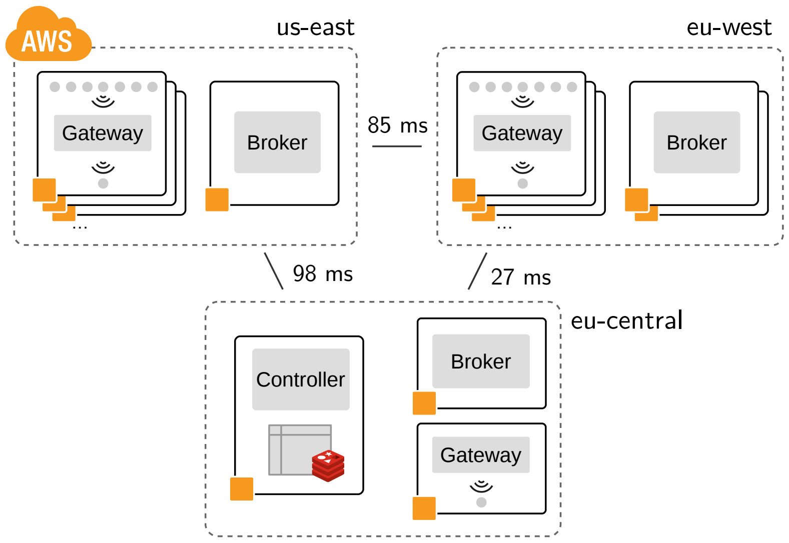


Figure 5.11: Evaluation environment for the EMMA system in AWS EC2

Scenario

The experimental scenario emulates the real-world edge computing scenarios presented in Section 5.1. In this particular experiment, we spawn client groups across two regions. Each client group consists of 10 VMs, each hosting a gateway, 1 subscriber and 7 publishers that exchange messages in a topic named like the region they are deployed in, namely eu-west and us-east. To show that the system can dynamically create topic bridges when necessary, a publisher and subscriber pair is deployed in each region that communicate in the topic *global*. A single subscriber to this topic is also deployed in eu-east where the initial broker and controller reside. To demonstrate the orchestration mechanisms, we trigger the following events manually at runtime:

1. Clients appear that communicate in topic *global* (one publisher and subscriber in both us-east and eu-west. One subscriber in eu-central)
2. Client group appears in the us-east region
3. Broker spawns in eu-west (1)
4. Client group appears in the eu-west region
5. Broker spawns in us-east
6. Broker spawns in eu-west (2)
7. Subscriber to topic *global* in eu-central disappears
8. Broker shuts down in us-east

Clients & load generation

For the purpose of generating load and recording message statistics, we developed a general purpose framework for benchmarking publish–subscribe systems which is also open source[2]. It provides an interactive shell that allows to spawn different types of MQTT clients and control message publishing frequency. For this particular experiment we used the XenQTT[3] client library. Messages generated by the application contain a payload in the JSON format that has a total of 118 bytes (including JSON overhead). An example is shown in Listing 5.1. They contain a UUID, a timestamp the message was sent, and a dummy payload with 14 bytes as placeholder for a sensor reading.

```json
{
  "id": "3e14463d-7dae-4b0a-8438-316bc262514f",
  "timestamp": "2017-08-30T16:37:18.606Z",
  "payload": "14 extra bytes"
}
```

Listing 5.1: Example sensor message payload sent by a client

We configure each publisher to generate messages at a fixed rate of 10 messages per second. To avoid noise resulting from interfering IO operations, we store message data in-memory and save the data to disk once the client connection has been terminated and the benchmarking runtime exits.

5.6.2 Experiment Results

The following figures show the main results from our experiment. Figure 5.12 shows the message throughput of the brokers during their lifetime. The circles and dotted lines indicate the events described previously. The x-axis indicates the time (in minutes) of the experiment, each labeled tick is 60 seconds apart. Figure 5.13 and Figure 5.14 show the average end-to-end latency of messages in the respective topics, aggregated every second over all subscribers of that topic.

Throughput & load balancing

Figure 5.12 shows how the rebalancing mechanism and the dynamic bridging approaches behave. At event 4, when the client group appears in eu-west, the output rate shows that it takes two rebalancing iterations to fully balance connections between the two active brokers. Because messages are bridged between the two brokers, the input rate does not change. However, as the output rate makes apparent, the balancing can significantly reduce the strain of multicast on a single broker. The input rate of the eu-central broker after event 7 shows that, once there are no subscriptions to a topic at a specific broker, the subscription tables are propagated and messages are no longer bridged to that broker.

[2] `https://git.dsg.tuwien.ac.at/emma/pubsub-benchmark`
[3] `https://github.com/TwoGuysFromKabul/xenqtt`

At event 8, when the broker in us-east shuts down, clients previously connected to that broker are immediately migrated to other available brokers. All three currently running brokers are in the same latency group for the clients deployed in the us-east region. As the output throughput graph shows, load is balanced between the three brokers.

Topic latencies

Figure 5.13 shows how rebalancing and migrations affect the end-to-end latencies between clients. In particular, the graphs reveal the cost of migrating clients. At event 3, when a new broker spawns and migration of clients begins, the message latencies spike. This is the result of a combination of the gateway buffering mechanism during reconnect, and the Java warm-up phase of newly spawned brokers. However, as the graph shows, the latency stabilizes a few seconds after the migration is complete, and the load is shared between the brokers eu-central and eu-west 1. The second rebalancing iteration has a less drastic effect.

At event 4, the graph for the eu-west topic shows that clients in the region immediately connect to the broker closest to them. Consequently, the load balancing engine moves some of the clients connected from us-east back to the eu-central broker. In this case, it took two balancing iterations to fully balance the connections.

At event 5, when the broker in us-east is spawned, all clients in the region migrate to that broker. This includes the publisher and subscriber in that region communicating via the *global* topic, which is why the average latency also drops dramatically. A major source of latency in this topic is the round-trip time between the us-east region and the brokers in eu regions. Once clients within the region communicate via a local broker, only messages that have to be forwarded are sent over the us–eu link. At event 8, when the broker in us-east shuts down, latencies spike due to reconnection buffering, and then stabilize at their previous levels.

Figure 5.14 shows the time window between event 5 and 8, where both client groups have a broker in their respective region. As the graphs show, the latencies average at around 3–5 ms. The variance in the eu-west topic shows how load balancing affects latencies for devices in close proximity. These results also indicate that messages are bridged efficiently, meaning that the system can provide low end-to-end latencies for clients in close proximity, and forward messages to other brokers that provide similar QoS with minimal overhead.

Global message dissemination

In each region, one subscriber to the *global* topic is deployed. Figure 5.15 shows the average end-to-end latency for each subscriber in the respective region during the experiment. As the us-east graph shows, messages sent from the us-east region have, at first, the highest round trip time, causing the high fluctuation in that region. At the later stages after event 5, where a broker is present in each region, message latencies average at roughly the average link latencies between clients and their local brokers. Looking at the subscriber

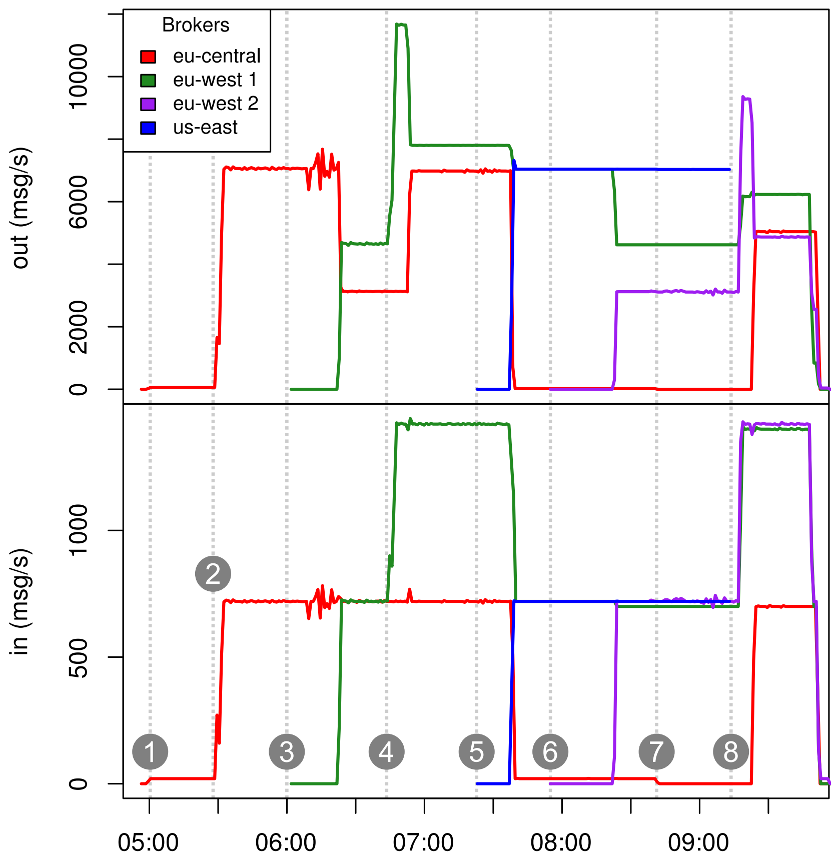


Figure 5.12: Broker message throughput

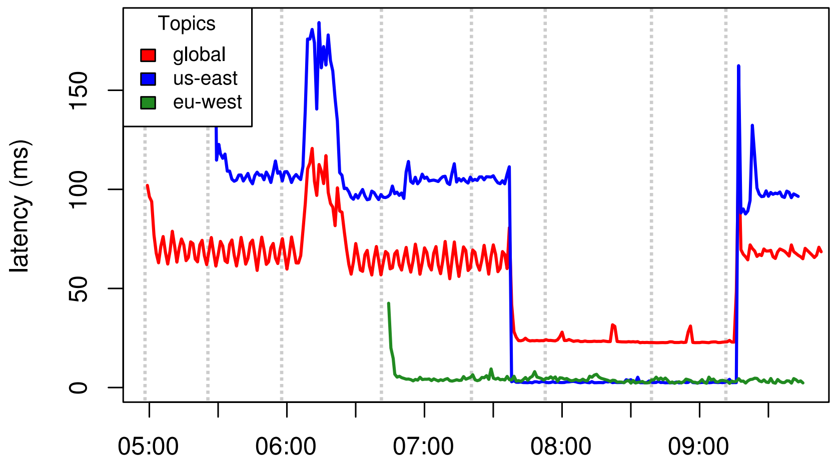


Figure 5.13: Aggregated topic latencies

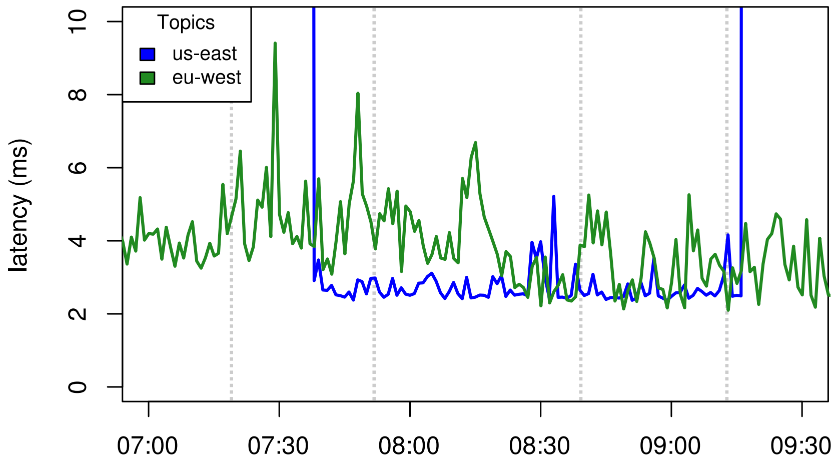


Figure 5.14: Latencies during intra-region communication

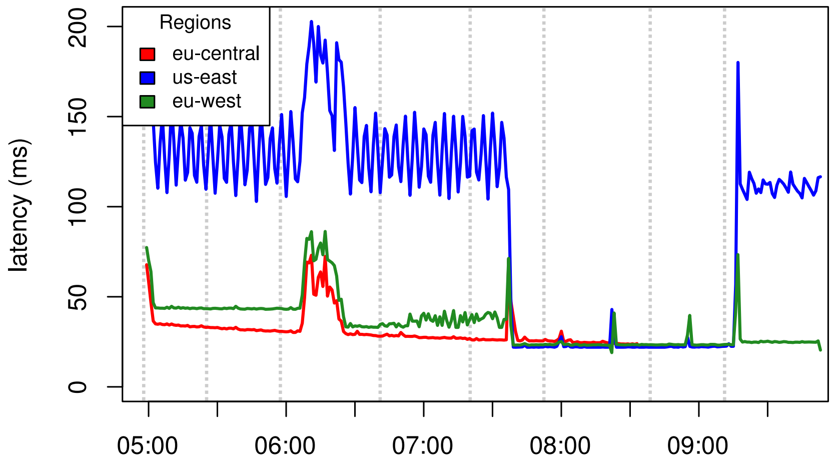


Figure 5.15: Average latency for *global* topic subscribers

in eu-central, we also observe that, even when topics are bridged from other regions, the latencies do not significantly change. This shows that the overhead of bridging messages to geographically dispersed locations is minimal even under broker load.

Message loss

The gateway buffering approach avoids message loss for publishers during a reconnection procedure. However, messages published while subscribers are migrated may not be delivered to these subscribers. Figure 5.16 shows the total amount of undelivered messages in both client groups at given points in time during the experiment. The boxes show the distribution of undelivered messages across the 10 subscribers of the respective group. Boxes are drawn in points in time where the amount of undelivered messages changed.

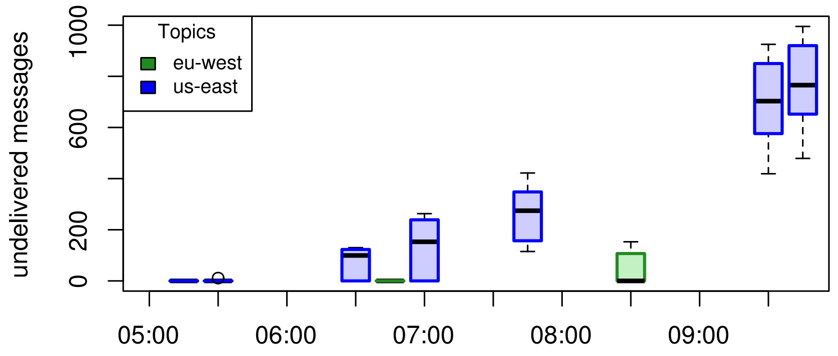


Figure 5.16: Message loss for clients in the respective regions

Each client group consists of 70 publishers that publish at a frequency of 10 msg/s, totaling at 700 msg/s. In the us-east topic, a total of 177,269 messages were published. As the last box for the topic shows, subscribers experienced a total message loss of about 765 messages, i.e., 0.43% per subscriber. For the *global* topic, message loss was minimal.

Providing consistency guarantees for mobile subscribers necessitates a transactional migration. To that end, in [Gei19], we introduced a consensus protocol for EMMA, that consists of additional control packets, a migration sequence, and state synchronization procedures. While this removes message loss, the transactional migration adds latency. A handoff requires eight synchronous control messages and is therefore sensitive to network latencies. The duration of the state synchronization depends on the number of messages the subscriber is behind. With an average of 30ms round trip time between the client and the involved brokers, the migration procedure may take between 500ms and 3000ms.

5.6.3 QoS monitoring network usage

QoS monitoring comes at the cost of using the network's bandwidth. We present a sample calculation that shows that the use grows linearly with the amount of brokers and clients in the network.

Measuring the QoS between two nodes involves sending a total of 22 UDP packets. A UDP packet in IPv4 Ethernet has 48 bytes of overhead. Hence, the UDP overhead for each measurement totals at 1058 bytes. The request/response packets sent between gateway and controller are 13 byte and 9 byte respectively. Then, 10 ping messages are sent to the broker, and, ideally, 10 response messages are returned, totaling 100 bytes. Including the UDP overhead, this means a total of 1180 bytes per measurement. Our prototype implements a fixed rate approach, where QoS is measured every 15 seconds. In a network of 100 brokers, this would mean roughly 7.9 kB/s network usage for a gateway. Figure 5.17 shows the usage over the entire network depending on the number of gateways (log scale) and brokers in the network, assuming a measurement interval of 15 seconds.

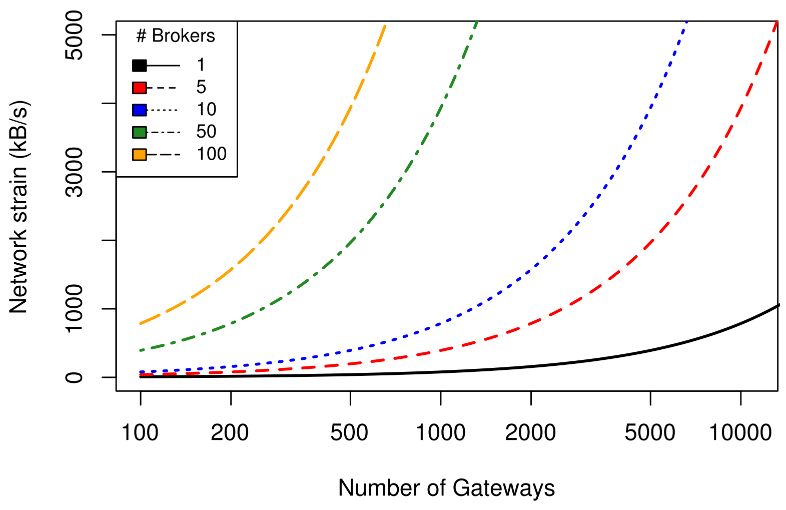


Figure 5.17: Overall network usage in kB/s by QoS monitoring

There is a lot of potential to optimize these values. Currently, every gateway requests QoS for every broker in a fixed interval. It may not be necessary to do so. For example, the frequency of QoS measurements could adapted to the initial amount of measured latency, such that brokers in close proximity are updated more frequently than others. Another approach would be to use virtual network coordinates, like Vivaldi [DCKM04].

To that end, in [Bru21], we adapted the EMMA monitoring protocol to use Vivaldi coordinates. Vivaldi uses the Euclidean distance model, and computes synthetic multi-dimensional coordinates to predict inter-host latencies. Each measurement (in our case, ping packets) reduces the error of the estimated coordinate, meaning that the prediction error may be significant when there are few measurements. We no longer need to the fully-connected network graph to estimate distances between nodes in the network. In our initial implementation, a client pings five random brokers to get initial coordinates, and then pings the closest five to increase the accuracy of the ones in lower latency groups (which are more sensitive to errors). This way, the network strain scales linearly with the number of clients in the system. In preliminary experiments, using the same scenario that we described in Section 5.6.1, we were able to reduce the network strain by 34%, while maintaining effectively the same latency distribution. More research is necessary to fully understand the effect of prediction errors on performance in different scenarios.

5.7 Related work

Messaging middleware, and pub/sub systems in particular, have enjoyed immense research interest over the past decades [EFGK03, HKM+17]. Systems like Hermes [PB02] or PADRES [JCL+10] have led to an entire body of research dedicated to optimizing and enabling robust message dissemination in complex peer-to-peer overlay networks. The advent of cloud computing and the IoT has changed the landscape of real-world messaging solutions dramatically. Over the past few years, many commercial cloud-based messaging solutions have emerged that target IoT platforms. Amazon's AWS IoT [Bar15], Microsoft's Azure IoT Hub [Sta19], or Google Pub/Sub [KG15], all aim at providing transparently scaling IoT device integration via pub/sub messaging. Although these solutions can deal with massive amounts of devices and messages, they cannot satisfy the stringent QoS requirements imposed by many IoT scenarios, as they completely disregard proximity and edge resources. In particular, research has shown that current protocols and solutions can not adequately deal with the scalability and QoS requirements of modern IoT scenarios [HKM$^+$17].

Scalability of pub/sub middleware under fluctuating load has primarily been addressed in cloud-based solutions. Centralized systems such as Dynamoth [GSGKK15] achieve scalability by providing load balancing in a broker cluster, and elasticity mechanisms for adding and removing broker nodes on demand. The evaluation shows that response times can be stabilized even under varying load. These cloud-based systems do not consider proximity of clients or latency incurring from link usage. Load balancing for edge computing is a fairly unexplored topic, and has only recently gained attention in the edge computing community [SYY$^+$17]. EMMA's network re-configuration approach includes a connection-based load-balancing approach, implicitly trading off broker load and network latency.

QoS awareness in pub/sub overlay networks has been addressed in different works spanning the past two decades [CAR05, CS05, KKY$^+$10, OR11, BCR14]. Specifically, these approaches focus on techniques for managing complex overlay networks using contextual QoS information [OR11], and efficient message routing within these networks [CAR05]. Contributions include optimal path selection for message routing based on QoS criteria and supporting mobility by providing efficient on-line re-configuration of overlays [BCR14]. Our approach does not require complex routing or overlay network management, as we apply a simple direct delivery model [GSKK17], which we discuss later. Generally, research on overlay network tends to focus on formal system models, and ignore system details such as interaction between components during re-configuration, or concrete proposals for QoS monitoring.

Some open-source solutions, such as JoramMQ [Sca14], Mosquitto[4] and HiveMQ[5], have shown tentative efforts to provide basic mechanisms for enabling real-world edge computing applications. For example, JoramMQ [Sca14] can model a hierarchical broker tree to allow low-latency communication in pre-defined subsystems. Mosquitto and HiveMQ can be configured, at deployment time, to bridge topics, i.e., forward messages of a specific topic, to a centralized broker [Gar16]. However, these mechanisms are all static in nature and do not address mobility or changing resource availability. EMMA introduces a dynamic orchestration mechanisms that can deal with high-churn network configurations.

[4]`https://mosquitto.org`
[5]`http://www.hivemq.com`

Few efforts have been made to engineer complete solutions for QoS aware pub/sub systems that address the challenges of IoT and edge computing. An et al. [AKGH17] present PubSubCoord, a cloud-based coordination system for a distributed broker network. The overlay layer is strictly structured into *edge* brokers and *routing* brokers, coordinated via a layer of ZooKeeper nodes. In a non-peer-reviewed work, Abdelwahab and Hamdaoui present FogMQ [AH16] which supports migration of broker *clones* at runtime to the edge, thereby enabling low-latency data analytics. MultiPub [GSKK17] manages a broker cluster that spans multiple cloud regions, and optimizes latency for clients based on their proximity to a region, by moving topics to those regions. Our message dissemination strategy is similar to MultiPub's direct delivery model. However, regions and brokers on regions are configured statically, only topic migration and client–broker connections are dynamic. EMMA goes one step further in that it considers brokers on arbitrary edge resources and dynamically re-configures client–broker connections at runtime. The presented solutions are also unspecific about monitoring strategies to facilitate proximity detection.

5.8 Summary

Efficient device-to-device communication plays a fundamental role in edge intelligence systems. In this chapter, we presented EMMA, an elastic, QoS-aware MQTT middleware that is designed for geo-distributed environments. We have shown how principles of osmotic computing can be applied to message-oriented middleware, to provide a distributed network of brokers that elastically diffuse to edge resources on demand. Gateways allow existing MQTT client infrastructure to transparently connect to the system. We demonstrated that EMMA can provide low-latency communication for devices in close proximity, while allowing message dissemination to geographically dispersed locations at minimal overhead costs. Unlike cloud-based IoT platforms that completely neglect the proximity of devices, EMMA makes proximity a first-class citizen, both for deploying new brokers, as well as optimizing responsiveness for clients. To that end, we implemented a proximity monitoring protocol, that continuously updates a global view of the network. In subsequent research [Bru21], we showed how using virtual network coordinate systems can reduce monitoring overhead significantly. Our proximity-aware network reconfiguration mechanism enables client mobility, dynamic broker provisioning, and broker load balancing. Our system can serve as stand-alone messaging middleware to facilitate device-to-device communication in IoT environments, and can also complement complex edge computing applications that rely on message brokers, such as distributed real-time data analytics applications.

The idea of elastic diffusion, i.e., using proximity to jointly drive autoscaling and load balancing decisions, translate to other edge computing systems, especially serverless systems that we will discuss in the next chapter. In these systems, requests to serverless functions are typically routed on application layer through a centralized API gateway or load balancer. That may not be an issue if the round trip time of the request is bound by computing time rather than by network latency from the client to the API gateway.

When workloads are network bound, however, centralized API gateways cause the same problems as centralized message brokers in edge scenarios. The idea of elastic diffusion could be applied to serverless function and API gateways to address this issue. API gateways have to be distributed in the same way that we distribute gateways in EMMA. Serverless functions then have to be scaled, not only based on resource usage, but also on the proximity to clients invoking the functions. We are exploring and evaluating this in ongoing research [Pal21].

CHAPTER 6

A serverless platform for edge computing

Serverless computing has emerged as powerful system abstraction that frees application developers and operators from the operational complexities of computing infrastructure. Developers provide their application code, and serverless platform operators take care of provisioning resources, running and scaling the application, and orchestrating the system. The characteristics of edge computing infrastructure we described in Chapter 3 dramatically increase the complexity of such system abstractions and operational mechanisms, as we can no longer rely on the assumptions that a centralized data center allows. Moreover, there is a tension between the requirement of serverless computing to abstract the underlying infrastructure, and the requirement of many edge computing services to run on very specific devices due to data locality or device capabilities.

In this chapter, we present our approach to resolve this tension. Together with the middleware we presented in Chapter 5, our serverless edge computing platform represents the second fundamental component for a seamless distributed computing fabric for edge intelligence. We begin in Section 6.1 with an introduction to serverless edge computing and outline our approach. In Section 6.2, we highlight the challenges of serverless edge computing with a discussion of application scenarios, and data from preliminary experiments. In Section 6.3 we present the design and implementation of our platform. We extend existing serverless computing platforms to run on a computing substrate described in section 2.3.2. It provides a high-level programming model for developing data-intensive serverless functions. We introduce a new container scheduling system that makes heuristic tradeoffs tailored to these environments. To deal with the diversity of edge computing scenarios, in Section 6.4 we present a method to automatically fine-tune scheduler parameters to achieve high-level operational goals. Section 6.5 presents our experimental evaluation using the methodologies from Chapter 4. Finally, we present related work in Section 6.6, and conclude the chapter in Section 6.7.

6.1 Introduction

Serverless edge computing has emerged as a compelling model for dealing with many of the challenges associated with edge infrastructure [ATC+21, GND17, NRS+17, RHM+19, BMHP19, YHZ+17]. It expands on the idea of serverless computing, which first drew widespread attention when Amazon introduced in 2015 its Lambda service [JSSS+19]. It allowed users to develop their applications as composable *cloud functions*, deploy them through a Function-as-a-Service (FaaS) offering, and leave operational tasks such as provisioning or scaling to the provider. Serverless computing is also a candidate for enabling edge AI workflows discussed in Section 2.2. Analogous to the idea of serverless cloud functions, we imagine that *edge functions* can significantly simplify the development and operation of certain edge computing applications. In [Sch19] we presented a case study demonstrating the benefits of serverless edge computing for smart home analytics applications. Results showed that application responsiveness can be improved by up to 85%, and runtime execution costs can be reduced to almost 0.

Many serverless platforms have emerged that allow different computational models, yet their support for data-intensive distributed computing [HFG+18], and edge computing environments [ATC+21] is still limited. Specifically, this manifests as follows. First, they do not consider the proximity and bandwidth between nodes [HFG+18], which is particularly problematic for edge infrastructure where the distance between compute nodes, data storage, and a function code repository (e.g., a container registry), incur significant latencies and inter-network traffic [SBCD09]. Second, fetching and storing data is typically part of the function code and left to application developers (e.g., manually accessing storage service APIs), which makes it hard for the platform to reason about data locality and data movement tradeoffs [RHM+19, SLF+20]. Third, they provide limited or no support for specialized compute platforms or hardware accelerators such as GPUs [HFG+18, IMS18], leaving potential edge resources that provide such capabilities underutilized.

This chapter presents our efforts in addressing these three issues, which we believe are the most immediate limiting factors in enabling the distributed compute fabric idea we outlined in Section 2.3. We present a platform for serverless edge computing that introduces a high-level programming model, and Skippy, a container scheduling system that facilitates the efficient placement of serverless edge functions on distributed and heterogeneous infrastructure. Our system interfaces with existing serverless execution runtimes, like the container orchestration system Kubernetes, that were not designed for edge computing scenarios, and makes them sensitive to the characteristics of edge systems. The programming model introduces concepts that allow the platform to reason about the dataflow of functions. Skippy is an online scheduler, modeled after the Kubernetes scheduler, which implements a greedy multi-criteria decision making (MCDM) algorithm. We introduce two additional components that are commonly missing in state-of-the-art container schedulers: a static bandwidth graph of the network holding the theoretical (or estimated) bandwidth between nodes, and a data index that maps data item identifiers to the storage nodes that hold the data. These two extra components facilitate our additional scheduling constraints that favor nodes based on (1) proximity to data storage nodes, (2) proximity to the container registry, (3) available compute capabilities (e.g., for favoring nodes that have hardware accelerators), and (4) edge/cloud locality (e.g., to favor nodes at the edge). We have found that these constraints are a critical to make an effective tradeoff between data and computation movement in edge systems. Furthermore, we recognize that tuning scheduler parameters for effective function placement is challenging, as it requires extensive operational data and expert knowledge about the production system. Instead, we propose a method that leverages the tight integration of the scheduler with a simulation framework, in combination with existing multi-objective optimization algorithms, to optimize high-level operational goals such as function execution time, network usage, edge resource utilization, or cloud execution cost.

We implement a prototype that targets the container orchestration system Kubernetes, and deploy it on an edge testbed we have built. Complementary, we evaluate our system with trace-driven simulations in different infrastructure scenarios, using traces generated from running representative workloads on our testbed. We show how the scheduler and optimization technique work in tandem to enable serverless platforms to be used in a wide range of edge computing scenarios. Our results show that (a) our scheduler significantly improves the quality of task placement compared to the state-of-the-art scheduler of Kubernetes, and (b) our method for fine-tuning scheduling parameters helps significantly in meeting operational goals.

The specific contributions can be summarized as follows:

- a design for a serverless platform that treats concepts from edge AI workflows as first-class citizens, both in the programming model and in the runtime platform, and

- Skippy: a container scheduling system that enables existing serverless frameworks to support *edge functions*, and make better use of edge resources. Our scheduler introduces components and constraints that target the characteristics of edge systems.

- A method to tune the weights attached to low-level constraints used by the scheduler, by optimizing high-level operational goals defined by cluster operators. To compute the optimization we use the serverless system simulator presented in Section 4.3.

- We demonstrate Skippy's performance in various scenarios using data from our testbed and running trace-driven simulations. We use Ether, the system presented in Section 4.2, to synthesize edge topologies for different scenarios.

- Open data set of traces from extensive profiling of our edge testbed, and synthetic traces from our simulations of different infrastructure scenarios [RR20].

6.2 Background & application scenarios

This section outlines the domain for which we have developed our system. Based on the discussion in Chapter 2, we drill down into different application and infrastructure scenarios to motivate serverless edge computing and highlight systems challenges. Furthermore, we provide background on the operational underpinnings of serverless platforms using as examples Kubernetes and OpenFaaS.

6.2.1 Data-intensive serverless edge computing

Typical characteristics and requirements associated with data-intensive edge computing applications can be summarized as follows:

- heterogeneous workloads: the application is composed of multiple functions that have different computational requirements (e.g., GPU acceleration)
- locality sensitive: some parts of the application are locality sensitive, e.g., should not be executed in the cloud because consumers are located at the edge
- latency sensitive: some parts of the application are required to provide service quality at low latency
- high bandwidth requirements: some parts of the application may exchange large amounts of data

Research has shown that edge AI applications that deal with video or image data typically have all of these requirements [RD19, WFG$^+$19, CHW$^+$17, VGGK18]. Smart city scenarios are also an illustrative example. Data from sensor arrays and cameras distributed throughout the city can be used to create analytics models such as traffic risk assessment, crowd behavior, flooding or fire spreading models, or ecosystem/plant monitoring [RD19]. They could also serve as sensory input for cognitive assistance applications [RD19, WFG$^+$19], Moreover, edge resources can be used for mobile AI workload offloading. Serverless computing may be a good model for implementing such applications at scale on a distributed computing substrate we discussed in Chapter 3.

We implemented a prototypical edge computing application that has the characteristics and requirements described above. Specifically, we found the most generalizable application to be a machine learning (ML) workflow with multiple steps, where each step has different computing requirements, and needs to make efficient use of a diverse set of edge resources. Similar to the smart home data analytics pipeline we presented in [Sch19], we consider a typical ML pipeline with three steps [HMR$^+$19]: (1) **data preprocessing**, (2) **model training** (that can be accelerated by GPUs), and (3) **model serving**. In our concrete example we use the MNIST dataset to train an image classifier because of the dataset's availability allowing reproducibility. We implement each step as a *serverless function.*

Serverless functions

In the serverless function abstraction, the programming model concepts are *functions*, *events*, and *triggers*. The serverless platform executes functions in response to events. The events that lead to a function execution are defined by a trigger. Functions are simply event handlers in the classic event-driven programming sense. For example, in case of model training, the event would be triggered by the previous ML workflow step, i.e., the data preprocessing. An example of a serverless function written in Python that implements an ML training step is shown in Listing 6.1. In OpenFaaS, the platform packages function code and its dependencies into a container image, pushes it to a registry, from where the code is pulled by a compute node after scheduling.

```python
import boto3
import numpy
# ... import ML libraries such as tensorflow or mxnet

def handle(req):
  s3 = boto3.client('s3')
  with open(tmpfile, 'wb') as f:
    s3.download_fileobj('bucket', req['train_data'], f)

    data = numpy.load(f)
    model = train_model(data, req['train_params'])

    s3.upload_fileobj(serialize(model), 'bucket', request['
   serialized_model']'])
```

Listing 6.1: Example of a data-intensive serverless function.

This function implements ML model training, and involves: connecting to a storage server (S3 in this case), downloading the training data from the file object encoded in the request (which was previously generated by the data preprocessing step), converting the data into some appropriate format for running a training algorithm, and then uploading the serialized model. As every data-intensive function has a similar format, i.e., fetching, processing, and then storing data, we developed a higher-level abstraction for these functions, which we will present in Section 6.3.2. Specifically, we elevate fetching and storing data as platform features, which allows the platform to reason over the dataflow of the function, e.g., which specific data is pulled (encoded by a URN for example) to locate the closest data store that holds the data. We use this feature in the scheduler (see Section 6.3.5) for determining the tradeoff between data and computation movement.

Data and computation movement

Figure 6.1 shows a comparison between the size of container images for each function of our application, and the total amount of data each function has to transfer during its execution. It also shows a back-of-the-envelope calculation on how much time cloud or edge resources spends on transferring either container images or data for each function step. We consider a typical scenario, where an edge network has a 1 GBit/s internal bandwidth, 25 MBit/s uplink and 100 MBit/s downlink. Data is located at the edge, and the container registry is located in the cloud, which also has an internal bandwidth of 1 GBit/s. We can see that the difference in uplink and downlink bandwidth play a significant role in trading off data and computation movement.

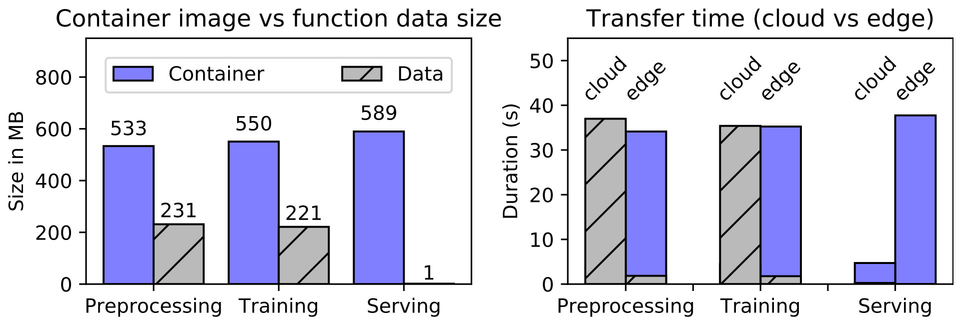


Figure 6.1: Comparison of container image sizes and total data transferred by functions. The right figure shows the time spent on either container image or data transfer in either cloud or edge networks.

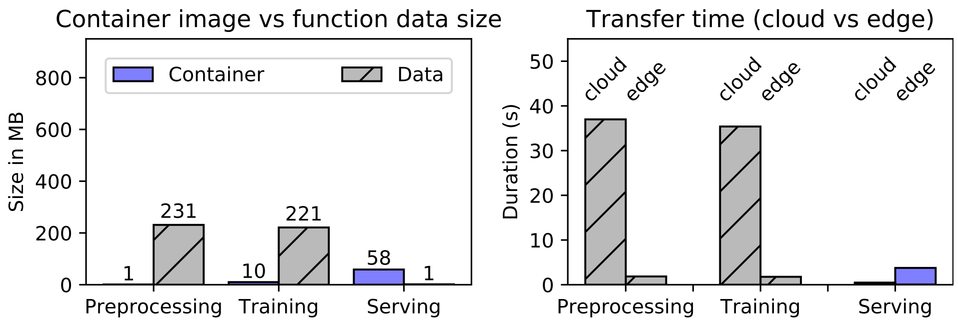


Figure 6.2: The same calculation as Figure 6.1 when subtracting shared layers between images and only considering unique image size.

The popular container platform Docker uses a layered file system, meaning that layers can be shared between container images. Because most containers build on similar base images, the unique image size when considering shared layers if often much smaller. For distributing container images this means that, if the base image has already been downloaded by some container, downloading a different container image that uses the same base image will also be much faster. When inspecting the images of our specific application, we found that almost 90% is shared across images. Figure 6.2 shows the same calculation as above, illustrating that, now, most of a function's latency comes from pulling data. These calculations show the importance of data and computation movement tradeoffs that serverless edge computing platforms have to make at runtime, in order to optimize function execution performance.

6.2.2 Edge computing infrastructure

Another important performance aspect is the type of hardware a particular function is executed on, and the impact on function execution time. In Section 3.1 we discussed the computing continuum, the various candidate computing architectures for edge infrastructure, and use cases for each one. Based on this discussion, we consider the following computers and architectures to test our platform: (1) **Small-scale data centers** that use VM-based infrastructure and are placed at the edge of the network, often termed cloudlets [SBCD09]. (2) **Small form-factor computers**, such as Intel's Next Unit of Computing (NUC) platform with built-in CPUs are used in, e.g., Cannonical's Ubuntu Orange Box [VN]. (3) **Single Board Computers (SBCs)** such as ARM-based Raspberry Pis used as IoT gateways or micro clusters [JBP+18]. (4) **Embedded AI hardware**, such as NVIDIA's Jetson TX2 that provide GPU acceleration and CUDA support [NVI].

We have profiled our ML workflow steps as OpenFaaS functions on these different devices. Table 6.1 lists the hardware specifications of the device instances we used. Figure 6.3 shows the results of 156 warm function executions. The Raspberry Pis were not able to run the model training step as they ran out of memory. The results show the impact of extreme device heterogeneity in an edge computing substrate. Also, we can see that the model training step benefits greatly from GPU acceleration, performing better on a Jetson TX2 compared to an Intel NUC despite the NUC's powerful i5 processor.

Device	Arch	CPU	RAM
VM	x86	4 x Core 2 @ 3 GHz	8 GB
SBC	arm32	4 x Cortex-A53 @ 1.4 GHz	1 GB
NUC	x86	4 x i5 @ 2.2 GHz	16 GB
TX2	aarch64	2 x Cortex-A57 @ 2 GHz	8 GB
		256-core Pascal GPU	

Table 6.1: Device type specifications.

6.2.3 Cluster infrastructure scenarios

In Section 3.3 we presented several scenarios where edge computing resources are federated to form a distributed computing substrate. To demonstrate that our system can deal with these scenarios, we use the *Edge Topology Synthesizer* framework presented in Section 4.2 to synthesize plausible cluster configurations. We briefly summarize the specific scenarios and respective cluster configurations that we consider. Table 6.2 lists the cluster hardware composition.

S1: Urban Sensing In the urban sensing scenario, cities deploy sensor arrays and cameras to enable smart city applications that require access to real-time data of urban environments [CBSG17]. For this scenario, we assume a total of 200 sensor nodes,

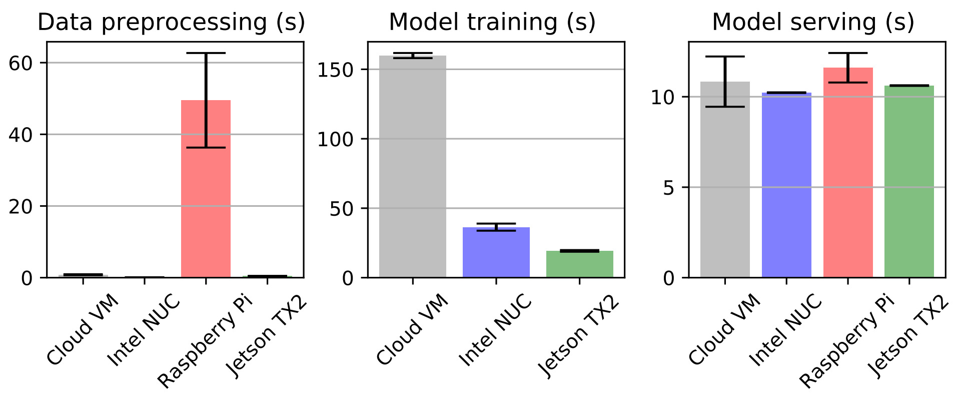


Figure 6.3: Average execution time and standard deviation of ML workflow functions in seconds on different device types illustrating both workload and device heterogeneity.

where each node is equipped with two SBCs (for data processing and communication). Furthermore, we assume that throughout the city, there are installations of cloudlets that comprise an Intel NUC, and two embedded GPU devices per sensor node camera for video processing tasks. To meet peak demands, 30 VMs hosted at a regional cloud provider are added as fallback resources into to the cluster. In terms of network topology, we assume that each municipal district forms an edge network. Each edge network has an internal LAN bandwidth of 1GBit/s and connected with 100/25 MBit/s down/uplink to the internet. Cloud nodes have an internal bandwidth of 10GBit/s, and a direct 1GBit/s uplink to the internet. These data are plausible extensions of the urban sensing scenario presented in Section 3.3, specifically the Array of Things (AoT) [CBSG17] project that operates a deployment of about 200 sensor nodes in Chicago. Each AoT node is connected via a mobile LTE network to the Internet.

S2: Industry 4.0 For this scenario, we assume an Industrial IoT (IIoT) [CWC+18] use cases, where several factories at different locations are equipped with edge computing hardware, and each location has an on-premises but provider-managed cloud (e.g., a managed Microsoft Azure deployment, where on-premises cloud resource use is billed). We assume ten factory locations, each having 4 SBCs as IoT gateways, 1 Intel NUC, 1 Jetson TX2 board, and 4 VMs on the on-premises cloud. The numbers are plausible extensions to the prototypes presented in [CWC+18], and the general trend towards using embedded AI hardware for analyzing real-time sensor and video data in IIoT scenarios [YMLL17]. Each edge and on-premises cloud has a data store. The SBCs are connected via 300 MBit/s Wi-Fi link to an AP that has a 10 GBit/s link to the edge resources, and a 1 GBit/s link to the on-premises cloud. Premises are connected via a business ISP with 500/250 MBit/s down/uplink.

S3: Cloud federation To compare our system in non-edge computing scenarios, we also consider a cloud computing configuration where there are no edge devices and less heterogeneity than in edge scenarios [VPK+15]. We model a cloud federation scenario across three cloud regions, where each region has, on average, 150 VMs. All regions contain several nodes with data stores. Region 1 has slightly more VMs and more storage nodes than the others. The bandwidth is 10 GBit/s within a region, and 1 GBit/s cross-region, as reported in Section 3.3.

Scenario	Category	nodes	% of compute device types			
			VMs	SBC	NUC	TX2
Our Testbed		7	14	57	14	14
S1: Urban Sensing	edge	1 170	3	39	19	39
S2: Industry 4.0	hybrid	110	40	40	10	10
S3: Cloud regions	cloud	450	100	0	0	0

Table 6.2: Cluster configurations of different scenarios.

6.2.4 Technical background: Kubernetes & OpenFaaS

Our system is designed to extend existing platforms that enable serverless computing and FaaS deployments, such as Kubernetes and OpenFaaS. Because our prototype was developed for these two systems, we present technical background on the interplay between the two. The core mechanisms, however, are found in similar systems.

Kubernetes & container scheduling

Kubernetes is a container orchestration system used for automating application deployment and management in distributed environments. It is a popular runtime for serverless computing, micro-service-based application deployments, and, increasingly, FaaS platforms [MSV18]. Using Kubernetes, FaaS platforms can package function code into lightweight container images, and can then make use of all the features of the Kubernetes platform, such as scheduling, autoscaling, or request routing. The Kubernetes scheduler is one of the critical components of the platform. The task of the scheduler is to assign a *pod* (the atomic unit of deployment in Kubernetes) to a cluster node. A pod can consist of one or more containers, shared storage volumes, network configuration, and metadata through which the pod can, for example, communicate its resource requirements. The Kubernetes scheduler is an online scheduler, meaning it receives pods over time and generally has no knowledge of future arrivals. It is therefore different from many SPP solutions that schedule several services at once [SNS+17]. Similar to many resource schedulers of real-world systems, it employs a greedy MCDM procedure, which we formalize in Section 6.4. Hard and soft constraints are implemented as follows. First, the available cluster nodes are filtered by *predicate functions*. These functions evaluate to true or false, and represent hard constraints that eliminate nodes incapable of hosting a pod, e.g., because they are unable to provide the resources requested by a pod. Second, the remaining set of feasible nodes are ranked by *priority functions* that represent soft constraints to favor nodes based on their suitability for hosting the pod. Calculating the score for a pod–node tuple involves invoking each active priority function, normalizing the values of each function to a range between 0 and 10, and building a weighted sum. The highest scoring node is then selected for placement. Kubernetes provides a variety of predicate and priority functions that can be configured in the scheduler. However, as we describe in more detail in Section 6.3.5, the default priority functions do not perform well in edge scenarios. In particular, they do not consider the critical tradeoff between data and computation movement which we have highlighted earlier. In the remainder of this paper, we use the terminology of Kubernetes (i.e., pod, node, priority function), to refer more generally to a unit of deployment, cluster resource, and soft constraint, respectively.

OpenFaaS

OpenFaaS is a serverless framework that uses Kubernetes as both execution runtime and deployment platform. Function code is packaged into Docker containers, and a small watchdog is added to the container that executes the function based on HTTP calls triggered by events or through invocations of the OpenFaaS API gateway. OpenFaaS can run on top of Kubernetes using the runtime driver *faas-netes*, which deploys functions as Kubernetes pods, and then delegates scheduling decisions to the Kubernetes scheduler. With OpenFaaS, the Kubernetes scheduler is triggered in two situations: an initial manual deployment of new functions, or automated function replica creation through autoscaling. If an OpenFaaS user runs the `faas-cli deploy` command to deploy function code, the code is packaged into a Kubernetes pod, and the pod is placed in the scheduling queue. Subsequent requests to the function triggered by events or HTTP endpoint calls are forwarded to Kubernetes, which takes care of load balancing requests among running replicas. This is the case if the function's autoscaling policy is set to at least one replica, and is useful to avoid cold starts for functions that are invoked frequently. A cold start refers to a function invocations where the container image has to be downloaded and the container is started for the first time, which incurs significant latency. For short-lived functions that are not invoked frequently, and would otherwise block a node's resources despite being idle, OpenFaaS allows a scale-to-zero policy, which removes such idle functions from nodes after a short time. This is useful for functions such as the data pre-processing or training step in our ML pipeline. In this case, a request to a function immediately triggers a replica creation and therefore scheduling.

6.3 Design and implementation

This section explains how our system works in tandem with existing serverless systems to enable serverless edge computing. We first discuss overall design goals, then present our high-level programming model, followed by a detailed explanation of the individual system components.

6.3.1 Design goals

In Section 2.3.3 we described the design goals for edge intelligence systems in general. Derived from these higher-level goals, and the presented scenarios, we discuss the design goals for a serverless edge computing platform, with a particular focus on edge AI applications.

Hiding operational complexity: Orchestrating application execution on edge resources is more involved than in cloud-based platforms as there are no well-defined APIs and, e.g., homogeneous dedicated learning clusters to submit training jobs to. Mapping complex workloads, such as AI workflows, to a multi-purpose edge-compute resources needs to make full use of hardware capabilities to execute efficiently (see Figure 2.6). A programming model and APIs for edge AI applications needs to hide this operational complexity from the developer. In particular, programmers should not have to worry about the distribution of data, and the heterogeneous capabilities of edge resources. The *stickiness* of tasks and resources should be easily configurable, and a scheduler needs to infer scheduling constraints and goals from application and device contexts.

Context-awareness: Being aware of the context of an edge device is essential for enabling transparent infrastructure for edge AI applications. From a service provider's perspective that manages fleets of edge devices and provides models across several domains, context-awareness can help manage operational complexity, by deploying models on demand to devices based on their context. Context-aware policies can also be used to specify retraining triggers, such as thresholds of new training data, or device battery and charging conditions.

Artifacts as first-class citizens: Locally trained models and data available on the edge exacerbates the issues around model and data management. To facilitate this abstraction, our edge AI serverless platform treats models and data as first class citizens. This allows the platform to make decisions on where data can reside or be transferred, based on application contexts and device constraints, and enables the scheduler to satisfy data–function locality objectives.

Fine-grained policy control: Developers should be able to express the *context* in which functions are allowed to execute or data is transferred. However, this requires that the programming model and API are intuitive for developers, but expressive enough to guide the execution platform in its decision on how to schedule functions or replicate data.

6.3.2 Programming model

We introduce AI workflow specific extensions to functions, by elevating concepts of the AI workflow to first-class citizens in the model to provide common abstractions that make developing edge AI workflow functions easier. Specifically, we add (a) the notion of *artifacts*: data artifacts such as training data sets, and model artifacts, i.e., machine learning models, (b) *selectors*: that allows fuzzy querying of artifact metadata to allows dynamic injection of data artifacts into functions, (c) *policies*: fine-grained control mechanisms for function scheduling. For workflow *composition*, we rely on systems like ModelOps [HMR+19] or OpenWhisk composer [opea], which offer different ways to compose AI workflows. Our examples are based on Python and the capabilities of the MXNet ML library [mxn], but the concepts could be applied to most languages.

Data & models: Current serverless platforms generally only allow JSON documents to be passed between functions. When dealing with large artifacts or streaming data, which is common in AI workflows, developers are required to manually read and write from cloud storage services. In particular, it is common for functions to consume or produce data or model artifacts. Our approach provides ways to annotate such functions, and provides a data API that hides platform data management tasks from the developer. Additional metadata from the annotations allow the platform to transparently handle data transfer and storage and respect data locality policies.

Model selectors: We allow the injection of models into functions based on *selectors*. This allows developers to define the requirements of a functions without specifying the exact model instance to use. For example, as shown in Listing 6.2 a selector can specify the type of model (regressor, or classifier), the type of data it was trained on, the type of data it processes, or model performance metrics such as the accuracy or robustness scores [WZC+18]. The model metadata collected by the AI lifecycle engine ModelOps [HMR+19] facilitates this.

```
@consumes.model(selector={'type': 'image_classifier',
    'data_tags': ['machine_x'], 'accuracy': '>=0.88'}})
def inference(model, request):
  data = # ... data prep tasks
  return model.estimate(data)
```

Listing 6.2: Artifact injection via model selector

Policies: Policies allow developers to define additional function execution conditions. The *deadline* policy is a function execution service-level objective (SLO), which is useful for, e.g., inferencing functions that have low-latency requirements. The runtime can also factor in latency incurred by data transfer. A *fn* policy defines properties that a node should or must have for a function to be scheduled to that node. The *data* policy is similar to a role, and defines constraints on data transfer. The *strict* keyword makes the policy a hard constraint. For example, the function may only be executed when the data is accessible from within the given network.

```
@policy.deadline('2s')
@policy.fn(node = 'user_device', capability = 'gpu')
@policy.data(network=['company_network'], strict=True)
```

Listing 6.3: Different policy annotation

Example functions

We demonstrate how this high-level API can be used to simplify the development of functions with two examples.

Listing 6.4 shows the example function from Listing 6.1 re-written with our API.

```
from skippy.data import DataArtifact, ModelArtifact, consumes,
    produces, policy
# ... import ML libraries such as tensorflow or mxnet

# can have multiple data annotations
@consumes.data(urn = 'my_bucket:train_data')
@produces.model(urn = 'my_bucket:model')
@policy.fn(capability = 'gpu')
def handle(req, data: DataArtifact) -> ModelArtifact:
    arr = data.to_ndarray()
    model = train_model(arr, req['train_params'])
    return model
```

Listing 6.4: Example of training function with metadata annotations.

A common scenario in the use cases we described are user or device-specific models refined from a base model using device-local data. Listing 6.5 shows how such a function can be defined using our programming model. To satisfy the constraints, the scheduler runs this function only a user's device (given by the `usr` variable), and when data is available within the specified network.

```
@consumes.model(selector={'urn': 'model:base'})
@consumes.data(batch = 100, selector=...)
@produces.model(type='regressor', urn='model:user:{usr}')
@policy.fn(node = 'local')
@policy.data(network = 'local', strict=True)
def refine(model: ModelArtifact, data: DataArtifact):
  ndarr = data.to_ndarray() # data artifact API
  # transfer learning code
  return refined_model
```

Listing 6.5: Example function with several constraints

6.3.3 System overview

We briefly outline the main components of Skippy, and the integration with Kubernetes and OpenFaaS. Figure 6.4 shows the specific integration with Kubernetes.

- **metadata schema:** Skippy makes heavy use of container and node labels, which are supported by many container platforms, to communicate information about functions and compute nodes to the scheduler. Skippy uses various mechanisms

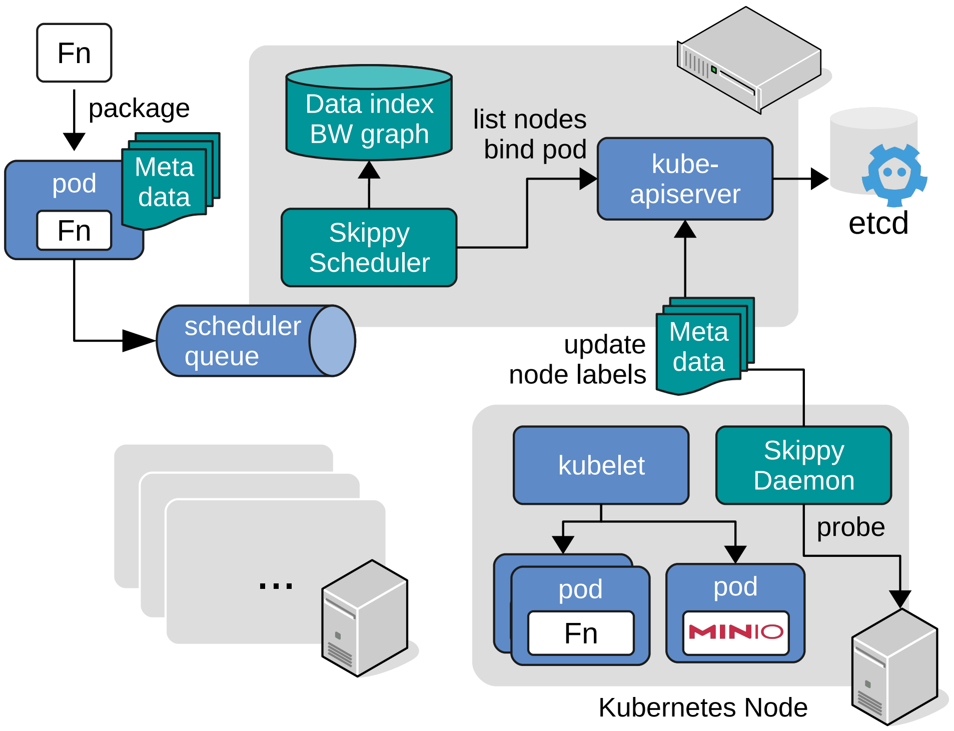


Figure 6.4: Overview of Skippy's components and their interaction in a deployment with Kubernetes.

to automate labeling, such as the skippy-daemon or annotation parsing which we describe in more detail later. All metadata labels of Skippy have the prefix `*.skippy.io`.as described in [RHM+19]

- **skippy daemon:** a daemon that runs on all cluster nodes alongside the primary node agent (e.g, kubelet in the case of Kubernetes). It scans a node's hardware capabilities, such as the availability of a GPU, and then labels the node with the corresponding metadata. It can do this periodically to react to pluggable capabilities, such as USB attached accelerators.

- **skippy scheduler:** the Skippy scheduler is an online scheduler that is sensitive to the characteristics of edge systems. It requires access to the cluster state and a programmatic way of binding containers to nodes. In the case of Kubernetes, the kube-apiserver provides these features via REST APIs.

- **data index:** Functions can access data via storage pods that host MinIO instances (an S3 compatible storage server), distributed across the cluster. Skippy currently does not automatically manage these storage nodes. Replication and data distribution is left to other system components. However, Skippy dynamically discovers storage nodes, keeps an index of the file tree, and is sensitive to the proximity between compute and storage nodes by using the bandwidth graph.

- **bandwidth graph:** We extended the ideas from Chapter 5, where we created a graph of the network based on the latency between nodes. The bandwidth graph is an in-memory representation of the network topology and the available bandwidth between nodes. Currently, we use static information about the network, and information we collect through the skippy daemon (e.g., available bandwidth of network devices). We are working on a system that updates the bandwidth graph at runtime by monitoring the network utilization of nodes [Mar21].

6.3.4 Node and function metadata collection

Resource schedulers depend on certain information about the cluster state and the workload requirements to make good placement decisions. Skippy makes heavy use of metadata about functions and compute nodes in the cluster to communicate this information to the scheduler. A goal of or system is to reduce the amount of manual configuration a human has to do, and rather automate the process. We use the Skippy daemon to collect node metadata, and our high-level programming API to collect function metadata. The node metadata are stored and accessed via the cluster orchestration system. In the case of Kubernetes, this is stored in etcd, a distributed key-value store. If the orchestrator does not provide a storage system, Skippy can also store the metadata in memory.

Node metadata: Skippy daemon

The Skippy daemon is deployed as a container on all cluster nodes. It automatically probes a node's capabilities and maintains its Skippy-specific labels. Currently, the daemon probes if a node provides an NVIDIA GPU, the availability and version of a CUDA installation, and if the node is running a MinIO storage pod. The daemon code allows straight-forward addition of custom capability probes. When nodes are added to the cluster at runtime, the Skippy daemon labels the node with an appropriate locality label (edge/cloud). The system overhead of running the daemon is minimal given a fairly simple Python implementation. It requires roughly 120 MB of disk space and 25–40 MB of RAM depending on the CPU architecture, making it feasible even for resource constrained devices.

Function metadata: annotation parsing

Serverless frameworks typically provide their own packaging and deployment tooling. In OpenFaaS, the `faas-cli` command builds the container image from the function code, and the manually configured metadata. We provide a wrapper for these tools that parses the function annotations and translate them into labels for the scheduler. Listing 6.6 shows the resulting labels of analyzing the function from Listing 6.4. These are then used as input for the priority functions described in Section 6.3.5.

```
{
  'data.skippy.io/recv': ['my_bucket:train_data'],
  'data.skippy.io/send': ['my_bucket:model'],
  'capability.skippy.io/gpu': ''
}
```

Listing 6.6: Example function labels resulting from metadata parsing.

6.3.5 Skippy scheduler

The Skippy scheduler enables serverless platforms to schedule *edge functions* more efficiently. Its design is based on the Kubernetes scheduler, which is a queue-based monolithic scheduler that implements a greedy MCDM heuristic (see Section 6.2.4).

Edge-friendly priority functions

We introduce four priority functions that target requirements of edge computing applications and characteristics of edge systems, which complement common cloud-centric scheduling constraints found in, e.g., Kubernetes [Kub19]. The additional functions are motivated by the following observations: First, in many data-intensive edge computing applications, data is stored at edge locations. Yet, container clusters typically rely on centralized cloud-based repositories such as Dockerhub for managing container images. When scheduling pods that operate on data, there is therefore an inherent tradeoff between sending computation to the edge or sending data to the cloud, as we have highlighted in Section 6.2 and Figure 6.1. The two priority functions *LatencyAwareImage-LocalityPriority* and *DataLocalityPriority* help the scheduler make this tradeoff at runtime. Second, the increasing diversity of specialized compute platforms for edge computing hardware provide new opportunities for accelerating the equally diverse workloads. The *CapabilityPriority* matches tasks and nodes based on their requirements and advertised capabilities, respectively. Third, it is often the case that functions should prioritize execution at the edge for a variety of reasons. The *LocalityTypePriority* enables the system to respect these placement preferences.

We explain each function in more detail and provide algorithmic descriptions. Note that the Kubernetes scheduler expected normalized values from priority functions, which are the result of mapping the range of scores to an integer range [0..10]. We omit the code for this step.

LatencyAwareImageLocalityPriority Favors nodes where the necessary container image can be deployed more quickly. We use knowledge about the network topology to estimate how long it will take in an ideal case to download the image. Algorithm 6.1 shows pseudocode for the function. Because the bandwidth graph is static and does not consider actual available bandwidth during runtime, the calculation is only an approximation. Making a plausible estimate of actual network download speed would be too complicated for a priority function, which has minimal runtime knowledge and needs to execute quickly. However, together with the implementation of the DataLocalityPriority, the function allows us to make a heuristic tradeoff between fetching the container image, or fetching data from a data store.

Algorithm 6.1: Scheduler priority function: LatencyAwareImageLocalityPriority

```
   Result: Estimation of how long it will take to download a pod's images
1  Function score:
       Input : pod
       Input : node
       Input : bandwidthGraph
2      size ← 0;
3      for container in pod's list of containers do
4          if container's image is not present on node then
5              size $\overset{+}{\leftarrow}$ size of the container's image;
6          end
7      end
8      bandwidth ← bandwidthGraph[registry][node];
9      time ← $\frac{\text{size}}{bandwidth}$;
10     return time;
```

DataLocalityPriority Estimates how much data the function will transfer, and favors nodes where the data transfer happens more quickly. From the labels extracted by our function annotation parser (see Section 6.3.4), we know the data items a function operates on. We query the data index to get all storage nodes that hold the specific data item. The data item size can be queried through the MinIO S3 API, which the data index keeps in a local cache. We then make the same network transfer time estimations as in LatencyAwareImageLocalityPriority using our bandwidth graph. Algorithm 6.2 shows pseudocode for the function.

CapabilityPriority Checks the computing platform requirements of a function, and favors nodes that have those capabilities (e.g., a GPU for an ML training function). The implementation uses node and function metadata gathered by the Skippy daemon and function annotation parsing. Algorithm 6.3 shows pseudocode for the function.

LocalityTypePriority Favors nodes in a certain locality, e.g., nodes located in edge networks or in the cloud. Through the programming model we have described, developers can specify a high-level placement prioritization, e.g., for preferring nodes in a certain network context. It checks the presence of the same values of *locality.skippy.io/type* in pod and node labels. We omit the code for this function as it can be implemented by a

Algorithm 6.2: Scheduler priority function: DataLocalityPriority

```
Result: Estimate how long it takes for a node to transfer the required runtime
        data
1  Function score:
       Input : pod
       Input : node
       Input : storageIndex, bandwidthGraph
2      time ← 0;
3      for urn in values of 'data.skippy.io/recv' do
4          storages ← storageIndex[urn];
5          bandwidth ← $\min_{s \in storages}$(bandwidthGraph[s][node]);
6          size ← of data item urn;
7          time $\xleftarrow{+}$ $\frac{\text{size}}{bandwidth}$;
8      end
9      for urn in values of 'data.skippy.io/send' do
10         ... analogous to 'recv'
11     end
12     return time;
```

Algorithm 6.3: Scheduler priority function: CapabilityPriority

```
Result: Scores how many of a pod's requested capabilities are provided by a
        node
1  Function score:
       Input : pod
       Input : node
2      nodeCapabilities ← get all of node's labels starting with capability.skippy.io;
3      podCapabilities ← get all of pod's labels starting with capability.skippy.io;
4      score ← 0;
5      for podCapability in podCapabilities do
6          if nodeCapabilities contains podCapability $\wedge$ values are equal then
7              score $\xleftarrow{+}$ 1
8          end
9      end
10     return score;
```

simple label check. This function makes an assumptions about the existence of a static system hierarchy, which the main premise of fog computing [GLME$^+$15, ASDL19], where infrastructure is separated into distinct cloud and edge regions. However, as we will show in Section 6.5, our scheduling parameter optimization automatically ignores this function in scenarios where such a hierarchy does not emerge, for example in the cloud federation scenario.

6.3.6 Integration with OpenFaaS

To enable the deployment of applications as serverless functions, our prototype makes use of OpenFaaS. It provides a framework for defining function deployments, an API gateway through which all function requests are routed, and several runtime components to manage monitoring, alerting, and autoscaling. We modified faas-netes (see Section 6.2) to label pods resulting from OpenFaaS function deployments, to indicate that these pods should be scheduled by Skippy instead of the default Kubernetes scheduler. Otherwise Skippy integrates with OpenFaaS only via Kubernetes, in that Skippy schedules the pods created by faas-netes.

6.4 Automatic tuning of scheduling parameters

Some operational goals, such as minimizing overall function execution time or uplink usage, depend on too many (and possibly at runtime unknowable) factors that they could be calculated efficiently in priority functions. Schedulers that employ MCDM, such as the Kubernetes scheduler, often allow users to assign weights to each constraint to tune the scheduler towards certain behavior. This fine-tuning to meet specific operational goals can be difficult. Many factors need to be considered, such as the cluster topology, node heterogeneity, or workload characteristics. This leaves operators to either rely on their intuition, or use trial-and-error in production systems, to find weights that achieve the desired behavior.

We propose an approach to automatically find weights of priority functions that result in good placements that meet certain high-level operational goals. We consider a placement to be good if it: (1) leads to **low function execution time** during runtime, (2) **uses edge resource** efficiently, (3) **reduces traffic** leaving or entering a network, and (4) **reduces costs** associated with executing functions in the cloud. To that end, we use multi-objective optimization techniques, and use the simulator we have developed (see Section 4.3) to evaluate the goodness of optimization solutions. We first formalize some key aspects.

6.4.1 Problem formulation

Let $\mathcal{S}$ be the set of priority functions $S \in \mathcal{S} : P \times N \to \mathbb{R}$ where P is the domain of pods and N is the domain of nodes (and all metadata attached to them). The function $\mathit{schedule} : P \to N$ maps a pod p to a node n by evaluating the scoring function score for each node, and selecting the highest scoring node. The scoring function is essentially a weighted sum model over all priority functions and feasible nodes. Formally, this can be expressed as

$$\mathit{schedule}(p) = \operatorname*{arg\,max}_{n \in N} \mathit{score}(p, n) : \sum_{i=0}^{|S|} w_i \cdot S_i(p, n). \tag{6.1}$$

The default Kubernetes scheduler sets every $w_i = 1$. Our goal is to find values for $\mathbf{w} = (w_1\ w_2\ \cdots\ w_{|\mathcal{S}|})$ that optimize towards the previously defined objectives.

We have explained the technical details of the simulator in Section 4.3, but formally a simulation run $sim(T, W, \mathbf{w})$ takes as input (1) the cluster topology T, (2) a workload profile W, (3) the vector of priority function weights $\mathbf{w}$, and simulates the function execution based on the profiling data we have gathered. The cluster topology T is formally a graph $T = (V, E)$, $V = N \cup L$, where N is the set of cluster nodes, L is the set of links that have an associated bandwidth (e.g., several nodes can be connected to a common Wi-Fi link), and E are weighted edges that indicate the latency between nodes and links. A workload profile W assigns each function (in our case, the ML workflow functions), an inter-arrival distribution, from which we sample at simulation time to generate workload. Our four goal functions $f_i(sim(T, W, \mathbf{w}))$ are calculated from the simulation traces as follows:

f_1: average function execution time over all functions

f_2: up/downlink usage, i.e., the number of bytes transferred between networks through up/downlinks

f_3: edge resource utilization, i.e., the percentage of allocated resources on edge devices compared to cloud servers

f_4: cloud usage costs given a pricing model that includes traffic and execution time in the cloud

We now want to find $\mathbf{w}$ s.t. f_1, f_2, f_4 are minimized, and f_3 is maximized.

6.4.2 Implementation

We implement the optimization using our simulator and the Python Platypus [Had17] framework for multi-objective optimization. Platypus implements the well-known NSGA-II genetic algorithm [DPAM02], which has been found to be one of the best performing algorithms in the framework [BT19].

To find an optimized value of $\mathbf{w}$, we execute the Platypus framework's NSGA-II implementation with 10 000 generations. Each generation executes a single simulation run $sim(T, W, \mathbf{w})$ with a predefined W and T, and the current evolution of $\mathbf{w}$. A run creates function deployments according to W until the cluster is fully utilized, using our scheduler for placement decisions. We store execution traces into Pandas data frames, and then calculate the goal functions f_i from the traces. The result is a set of 100 solutions that are at the Pareto frontier of the solution space. As input for the scheduler, we select from that set a single solution $\mathbf{w}$ that is balanced across all goals.

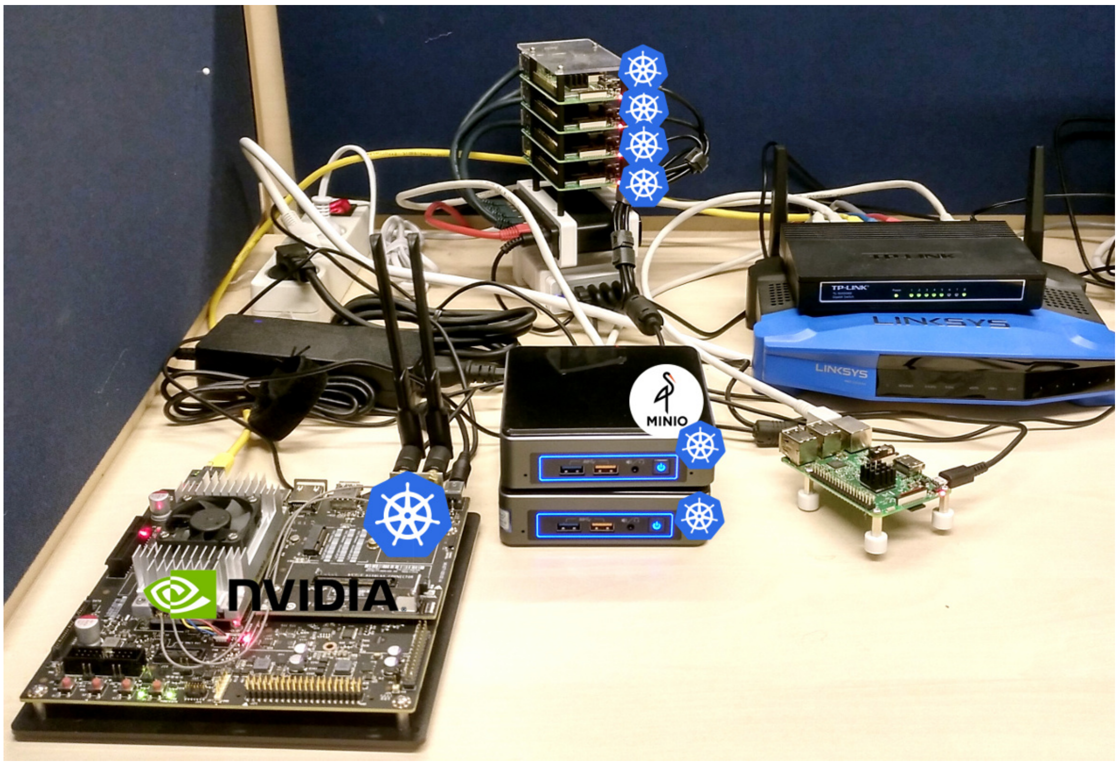


Figure 6.5: Our edge cloud testbed comprising a Raspberry Pi cluster, an NVIDIA Jetson TX2 board, and two Intel NUCs, one acting as edge storage node by hosting a MinIO pod. A VM hosted on our on-premises cloudlet is also part of the cluster.

6.5 Evaluation

This section presents our experiment setup, results, and a discussion of limitations. We first present the testbed we have built that we used to test our prototype implementation, and generate traces for the simulator. The scenarios we have defined in Section 6.2, and the traces generated from our testbed, are then used as input for our serverless simulator. We investigate how the scheduling decisions and parameter optimization impact application and system performance. We discuss the scheduler's performance in terms of scheduling throughput and latency, and, finally, discuss the current limitation of our system.

6.5.1 Edge system testbed & profiling

The testbed we have built comprises several edge computing devices listed in Section 6.2.2. Figure 6.5 shows the current setup. The nodes marked with a Kubernetes logo are part of the Kubernetes cluster used as runtime for OpenFaaS. The OpenFaaS gateway and Kubernetes master are hosted on a VM in our on-premises cloudlet.

For profiling, we follow the methodology described in Section 4.1. We run the application we have described in Section 6.2 on the testbed using our system prototype. That is, we implement each task as an OpenFaaS function, and execute each task on each device in both cold and warm state using different bandwidth and latency configurations. The functions are implemented in Python and use the Apache MXNet machine learning framework. We measure various system and application metrics, such as the system resource utilization, task execution time, the bandwidth requirements, e.g., when the container image that holds the function has to be downloaded, as well as the traffic produced by function invocations.

6.5.2 Experiment setup

We perform the experiments with parameters drawn from the infrastructure scenarios described in Section 6.2.3, and compare three scheduler implementations: the default Kubernetes scheduler as baseline, Skippy with weights set to 1, and Skippy with optimized weights. The simulator directly calls the Skippy scheduler code for scheduling functions, with the only difference that it does not call the Kubernetes API for requesting the cluster state and performing node bindings. The experiment process is as follows. We generate random application deployments, in our case ML workflow pipelines that comprise three ML functions, inject them into the scheduler queue, generate random requests given some workload profile, and then run the simulation for a certain number of invocations. Specifically, we deploy a new pipeline every few minutes and start generating requests to those functions a few seconds afterwards. After a specified number of function instances have been deployed, we generate another several thousand requests, until a request limit for that scenario has been reached. For example, in Scenario 2, each experiment ends after 30 000 invocations. Having the same amount of deployments and function invocations allows for a fair comparison of overall network traffic.

For simulating code movement we make an assumption based on observations of our images we described in Section 6.2: around 90% of an image's size comes from layers that are shared with other images. Meaning that, if any one of the images has already been pulled before, only 10% of another image's unique data has to be pulled. For our evaluation this means that we are not biasing the simulation towards the estimation that the Skippy scheduler makes through the LatencyAwareImageLocalityPriority. Our simulator also implements the basic autoscaling behavior of OpenFaaS. In particular, it includes OpenFaaS' *faas-idler* component that enacts the scale-to-zero policy: when a function is idle for 5 minutes or more, the respective function replica is stopped and the underlying Kubernetes pod removed. A subsequent call to the function incurs a cold start.

For synthesizing pipelines and requests we make the following assumptions. Each pipeline has three steps, where each step as an individual container image. However, as we have discussed in Section 6.2, we consider the commonalities across images. We synthesize both pipeline instances (i.e., functions deployed in Kubernetes pods) as well as container images, and assume a Pareto distribution of images. That is, not every function has

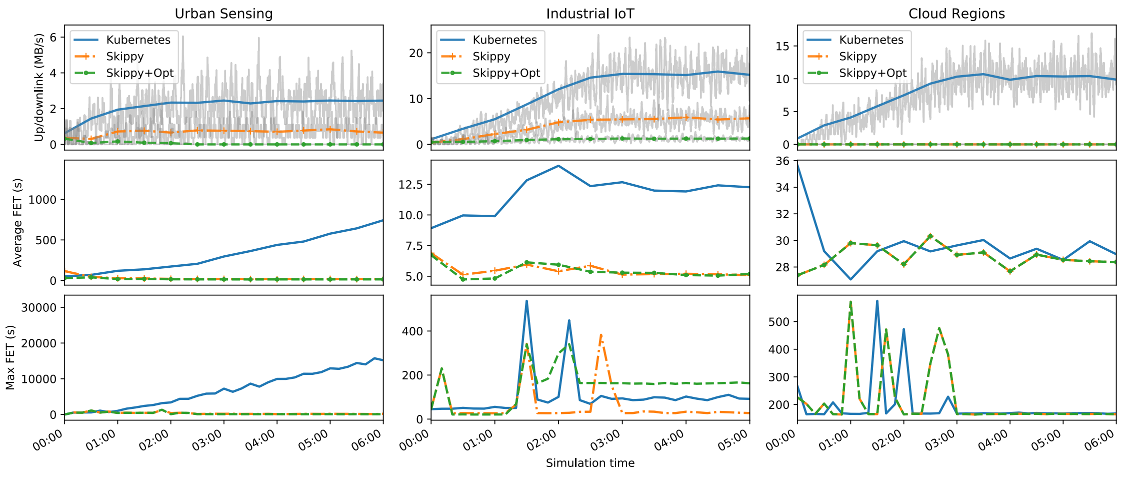


Figure 6.6: Drill-down into time series data from simulated experiment runs. The first row shows the average data rate of traffic going over up/downlinks. The second row shows the average function execution time (FET) over time (10 min rolling window). The third row shows the maximum function execution time (FET) over time (10 min rolling window).

a unique image. Instead we assume a Pareto-distributed relation of container images to pod instances, i.e., 80 percent of pods use the same 20 percent of images. For the workload profile W, we assume a typical [Fel00] Poisson arrival process where inter-arrivals are described by an exponential distribution. We set distribution parameters s.t. model serving requests of an individual pipeline are triggered at 40 requests per second, and data-preprocessing requests happen every few minutes allowing the faas-idler to occasionally shut down a replica. For synthesizing data items (e.g., training data as input for training functions), we assume that data items are distributed uniformly across data stores and workflows.

Experiment runs that compare different scenarios and schedulers use the same random seed for distribution sampling to guarantee comparability between scenario runs.

6.5.3 Experiment results

This section presents the results of our experiments. The results show (1) how function placement affects system performance, (2) how function placement affects system scalability, and (3) which priority functions have the highest impact on optimization goals.

Runtime performance of placements

Figure 6.6 shows key performance indicators from simulation runs in each scenario for the schedulers: the default Kubernetes scheduler, the Skippy scheduler, and Skippy using optimized priority function weights. The first row shows the average data rate going

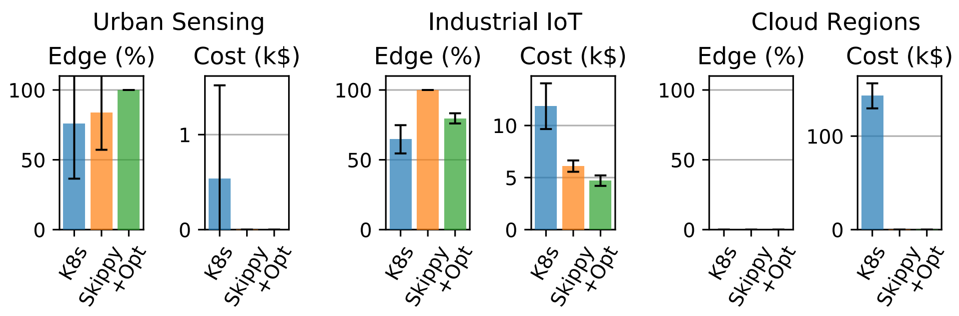


Figure 6.7: Edge resources utilization and execution cost of placements in three scenarios. Bars show the average across ten runs, error bars show one σ.

over up and downlinks. Ideally, a placement keeps traffic within networks, resulting in a low up/downlink usage. As the deployments are injected in the first phase of the simulation, the data rate grows, but is overall significantly higher with the Kubernetes scheduler. The Cloud Regions scenario (S3) highlights the problem when there are many nodes within a network, and few up/downlinks between them. The second and third row show the function execution duration over time. In the Urban Sensing scenario (S1), the Kubernetes scheduler's placements run into queuing issues early on. Function time keeps increasing because the network cannot keep up transferring data necessary by the function executions. In the Industrial IoT scenario (S2), while there are no queuing issues, the Kubernetes scheduler's placements lead to overall higher function execution times. There is no significant difference in the Cloud Regions scenario, because the devices within the cluster (cloud VMs) are fairly homogeneous in terms of task execution performance. Overall, the second row shows the interplay between using resources effectively, and trading-off data movement costs.

Figure 6.7 shows the aggregated results from several runs with different random seeds for the other two performance goals we have defined: edge resource utilization and execution cost. For calculating the cost we use the pricing model of AWS Lambda [Ama20]. For S1 we observe the effect of a low amount of cloud resources: almost no cloud execution cost and generally high edge resource utilization. Skippy and the optimization perform slightly better. In S2, where data is also placed in on-premises managed cloud instances, we observe that optimized Skippy can make a useful tradeoff between cost and edge utilization by preferring cloud resource in favor of moving data. S3 has no edge resources, but we can see that the Kubernetes scheduler's decision to place functions across regions leads to a high cost incurred by data movement. It also illustrates that most of the costs in our scenario comes from data movement (specifically data egress), rather than compute time, corroborating the results of a study about the unintuitive nature of serverless pricing models [Eiv17].

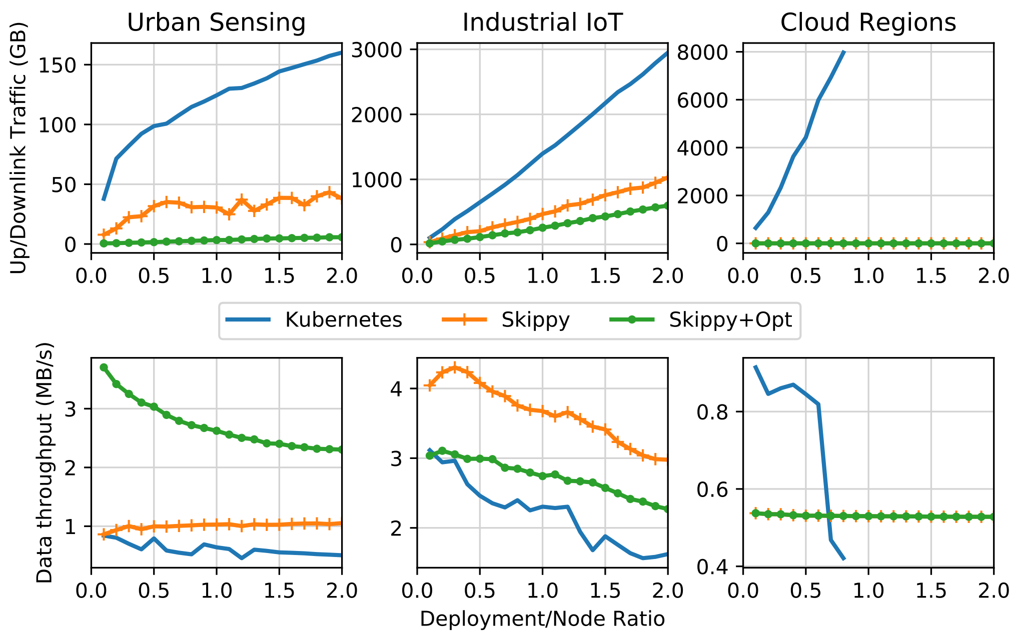


Figure 6.8: Scalability analysis of placements with increasing number of deployments in each scenario. The first row shows the raw inter-network traffic in GB. The second row shows the data throughput of functions, i.e., the overall network traffic per compute second.

Impact of placements on system scalability

We investigate how function placements affect runtime scalability properties of the system, which is one of the major challenges of edge intelligence, as described in Section 2.3.3. In ad-hoc experiments we found that network bottlenecks were the biggest challenge for guaranteeing low function execution times and high throughput. In our scenarios in particular, we were not able to saturate cluster resources before running into extreme network congestion (flows receiving less than 0.1 MBit/s link bandwidth). The most important metric of scalability in our scenarios is therefore network throughput, and whether the placement can maintain high data throughput in the face of an increasing amount of active deployments. To examine this, we run experiments that inject an increasingly larger number of deployments per node. As mentioned earlier, a deployment in our scenario is an instance of one ML pipeline with its three functions. We start at a ratio of 0.1 deployments per node up to 2 deployments per node. Figure 6.8 shows the results of experiment runs without the scale-to-zero policy.

Two things in particular are noteworthy. First, in S2, while the optimized Skippy has a lower data throughput than Skippy, as we have seen in Figure 6.7, it does this to trade off execution cost while maintaining similar function execution times (see Figure 6.6). Second, in some situations, the Kubernetes scheduler produced infeasible placements even with very few deployments. In particular in S3, the inter-region bandwidth was quickly

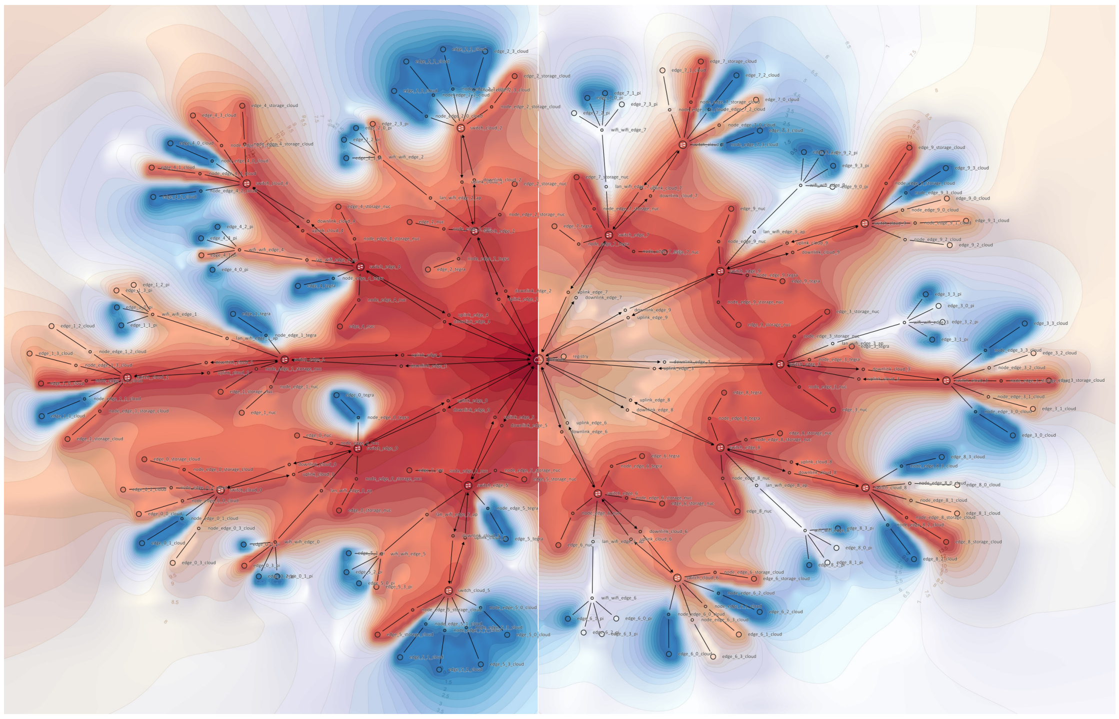


Figure 6.9: A synthetic Industrial IoT scenario topology, augmented with a TAM [PSK+20] that shows the overall link traffic in log-bytes. Blue areas have almost no traffic, red areas have high traffic. The left side shows a placement made by Kubernetes, the right side shows a placement made by the optimized Skippy scheduler.

saturated and leading to infeasible placements. We consider a placement infeasible if it, during the course of a simulation, leads to bottlenecks in the network that degrade the bandwidth allocated to a flow to less than 0.1 MBit/s. Another finding was that, if the OpenFaaS scale-to-zero policy was used, the default scheduler produced no feasible placements in the first scenario. Functions would be rescheduled such that the network was quickly congested with inter-network traffic.

We further demonstrate how Skippy achieves more scalable placements for edge scenarios. Figure 6.9 shows a topology from S2, the IIoT scenario, and traffic data from a single simulation run. The center of the topology is the Internet node, and the branches are individual factory premises (see Section 6.2), connected by up and downlinks. The topographic attribute map (TAM) immediately reveals how Skippy manages to isolate data transfer of functions and storage nodes within edge networks, leading to much less traffic over Internet links.

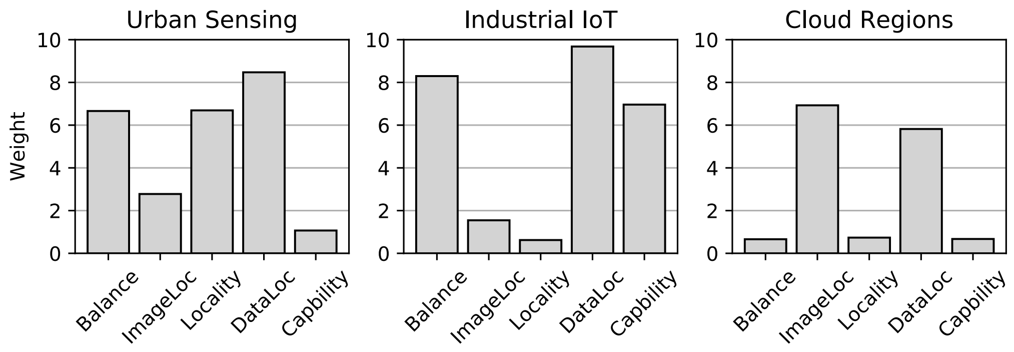


Figure 6.10: Optimized priority function weights in each scenario.

Optimized priority function weights

Figure 6.10 shows the values of $\mathbf{w}$ assigned by the optimization as described in Section 6.4.2, i.e., the optimized weight of each priority function in the different evaluation scenarios. In S1, the capability priority is less relevant, as there is a high percentage of GPU nodes available, which are not saturated. Locality plays a much bigger role in avoiding using the scarce cloud resources. In S2, because there are few GPU nodes, and data is also distributed to on-premises cloud, the data locality and capability priorities are favored. In S3, the results confirm the intuition that resource balance, locality, and capabilities do not have much weight for scheduling in relatively homogeneous environments.

6.5.4 Scheduling latency and throughput

The main source of latency in greedy online MCDM schedulers comes from iterating over nodes and computing priority functions. Because Skippy requires a significant number of priority functions compared to the default Kubernetes scheduler, we think it is worth discussing the resulting impact on scheduling latency and throughput. Let N be the set of all nodes in the cluster, N^c be the set of feasible nodes for scheduling container c, and $\mathcal{S}$ be the set of priority functions. Scheduling requires the evaluation of every priority function $S \in \mathcal{S}$ for every feasible node $n \in N^c$. The algorithmic complexity of scheduling one container c therefore depends on the complexity of the individual priority functions. If we neglect this, i.e., assume that invoking any S is $O(1)$, the complexity of the scoring step is $O(|N^c| \cdot |\mathcal{S}|)$, where $N^c = N$ in the worst case. Because $|N|$ can reach several thousands in a production cluster, the Kubernetes scheduler employs a sampling heuristic to reduce $|N^c|$. The percentage of nodes that are sampled, progressively decreases with the number of nodes in the cluster. Once the cluster reaches $|N| \geq 6500$, the scheduler only considers 5% of available nodes for scoring. This heuristic works under the assumption that the cluster and the network are relatively homogeneous, and that aggressive sampling will not significantly impair placement quality. However, in the case of edge infrastructure, where these assumptions may not hold, this heuristic would introduce extreme variance in the placement quality, which is why we disable it and have to consider all nodes in the

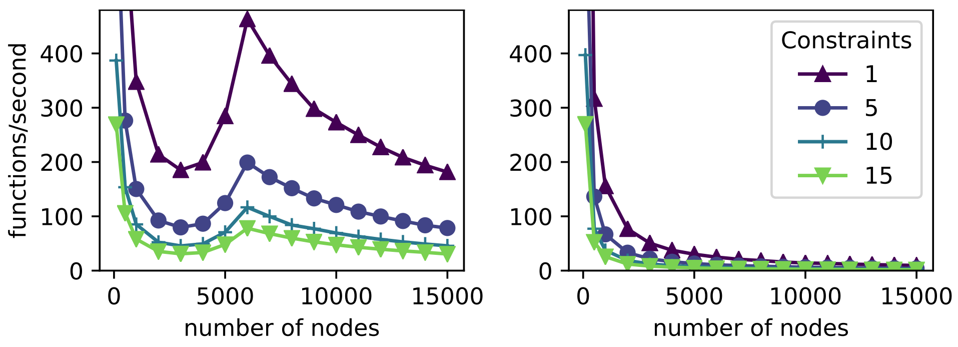


Figure 6.11: Scheduler throughput in functions/second with sampling heuristic (left), and without (right).

cluster. This leads to a general degradation in scheduling throughput. We measured the throughput given different cluster sizes and number of priority functions. Our results in Figure 6.11 roughly match those of a recent Kubernetes performance evaluation [Den16]. The default Kubernetes scheduler only uses two priority functions and the sampling heuristic, which allows it to process around 170 pods per second in a cluster of 10 000 nodes. Whereas Skippy uses by default five priority functions and scores all nodes, which, at 10 000 nodes, yields a throughput of around 15 pods per second. While in our scenarios this is not an issue because scheduling latency is only a small fraction of the overall round-trip time, it does negatively affect scheduling throughput.

6.5.5 Challenges & limitations

Beyond scheduling performance, our system has several limitations that need to be discussed. We also identify several open challenges for serverless edge platforms.

Our system currently makes no particular considerations of the dynamic nature of edge systems. Reconciling deployment and runtime aspects of serverless edge computing applications is especially challenging. Generally, we can distinguish function deployment, function scaling, and function requests to already running functions. Finding good placements for long-living functions is challenging, especially when they are network bound. For functions such as those serving static content or simple image classification tasks, the request RTT perceived by clients will be dominated by link latency between the client and the node hosting the function. Therefore, to make better placement decision for such functions, the system would require knowledge about the location of clients with respect to nodes in the cluster [HKW$^+$18]. In this case, an autoscaling strategy could, for example, spin up replicas that favor nodes in close proximity to clients. This falls into the category of dynamic service migration problems [UWH$^+$15], and is a challenge that confronts edge computing systems in general. The centralized API gateway architecture of OpenFaaS, Kubernetes, and similar systems, presents a serious obstacle in solving this issue, as generally all traffic goes through a type of *ingress* to allow dynamic request routing and load balancing. A strategy could be to replicate API gateways across the network and using a localization mechanism to resolve an API gateway in proximity. This may not be fine-grained enough for scenarios such as the urban sensing infrastructure where the resolution would have to be on city neighborhood level. However, using the principles of elastic diffusion presented in Chapter 5, as we discussed in Section 5.8, could be a promising approach.

Another issues of using state-of-the-art serverless platforms for edge infrastructure is the rudimentary way they model node resources and function requirements [HFG+18]. For example, in Kubernetes, a node has three capacities: CPU, memory and the maximum number of pods that can be scheduled onto the node. Modeling capabilities of edge resources is challenging, as their availability may not be known at design time, and whether they are shareable at runtime. This is particularly important for scarce, (potentially) non-shareable and discrete resources such as GPUs, where containers that use the resource may completely block other containers from execution, while not requiring them often. We therefore see resource modeling as an important aspect of future edge computing platforms.

Using container-based systems can have several drawbacks with respect to isolation and multitenancy. It is currently unclear how our system would behave in a multitenant scenario, where cluster resources are shared between multiple runtimes. Further research is necessary to investigate the effect of, e.g., workload interference.

Our system currently makes the assumption that function code is distributed in container images. Some FaaS platforms, such as OpenWhisk, have platform-level facilities for distributing function code, that may not benefit from the computation movement estimation made by the *LatencyAwareImageLocalityPriority*. Although we could conceive a more higher-level abstraction for a *code movement* soft-constraint, it would require additional facilities to allow the scheduler to query the runtime for function metadata (like its code size), and whether a function's code has been deployed at a particular node.

6.6 Related work

Serverless computing in the form of cloud functions is seen by many in both industry and academia as a computational paradigm shift [JSSS+19, aws, ACR+18, OYZ+18]. Only recently has the serverless model, and in particular the FaaS abstraction, been investigated for edge computing. Gilkson et al. [GND17] proposed the term *Deviceless Edge Computing*, to emphasize how serverless edge computing helps to hide the underlying compute fabric, which consists of edge *devices* rather than *servers*. The characteristics of edge infrastructure exacerbate the challenges of serverless computing [ATC+21], such as platform architecture [NRS+17, RHM+19], runtime overhead [HR19], cold starts [BKB20], or scheduling [RHM+19]. In a recent effort, Baresi and Mendonça [BM19] proposed a serverless edge computing platform based on OpenWhisk. They focus on the complete system architecture design and the implementation of load balancer that considers distributed infrastructure. In industry, AWS IoT Greengrass [aws] enables on-premises execution of AWS Lambda functions, Amazon's serverless computing platform. However, the configuration of AWS IoT Greengrass devices is highly static, since the functions running on a device are defined using a local configuration file. We analyzed the issues surrounding AWS Greengrass for serverless edge AI applications in [Sch19]. In an effort to extend existing serverless runtimes to enable serverless edge computing, Xiong et al. [XSXH18] implemented a set of extensions to Kubernetes called *KubeEdge*. Its most important component, the *EdgeCore* node agent, manages networking, synchronizes state, and potentially masks network failures. Our approach is complementary, in that Skippy provides an edge-enabled scheduling system for making better function placement decisions on edge infrastructure.

Serverless computing is also actively being used and researched in the AI and ML systems problem space. Ishakian et al. [IMS18] discuss the advantages and open challenges of serving ML models using serverless functions. AWS IoT Greengrass currently allows machine learning inference on edge devices, using models trained in the cloud. Our approach is similar in terms of architecture, but different in that it considers AI workflow concepts as first-class citizens in the programming model and underlying runtime, which affects the overall design. Moreover, we go beyond only serving models, and consider all steps in end-to-end AI pipelines. Carriera et al. [CFT+18] implement ML workflows in serverless platforms, and outline an approach that includes an API to develop serverless ML functions, stateful-client resource manager, a worker runtime, and distributed data store. Our approach extends this idea to include management of edge resources.

There is a strong relation between serverless function scheduling and the service placement problem (SPP). Many variants of the SPP for different edge computing system models and operational goals exist [SNS+17, YPI+18, SCY18, HKW+18, ASDL19]. Typically, the problem is formulated as an optimization problem, and an algorithm is implemented to solve an instance of the problem heuristically by leveraging assumptions within the system model. Gog et al. [GSG+16] map the service placement problem to a graphic data structure and model it as a min-cost max-flow (MCMF) problem. Hu et al. [HZdLZ18] pursue a similar approach by modeling the service placement as a min-cost flow problem (MCFP) which allows encoding multi-resource requirements and affinities to other containers. Their scheduler considers the costs for offloading tasks from the edge nodes to rented cloud resources. Aryal and Altmann [AA18] map the service placement problem to a multi-objective optimization problem (MOP) and propose a genetic algorithm to make placement decisions. Bermbach et al. [BMHP19] propose a distributed auction-based approach for a serverless platform in which application developers bid on resources. Generally, scheduling algorithms described in academic literature often assume very detailed information about the system state and service requirements, whereas in production systems, both may not be available. For example, the presented and other approaches [THLL17, XE16], assume that constraints considered by the schedulers are defined a priori, which allows strong assumptions when implementing heuristics. Moreover, they assume that services are known at scheduling time, and a placement is calculated for all services at once. This is not the case for online schedulers, which are the norm for serverless systems, where new services can arrive at any time.

Many online container schedulers, such as the ones from Docker Swarm, Kubernetes, or Apache Mesos, implement a greedy MCDM procedure. A key phase in this procedure is *scoring*, i.e., calculating the score of a feasible node by invoking a set of *priority functions*, building a weighted sum of priority scores, and selecting the highest scoring node for scheduling. The Kubernetes scheduler implements this procedure in a very general and flexible way [MSV18], which is why we build on its model, as it generalizes to many other container schedulers. It allows to dynamically plug in and configure different hard- and soft-constraints, theoretically even at runtime. It is unclear whether and how existing SPP approaches could be implemented in this framework. Our work is an effort to examine how ideas from service placement in edge computing, such as using latency and proximity awareness for placement decision, can translate to, or be implemented in, these types of schedulers.

Another related field is workload offloading, where mobile device workloads are offloaded to cloud or proximate edge or resources. The goal of schedulers in mobile offloading is to determine whether to offload a task or not. The main concern is to minimize execution time and energy consumption of the mobile device [Kha15], and they are often geared towards a specific application. However, the underlying constraints and priorities can be similar to what Skippy does, i.e., determining a tradeoff between data and computation movement, and considering the capabilities of resources to accelerate specific workloads. For example, Eom et al. [EJF$^+$13] model scheduling as a binary decision problem: either offload a workload or run it locally, and investigate the performance of binary classifiers given a simple model. The model considers the ratio between the local execution duration, and the potential data transfer when offloading. Skippy is overall much more general-purpose, and has a flexible way of defining both low-level constraints, as well as high-level operational goals. While there may be some ideas that apply to serverless scheduling, a complete investigation of mobile offloading strategies is out of scope.

6.7 Summary

Serverless computing is a powerful system abstraction that helps platform providers hide operational complexity of the underlying infrastructure from application operators. This makes it attractive for edge computing systems, where, as we have demonstrated in Chapter 3, the operational complexity is particularly high. Analogously to serverless *cloud functions*, we believe that *edge functions* are a promising approach to manage applications that run on a distributed edge computing substrate. We have demonstrated several limitations of existing serverless platforms when they are used in such scenarios, leading to poor function placement on heterogeneous geo-distributed infrastructure that has limited up/downlink connections between edge networks. Our evaluation has also revealed specific limitations of Kubernetes and OpenFaaS for edge scenarios. First, the sampling heuristic of the default Kubernetes scheduler can produce poor or infeasible placement results when the network topology includes geographic distribution and link bottlenecks. Because many different pods may be scheduled to just a few nodes, the nodes become busy pulling new container images, which can consequently lead to network congestion detrimental to system performance. OpenFaaS' scale-to-zero policy exacerbates the issue as pods that are scaled to zero are often restarted on different nodes. Second, the prevalent centralized API gateway architecture is problematic when considering network-bound functions, where most of the round-trip time comes from network link latency, as is the case with, e.g., serving low-latency ML models.

We proposed a serverless platform that elevates concepts from the AI workflow to first-class citizens, and provides a more approachable way to develop and operate edge AI functions. This has the added benefit, that the platform can now reason over the dataflow of functions. The goals of the deviceless function scheduling approach is to consider both edge and cloud resources for function execution, and place functions according to contextual policies. To that end, we presented Skippy, a container scheduling system that enables existing container orchestrators, such as Kubernetes, to support serverless *edge functions*. Skippy runs alongside Kubernetes and OpenFaaS to make better function placement decision in edge computing scenarios. Specifically, it respects device capabilities (such as added AI accelerators), and data locality, and can thereby hide complex operational data and model management tasks from the developer. We introduced scheduling constraints that leverage additional knowledge of node capabilities, the application's dataflow, and the network topology. Overall our experiments show that (1) Skippy enables locality-sensitive function placement, resulting in higher data throughput and less traffic going over up/downlinks, (2) in scenarios where there is a fairly even distribution of cloud and edge resources, the optimization helps significantly in trading off execution cost and overall application latency, and (3) the improved placement quality comes at the cost of scheduler performance. We have shown that the most critical aspect of function placement in data-intensive serverless edge computing is the tradeoff between data and computation movement. However, making this tradeoff in a generalized way is challenging due to the wide range of edge infrastructure scenarios. By introducing higher-level operational goals, we can fine-tune the underlying scheduler parameters to consider infrastructure-specific aspects.

Adding soft constraints to improve placement quality comes at the cost of scheduling throughput. In fact, in large cluster configurations of 10 000 nodes and five soft constraints, the scheduling throughput drops by an order of magnitude. We showed how this is due to a general scalability limitation of state-of-the-art greedy online MCDM serverless scheduling approaches, and their monolithic queue-based design. We doubt that this design will work for more complex scenarios, where many tenants consolidate edge devices to form huge cluster configurations, requiring many soft constraints. Exploring alternative architectures may therefore be a promising direction. Omega [SKAEMW13], a disaggregated shared-state scheduler, has been designed to cope with Google's real-life production workload. Firmament[GSG$^+$16] promises to accomplish sub-second latencies for placing more than ten thousand machines by using multiple min-cost max-flow (MCMF) optimizations. Combining these models could be a promising approach for operating edge cloud systems in general.

There are several open issues to fully realize the idea of edge functions on a distributed compute fabric. For example, the centralized API gateway architecture employed by most state-of-the-art serverless platforms may be impractical for edge computing, particularly with dispersed clients that consume network-bound functions. A possible solution may be using principles of osmotic computing as we discussed in the summary of Chapter 5. Moreover, the dynamic nature of edge systems requires the continuous re-evaluation of placement decisions, necessitating context-aware autoscaling and workload migration strategies. Finally, the automatic characterization of workloads and mapping to their preferred node capabilities could significantly improve function placement, which we explore in subsequent research [Rai21].

CHAPTER 7

Conclusion

This chapter concludes the thesis. First, in Section 7.1 we give a brief summary of the covered topics and the presented contributions. Then, in Section 7.2 we revisit the research questions posited in the introduction. Finally, in Section 7.3 we give a brief outlook on future research directions and open issues.

7.1 Summary

Edge intelligence is a collection of infrastructure architectures, computing paradigms, and operational mechanisms, that deal with the challenges arising from the increasing interconnectedness of humans, Internet connected things, and AI. In this paradigm, new applications have emerged that create new and unique challenges for distributed systems. In particular, they necessitate new distributed systems architectures, and operational mechanisms tailored to deal with the complexity of AI systems, geographic dispersion of compute resources, resource and workload heterogeneity, extreme bandwidth and latency requirements, and stringent privacy constraints. The prevalent cloud-based model with its centralized infrastructure architecture is clearly at odds with these requirements.

In this thesis, we made the case for a distributed computing fabric. First, in Chapter 2 we explained the key principles. The fabric is composed of edge and cloud computing resources, federated to form a diverse and computing substrate. In Chapter 3 we discussed and analyzed different infrastructure models for such a substrate.

Together with three orthogonal systems and their operational mechanisms, we enable a serverless computing model that abstracts this complex distributed infrastructure architecture. (1) In Chapter 3, we presented a prototype for a portable energy-aware edge computer cluster, that could be powered by a car battery or similar power source, making it useful for forward-deployed scenarios. The resulting system efficiently serves stateless services such as image recognition services, while automatically trading off application latency and energy consumption. (2) In Chapter 5 we presented a prototype for a distributed MQTT middleware, that continuously monitors proximity between clients and brokers, can deal with dynamic network topologies, and drastically reduce application latency by leveraging proximate edge resources for message brokering. Complementary, we outlined a system design based on osmotic computing principles. We developed a generalizable mechanism for proximity-based autoscaling that dynamically migrates brokers and clients to the edge. (3) In Chapter 6, we presented a serverless code execution platform tailored to edge intelligence applications. The key contributions are a programming model for serverless edge functions, a custom scheduler, and a method to automatically fine-tune scheduler parameters to the specific infrastructure configuration. The scheduler makes tradeoffs between data and computation movement, and intelligently maps workloads to their appropriate edge resource based on their characteristics. In a series of experiments, we showed how our system is able to maintain a serverless illusion, even in the face of extreme resource heterogeneity and geo-distributed networks.

In Chapter 4, we presented three contributions to improve the evaluation of edge computing systems. First, we presented an experimentation and analytics framework for distributed systems, which we use to create profiling pipelines of the edge cluster testbeds we built. Second, the profiling data is fed into a trace-driven simulation framework that can then be used for scalability analyses, or capacity planning. Third, we presented a model and tool for generating synthetic edge infrastructure configurations that are grounded in the scenarios we analyzed in Chapter 3. We also made the case how these three systems together can be used in a closed loop for developing operational mechanisms for distributed systems in general.

7.2 Research questions

In Section 1.2.1 we posited three core research question that have driven the research presented in this thesis. Their purpose was to (1) examine the current state of edge computing infrastructure architectures, (2) understand the operational mechanisms necessary to manage applications on these systems, and (3) investigate and develop evaluation and development methodologies for these operational mechanisms. We now discuss the research questions and contextualize the contributions of this thesis.

RQ1. Which are appropriate infrastructure architectures that enable a distributed compute fabric for edge intelligence?

From the analysis in Chapter 3 of existing and emerging edge computing infrastructure architectures, we can make the following observations. First, edge computers will not live in isolation. Much like data center servers live together in racks as cohesive infrastructure units, multiple edge computers will form high-density compute clusters. Unlike data center servers, these clusters will be much smaller and more diverse in terms of hardware platforms. General-purpose computing platforms will be complemented by GPUs, modular AI accelerators, FPGAs, or ASICs. We therefore argued that multi-purpose edge computers are the cohesive infrastructure unit analog to server racks, which will come in various shapes and sizes. For portable or forward-deployed scenarios, in Section 3.2 we have introduced a concrete proposal for an energy-aware cluster-based edge computer, that takes energy into account as first-class citizen in their operational mechanisms.

In the past, efforts to centralize or decentralize computing systems have oscillated into both extremes, ranging from pure peer-to-peer networks, to consolidation of IT resource into massive cloud data centers. While edge computing has ushered in a new wave of decentralization, we observe a certain convergence to hybrid and cooperative systems. Multi-purpose edge computers that are geographically dispersed will be federated together with centralized cloud resources, and form a geographically dispersed *computing substrate*. Mechanisms for cluster federation and decentralized resource management will become more relevant as the dispersion of edge computing resources progresses. System abstractions for cloud computing have become very effective at hiding the underlying computing hardware, as demonstrated by the serverless computing model. We see container-based virtualization as the way forward in these systems, as opposed to VM-based hardware virtualization, since the lightweight nature of containers allows us to include a much wider range of devices into the computing substrate. Maintaining this serverless illusion in edge computing systems will be extremely valuable, but significantly more challenging, as we have shown in Chapter 6.

RQ2. Which aspects of edge intelligence systems should be abstracted as first-class citizens into platforms that manage such systems?

From our analysis of edge intelligence applications and their requirements in Chapter 2 and Chapter 3, we can make the general observation that all edge computing systems have at least three goals: (1) optimize application responsiveness, (2) balance system resource usage, and (3) minimize dependence on cloud resources (for reasons such as fault tolerance, privacy constraints, cost, or to reduce Internet traffic). Moreover, when juxtaposed with cloud computing systems, the discerning characteristics of edge computing systems are: (a) decentralization of computing resources and data, (b) clients and computing resources may be mobile, (c) resulting physical and logical distance between clients and resources, and (d) heterogeneity of computing resources to enable a wide range of use cases. To reconcile edge computing goals and system characteristics, we identify the following three critical concepts that should be considered as first-class citizens in operational mechanisms.

Proximity

Proximity plays a fundamental role in the operational mechanisms of edge computing systems, as we demonstrated in both Chapter 5 and Chapter 6. Its impact on system design is so profound, in fact, that we dedicate to it a longer discussion.

Two issues arise when using proximity in the way we presented: its definition, and its measurement. Proximity can be defined in different ways. Classic spatial or geographic proximity as it is used in CDNs or cloud availability regions, plays a lesser role for the systems we are interested in. Instead, we describe proximity as a function of network latency or available bandwidth. This turns out to be useful because these values can change due to resource contention, which has the same potential consequence on system performance as physical distance. In a messaging middleware for example, a spatially *close* broker that is slow to respond behaves as if it were farther away. Moreover, we have shown the challenges of measuring proximity in a scalable way. Opportunistic monitoring techniques and abstraction via virtual network coordinate systems help significantly in reducing the network strain caused by continuous monitoring, but also tend to reduce the precision of distance estimations.

We have shown how considering proximity as a first-class citizen in operational mechanisms can help significantly in meeting certain operational goals in edge computing systems. From the prototypes we have built, we can formulate the following generalizable mechanisms.

Proximity-based autoscaling. In cloud computing, autoscaling decisions are typically made independently of placement decisions. This is firstly because it is easier to implement, and secondly because, in cloud computing, it generally does not matter where service replicas are placed since they will all live in the same data center. As an example, the default configuration of the Kubernetes scheduler uses soft constraints that favor nodes that already have the necessary container image to run the service, and prefer nodes with more free resources in order to balance load throughout the system. Proximity between nodes is not considered to have a particular significance. However, in edge computing systems, where clusters of clients may spontaneously appear and disappear, and proximity to compute resources matters significantly for application performance, the two decisions should be made in tandem. It is not only important *when* to autoscale, but also *where* to autsocale. The results presented in Chapter 5 demonstrate the importance of elasticity towards the edge using proximity-based autoscaling and placement. Although our prototype was developed specifically in the context of messaging middleware, the ideas translate to other systems as well. For example, similar principles can be applied to the problem of API gateways in microservice-based architectures, or serverless function scaling. Replication of functions also necessitates request routing and load balancing. The classic round-robin load balancing mechanisms treat each backend server equally. When proximity between clients and servers matter, however, servers are not equally performant from the client's point of view. This is especially the case for network-bound workloads, such as low-latency ML inferencing, or message brokering. We showed how grouping servers into proximity groups can help balance the goals of load distribution and latency minimization.

Data and computation movement tradeoffs. We motivated the work in Chapter 6 with experiments highlighting the challenges of treating geographically distributed computing and storage resources as a single federated computing substrate. The resulting variance in the up/downlink bandwidth between networks necessitates the explicit consideration of distance between compute and storage nodes. This becomes especially apparent when scheduling data-intensive functions, such as ML model training jobs, that operate on data stored in unstructured storage systems (like distributed S3 instances). In these scenarios, there is a tradeoff to be made between moving the data to the cloud, or moving the computation to the edge, where moving computation means pulling a container or VM image from the cloud. Here, a definition of proximity based on network bandwidth is useful to re-use mechanisms we built that leverage proximity as an abstraction in their algorithms.

Energy

As long as we rely on burning fossil fuels for generating electrical energy, there will be a case for energy-aware solutions. More practically, however, energy-awareness plays a critical role in forward-deployed or mobile computing scenarios, to improve battery lifetime and therefore the overall utility of the system. In Section 3.2, we presented a prototype for a portable cluster-based edge computer, that could be powered for several hours with a car battery. We showed how the tradeoff between performance and energy consumption at this system scale are much more pronounced than in data centers. Also, we showed some unintuitive system behavior when using modern high-density CPU platforms, that are already extremely energy efficient, in a cluster that applies another layer of energy optimization. As part of our simulation model presented in Section 4.3, we described an ML-based energy modelling technique, that can help operational mechanisms make decisions based on estimated energy use of individual nodes and the entire system as a whole. These models could become a first-class citizen of resource managers or load balancers. Much like laptops or mobile phones have a power-saving mode, we conceive that cluster-based edge computers have such a mode that gives more weight to energy-conserving mechanisms.

Device and workload heterogeneity

A characteristic often ascribed to edge systems is heterogeneity, the meaning of which remains somewhat vague. In our model for synthesizing infrastructure configurations (see Section 4.2), we presented a mathematical description of heterogeneity, that allows us to use it as a quantifiable system property. The model focuses on system resources such as CPU cores, available memory, and additional capabilities like GPUs or AI accelerators. We showed how sensitive some operational mechanisms are to this type of system heterogeneity. For example, in the motivation of Chapter 6, we demonstrated the impact of correctly mapping device capabilities to workload characteristics. System performance depends greatly on precise placement decisions, like placing functions that perform ML model training on edge nodes with GPUs or AI accelerators. The problem is that VMs or containers are black-boxes, and we generally don't have a way of knowing a-priori whether they can benefit from certain accelerators. Moreover, as we demonstrated in Section 4.3, without explicit modelling, it is hard to reason about the effect of resource contention on system performance. We proposed a black-box model that characterizes workloads based on their resource usage. By measuring performance metrics and resource usage on certain devices, we can, during system runtime, determine which cluster nodes are best suited for which workloads.

RQ3. What are appropriate methods for developing and evaluating edge computing systems to allow more generalizable conclusions?

Through our work on synthesizing edge infrastructure configurations in Section 4.2, we have shown how drastically different edge computing infrastructure can be, depending on their use case. As we just discussed in the previous paragraph on heterogeneity, operational mechanisms can be very sensitive to these differences. Because we can no longer build on the assumptions of a centralized and homogeneous datacenter, building generalizable operational mechanisms is much more challenging. Instead, we need to be able to rapidly vary the parameters of infrastructure like resource heterogeneity or network topologies. It is challenging to do this using real testbeds, which is why simulation and emulation will continue to play an important role in edge computing systems.

As we have shown in Section 4.3, many existing simulation or emulation tools make simplified assumptions that lead to inaccurate results. It is unclear to which degree that affects the validity of results presented in systems research that make use of the simulators. We argue that using stochastic trace-driven simulation, as compared to prevalent analytical or queue-based models, should be the preferred way of doing systems evaluations.

A streamlined way of profiling edge computing systems is therefore a critical part of the evaluation methodology. With our profiling system, we made a step forward to a more systematic approach for benchmarking and profiling. However, the biggest issue is still the scarcity of reference benchmarks for edge computing. We hope that, as the number of applications and requirements for edge computing systems increases, standardized benchmarks will emerge that allow us to better compare operational mechanisms.

7.3 Future work

It is possible that no consensus on edge computing infrastructure architectures will be achieved, and rather there will be a variety of co-existing models used for different use cases. In this scenario, developing generalizable operational mechanisms is challenging. Schedulers, autoscalers, and load-balancers, need to have fine-grained parameters that can be tuned to the infrastructure. The improvement of methods for self-adaptive systems will be particularly important, to reduce the manual, and potentially error prone, configuration a human has to do. A benefit of having simulators that are grounded in real profiling data, is that we can use them to make probabilistic predictions about the system behavior given different runtime situations. With a tight loop between profiling and simulation, we see an opportunity to use simulation not only for evaluating systems, but also as a core tool in operational mechanisms. For example, we showed in Section 6.4, how simulation can be used to optimize scheduler parameters used for the system being simulated. We conceive that, in the future, these simulators will be updated at runtime using monitoring data, and then used by schedulers, autoscalers, or load balancers, to play through various scenarios. The success of this approach is however contingent on accurate and performant simulations.

In Chapter 6 we juxtaposed serverless function scheduling with the Service Placement Problem (SPP), where generally multiple services are scheduled at once. We argued that these approaches are not useful for the systems we are interested in, because in these systems, workloads (such as serverless functions), are generally not known upfront and can arrive at any point during runtime. What could be done, however, is leveraging existing SPP solutions to determine the optimal placement of the already scheduled functions, and then determining a set of operations to converge from the current state to the optimal state. Coming up with ways to calculate the cost–benefit tradeoff of migrating serverless functions in this way, could be promising research.

The systems presented in this thesis mostly rely on centralized control mechanisms that govern the system. This can be problematic. As we showed in Chapter 6, a centralized monolithic scheduler struggles with large clusters and many scheduling constraints. A divide-and-conquer strategy may solve the scalability issues, but it is unclear whether the added complexity of decentralized control mechanisms would outweigh their benefits.

Finally, for the full end-to-end realization of smart city scale edge intelligence, there are still many non-functional issues that warrant further investigation. In particular concerning ownership and stake of edge computing infrastructure. With respect to stake, we observe that there are three categories of edge intelligence use cases: public (such as smart public spaces), private (personal health assistants (personal), predictive maintenance (corporate)), and intersecting (such as autonomous vehicles). It is unclear who will own the future fabric for edge intelligence, whether utility-based offerings for edge computing will take over as is the case in cloud computing, whether telecommunications will keep up with the development of mobile edge computing, what role governments and the public will play, and how the answers to these questions will impact engineering practices and system architectures.

Bibliography

[5G-19] 5G-TRANSFORMER project. 5G-TRANSFORMER final system design and techno-economic analysis. Public Deliverable D1.4, November 2019.

[5GA19] 5GAA. C-ITS vehicle to infrastructure services: How C-V2X technology completely changes the cost equation for road operators. White paper, 5G Automotive Association, January 2019.

[AA18] R. G. Aryal and J. Altmann. Dynamic application deployment in federations of clouds and edge resources using a multiobjective optimization AI algorithm. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 147–154, April 2018.

[ACR$^+$18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[AFGM$^+$15] A Al-Fuqaha, M Guizani, M Mohammadi, M Aledhari, and M Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.

[AH16] Sherif Abdelwahab and Bechir Hamdaoui. FogMQ: A message broker system for enabling distributed, internet-scale IoT applications over heterogeneous cloud platforms. *CoRR*, abs/1610.0, 2016.

[AKGH17] Kyoungho An, Shweta Khare, Aniruddha Gokhale, and Akram Hakiri. An autonomous and dynamic coordination and discovery service for wide-area peer-to-peer publish/subscribe: Experience paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 239–248, New York, NY, USA, 2017. ACM.

[Ama20] Amazon. AWS lambda pricing. AWS. Online. Accessed 2020-02-19. `https://aws.amazon.com/lambda/pricing/`, 2020.

[AP08] Charu C Aggarwal and S Yu Philip. A general survey of privacy-preserving data mining models and algorithms. In *Privacy-preserving data mining*, pages 11–52. Springer, 2008.

[ASDL19] Farah AIT SALAHT, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. Research Report RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France, October 2019.

[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 *19th Australasian Symposium on Parallel and Distributed Computing*, AusPDC'21, 2021.

[Aus] Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology. Senderkataster Austria. `https://www.senderkataster.at/`.

[aws] AWS IoT greengrass. `https://aws.amazon.com/greengrass/`.

[Bar15] Jeff Barr. AWS IoT – cloud services for connected devices. AWS Blog. Online. Posted 2015-01-08. Accessed 2017-09-18. `https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/`, 2015.

[BB10] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831. IEEE, 2010.

[BB12] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, sep 2012.

[BBC+17] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, and Others. 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. ACM, 2017.

[BBD+17] B Bhattacharjee, S Boag, C Doshi, P Dube, B Herta, V Ishakian, K R Jayaram, R Khalaf, A Krishna, Y B Li, V Muthusamy, R Puri, Y Ren, F Rosenberg, S R Seelam, Y Wang, J M Zhang, and L Zhang. IBM deep learning service. *IBM Journal of Research and Development*, 61(4/5):10:1–10:11, 2017.

[BBH+17] Arsany Basta, Andreas Blenk, Klaus Hoffmann, Hans Jochen Morper, Marco Hoffmann, and Wolfgang Kellerer. Towards a cost optimal design for a 5G mobile core network based on SDN and NFV. *IEEE Trans. Network and Service Management*, 14(4):1061–1075, 2017.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BCR14] P Bellavista, A Corradi, and A Reale. Quality of service in wide scale publish–subscribe systems. *IEEE Communications Surveys Tutorials*, 16(3):1591–1616, 2014.

[Bec18] Pete Beckman. Artificial intelligence at the edge: How machine learning at the edge is creating a computing continuum. Talk. `https://www.youtube.com/watch?v=N_k8Uh8Bl0E`, 2018. Research Computing Centre.

[BJP+20] Philip J. Basford, Steven J. Johnston, Colin S. Perkins, Tony Garnock-Jones, Fung Po Tso, Dimitrios Pezaros, Robert D. Mullins, Eiko Yoneki, Jeremy Singer, and Simon J. Cox. Performance analysis of single board computer clusters. *Future Generation Computer Systems*, 102:278–291, 2020.

[BKB20] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th ACM Symposium on Applied Computing (SAC 2020)*. ACM, 2020.

[BM19] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing*, ICFC'19, pages 1–10. IEEE, 2019.

[BMHP19] David Bermbach, Setareh Maghsudi, Jonathan Hasenburg, and Tobias Pfandzelter. Towards auction-based function placement in serverless fog platforms. *arXiv preprint arXiv:1912.06096*, 2019.

[BPB+19] Lorenzo Bertizzolo, Michele Polese, Leonardo Bonati, Abhimanyu Gosain, Michele Zorzi, and Tommaso Melodia. mmBAC: Location-aided mmwave backhaul management for UAV-based aerial cells. In *Proceedings of the 3rd ACM Workshop on Millimeter-wave Networks and Sensing Systems*, pages 7–12, 2019.

[Bru21] Andreas Bruckner. Self-adaptive distributed MQTT middleware for edge computing applications. Master's thesis, TU Wien, 2021.

[BT19] Dimo Brockhoff and Tea Tušar. Benchmarking algorithms from the platypus framework on the biobjective bbob-biobj testbed. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1905–1911, 2019.

[Bur17] Doug Burger. Microsoft unveils project Brainwave for real-time AI. *Microsoft Research, Microsoft*, 22, 2017.

[BWFS14] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. In *AFIN 2014 : The Sixth International Conference on Advances in Future Internet*. Citeseer, 2014.

[CAR05] N Carvalho, F Araujo, and L Rodrigues. Scalable QoS-based event routing in publish-subscribe systems. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 101–108, jul 2005.

[Cas19] Stephen Cass. Taking AI to the edge: Google's TPU now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[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*, pages 26–33. ACM, 2017.

[CDOK17] Vittorio Cozzolino, Aaron Yi Ding, Jorg Ott, and Dirk Kutscher. Enabling fine-grained edge offloading for IoT. In *Proceedings of the SIGCOMM Posters and Demos*, pages 124–126. 2017.

[CDZ97] Kenneth L Calvert, Matthew B Doar, and Ellen W Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, 1997.

[CFT$^+$18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. *Workshop on Systems for ML and Open Source Software at NeurIPS 2018*, 2018.

[Cha99] Xinjie Chang. Network simulations with OPNET. In *WSC'99. 1999 Winter Simulation Conference Proceedings. 'Simulation - A Bridge to the Future' (Cat. No. 99CH37038)*, volume 1, pages 307–314. IEEE, 1999.

[CHW$^+$17] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, and et al. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In

Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC '17, 2017.

[Cis19] Cisco public. Redefine connectivity by building a network to support the Internet of Things. White Paper. `https://www.cisco.com/c/en/us/solutions/service-provider/a-network-to-support-iot.html`, 2019.

[Cis21] Cisco public. Cisco annual internet report (2018–2023). White Paper. `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, Apr 2021.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[Cor17] IBM Corporation. Medtronic builds a cognitive mobile personal assistant app to assist with daily diabetes management, 2017. IBM Case Studies.

[Cor18] IBM Corporation. An AI-powered assistant for your field technician. Technical report, 2018.

[CPSC16] Xudong Chen, Swee King Phang, Mo Shan, and Ben M Chen. System integration of a vision-guided UAV for autonomous landing on moving platform. In *12th IEEE International Conference on Control and Automation (ICCA)*, pages 761–766. IEEE, 2016.

[CRB+11] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF 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.

[CS05] Yuan Chen and Karsten Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In *Middleware 2005: ACM/IFIP/USENIX 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005. Proceedings*, pages 354–374. Springer Berlin Heidelberg, 2005.

[Cut18] Bruce Cutler. Examining cross-region communication speeds in AWS. Medium. Online. Posted 208-05-30. Accessed 2020-02-15. `https://medium.com/slalom-technology/examining-cross-region-communication-speeds-in-aws-9a0bee31984f`, 2018.

[CWC+18] Baotong Chen, Jiafu Wan, Antonio Celesti, Di Li, Haider Abbas, and Qin Zhang. Edge computing in IoT-based manufacturing. *IEEE Communications Magazine*, 56(9):103–109, 2018.

[CZVZ14] Angelo Cenedese, Andrea Zanella, Lorenzo Vangelista, and Michele Zorzi. Padova smart city: An urban internet of things experimentation. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *ACM SIGCOMM Computer Communication Review*, 34(4):15–26, 2004.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[Den16] Hongchao Deng. Improving Kubernetes scheduler performance. CoreOS Blog. Online. Posted 2016-02-22. Accessed 2019-03-14. `https://coreos.com/blog/improving-kubernetes-scheduler-performance.html`, 2016.

[DNŠ17] Schahram Dustdar, Stefan Nastić, and Ognjen Šćekić. *Smart Cities: The Internet of Things, People and Systems.* Springer, 2017.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[DPVW+17] Yvonne Dierikx-Platschorre, Jaap Vreeswijk, Anton Wijbenga, Patrick Hofman, and Ruben van Ardenne. Guideline placement C-ITS roadside units. In *Proc. 12th ITS European Congress*, 2017.

[Duc18] Chris Duckett. Baidu creates Kunlun silicon for AI. ZDNet. Online. https://www.zdnet.com/article/baidu-creates-kunlun-silicon-for-ai/, 2018.

[EFGK03] Patrick Th. Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, jun 2003.

[Eiv17] Adam Eivy. Be wary of the economics of serverless cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[EJF$^+$13] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer. Machine learning-based runtime scheduler for mobile offloading framework. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC'13, pages 17–25, 2013.

[EKR03] E N (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In Babak Falsafi and T N Vijaykumar, editors, *Power-Aware Computer Systems*, pages 179–197, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[EPM13] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. Cloud computing: Concepts, technology & architecture, 2013.

[EPR$^+$17] Yehia Elkhatib, Barry Porter, Heverson B Ribeiro, Mohamed Faten Zhani, Junaid Qadir, and Etienne Riviere. On using micro-clouds to deliver the fog. *IEEE Internet Computing*, 21(2):8–15, mar 2017.

[ER59] Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6(290-297):18, 1959.

[ETS19] ETSI GS MEC 003. *Multi-Access Edge Computing (MEC); Framework and Reference Architecture*, January 2019. V2.1.1.

[FBCC$^+$10] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building Watson: An overview of the DeepQA project. *AI magazine*, 31(3):59–79, 2010.

[FCFMO$^+$19] 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*, page 101966, 2019.

[Fel00] Anja Feldmann. Characteristics of TCP connection arrivals. *Self-Similar Network Traffic and Performance Evaluation*, pages 367–399, 2000.

[FLP14] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 500–507. IEEE, feb 2014.

[Fra16] Dustin Franklin. Nvidia® Jetson TX1 supercomputer-on-module drives next wave of autonomous machines. *Parallel Forall. NVIDIA Corporation*, 12, 2016.

[Gar16] Michael Garcia. How to bridge mosquitto MQTT broker to AWS IoT. *The Internet of Things on AWS – Official Blog*, 2016.

[Gar17] Gartner. Integrate DevOps and artificial intelligence to accelerate IT solution delivery and business value. Online. Accessed 2018-09-05. `https://www.gartner.com/doc/3787770/integrate-devops-artificial-intelligence-accelerate`, 2017.

[GB14] Nikolay Grozev and Rajkumar Buyya. Multi-cloud provisioning and load distribution for three-tier applications. *TAAS*, 9:13:1–13:21, 2014.

[Gei19] Manuel Geier. Enabling mobility and message delivery guarantees in distributed MQTT networks. Master's thesis, TU Wien, 2019.

[GHBRK12] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):14, 2012.

[GKK$^+$19] Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Schahram Dustdar, Ognjen Scekic, Thomas Rausch, Stefan Nastic, Sasko Ristov, and Thomas Fahringer. A deviceless edge computing approach for streaming IoT applications. *IEEE Internet Computing*, 23(1):37–45, 2019.

[GLME$^+$15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[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 '17, 2017.

[GOM18] David Guyon, Anne-Cécile Orgerie, and Christine Morin. An experimental analysis of paas users parameters on applications energy consumption. In *IC2E 2018-IEEE International Conference on Cloud Engineering*, pages 1–7, 2018.

[GSG$^+$16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, Savannah, GA, 2016. USENIX Association.

[GSGKK15] J Gascon-Samson, F P Garcia, B Kemme, and J Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496, jun 2015.

[GSKK17] Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. Multipub: Latency and cost-aware global-scale cloud publish/subscribe. In *37th International Conference on Distributed Computing Systems*, ICDCS'17, pages 2075–2082. IEEE, 2017.

[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.

[GŽB$^+$14] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37, 2014.

[Had17] D Hadka. Platypus: A free and open source python library for multi-objective optimization. Open source project. `https://github.com/Project-Platypus/Platypus`, 2017.

[HAHZ15] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 9–16. IEEE, 2015.

[HAW17] Ilija Hadžić, Yoshihisa Abe, and Hans C Woithe. Edge computing in the ePC: A reality check. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 1–10, 2017.

[HBS$^+$15] Kai Hwang, Xiaoying Bai, Yue Shi, Muyang Li, Wen-Guang Chen, and Yongwei Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on parallel and distributed systems*, 27(1):130–143, 2015.

[HCH$^+$14] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81. ACM, 2014.

[HDB17] Jeremy Hermann and Mike Del Balso. Meet michelangelo: Uber's machine learning platform, 2017.

[HFG$^+$18] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[HJO$^+$10] Ali R Hurson, Evens Jean, Machigar Ongtang, Xing Gao, Yu Jiao, and Thomas E Potok. Recent advances in mobile agentoriented applications. *Mobile Intelligence: Mobile Computing and Computational Intelligence*, pages 106–139, 2010.

[HKM$^+$17] Daniel Happ, Niels Karowski, Thomas Menzel, Vlado Handziski, and Adam Wolisz. Meeting IoT platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1-2):41–52, feb 2017.

[HKW$^+$18] Ting He, Hana Khamfroush, Shiqiang Wang, Tom La Porta, and Sebastian Stein. It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In *2018 IEEE 38th International Conference on Distributed Computing Systems*, ICDCS'18, pages 365–375. IEEE, 2018.

[HLR$^+$08] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[HMD15] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[HMR$^+$19] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, and Kaoutar El Maghraoui. Modelops: Cloud-based lifecycle management for reliable and trusted AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019.

[HR19] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI'19, pages 225–236. Association for Computing Machinery, 2019.

[HRM$^+$18] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[HZC$^+$17a] 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. *CoRR*, abs/1704.04861, 2017.

[HZC$^+$17b] 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.

[HZdLZ18] Yang Hu, Huan Zhou, Cees de Laat, and Zhiming Zhao. ECSched: Efficient container scheduling on heterogeneous clusters. In *European Conference on Parallel Processing*, pages 365–377. Springer, 2018.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IBM18] IBM Corporation. IBM Watson machine learning. Online. `https://developer.ibm.com/clouddataservices/docs/ibm-watson-machine-learning/`, 2018.

[IBW$^+$19] Haris Isakovic, Vanja Bisanovic, Bernhard Wally, Thomas Rausch, Denise Ratasich, Schahram Dustdar, Gerti Kappel, and Radu Grosu. Sensyml: Simulation environment for large-scale IoT applications. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 3024–3030. IEEE, 2019.

[IHM$^+$16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and$<$ 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[IMS18] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering*, IC2E '18, pages 257–262, 2018.

[inta] AWS inter-region latency monitoring. `https://www.cloudping.co/`.

[intb] InterCor: Interoperable corridors deploying cooperative intelligent transport systems, connecting europe facility (CEF) programme (2014-2020). `https://intercor-project.eu/`.

[Int18] Intel. Intel unveils the Intel Neural Compute Stick 2 at Intel AI devcon Beijing for building smarter AI edge devices. Online. `https://newsroom.intel.com/news/intel-unveils-intel-neural-compute-stick-2/`, 2018.

[IRH$^+$18] Haris Isakovic, Denise Ratasich, Christian Hirsch, Michael Platzer, Bernhard Wally, Thomas Rausch, Dejan Nickovic, Willibald Krenn, Gerti Kappel, Schahram Dustdar, et al. CPS/IoT Ecosystem: A platform for research and education. In *Cyber Physical Systems. Model-Based Design*, pages 206–213. Springer, 2018.

[JBP$^+$18] Steven J Johnston, Philip J Basford, Colin S Perkins, Herry Herry, Fung Po Tso, Dimitrios Pezaros, Robert D Mullins, Eiko Yoneki, Simon J Cox, and Jeremy Singer. Commodity single board computer clusters and their applications. *Future Generation Computer Systems*, 89:201–212, 2018.

[JCL$^+$10] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. IGI Global, 2010.

[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.

[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.

[KG15] S. P. T. Krishnan and Jose L. Ugia Gonzalez. *Google Cloud Pub/Sub*, pages 277–292. Apress, Berkeley, CA, 2015.

[KGL$^+$20] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. DC-store: Eliminating noisy neighbor containers using deterministic i/o performance and resource isolation. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 183–191, 2020.

[Kha15] Minhaj Ahmad Khan. A survey of computation offloading strategies for performance improvement of applications running on mobile devices. *Journal of Network and Computer Applications*, 56:28 – 40, 2015.

[KKH$^+$09] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, mar 2009.

[KKY$^+$10] Minkyong Kim, Kyriakos Karenos, Fan Ye, Johnathan Reason, Hui Lei, and Konstantin Shagin. Efficacy of techniques for responsiveness in a wide-area publish/subscribe system. In *Proceedings of the 11th International Middleware Conference Industrial Track*, Middleware Industrial Track '10, pages 40–45. ACM, 2010.

[KRM06] Diwakar Krishnamurthy, Jerome A Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, 32(11):868, 2006.

[Kub19] Kubernetes Community. Kubernetes scheduler, 2019. Kubernetes Documentation. Accessed 2019-01-04. `https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/`.

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[LCC^{+}15] J E Luzuriaga, J C Cano, C Calafate, P Manzoni, M Perez, and P Boronat. Handling mobility in IoT applications using the MQTT protocol. In *2015 Internet Technologies and Applications (ITA)*, pages 245–250, Sep 2015.

[LES^{+}14] Grace Lewis, Sebastian Echeverria, Soumya Simanta, Ben Bradshaw, and James Root. Tactical cloudlets: Moving cloud computing to the edge. In *2014 IEEE Military Communications Conference*, pages 1440–1446. IEEE, oct 2014.

[LH18] Qiang Liu and Tao Han. Dare: Dynamic adaptive mobile augmented reality with edge computing. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2018.

[LMFJ^{+}04] K. Lorincz, D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, Oct 2004.

[LRD19a] Clemens Lachner, Thomas Rausch, and Schahram Dustdar. Context-aware enforcement of privacy policies in edge computing. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 1–6. IEEE, 2019.

[LRD19b] Clemens Lachner, Thomas Rausch, and Schahram Dustdar. ORIOT: A source location privacy system for resource constrained IoT devices. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[LRD21] Clemens Lachner, Thomas Rausch, and Schahram Dustdar. A privacy preserving system for ai-assisted video analytics. In *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, 2021.

[LWB16] P. Liu, D. Willis, and S. Banerjee. ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13, 2016.

[Mar21] Cynthia Marcelino. Locality-aware data management for serverless edge computing. Master's thesis, TU Wien, 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.

[MB17] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[MBLN+19] Johann Marquez-Barja, Bart Lannoo, Dries Naudts, Bart Braem, Carlos Donato, Vasilis Maglogiannis, Siegfried Mercelis, Rafael Berkvens, Peter Hellinckx, Maarten Weyn, Ingrid Moerman, and Steven Latre. Smart highway: ITS-G5 and C-V2X based testbed for vehicular communications in real environments enhanced by edge/cloud technologies. In *Proc. 28th European Conference on Networks and Communications (EuCNC)*, 2019.

[MG18] Gabriel Axel Montes and Ben Goertzel. Distributed, decentralized, and democratized artificial intelligence. *Technological Forecasting and Social Change*, 2018.

[MGG+17] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *2017 IEEE Fog World Congress*, FWC'17, pages 1–6. IEEE, 2017.

[MJ10] Vinod Muthusamy and Hans-Arno Jacobsen. Mobility in publish/subscribe systems. *Mobile Intelligence*, pages 62–86, 2010.

[MJS+17] Ahsan Morshed, Prem Prakash Jayaraman, Timos Sellis, Dimitrios Georgakopoulos, Massimo Villari, and Rajiv Ranjan. Deep osmosis: Holistic distributed deep learning in osmotic computing. *IEEE Cloud Computing*, 4(6):22–32, 2017.

[MK20] Andras Markus and Attila Kertesz. A survey and taxonomy of simulation environments modelling fog computing. *Simulation Modelling Practice and Theory*, 101:102042, 2020.

[MKN+17] Simone Mangiante, Guenter Klas, Amit Navon, Zhuang GuanHua, Ju Ran, and Marco Dias Silva. VR is on the edge: How to deliver 360 videos in mobile networks. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 30–35, 2017.

[MLDD17] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. ModelHub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering*, ICDE'17, pages 1393–1394. IEEE, 2017.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An approach to universal topology generation. In *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 346–353. IEEE, 2001.

[MM08] Bratislav Milic and Miroslaw Malek. *NPART - Node Placement Algorithm for Realistic Topologies in Wireless Multihop Network Simulation.* Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Institut für Informatik, 2008.

[MMR$^+$13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, 2013.

[MMRyA16] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[Mor97] Robert Morris. TCP behavior with many flows. In *Proceedings 1997 International Conference on Network Protocols*, pages 205–211. IEEE, 1997.

[Mor17] R. Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[MPD19] Salma Abdel Magid, Francesco Petrini, and Behnam Dezfouli. Image classification on IoT edge devices: profiling and modeling. *Cluster Computing*, pages 1–19, 2019.

[MS18] Ryan Maguire and Rob Schmit. Hybrid machine learning: From the cloud to the edge. Talk. `https://www.youtube.com/watch?v=M-29naAVmI4`, 2018. CloudNext '18.

[MSS12] V. Mathew, R. K. Sitaraman, and P. Shenoy. Energy-aware load balancing in content delivery networks. In *2012 Proceedings IEEE INFOCOM*, pages 954–962, March 2012.

[MSV18] Janakiram MSV. How Kubernetes is transforming into a universal scheduler. The New Stack. Online. Posted 2018-09-07. Accessed 2019-03-14. `https://thenewstack.io/how-kubernetes-is-transforming-into-a-universal-scheduler`, 2018.

[mxn] MXNet: a scalable deep learning framework. Open source project. `https://mxnet.apache.org/`.

[NND$^+$17] Matteo Nardelli, Stefan Nastic, Schahram Dustdar, Massimo Villari, and Rajiv Ranjan. Osmotic flow: Osmotic computing+ IoT workflow. *IEEE Cloud Computing*, 4(2):68–75, 2017.

[NRS$^+$17] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[NTD16] S. Nastic, H. Truong, and S. Dustdar. A middleware infrastructure for utility-based provisioning of IoT cloud systems. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, volume 00, pages 28–40, 2016.

[Nun05] Flávio Nunes. Aveiro, portugal: Making a digital city. *Journal of Urban Technology*, 12(1):49–70, 2005.

[NVI] NVIDIA. NVIDIA Jetson - the AI platform for autonomous machines. Online. `https://developer.nvidia.com/embedded/develop/hardware`.

[opea] Apache OpenWhisk Composer. Open source project. `https://github.com/apache/incubator-openwhisk-composer`.

[opeb] OpenCellid. `https://opencellid.org`.

[opec] OpenSignal coverage maps. `https://www.opensignal.com/networks`.

[OR11] Melih Onus and Andréa W Richa. Minimum maximum-degree publish–subscribe overlay network design. *IEEE/ACM Transactions on Networking*, 19(5):1331–1343, 2011.

[OYZ$^+$18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[PAA$^+$16] Larry L. Peterson, Ali Al-Shabibi, Tom Anshutz, Scott Baker, Andy C. Bavier, Saurav Das, Jonathan Hart, Guru Palukar, and William Snow. Central office re-architected as a data center. *IEEE Communications Magazine*, 54(10):96–101, 2016.

[Pal21] Jacob Palecek. Improving serverless edge computing for network bound workloads. Master's thesis, TU Wien, 2021.

[Pan18] Kasey Panetta. 5 trends emerge in the Gartner hype cycle for emerging technologies, 2018. 2018.

[PB02] Peter R Pietzuch and Jean M Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618. IEEE, 2002.

[PDCB15] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. A framework and algorithm for energy efficient container consolidation in cloud data centers. *Proceedings - 2015 IEEE International Conference on Data Science and Data Intensive Systems; 8th IEEE International Conference Cyber, Physical and Social Computing; 11th IEEE International Conference on Green Computing and Communications and 8th IEEE Inte*, pages 368–375, 2015.

[PDT18] G. Premsankar, M. Di Francesco, and T. Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275–1284, 2018.

[PG07] Cassio Pennachin and Ben Goertzel. *Contemporary Approaches to Artificial General Intelligence*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[PSK$^+$20] Reinhold Preiner, Johanna Schmidt, Katharina Krösl, Tobias Schreck, and Gabriel Mistelbauer. Augmenting node-link diagrams with topographic attribute maps. In *Computer Graphics Forum*, volume 39, pages 369–381. Wiley Online Library, 2020.

[PVCM20] David Perez Abreu, Karima Velasquez, Marilia Curado, and Edmundo Monteiro. A comparative analysis of simulators for the cloud to fog continuum. *Simulation Modelling Practice and Theory*, 101:102029, 2020. Modeling and Simulation of Fog Computing.

[PY$^+$10] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[RAD18] Thomas Rausch, Cosmin Avasalcai, and Schahram Dustdar. Portable energy-aware cluster-based edge computers. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC'18, pages 260–272, 2018.

[Rai21] Philipp Raith. Container scheduling on heterogeneous clusters using machine learning-based workload characterization. Master's thesis, TU Wien, 2021.

[Ras20] Alexander Rashed. Optimized container scheduling for serverless edge computing. Master's thesis, TU Wien, 2020.

[Rau17] Thomas Rausch. Message-oriented middleware for edge computing applications. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*, pages 3–4, 2017.

[Rau20a] Thomas Rausch. Ether: edge topology synthesizer. Source code repository. `https://github.com/edgerun/ether`, 2020.

[Rau20b] Thomas Rausch. Galileo: a framework for distributed load testing experiments. Source code repository. `https://github.com/edgerun/galileo`, 2020.

[RD19] Thomas Rausch and Schahram Dustdar. Edge intelligence: The convergence of humans, things, and AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019.

[RDR18] Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic message-oriented middleware for the internet of things. *IEEE Cloud Computing*, 5(2):17–25, 2018.

[RET14] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[RHLS17] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355. IEEE, 2017.

[RHM$^+$19] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'19, 2019.

[RHM20a] Thomas Rausch, Waldemar Hummer, and Vinod Muthusamy. PipeSim: Trace-driven simulation of large-scale AI operations platforms. *arXiv preprint arXiv:2006.12587*, 2020.

[RHM20b] Thomas Rausch, Waldermar Hummer, and Vinod Muthusamy. An experimentation and analytics framework for large-scale AI operations platforms. In *2020 USENIX Conference on Operational Machine Learning*, OpML'20, 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)*, 2021.

[RLF+20] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *3rd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'20, 2020.

[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'18, pages 191–197. IEEE, 2018.

[RR20] Alexander Rashed and Thomas Rausch. Execution traces of an MNIST workflow on a serverless edge testbed, 2020.

[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.

[RRF+20] Gianluca Rizzo, Sasko Ristov, Thomas Fahringer, Marjan Gusev, Matija Dzanko, Ivana Bilic, Christian Esposito, and Torsten Braun. Emergency networks for post-disaster scenarios. *Guide to Disaster-Resilient Communication Networks*, pages 271–298, 2020.

[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*, SEC'19, pages 307–309, 2019.

[RRR20] Thomas Rausch, Philipp Raith, and Alexander Rashed. faas-sim: a framework for trace-driven simulation of serverless function-as-a-service platforms. Source code repository. `https://github.com/edgerun/faas-sim`, 2020.

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[Sat04] M. Satyanarayanan. From the editor in chief: Augmenting cognition. *IEEE Pervasive Computing*, 3(2):4–5, 2004.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(June):30–39, Jan 2017.

[SBCD09] M Satyanarayanan, P Bahl, R Caceres, and N Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[Sca14] ScalAgent. JoramMQ, a distributed MQTT broker for the internet of things, 2014.

[Sch19] David Schneiderbauer. Predictive analytics for smart homes using serverless edge computing. Master's thesis, TU Wien, 2019.

[SCY18] Yuvraj Sahni, Jiannong Cao, and Lei Yang. Data-aware task allocation for achieving low latency in collaborative edge computing. *IEEE Internet of Things Journal*, 6(2):3512–3524, 2018.

[SD16] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.

[SGL19] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 45–50, 2019.

[SHM$^+$16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[SHS$^+$18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SKAEMW13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[SLF$^+$20] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[SNS$^+$17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, 2017.

[SNSD17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards QoS-aware fog service placement. In *2017 IEEE 1st international conference on Fog and Edge Computing*, ICFEC'17, pages 89–96. IEEE, 2017.

[SOE17] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. EdgeCloudSim: An environment for performance evaluation of edge computing systems. In *2017 Second International Conference on Fog and Mobile Edge Computing*, FMEC'17, pages 39–44. IEEE, 2017.

[SSBL16] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource provisioning for IoT services in the fog. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 32–39. IEEE, 2016.

[SSS$^+$17] Pablo Sotres, Juan Ramón Santana, Luis Sánchez, Jorge Lanza, and Luis Muñoz. Practical lessons from the deployment and management of a smart city internet-of-things infrastructure: The smartsantander testbed case. *IEEE Access*, 5:14309–14322, 2017.

[SSX$^+$15] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[Sta19] Robert Stackowiak. *Azure IoT Hub*, pages 73–85. Apress, Berkeley, CA, 2019.

[STM15] G Siegemund, V Turau, and K Maâmra. A self-stabilizing publish/subscribe middleware for wireless sensor networks. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–8, Mar 2015.

[SVDI16] J. M. Schleicher, M. Vögler, S. Dustdar, and C. Inzinger. Enabling a smart city application ecosystem: Requirements and architectural aspects. *IEEE Internet Computing*, 20(2):58–65, 2016.

[SVK$^+$17] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 535–546. IEEE, 2017.

[SXP$^+$13] Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, and Mahadev Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, page 139–152, New York, NY, USA, 2013. Association for Computing Machinery.

[SYY$^+$17] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue. An approach to QoS-based task distribution in edge computing networks for IoT applications. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 32–39, June 2017.

[Tel19] Telefonica. Telefonica open access and edge computing. White paper, February 2019.

[TGJ+02] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Network topology generators: Degree-based vs. structural. *ACM SIGCOMM Computer Communication Review*, 32(4):147–159, 2002.

[THLL17] Haisheng Tan, Zhenhua Han, Xiang-Yang Li, and Francis CM Lau. Online job dispatching and scheduling in edge-clouds. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[TLG16] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[TS17] V. Turau and G. Siegemund. Scalable routing for topic-based publish/-subscribe systems under fluctuations. In *2017 IEEE 37th International Conference on Distributed Computing Systems*, ICDCS'17, pages 1608–1617, June 2017.

[UWH+15] Rahul Urgaonkar, Shiqiang Wang, Ting He, Murtaza Zafer, Kevin Chan, and Kin K Leung. Dynamic service migration and workload scheduling in edge-clouds. *Performance Evaluation*, 91:205–228, 2015.

[VFD+16] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[VGGK18] Srikumar Venugopal, Michele Gazzetti, Yiannis Gkoufas, and Kostas Katrinis. Shadow puppets: Cloud-level accurate AI inference at the speed and economy of edge. In *USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'18, 2018.

[VN] Steven J. Vaughan-Nichols. Canonical's cloud-in-a-box: The ubuntu orange box. ZDNet. Online. `https://zdnet.com/article/canonicals-cloud-in-a-box-the-ubuntu-orange-box`.

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[VSDTD12] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36. ACM, 2012.

[VSI$^+$15] M. Vögler, J. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar. LEONORE – large-scale provisioning of resource-constrained IoT deployments. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 78–87, 2015.

[Wax88] B. M. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, Dec 1988.

[Wei16] Joe Weinman. Hybrid cloud economics. *IEEE Cloud Computing*, 3(1):18–22, 2016.

[WFC$^+$19] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa Anna George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Edge-based live video analytics for drones. *IEEE Internet Computing*, 23(4):27–34, 2019.

[WFG$^+$19] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 152–165. ACM, 2019.

[WSS05] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: A lightweight network location service without virtual coordinates. *ACM SIGCOMM Computer Communication Review*, 35(4):85–96, 2005.

[WXZL11] Jun Wu, Xin Xu, Pengcheng Zhang, and Chunming Liu. A novel multi-agent reinforcement learning approach for job scheduling in grid computing. *Future Generation Computer Systems*, 27(5):430 – 439, 2011.

[WZC$^+$18] Tsui-Wei Weng, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, Dong Su, Yupeng Gao, Cho-Jui Hsieh, and Luca Daniel. Evaluating the robustness of neural networks: An extreme value theory approach. *arXiv e-prints*, page arXiv:1801.10578, 2018.

[XE16] Xuan-Qui Pham and Eui-Nam Huh. Towards task scheduling in a cloud-fog computing system. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–4, Oct 2016.

[XLD$^+$19] Xiaolong Xu, Daoming Li, Zhonghui Dai, Shancang Li, and Xuening Chen. A heuristic offloading method for deep learning edge services in 5G networks. *IEEE Access*, 7:67734–67744, 2019.

[XSXH18] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

[YHZ$^+$17] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[YMLL17] Jihong Yan, Yue Meng, Lei Lu, and Lin Li. Industrial big data in an industry 4.0 environment: Challenges, schemes, and applications for predictive maintenance. *IEEE Access*, 5:23484–23491, 2017.

[YPI$^+$18] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. QoS-aware dynamic fog service provisioning. *arXiv preprint arXiv:1802.00800*, 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.

[ZIH$^+$13] J. Zhang, B. Iannucci, M. Hennessy, K. Gopal, S. Xiao, S. Kumar, D. Pfeffer, B. Aljedia, Y. Ren, M. Griss, S. Rosenberg, J. Cao, and A. Rowe. 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, 2013.

[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.

[ZWT12] B. Zhou, X. Wang, and X. Tang. Understanding collective crowd behaviors: Learning a mixture model of dynamic pedestrian-agents. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2871–2878, 2012.

Curriculum Vitae

Thomas Rausch
thomas@thrau.at
`https://thrau.at`
Born 27.10.1986 in Vienna, Austria.
CV date: 15.02.2021

Education

Ph.D., Computer Science, TU Wien, ongoing
Supervised by Schahram Dustdar at the Distributed Systems Group.
Topical focus: Distributed systems, systems for AI, edge computing, Internet of Things.
Thesis title: "A Distributed Compute Fabric for Edge Intelligence".

M.Sc., Software Engineering & Internet Computing, TU Wien, 2016
Module focus: programming languages, software engineering, information systems, and distributed systems and networking.
Thesis title: "Build Failure Prediction in Continuous Integration Workflows".
Graduated with distinction and a GPA of 1.1 (1 best, 5 worst).

B.Sc., Software & Information Engineering, TU Wien, 2014
Thesis title: "Efficient Querying of Versioned Tool Data in Data Integration Environments"

Secondary, Höhere Graphische Bundes-Lehr- und Versuchsanstalt, 2006
Specialization: Print and media technology.
Working with a diverse range of print, multimedia, and web technologies
In 2015, following professional experience, awarded the title Ingenieur (Ing.)

Experience

Founder & Technical Lead, Cognitive XR
Nov 2020 - ongoing
Vienna, Austria

Academic spin-off at the intersection of edge computing, augmented reality, and AI, building an any-scale platform for wearable cognitive augmentation applications.

University Assistant, TU Wien
Oct 2016 - May 2021 (4 yrs 7 mos)
Vienna, Austria

University Assistant at the Distributed Systems Group of the Institute for Information Systems Engineering.
Research on distributed systems, edge computing, AI systems, message-oriented middleware, container scheduling techniques. Cloud/Edge systems engineering in the CPS/IoT ecosystem project. Teaching the courses Distributed Systems, Distributed Systems Technologies and Distributed Systems Engineering. Member of the Scientific Staff Council.

Visiting Researcher, Carnegie Mellon University
Jul 2019 – Sep 2019 (3 mos)
Pittsburgh, PA, USA

Working on edge computing systems with Professor Mahadev Satyanarayanan and his lab. Demo at SEC'19.

Graduate Research Intern, IBM Research
Jun 2018 – Sep 2018 (4 mos)
New York, NY, USA

Graduate research intern working on IBM's next-generation AI platform, with a focus on automating AI application operations workflows. Resulted in a publication at USENIX OpML'20.

Software Engineer, Christian Doppler Laboratory "Software Engineering Integration for Flexible Automation Systems"
Jun 2013 – Jun 2016 (3 yrs 1 mo)
Vienna, Austria

Design and implementation of a data integration platform for systems engineering tools.

Teaching Assistant, TU Wien
Apr 2013 – Mar 2016 (3 yrs)
Vienna, Austria

Teaching Assistant for the courses: Advanced Internet Computing, Software Testing, Software Engineering & Project Management. Semestral Git introduction lecture.

Full-Stack Web Developer, Freelance
Oct 2010 – Mar 2015 (4 yrs 6 mos)
Vienna, Austria

Lead developer on a large multi-website CMS and content sharing platform.

Teaching

Courses at TU Wien

- 184.167 Distributed Systems UE, 3.0 ECTS
- 184.153 Distributed Systems Engineering VU, 3.0 ECTS
- 184.260 Distributed Systems Technologies VU, 6.0 ECTS
- 184.715 Project in Software Engineering & Internet Computing PR, 12.0 ECTS
- 194.058 Project in Computer Science 1 PR, 6.0 ECTS
- 194.059 Project in Computer Science 2 PR, 6.0 ECTS
- 184.194 Seminar in Distributed Systems SE, 3.0 ECTS

Supervised theses

I've served as assistant supervisor on the following theses:

- Jacob Palecek. Improving Serverless Edge Computing for Network Bound Workloads. Master's Thesis, in progress (Winner of a 2020 Netidee grant).
- Cynthia Marcelino. Locality-Aware Data Management for Serverless Edge Computing. Master's Thesis, in progress.
- Andreas Bruckner. Self-adaptive Distributed MQTT Middleware for Edge Computing Applications. Master's Thesis, in progress.
- Dominik Schubert. Proactive Autoscaling for a Coding Assignment Submission System. Bachelor's Thesis, in progress.
- Daniel Rohr. Automated Error Cause Detection for Programming Assignments Using Machine Learning. Bachelor's Thesis, in progress.
- Philipp Raith. Container Scheduling on Heterogeneous Clusters using Machine Learning-based Workload Characterization. Master's Thesis, TU Wien, 2021 (Winner of a 2020 Netidee grant).
- Silvio Vasiljevic. Energy-Efficient Load-Balancing in Portable Edge Clusters. Bachelor's Thesis, TU Wien, 2020.
- Alexander Rashed. Optimized Container Scheduling for Serverless Edge Computing. Master's Thesis, TU Wien, 2020.
- Valentin Bauer. Hangar: Decentralized Storage and Provisioning of System Environments. Bachelor's Thesis, TU Wien, 2019.

- Ammar Voloder. Using In-Memory Key-Value Stores as a Transport Mechanism for RPC. Bachelor's Thesis, TU Wien, 2019.
- David Schneiderbauer. Predictive Analytics for Smart Homes using Serverless Edge Computing. Master's Thesis, TU Wien, 2019.
- Jacob Palecek. A Monitoring Infrastructure for Energy-Aware Cluster-Based Mobile Cloudlets, Bachelor's Thesis, TU Wien, 2019.
- Manuel Geier. Enabling Mobility and Message Delivery Guarantees in Distributed MQTT Networks. Master's Thesis, TU Wien, 2019.
- Philipp Raith. Performance Evaluation of Java IO Models. Bachelor's Thesis, TU Wien, 2018.

Awards and honors

- Finalist at the 2020 TU Wien Best Teaching Awards in the category "Best Teacher of the Faculty of Informatics".
- SEC'19 best demo award for "A system for operating energy-aware cloudlets".
- Middleware'17 travel grant recipient.

Academic activities

Technical program committees

- 9th IEEE International Conference on Cloud Engineering (IC2E'21)
- 2nd IEEE International Conference on Joint Cloud Computing (JCC'21)
- 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)

Conference organization committees

- 6th ACM/IEEE Symposium on Edge Computing (SEC'21): Publicity Co-Chair
- 9th IEEE International Conference on Cloud Engineering (IC2E'21): Poster Chair
- 5th ACM/IEEE Symposium on Edge Computing (SEC'20): Publicity Co-Chair

Reviewer

I have been an official reviewer for:

- Springer Artificial Intelligence Review
- IEEE Transactions on Cloud Computing
- IEEE Computer
- Springer Computing
- Future Generation Computer Systems
- Spring Journal of Grid Computing
- Informatik-Spektrum
- IEEE Internet Computing
- ACM Transactions on Internet Technology
- IEEE Transactions on Network and Service Management
- IEEE Transactions on Services Computing
- IEEE Transactions on Vehicular Technology

I have also served as sub-reviewer for:

- IEEE International Conference on Distributed Computing Systems (ICDCS)
- International Conference on Software Engineering (ICSE)
- International World Wide Web Conference (TheWebConf)
- IEEE Software
- ACM Computing Surveys
- IEEE International Conference on Cloud Computing (CLOUD)
- USENIX Workshop on Hot Topics in Edge Computing (HotEdge)
- IEEE International Conference on Cloud Engineering (IC2E)
- IEEE International Conference on Service-Oriented System Engineering (SOSE)
- IEEE International Conference on Big Data Service and Applications (BigDataService)
- IEEE International Conference on Cloud Computing Technology and Science (CloudCom)
- IEEE Global Communications Conference (GLOBECOM)
- IEEE International Conference on Fog Computing (ICFC)

Publications

Journal articles

- Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021
- Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Schahram Dustdar, Ognjen Scekic, Thomas Rausch, Stefan Nastic, Sasko Ristov, and Thomas Fahringer. A deviceless edge computing approach for streaming IoT applications. *IEEE Internet Computing*, 23(1):37–45, 2019
- Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic message-oriented middleware for the internet of things. *IEEE Cloud Computing*, 5(2):17–25, 2018
- S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017

Conference and workshop papers

- 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)*, 2021
- Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *3rd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'20, 2020
- Thomas Rausch, Waldermar Hummer, and Vinod Muthusamy. An experimentation and analytics framework for large-scale AI operations platforms. In *2020 USENIX Conference on Operational Machine Learning*, OpML'20, 2020
- 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*, SEC'19, pages 307–309, 2019
- Haris Isakovic, Vanja Bisanovic, Bernhard Wally, Thomas Rausch, Denise Ratasich, Schahram Dustdar, Gerti Kappel, and Radu Grosu. Sensyml: Simulation environment for large-scale IoT applications. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 3024–3030. IEEE, 2019
- Clemens Lachner, Thomas Rausch, and Schahram Dustdar. ORIOT: A source location privacy system for resource constrained IoT devices. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019

- Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, and Kaoutar El Maghraoui. Modelops: Cloud-based lifecycle management for reliable and trusted AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019
- Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'19, 2019
- Thomas Rausch and Schahram Dustdar. Edge intelligence: The convergence of humans, things, and AI. In *2019 IEEE International Conference on Cloud Engineering*, IC2E'19, Jun 2019
- Clemens Lachner, Thomas Rausch, and Schahram Dustdar. Context-aware enforcement of privacy policies in edge computing. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 1–6. IEEE, 2019
- 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'18, pages 191–197. IEEE, 2018
- Thomas Rausch, Cosmin Avasalcai, and Schahram Dustdar. Portable energy-aware cluster-based edge computers. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC'18, pages 260–272, 2018
- Haris Isakovic, Denise Ratasich, Christian Hirsch, Michael Platzer, Bernhard Wally, Thomas Rausch, Dejan Nickovic, Willibald Krenn, Gerti Kappel, Schahram Dustdar, et al. CPS/IoT Ecosystem: A platform for research and education. In *Cyber Physical Systems. Model-Based Design*, pages 206–213. Springer, 2018
- Thomas Rausch. Message-oriented middleware for edge computing applications. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*, pages 3–4, 2017
- Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355. IEEE, 2017

Tech reports

- Thomas Rausch, Waldemar Hummer, and Vinod Muthusamy. PipeSim: Trace-driven simulation of large-scale AI operations platforms. *arXiv preprint arXiv:2006.12587*, 2020