TU WIEN Informatics

Optimized Container Scheduling for Serverless Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Rashed, BSc
Matrikelnummer 01325897

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Ing. Dipl.-Ing. Thomas Rausch, BSc

Wien, 11. Dezember 2019

Alexander Rashed	Schahram Dustdar

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

TU WIEN Informatics

Optimized Container Scheduling for Serverless Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Alexander Rashed, BSc
Registration Number 01325897

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Ing. Dipl.-Ing. Thomas Rausch, BSc

Vienna, 11th December, 2019

Alexander Rashed	Schahram Dustdar

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

Erklärung zur Verfassung der Arbeit

Alexander Rashed, 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, 11. Dezember 2019

Alexander Rashed

Acknowledgements

First and foremost, I want to thank my advisors Schahram Dustdar and Thomas Rausch. I'm particularly grateful for Thomas' continued support, all the fruitful discussions and meetings, and the invaluable feedback I received during this journey.

Furthermore, I want to thank my family and friends. They support me, wherever they can. They distract me, whenever it's necessary. And they sharpen my focus, when I'm loosing track.

A special thanks to my mother for giving me the book "Delphi for Kids" when I was 12, which probably shaped my path towards the awesome field of computer science.

Finally, I would like to express my gratitude to Carina, for her caring, her patience, and her endless support.

Kurzfassung

Edge Computing ist ein neues Paradigma, welches heterogene Rechen- und Speicherressourcen am Netzwerkrand in unmittelbarer Nähe zu den eigentlichen Datenproduzenten wie Sensoren und mobilen Geräten ermöglicht. Aufgrund ihrer Heterogenität leiden Edge-Computing-Umgebungen unter hoher Komplexität. Diese Komplexität kann jedoch durch den Einsatz von Serverless Computing reduziert werden, einem neuen Cloud Computing-Ausführungsmodell, bei dem die Infrastrukturkomponenten für den Anwendungsentwickler völlig transparent sind. Darüber hinaus passt der ereignisgesteuerte Charakter des Internet der Dinge – das eines der Hauptanwendungsgebiete von Edge Computing sein wird – perfekt zum ereignisgesteuerten Charakter von Function-as-a-Service, einer speziellen Implementierung von Serverless Computing.

Die meisten Function-as-a-Service Plattformen verwenden containerbasierte Virtualisierung. Um Container im großen Maßstab in einem Cluster zu integrieren und zu verwalten wurden mehrere Container-Orchestrierungsplattformen eingeführt, wobei Kubernetes zum De-Facto-Standard wurde. Der Kubernetes-Scheduler ist zwar flexibel, basiert aber auf der Grundannahme, dass die Cluster-Infrastruktur sehr homogen ist.

Diese Arbeit beschreibt das Design, die Implementierung und die Evaluierung eines integrierten, latenz- und fähigkeitsbewussten Schedulers für den Betrieb einer Function-as-a-Service Plattform in einer gemischten Cloud-Edge-Computerumgebung. Wir stellen den Skippy-Scheduler vor, der den standardmäßigen Kubernetes-Scheduler um domänenspezifische Prioritätsfunktionen erweitert, um die Platzierungsqualität für Funktionen in gemischten Cloud-Edge-Clustern zu erhöhen. Diese domänenspezifischen Prioritätsfunktionen verwenden zusätzliche Metadaten der Funktionen sowie der Knoten im Cluster. Um das Sammeln dieser Metadaten der Knoten zu automatisieren, wird der Skippy-Daemon eingeführt. Da die Konfiguration der einzelnen Gewichte der Prioritätsfunktionen nicht trivial ist, implementieren wir einen Optimierungsansatz.

Unsere Ergebnisse zeigen, dass unser Scheduler den standardmäßigen Kubernetes-Scheduler im Durchschnitt über alle getesteten Szenarien übertrifft. Darüber hinaus erhöht die Optimierung die Platzierungsqualität zusätzlich deutlich. Im Vergleich zum standardmäßigen Kubernetes-Scheduler reduziert unser optimierter Scheduler die Bandbreitenauslastung um 67,52%, senkt die Kosten um 100%, erhöht die Ressourcenauslastung der Edge-Geräte um 245,37% und reduziert die Task-Ausführungszeit um 71,18% im Durchschnitt über alle getesteten Szenarien.

Abstract

Edge computing is a new paradigm which enables heterogeneous computing and storage resources located at the network edge, in close proximity to the actual data producers like sensors and mobile devices. Due to their heterogeneous nature, edge computing environments are suffering from high complexity. However, this complexity can be decreased by leveraging serverless computing, a new cloud computing execution model in which the infrastructure components are entirely transparent to the application developer. Additionally, the event-centric nature of the Internet of Things – which will be one of the main application fields of edge computing – fits perfectly to the event-driven nature of Function-as-a-Service, a specific implementation of serverless computing.

Most Function-as-a-Service platforms are utilizing container-based virtualization. In order to integrate and manage containers at scale in a cluster, multiple container orchestration platforms have been introduced, with Kubernetes becoming the de-facto standard. While the Kubernetes scheduler is flexible, it is based on the basic assumption that the cluster infrastructure is highly homogeneous.

This thesis describes the design, implementation, and evaluation of an integrated, latency-, and capability-aware scheduler for running a Function-as-a-Service platform in a mixed cloud-edge computing environment. We introduce the Skippy scheduler, which extends the default Kubernetes scheduler with domain-specific priority functions in order to increase the placement quality for functions in mixed cloud-edge clusters. Those domain-specific priority functions are utilizing additional metadata of the functions as well as of the nodes in the cluster. In order to automate the gathering of this metadata of nodes, the Skippy daemon is introduced. Since the configuration of the individual weights of the priority functions isn't trivial, we implement an optimization approach.

Our results show that our scheduler outperforms the default Kubernetes scheduler on average across all tested scenarios. Furthermore, the optimization additionally increases its placement quality significantly. Compared to the default Kubernetes scheduler, our optimized scheduler decreases the bandwidth usage by 67.52%, decreases the cost by 100%, increases the edge device resource utilization by 245.37%, and decreases the task execution time by 71.18% on average across all tested scenarios.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Edge computing refers to a new paradigm in which computing and storage resources located at the network edge, in close proximity to sensors or mobile devices, facilitate the computation and storage of data close to the actual data source [Sat17; SD16]. The proximity of these resources brings many benefits. Due to the reduced physical distance to these nodes the latency can be decreased, scalability and privacy can be increased by processing data right at the edge, cloud outages can be masked by failover-services thus increasing resiliance, and already available on-site computing or storage resources can be better utilized [Sat17; Ska+16]. With the rise of the Internet of Things (IoT), edge computing will take a key role in the infrastructure, delivering a whole new category of emerging services and applications [SD16; Bon+12; Bon+14]. The seamless orchestration and operation of such an infrastructure is hard due to the heterogeneity of the involved devices and their network connectivity. Accordingly there is no established solution available yet.

Serverless computing, and more specifically *Function-as-a-Service (FaaS)*, is a new cloud computing execution model in which the infrastructure components are entirely transparent to the application developer. Considering the traditional cloud service models – Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) – FaaS can be seen as the logical further development of these models towards a cloud-native environment [Fox+17; Bal+17a]. As the developer only submits stateless functions which are triggered by events, FaaS facilitates application structures similar to functional reactive programming [Cas+17]. This model provides seamless and (near) endless scalability of the application as well as the costs (pay-per-invocation model) without any effort by the application developer. The first commercial FaaS platform – Amazon's AWS Lambda – was introduced in November 2014. FaaS is gaining traction ever since but also still comes with some restrictions [Lei+19; Hel+18]. Especially when it comes to function

composition and the data exchange between these functions, developers are sometimes facing tedious limitations [Lei+19; Bal+17b].

1.2 Problem Statement

Until now, not much effort has been made to combine these emerging two models by bringing FaaS platforms to the edge of the network. However the data-centric event-driven nature of the IoT fits well to the event-driven paradigm inherent in FaaS platforms. Furthermore, the close proximity of producers (IoT devices) and consumers (functions) would be beneficial for the latency of such applications.

Scheduling techniques for FaaS in edge environments need to go beyond the current established methods used in cloud computing, which generally assume a homogeneous infrastructure within the bounds of a data center. In particular, such scheduling techniques need to consider the proximity of functions to their data sources and data stores; or the specific resource capabilities of the available edge nodes, in particular when scheduling functions that could leverage application specific hardware, such as machine learning workloads on machine learning accelerators.

From a platform provider's perspective, the different, possibly conflicting, objectives of the scheduler – such as the resource utilization of the involved edge devices versus the overall task execution time – may be prioritized differently for each tenant. Therefore it is necessary to configure the scheduling algorithm based on these objectives.

1.3 Solution Approach

The aim of this thesis is to design, develop, and evaluate an integrated, optimized, latency- and capability-aware scheduler for running an FaaS platform in a mixed cloud-edge computing environment.

Based on the current state of open-source FaaS platforms and their underlying orchestration systems, a set of extensions needs to be developed and integrated in a pre-selected open-source technology stack.

Current FaaS platforms focus on homogeneous cloud environments only. They simply delegate the need for a specific function or runtime deployment to the orchestration system in place (which in many cases is exchangeable) without defining any constraints or preferences. In order to allow the orchestration system a (near-)optimal placement of these newly scheduled function deployments in a mixed environment, it is necessary to handover additional metadata from the FaaS platform to the orchestration system. This would give its scheduler the possibility to respect this metadata during the selection of the node to deploy the function or runtime onto.

To overcome this lack of metadata, additional means of configuration have to be integrated. First of all, it is necessary to handover the previously mentioned additional information

about the functions from the FaaS platform to the orchestration system. Secondly, the structure of the network itself needs to be described for the orchestration system. Thirdly, the metadata about the nodes in the cluster – especially their specific resource capabilities – needs to be collected and maintained automatically in order to provide a scalable and easily maintainable platform. This information will then be used by a specific scheduler implementation within the orchestration system to calculate the best deployment configuration and finally schedule the function or runtime deployment on the selected node.

In order to find optimal scheduler configurations for the objective prioritization it is necessary to utilize multi-objective optimization techniques. This optimized configuration will then allow the scheduler to maximize the placement quality considering these individual, pre-selected objectives and their specific trade-offs.

In this mixed cloud-edge computing environment, our scheduler places functions in closer range to the container image registry and used data stores (from a network perspective), and explicitly favors edge nodes or nodes with specific hardware accelerators supported by the function, thus significantly increasing the placement quality.

1.4 Structure

The remainder of this thesis is structured as follows. Initially, Chapter 2 describes the fundamentals of this work. We give an overview of edge computing and serverless computing, describe the characteristics of scheduling, outline container orchestration, introduce the multi-objective optimization problem, and characterize machine learning workflows. Chapter 3 shows state-of-the-art research and other related work relevant in the context of this thesis. It focuses on the intersections of serverless computing, container orchestration, and scheduling with edge computing respectively. Chapter 4 explains the methodology of this work. It justifies the selection of specific technologies, outlines a motivational scenario, and describes the methodology behind our empirical measurements. Furthermore, it specifies the schema of our metadata and briefly outlines the implemented scheduler, its simulation, and finally its optimization. Chapter 5 elucidates the main contribution of this work. First, it explains the default scheduler we are competing with. Then, our specific enhancements are listed and described in-depth. In addition, this chapter also covers the specification and implementation of our simulation environment. Chapter 6 shows the implementation of our optimization efforts. The results of the evaluation of all our contributions are presented in Chapter 7. Finally, Chapter 8 concludes this work and outlines possible future research.

CHAPTER 2

Fundamentals

This chapter provides an overview of the fundamental concepts and technologies used in this thesis. Since this thesis aims on creating a scheduler specifically for serverless edge computing, Section 2.1 outlines edge computing, its benefits and challenges. Section 2.2 then covers serverless computing in general and Function-as-a-Service in specific. The process of scheduling itself is described in Section 2.3. Our scheduler is focused on containers. Therefore, our scheduler will be operating within the boundaries of a container orchestration system. Section 2.4 explains container orchestration and its underlying technologies. The configuration and optimization of our scheduler is hard, as it directly influences the placement decisions and therefore the performance of the scheduler regarding different conflicting objectives. Section 2.5 gives a description, as well as a mathematical definition of the multi-objective optimization problem and its terminology, forming the basis of our optimization efforts. Finally, Section 2.6 characterizes machine learning workflows, which are essential for our motivational scenario.

2.1 Edge Computing

Cloud computing has had a major impact on software engineering and operations. Organizations can avoid creating and maintaining their own data center by renting large amounts of computing and storage resources on demand from cloud service providers [Arm+10]. Instead of a big up-front commitment, resources are paid on a short-term basis and only for the time of their allocation. Developers no longer have to manage physical infrastructure, but the management of virtual resources has proliferated. Due to the economics of scale the overall costs were significantly reduced as very large data centers were built by big service providers [Jon+19]. This trend caused a major consolidation of the global computing capacity into a set of large data centers which are spread across the globe and operated by those service providers [Sat17].

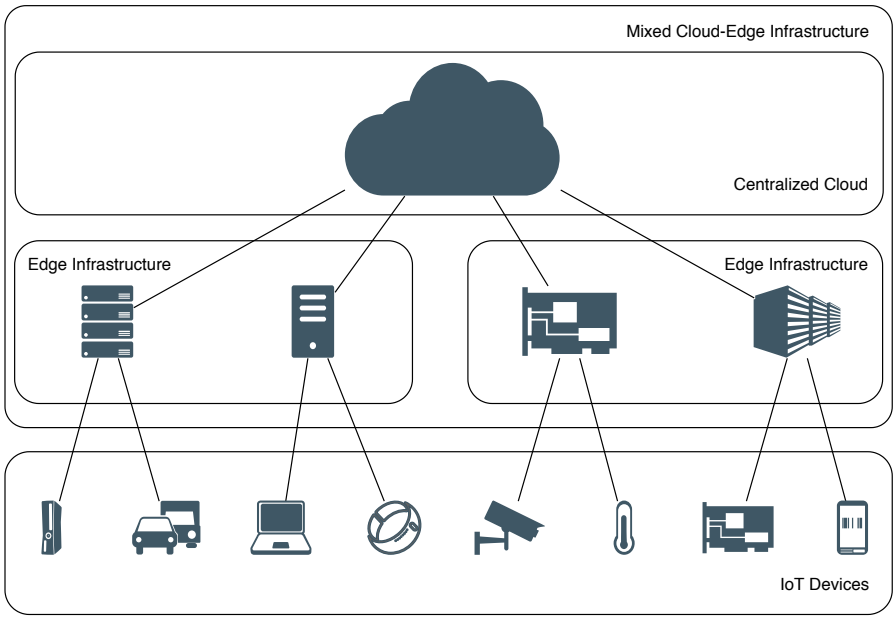


Figure 2.1: A mixed cloud-edge infrastructure

Traditionally, the devices at the network edge were mostly consuming data. But with the emergence of the IoT and the continuous evolution of mobile devices and applications these devices' role is more and more shifting from data consumer to data producer [SD16]. The Cisco Global Cloud Index predicts that the total amount of data created by any device will reach 847 zettabytes per year by 2021 [Cis18]. At the same time the annual global data center IP traffic is predicted to reach 20.6 zettabytes. These numbers clearly illustrate that the vast majority of the produced data needs to be processed outside of centralized data centers. There is an increasing need for the decentralization of those data centers for the sake of scalability.

Edge computing refers to a new paradigm which enables such a decentralization. Computing and storage resources located at the network edge, in close proximity to the actual data producers, like sensors and mobile devices, facilitate the processing and storage of data close to the actual data source [Sat+09; Sat17; SD16]. These devices are often referred to as cloudlets or fog/edge nodes[1]. Once these edge devices are installed they can be managed together with traditional cloud resources, forming a mixed cloud-edge infrastructure. Figure 2.1 illustrates the concept of such a mixed cloud-edge infrastructure.

[1]There are efforts for a distinction between the terms cloud-, fog-, and edge computing towards a three tier architecture [VR14; Bon+12; Bon+14]. We do not consider these subtle differences, instead we consider *edge computing* and *fog computing* as synonyms.

2.1.1 Benefits

Using those newly enabled resources in a mixed cloud-edge-infrastructure could yield many benefits [Sat17; SD16; Ska+16]. The following four are considered the most promising.

Reduced Latency

The increased physical proximity of a processing device or storage resource to a user's device may decrease the end-to-end latency as well as the requirements for bandwidth to the cloud. This becomes increasingly necessary for emerging low-latency applications like augmented reality or self-driving cars.

Increased Scalability

The vast majority of the generated data will not be processed in data centers. Instead edge devices will preprocess the data and either completely fulfill the user's request or only send the results of the preprocessing to the data center. As a result, the ingress bandwidth into the data center can be reduced by multiple orders of magnitude.

Increased Privacy

If executed in a systematic fashion the preprocessing of generated data on edge devices could also help enforcing privacy policies. For example, an edge device could anonymize sensitive health data before it is uploaded to the data center.

Increased Reliability

The increased decentralization may be used to harden the system's reliability. Outages in the data center or the intermediate infrastructure could be masked by providing fallback services on the edge devices.

2.1.2 Challenges

The heterogeneous and decentralized nature of edge computing resources introduces several new systems engineering challenges [SD16; Shi+16; Ska+16].

Device Heterogeneity

Previously unused available computational and storage resources can be enabled by registering them as edge devices in a mixed cloud-edge infrastructure. On the one hand this increases the utilization those otherwise unused resources. On the other hand, the heterogeneity of edge devices is challenging for application developers as the software needs to be built and partitioned in a way such that the application can run on different processor architectures and with different amounts of resources.

Programmability

Users of cloud computing services have none or at most partial knowledge of where and how the application is deployed and running. When creating a mixed cloud-edge infrastructure, a unifying platform is necessary to create a layer of abstraction over the heterogeneous and highly distributed edge device infrastructure and thereby allow the seamless development, orchestration, and operation of software systems in such an environment.

Privacy and Security

Edge computing could not only be beneficial for data privacy, it also brings some challenges thereof. The environment on the edge of the network is highly dynamic. This makes the network more vulnerable to threats. Also there is a lack of efficient tools to secure devices at the edge of the network.

Scheduling

The heterogeneity of the system also causes challenges when it comes to workload allocation. The network contains many different devices at different layers with different resources and capabilities. This makes the allocation of resources for a specific workload hard, as there are multiple conflicting objectives. With the creation of a scheduler specifically designed to operate in mixed cloud-edge computing infrastructures, and with the specific focus on serverless function workloads, in this thesis we aim to make advances in tackling this problem in our specific domain. Scheduling itself is described in detail in Section 2.3.

2.2 Serverless

In November 2014, Amazon released AWS Lambda[2], drawing large attention to serverless computing and FaaS in particular. Serverless computing is a new cloud computing execution model in which the infrastructure components are entirely transparent to the application developer, releasing them of the burden of managing all the virtual resources.

There are three distinctions between serverless and serverful computing [Jon+19]:

1. *Decoupled computation and storage:* The computation can be provisioned, scaled and priced independently from the storage. Because of this decoupling, the computation is necessarily stateless due to the degree that no ephemeral storage is used.

2. *Execute code without managing resource allocation:* The developer only provides a piece of code and the platform automatically manages the lifecycle and executions of the code, including the provisioning of the necessary cloud resources.

[2] `https://aws.amazon.com/lambda/` (visited on Nov. 26, 2019)

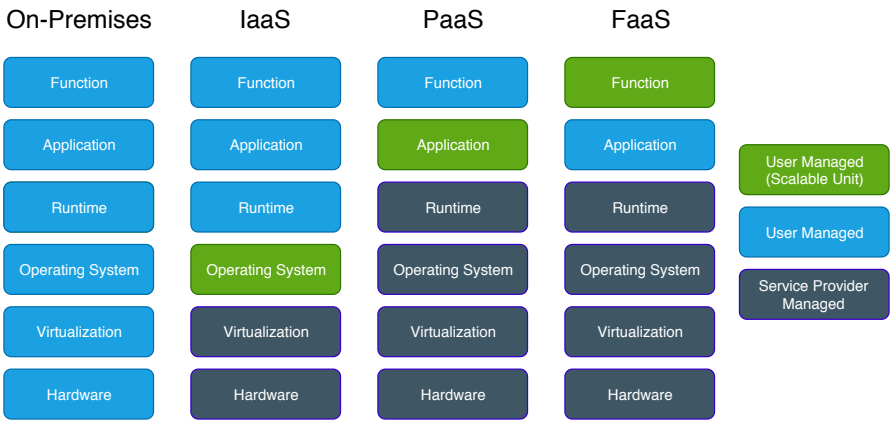


Figure 2.2: Comparison of cloud computing execution models

3. *Paying in proportion to resources used instead of for resources allocated:* The billing is not based on the size and number of virtual resources allocated, but is rather associated with the execution, usually the execution time.

While the terms *serverless computing* and *FaaS* are often used as synonyms, there is a subtle difference. FaaS only refers to the implementation of Item 2, being one of multiple services in a serverless offering alongside with e.g. storage and authentication solutions.

2.2.1 Function-as-a-Service

FaaS is the abstraction and generalization of cloud services towards the theory of traditional functional programming [Hel+18]. An application is a composition of multiple stateless functions. Each function is a mapping from inputs to outputs. FaaS platforms allow their users to register functions in the cloud, compose them into applications, and define execution triggers. The resulting application structures are similar to those in functional reactive programming [Cas+17].

Considering the traditional cloud service models – IaaS and PaaS – FaaS can be seen as the logical further development of these models towards a cloud-native environment [Fox+17; Bal+17a]. Figure 2.2 describes the difference between those three models and the traditional on-premises infrastructure.

It requires a different mental model when designing systems in a serverless fashion compared to traditional cloud computing [Lei+19], but serverless computing, and FaaS in particular, are being adopted heavily and are predicted to grow to dominate the future of cloud computing [Fox+17; Jon+19].

2.2.2 Benefits

This new execution model has many benefits compared to the traditional cloud computing offerings [Bal+17a; Hel+18; MB17].

Scalability

The decoupling of the storage from the computation together with the premise that the computation has to be stateless – or at least must not use ephemeral storage – allows (near) endless scalability as new instances of the functions can be spawned and killed without taking care of any state. Also the storage can be scaled in size and power independently of the functions. Functions can be scaled to zero if no executions are requested at the moment.

Pricing

As described in Section 2.2, the billing model is fundamentally different from traditional cloud computing. Since FaaS platform users are paying in proportion to the actual resource usage and functions scale to zero, the service is free for periods of time in which there are no triggers invoking any function executions. If there are executions, the price scales linearly with the execution time, based on fine-grained execution time measurements (usually in 100 milliseconds time slices).

No Resource Management

Developers no longer have to manage any resources like servers, virtual machines (VMs) or even containers. Instead they just focus on the implementation of the business logic functions and compose them to form the complete application.

Polyglot / Microservice Architecture

FaaS platforms usually support many different programming languages for its functions, like JavaScript, Go, Python, or Java. Some platforms, like OpenFaaS[3], even allow the creation of custom runtimes or submitting container images as functions. The fine granularity of those managed polyglot functions maps naturally to the microservice software architecture.

2.2.3 Challenges

The currently available serverless platforms are coming with several restrictions and challenges [Bal+17a; Lei+19; Hel+18; Bal+17b].

Vendor Lock-In

Currently each platform only supports its own Application Programming Interface (API). A function has to have a specific function signature, implement a given interface, or be reachable via a specific protocol in order to allow the platform to invoke the function. The platform may also provide a whole ecosystem of services like logging, authentication,

[3] `https://www.openfaas.com/` (visited on Nov. 26, 2019)

authorization, state management, or storage. Those factors introduce a high risk of vendor lock-in for each platform.

Limited Function Lifetimes

All major serverless providers do have hard function lifetime restrictions. For AWS Lambda a function may not run longer than 15 minutes. Afterwards the function is shut down by the infrastructure.

Cold Start

Functions can be scaled to zero, which introduces the problem of cold starts. If there are no running instances of a function available at the time of a trigger execution it takes some time to spawn a new runtime and trigger the function invocation.

Communication through Slow Storage

Functions are not directly addressable while running. As a result, composed lambda functions can only communicate through an intermediary storage service. Such storage systems, like S3, are significantly slower and more expensive than traditional point-to-point network communication.

No Specialized Hardware

As predicted by Hennessy and Patterson in the Turing Lecture 2018, the development of domain-specific hardware will be accelerated in the near future [HP18]. Traditional cloud offerings already allow the provisioning of specialized hardware to some degree, but there is no serverless product which allows the platform user to select a function's hardware accelerators yet.

2.2.4 Serverless at the Edge

Serverless computing was originally created for centralized cloud environments. But due to their heterogeneous nature, edge computing environments are suffering even more from high complexity and therefore could also highly benefit from the paradigm of serverless computing [GND17]. Content Delivery Network (CDN) operators are already providing services to run functions closer to the actual user [Jon+19]. As edge computing will focus on events being shared from IoT devices with the edge infrastructure, FaaS with its event-driven nature perfectly fits into this architecture [Fox+17]. Section 3.1 describes the current developments in the area of serverless edge computing in detail.

2.3 Scheduling

Scheduling is the process of decision-making on resource allocations to tasks over time periods. Its goal is to optimize towards different objectives [Pin12]. In a cluster-setting,

there are many different, in some cases conflicting, objectives when it comes to those resource allocations, for example:

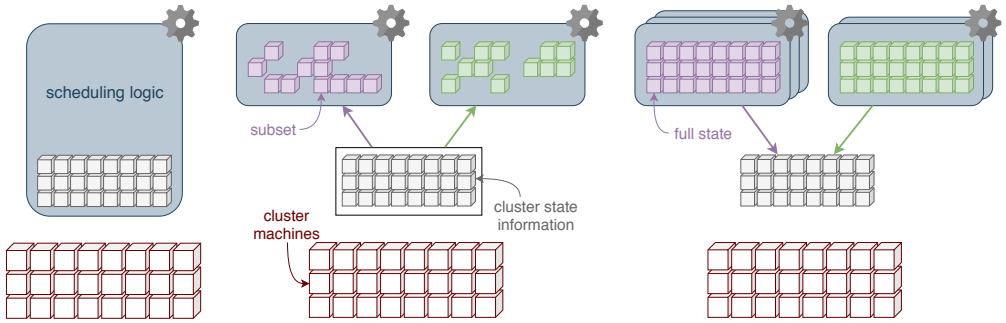
- *Resource utilization:* The available resources should be utilized as much as possible in order to execute as many tasks as possible without the need of adding additional resources.
- *Cost:* The costs of executing a task should be minimized. The costs depend on the resources they are allocated to as well as the time the tasks makes use of the resources (i.e. the task execution time).
- *Task execution time (TET):* The time it takes to execute a task should be minimized.
- *Bandwidth usage:* The bandwidth usage should be minimized in order to avoid unnecessary data transfer times.

While optimizing towards those objectives, the scheduler also needs to respect different constraints. The following are considered the most important:

- *Resource requirements:* Each workload may have different requirements on CPU or RAM. The scheduler needs to make sure that a node has enough free resources for the workload placed onto it.
- *Storage requirements:* If the workload needs to store data on the node it is running on, the scheduler needs to make sure that the necessary amount of storage is available.
- *Anti-Affinities:* For the sake of reliability and fault-tolerance, some workload may be configured not to run on the same node as another. The scheduler needs to make sure that different workload assigned to one node is not violating those configured anti-affinities.

Increasing the quality of task placements by the scheduler in a cluster can lead to increased fault tolerance, a decreased TET, and a more predictable system performance [Gog+16]. The costs of running the cluster can be decreased by minimizing the amount of resources necessary to execute the scheduled tasks as well as their execution time.

Achieving a high task placement quality means solving an algorithmically complex optimization problem [Gog+16]. In fact, the service placement problem has been shown to be $\mathcal{NP}$-complete [Ska+16]. Therefore it is infeasible to solve this problem for each placement while maintaining a low placement latency. This means that schedulers have to choose their focus, either a high placement quality by using sophisticated algorithms and solvers, or a low placement latency by using simple heuristics [Gog+16].



(a) Monolithic (b) Two-level (c) Shared state

Figure 2.3: Scheduling architectures according to Schwarzkopf et al. [Sch+13]

2.3.1 Scheduler Architectures

In order to cope with the complexity of the scheduling problem while maintaining a low placement latency, different scheduler architectures have been used over the years [Sch+13]. As shown in Figure 2.3, their essential difference is how they handle the cluster state information.

Monolithic Schedulers

Monolithic schedulers only consist of a single centralized component handling the scheduling of all incoming tasks. As shown in Figure 2.3a, there is no need for the synchronization or even exchange of any cluster state as it's only manipulated by this single component.

The default scheduler of Kubernetes is a prominent example of a monolithic scheduler. In addition to production ready solutions, there are academic developments working towards optimizing monolithic schedulers, like the Firmament scheduler [Gog+16].

Two-level Schedulers

Two-level schedulers have a single centralized coordinator which is actively managing the resources. It offers available compute resources to multiple (usually pluggable) schedulers which in turn are executed in parallel. Figure 2.3b illustrates that the coordinator is dividing the resources exclusively among the schedulers. This division of the available resources in disjunct sets completely mitigates any conflicts as no two schedulers are offered the same resource.

This approach is most prominently used in Apache Mesos [Hin+11].

Shared-state Schedulers

Shared-state schedulers are sharing the cluster state among the distributed scheduler without any coordinating party. Figure 2.3c shows that each instance of the scheduler

has a replica of the complete cluster state. Changes to the state, e.g. task placements, are done using lock-free optimistic concurrency control.

The scheduler of Omega, the predecessor of Google's Kubernetes, as well as Microsoft's Apollo have sucessfully been structured this way [Sch+13; Bou+14].

Hybrid Schedulers

In addition to these architectures, hybrid solutions have been proposed. Mercury, for example, allows offloading work from the centralized scheduler by using additional distributed schedulers [Kar+15].

2.4 Container Orchestration

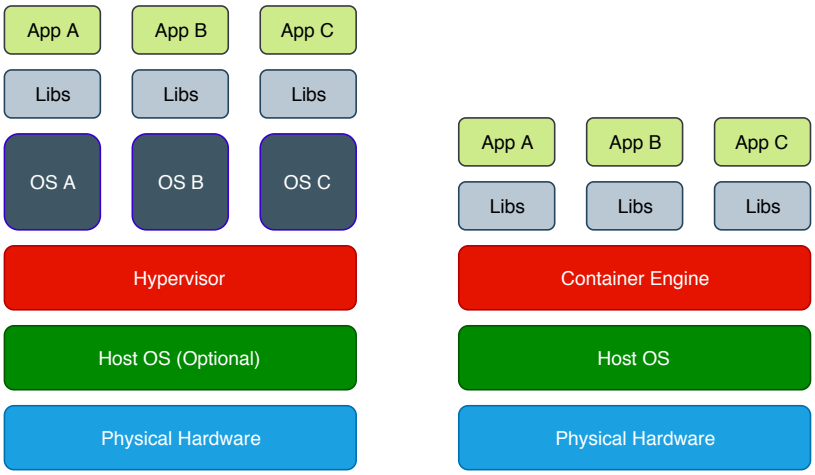
Virtualization was the key enabling technology for cloud computing. Most of the public cloud providers as well as private on-premises installations use virtualization technologies to power their infrastructure. Those VMs are powered by hypervisors, which provide isolation between the VMs running on top of it and the physical hardware. They allow running multiple different operating systems (OSs), and kernels respectively, side-by-side on a single host. On the one hand this approach provides a very strong isolation, on the other hand however, there is a lot of overhead which makes it expensive and decreases performance [Joy15].

2.4.1 Containers

Recent advancements in the Linux kernel development, most importantly *control groups (cgroups)* and *namespaces*, led to the creation of Linux Containers (LXC) [Joy15; Bur+16]. LXC allows running multiple isolated Linux systems – named *containers* – on a single control host sharing its single Linux kernel instance. This approach brings a major performance increase compared to classical virtualization as the hardware does not have to be emulated anymore [Joy15]. Instead all containers on the control host are using one single host kernel, while nevertheless being isolated from each other. Figure 2.4 compares the architecture of the hypervisor-based virtualization of VMs with container-based virtualization.

In addition to the already mentioned performance benefit, containers have a number of additional advantages [Joy15; MKK15]:

- *Portable deployments:* Container images are portable. Applications can be bundled into a single unit of execution including all their dependencies and runtimes. Hence they can easily be deployed to various environments.
- *Fast application packaging and delivery:* Container image formats are standardized lightweight formats which makes it easy and fast to create new images, thus



(a) Virtual Machines (b) Containers

Figure 2.4: Architectural comparison between virtual machines and containers according to Bernstein [Ber14]

accelerating application development processes and potentially avoiding vendor lock-ins which accompany proprietary virtualization solutions.

- *Higher workload density:* Due to the decreased overhead compared to traditional hypervisor-based virtualizations more workload can be deployed on the same hardware.

2.4.2 Docker

Docker[4] is currently one of the most popular container-based virtualization solutions. It was formerly built on LXC, but is now using another container runtime, *containerd*.

Containerd[5] is a container runtime which manages the complete lifecycle of the containers, from the image transfer and storage to the container execution, it's supervision, the container networking, and -storage. In order to spawn new containers, there is another layer of abstraction. containerd allows using any Open Container Initiative (OCI) runtime specification[6] compliant tool.

runC[7] is the reference implementation of the OCI runtime specification. It is a lightweight command line interface (CLI) tool to spawn and run containers.

Figure 2.5 shows how these different components are interacting with each other.

[4] `https://www.docker.com` (visited on Nov. 26, 2019)
[5] `https://containerd.io` (visited on Nov. 26, 2019)
[6] `https://github.com/opencontainers/runtime-spec` (visited on Nov. 26, 2019)
[7] `https://github.com/opencontainers/runc` (visited on Nov. 26, 2019)

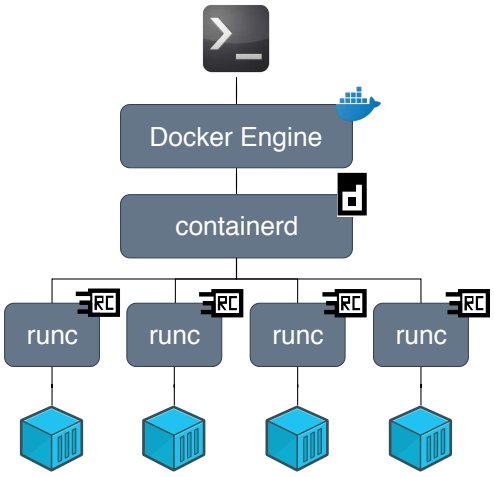


Figure 2.5: Docker architecture: The user interacts with the *Docker Engine*, either by using the Docker CLI or via other tools like a container orchestrator. The Docker Engine instructs *containerd* to create the container. containerd is then using *runC* (or any other OCI compliant runtime) to actually spawn the container, whose library *libcontainer* in turn is then finally using the kernel's API.

Besides the well structured layered architecture, Docker introduced a lot of new features [Ber14; Joy15; TRA15]:

- *Layered Filesystem:* Docker uses a copy-on-write filesystem. This dramatically decreases the container image build time, as only changed and subsequent layers have to be built again. Moreover, this approach allows the usage of base images which can be shared and built upon by multiple different container images.
- *Tooling:* Docker provides a sophisticated set of tools. By default the Docker daemon running on a host machine can be remote controlled via an HTTP API. This API is used by the Docker CLI as well as lots of other tools.
- *Ecosystem:* Docker has a very vivid community. With DockerHub, there is a public registry for sharing Docker container images. As of November 2019, more than 2.8 million images are hosted there[8]. They can be pulled and instantiated but, due to the layered filesystem, also being built upon for free.

2.4.3 Orchestration

Mordern microservice-based applications are composed of a lot of small services. Each of these services should be replicated, potentially in a geographically dispersed manner, to cope with system failures. The services have to be able to discover each other and the

[8] `https://hub.docker.com/search/` (visited on Nov. 26, 2019)

network traffic needs to be balanced between multiple instances of the same service. This results in very complex distributed application configurations which cannot be handled manually anymore. In fact, the decision making for placing m containers on n nodes has a complexity of $\mathcal{O}(n^2)$ [Kha17].

Container orchestration (CO) platforms provide the necessary means of integration and management of containers at scale. Multiple containers can be managed as one entity, while the CO platform ensures fault tolerance, availability, scalability, and reliability.

There are a number of different open-source CO platforms. The most prominent ones are the follows [Tru+19]:

- *Docker Swarm*[9]*:* A CO platform built by the Docker team. It combines multiple Docker hosts to one virtual Docker host and exposes the default Docker API. This allows reusing all the tools which can interact with the Docker daemon.
- *Marathon*[10]*:* A CO platform which is based on Apache Mesos[11]. It runs on top of Mesos and orchestrates the containers on Mesos nodes.
- *Kubernetes*[12]*:* A CO platform originally developed by Google. Kubernetes is described in detail in Section 2.4.4.

2.4.4 Kubernetes

Kubernetes (K8s) is a CO platform which has been announced by Google in June 2014 at the Google Developer Forum. It is the third CO platform created by Google. First there was Borg, then Omega and finally K8s, each heavily influenced by its predecessors [Bur+16]. K8s, however, was the first to be fully open-sourced.

Since the announcement, the K8s open-source community as well as the amount of users has grown significantly. As of September 2019, the GitHub project has more than 60,500 stars, 21,400 forks, and 2,300 contributors[13]. K8s has become the de-facto standard for CO [SD19].

At its core, K8s is configured in a declarative manner. Everything in K8s is a declarative configuration object which represents the desired state of an object within the system. K8s continuously tries to make sure that the current actual state matches the currently desired state [HBB17]. This feature also describes the self-healing abilities of K8s as every single failure of a component is basically just a mismatch of the actual state to the desired state.

The architecture of a K8s cluster is shown in Figure 2.6. The responsibilities of the different components are as follows.

[9] `https://docs.docker.com/engine/swarm` (visited on Nov. 26, 2019)
[10] `https://mesosphere.github.io/marathon` (visited on Nov. 26, 2019)
[11] `https://mesos.apache.org` (visited on Nov. 26, 2019)
[12] `https://kubernetes.io` (visited on Nov. 26, 2019)
[13] `https://github.com/kubernetes/kubernetes/` (visited on Nov. 26, 2019)

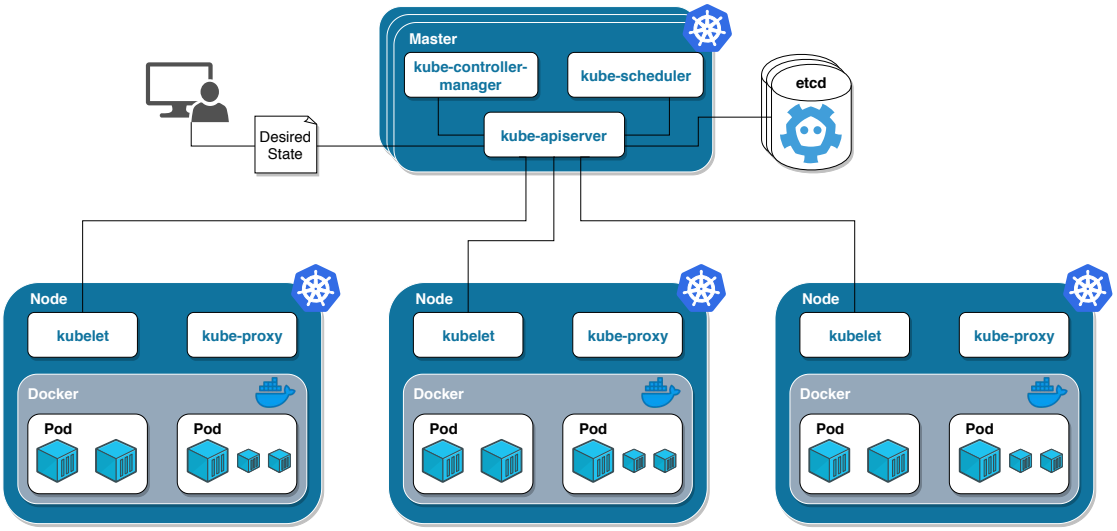


Figure 2.6: Kubernetes architecture

Pods

A pod is the smallest and simplest deployable unit in K8s and represents a single instance of an application or service. It is a group of one or more containers which are always scheduled together and will be co-located on the same node next to each other in a shared context.

The shared context of the containers within a single pod shares the configured storage resources, a unique IP address, and can even find each other via localhost or communicate using standard inter-process communication like shared memory.

Nodes

A node, previously known as a minion, is a single worker machine in K8s. Depending on the cluster it can be a physical machine or just a VM. It runs all services necessary to spawn and manage the pods like the kubelet, kube-proxy, and most importantly a container runtime like Docker.

kube-apiserver

As the name indicates, the kube-apiserver's purpose is to serve the K8s HTTP API.

In contrast to its predecessor Omega, K8s is not exposing its shared persistent store directly to trusted components, but only allows accessing the state through the domain-specific HTTP API served by this component[Bur+16].

kube-controller-manager

The kube-controller-manager is running the different controllers which in turn are running a control-loop for a specific purpose. A control loop is a non-terminating loop which continuously regulates a system.

Within their control loops, these controllers are continuously watching the shared state of the cluster – like every component it only accesses the cluster state through the API server – and, if necessary, invokes changes in order to move the current state towards the desired state.

An example of such a controller is the replication controller which continuously monitors the amount of different instances of a pod. If an instance dies or the pod's replication count is increased it will spawn new ones, if the replication count is decreased it will terminate the spare instances.

kube-scheduler

kube-scheduler is the K8s component which performs the scheduling as described in Section 2.3. It continuously watches for newly created pods that have not been assigned to a node yet. If a new pod is discovered, the kube-scheduler is responsible for assigning the pod the best suiting node to run on.

etcd

etcd[14] is strongly consistent, distributed key-value store. K8s stores all its management data in etcd, including the configuration, the desired state, and the actual state of the cluster.

kubelet

Each node runs a kubelet, which is the primary agent controlling the pods running on the node. It takes a pod's specification (primarily) from the API server and ensures that the container running on the node meets the given specification. E.g. it spawns new containers if they haven't been created yet or restarts unhealthy ones.

kube-proxy

kube-proxy is also running on each node and is taking care of the node's network rules. These rules are necessary to enable the network communication between the node's pods and any other component inside or outside of the cluster. By default it uses the packet filtering layer of the OS – e.g. iptables – if available.

[14] `https://etcd.io/` (visited on Nov. 26, 2019)

2.5 Multi-objective Optimization

The *optimization problem* is the problem of finding the best solution of all *feasible* solutions and is defined as follows[15] [BV04]:

Problem 1 (Optimization Problem)**.**

$$\begin{aligned} &\min/\max && f_0(x), \\ &\text{subject to} && f_i(x) \leq b_i,\ i = 1, \ldots, m. \end{aligned} \tag{2.1}$$

where the vector $x = (x_1, \ldots, x_n)$ is the *optimization variable* of the problem, the function $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the *objective function*, the functions $f_i(x) : \mathbb{R}^n \to \mathbb{R}, i = 1, \ldots, m$ are the (inequality) *constraint functions*, and the constants $b_1 \ldots, b_m$ are the limits, or bounds, for the constraints.

Definition 1 (Solution)**.** A vector x^* is called *optimal*, or a *solution* of the Problem 2.1, if it has the smallest objective value among all vectors that satisfy the constraints: for any $z \neq x^*$ with $f_1(z) \leq b_1, \ldots, f_m(z) \leq b_m$, we have $f_0(z) \triangleright f_0(x^*)$ where $\triangleright$ is $\geq$ if f_0 is to be minimized and $\triangleright$ is $\leq$ if it is maximized.

But in most practical decision-making problems, multiple – sometimes conflicting – objectives have to be taken into account. Therefore the definition of Problem 2.1 is extended by defining multiple objective functions and introducing more complex constraints to define the *multi-objective optimization problem (MOP)* as follows [Deb01]:

Problem 2 (Multi-objective Optimization Problem)**.**

$$\begin{aligned} &\min/\max && f_m(x), && m = 1, \ldots, M; \\ &\text{subject to} && g_j(x) \geq 0, && j = 1, \ldots, J; \\ & && h_k(x) = 0, && k = 1, \ldots, K; \\ & && x_i^{(L)} \leq x_i \leq x_i^{(U)}, && i = 1, \ldots, n. \end{aligned} \tag{2.2}$$

where the vector $x = (x_1, \ldots, x_n)$ is the *optimization variable* of the problem, the functions $f_m(x) : \mathbb{R}^n \to \mathbb{R}, m = 1, \ldots, M$ are the *objective functions*, the functions $g_j(x) : \mathbb{R}^n \to \mathbb{R}, j = 1, \ldots, J$ define the *inequality constraints*, the functions $h_k(x) : \mathbb{R}^n \to \mathbb{R}, k = 1, \ldots, K$ define the *equality constraints*, and $x_i^{(L)} \leq x_i \leq x_i^{(U)}, i = 1, \ldots, n$ define the *variable bounds*.

Definition 2 (Feasibility)**.** A solution $x = (x_1, \ldots, x_n)$ is a *feasible* solution if it fulfills all constraints and variable bounds.

[15] This definition has been extended to allow maximization, which is admissible due to the *duality principle* in optimization [Deb01]. It suggests that a minimization problem can be converted into a maximization problem by multiplying the objective function by -1.

If you have an optimization problem with only a single objective, the superiority of a feasible solution over another one is easily determined as the results of their objective functions can be compared directly. However, if you have more than one objective function that are to be minimized or maximized the answer is a set of solutions that define the best trade-off between the competing objectives. It is not beneficial to consider all feasible solutions, as most of them will be *dominated* by others.

Definition 3 (Domination)**.** A feasible solution y is said to *dominate* another feasible solution z ($y \succ z$), iff $f_m(y) \not\triangleleft f_m(z)$ for $m = 1, \ldots, M$ and $f_m(y) \triangleleft f_m(z)$ for at least one $m \in 1, \ldots, M$ where $\triangleleft$ is $<$ for the functions f_m which are minimized and $\triangleleft$ is $>$ for those which are maximized.

In other words y dominates z iff y is no worse than z in all objectives and y is strictly better than z in at least one objective. The following definitions build upon the concept of domination and lead to the definition of the *pareto-optimal front*.

Definition 4 (Pareto-optimality)**.** A solution is said to be *pareto-optimal* iff it is *non-dominated*, i.e. there is no other solution which dominates it.

Definition 5 (Pareto-optimal set)**.** The set of all pareto-optimal solutions in the solution space is called the *pareto-optimal set*.

Definition 6 (Pareto-optimal front)**.** The boundary which is defined by the set of all points mapped from the pareto-optimal set is called the *pareto-optimal front*.

The pareto-optimal front of an MOP with two objectives, each of which to be maximized, is illustrated in Figure 2.7.

The goal of a multi-objective optimization algorithm is to find non-dominated, feasible solutions as close to the pareto-optimal front as possible. Each of these solutions constitute a specific trade-off between the different competing objectives [Deb14]. Users of the optimization algorithm can then make a choice based on their preferences.

2.6 Machine Learning Workflows

Machine learning (ML) refers to systems used for the automated detection of patterns in data, which do have the ability to ”learn” and adopt [SB14].

In the past decade, there has been a rapid adoption of ML into a variety of applications and is thereby already influencing a significant portion of our daily lifes [RD19]. This was possible due to tremendous advances in artificial neuronal networks, an immense growth of available data, continuous advances in processing power, as well as the development and usage of specialized hardware and software components [LBH15]. Modern ML frameworks like Apache MXNet[16], Tensorflow[17], or PyTorch[18] have greatly reduced the

[16] `https://mxnet.apache.org/` (visited on Nov. 26, 2019)
[17] `https://www.tensorflow.org/` (visited on Nov. 26, 2019)
[18] `https://pytorch.org/` (visited on Nov. 26, 2019)

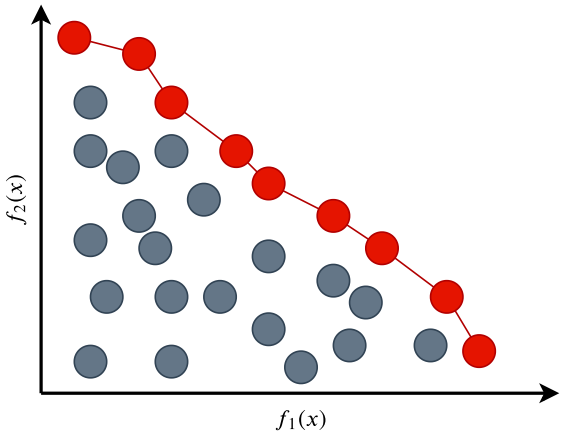


Figure 2.7: Pareto-optimal front: Assuming $f_1(x)$ and $f_2(x)$ are to be maximized, the red line represents the pareto-optimal front. Each dot represents a configuration. Those configurations located on the pareto-optimal front, here in red, are forming the pareto-optimal set. All other configurations are Pareto-dominated by at least one configuration on the frontier.

required effort and skills to implement ML systems [Boa+17].

Even though there are lots of different algorithms and types of ML systems, their simplest workflow can be broken down to the following three steps as illustrated in Figure 2.8 [Hum+19].

1. *Data Preprocessing:* ML algorithms are mathematical algorithms which do expect a certain type of input data. Therefore the raw data usually cannot be used directly but has to be preprocessed. Typical steps include the handling of null values, data standardization (i.e. transforming the raw data such that the mean of the values is 0 and the standard deviation is 1), one-hot encoding (i.e. creating one binary column for each unique value of a nominal variable), or the handling of multicollinearity (i.e. removing features which are strongly dependent on each other).

2. *Model Training:* This step is the execution of the actual ML algorithm. The preprocessed data is used to incrementally improve an ML model.

3. *Model Serving:* The trained model is used in order to analyze new data. For example, a webservice is using the trained model in order to recognize numbers in an image.

Due to the rapid adoption into production systems, it becomes necessary to operationalize ML and create platforms handling the ML workflows in production [Hum+19; Li+17; Bay+17; Boa+17; Car+18].

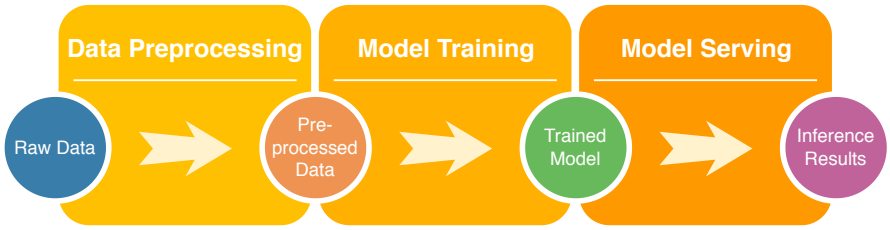


Figure 2.8: Simple machine learning workflow

The extension of ML systems closer to the user, embracing edge computing, will enable a new category of applications and could ultimately support the rise of human augmentation – a field of research that aims to enhance the abilities of human beings through modern medicine and technology. However, this means moving the automated ML application lifecycle management from the cloud to the edge of the network [RD19].

CHAPTER 3

Related Work

This chapter discusses related work relevant in the context of this thesis. Section 3.1 describes currently existing commercial products as well as academic and experimental efforts aiming to combine serverless computing with edge computing. In Section 3.2, current developments on how to manage containers in edge computing are summarized. Finally, Section 3.3 outlines state-of-the-art work on resource provisioning and deployment scheduling in edge computing.

3.1 Serverless at the Edge

As an initiative to combine edge computing with serverless computing, Nastic and Dustdar defined the term *Deviceless Edge Computing* and proposed a high-level reference architecture of such systems [ND18]. They introduce the concept of *Software-Defined Gateways* and the *Intent-based programming model*, aiming to facilitate the application development, and the automated provisioning and management. While these concepts are defined on a high level, they describe a uniformed platform based on a mixed cloud-edge infrastructure allowing transparent deployments of functions similar to our platform.

In a previous publication, Nastic et al. presented a serverless edge-data analytics platform and application model [Nas+17]. The core of the platform is a stream-processing model abstracting the heterogeneous edge computing infrastructure. It thereby allows the seamless integration of and deployment on edge devices but is restricted to the domain of stream-processing.

Cheng et al. proposed a *data-centric programming model* and an underlying orchestration mechanism in order to apply the simplicity and flexibility of FaaS on edge computing [Che+19]. Its orchestrator includes the *data context* (metadata of available data), the *system context* (available resources at each node), and the *usage context* (a high-level

usage description). The proposed system has been implemented in FogFlow[1], and shows benefits concerning the internal data traffic and ultimately the service latency, however their scheduling algorithm does not encompass the possibility to utilize the benefits of specific hardware accelerators.

In another recent effort, Baresi and Mendonça proposed a *Serverless Edge Platform* [BM19]. They created a prototype platform based on OpenWhisk. However they do not handle the scheduling of the function containers at all as they proposed running all components of the FaaS framework on each node. This is only feasible as they assume to use less resource constrained nodes, each having 16 GB of RAM and 12 CPU cores.

Currently there are already several commercial products extending their cloud offerings to the edge. AWS IoT Greengrass[2], an extension of AWS Lambda, allows connected devices to invoke AWS Lambda functions on the AWS IoT Greengrass Core device placed on the network edge. However the deployment configuration, i.e. the set of functions invokable by the IoT devices, has to be done manually. The same terms apply for Microsoft's Azure IoT Edge[3]. The IoT products by the two remaining global cloud computing competitors, Google Cloud IoT Core[4] and IBM's Watson IoT Platform[5], do not allow any edge-located devices to execute FaaS functions yet.

OpenWhisk[6], a popular open-source FaaS platform initiated by IBM and now an Apache project, has been slightly modified by Breitgand to demonstrate an exemplary setup of OpenWhisk in an edge scenario [Bre18]. Yet, it is just a slimmed version of OpenWhisk assuming that the complete platform runs on the edge device instead of a mixed cloud-edge infrastructure. This means that the functions cannot run on different edge devices or even in the cloud. Our platform however encompasses many nodes in a mixed cloud-edge cluster and for each workload individually decides where to run the functions.

3.2 Container Orchestration at the Edge

In an evaluation of the usage of Docker containers in an edge computing platform, Ismail et al. identified several aspects of Docker which would be beneficial when being used in such a platform, including *a*) its low footprint, *b*) its portability, and *c*) its performance [Ism+15]. All major open-source FaaS platforms are based on Docker for their function deployments. Therefore a specialized scheduler for Docker containers can easily be integrated and fits well for edge computing environments.

K8s, initially created by Google, is currently the most popular CO system and has emerged to a de-facto standard in its domain (see Section 2.4.4). In order to address

[1] `https://github.com/smartfog/fogflow` (visited on Nov. 26, 2019)
[2] `https://aws.amazon.com/greengrass/` (visited on Nov. 26, 2019)
[3] `https://azure.microsoft.com/en-us/services/iot-edge/` (visited on Nov. 26, 2019)
[4] `https://cloud.google.com/iot-core/` (visited on Nov. 26, 2019)
[5] `https://www.ibm.com/internet-of-things/solutions/iot-platform/watson-iot-platform` (visited on Nov. 26, 2019)
[6] `https://openwhisk.apache.org/` (visited on Nov. 26, 2019)

the emerging topic of edge computing a new working group has been instituted in June 2018, focusing on how K8s can be used in such scenarios [Kubb]. One result of the working group will be a whitepaper on different use-cases, how they can look like from an infrastructural perspective, and how they can be handled with current and upcoming features of K8s. Currently, only the work-in-progress document is available yet.

As edge computing enables heterogeneous – often resource-constrained – devices, *K3s*[7] aims to create a lightweight version of K8s. While being a fully certified K8s distribution, it removes lots of non-mandatory components of a default K8s distribution and specifically aims at supporting ARM devices. The support of these processors, which are heavily used in resource constrained devices, as well as the drastically lower resource consumption perfectly aims on edge devices. However, the project just released its first stable version. Once the project becomes mature, our scheduler can be integrated in *K3s* clusters like in any other K8s cluster. This will ultimately allow the creation of even more resource preserving clusters without any integration efforts.

There are also some scholarly efforts on using or even extending K8s to handle edge deployments. Wöbker et al. used existing features provided by the K8s Scheduler (namely *labels* and *nodeSelectors*) to define the capabilities of nodes and influence the scheduling [Wöb+18]. As these labels and selectors are assigned in advance and are not adopted, they do not dynamically adjust to changes in the infrastructure. While we are also using labels as means of providing scheduling metadata, we additionally replaced the scheduler itself. This allowed us to implement highly domain-specific scheduling logic.

Xiong et al. implemented a set of extensions to K8s, called *KubeEdge* [Xio+18]. Its most important component, the *EdgeCore* client running on the nodes, manages the networking and potentially masks network failures. While this is a promising approach on extending K8s to handle edge infrastructures, it does not address any changes to the K8s scheduler in order to include the different network characteristics of such an infrastructure. As with *K3s*, once this project becomes mature our scheduler could be integrated into a *KubeEdge* cluster.

Due to the different layers of abstraction and its thoughtful design, K8s is highly pluggable and customizeable. Buzachis et al. compared different K8s overlay networks, responsible for the networking between the deployed pods, focusing on their usage in an edge computing environment [Buz+18]. Even though their study does not directly relate to this thesis, it shows the immense amount of possible layers of optimization and the benefit of such a well-structured architecture.

3.3 Scheduling at the Edge

Skarlat et al. described a model for automatic Quality-of-Service-aware deployments in edge infrastructures by taking into account various non-functional requirements relevant in edge computing infrastructures [Ska+16]. The model has later been refined and they

[7] `https://k3s.io` (visited on Nov. 26, 2019)

defined a general problem definition based upon it [Ska+17b; Ska+17a]. As the problem is proven to be $\mathcal{NP}$-hard, a heuristic was proposed to find near-optimal solutions in polynomial time. However, the model is highly based on the notion of *colonies* which form a three-tier hierarchical architecture, as each colony has its own control node, managing a set of cells, which in turn is controlled by the cloud middleware. Furthermore, the model would need to be extended, as it only considered CPU, RAM, and storage constraints.

Scoca et al. proposed a two staged score-based algorithm that considers several hardware and network metrics in order to find the best deployment configuraton for latency-sensitive applications in edge computing infrastructures [Sco+18]. First, the eligibility of each node to host a given workload is determined in the form of a score. The scheduler then tries to find a configuration which maximizes the score. This approach is actually very similar to the implementation of the default scheduler in K8s (see Section 5.2).

In an attempt to increase the efficiency of K8s clusters, Ungureanu, Vlădeanu, and Kooij replaced the default scheduler with a hybrid shared-state scheduler [UVK19]. It delegates most of the tasks to the distributed scheduling agents. A centralized correction function manages unscheduled and unprioritized pods. The *master-state agent* is also centralized and takes care of the cluster state synchronization.

CHAPTER 4

Methodology

As stated in Section 1.3, the aim of this thesis is to design, develop, and evaluate an integrated, optimized, latency- and capability-aware scheduler for running an FaaS platform in a mixed cloud-edge computing environment. Initially, we specify the platform to build upon. Section 4.1 describes the software foundation and the motives of their selection. In Section 4.2, an exemplary scenario is defined and motivated. This leads to the setup of the testbed, as described in Section 4.3. This test environment is then used to execute well-defined empirical measurements. Their design is described in Section 4.4. Our scheduler is using additional function-specific metadata within its placement algorithm. Section 4.5 defines the schema of the metadata. All those developments form the foundation for the actual scheduler implementation, which is briefly outlined in Section 4.6 (see Chapter 5 for the detailed description). In order to speed up the tests and the following optimizations, it becomes necessary to implement a simulation environment for the scheduler, which is described in Section 4.7. Finally, Section 4.8 outlines the usage of the scheduler simulation for the efforts on optimizing the scheduler configuration.

4.1 Framework Selection

4.1.1 Container Orchestration Platform

The most prominent CO platforms have been listed in Section 2.4.3, namely Docker Swarm, Marathon, and K8s. In addition to the fact that K8s has emerged to a de-facto standard when it comes to CO (see Section 2.4.4), its architecture and flexibility brings several advantages:

Pluggable Scheduler K8s allows replacing the default scheduler or even running multiple schedulers side-by-side out-of-the-box.

Diverse Hardware Support K8s and its underlying container runtime containerd are written in Go. Their releases are compiled for Windows, Linux, and lots of different CPU architectures by default.

Labels In addition to specified resource-limits and -requests for containers, K8s allows setting additional metadata. These *labels* are intended to be used to specify identifying attributes of objects. They allow a structured labeling of any K8s object in a loosely coupled fashion and they can be read and modified at any time through the *kube-apiserver*.

FaaS Integration As stated in Section 4.1.2, all but one of the evaluated FaaS frameworks either primarily or optionally integrate with K8s as CO for its functions. Therefore an integration of the selected FaaS platform is trivial.

Scientific Relevance Due to its prominence, K8s is becoming a focal point in research when it comes to CO.

Due to these advantages of K8s over Docker Swarm and Marathon, K8s has been selected to be the CO platform of choice for the implementation of our project.

4.1.2 Function-as-a-Service Platform

When selecting the FaaS platform for this project, the following criteria are being considered:

Is the project open-source? We need full control over the platform, which can only be guaranteed by using an open-source solution.

Does the open-source license allow usage and modification? The granted privileges of open-source licenses can vary a lot. The platform needs to be licensed in a way that allows the usage as well as the modification of its components for non-commercial usage.

Is it actively developed (is the project "alive")? An active community is essential for an open-source project. It indicates how fast bugs are fixed and how much help the user can rely on. As there is no universal metric, the amount of GitHub stars, the latest commit, and the number of contributors are widely used indicators of open-source project liveliness.

Does it integrate with popular CO platforms? We do not only want to modify the FaaS platform but also the underlying CO platform. Therefore the CO platform it integrates with is fundamental for the FaaS platform selection.

Does it support different CPU architectures? As stated in Section 2.1, there is a high heterogeneity among edge devices. Especially resource constrained edge

devices usually do not work on the usual AMD64 CPU architecture, but on the (increasingly prominent) ARM(64) architecture. In order to build our testbed, it is mandatory that the platform supports those architectures as well.

Table 4.1 shows the most prominent open-source FaaS platforms with their licenses, the CO platforms they integrate with, their supported CPU architectures, their GitHub stars, and their GitHub contributor count as of Oct. 16, 2019.

OpenFaaS appears to be one of the most active projects among all of the evaluated ones. It has recent commits, a high amount of contributors, and by far the most GitHub stars. Moreover it integrates with K8s and Docker Swarm, and has a very open license. Mohanty, Premsankar, Di Francesco, et al. found OpenFaaS to be easily extendable with a flexible architecture [MPD+18]. It also shows a reasonable performance in terms of request throughput [PKC19]. However, the most important factor is that it is the only project which supports CPU architectures other than AMD64.

4.2 Scenario

Possibly the most prominent scenario, and one of the main driver of IoT, is its usage in the area of manufacturing. As more and more manufacturing machines are equipped with sensors, measuring all the different runtime properties of the machines, the collected data holds great potential for a broad range of applications. For example, the sensor data can be used to draw conclusions about the health or the production efficiency of the system under consideration. Thereby, the area of predictive maintenance is increasingly becoming a point of focus. By using these newly available vast amounts of data in combination with modern ML techniques, it becomes feasible to predict upcoming failures before they actually happen. This allows the targeted scheduling of maintenance tasks in order to avoid those failures from actually happening.

The sensors on the machines are continuously read and their data is sent to the predictive maintenance system. This system, in turn, then *a*) uses its current ML model to detect if the data indicates possible upcoming failures, and *b*) repeatedly re-trains its model using the latest sensor data as well as reports on failures which have not been detected in advance. Figure 4.1 illustrates the usage of edge devices on the premises of the factory.

As stated in Section 3.1, there already are commercial products available (like AWS Greengrass) which allow the execution of FaaS functions on-premises on devices in the factory itself. However, their configuration is highly static, they do not allow the usage of specialized hardware, and cannot easily fall back to using cloud-resources in case of failing nodes or high utilization. Therefore, for the sake of seamless scalability and failure-safety, it is necessary to avoid separation and manual configuration, but rather create one heterogeneous cluster reaching to the edge of the network intelligently placing a new workload on the one node which suits it best.

Platform	GitHub Repo	License	CO	CPU Archs	Latest Commit	Stars	Contr.
OpenWhisk	apache/openwhisk	Apache License 2.0	K8s Docker Compose Mesosphere	AMD64	15.10.2019	4290	163
Kubeless	kubeless/kubeless	Apache License 2.0	K8s	AMD64	02.09.2019	5128	81
Fission	fission/fission	Apache License 2.0	K8s	AMD64	12.10.2019	4664	90
Knative	knative/serving	Apache License 2.0	K8s	AMD64	16.10.2019	2344	150
Fn Project	fnproject/fn	Apache License 2.0	Docker	AMD64	03.10.2019	4270	84
OpenFaaS	openfaas/faas	MIT License	K8s Docker Swarm	AMD64 ARM ARM64	15.10.2019	15806	125
Riff	projectriff/system	Apache License 2.0	K8s	AMD64	16.10.2019	746	20
nuclio	nuclio/nuclio	Apache License 2.0	K8s	AMD64	02.10.2019	2995	46
IronFunctions	iron-io/functions	Apache License 2.0	K8s Docker Swarm Mesosphere	AMD64	20.08.2018	2652	32

Table 4.1: Comparison of open-source FaaS platforms as of Oct. 16, 2019

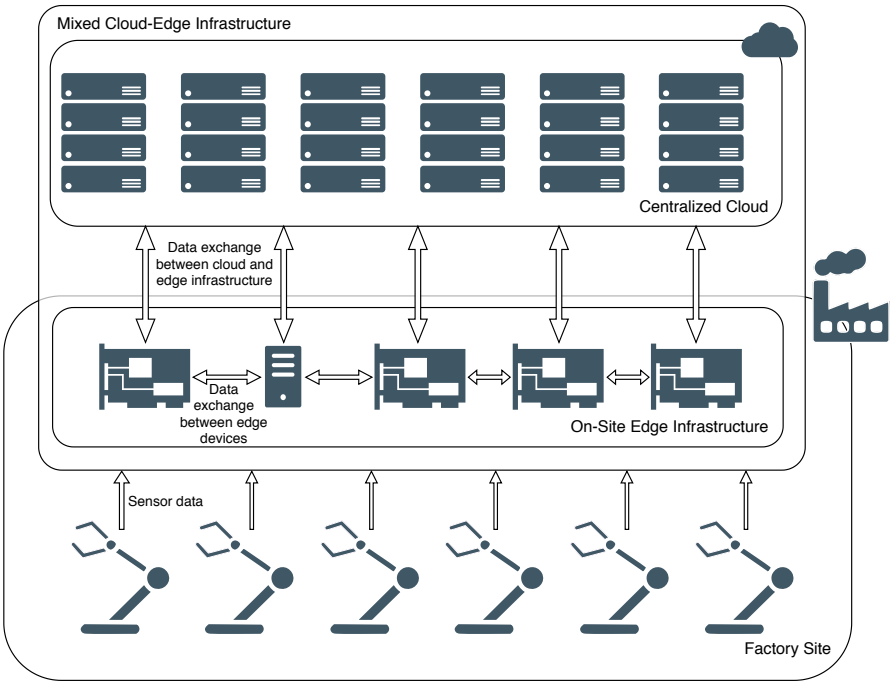


Figure 4.1: Illustration of a predictive maintenance scenario

4.3 Testbed

In order to perform empirical measurements (as described in Section 4.4) and to show the feasibility of our scheduler being fully integrated in a running system, a testbed – running the selected technologies described in the previous sections – has been created. With the exception of the initial OS setup, all installation steps have been automated using Ansible[1] and bash scripts. This enables the scalability in terms of nodes in the testbed and simplifies setting up or resetting the testbed.

4.3.1 Nodes

The testbed needs to simulate a heterogeneous mixed cloud-edge environment. Therefore the setup consists if six different nodes. All nodes are interconnected using a 1 GBit/s Ethernet connection (their connectivity is programmatically limited using a kernel-module as explained in Section 4.4). Figure 4.2 illustrates the testbed's node composition.

Five nodes are created using edge hardware, namely four *Raspberry Pi 3B+* and one *NVidia Jetson TX2 Developer Kit*.

The *NVidia Jetson TX2* is available as an integrated module (meant for production settings) and as a developer kit (as the name suggest for the purpose of creating new

[1] `https://www.ansible.com/` (visited on Nov. 26, 2019)

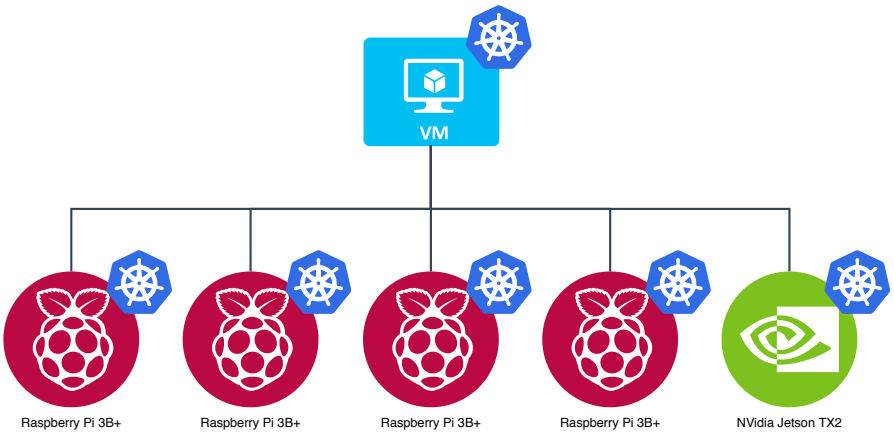


Figure 4.2: Illustration of the nodes in the testbed

applications). Even though these two different types differ in their dimensions and the provided peripherals, their hardware components are equivalent. Therefore, even if we used the bulky NVidia Jetson TX2 Developer Kit in our testbed, it is substitutable with its compact module version. It has specifically been created for ML applications at the edge with a 256-core Graphics Processing Unit (GPU).

The *Raspberry Pi 3B+* on the other hand is not equipped with any ML hardware accelerator. But due to its small dimensions, its very low price, and its open-source design the different versions of the Raspberry Pi have already been sold more than 25 million times and are heavily used in industry and research [Ols19].

The hardware specifications of the different edge devices are shown in Table 4.2.

The remaining node is the most powerful one and represents a cloud node. It is a VM instance provisioned in an on-premises cluster featuring a 3 GHz Quad-Core (Intel Core 2 Duo P9xxx) and 8 GB RAM but without any GPU.

On the VM as well as the NVidia device *Ubuntu 18.04.3*[2] was installed. The Raspberry Pi devices were set up with *HypriotOS 1.9.0*[3] (a Raspbian-based distribution created to ease running Docker containers on Raspberry Pi devices).

A picture of the testbed's edge nodes and their network devices is shown in Figure 4.3.

4.3.2 Kubernetes

Once the nodes are set up and interconnected in a network, they are combined to form a K8s cluster using *kubeadm*, a CLI tool helping to bootstrap K8s clusters that conform to best practices. The VM node has the most resources and thereby acts as master, running the *kube-apiserver*, the *kube-controller-manager*, an *etcd* instance, and most importantly

[2] `https://ubuntu.com` (visited on Nov. 26, 2019)
[3] `https://hypriot.com` (visited on Nov. 26, 2019)

	Raspberry Pi 3B+	NVidia Jetson TX2 Dev Kit	NVidia Jetson TX2 Module
Release date	3.2018	3.2017	3.2017
Price	~35 €	~420 €	~500 €
CPU architecture	ARMv8-A (64/32-bit)	ARMv8-A (64/32-bit)	
Number of cores	4	6	
CPU clock	1.4 GHz Quad-Core ARM Cortex-A53	2 GHz Dual-Core NVidia Denver2	
		2 GHz Quad-Core ARM Cortex-A57	
GPU	Broadcom VideoCore IV	256-core Pascal GPU	
RAM	1 GB	8 GB	
Ethernet	1 GBit/s	1 GBit/s	
Height	85.6 mm	170 mm	87 mm
Width	56.5 mm	170 mm	50 mm
Depth	17 mm	50 mm	10 mm
Weight	45 g	-	88 g

Table 4.2: Edge device hardware comparison

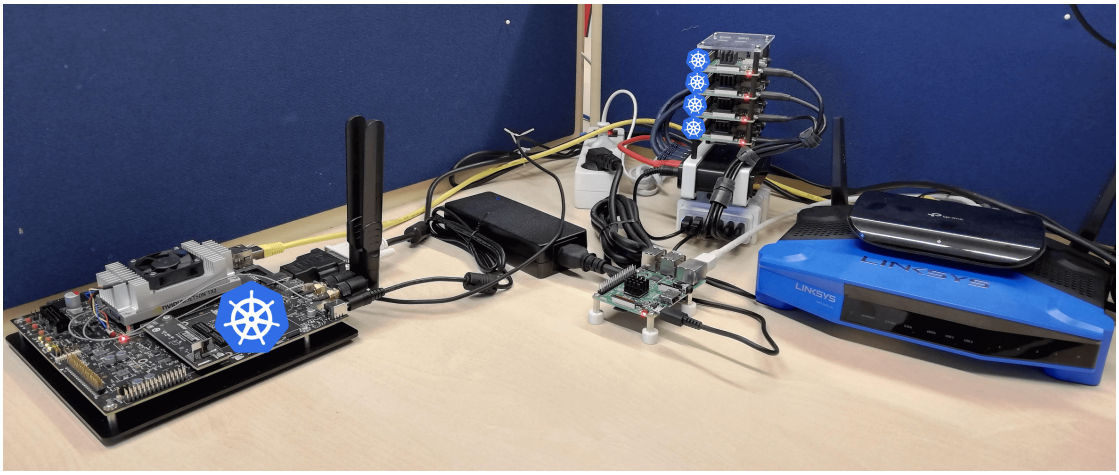


Figure 4.3: Picture of the testbed's edge nodes and their network devices

the *kube-scheduler* (as described in Section 2.4.4). Afterwards, all nodes – including the VM node – are joining the master as K8s nodes.

Due to the NVidia specific Ubuntu distribution, it is necessary to re-build its Linux kernel with additional flags in order to enable the cluster networking (managed by the local *kube-proxy* instance). Finally, due to NVidia's lack of support for their own ARM64 devices when it comes to their custom Docker runtime *nvidia-docker*[4], the creation and usage of a custom *runC* fork[5] is required to allow Docker containers, running on the device, access the GPU. Appendix A lists the necessary kernel flags and the patch for the container runtime.

4.3.3 Software

Once the K8s cluster is up and running, several other software components are deployed by either using *Helm*[6] (a package manager for K8s) or the CLI tool *kubectl*:

K8s Dashboard[7] The official dashboard is a web-based user interface (UI) for K8s clusters allowing users to manage the cluster itself as well as applications deployed on it.

Træfik Ingress Controller[8] Ingress controllers are responsible for routing traffic to K8s services and can add additional functionality like Transport Layer Security (TLS) termination, path rewrites, name based virtual hosts, and TLS certificate management. Træfik is a cloud-native open-source reverse-proxy and load-balancer. It fully integrates with K8s by providing a K8s Ingress Controller.

MinIO[9] As the FaaS functions need to be stateless, their only possibility to exchange data is through an external storage system. MinIO is an open-source object storage server which is fully compatible with Amazon's S3 cloud storage service.

OpenFaaS Last but not least, the selected FaaS platform is deployed on the cluster by using a Helm chart in OpenFaaS' custom Helm repository[10].

As soon as the ingress controller and OpenFaaS have been installed, the FaaS platform can be accessed and controlled either by using the OpenFaaS API gateway portal – as pictured in Figure 4.4 – in a browser or the CLI tool of OpenFaaS[11].

[4] `https://github.com/NVIDIA/nvidia-docker` (visited on Nov. 26, 2019)
[5] `https://github.com/alexrashed/runc/` (visited on Nov. 26, 2019)
[6] `https://helm.sh/` (visited on Nov. 26, 2019)
[7] `https://github.com/kubernetes/dashboard` (visited on Nov. 26, 2019)
[8] `https://traefik.io/` (visited on Nov. 26, 2019)
[9] `https://min.io/` (visited on Nov. 26, 2019)
[10] `https://github.com/openfaas/faas-netes` (visited on Nov. 26, 2019)
[11] `https://github.com/openfaas/faas-cli` (visited on Nov. 26, 2019)

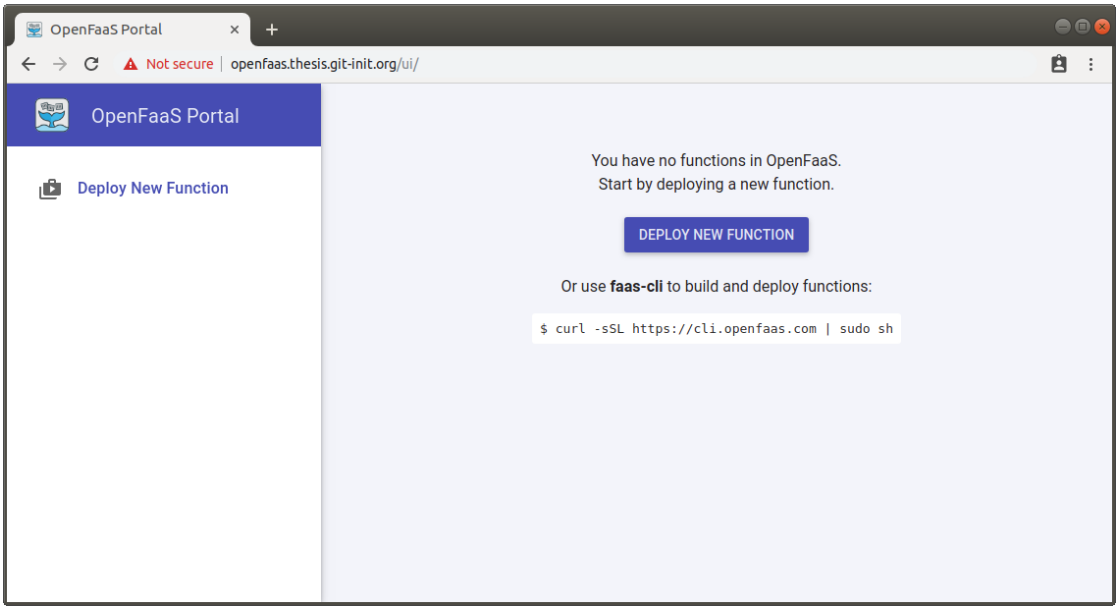


Figure 4.4: Screenshot of the OpenFaaS API gateway portal

Figure 4.5: MNIST example images [LCB98]

4.3.4 Test-Workflow

For the simulation and evaluation of our scheduler, it is necessary to create an example workload for our system. As described in Section 2.6 and Section 4.2, a workflow executing a series of ML tasks suits the usual requirements of our scenario. For the implementation, the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits is utilized [LCB98]. This database is widely used in ML research, experiments, and teaching. It contains 60,000 images for training a model and additional 10,000 for testing its precision. An example set of digits contained in the MNIST database is shown in Figure 4.5.

Since the functions need to be able to run on each node, the functions are implemented using Apache MXNet as it is the only popular ML framework supporting ARM devices. However, the Python wheel for the usage of MXNet in ARM Docker images has to be compiled manually. For the serving of the MXNet model, MXNet-Model-Server[12] is used. MXNet-Model-Server does not have any functionality to update currently served models once they change in their remote storage. Therefore an additional component is used to watch for changes in the remote storage and updates the served model once it detects any changes.

[12] `https://github.com/awslabs/mxnet-model-server` (visited on Nov. 26, 2019)

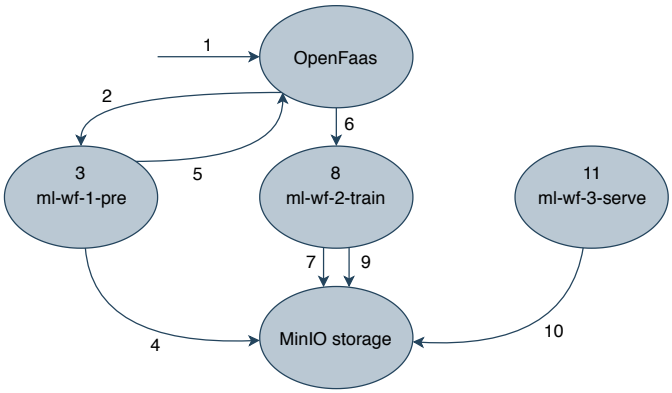


Figure 4.6: Execution-graph of a single workflow execution

As illustrated in Figure 4.6, the following steps are performed during one workflow execution:

1. OpenFaaS detects a function trigger. There are several ways to trigger a function execution, for example via an incoming HTTP request, a message on a specific message queue topic, or by reaching a scheduled task execution using cron-task syntax.
2. The FaaS framework triggers the execution of the first workflow step *ml-wf-1-pre*.
3. It downloads the MNIST images and converts them to a specific data format. This task represents the *Data Preprocessing* as described in Section 2.6.
4. The preprocessed data is stored in the external storage.
5. The first workflow step reports its completion to OpenFaaS.
6. OpenFaaS triggers the execution of the second workflow step *ml-wf-2-train*.
7. This function first downloads the preprocessed data from the external storage.
8. Afterwards, the data is used to train a model utilizing hardware accelerators (i.e. the GPU of the node) if possible.
9. The created model is again stored in the external storage.
10. In the meantime, the watchdog of the third function – *ml-wf-3-serve* – is continuously watching the external storage for changes on the model. Since it detects a change after the completion of *ml-wf-2-train*, it downloads the new model.
11. Finally, the new model is served by the MXNet-Model-Server in *ml-wf-3-serve*.

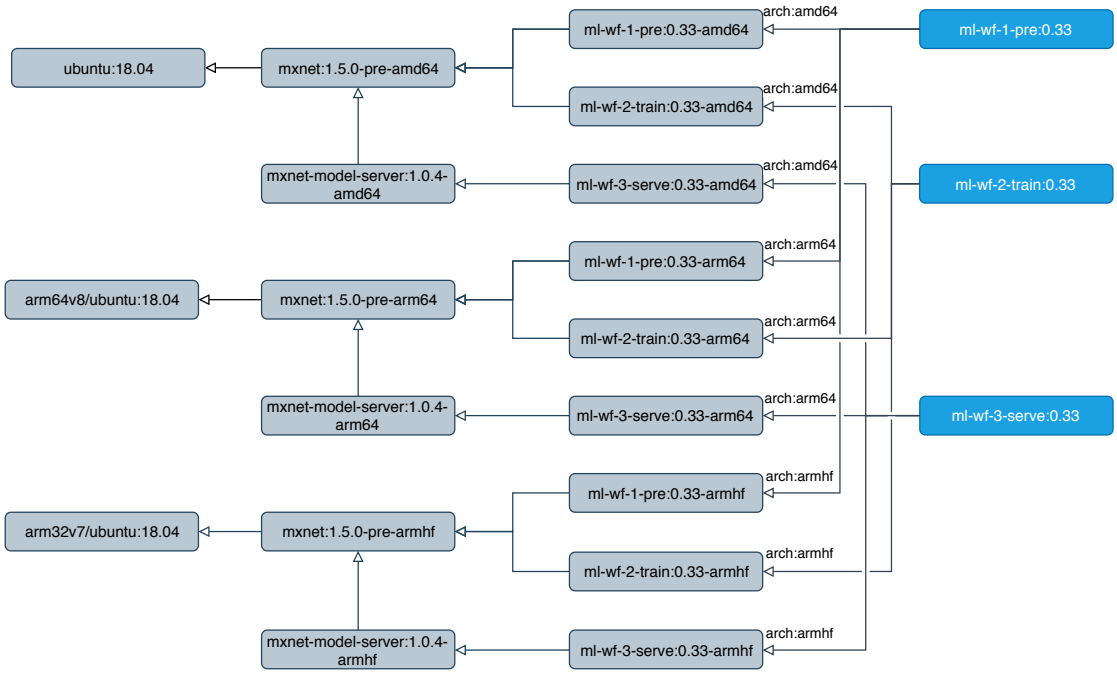


Figure 4.7: Hierarchy diagram of the workflow function Docker images

Workflow Step	CPU Architecture	Size (MB)
ml-wf-1-pre	armhf	440.10
	arm64	513.09
	amd64	505.73
ml-wf-2-train	armhf	482.59
	arm64	555.83
	amd64	522.01
ml-wf-3-serve	armhf	483.29
	arm64	558.50
	amd64	558.79

Table 4.3: Workflow function Docker image sizes per CPU architecture

To create a single Docker tag for a function, which in turn references one Docker image per CPU architecture, it is necessary to create cross-compiled images which then are referenced using a – currently experimental – Docker feature called *manifest list*. This results in a complex Docker image hierarchy shown in Figure 4.7.

The size of an image impacts the time it takes for the host to pull the image from the registry. Table 4.3 lists the compressed image sizes for each function and CPU architecture.

Download Speed	Delay	Deviation
200 MBit/s	12 ms	1 ms
100 MBit/s	32 ms	3 ms
10 MBit/s	100 ms	10 ms

Table 4.4: Characteristics of bandwidth presets for the empirical measurements

4.4 Empirical Measurements

Empirical measurements form the basis for the evaluation of the placement quality of scheduling decisions within the simulator. Therefore it is necessary to measure a broad range of placements under different configurations of the cluster nodes.

The following variables are considered during the execution of the measurements:

Nodes The placements have to be executed on each node of the testbed.

Workflow Steps Each workflow steps shows different characteristics, therefore each step has to be measured independently.

Image State The function images are highly specific and use complex frameworks and libraries, which results in rather large Docker images. If the Docker image of a function is not present on a node when it's selected by the scheduler to run the same function, it needs to be pulled from the registry before the container can be started. The measurements need to consider both, the image to be present on the node and the image to be downloaded from the registry.

Bandwidth Each node may have different network characteristics. An edge node does not have the same network connection to a centralized registry or data storage than a cloud node. Therefore, the measurements are first run without any bandwidth restrictions. Furthermore, three different bandwidth presets have been defined to represent different network characteristics, listed in Table 4.4. These bandwidth characteristics are individually applied for the inbound and outbound traffic of the node during the measurements using the Linux utility program *traffic control (tc)*, which in turn uses several Linux kernel modules.

Startup/Execution For each placement two different times are of interest: *a*) the time it takes from the placement decision until the container is running on the selected node (i.e. the startup time), and *b*) the time it takes for one single execution of the already deployed function (i.e. the execution time).

The measurements are implemented using PyTest[13] and are run on an additional *Raspberry Pi 3B+*. One iteration of these measurements records the times for each permutation

[13] `http://pytest.org/` (visited on Nov. 26, 2019)

of the described variables, resulting in 288 measurements per iteration. In total, 104 iterations were conducted over a period of 77 days, totaling in 29,952 individual measurements [Ras19a].

4.5 Metadata

Our specialized scheduler needs to consider additional metadata of the FaaS functions as well as the nodes in order to make informed placement decisions once the functions are to be deployed in the mixed cloud-edge cluster. This metadata includes information regarding *a*) the requested memory and CPU of the function, *b*) the available memory and CPU of the nodes, *c*) the hardware capabilities accelerating a function, *d*) the hardware capabilities available on the nodes, *e*) the locations of the nodes, and *f*) the amount of data functions are downloading and uploading.

K8s already provides means of defining the requested memory and CPU resources of containers and monitors them on the nodes respectively. As described in Section 4.1.1, K8s allows annotating it's objects with *labels*. These labels can be used to annotate the K8s objects with any additional metadata needed. There is no formal schema for these labels, but there is a prevailing best-practice:

$$< topic > . < domain > / < key >:< value >$$

For example, *app.kubernetes.io/name: myservice* defines that the annotated object belongs to the application *myservice*. The metadata for our scheduler is defined in compliance with this best-practice.

OpenFaaS already allows settings both, the memory and CPU resource requirements as well as additional labels, on function deployments. The labels supported by our scheduler, their usage, as well as our automated capability detection and labeling mechanisms are described in detail in Chapter 5.

4.6 Scheduler

The main contribution of this thesis is a scheduler specialized for FaaS functions in mixed cloud-edge clusters. It is plugged into the K8s cluster and becomes responsible for assigning nodes to the functions at the time of their deployment. In order to make those specialized decisions, the previously described metadata is used by custom *predicate*- and *priority functions*.

First, the *predicate functions* are excluding nodes which are violating hard constraints, for example if they do not have as many available memory or CPU as requested by the function.

Then, the *priority functions* are calculating individual scores for each node, for example they assign a higher score for nodes which provide a hardware capability which accelerates

the function's execution. Each score is then multiplied by it's priority function's configured weight.

Finally, the scores are added up and the node with the highest score is selected for the function to be deployed on. Chapter 5 describes the complete algorithm, the specific predicate- and priority functions, as well as their implementation in detail.

4.7 Simulation

For the purpose of the execution of the optimization and the thorough evaluation of our contributions, the scheduling algorithm needs to be executed a vast amount of times. The execution on our testbed is neither fast enough nor flexible enough for this purpose, making a simulation environment indispensable.

To simulate different cluster settings it is necessary to define *a*) the different nodes with all their important hardware capabilities, *b*) the connection characteristics between all those nodes, and *c*) the workload which is deployed onto the simulated cluster during the whole simulation.

The implementation of the simulation environment is explained in Section 5.7.

4.8 Optimization

Section 4.6 briefly outlines the algorithm of the scheduler. The scores calculated by the different priority functions are multiplied by their configured weight. For this reason, these weights do have a direct impact on the placement decisions of the scheduler.

Finding the correct weights is not trivial though, because their selection is an instance of the MOP as described in Section 2.5. But the MOP is a well-known problem with prominent solving algorithms. Therefore we are utilizing one of these algorithms to find near-pareto-optimal solutions for the weights in respect of the different objectives. The execution of such an algorithm comprises a vast number of executions, thus making it necessary to use a simulation environment. Once the solutions are found, they can be used to make an informed decision on the configuration of the scheduler.

A thorough description of the optimization strategy can be found in Chapter 6.

CHAPTER 5

Skippy Scheduler

This chapter focuses on the main contribution of this thesis, the *Skippy*[1] scheduler. First, Section 5.1 gives a brief overview of our scheduler and how it fits into the K8s architecture. Section 5.2 describes the default K8s scheduler in detail, whose reconstruction serves as a baseline of our evaluation in Chapter 7 and forms the basis of the Skippy scheduler. Thereafter, Section 5.3 explains the modifications and specifics of the Skippy scheduler. Section 5.4 describes the role and inner structure of the Skippy daemon. In order to use those components, the FaaS platform OpenFaaS has been slightly modified. Section 5.5 explains these changes. In Section 5.6, the different components are deployed and their integration into the FaaS platform and the K8s cluster is demonstrated. Finally, Section 5.7 outlines the implementation of the simulation environment, which is essential for the optimization and evaluation.

5.1 Overview

The architecture of K8s and the role of the scheduler has been briefly outlined in Section 2.4.4. It continuously watches for unassigned pods, searches for the most suitable node to assign it to, and creates the assignment. Therefore, the scheduler is an essential component in each K8s cluster without which no workload could actually be deployed. K8s allows a pod to define the scheduler which should take care of its placement on a feasible node. This becomes a powerful feature, as it allows us to run our Skippy scheduler alongside the default scheduler. Then, the Skippy scheduler handles only the function placements while the default scheduler takes care of any other pods. Theoretically, a scheduler can run on any node in the K8s cluster, but due to the extensive communication between the scheduler and the kube-apiserver, it is usually placed on the same (master)

[1] It is common for the names of components in the K8s ecosystem to contain nautical references. A *skipper* is a person who is in command of a boat, i.e. the current captain in charge. The skipper has command over the whole crew. Furthermore, the file ending of python modules usually is *py*.

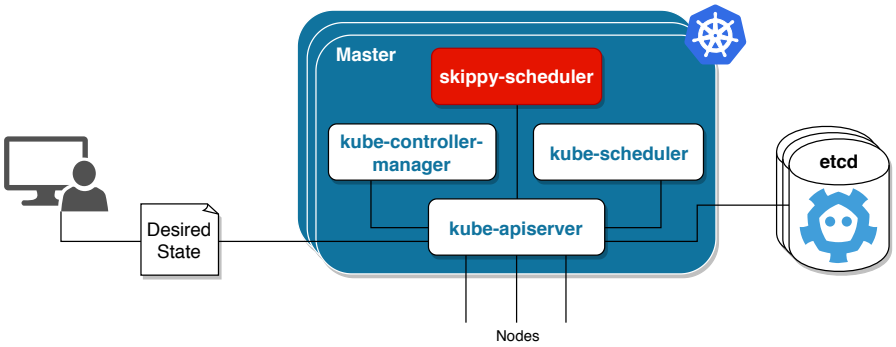


Figure 5.1: Skippy scheduler in the K8s architecture

node. Figure 5.1 shows how the Skippy scheduler integrates into the K8s architecture. The Skippy scheduler is built upon the default K8s scheduler. It reuses its basic control loop and core concept on filtering and scoring the nodes. The following section explains the inner workings of the default scheduler, and thereby the foundation of the Skippy scheduler.

5.2 Default Scheduler

The default K8s scheduler – kube-scheduler – is installed by default on each master node. This scheduler has been recreated in Python[2] as a foundation for the Skippy scheduler and subsequently to act as a baseline for the evaluation. In this section, its essential – and therefore recreated – parts are presented.

Algorithm 5.1 illustrates the control loop of the default K8s scheduler. Once an unassigned pod is in the queue, it decides if it is responsible, and schedules the pod on the most suitable node. Afterwards the loop starts over and the scheduler either picks the next unassigned pod from the queue or waits until a new one arrives.

Algorithm 5.1: Control loop of the scheduler

Result: Unplaced pods are assigned to nodes

```
1 while true do
2 |   pod ← wait for next unassigned pod;
3 |   if pod wants this scheduler then
4 |   |   schedule pod;
5 |   end
6 end
```

In order to find the most suitable node, the scheduler first calculates the amount of nodes which should be considered. Section 5.2.1 describes in detail how this amount

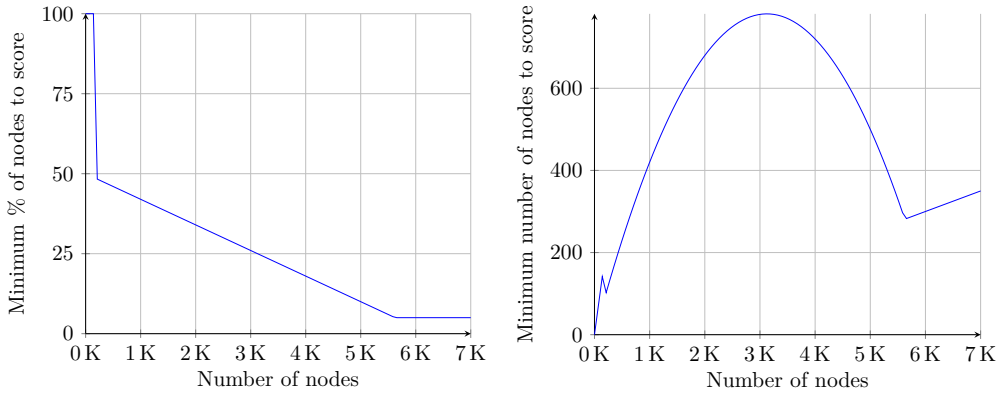
[2] `https://www.python.org/` (visited on Nov. 26, 2019)

is calculated. Then, it iterates over the nodes and checks for each node if it passes all configured *predicate functions* until the calculated number of feasible nodes is reached. Predicate functions are hard constraints for scheduling decisions and are explained in detail in Section 5.2.2. Afterwards, the feasible nodes are individually scored. Each configured *priority function* first scores all feasible nodes. In order to avoid dominating priority functions, the scores for each priority function are normalized by transforming them on a scale from 0 to 10. Section 5.2.2 describes the different priority functions in detail. Finally, the different scores for each node are summed up and the pod is assigned to the node with the highest score. Algorithm 5.2 shows the pseudocode of the algorithm.

Algorithm 5.2: Scoring algorithm of the scheduler

```
   Input : pod
   Result: Suggested node for unassigned pod
 1 nodes ← getAvailableNodes();
 2 numOfNodes ← length(nodes);
 3 numOfNodesToFind ← calcNumOfNodesToFind(numOfNodes);
 4 for node in nodes do
 5 |   for predicateFunction in predicateFunctions do
 6 |   |   if node passes predicateFunction then
 7 |   |   |   feasibleNodes.add(node);
 8 |   |   |   if length(feasibleNodes) ≥ numOfNodesToFind then
 9 |   |   |   |   break;
10 |   |   |   end
11 |   |   end
12 |   end
13 end
14 for priorityFunction in priorityFunctions do
15 |   for node in feasibleNodes do
16 |   |   score node with priorityFunction;
17 |   end
18 |   normalize the scores over all nodes;
19 |   multiply each score with the weight of priorityFunction;
20 |   add the score to the node's overall score;
21 end
22 assign pod to node with highest overall score;
```

As stated in Section 2.4.4, the scheduler solely communicates with the cluster through the *kube-apiserver*. It provides all contextual information – like newly arrived pods, the available nodes, and their available resources – and executes the actual binding of the pod to the node.



(a) Minimum percentage of nodes to score (b) Minimum number of nodes to score

Figure 5.2: Minimum nodes to score of the default K8s scheduler

5.2.1 Number of Nodes to Score

The execution of each configured priority function on each node is computationally expensive. Part of the scheduling logic is a heuristic to reduce the number of nodes that need to be scored. To find the most suitable node for a pod, the scheduler initially determines how many of the feasible nodes it should consider for scoring. This algorithm has been initially introduced in K8s version 1.12 (by using a static percentage of nodes), and has been refined recently in version 1.16. Equation (5.2) shows the function to calculate the number of feasible nodes to score as of version 1.16, where x is the number of nodes in the cluster.

$$p(x) = 0.5 - \frac{x}{12500} \tag{5.1}$$

$$f(x) = \begin{cases} x, & \text{if } p(x) * x < 100. \\ x * 0.05, & \text{if } p(x) * x \geq 100 \wedge p(x) < 0.05. \\ x * p(x), & \text{otherwise.} \end{cases} \tag{5.2}$$

Once the cluster has more than 100 feasible nodes, the percentage of nodes to consider for scoring decreases linearly from 50% to 5%, where it hits the lower limit. Figure 5.2a shows the linear decrease of the percent of nodes to score, while Figure 5.2b shows the course of the absolute number of nodes to score.

This approach mitigates the problem of scheduler performance degradation in clusters with a high amount of nodes. On the other hand, when only considering a subset of all available nodes, there is a risk of not considering a potentially better suiting node for the current pod, hence decreasing the placement quality. This risk may be low when considering highly homogeneous clusters, like typical cloud infrastructures, but gets higher the more heterogeneous the devices in a cluster are.

5.2.2 Predicate Functions

The predicate functions are used to enforce hard constraints. Even though the default K8s scheduler implements several predicate functions (e.g., to enforce cloud platform specifics, mandatory volume mounts, or advanced scheduling features like affinities or taints and tolerances) the only necessary predicate function for our use-case is the *PodFitsResourcesPred*.

PodFitsResourcesPred

This predicate function ensures that a node, which is about to be scored, has enough resources to cope with the containers in the pod. For this reason, the predicate function iterates over the pod's containers and sums up their CPU and RAM requirements respectively. If a pod's container does not specify its resource requirements, 0.1 CPU cores and 200 MB of RAM are assumed. If the node does not have enough of each resource, it is dropped by the predicate function. Algorithm 5.3 shows the pseudo-code of the *PodFitsResourcesPred*.

Algorithm 5.3: PodFitsResourcesPred

Input : pod
Input : node
Result: Nodes with insufficient available resources are discarded
1 allocatableCPU ← allocatable CPU for node;
2 allocatableRAM ← allocatable RAM for node;
3 requestedCPU ← 0;
4 requestedRAM ← 0;
5 **for** *container in pod's list of containers* **do**
6 requestedCPU $\overset{+}{\leftarrow}$ requested CPU for container or 0.1;
7 requestedRAM $\overset{+}{\leftarrow}$ requested RAM for container or 200;
8 **end**
9 **return** requestedCPU $\leq$ allocatableCPU $\wedge$ requestedRAM $\leq$ allocatableRAM;

5.2.3 Priority Functions

Priority functions act as soft constraints, and rate how well the pod fits on the node considering the priority functions specific purpose. Therefore, priority functions are used to define soft constraints and to ensure that the cluster converges to a desirable state. Just as with the predicate functions, the default scheduler implements numerous priority functions. For our use-case, the *BalancedResourcePriority* and the *ImageLocalityPriority* are important and have been recreated.

BalancedResourcePriority

This priority function favors nodes with a balanced resource usage rate. It calculates the difference between the CPU and the RAM fraction of capacity, and prioritizes nodes based on how close these two fractions are to each other. Algorithm 5.4 contains the pseudo-code of the *BalancedResourcePriority*.

Algorithm 5.4: BalancedResourcePriority

```
Input : pod
Input : node
Result: Score how well a pod balances the resources on the node
1  allocatableCPU ← allocatable CPU for node;
2  allocatableRAM ← allocatable RAM for node;
3  requestedCPU ← 0;
4  requestedRAM ← 0;
5  for container in pod's list of containers do
6  |   requestedCPU +← requested CPU for container or 0.1;
7  |   requestedRAM +← requested RAM for container or 200;
8  end
9  fractionCPU ← requestedCPU / allocatableCPU;
10 fractionRAM ← requestedRAM / allocatableRAM;
11 if fractionCPU ≥ 1 ∨ fractionRAM ≥ 1 then
12 |   return 0
13 end
14 diff ← |fractionCPU - fractionRAM|;
15 return (1 − diff) ∗ maxPriority;
```

ImageLocalityPriority

As mentioned in Section 4.3.4, the container images may become large in size (for example, the compressed ARM64 model training image of our workflow has 555 MB). If such an image is not present on the node, it may take a considerable amount of time to pull the image from the registry. Therefore, the *ImageLocalityPriority* favors nodes who already have the images of the containers of the pod on their local storage. In order to mitigate the ”node heating problem” – i.e. pods get assigned to the same nodes over and over again due to image locality – it is necessary to scale the score based on the ratio of the amount of nodes the image has already spread to and the total number of nodes. Additionally, a minimum and maximum value is introduced and the score is transformed on the scale from 0 to 10 (the maximum score of the priority function). Algorithm 5.5 shows the pseudo-code of the priority function.

Algorithm 5.5: ImageLocalityPriority

Input : pod
Input : node
Result: Score how many of a pod's container images are present on the node

```
1  imageScore ← 0;
2  for container in pod's list of containers do
3  |  if container's image is present on node then
4  |  |  spread ← (number of nodes the container's image is present on) / (total number of nodes);
5  |  |  imageScore +← size of the container's image * spread;
6  |  end
7  end
8  if imageScore < 23 MB then
9  |  imageScore ← 23 MB;
10 else if imageScore > 1000 MB then
11 |  imageScore ← 1000 MB;
12 end
13 return maxPriority * (imageScore − 23 MB)/(1000 MB − 23 MB);
```

5.3 Skippy Scheduler

In comparison to the default scheduler, the Skippy scheduler replaces an existing priority function and adds several new priority functions. These priority functions are highly domain-specific and leverage additional metadata as described in Section 4.5. Section 5.4 outlines how node-specific metadata is gathered automatically. Section 5.6.2 shows how the additional metadata is applied to functions at deploy time. Furthermore, Skippy uses a bandwidth graph in order to facilitate bandwidth specific priority functions. We assume this bandwidth graph to be available, as the bandwidth monitoring is out of the scope of this thesis. Details on how we simulated the bandwidth graph can be found in Section 5.7.

5.3.1 Number of Nodes to Score

As stated in Section 5.2.1, scoring only a subset of the available nodes may decrease the placement quality especially in heterogeneous clusters. Therefore, the Skippy scheduler does not pre-select a certain subset of the nodes but always scores all feasible nodes in the cluster.

5.3.2 Predicate Functions

No additional predicate functions have been implemented for the Skippy scheduler yet. The default scheduler's only predicate function – the *PodFitsResourcesPred* described in Section 5.2.2 – is reused.

5.3.3 Priority Functions

The core of the Skippy scheduler are the additional domain-specific priority functions. While the default scheduler's *BalancedResourcePriority* is reused, the *ImageLocalityPriority* is replaced with the *LatencyAwareImageLocalityPriority*. The remaining three new priority functions – the *LocalityTypePriority*, the *DataLocalityPriority*, and the *CapabilityPriority* – are mere additions. Their individual purpose and operating principles are as follows.

LatencyAwareImageLocalityPriority

As mentioned in the preceding section, the container images may have a significant size. While the *ImageLocalityPriority* takes this size into account, it has no means to respect the individual network connectivity between the node to score and the container registry. This is where the *LatencyAwareImageLocalityPriority* comes in. It replaces the *ImageLocalityPriority* and scores the nodes based on how long it would take them to download all necessary images from the container registry. The lesser the time to download the necessary images, the higher the node's score. It is the first priority function listed here to normalize the scores in a subsequent step, in order to know the upper boundary for the normalization. The pseudo-code for the priority function is shown in Algorithm 5.6.

LocalityTypePriority

Based on a predefined mapping, this priority function simply scores nodes according to their locality. The locality is part of a node's metadata. Section 5.4 describes how this data is gathered. In our scenario, we only have two different locality types: *a*) cloud, and *b*) edge. Since the execution on edge devices is beneficial – as described in Section 2.1.1 – devices on the edge are scored with the maximum value while devices in the cloud are scored with 0. Algorithm 5.7 shows the pseudo-code of the *LocalityTypePriority* function.

DataLocalityPriority

In contrast to the *LocalityTypePriority*, the *DataLocalityPriority* uses the metadata of a function which is to be deployed. Based on the metadata, it calculates the amount of data transferred from the next data storage node to the function during its execution. Analogous to the *ImageLocalityPriority*, it calculates the amount of time this data needs to be transferred and normalizes the scores in a subsequent step. The lesser the time to transfer the data to and from the function, the higher the node's score. The pseudo-code for this priority function is shown in Algorithm 5.8.

Algorithm 5.6: LatencyAwareImageLocalityPriority

Result: Score how long it takes for a node to download a pod's images

Function *score*:
- **Input :** pod
- **Input :** node
- size $\leftarrow 0$;
- **for** *container in pod's list of containers* **do**
  - **if** *container's image is not present on node* **then**
    - size $\overset{+}{\leftarrow}$ size of the container's image;
  - **end**
- **end**
- bandwidth $\leftarrow$ get bandwidth from node to registry;
- time $\leftarrow \frac{\text{size}}{bandwidth}$;
- **return** time;

Function *normalize*:
- **Input :** pod
- **Input :** nodeScores
- minScore $\leftarrow$ find minimum in nodeScores;
- maxScore $\leftarrow$ find maximum in nodeScores;
- **for** *nodeScore in nodeScores* **do**
  - normalizedNodeScores $\overset{\text{append}}{\longleftarrow}$ maxPriority $* \frac{\text{minScore}+\text{maxScore}-\text{nodeScore}}{\text{maxScore}}$;
- **end**
- **return** normalizedNodeScores;

Algorithm 5.7: LocalityTypePriority

Result: Score how long it takes for a node to download a pod's images

Input : pod

Input : node

priorityMapping $\leftarrow \{edge : \text{maxPriority}, cloud : 0\}$;

localityType $\leftarrow$ get *locality.skippy.io/type* label of node;

if *priorityMapping contains localityType* **then**
- **return** priorityMapping[localityType];

else
- **return** 0;

end

Algorithm 5.8: DataLocalityPriority

Result: Score how long it takes for a node to transfer runtime data

1 **Function** *score*:
  Input : pod
  Input : node
2 size ← 0;
3 **if** *pod has 'data.skippy.io/receives-from-storage' label* **then**
4 size $\xleftarrow{+}$ parsed size of pod's *data.skippy.io/receives-from-storage* label;
5 **end**
6 **if** *pod has 'data.skippy.io/sends-to-storage' label* **then**
7 size $\xleftarrow{+}$ parsed size of pod's *data.skippy.io/sends-to-storage* label;
8 **end**
9 nextStorageNode ← find nearest storage from node;
10 bandwidth ← get bandwidth from node to nextStorageNode;
11 time ← $\frac{\text{size}}{bandwidth}$;
12 **return** time;
13 **Function** *normalize*:
  Input : pod
  Input : nodeScores
14 minScore ← find minimum in nodeScores;
15 maxScore ← find maximum in nodeScores;
16 **for** *nodeScore in nodeScores* **do**
17 normalizedNodeScores $\xleftarrow{\text{append}}$ maxPriority $*$ $\frac{\text{minScore}+\text{maxScore}-\text{nodeScore}}{\text{maxScore}}$;
18 **end**
19 **return** normalizedNodeScores;

CapabilityPriority

As mentioned in Section 2.2.3, the development of domain-specific hardware will be accelerated in the near future. This specific hardware can potentially have a major impact on the execution time of a suitable task. A classic example is the usage of GPUs to accelerate the training of ML models. The purpose of the *CapabilityPriority* is the facilitation of such hardware capabilities available on some of the nodes in the cluster. For this reason, it uses metadata of both, the function which is to be deployed and the node to score. If the function is labeled with one or more capabilities, nodes which are labeled with some or all of those capabilities are favored. These labels being key-value pairs allow it to differ between the certain versions of capabilities. For example, *capability.skippy.io/nvidia-cuda: "10"* is not compatible with *capability.skippy.io/nvidia-cuda: "9"*. Algorithm 5.9 shows the pseudo-code of the *CapabilityPriority*.

Algorithm 5.9: CapabilityPriority

Result: Score how 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* ∧ *values are equal* **then**
7 score $\overset{+}{\leftarrow}$ 1
8 **end**
9 **end**
10 **return** score;
11 **Function** *normalize*:
  Input : pod
  Input : nodeScores
12 maxScore ← find maximum in nodeScores;
13 **for** *nodeScore in nodeScores* **do**
14 normalizedNodeScores $\overset{\text{append}}{\longleftarrow}$ maxPriority $* \frac{\text{nodeScore}}{\text{maxScore}}$;
15 **end**
16 **return** normalizedNodeScores;

5.4 Skippy Daemon

The priority functions described in the previous section depend on the metadata of the functions and the nodes. While a function's metadata is maintained by its developer, the node's metadata needs to be populated and maintained too. Since there may be a high fluctuation of devices in a mixed cloud-edge infrastructure, the node label management has been automated by introducing the *Skippy daemon*. This small component is deployed on the cluster as a K8s *daemonset*, i.e. an instance of the component is deployed on every single node in the cluster. Once running on the node, the Skippy daemon autonomously populates and maintains its node's metadata.

The following capabilities are currently detected by the daemon:

NVidia GPU If the node provides an NVidia GPU, the *capability.skippy.io/nvidia-gpu* label is set.

CUDA If the node provides a Compute Unified Device Architecture (CUDA)[3] installation, the version string is read, parsed, and set as value for the label

[3]CUDA is a platform and API by NVidia allowing developers to use their GPUs for general purpose processing.

capability.skippy.io/nvidia-cuda.

Locality Type If the node does not have a *locality.skippy.io/type* label, it is set to *locality.skippy.io/type: edge*. This means that new devices are assumed to be edge devices since we expect a higher fluctuation of these devices and the maintainers of cloud devices are expected to have a higher degree of control over the cluster (and therefore can make sure to label new cloud devices responsibly when adding them to the cluster).

Storage Nodes If the node is running a MinIO pod, it is marked as a storage node by setting the *data.skippy.io/storage-node* label.

5.5 OpenFaaS Modifications

Section 5.1 describes how the Skippy scheduler is deployed next to the default K8s scheduler. This allows us to use our scheduler only for a very specific subset of deployments, namely the FaaS function deployments. But for these pods to be scheduled by the Skippy scheduler, they must declare it at deploy time. Since the deployment is created by OpenFaaS – more specifically a component called *faas-netes* – a slight modification was necessary to specify the Skippy scheduler as scheduler for each deployment created by OpenFaaS.

In addition to this modification, for OpenFaaS to be deployable in our heterogeneous testbed, it was necessary for each of its components to be available as Docker manifest list referencing images for each of the CPU architectures used in our cluster. Even though in Section 4.1.2 we discovered that OpenFaaS claims to be compatible with all of the CPU architectures in our cluster (AMD64, ARM, and ARM64), the corresponding images have not been maintained in quite some time and they were not covered by their build automation. Therefore it was necessary to cross-compile each of the components and publish them on DockerHub[4].

The patch for both of these changes can be found in Appendix B.

5.6 Integration

Once all components are ready, they can finally be deployed and integrated with each other in the cluster.

[4] `https://hub.docker.com/repository/docker/alexrashed/gateway`
`https://hub.docker.com/repository/docker/alexrashed/openfaas-operator`
`https://hub.docker.com/repository/docker/alexrashed/faas-idler`
`https://hub.docker.com/repository/docker/alexrashed/prometheus`
`https://hub.docker.com/repository/docker/alexrashed/faas-netes`
`https://hub.docker.com/repository/docker/alexrashed/faas-netes-skippy`
`https://hub.docker.com/repository/docker/alexrashed/queue-worker` (all visited on Nov. 26, 2019)

5.6.1 Deploying Skippy

The Skippy scheduler is deployed just as any other deployment in a K8s cluster. Listing 5.1 shows the (truncated) K8s deployment specification for the Skippy scheduler. Line 33 shows how the individual weights for the priority functions are defined during the deployment as arguments to the scheduler's container.

```
1  ...
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: skippy-scheduler
6    namespace: kube-system
7    labels:
8      app: skippy-scheduler
9  spec:
10   replicas: 1
11   selector:
12     matchLabels:
13       app: skippy-scheduler
14   template:
15     metadata:
16       labels:
17         app: skippy-scheduler
18     spec:
19       # Make sure it's executed on the master node
20       nodeSelector:
21         node-role.kubernetes.io/master: ""
22       # Tolerate NoExecute taints
23       tolerations:
24       - key: "node-role.kubernetes.io/master"
25         operator: "Exists"
26
27       # Set the service name
28       serviceAccountName: skippy-scheduler
29
30       containers:
31       - name: skippy-scheduler
32         image: alexrashed/skippy-scheduler:1.0
33         args: ["--debug", "--weights", "[2.6923720489533647, 2.698796313141462,
       5.964581321182787, 4.720991059190411, 9.122271417461349]"]
34         ...
```

Listing 5.1: Skippy scheduler deployment specification

As mentioned in Section 5.4, the Skippy daemon is deployed as a K8s daemonset. In order to allow the daemon to determine on which node it is running and to detect this node's certain hardware capabilities, the daemonset's deployment specification needs allow the daemon to access this information as shown in Listing 5.2.

```
1  ...
2  apiVersion: extensions/v1beta1
3  kind: DaemonSet
4  metadata:
5    name: skippy-daemon-daemonset
6    namespace: kube-system
7  spec:
8    template:
```

```
9     metadata:
10      annotations:
11        scheduler.alpha.kubernetes.io/critical-pod: ""
12      labels:
13        name: skippy-daemon
14    spec:
15      priorityClassName: system-node-critical
16      serviceAccountName: skippy-daemon
17      tolerations:
18        ...
19      containers:
20      - image: alexrashed/skippy-daemon:0.6
21        name: skippy-daemon
22        volumeMounts:
23          - name: cuda
24            mountPath: /usr/local/cuda
25          - name: nvidia-smi-local-bin
26            mountPath: /usr/local/bin/nvidia-smi
27          - name: nvidia-smi-bin
28            mountPath: /usr/bin/nvidia-smi
29        env:
30          - name: NODE_NAME
31            valueFrom:
32              fieldRef:
33                fieldPath: spec.nodeName
34      volumes:
35        - name: cuda
36          hostPath:
37            path: /usr/local/cuda
38        - name: nvidia-smi-local-bin
39          hostPath:
40            path: /usr/local/bin/nvidia-smi
41        - name: nvidia-smi-bin
42          hostPath:
43            path: /usr/bin/nvidia-smi
44  ...
```

Listing 5.2: Skippy scheduler deployment specification

5.6.2 Deploying a Function

Finally, all components are in place for the FaaS functions with its additional metadata to be deployed. Listing 5.3 shows the (truncated) function specification file for the training workflow step which can be deployed using OpenFaaS' CLI tool *faas-cli*. While Line 12 defines the RAM resource requirement of 1 GB, all other function specific metadata is defined in the Lines 14-17 according to the metadata schema described in Section 4.5.

```
1   provider:
2     name: openfaas
3     gateway: https://openfaas.thesis.git-init.org
4   functions:
5     ml-wf-2-train:
6       lang: Dockerfile
7       skip_build: true
8       image: alexrashed/ml-wf-2-train:0.33
9       environment:
10        ...
11      requests:
```

```
12          memory: 1Gi
13    labels:
14      capability.skippy.io/nvidia-cuda: "10"
15      capability.skippy.io/nvidia-gpu: ""
16      data.skippy.io/receives-from-storage: 209Mi
17      data.skippy.io/sends-to-storage: 1500Ki
```

Listing 5.3: OpenFaaS function specification including Skippy metadata

Figure 5.3 shows the interaction between the different components in the cluster on such a function deployment. The following steps are executed between the user's deployment command and the function's container being started in a node:

1. Initially, the user initiates the deployment of the function by sending the specification to the *openfaas-gateway*.

2. The *openfaas-gateway* uses our modified component – *faas-netes* – to create the deployment specification for the newly deployed function and sending the specification to the *kube-apiserver*.

3. Since the deployment specification defines the Skippy scheduler as the pod's scheduler, the unassigned pod is picked up by the *skippy-scheduler*.

4. The Skippy scheduler scores all nodes while taking into account all additional metadata in its custom priority functions. It selects the highest scoring node and sends the binding command to the *kube-apiserver* who takes notice of the binding in its database.

5. The *kubelet* running on the selected node realizes that a new pod has been assigned to its node and starts the pod by instructing the local container runtime to spawn all containers defined in the pod's specification.

5.7 Simulation Environment

In order to execute the optimization and to allow a thorough evaluation of our scheduler, a vast amount of task placements – and thus scheduler runs – need to be executed. As stated in Section 4.7, our testbed is neither fast enough nor flexible enough for this purpose, making a simulation environment indispensable. This simulation environment has been implemented using SimPy[5], a Python discrete-event simulation framework. For a clean separation of concern and to increase code re-usability, the cluster context has been abstracted – as described in Section 5.7.1. Before starting a simulation, the following cluster and deployment settings need to be defined:

[5] `https://simpy.readthedocs.io/` (visited on Nov. 26, 2019)

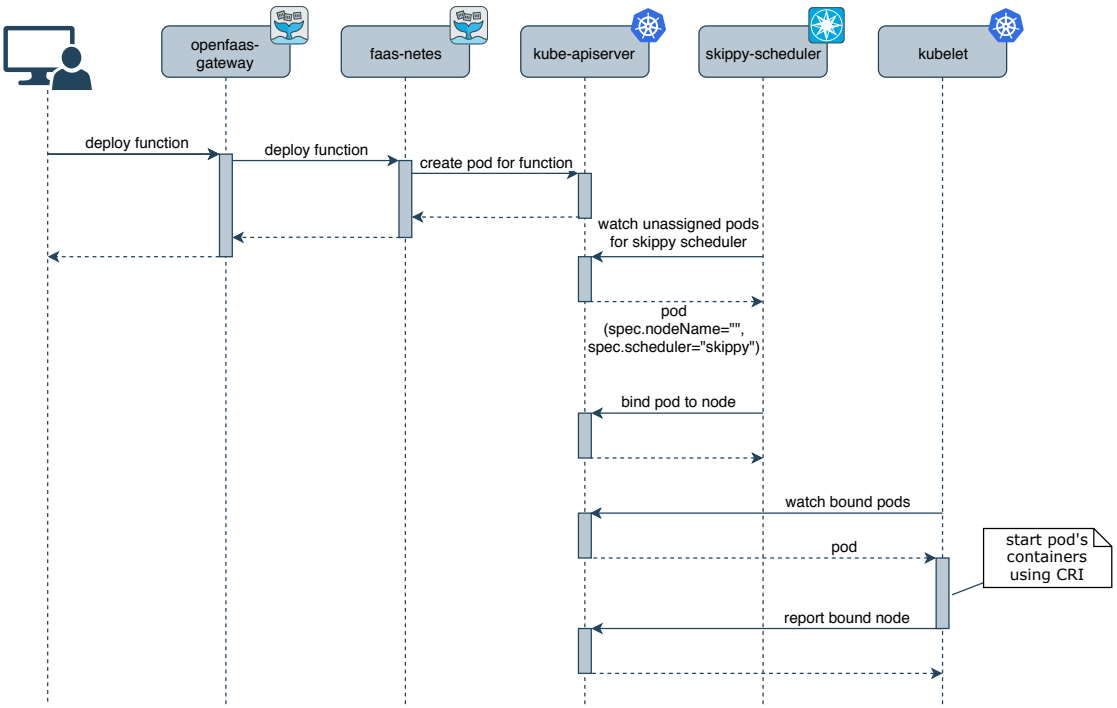


Figure 5.3: Sequence diagram of a function deployment

- The connection characteristics between all those nodes. Section 5.7.2 describes the simulated bandwidth graph.
- The tasks which are to be deployed onto the simulated cluster during the whole simulation. Section 5.7.3 explains the synthetic workload based on our scenario.
- The nodes which form the cluster. Section 5.7.4 discusses the definition of synthetic nodes for the simulation.

Once all these settings are defined, a simulation runs for a defined simulation runtime. During such a simulation run, the simulation environment calls the scheduler for as many task placements – and thereby scheduling algorithm executions – as possible. Once a scheduling decision has been taken, its quality is estimated by several oracles. These oracles are heavily based on the empirical measurements, as described in Section 4.4, as well as on the bandwidth graph. Section 5.7.5 explains the different oracles in detail.

5.7.1 Cluster Abstraction

Figure 5.4 shows a class diagram of the scheduler, its interfaces for predicate and priority functions, and most importantly the abstraction of the cluster context. This abstraction is necessary to re-use the same scheduling logic for both, the simulation and the actual deployment in our K8s cluster. When instantiating the scheduler, the

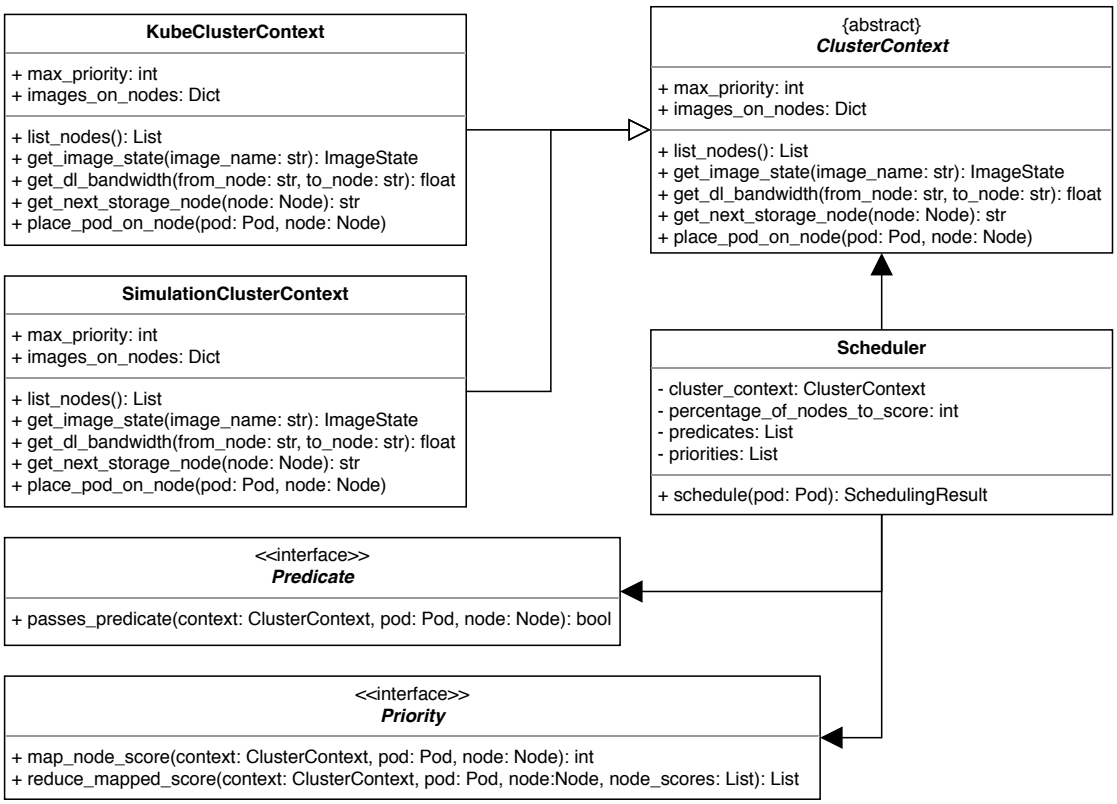


Figure 5.4: Class diagram of the Scheduler and the ClusterContext

respective *ClusterContext* for the current purpose, a list of predicate functions, and a list of tuples of priority functions with their weights are passed to the constructor. While the *KubeClusterContext* uses the *kube-apiserver* to retrieve data about the cluster and execute task placements, the *SimulationClusterContext* only uses the predefined simulation settings. This results in a high degree of code-reuse while enabling a fine grained configuration of the scheduler for each simulation run.

5.7.2 Bandwidth Graph

Other than in a homogeneous cloud infrastructure, the nodes in a mixed cloud-edge infrastructure may each have different network connectivity. To be able to take this into account in our scheduler, it is necessary to define the different bandwidths between each other – i.e. define the bandwidth graph.

As described in Section 5.7.5, the estimation of the different task placement quality metrics are based on the empirical measurements described in Section 4.4. Therefore it is necessary for assumed bandwidth values to be included in the list of bandwidth presets as defined in Section 4.4. Otherwise, if an assumed bandwidth between two nodes is not contained in the list of bandwidth presets, the according empirical measurements would

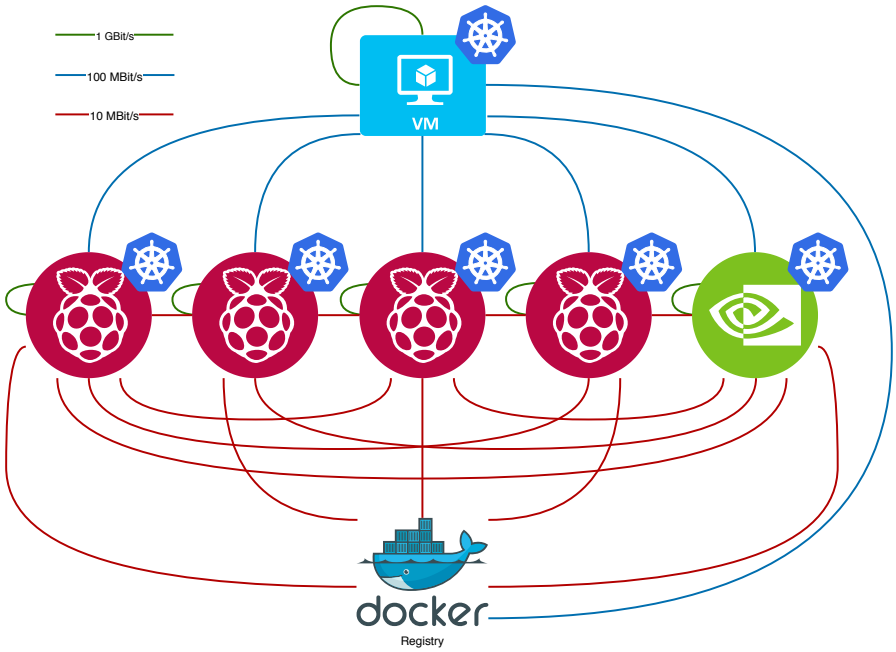


Figure 5.5: Simulated bandwidth graph

be missing.

We assume the following network connectivities between the different device types:

- Edge device to edge device: 10 MBit/s
- Edge device to cloud device: 100 MBit/s
- Edge device to container registry: 10 MBit/s
- Cloud device to edge device: 100 MBit/s
- Cloud device to cloud device: 1 GBit/s
- Cloud device to container registry: 100 MBit/s
- Loop-back for each device: 1 GBit/s

In our testbed, this results in the bandwidth graph shown in Figure 5.5.

5.7.3 Workload

Section 4.3.4 describes the different functions of our scenario workflow. The simulated workload is aligned on the typical scenario execution by iterating over the steps of the workflow in a round-robin fashion. The function metadata for each workflow step is shown in Table 5.1.

Step	Resources	Labels
ml-wf-1-pre	memory: 100Mi	'data.skippy.io/receives-from-storage': '12Mi' 'data.skippy.io/sends-to-storage': '209Mi'
ml-wf-2-train	memory: 1Gi	'capability.skippy.io/nvidia-cuda': '10' 'capability.skippy.io/nvidia-gpu': '' 'data.skippy.io/receives-from-storage': '209Mi' 'data.skippy.io/sends-to-storage': '1500Ki'
ml-wf-3-serve		'data.skippy.io/receives-from-storage': '1500Ki'

Table 5.1: Metadata of simulated workflow functions

Device Type	Resources	Labels
VM	cpus: 4 memory: 7.79Gi	'beta.kubernetes.io/arch': 'amd64' 'locality.skippy.io/type': 'cloud'
Raspberry Pi 3B+	cpus: 4 memory: 975.62Mi	'beta.kubernetes.io/arch': 'arm' 'locality.skippy.io/type': 'edge'
NVidia Jetson TX2	cpus: 4 memory: 7.67Gi	'beta.kubernetes.io/arch': 'arm64' 'capability.skippy.io/nvidia-cuda': '10' 'capability.skippy.io/nvidia-gpu': '' 'locality.skippy.io/type': 'edge'

Table 5.2: Metadata of simulated cluster nodes

5.7.4 Nodes

For the simulation, we synthesized the same devices as we used in our testbed. Table 5.2 shows the different device types with their characteristics. The resources available on each device type have been extracted from the node metadata in our testbed using the K8s CLI *kubectl*. The percentage distribution of the different devices have a great influence on the characteristics of the cluster. For the evaluation of our contributions, in Section 7.1 we defined a set of cluster configurations – i.e. the size of a simulated cluster together with the percentage distributions of the different devices therein.

5.7.5 Oracles

Since we are not actually deploying and executing the tasks in a real cluster, we cannot measure the different metrics necessary to rate the quality of the task placements, but have to estimate them. After each simulated task placement, oracle functions are executed to create informed estimations of the different metrics. Each of the following five oracles is responsible for the estimation of one specific metric.

Startup Time

The startup time estimation is solely based on the empirical measurements described in Section 4.4. Initially, it loads all data sets of the measurements of the startup time. When estimating a simulated placement, it first determines the following properties:

- Which workflow function has been scheduled?
- Which bandwidth does the selected node have to the container registry?
- Is the workflow function image already present on the node?
- On which of the device types has the function been placed on?

Then, the data set is filtered for matching measurements and a random sample is selected as estimation.

Execution Time

The execution time estimation works analogous to the startup time estimation, but instead of the startup time it loads the data set for the execution time measurements.

Bandwidth Usage

In order to estimate the bandwidth usage, the following aspects are taken into account:

- Which workflow function has been scheduled?
- Is the workflow function image already present on the node?
- Does the workflow function define metadata concerning data being transferred?

The estimated bandwidth usage is the sum of data defined in the function metadata and the size of the workflow function image (if it's not yet present on the selected node).

Edge Resource Utilization

The resource utilization of the edge devices is estimated using the node's locality type, the resource capabilities of the nodes, and the resource requirements of the placed function. For nodes with locality types other than "edge", the resource utilization 0 is estimated, as we only want to consider edge devices in this metric. For edge devices, the resource utilization estimation is the sum of the fraction of the node's CPU and the fraction of the node's RAM needed by the function.

Cost

The cost estimation is based on the AWS Lambda pricing of function executions on machines with 1 GB RAM[6]. If the function is placed on a cloud device, the cost is estimated by multiplying this price – 0.00001667 USD per second – with the estimated function execution time. If the function is placed on an edge device, no costs are estimated.

[6] `https://aws.amazon.com/lambda/pricing/` (visited on Nov. 26, 2019)

CHAPTER 6

Optimization

Section 5.6.1 shows that the Skippy scheduler is configured by defining the weight for each priority function. As mentioned briefly in Section 4.8, finding these weights is not trivial. Each weight has a direct impact on the placement decisions of the scheduler, and thus on the placement quality. Section 6.1 defines the placement quality as well as the different objectives it is based on.

These objectives represent coarse-grained goals, that are approached by finding the best set of weights for fine-grained scores. These scores are obtained by the different priority functions, each responsible to focus on one specific aspect. This abstraction of not implementing the objectives directly in priority functions, but defining the objectives outside of the scheduler, is beneficial in several ways:

- It allows defining objectives without changing the source code of the scheduler, avoiding recompilation and redeployment of the scheduler.
- It ensures a clean separation of concerns. Each priority function defines one distinctive fine-grained aspect. The coarse-grained objectives are then approximated by combining these fine-grained priority functions.
- It would still be necessary to run the optimization, as it cannot be guaranteed that priority functions which are directly representing a specific objective would have the desired influence. The optimization also ensures that a selected set of weights is not dominated by another set of weights – i.e. it ensures that there is no other set of weights which is better in at least one objective but not worse in the others.

When considering these conflicting objectives, the selection of the weights of the priority functions of the scheduler is an instance of the MOP as described in Section 2.5. Fortunately, MOP is a well-known problem with prominent solving algorithms. Section 6.2

describes the implementation of our optimization approach. It explains the utilized MOP solving algorithm, the employed framework implementing the algorithm, as well as the calculation of the different objective's estimation values.

Finally, Section 6.3 describes how to interpret and use the results of the optimization with the Skippy scheduler.

6.1 Placement Quality

A cluster scheduler's resource allocations should optimize the cluster state towards different objectives, as described in Section 4.6. These objectives are often conflicting and the priority of each individual objective differs among different clusters. Therefore, it is not possible to define the placement quality as a single value. Instead, each of the different objective values defines one dimension of the placement quality of a single task placement decision. For our use-case we consider the following four objectives – i.e. dimensions of the placement quality.

6.1.1 Bandwidth Usage

The bandwidth usage defines how much traffic a placement causes in the cluster. This includes pulling function images from the container registry, but also runtime data transferred during a single function execution. This objective is to be *minimized*, since each data transfer needs a share of the limited networking resources in a cluster and blocks the execution of the function.

6.1.2 Task Execution Time

The TET defines the time it takes from the scheduler's placement decision to the end of the function's execution. This may include pulling a function's image (if it is not already stored locally on the selected node), starting the container, and the actual execution of the function's code. Since the TET defines the latency of a single request, it should be *minimized*.

6.1.3 Cost

As stated in Section 2.2.2, the billing model in FaaS platforms is fundamentally different from traditional cloud computing billing models. The cost of a function execution is based on the execution time in the cloud, i.e. the amount of time certain resources are allocated for the execution. In a mixed cloud-edge infrastructure, and especially in the predictive maintenance scenario described in Section 4.2, edge devices are added by the customer in addition to the cloud resources and are running on-premises of the customer. Since the acquisition and the operation of these edge devices are paid by the customer, the function execution on those devices is not being considered in our pricing model. Therefore, the cost is defined by the execution time of a function placement on cloud

resources multiplied with the price per time interval. Obviously, the costs should be *minimized*.

6.1.4 Edge Resource Utilization

The edge device resource utilization is defined as the sum of the ratios of resources (CPU and RAM) a function placed on an edge device is using. While the pricing of cloud resources is based on the execution time, the edge devices are operated on-premises and should be utilized as much as possible. Therefore the edge device resource utilization should be *maximized*.

6.2 Implementation

There are several different algorithms aiming to find solutions to the MOP. One of the most prominent ones is the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [Deb+02]. It is considered a standard solver, is well tested, efficient, and only has a single parameter (the number of generations) [Deb14; ED18; KCS06].

An implementation of this algorithm is provided by Platypus[1], a Python framework focusing on evolutionary algorithms solving the MOP. A benchmark of the implementation of six algorithms of the Platypus framework has shown that NSGA-II has the best performance.

For each optimization, we execute the Platypus framework's NSGA-II implementation with 10,000 generations. This explains the need for a simulation environment, as stated in Section 5.7, since each optimization run comprises 10,000 generations, each executing a single simulation. During one of those simulation runs, the simulation environment calls the scheduler for as many task placements as possible.

After each simulation run, the different objective values are estimated as described in Section 6.2.1. These estimated objectives are then used by the NSGA-II implementation in order to decide on the variation of the priority weights for the next generation.

Due to this high amount of generations, and the number of computations in each generation (i.e. simulation), the optimization is computationally expensive but also highly parallelizable. The executions were executed on a machine with 16 CPUs.

For the evaluation of our contributions described in Chapter 7, the optimization was executed once for every cluster configuration described in Section 7.1. The optimization for a specific cluster configuration took about 22 minutes for the testbed cluster (6 nodes), about 1 hour and 53 minutes (± 5 minutes) for clusters comprising 100 nodes, and about 4 hours 15 minutes (± 8 minutes) in clusters containing 1000 nodes.

[1] `https://github.com/Project-Platypus/Platypus` (visited on Nov. 26, 2019)

6.2.1 Objectives

After each generation of the NSGA-II, it is necessary to estimate the quality of the simulated placements concerning each objective defined in Section 6.1. For this estimation, we rely on the different metrics estimated by the oracles described in Section 5.7.5. During a simulation run, these oracles are executed on each task placement and their resulting metrics are added to the simulation as metadata. The objective estimations utilizing those metrics are implemented as follows.

Bandwidth Usage is estimated by calculating the mean value of the bandwidth metric over all task placement decisions of the simulation run.

TET is estimated by summing up the mean value of the startup time metric and the mean value of the execution time metric over all task placement decisions of the simulation run.

Cost is estimated by calculating the mean value of the cost metric over all task placement decisions of the simulation run.

Edge Resource Utilization is estimated by calculating the mean value of the edge device resource utilization metric over all task placement decisions of the simulation run.

6.3 Usage

The result of the optimization is a set of solutions. A solution contains exactly one weight for each priority function. As described in Section 2.5, these solutions are non-dominated solutions as near to the pareto-optimal front as possible. Each of these solutions therefore constitute a specific trade-off between the different objectives. A cluster administrator can now choose between those trade-offs, select the respective solution, and use these weights to configure the Skippy scheduler in order to optimize the placement decisions in the cluster towards these trade-offs.

In a future work, this selection could be assisted by utilizing multi-criteria decision-making (MCDM) methods. For example, Rezaei presents such a decision-making method in [Rez15; Rez16]. This would allow the administrator to select the most suitable optimization solution by identifying the most important objective and the least important objective. Then all other objectives are rated on a scale from 0 to 9 compared to those two objectives. Finally, the best suiting solutions could be selected based on those comparisons using the best-worst method described in [Rez15].

CHAPTER 7

Evaluation

In this chapter we present the evaluation of our implemented scheduler as well as our optimization approach. We have conduct several empirical and simulated experiments in order to validate the performance of our scheduler in terms of its quality of placement, its speed, and its scalability. First, Section 7.1 outlines the specifics of our simulation environment. Then, Section 7.2 shows the results of the deployment of our workflow functions in our testbed for the different scheduler configurations. Section 7.3 describes the results of the evaluation of the simulated experiments concerning the placement quality. Finally, Section 7.4 discusses the performance and scalability characteristics of our solution.

7.1 Evaluation Environment

Section 6.1 defines the placement quality and the different objectives it is based on. The metrics used to assess the placement quality of each task placement decision are defined in Section 5.7.5. Section 6.2.1 shows the method of calculating the different objectives – namely the TET, the bandwidth usage, the costs, and the edge device resource utilization – based on those metrics. Each of those four different objective values defines one dimension of the placement quality for the specific task placement decision.

In order to maximize the placement quality of our scheduler towards a specific trade-off between the different objectives, the scheduler can be configured by defining individual weights for each priority function (as described in Section 5.6.1).

But the importance of each priority function, and therefore its optimal weight, may differ for each cluster configuration. Therefore, a set of cluster configurations which are representative for our scenario have been defined. As stated in Section 5.7.4, a cluster configuration defines the size of a cluster and the percentage distributions of the different device types in that cluster. The three different device types – VM, NVidia Jetson

Configuration name	% per device model			# of nodes	Simulation runtime (s)
	VM	TX2	RPi		
testbed	12.50	12.50	75.00	6	40
equal_100	33.33	33.33	33.33	100	1000
equal_1000	33.33	33.33	33.33	1000	1000
50p_cloud_100	50.00	25.00	25.00	100	1000
50p_cloud_1000	50.00	25.00	25.00	1000	1000
edge_100	14.29	28.57	57.14	100	1000
edge_1000	14.29	28.57	57.14	1000	1000
cloud_100	66.67	16.67	16.67	100	1000
cloud_1000	66.67	16.67	16.67	1000	1000

Table 7.1: Simulated cluster configurations

TX2 (TX2), and Raspberry Pi 3B+ (RPi) – are described in detail Section 4.3.1 and Section 5.7.4. Table 7.1 shows the characteristics of each configuration. Afterwards, the optimization, simulation and evaluation have been conducted for each of those simulated clusters with its distinctive characteristics.

Testbed The *testbed* configuration represents our testbed's cluster node setup as described in Section 4.3.1. Our scheduler cannot find a suitable node once the cluster is over-provisioned. As described in Section 5.7, during a simulation run the simulation environment calls the scheduler for as many task placements as possible using our synthesized workload defined in Section 5.7.3. In our simulation environment this small cluster of only 6 nodes becomes fully provisioned in just over 40 seconds. Afterwards the scheduling fails, since there are no resources left in the cluster for the pending tasks. Therefore, for the evaluation in the testbed, the simulation runtime is limited to 40 seconds to avoid failing scheduling attempts in our data.

Equal *equal_100* and *equal_1000* represent configurations where the device types are equally distributed. When transferred to our scenario, a company operates twice as many edge devices within the factory as the amount of VMs available in the cloud.

50% *50p_cloud_100* and *50p_cloud_1000* represent configurations where the device locality types are equally distributed. When transferred to our scenario, a company operates just as many edge devices within the factory as the amount of VMs available in the cloud.

Edge *edge_100* and *edge_1000* represent configurations where the majority of the devices are edge devices. When transferred to our scenario, the majority of resources used by a company are edge devices within the factory. Only a small amount of VMs in the cloud are available.

Cloud *cloud_100* and *cloud_1000* represent configurations where the majority of the resources are in the cloud. When transferred to our scenario, the majority of resources used by a company are in the cloud. Only a small amount of edge devices are available.

For each of these distributions, with the exception of the testbed, two different configurations have been created. Configurations with 100 nodes over a simulation period of 1000 seconds represent clusters which will become highly utilized. On the other hand, those configurations with 1000 nodes simulate clusters with a considerable amount of available resources, allowing the scheduler to easily choose between the different nodes.

7.2 Empirical Experiments

In order to demonstrate the execution of our solution in an actual cluster, and to illustrate the impact of our scheduler on the scheduling decisions, we conducted an empirical experiment. We deployed the three different functions of our scenario's workflow in the testbed's idle FaaS platform. The functions are deployed one after another, in the order of their execution in the workflow. The experiment was repeated for each of the following scheduler configurations:

1. *Default Scheduler:* The recreated default scheduler without any weight adjustments.
2. *Non-Optimized Skippy Scheduler:* The Skippy scheduler with its domain-specific scoring functions without any weight adjustments.
3. *Optimized Skippy Scheduler:* The Skippy scheduler with its domain-specific scoring functions with weights optimized for the TET in the testbed-specific optimization preset.

The individual result for each configuration is outlined in its respective subsection below. The logging output of the different experiments are enclosed in Appendix C.

7.2.1 Default Scheduler

As described in Section 5.2, the recreated default scheduler only comprises the predicate function *PodFitsResourcesPred* and the two priority functions *BalancedResourcePrio* and *ImageLocalityPrio*. Table 7.2 shows the individual predicates and priorities of the default scheduler during the deployment of our workflow functions. The predicate function – *PodFitsResourcesPred* – results in a boolean value depicting if the node is feasible to run the pod. The different priority functions each result in a score for each node. The higher the score, the better the node suits the pod concerning the individual priority function's purpose. Each score is multiplied by the priority function's configured weight. In case of the default scheduler, all priority functions are weighted by 1 – i.e. there is no individual

	cloud1	tegra1	pi1	pi2	pi3	pi4	
1-pre	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	9	9	9	9	9	9	BalancedResourcePrio
	0	0	0	0	0	0	ImageLocalityPrio
	9	9	9	9	9	9	Sum
2-train	✓	✓	✗	✗	✗	✗	PodFitsResourcesPred
	8	8					BalancedResourcePrio
	0	0					ImageLocalityPrio
	8	8					Sum
3-serve	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	9	9	8	8	8	8	BalancedResourcePrio
	0	0	0	0	0	0	ImageLocalityPrio
	9	9	8	8	8	8	Sum

Table 7.2: Results of the predicate and priority functions for each workflow function placement of the default scheduler in the testbed

weighting. For each placement, the scheduler finally selects the node which passes the predicate functions and has the highest sum of priority function scores. Due to the fact that none of the images are present at any of the nodes, the *ImageLocalityPrio* does not have any impact. As the cloud VM node has the most resources, it is favored by the *BalancedResourcePrio* for each of the three placements.

7.2.2 Non-Optimized Skippy Scheduler

Compared to the default scheduler, Skippy replaces the *ImageLocalityPrio* and adds several domain-specific priority functions. A detailed description of the different functions can be found in Section 5.3. When executed with the – non-optimized – default configuration, each priority function is weighted with 1. Table 7.3 shows the individual predicates and priorities of the non-optimized Skippy scheduler during the deployment of our workflow functions. All three functions are placed on edge devices. This can be explained by the the smaller function image sizes for ARM devices – as listed in Section 4.3.4 – which has an impact on the *LatencyAwareImageLocalityPrio*, and the capability provided by the NVidia Jetson TX2 which impacts the *CapabilityPrio* score. Therefore the preprocessing and the serving functions are placed on Raspberry Pi devices while the training function is placed on the GPU accelerated NVidia Jetson TX2.

7.2.3 Optimized Skippy Scheduler

In comparison to the experiment described in the previous section, the last empirical experiment uses the Skippy scheduler with optimized priority weights. Therefore the optimization results with the lowest TET in the testbed configuration – as described in

	cloud1	tegra1	pi1	pi2	pi3	pi4	
1-pre	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	9	9	**9**	9	9	9	BalancedResourcePrio
	8	8	**10**	10	10	10	LatencyAwareImageLocalityPrio
	0	10	**10**	10	10	10	LocalityTypePrio
	10	0	**0**	0	0	0	DataLocalityPrio
	0	0	**0**	0	0	0	CapabilityPrio
	28	28	**29**	29	29	29	Sum
2-train	✓	✓	✗	✗	✗	✗	PodFitsResourcesPred
	8	**8**					BalancedResourcePrio
	10	**9**					LatencyAwareImageLocalityPrio
	0	**10**					LocalityTypePrio
	10	**0**					DataLocalityPrio
	0	**10**					CapabilityPrio
	28	**37**					Sum
3-serve	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	9	9	7	**8**	8	8	BalancedResourcePrio
	8	8	10	**10**	10	10	LatencyAwareImageLocalityPrio
	0	10	10	**10**	10	10	LocalityTypePrio
	10	0	0	**0**	0	0	DataLocalityPrio
	0	0	0	**0**	0	0	CapabilityPrio
	27	27	27	**28**	28	28	Sum

Table 7.3: Results of the predicate and priority functions for each workflow function placement of the non-optimized Skippy scheduler in the testbed

Section 7.1 – have been used. The optimization resulted in the priority weights shown in Figure 7.1.

Analogous to the two previous experiments, Table 7.4 shows the individual predicates and priorities of the optimized Skippy scheduler during the deployment of our workflow functions. In this specific experiment, the priority scores result in the same placement decisions as with the non-optimized Skippy scheduler. However, the evaluation of the simulated experiments – described in Section 7.3 – outlines the advantages of the optimized weights compared to the defaults.

7.3 Placement Quality

In order to evaluate the placement quality of our scheduler, the testbed does not have the necessary cluster size. Instead we used the simulation environment – described in Section 5.7 – and conducted one simulated experiment for each cluster configuration defined in Section 7.2.

	cloud1	tegra1	pi1	pi2	pi3	pi4	
1-pre	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	24.23	24.23	**24.23**	24.23	24.23	24.23	BalancedResourcePrio
	21.59	21.59	**26.99**	26.99	26.99	26.99	LatencyAwareImageLocalityPrio
	0.00	59.65	**59.65**	59.65	59.65	59.65	LocalityTypePrio
	47.21	0.00	**0.00**	0.00	0.00	0.00	DataLocalityPrio
	0.00	0.00	**0.00**	0.00	0.00	0.00	CapabilityPrio
	93.03	105.47	**110.87**	110.87	110.87	110.87	Sum
2-train	✓	✓	✗	✗	✗	✗	PodFitsResourcesPred
	21.54	**21.54**					BalancedResourcePrio
	26.99	**24.29**					LatencyAwareImageLocalityPrio
	0.00	**59.64**					LocalityTypePrio
	47.21	**0.0**					DataLocalityPrio
	0.00	**91.22**					CapabilityPrio
	95.74	**196.69**					Sum
3-serve	✓	✓	✓	✓	✓	✓	PodFitsResourcesPred
	24.23	24.23	18.84	**21.54**	21.54	21.54	BalancedResourcePrio
	21.59	21.59	26.99	**26.99**	26.99	26.99	LatencyAwareImageLocalityPrio
	0.00	59.65	59.65	**59.65**	59.65	59.65	LocalityTypePrio
	47.21	0.00	0.00	**0.00**	0.00	0.00	DataLocalityPrio
	0.00	0.00	0.00	**0.00**	0.00	0.00	CapabilityPrio
	93.03	105.47	105.49	**108.18**	108.18	108.18	Sum

Table 7.4: Results of the predicate and priority functions for each workflow function placement of the optimized Skippy scheduler in the testbed

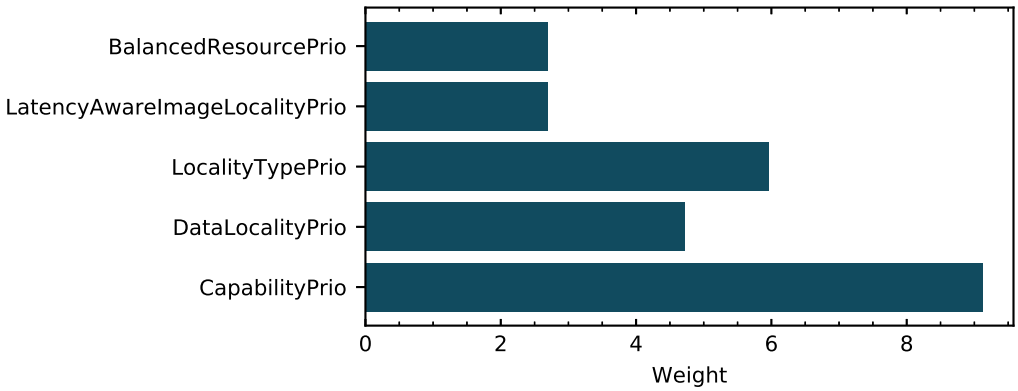


Figure 7.1: TET optimized priority weights for the testbed configuration

First, we analyzed the results and picked three different configurations to focus on:

1. *edge_1000* showing the best results of Skippy compared to the default scheduler
2. *cloud_100* showing the highest optimization benefits
3. *edge_100* showing the worst results of Skippy compared to the default scheduler

The following sections each focus on the evaluation of the results for one of these configurations. All other placement quality comparison matrix plots can be found in Appendix D. The complete datasets of the simulation together with all generated plots are available as open data [Ras19b].

7.3.1 edge_1000

In Chapter 6 we identified the following four different objectives: *a*) the TET, *b*) the bandwidth usage, *c*) the cost of running functions on cloud resources, and *d*) the resource utilization of the edge devices. Our optimization has been configured towards those four objectives. The TET, bandwidth, and cost should be minimized while the resource utilization should be maximized. As the configuration of our scheduler always impacts its performance on all different objectives, one objective cannot be analyzed without considering the performance gain or loss on each of the other objectives.

Figure 7.2 shows a matrix of line plots to visualize and compare the placement qualities of the different scheduler configurations. Each column represents one configuration of the optimized Skippy scheduler towards one objective. Each row focuses on the cumulative values for one objective. Therefore each column shows the performance of one configuration for each of the objectives while each row shows the performance of one objective across all different configurations.

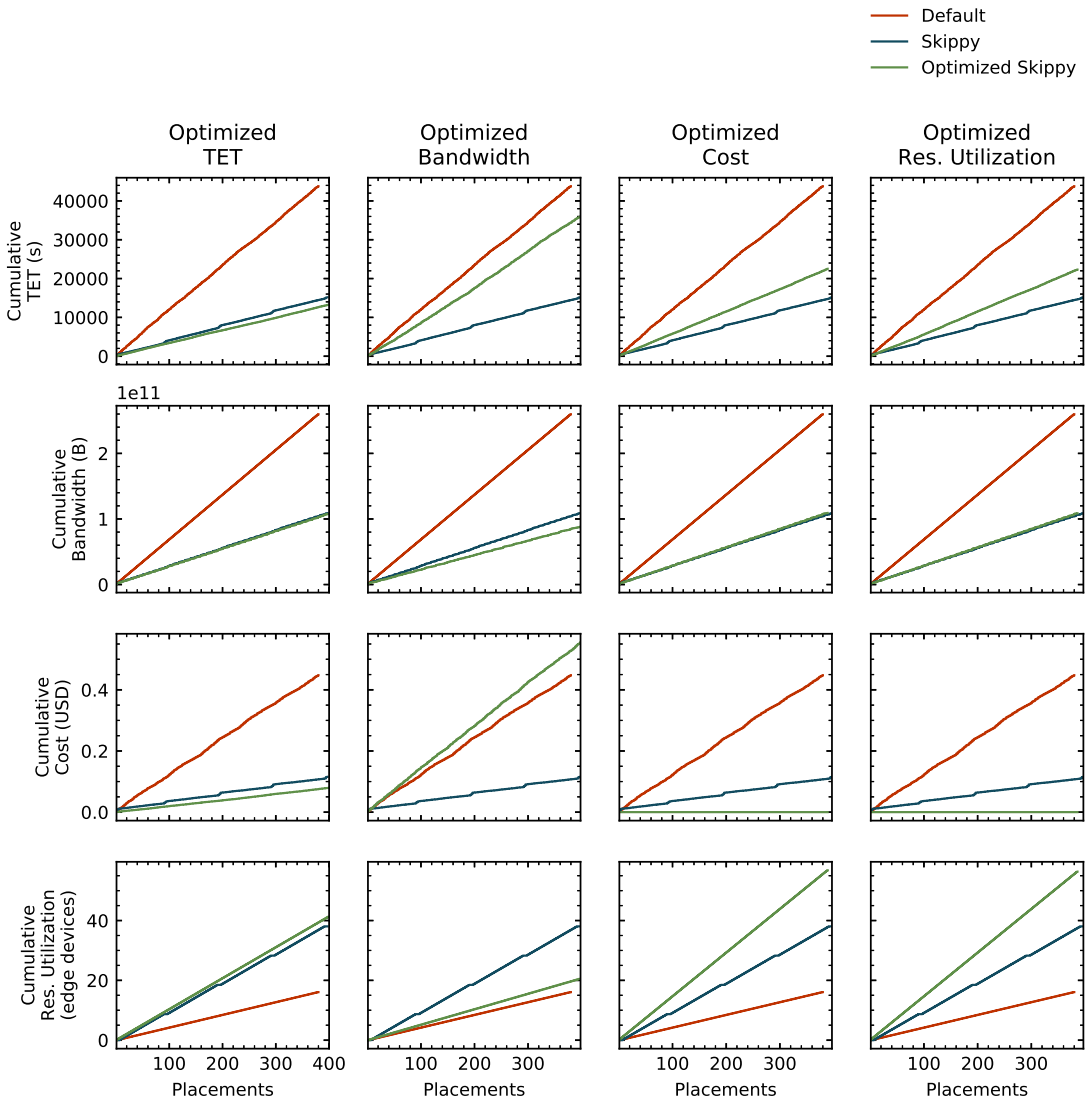


Figure 7.2: Placement quality comparison matrix plot for the *edge_1000* cluster configuration

	Mean	σ
Default	0.001179	0.001739
Skippy	0.000291	0.000639
Optimized Skippy	0.001396	0.001632

Table 7.5: Mean value and standard deviation of the cost objective for different scheduler configurations in the *edge_1000* cluster configuration when optimized towards bandwidth

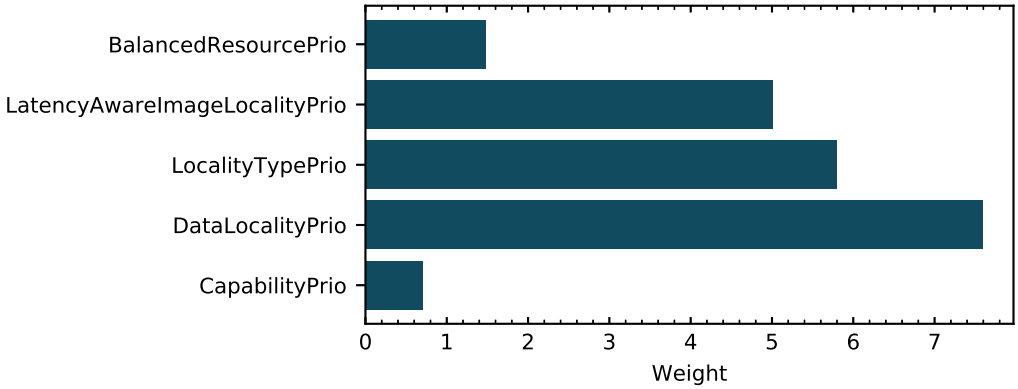


Figure 7.3: TET optimized priority weights for the edge_1000 configuration

In this cluster configuration, the non-optimized Skippy scheduler outperforms the default scheduler in each objective and optimization. The optimized Skippy scheduler, however, is outperformed by the default scheduler in one objective and optimization. It has a slightly worse cost value than the default scheduler when optimized towards lowering the bandwidth usage. Table 7.5 shows the mean and standard deviation in that case. The weights calculated by the optimization for this specific setting are visualized in Figure 7.3. As we can see, the *DataLocalityPrio* is weighted the highest. As the data store in our simulation is on a cloud node and the nodes with the highest bandwidth to the data store are preferred, more cloud nodes are selected leading to higher costs. For all other objectives and optimizations, the Skippy scheduler outperforms the default scheduler.

For the in-depth analysis of the results for each cluster configuration we focus on the TET – with the optimization also towards the TET.

Figure 7.4 visualizes the TET per workflow function image for each placement during the simulation. The results show that the Skippy scheduler generally outperforms the default scheduler in all cases. Moreover, when comparing the optimized Skippy with the non-optimized Skippy, the graphs for the second and third workflow function image illustrate the improvements due to the optimization. While the non-optimized Skippy causes some peaks in the TET, the optimized Skippy avoids those placements.

This characteristic can also be seen in Figure 7.5, which shows the boxplots for the TET

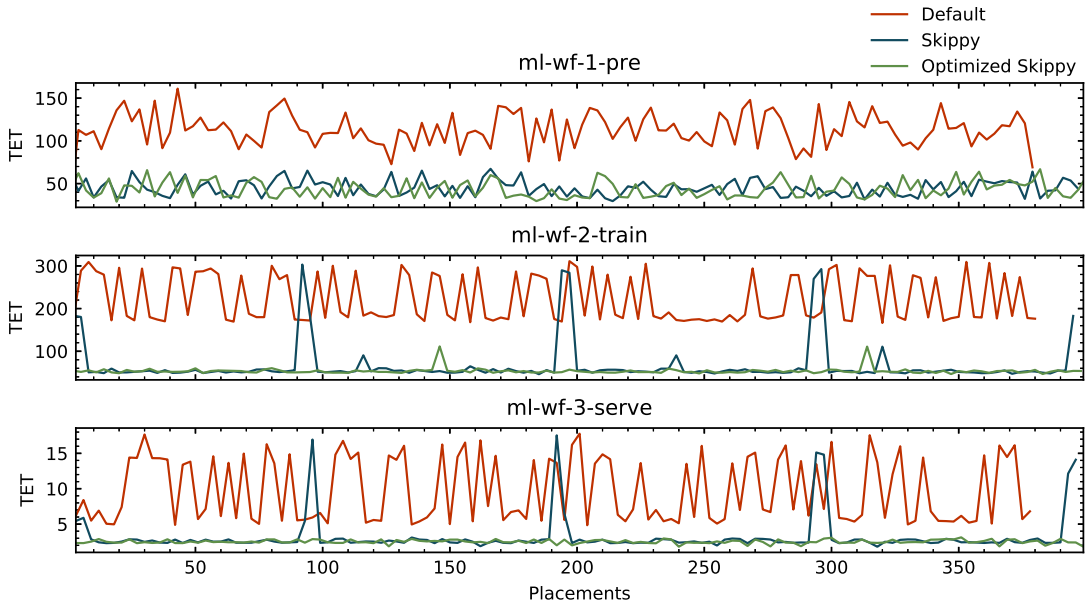


Figure 7.4: TET per image in the *edge_1000* cluster configuration over time

for each workflow function image and scheduler configuration. The optimized Skippy drastically limits the amount of outliers for the aforementioned workflow function images.

Table 7.6 lists statistics for the TET in this cluster configuration for each scheduler configuration and workflow function image. The optimization reduces the standard deviation by 84.66% from 49.93 to 7.66 seconds for the second workflow function image, and by 89.63% from 2.70 to 0.28 seconds for the third workflow function image while also decreasing the mean value. Compared to the default scheduler, the optimized Skippy scheduler reduces the mean TET for the second workflow image by 75.90% from 222.17 to 53.54 seconds while the standard deviation decreased by 86.04% from 54.89 to 7.66 seconds. Over all workflow images, when comparing the default scheduler with the optimized Skippy scheduler, the mean TET decreased by 71.17% from 115.23 to 33.21 seconds, while the standard deviation decreased by 57.77% from 93.14 to 39.33 seconds.

As stated in Section 7.1, this cluster configuration – with a high number of nodes compared to the scheduled workload – allow the scheduler to choose between lots of feasible nodes. This is where the benefits of our domain-specific scheduler are most visible. The default scheduler does not have any knowledge about the location of the devices, their data usage, or their hardware capabilities. This is aggravated by the default scheduler's approach of only scoring a subset of the feasible nodes – as described in Section 5.2.1. In this cluster configuration, containing 1000 nodes, the default scheduler therefore only scores 420 of the feasible nodes for each placement decision. The Skippy scheduler, on the other hand, scores all feasible nodes, and utilizes additional node and function metadata in its domain-specific priority functions.

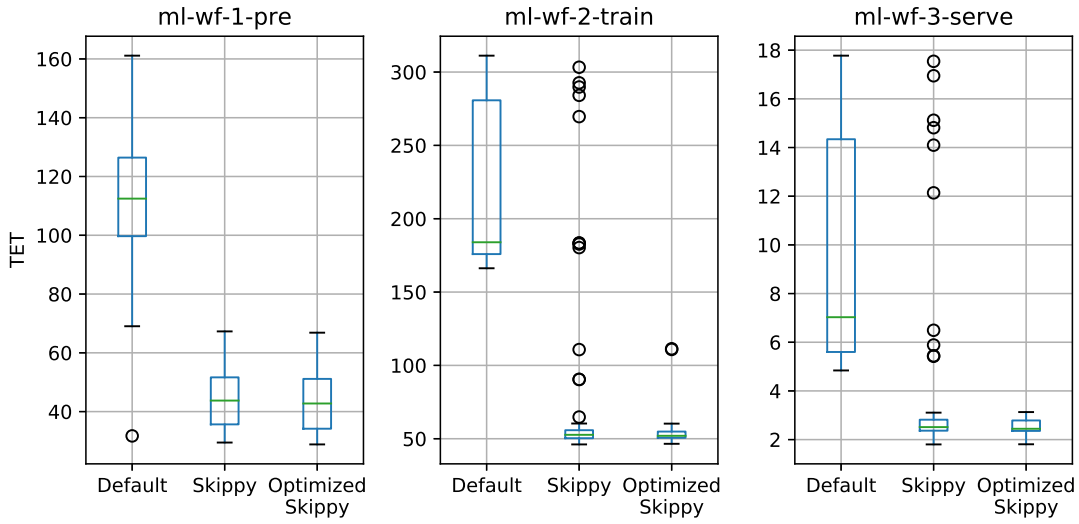


Figure 7.5: TET per image in the *edge_1000* cluster configuration

Image	Scheduler	Mean	σ
ml-wf-1-pre	Default	112.94	20.09
	Optimized Skippy	43.50	9.93
	Skippy	44.78	9.32
	Optimized Skippy	43.50	9.93
ml-wf-2-train	Default	222.17	54.89
	Skippy	66.58	49.94
	Optimized Skippy	53.54	7.66
ml-wf-3-serve	Default	9.76	4.56
	Optimized Skippy	2.51	0.28
	Skippy	3.22	2.70
	Optimized Skippy	2.51	0.28

Table 7.6: Mean and standard deviation of different scheduler configurations per workflow function image in the *edge_1000* cluster configuration when optimized towards TET

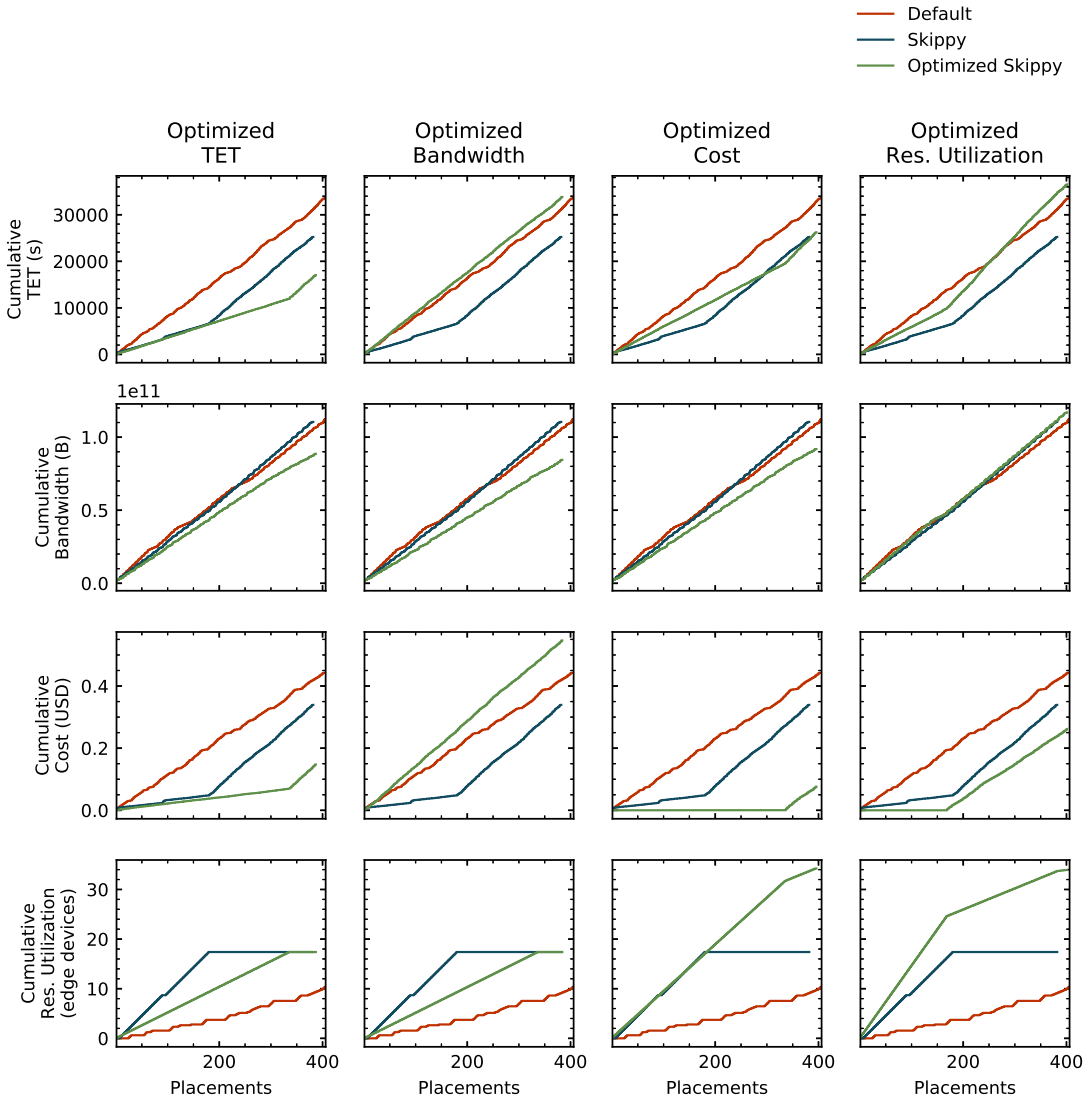


Figure 7.6: Placement quality comparison matrix plot for the *cloud_100* cluster configuration

7.3.2 cloud_100

In the previous section, we identified that the optimization caused a decrease of the standard deviation by reducing the amount of placements causing extreme values. The evaluation of the simulation experiments using the *cloud_100* cluster configuration yielded an even higher potential of our optimization efforts. Analogous to the previous evaluation, Figure 7.6 shows a matrix of line plots to visualize and compare the placement qualities of the different scheduler configurations. The plots on the main diagonal of the matrix are representing the performance of the the default scheduler and the non-optimized Skippy

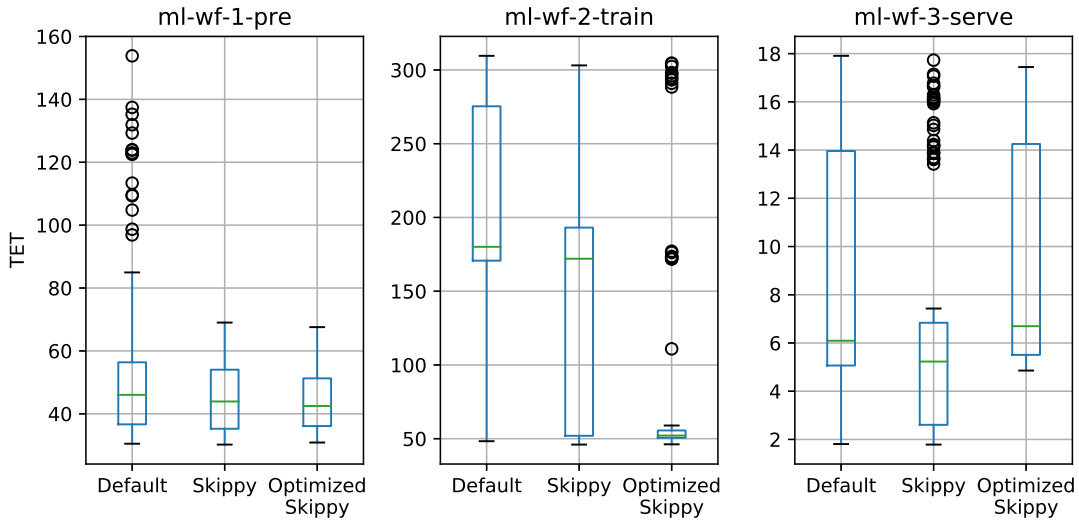


Figure 7.7: TET per image in the *cloud_100* cluster configuration

scheduler compared to the Skippy scheduler specifically optimized towards the particular objective. Compared to the previously evaluated cluster configuration, the plots on the diagonal indicate that the optimization has a higher influence on the placement quality of the Skippy scheduler.

When analyzing the boxplot for the TET for each workflow function image and scheduler with the optimization towards the TET, shown in Figure 7.7, it stands out that the median value of the TET of the second workflow image is significantly lower than those of the default scheduler and the non-optimized Skippy scheduler.

Table 7.7 shows statistics for the TET in this cluster configuration for each scheduler configuration and workflow function image. These values underline the impact of the optimization for the training workflow image, as the mean value dropped by 46.94% from 147.31 to 78.16 seconds while also reducing the standard deviation by 26.18% from 94.33 to 69.63 seconds. Compared to the default scheduler, the optimized Skippy scheduler even caused a decrease of the mean TET for the training workflow image by 57% from 186.05 to 78.16 seconds while reducing the standard deviation by 16.33% from 83.22 to 69.63 seconds. Over all workflow images, the mean TET decreased by 46.85% from 82.75 to 43.98 seconds, while the standard deviation decreased by 45.56% from 90.64 to 49.34 seconds.

This is also illustrated in Figure 7.8, which shows the TET per workflow function image for each placement during the simulation. It shows that the optimized Skippy scheduler places the training workflow step on machines which can execute the specific task very fast until more than 330 placements have been executed. At this time, the number of GPU accelerated devices is exhausted which forces the scheduler to use nodes without

Image	Scheduler	Mean	σ
ml-wf-1-pre	Default	53.98	26.74
	Skippy	44.68	10.08
	Optimized Skippy	44.44	9.28
ml-wf-2-train	Default	186.05	83.22
	Skippy	147.31	94.33
	Optimized Skippy	78.16	69.63
ml-wf-3-serve	Default	8.42	5.07
	Skippy	6.36	4.88
	Optimized Skippy	9.35	4.65

Table 7.7: Mean and standard deviation of different scheduler configurations per workflow function image in the *cloud_100* cluster configuration when optimized towards TET

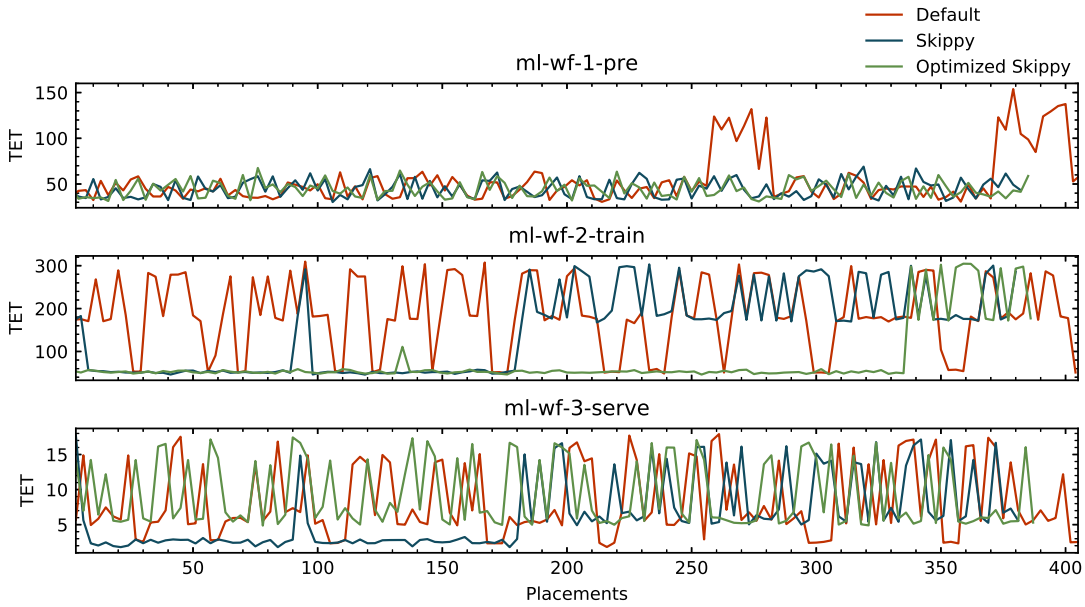


Figure 7.8: TET per image in the *cloud_100* cluster configuration over time

a matching capability. This indicates that the performance gain of the optimization in this cluster configuration is likely to decrease, as it cannot realize its full potential any further.

7.3.3 edge_100

This cluster configuration shows the least benefits for our solution. Figure 7.9 once more shows the matrix of line plots to visualize and compare the placement qualities of the

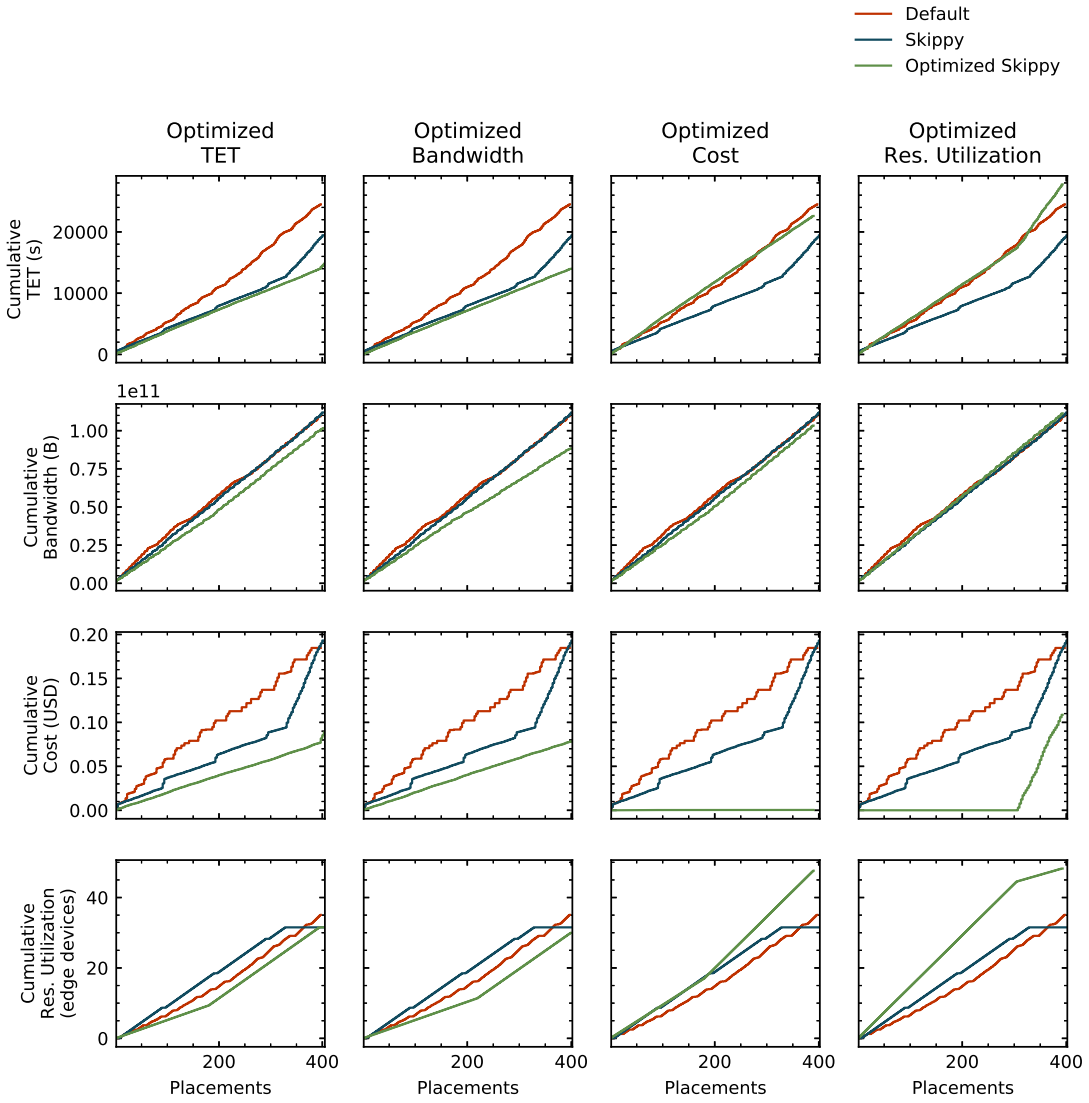


Figure 7.9: Placement quality comparison matrix plot for the *edge_100* cluster configuration

different scheduler configurations. Here most of the devices are edge devices, with a total of 28.57% of devices having a GPU. The limited number of devices also causes the default scheduler to always score all nodes. Therefore, the default scheduler is likely to pick an edge device. However, for most of the objectives and optimizations our optimized Skippy scheduler still outperforms the default scheduler.

Analogous to the previous sections, Figure 7.10 shows the TET per workflow function image for each placement during the simulation. It confirms that, in case of the training workflow image, the default scheduler is more often selecting nodes with a low TET –

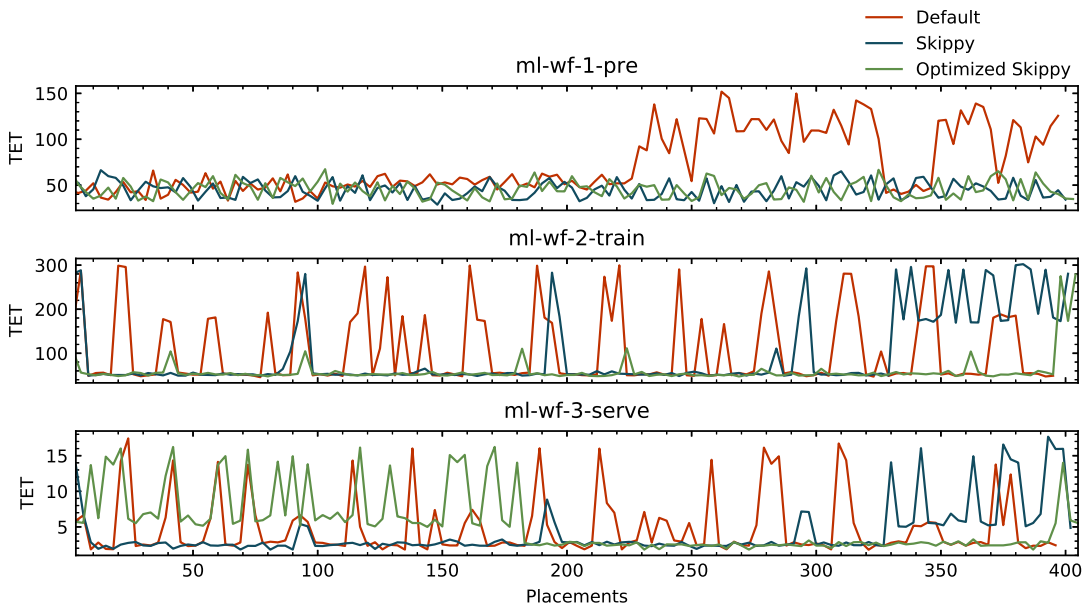


Figure 7.10: TET per image in the *edge_100* cluster configuration over time

the GPU accelerated NVidia Jetson TX2 – than in the previous cluster configurations. However, the Skippy scheduler, and especially the optimized version, are outperforming the default scheduler by further increasing the amount of training workflow image placements on that particular device type.

The boxplot in Figure 7.11, which shows the TET for each workflow function image and scheduler with the optimization towards the TET, illustrates that the median value for each image does not differ quite much among the different scheduler configurations.

As in the previous sections, Table 7.8 lists the numerical values for the mean value and the standard deviation for the TET in this cluster configuration for each scheduler configuration and workflow function image. It shows that, when comparing the default scheduler with the optimized Skippy scheduler, the mean TET for the training image dropped by 45.20% from 107.61 to 58.96 seconds with a standard deviation decrease by 63.49% from 84.86 to 30.95 seconds.

Over all workflow images, when comparing the default scheduler with the TET optimized Skippy scheduler, the mean TET decreased by 40.68% from 61.75 to 36.63 seconds, while the standard deviation decreased by 52.14% from 67.66 to 29.51.

7.3.4 Summary

Figure 7.12 shows the mean values for each of the objectives and schedulers over all simulated experiments. It shows that even the non-optimized Skippy scheduler outperforms the default scheduler on average in every objective. When analyzing the results for each

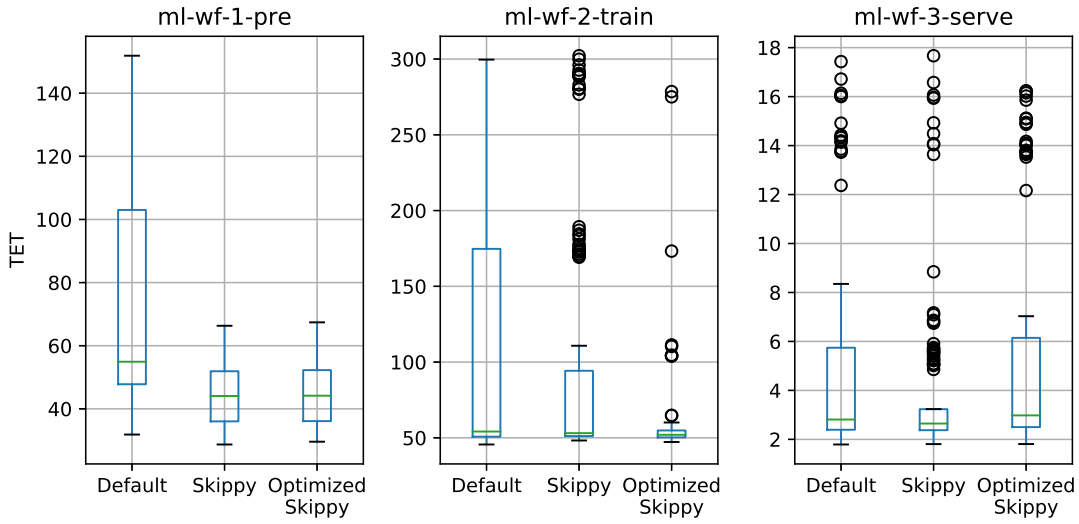


Figure 7.11: TET per image in the *edge_100* cluster configuration

Image	Scheduler	Mean	σ
ml-wf-1-pre	Default	72.71	33.57
	Skippy	44.77	9.32
	Optimized Skippy	45.46	9.59
ml-wf-2-train	Default	107.61	84.76
	Skippy	96.40	81.78
	Optimized Skippy	58.96	30.95
ml-wf-3-serve	Default	4.86	4.18
	Skippy	4.05	3.50
	Optimized Skippy	5.47	4.24

Table 7.8: Mean and standard deviation of different scheduler configurations per workflow function image in the *edge_100* cluster configuration when optimized towards TET

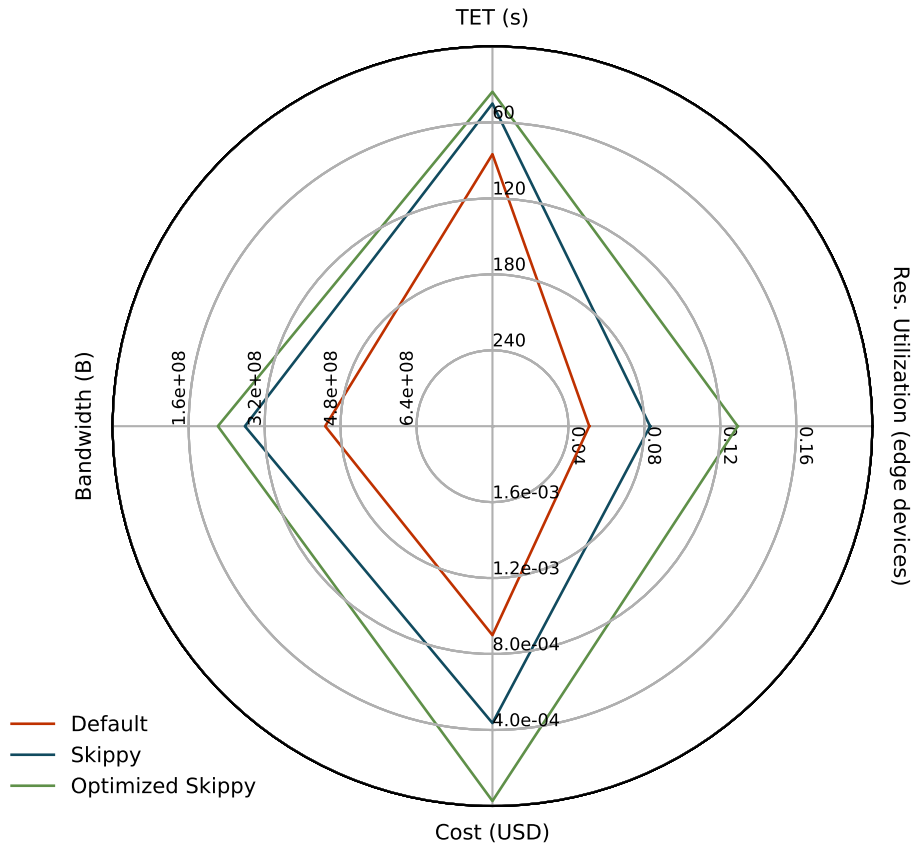


Figure 7.12: Mean values for each objective over all simulated experiments per scheduler

of the four objectives in each of the nine cluster configurations, the non-optimized Skippy scheduler causes a better mean objective value in 29 of the 36 combinations.

Once the Skippy scheduler is optimized towards the respective objective, it outperforms the default scheduler in all cases. This is visualized in the plots on the diagonal of the matrix plots in the preceding sections and in Appendix D. When comparing the mean objective values of the *default scheduler* and the *optimized Skippy scheduler* per cluster configuration, the range of performance gain for the different objectives are as follows.

Bandwidth Usage The reduction of the average bandwidth usage ranges from 19.57% (in *equal_100*) up to 67.52% (in *edge_1000*).

Cost The reduction of the average cost ranges from 82.42% (in *cloud_100*) up to 100% (in *edge_1000*). A reduction of 100% is possible if all placements are assigned to edge devices, resulting in a cost value of 0.

Resource Utilization The increase of the average edge device resource utilization ranges from 38.9% (in *edge_100*) up to 411.39% (in *cloud_1000*). In *cloud_1000*

the majority of devices are cloud devices. Therefore the default scheduler most likely picks those cloud devices resulting in an unprofitable edge device resource utilization. The Skippy scheduler can specifically choose edge devices due to the additional locality type metadata.

TET The reduction of the average TET ranges from 37.58% (in *testbed*) up to 71.18% (in *edge_1000*).

Comparing the *non-optimized Skippy scheduler* with the *optimized Skippy scheduler*, the following range of performance gain for the different objectives demonstrate the performance of the optimization approach.

Bandwidth Usage The reduction of the average bandwidth usage ranges from 18.94% (in *cloud_1000*) up to 24.26% (in *testbed*).

Cost The reduction of the average cost ranges from 78.4% (in *cloud_100*) up to 100% (in *edge_1000*, equal_1000 and 50p_cloud_1000). As described previously, a reduction of 100% is possible if all placements are assigned to edge devices, resulting in a cost value of 0.

Resource Utilization The increase of the average edge device resource utilization ranges from 50.63% (in *cloud_1000*) up to 108.7% (in *testbed*).

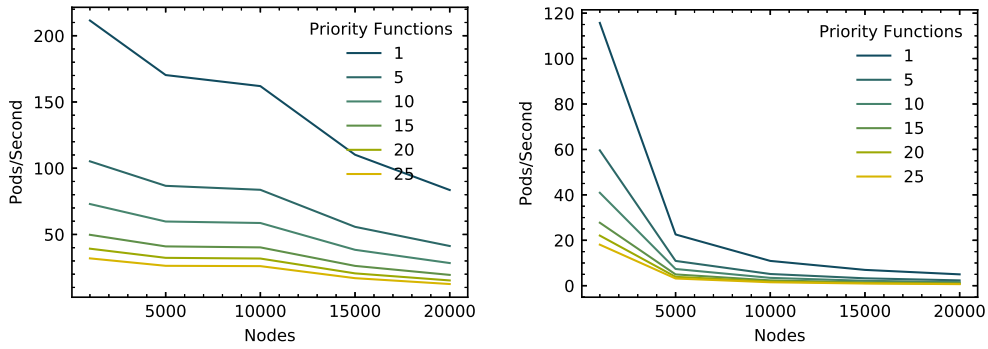
TET The reduction of the average TET ranges from 37.58% (in *testbed*) up to 71.18% (in *edge_1000*).

7.4 Scalability

The scalability of the scheduler highly depends on the number of nodes to score and the configured priority functions. As presented in Section 5.2.1, the default configuration of the default K8s scheduler has a very specific algorithm for the calculation of the minimum percentage of nodes to score. But due to the heterogeneity of the devices in a mixed cloud-edge infrastructure, Skippy scheduler always scores all available nodes in order to increase the placement quality. Using our simulation environment, we evaluated the raw scheduling throughput given different number of nodes and priority functions for both approaches.

Figure 7.13 and Figure 7.14 show the results, which are nearly identical to previously published measurements on an early prototype of the scheduler [Rau+19]. These measurements roughly match those of a recent Kubernetes performance evaluation [Den16].

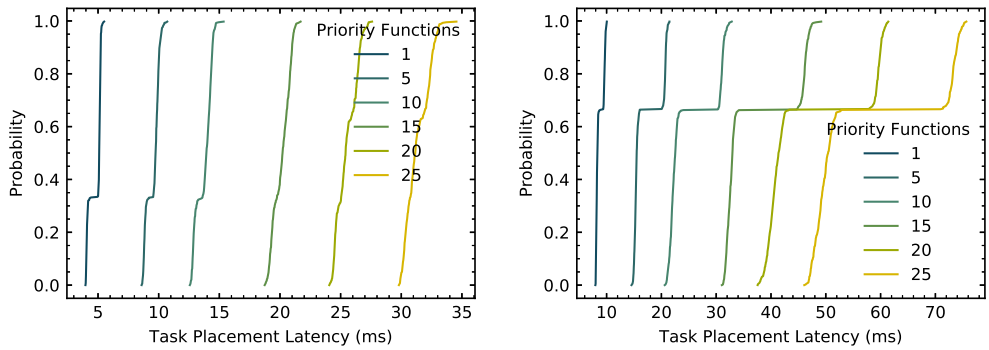
Figure 7.13 shows that the raw scheduling throughput decreases drastically with the number of nodes and priority functions. Figure 7.13a shows the results when using the default algorithm on selecting a subset of the nodes to score for each placement, while Figure 7.13b shows the results of the same measurements when scoring all available nodes – i.e. not using the aforementioned algorithm to select a subset of nodes to score.



(a) Scheduling throughput when using the default algorithm

(b) Scheduling throughput when scoring all available nodes

Figure 7.13: Raw scheduling throughput for different number of nodes and priority functions



(a) CDF of the TPL when using the default algorithm

(b) CDF of the TPL when scoring all available nodes

Figure 7.14: TPLs for different number priority functions in a cluster with 1000 nodes

As of version 1.16, K8s supports clusters with up to 5000 nodes [Kuba]. A special interest group (SIG) is working on further increasing this limit [Kubc]. While the default scheduler in a cluster of this size with 10 priority functions still has a throughput of ~ 60 pod placements per second, the Skippy scheduler's throughput with the same amount of priority functions – which scores all available nodes – drastically decreases to ~ 7 pod placements per seconds.

The cumulative distribution functions (CDFs) in Figure 7.14 clearly visualize the increase of the task placement latency (TPL) with each priority function. Our reconstruction of the default scheduler uses one predicate function and two priority functions, each described in Section 5.2. The Skippy scheduler currently uses one predicate function and four priority functions, as detailed in Section 5.3.

As our approach increases the number of priority functions, and mixed cloud-edge clusters may include huge node populations, the scheduler needs to maintain a high scheduling throughput. These results highlight the complexity of the scheduling problem, and the limitations of Kubernetes' queue-based monolithic scheduler architecture.

As describe in Section 2.3.1, alternative architectures have been proposed to cope with the complexity of the scheduling problem while maintaining a low placement latency. Since the default scheduler's approach on increasing the scheduling throughput by only scoring a subset of the available nodes has a negative impact on the placement quality in our heterogeneous cluster, in a future work a possible solution instead could be to move away from the current queue-based monolithic architecture of the scheduler towards a distributed architecture.

CHAPTER 8

Conclusion

A scheduler that assigns workloads to specific nodes is a key component for resource management in a cluster. Through experiments, we have uncovered significant limitations of such schedulers for edge computing scenarios. This thesis describes the design, implementation, and evaluation of an integrated, optimized, latency-, and capability-aware scheduler for running an FaaS platform in a mixed cloud-edge computing environment. Production grade schedulers are flexible, but they are based on the basic assumption that the cluster infrastructure is highly homogeneous. They need to be extended in order to efficiently execute workloads in (heterogeneous) mixed cloud-edge clusters. By adding soft constraints that target edge computing systems, the scheduler can be customized for this specific domain. The optimization towards high level operational goals determines weights for the low level scheduler constraints, thereby enabling cluster administrators to optimally configure the scheduler towards their specifically selected trade-off between those goals. The individual contributions of this thesis can be summarized as follows.

8.1 Contributions

Skippy Scheduler The Skippy scheduler is the core contribution of this thesis. Based on the recreated Kubernetes default scheduler, it implements several domain-specific priority functions leveraging additional metadata assigned to the functions which are being deployed as well as to the nodes in the cluster. This metadata includes special hardware capabilities beneficial for a workload's execution (for example, GPUs accelerating the training of an ML model), and the amount of data being transferred during the execution of a function.

Skippy Daemon The domain-specific priority functions of the Skippy scheduler depend on additional metadata of the functions and the nodes in the cluster. While a function's metadata is maintained by its developer, the node's metadata needs to be populated and

maintained too. The Skippy daemon is automatically deployed on every single node of the cluster. Once running on the node, it autonomously detects, populates and maintains the node's metadata.

Optimization The configuration of the Skippy scheduler – i.e. defining the individual weights of the priority functions – is an instance of the multi-objective optimization problem. We implemented an optimization utilizing an MOP solving algorithm based on our simulation environment which finds near-pareto-optimal solutions for the configuration based on several predefined objectives.

The evaluation of our contributions shows that the Skippy scheduler – even when not optimized – outperforms the recreated default scheduler on average in every objective. If the Skippy scheduler is optimized towards a specific objective, it clearly outperforms the recreated default scheduler. The bandwidth usage was reduced by up to 67.52%, the costs were reduced by up to 100% (since all workloads were executed on edge devices), the resource utilization of the edge devices was increased by up to 411.39%, and the TET was reduced by up to 71.18%.

The complete source-code of the Skippy scheduler, the simulation environment, as well as the optimization are open-source [Ras19c].

8.2 Future Work

In the course of this thesis we identified several areas which could be further improved. This section focuses on these challenges and proposes possible future research areas.

8.2.1 Further Development of the Skippy Scheduler

Additional Features

Even though the K8s default scheduler has limited capabilities in mixed cloud-edge infrastructures, it is a highly complex and mature component with a rich feature-set. During this thesis, only the essential parts of the default scheduler have been recreated. For the Skippy scheduler to be production-ready, several features of the default scheduler are currently missing. This includes the handling of pod preemption – i.e. evicting running pods from a node to make way for a higher prioritized pod – and several other scheduling features neglected in the initial prototype, like affinities and anti-affinities, or taints and tolerances.

Integration

By utilizing MCDM algorithms, the developed optimization approach can be automated and fully integrated into the system. Such an automated system determines the current state of the cluster and creates a simulation configuration based on the usually deployed functions, the nodes within the cluster, and their capabilities. Based on the simulation

configuration, the optimization is executed. The most suitable solution is selected based on a minimal initial configuration by the administrator and said MCDM algorithms. This solution is then used to configure the weights of the priority functions of the scheduler in the cluster. The resulting system therefore automatically adjust itself to changes in the function deployment and the nodes in the cluster.

Distributed Skippy

The scalability evaluation has shown that the Skippy scheduler's additional priority functions as well as the increased number of scored nodes have a negative influence on the performance of the scheduler. A possible solution could be to move away from the current queue-based monolithic architecture of the Skippy scheduler towards a hybrid architecture.

If the edge devices are provided by the customers of the platform itself, these devices should be solely used for workloads of this specific customer. We propose one instance of the Skippy scheduler per tenant. This instance of the scheduler exclusively handles the edge devices on-premises of the customer using its current monolithic architecture. The cloud resources, however, are shared among all tenants. These resources are handled by all schedulers in an optimistic lock-free fashion, resulting in a distributed shared-state scheduling architecture. This completely prevents collisions for edge devices, since they are all handled exclusively by the scheduler responsible for the tenant providing the devices. When competing about shared cloud resources, the optimistic lock-free approach was shown to be efficient [Sch+13].

8.2.2 Improving Support and Programmability of Edge Devices

Most of the current frameworks and platforms are solely focused on homogeneous cloud infrastructures. This results in a lack of support for CPU architectures other than AMD64. This can be seen in various areas. NVidia does not even provide a Docker runtime capable of handling its own GPUs on their own edge device – the NVidia Jetson TX2 – since it's based on an ARM64 processor. Creating a Docker image supporting multiple processor architectures is still an experimental feature and the cross-compilation is most cumbersome. In addition, it's currently not possible to create a Docker manifest-list which selects the correct image based on other aspects than the OS and the processor architecture, like specific hardware capabilities.

APPENDIX A

NVidia Jetson TX2 Setup

A.1 Kernel Flags

A.1.1 kube-proxy

Listing A.1 shows all kernel flags which need to be enabled for kube-proxy to be able to manage the K8s networking rules based on *iptables*.

```
CONFIG_NETFILTER_XT_SET=m
CONFIG_NETFILTER_XT_MATCH_MULTIPORT=m
CONFIG_NETFILTER_XT_MATCH_PHYSDEV=m
CONFIG_NETFILTER_XT_MATCH_RECENT=m
CONFIG_NETFILTER_XT_TARGET_REDIRECT=m
CONFIG_IP_SET=m
CONFIG_IP_SET_MAX=256
CONFIG_IP_SET_HASH_IP=m
CONFIG_IP_SET_HASH_NET=m
CONFIG_NF_NAT_REDIRECT=m
CONFIG_IP_NF_TARGET_REDIRECT=m
CONFIG_NET_SCH_NETEM=m
CONFIG_IFB=m
```

Listing A.1: NVidia Jetson TX2 kernel flags to enabled kube-proxy iptables management

A.1.2 traffic control

In order to enable a kernel-lavel traffic management used for the empirical measurements described in Section 4.4, the kernel flags shown in Listing A.2 need to be set.

```
CONFIG_NET_EGRESS=y
CONFIG_NET_SCH_CBQ=m
CONFIG_NET_SCH_HTB=m
CONFIG_NET_SCH_HFSC=m
CONFIG_NET_SCH_PRIO=m
CONFIG_NET_SCH_RED=m
```

```
CONFIG_NET_SCH_SFQ=m
CONFIG_NET_SCH_TEQL=m
CONFIG_NET_SCH_TBF=m
CONFIG_NET_SCH_GRED=m
CONFIG_NET_SCH_DSMARK=m
CONFIG_NET_SCH_NETEM=m
CONFIG_NET_SCH_DRR=m
CONFIG_NET_SCH_MQPRIO=m
CONFIG_NET_SCH_CHOKE=m
CONFIG_NET_SCH_QFQ=m
CONFIG_NET_SCH_CODEL=m
CONFIG_NET_SCH_FQ_CODEL=m
CONFIG_NET_SCH_FQ=m
CONFIG_NET_SCH_HHF=m
CONFIG_NET_SCH_PIE=m
CONFIG_NET_SCH_TEGRA=m
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_SCH_PLUG=m
CONFIG_NET_CLS_BASIC=m
CONFIG_NET_CLS_FW=m
CONFIG_CLS_U32_PERF=y
CONFIG_CLS_U32_MARK=y
CONFIG_NET_ACT_MIRRED=y
CONFIG_IFB=m
```

Listing A.2: NVidia Jetson TX2 kernel flags to enable traffic control

After configuring the kernel modules, the kernel can be rebuilt on the NVidia Jetson TX2 itself[1].

A.2 Container Runtime Patch

The runC container runtime[2] has been patched using the patch file shown in Listing A.3. Once the code is patched, the container runtime can be built and the resulting binary can be linked in the Docker daemon's configuration file.

```
diff --git a/spec.go b/spec.go
index 26e9754e..c371dc0a 100644
--- a/spec.go
+++ b/spec.go
@@ -128,9 +128,145 @@ func loadSpec(cPath string) (spec *specs.Spec, err error) {
  if err = json.NewDecoder(cf).Decode(&spec); err != nil {
    return nil, err
  }
+
+ // add necessary volume mounts
+ additionalMounts := []specs.Mount{
+  {
+   Destination: "/usr/lib/aarch64-linux-gnu",
+   Source: "/usr/lib/aarch64-linux-gnu",
+   Type: "bind",
+   Options: []string{"rbind", "rprivate"},
+  },
```

[1] https://github.com/alexrashed/buildJetsonTX2Kernel (visited on Nov. 26, 2019)
[2] https://github.com/opencontainers/runc (visited on Nov. 26, 2019)

```
+  {
+    Destination: "/usr/local/cuda/lib64",
+    Source: "/usr/local/cuda/lib64",
+    Type: "bind",
+    Options: []string{"rbind", "rprivate"},
+  },
+ }
+ spec.Mounts = append(spec.Mounts, additionalMounts...)
+
+ // add necessary devices
+ fileMode := os.FileMode(int(8624))
+ uid0 := uint32(0)
+ gid0 := uint32(0)
+ gid44 := uint32(44)
+ additionalDevices := []specs.LinuxDevice{
+  {
+    Path: "/dev/nvhost-ctrl",
+    Type: "c",
+    Major: 242,
+    Minor: 0,
+    FileMode: &fileMode,
+    UID: &uid0,
+    GID: &gid44,
+  },
+  {
+    Path: "/dev/nvhost-ctrl-gpu",
+    Type: "c",
+    Major: 506,
+    Minor: 2,
+    FileMode: &fileMode,
+    UID: &uid0,
+    GID: &gid44,
+  },
+  {
+    Path: "/dev/nvhost-prof-gpu",
+    Type: "c",
+    Major: 506,
+    Minor: 4,
+    FileMode: &fileMode,
+    UID: &uid0,
+    GID: &gid0,
+  },
+  {
+    Path: "/dev/nvhost-gpu",
+    Type: "c",
+    Major: 506,
+    Minor: 0,
+    FileMode: &fileMode,
+    UID: &uid0,
+    GID: &gid44,
+  },
+  {
+    Path: "/dev/nvhost-as-gpu",
+    Type: "c",
+    Major: 506,
+    Minor: 1,
+    FileMode: &fileMode,
+    UID: &uid0,
+    GID: &gid44,
+  },
+  {
+    Path: "/dev/nvmap",
```

```
80  +    Type: "c",
81  +    Major: 10,
82  +    Minor: 61,
83  +    FileMode: &fileMode,
84  +    UID: &uid0,
85  +    GID: &gid44,
86  +   },
87  + }
88  + spec.Linux.Devices = append(spec.Linux.Devices, additionalDevices...)
89  + additionalDeviceResources := []specs.LinuxDeviceCgroup{
90  +  {
91  +    Access: "rwm",
92  +    Allow: true,
93  +    Major: createPointer(242),
94  +    Minor: createPointer(0),
95  +    Type: "c",
96  +   },
97  +  {
98  +    Access: "rwm",
99  +    Allow: true,
100 +    Major: createPointer(506),
101 +    Minor: createPointer(2),
102 +    Type: "c",
103 +   },
104 +  {
105 +    Access: "rwm",
106 +    Allow: true,
107 +    Major: createPointer(506),
108 +    Minor: createPointer(4),
109 +    Type: "c",
110 +   },
111 +  {
112 +    Access: "rwm",
113 +    Allow: true,
114 +    Major: createPointer(506),
115 +    Minor: createPointer(0),
116 +    Type: "c",
117 +   },
118 +  {
119 +    Access: "rwm",
120 +    Allow: true,
121 +    Major: createPointer(506),
122 +    Minor: createPointer(1),
123 +    Type: "c",
124 +   },
125 +  {
126 +    Access: "rwm",
127 +    Allow: true,
128 +    Major: createPointer(10),
129 +    Minor: createPointer(61),
130 +    Type: "c",
131 +   },
132 + }
133 + spec.Linux.Resources.Devices = append(spec.Linux.Resources.Devices,
      additionalDeviceResources...)
134 +
135   return spec, validateProcessSpec(spec.Process)
136  }
137
138 +// Hack Helper Function
139 +func createPointer(x int64) *int64 {
140 + return &x
```

```
141 +}
142 +
143 +
144  func createLibContainerRlimit(rlimit specs.POSIXRlimit) (configs.Rlimit, error) {
145   rl, err := strToRlimit(rlimit.Type)
146   if err != nil {
```

Listing A.3: runC patch for NVidia Jetson TX2

APPENDIX B

OpenFaaS Modifications

Listing B.1 shows the changes which were necessary to *faas-netes*. It shows *a*) the modifications to the build scripts and Dockerfiles for the cross-compiled build and manifest list creation, and *b*) the additions to the deployment specification creation in order to set the Skippy scheduler for each deployment

```
From 1dbd14e2fcb3d33f3af34718a47534262d5b4e7d Mon Sep 17 00:00:00 2001
From: Alexander Rashed <alexander.rashed@gmail.com>
Date: Fri, 5 Jul 2019 12:15:05 +0200
Subject: [PATCH] Set skippy-scheduler as scheduler for new function pods,
 create multiarch images

---
 Dockerfile.arm64 | 5 +++--
 Dockerfile.armhf | 7 +++++---
 build.sh         | 11 ++++++++++-
 handlers/deploy.go | 1 +
 4 files changed, 18 insertions(+), 6 deletions(-)

diff --git a/Dockerfile.arm64 b/Dockerfile.arm64
index 6d0f8c61..1a473510 100644
--- a/Dockerfile.arm64
+++ b/Dockerfile.arm64
@@ -11,12 +11,13 @@ RUN gofmt -l -d $(find . -type f -name '*.go' -not -path
     "./vendor/*") \
     && go test ./test/ \
     && VERSION=$(git describe --all --exact-match `git rev-parse HEAD` | grep tags |
         sed 's/tags\///') \
     && GIT_COMMIT=$(git rev-list -1 HEAD) \
-    && CGO_ENABLED=0 GOOS=linux go build --ldflags "-s -w \
+    && CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build --ldflags "-s -w \
        -X github.com/openfaas/faas-netes/version.GitCommit=${GIT_COMMIT}\
        -X github.com/openfaas/faas-netes/version.Version=${VERSION}" \
        -a -installsuffix cgo -o faas-netes .

-FROM alpine:3.9
+FROM arm64v8/alpine:3.9
+COPY --from=multiarch/qemu-user-static:x86_64-aarch64 /usr/bin/qemu-* /usr/bin
 RUN apk --no-cache add ca-certificates
```

```
 WORKDIR /root/

diff --git a/Dockerfile.armhf b/Dockerfile.armhf
index 9cabd907..e38a9669 100644
--- a/Dockerfile.armhf
+++ b/Dockerfile.armhf
@@ -11,12 +11,13 @@ RUN gofmt -l -d $(find . -type f -name '*.go' -not -path
    "./vendor/*") \
    && go test ./test/ \
    && VERSION=$(git describe --all --exact-match `git rev-parse HEAD` | grep tags |
       sed 's/tags\///') \
    && GIT_COMMIT=$(git rev-list -1 HEAD) \
-   && CGO_ENABLED=0 GOOS=linux go build --ldflags "-s -w \
+   && CGO_ENABLED=0 GOOS=linux GOARCH=arm go build --ldflags "-s -w \
       -X github.com/openfaas/faas-netes/version.GitCommit=${GIT_COMMIT}\
       -X github.com/openfaas/faas-netes/version.Version=${VERSION}" \
       -a -installsuffix cgo -o faas-netes .

-FROM alpine:3.9 as ship
+FROM arm32v7/alpine:3.9 as ship
+COPY --from=multiarch/qemu-user-static:x86_64-arm /usr/bin/qemu-* /usr/bin

 RUN apk --no-cache add ca-certificates
 WORKDIR /root/
@@ -25,6 +26,6 @@ EXPOSE 8080
 ENV http_proxy ""
 ENV https_proxy ""

-COPY --from=build /go/src/github.com/openfaas/faas-netes/faas-netes .
+COPY --from=0 /go/src/github.com/openfaas/faas-netes/faas-netes .

 CMD ["./faas-netes"]
diff --git a/build.sh b/build.sh
index 5877f30c..466a13d8 100755
--- a/build.sh
+++ b/build.sh
@@ -1,6 +1,15 @@
 #!/bin/sh
-
 make build
+docker tag openfaas/faas-netes:latest alexrashed/faas-netes-skippy:0.1-amd64
+docker push alexrashed/faas-netes-skippy:0.1-amd64
+make build-arm64
+docker tag openfaas/faas-netes:latest-arm64 alexrashed/faas-netes-skippy:0.1-arm64
+docker push alexrashed/faas-netes-skippy:0.1-arm64
+make build-armhf
+docker tag openfaas/faas-netes:latest-armhf alexrashed/faas-netes-skippy:0.1-armhf
+docker push alexrashed/faas-netes-skippy:0.1-armhf
+docker manifest create --amend alexrashed/faas-netes-skippy:0.1
    alexrashed/faas-netes-skippy:0.1-amd64 alexrashed/faas-netes-skippy:0.1-arm64
    alexrashed/faas-netes-skippy:0.1-armhf
+docker manifest push alexrashed/faas-netes-skippy:0.1

 # os="$(uname -s)"

diff --git a/handlers/deploy.go b/handlers/deploy.go
index b11d5286..f1f93e1b 100644
--- a/handlers/deploy.go
+++ b/handlers/deploy.go
@@ -195,6 +195,7 @@ func makeDeploymentSpec(request requests.CreateFunctionRequest,
    existingSecrets
       },
```

```
89        Spec: apiv1.PodSpec{
90          NodeSelector: nodeSelector,
91  +       SchedulerName: "skippy-scheduler",
92          Containers: []apiv1.Container{
93            {
94              Name: request.Service,
```

Listing B.1: faas-netes patch for cross-compile builds and setting the Skippy scheduler for function deployments

APPENDIX C

Empirical Experiment Logs

C.1 Default Scheduler

Listing C.1 shows the (truncated) log output of the recreated default scheduler.

```
1  DEBUG:root:Loading in-cluster config...
2  DEBUG:root:Using default scheduler priority functions
3  DEBUG:root:Watching for new pod events across all namespaces...
4  DEBUG:root:Watching for new pods with defined scheduler-name 'skippy-scheduler'...
5  DEBUG:root:Starting liveness / readiness probe...
6  INFO:root:Everything is in place for new pods to be scheduled. Waiting for new
       events...
7  ...
8  DEBUG:root:There's a new pod to schedule: ml-wf-1-pre-5b64df4f45-l4qwv
9  DEBUG:root:Received a new pod to schedule: ml-wf-1-pre-5b64df4f45-l4qwv
10 ...
11 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clustercloud1: 4000 / 8361611264.Passed: True
12 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
13 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clusterpi1: 4000 / 1023012864.Passed: True
14 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clusterpi1 /
       PodFitsResourcesPred: Passed
15 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: True
16 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clusterpi2 /
       PodFitsResourcesPred: Passed
17 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: True
18 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clusterpi3 /
       PodFitsResourcesPred: Passed
19 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: True
20 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clusterpi4 /
       PodFitsResourcesPred: Passed
21 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv requests 100 / 104857600. Available on
       node ara-clustertegra1: 4000 / 8240386048.Passed: True
```

```
22 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / Node ara-clustertegra1 /
       PodFitsResourcesPred: Passed
23 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / BalancedResourcePriority: [9.0, 9.0,
       9.0, 9.0, 9.0, 9.0]
24 DEBUG:root:Pod ml-wf-1-pre-5b64df4f45-l4qwv / ImageLocalityPriority: [0.0, 0.0, 0.0,
       0.0, 0.0, 0.0]
25 DEBUG:root:Node scores: [(ara-clustercloud1, 9.0), (ara-clusterpi1, 9.0),
       (ara-clusterpi2, 9.0), (ara-clusterpi3, 9.0), (ara-clusterpi4, 9.0),
       (ara-clustertegra1, 9.0)]
26 INFO:root:Creating namespaced binding: Pod ml-wf-1-pre-5b64df4f45-l4qwv on Node
       ara-clustercloud1
27 DEBUG:kubernetes.client.rest:response body:
       {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
28 DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
       Capacity(CPU: 3900 Memory: 8256753664)
29 DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clustercloud1,
       feasible_nodes=6, needed_images=['alexrashed/ml-wf-1-pre:0.33'])
30 DEBUG:root:There's a new pod to schedule: ml-wf-2-train-5f57856bfc-2mxrb
31 DEBUG:root:Received a new pod to schedule: ml-wf-2-train-5f57856bfc-2mxrb
32 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clustercloud1: 3900 / 8256753664.Passed: True
33 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
34 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clusterpi1: 4000 / 1023012864.Passed: False
35 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clusterpi1 /
       PodFitsResourcesPred: Failed
36 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: False
37 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clusterpi2 /
       PodFitsResourcesPred: Failed
38 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: False
39 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clusterpi3 /
       PodFitsResourcesPred: Failed
40 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: False
41 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clusterpi4 /
       PodFitsResourcesPred: Failed
42 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb requests 100 / 1073741824. Available on
       node ara-clustertegra1: 4000 / 8240386048.Passed: True
43 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / Node ara-clustertegra1 /
       PodFitsResourcesPred: Passed
44 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / BalancedResourcePriority: [8.0, 8.0]
45 DEBUG:root:Pod ml-wf-2-train-5f57856bfc-2mxrb / ImageLocalityPriority: [0.0, 0.0]
46 DEBUG:root:Node scores: [(ara-clustercloud1, 8.0), (ara-clustertegra1, 8.0)]
47 INFO:root:Creating namespaced binding: Pod ml-wf-2-train-5f57856bfc-2mxrb on Node
       ara-clustercloud1
48 DEBUG:kubernetes.client.rest:response body:
       {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
49 DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
       Capacity(CPU: 3800 Memory: 7183011840)
50 DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clustercloud1,
       feasible_nodes=2, needed_images=['alexrashed/ml-wf-2-train:0.33'])
51 DEBUG:root:There's a new pod to schedule: ml-wf-3-serve-689b94855-vn5n5
52 DEBUG:root:Received a new pod to schedule: ml-wf-3-serve-689b94855-vn5n5
53 DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
       node ara-clustercloud1: 3800 / 7183011840.Passed: True
54 DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
55 DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
       node ara-clusterpi1: 4000 / 1023012864.Passed: True
```

```
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clusterpi1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
    node ara-clusterpi2: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clusterpi2 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
    node ara-clusterpi3: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clusterpi3 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
    node ara-clusterpi4: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clusterpi4 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 requests 100 / 209715200. Available on
    node ara-clustertegra1: 4000 / 8240386048.Passed: True
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / Node ara-clustertegra1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / BalancedResourcePriority: [9.0, 8.0,
    8.0, 8.0, 8.0, 9.0]
DEBUG:root:Pod ml-wf-3-serve-689b94855-vn5n5 / ImageLocalityPriority: [0.0, 0.0, 0.0,
    0.0, 0.0, 0.0]
DEBUG:root:Node scores: [(ara-clustercloud1, 9.0), (ara-clusterpi1, 8.0),
    (ara-clusterpi2, 8.0), (ara-clusterpi3, 8.0), (ara-clusterpi4, 8.0),
    (ara-clustertegra1, 9.0)]
INFO:root:Creating namespaced binding: Pod ml-wf-3-serve-689b94855-vn5n5 on Node
    ara-clustercloud1
DEBUG:kubernetes.client.rest:response body:
    {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
    Capacity(CPU: 3700 Memory: 6973296640)
DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clustercloud1,
    feasible_nodes=6, needed_images=['alexrashed/ml-wf-3-serve:0.33'])
```

Listing C.1: Testbed log output - Default scheduler

C.2 Non-Optimized Skippy Scheduler

Listing C.2 shows the (truncated) log output of the Skippy scheduler without any optimized priority weights.

```
DEBUG:root:Loading in-cluster config...
DEBUG:root:Watching for new pod events across all namespaces...
DEBUG:root:Watching for new pods with defined scheduler-name 'skippy-scheduler'...
DEBUG:root:Starting liveness / readiness probe...
INFO:root:Everything is in place for new pods to be scheduled. Waiting for new
    events...
...
DEBUG:root:There's a new pod to schedule: ml-wf-1-pre-569555ff57-55hzz
DEBUG:root:Received a new pod to schedule: ml-wf-1-pre-569555ff57-55hzz
...
DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
    node ara-clustercloud1: 4000 / 8361611264.Passed: True
DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clustercloud1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
    node ara-clusterpi1: 4000 / 1023012864.Passed: True
```

```
13 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clusterpi1 /
       PodFitsResourcesPred: Passed
14 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: True
15 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clusterpi2 /
       PodFitsResourcesPred: Passed
16 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: True
17 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clusterpi3 /
       PodFitsResourcesPred: Passed
18 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: True
19 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clusterpi4 /
       PodFitsResourcesPred: Passed
20 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz requests 100 / 104857600. Available on
       node ara-clustertegra1: 4000 / 8240386048.Passed: True
21 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / Node ara-clustertegra1 /
       PodFitsResourcesPred: Passed
22 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / BalancedResourcePriority: [9.0, 9.0,
       9.0, 9.0, 9.0, 9.0]
23 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / LatencyAwareImageLocalityPriority: [8.0,
       10.0, 10.0, 10.0, 10.0, 8.0]
24 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / LocalityTypePriority: [0.0, 10.0, 10.0,
       10.0, 10.0, 10.0]
25 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / DataLocalityPriority: [10.0, 0.0, 0.0,
       0.0, 0.0, 0.0]
26 DEBUG:root:Pod ml-wf-1-pre-569555ff57-55hzz / CapabilityPriority: [0.0, 0.0, 0.0, 0.0,
       0.0, 0.0]
27 DEBUG:root:Node scores: [(ara-clustercloud1, 27.0), (ara-clusterpi1, 29.0),
       (ara-clusterpi2, 29.0), (ara-clusterpi3, 29.0), (ara-clusterpi4, 29.0),
       (ara-clustertegra1, 27.0)]
28 INFO:root:Creating namespaced binding: Pod ml-wf-1-pre-569555ff57-55hzz on Node
       ara-clusterpi1
29 DEBUG:kubernetes.client.rest:response body:
       {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
30 DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
       Capacity(CPU: 3900 Memory: 918155264)
31 DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clusterpi1,
       feasible_nodes=6, needed_images=['alexrashed/ml-wf-1-pre:0.33'])
32 DEBUG:root:There's a new pod to schedule: ml-wf-2-train-66777dccf7-rfqg9
33 DEBUG:root:Received a new pod to schedule: ml-wf-2-train-66777dccf7-rfqg9
34 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
       node ara-clustercloud1: 4000 / 8361611264.Passed: True
35 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
36 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
       node ara-clusterpi1: 3900 / 918155264.Passed: False
37 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clusterpi1 /
       PodFitsResourcesPred: Failed
38 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: False
39 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clusterpi2 /
       PodFitsResourcesPred: Failed
40 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: False
41 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clusterpi3 /
       PodFitsResourcesPred: Failed
42 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: False
43 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clusterpi4 /
       PodFitsResourcesPred: Failed
44 DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 requests 100 / 1073741824. Available on
```

```
    node ara-clustertegra1: 4000 / 8240386048.Passed: True
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / Node ara-clustertegra1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / BalancedResourcePriority: [8.0, 8.0]
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / LatencyAwareImageLocalityPriority:
    [10.0, 9.0]
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / LocalityTypePriority: [0.0, 10.0]
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / DataLocalityPriority: [10.0, 0.0]
DEBUG:root:Pod ml-wf-2-train-66777dccf7-rfqg9 / CapabilityPriority: [0.0, 10.0]
DEBUG:root:Node scores: [(ara-clustercloud1, 28.0), (ara-clustertegra1, 37.0)]
INFO:root:Creating namespaced binding: Pod ml-wf-2-train-66777dccf7-rfqg9 on Node
    ara-clustertegra1
DEBUG:kubernetes.client.rest:response body:
    {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
    Capacity(CPU: 3900 Memory: 7166644224)
DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clustertegra1,
    feasible_nodes=2, needed_images=['alexrashed/ml-wf-2-train:0.33'])
DEBUG:root:There's a new pod to schedule: ml-wf-3-serve-56ddbcb9fc-kg6pq
DEBUG:root:Received a new pod to schedule: ml-wf-3-serve-56ddbcb9fc-kg6pq
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clustercloud1: 4000 / 8361611264.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clustercloud1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clusterpi1: 3900 / 918155264.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clusterpi1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clusterpi2: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clusterpi2 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clusterpi3: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clusterpi3 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clusterpi4: 4000 / 1023012864.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clusterpi4 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq requests 100 / 209715200. Available on
    node ara-clustertegra1: 3900 / 7166644224.Passed: True
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / Node ara-clustertegra1 /
    PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / BalancedResourcePriority: [9.0, 7.0,
    8.0, 8.0, 8.0, 9.0]
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / LatencyAwareImageLocalityPriority:
    [8.0, 10.0, 10.0, 10.0, 10.0, 8.0]
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / LocalityTypePriority: [0.0, 10.0,
    10.0, 10.0, 10.0, 10.0]
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / DataLocalityPriority: [10.0, 0.0, 0.0,
    0.0, 0.0, 0.0]
DEBUG:root:Pod ml-wf-3-serve-56ddbcb9fc-kg6pq / CapabilityPriority: [0.0, 0.0, 0.0,
    0.0, 0.0, 0.0]
DEBUG:root:Node scores: [(ara-clustercloud1, 27.0), (ara-clusterpi1, 27.0),
    (ara-clusterpi2, 28.0), (ara-clusterpi3, 28.0), (ara-clusterpi4, 28.0),
    (ara-clustertegra1, 27.0)]
INFO:root:Creating namespaced binding: Pod ml-wf-3-serve-56ddbcb9fc-kg6pq on Node
    ara-clusterpi2
DEBUG:kubernetes.client.rest:response body:
    {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
```

```
    Capacity(CPU: 3900 Memory: 813297664)
79  DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clusterpi2,
    feasible_nodes=6, needed_images=['alexrashed/ml-wf-3-serve:0.33'])
```

Listing C.2: Testbed log output - Non-Optimized Skippy Scheduler

C.3 Optimized Skippy Scheduler

Listing C.3 shows the (truncated) log output of the Skippy scheduler with testbed specific TET optimized priority weights.

```
1  DEBUG:root:Loading in-cluster config...
2  INFO:root:Using custom weights: [2.6923720489533647, 2.698796313141462,
       5.964581321182787, 4.720991059190411, 9.122271417461349]
3  DEBUG:root:Watching for new pod events across all namespaces...
4  DEBUG:root:Watching for new pods with defined scheduler-name 'skippy-scheduler'...
5  DEBUG:root:Starting liveness / readiness probe...
6  INFO:root:Everything is in place for new pods to be scheduled. Waiting for new
       events...
7  ...
8  DEBUG:root:There's a new pod to schedule: ml-wf-1-pre-644f8966dc-rx7nb
9  DEBUG:root:Received a new pod to schedule: ml-wf-1-pre-644f8966dc-rx7nb
10 ...
11 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clustercloud1: 4000 / 8361611264.Passed: True
12 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
13 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clusterpi1: 4000 / 1023012864.Passed: True
14 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clusterpi1 /
       PodFitsResourcesPred: Passed
15 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: True
16 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clusterpi2 /
       PodFitsResourcesPred: Passed
17 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: True
18 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clusterpi3 /
       PodFitsResourcesPred: Passed
19 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: True
20 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clusterpi4 /
       PodFitsResourcesPred: Passed
21 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb requests 100 / 104857600. Available on
       node ara-clustertegra1: 4000 / 8240386048.Passed: True
22 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / Node ara-clustertegra1 /
       PodFitsResourcesPred: Passed
23 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / BalancedResourcePriority:
       [24.231348440580284, 24.231348440580284, 24.231348440580284, 24.231348440580284,
       24.231348440580284, 24.231348440580284]
24 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / LatencyAwareImageLocalityPriority:
       [21.590370505131695, 26.987963131414617, 26.987963131414617, 26.987963131414617,
       26.987963131414617, 21.590370505131695]
25 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / LocalityTypePriority: [0.0,
       59.64581321182787, 59.64581321182787, 59.64581321182787, 59.64581321182787,
       59.64581321182787]
26 DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / DataLocalityPriority:
       [47.20991059190411, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
DEBUG:root:Pod ml-wf-1-pre-644f8966dc-rx7nb / CapabilityPriority: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
DEBUG:root:Node scores: [(ara-clustercloud1, 93.03162953761608), (ara-clusterpi1, 110.86512478382278), (ara-clusterpi2, 110.86512478382278), (ara-clusterpi3, 110.86512478382278), (ara-clusterpi4, 110.86512478382278), (ara-clustertegra1, 105.46753215753985)]
INFO:root:Creating namespaced binding: Pod ml-wf-1-pre-644f8966dc-rx7nb on Node ara-clusterpi1
DEBUG:kubernetes.client.rest:response body:
{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
DEBUG:root:Found best node. Remaining allocatable resources after scheduling: Capacity(CPU: 3900 Memory: 918155264)
DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clusterpi1, feasible_nodes=6, needed_images=['alexrashed/ml-wf-1-pre:0.33'])
DEBUG:root:There's a new pod to schedule: ml-wf-2-train-fd88fdd5c-98g7d
DEBUG:root:Received a new pod to schedule: ml-wf-2-train-fd88fdd5c-98g7d
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clustercloud1: 4000 / 8361611264.Passed: True
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clustercloud1 / PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clusterpi1: 3900 / 918155264.Passed: False
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clusterpi1 / PodFitsResourcesPred: Failed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clusterpi2: 4000 / 1023012864.Passed: False
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clusterpi2 / PodFitsResourcesPred: Failed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clusterpi3: 4000 / 1023012864.Passed: False
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clusterpi3 / PodFitsResourcesPred: Failed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clusterpi4: 4000 / 1023012864.Passed: False
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clusterpi4 / PodFitsResourcesPred: Failed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d requests 100 / 1073741824. Available on node ara-clustertegra1: 4000 / 8240386048.Passed: True
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / Node ara-clustertegra1 / PodFitsResourcesPred: Passed
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / BalancedResourcePriority: [21.538976391626917, 21.538976391626917]
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / LatencyAwareImageLocalityPriority: [26.987963131414617, 24.289166818273156]
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / LocalityTypePriority: [0.0, 59.64581321182787]
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / DataLocalityPriority: [47.20991059190411, 0.0]
DEBUG:root:Pod ml-wf-2-train-fd88fdd5c-98g7d / CapabilityPriority: [0.0, 91.22271417461349]
DEBUG:root:Node scores: [(ara-clustercloud1, 95.73685011494564), (ara-clustertegra1, 196.69667059634145)]
INFO:root:Creating namespaced binding: Pod ml-wf-2-train-fd88fdd5c-98g7d on Node ara-clustertegra1
DEBUG:kubernetes.client.rest:response body:
{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
DEBUG:root:Found best node. Remaining allocatable resources after scheduling: Capacity(CPU: 3900 Memory: 7166644224)
DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clustertegra1, feasible_nodes=2, needed_images=['alexrashed/ml-wf-2-train:0.33'])
DEBUG:root:There's a new pod to schedule: ml-wf-3-serve-566c4b9b97-s6bvm
DEBUG:root:Received a new pod to schedule: ml-wf-3-serve-566c4b9b97-s6bvm
```

```
59 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clustercloud1: 4000 / 8361611264.Passed: True
60 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clustercloud1 /
       PodFitsResourcesPred: Passed
61 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clusterpi1: 3900 / 918155264.Passed: True
62 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clusterpi1 /
       PodFitsResourcesPred: Passed
63 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clusterpi2: 4000 / 1023012864.Passed: True
64 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clusterpi2 /
       PodFitsResourcesPred: Passed
65 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clusterpi3: 4000 / 1023012864.Passed: True
66 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clusterpi3 /
       PodFitsResourcesPred: Passed
67 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clusterpi4: 4000 / 1023012864.Passed: True
68 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clusterpi4 /
       PodFitsResourcesPred: Passed
69 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm requests 100 / 209715200. Available on
       node ara-clustertegra1: 3900 / 7166644224.Passed: True
70 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / Node ara-clustertegra1 /
       PodFitsResourcesPred: Passed
71 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / BalancedResourcePriority:
       [24.231348440580284, 18.84660434267355, 21.538976391626917, 21.538976391626917,
       21.538976391626917, 24.231348440580284]
72 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / LatencyAwareImageLocalityPriority:
       [21.590370505131695, 26.987963131414617, 26.987963131414617, 26.987963131414617,
       26.987963131414617, 21.590370505131695]
73 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / LocalityTypePriority: [0.0,
       59.64581321182787, 59.64581321182787, 59.64581321182787, 59.64581321182787,
       59.64581321182787]
74 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / DataLocalityPriority:
       [47.20991059190411, 0.0, 0.0, 0.0, 0.0, 0.0]
75 DEBUG:root:Pod ml-wf-3-serve-566c4b9b97-s6bvm / CapabilityPriority: [0.0, 0.0, 0.0,
       0.0, 0.0, 0.0]
76 DEBUG:root:Node scores: [(ara-clustercloud1, 93.03162953761608), (ara-clusterpi1,
       105.48038068591603), (ara-clusterpi2, 108.1727527348694), (ara-clusterpi3,
       108.1727527348694), (ara-clusterpi4, 108.1727527348694), (ara-clustertegra1,
       105.46753215753985)]
77 INFO:root:Creating namespaced binding: Pod ml-wf-3-serve-566c4b9b97-s6bvm on Node
       ara-clusterpi2
78 DEBUG:kubernetes.client.rest:response body:
       {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
79 DEBUG:root:Found best node. Remaining allocatable resources after scheduling:
       Capacity(CPU: 3900 Memory: 813297664)
80 DEBUG:root:Pod yielded SchedulingResult(suggested_host=ara-clusterpi2,
       feasible_nodes=6, needed_images=['alexrashed/ml-wf-3-serve:0.33'])
```

Listing C.3: Testbed log output - TET Optimized Skippy Scheduler

APPENDIX D

Additional Evaluation Results

The following figures each show a matrix of line plots to visualize and compare the placement qualities of the different scheduler configurations – one for each of the cluster configurations which have not been covered in detail in Section 7.3. Each column represents one configuration of the optimized Skippy scheduler towards one objective. Each row focuses on the cumulative values for one objective. Therefore each column shows the performance of one configuration for each of the objectives while each row shows the performance of one objective across all different configurations. The plots on the diagonal of the matrix always show the performance of the Skippy scheduler when optimized towards the respective objective, revealing the highest potential of the Skippy scheduler.

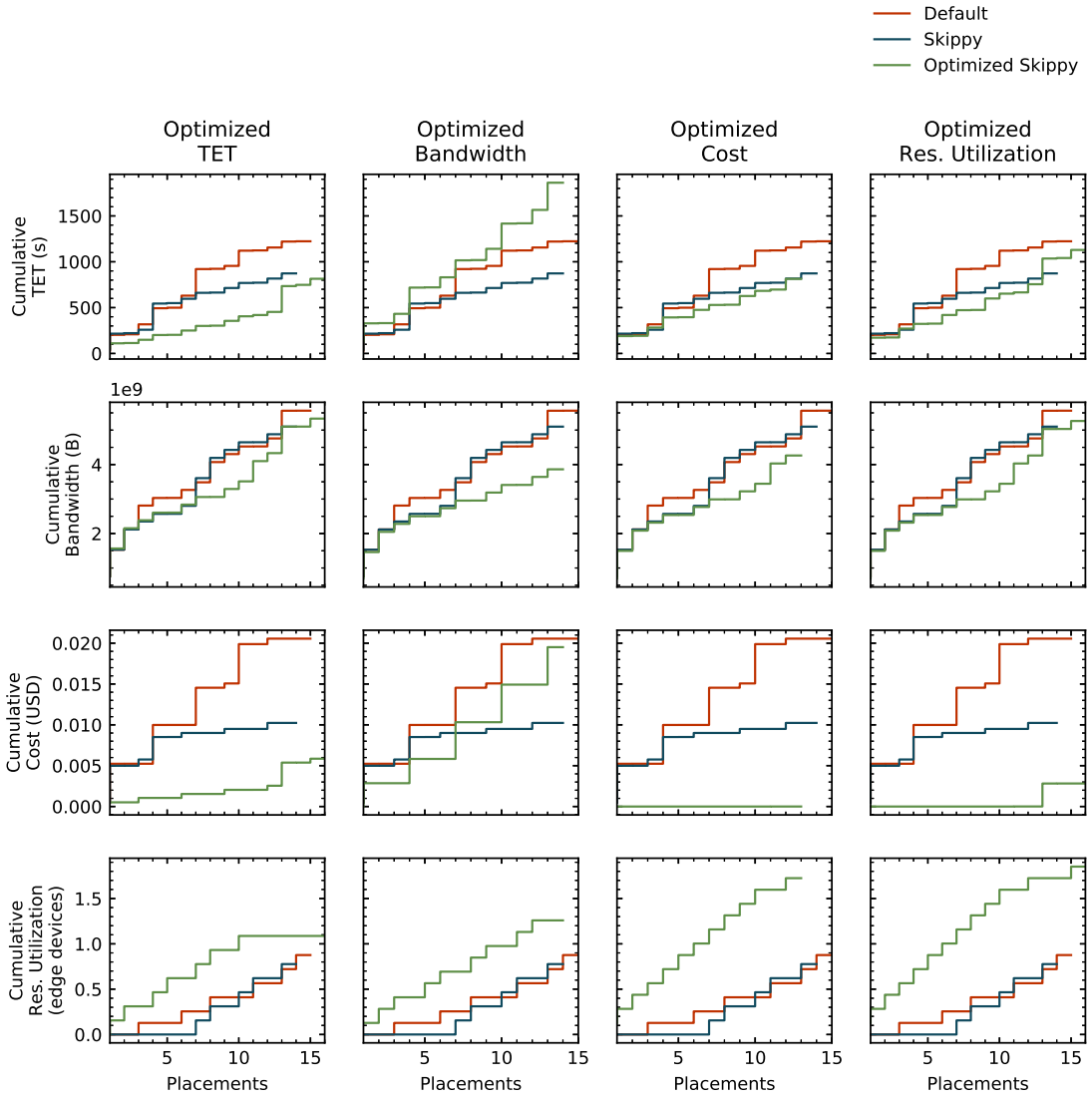


Figure D.1: Placement quality comparison matrix plot for the *testbed* cluster configuration

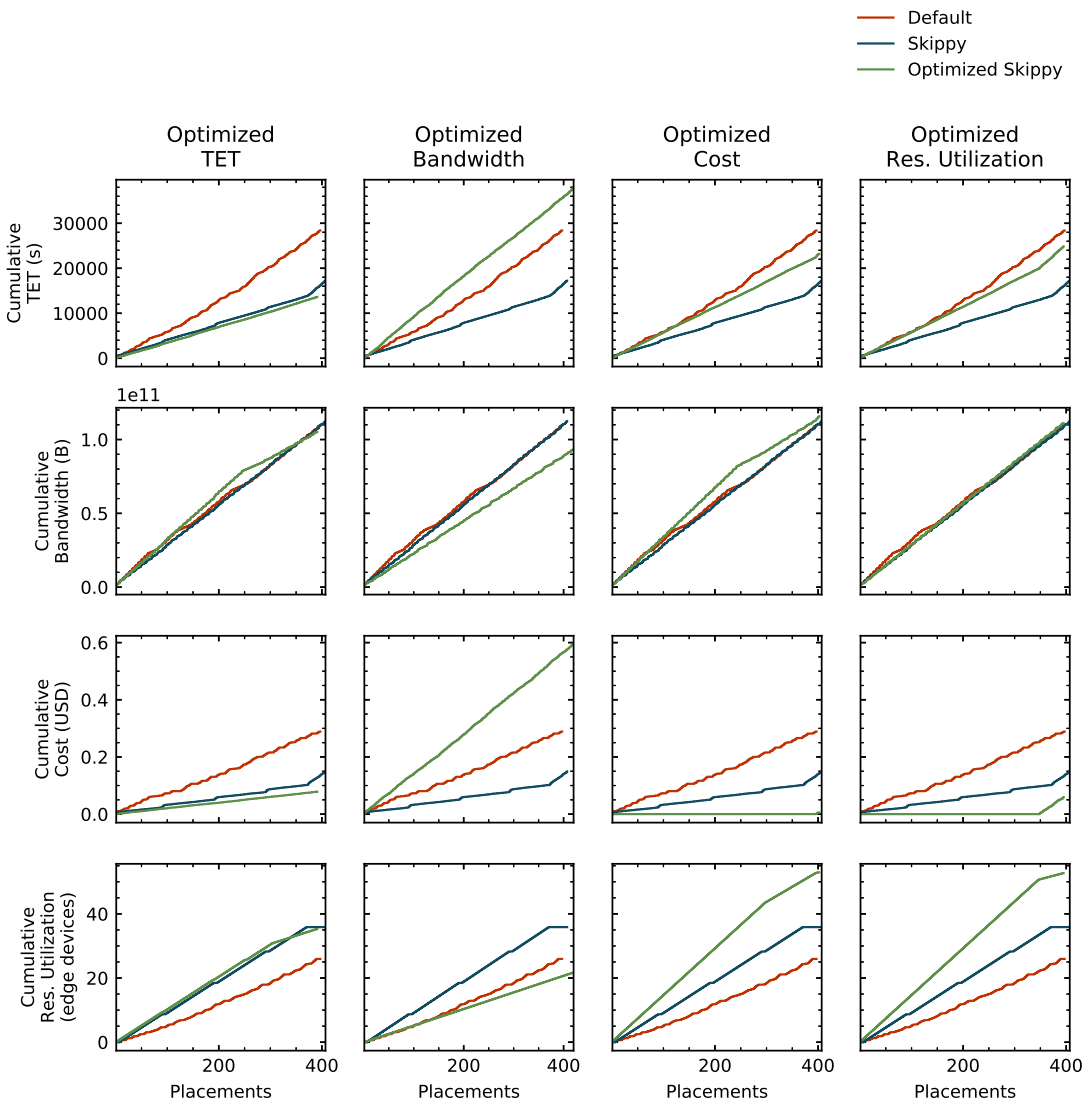


Figure D.2: Placement quality comparison matrix plot for the *equal_100* cluster configuration

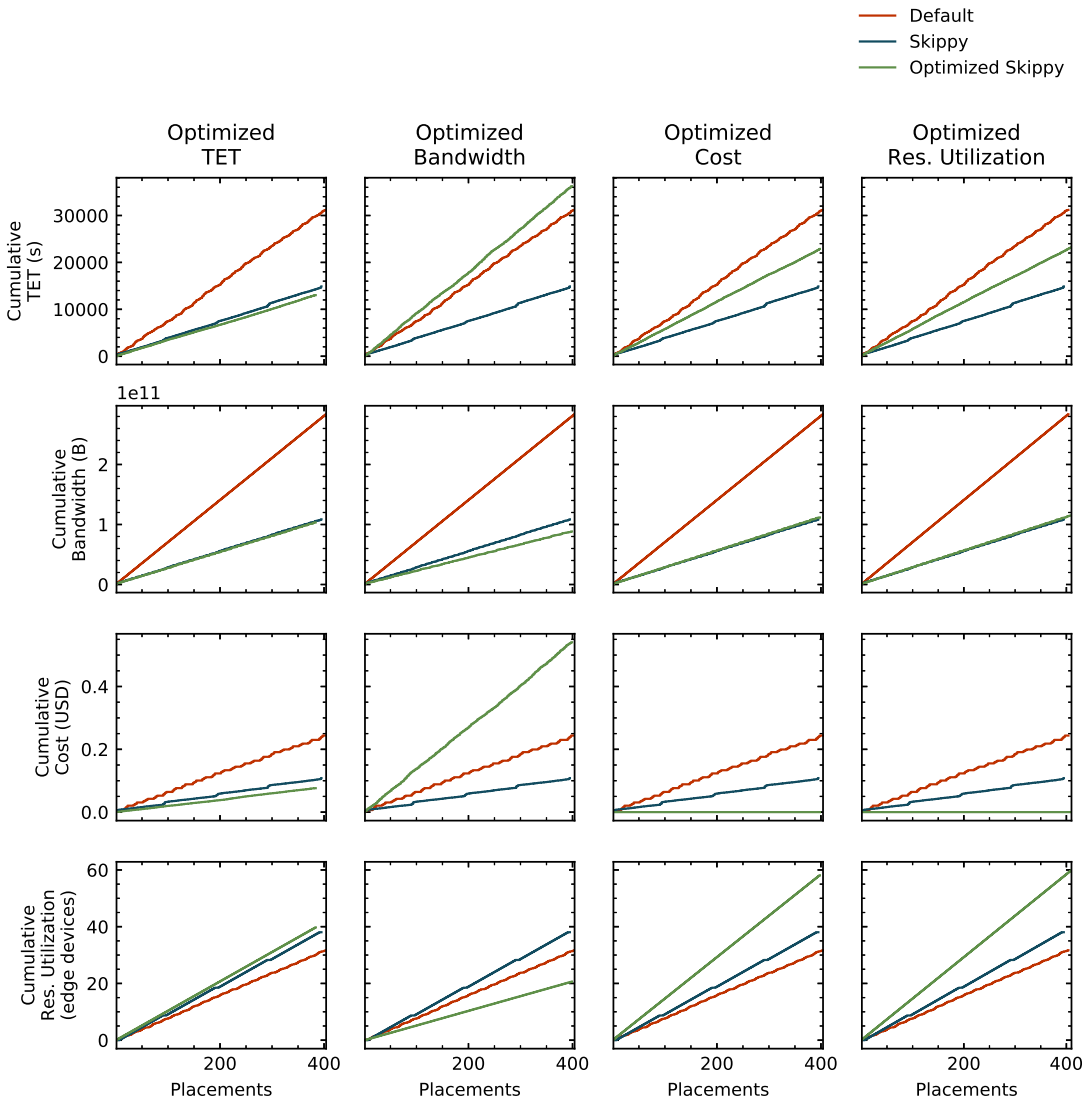


Figure D.3: Placement quality comparison matrix plot for the *equal_1000* cluster configuration

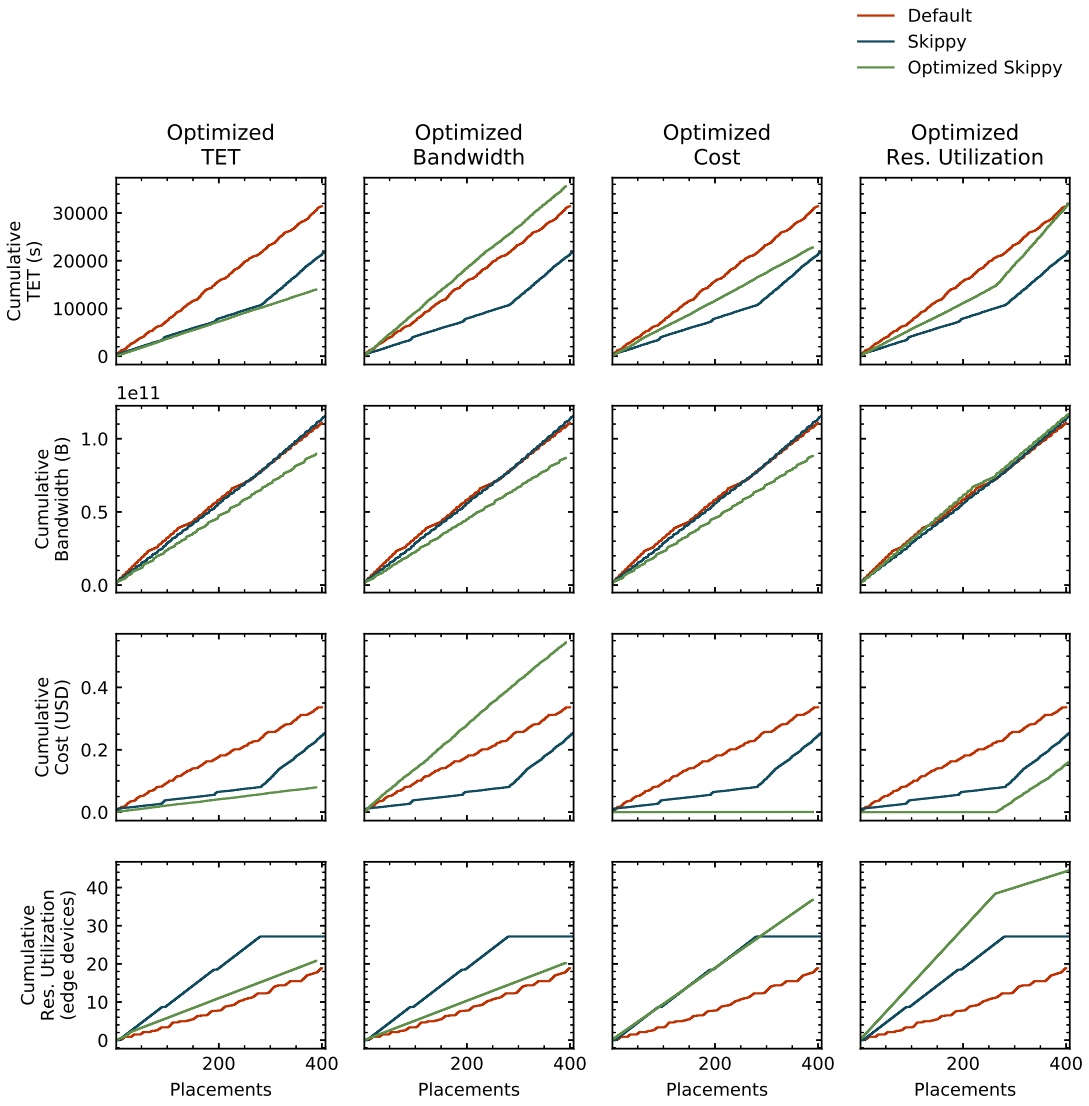


Figure D.4: Placement quality comparison matrix plot for the *50p_cloud_100* cluster configuration

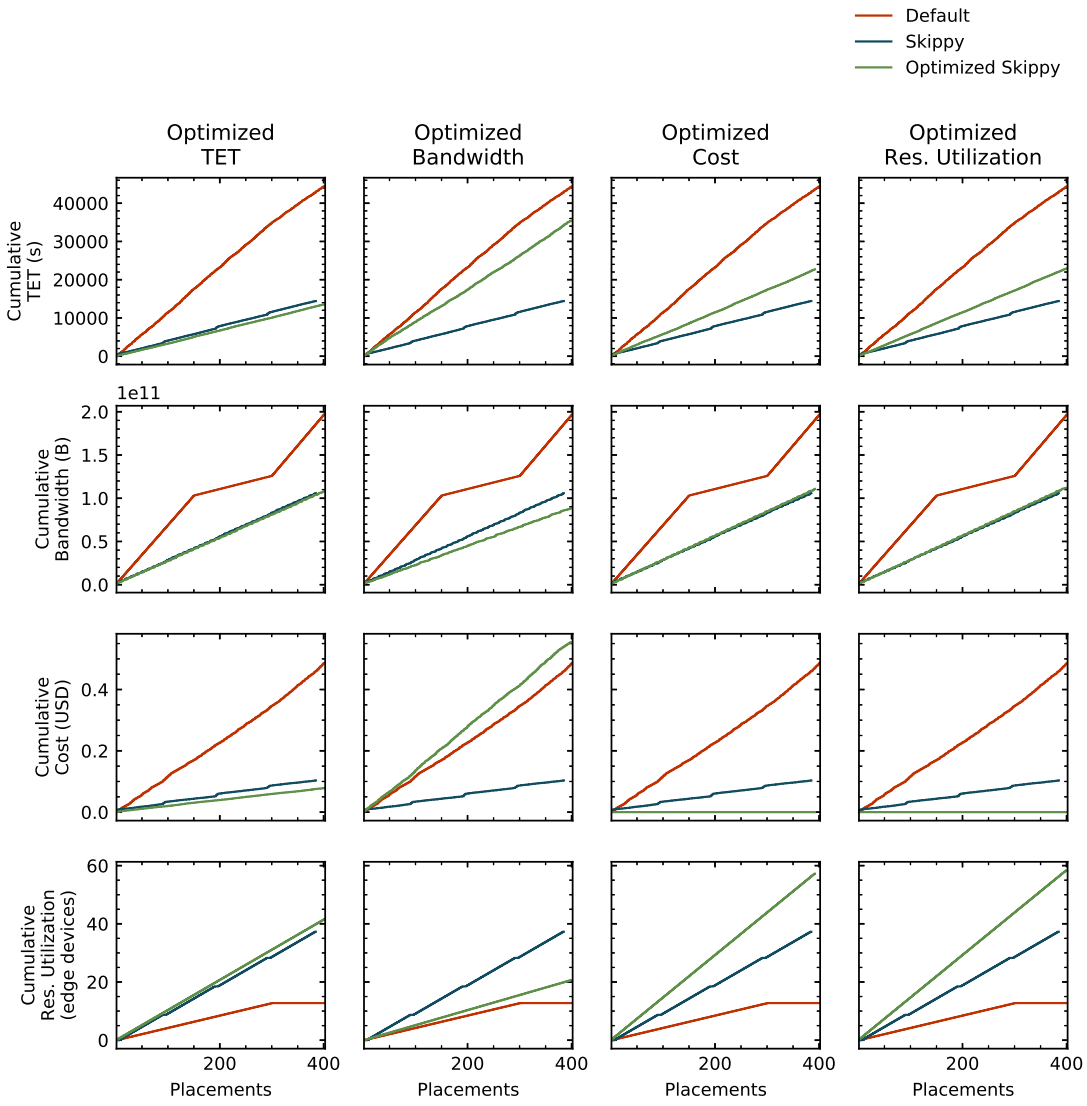


Figure D.5: Placement quality comparison matrix plot for the *50p_cloud_1000* cluster configuration

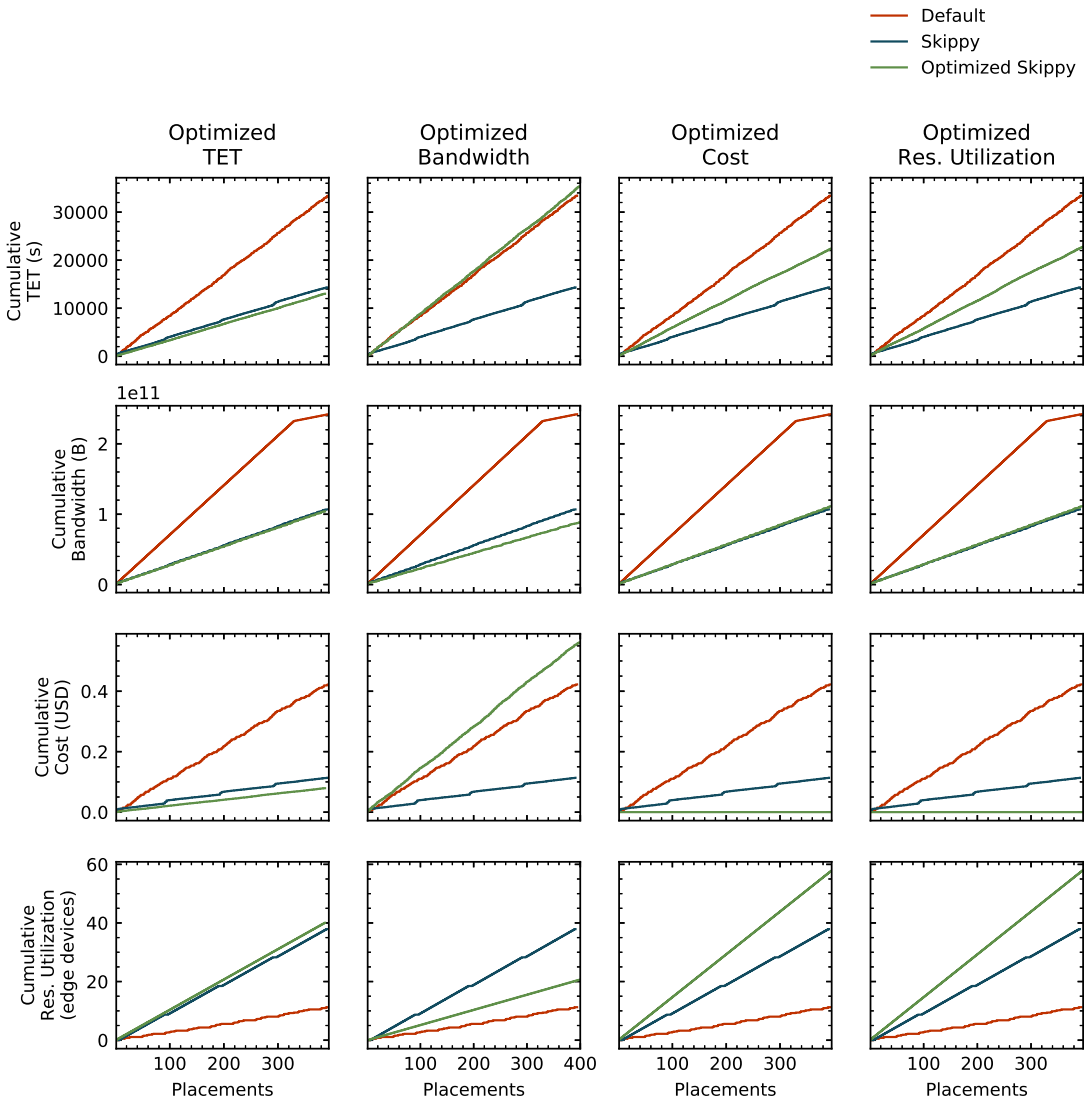


Figure D.6: Placement quality comparison matrix plot for the *cloud_1000* cluster configuration

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

API Application Programming Interface. 10, 16–19, 36, 37, 53, 121

CDF Cumulative Distribution Function. 88

CDN Content Delivery Network. 11

cgroups control groups. 14

CLI Command Line Interface. 15, 16, 34, 36, 56, 61

CO Container Orchestration. xi, 3, 5, 17, 26, 29–32

CUDA Compute Unified Device Architecture. 53

FaaS Function-as-a-Service. xi, 1–3, 5, 8–11, 25, 26, 29–32, 36, 38, 41, 43, 54, 56, 66, 71, 91, 123

GPU Graphics Processing Unit. 34–36, 38, 52, 53, 72, 81, 83, 84, 91, 93

HTTP Hypertext Transfer Protocol. 16, 18, 38

IaaS Infrastructure-as-a-Service. 1, 9

IoT Internet of Things. xi, 1, 2, 6, 11, 26, 31

K8s Kubernetes. xi, 13, 14, 17–19, 26–32, 34, 36, 41, 43, 44, 46, 47, 53–55, 58, 61, 87, 88, 91, 92, 95, 121

LXC Linux Containers. 14, 15

MCDM Multi-Criteria Decision-Making. 68, 92, 93

ML Machine Learning. 2, 3, 5, 21–23, 31, 34, 37, 52, 91, 121

MNIST Modified National Institute of Standards and Technology. 37, 38, 121

MOP Multi-objective Optimization Problem. 3, 5, 20, 21, 42, 65–67, 92

NSGA-II Non-dominated Sorting Genetic Algorithm II. 67, 68

OCI Open Container Initiative. 15, 16

OS Operating System. 14, 19, 93

PaaS Platform-as-a-Service. 1, 9

SIG Special Interest Group. 88

tc traffic control. 40

TET Task Execution Time. xi, 2, 12, 66, 68, 69, 71, 72, 75, 77–79, 81–85, 87, 92, 110, 121–123

TLS Transport Layer Security. 36

TPL Task Placement Latency. 88, 122

UI User Interface. 36

VM Virtual Machine. 10, 14, 18, 34, 36, 61, 69, 70, 72

Bibliography

[Arm+10] Michael Armbrust et al. „A View of Cloud Computing“. In: *Communications of the ACM* 53.4 (2010). DOI: 10.1145/1721654.1721672.

[Bal+17a] Ioana Baldini et al. „Serverless computing: Current trends and open problems“. In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1.

[Bal+17b] Ioana Baldini et al. „The serverless trilemma: function composition for serverless computing“. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM. 2017, pp. 89–103. DOI: 10.1145/3133850.3133855.

[Bay+17] Denis Baylor et al. „Tfx: A tensorflow-based production-scale machine learning platform“. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2017, pp. 1387–1395. DOI: 10.1145/3097983.3098021.

[Ber14] David Bernstein. „Containers and Cloud: From LXC to Docker to Kubernetes“. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[BM19] Luciano Baresi and Danilo Mendonça. „Towards a Serverless Platform for Edge Computing“. In: Mar. 2019. DOI: 10.1109/ICFC.2019.00008.

[Boa+17] Scott Boag et al. „Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs“. In: *Workshop on ML Systems, NIPS*. 2017.

[Bon+12] Flavio Bonomi et al. „Fog computing and its role in the internet of things“. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16. DOI: 10.1145/2342509.2342513.

[Bon+14] Flavio Bonomi et al. „Fog computing: A platform for internet of things and analytics“. In: *Big data and internet of things: A roadmap for smart environments*. Springer, 2014, pp. 169–186. DOI: 10.1007/978-3-319-05029-4_7.

[Bou+14] Eric Boutin et al. „Apollo: scalable and coordinated scheduling for cloud-scale computing“. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 285–300.

[Bre18] David Breitgand. *Lean OpenWhisk: Open Source FaaS for Edge Computing*. July 2018. URL: https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b (visited on Nov. 26, 2019).

[Bur+16] Brendan Burns et al. „Borg, Omega, and Kubernetes“. In: *ACM Queue* 14 (2016), pp. 70–93.

[Buz+18] Alina Buzachis et al. „Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments“. In: *Fog and Edge Computing (ICFEC), 2018 IEEE 2nd International Conference on*. IEEE. 2018, pp. 1–10. DOI: 10.1109/CFEC.2018.8358729.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. DOI: 10.1017/CBO9780511804441.

[Car+18] Joao Carreira et al. „A Case for Serverless Machine Learning“. In: *Workshop on Systems for ML and Open Source Software at NeurIPS*. Vol. 2018. 2018.

[Cas+17] Paul Castro et al. „Serverless Programming (Function as a Service)“. In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 2658–2659. DOI: 10.1109/ICDCS.2017.305.

[Che+19] Bin Cheng et al. „Fog Function: Serverless Fog Computing for Data Intensive IoT Services“. In: *2019 IEEE International Conference on Services Computing (SCC)*. IEEE. 2019, pp. 28–35. DOI: 10.1109/SCC.2019.00018.

[Cis18] Cisco Systems, Inc. *Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper*. Nov. 2018. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html (visited on Nov. 26, 2019).

[Deb+02] Kalyanmoy Deb et al. „A fast and elitist multiobjective genetic algorithm: NSGA-II“. In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[Deb01] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons, 2001. ISBN: 047187339X.

[Deb14] Kalyanmoy Deb. „Multi-objective optimization“. In: *Search methodologies*. Springer, 2014, pp. 403–449. DOI: 10.1007/0-387-28356-0_10.

[Den16] Hongchao Deng. *Improving Kubernetes Scheduler Performance*. Feb. 2016. URL: https://coreos.com/blog/improving-kubernetes-scheduler-performance.html (visited on Nov. 26, 2019).

[ED18] Michael TM Emmerich and André H Deutz. „A tutorial on multiobjective optimization: fundamentals and evolutionary methods“. In: *Natural computing* 17.3 (2018), pp. 585–609. DOI: 10.1007/s11047-018-9685-y.

[Fox+17] Geoffrey C Fox et al. „Status of serverless computing and function-as-a-service (faas) in industry and research“. In: *arXiv preprint arXiv:1708.08028* (2017). DOI: 10.13140/RG.2.2.15007.87206.

[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. ACM, 2017, 28:1–28:1. DOI: 10.1145/3078468.3078497.

[Gog+16] Ionel Gog et al. „Firmament: Fast, Centralized Cluster Scheduling at Scale“. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Nov. 2016, pp. 99–115.

[HBB17] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. "O'Reilly Media, Inc.", 2017. ISBN: 978-1-491-93567-5.

[Hel+18] Joseph M Hellerstein et al. „Serverless Computing: One Step Forward, Two Steps Back“. In: *arXiv preprint arXiv:1812.03651* (2018).

[Hin+11] Benjamin Hindman et al. „Mesos: A platform for fine-grained resource sharing in the data center.“ In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[HP18] John L Hennessy and David A Patterson. „A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development“. In: *Turing Lecture* (2018).

[Hum+19] Waldemar Hummer et al. „Modelops: Cloud-based lifecycle management for reliable and trusted AI“. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2019, pp. 113–120. DOI: 10.1109/IC2E.2019.00025.

[Ism+15] Bukhary Ikhwan Ismail et al. „Evaluation of docker as edge computing platform“. In: *2015 IEEE Conference on Open Systems (ICOS)*. IEEE. 2015, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

[Jon+19] Eric Jonas et al. „Cloud programming simplified: a berkeley view on serverless computing“. In: *arXiv preprint arXiv:1902.03383* (2019).

[Joy15] Ann Mary Joy. „Performance comparison between linux containers and virtual machines“. In: *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[Kar+15] Konstantinos Karanasos et al. „Mercury: Hybrid centralized and distributed scheduling in large shared clusters“. In: *2015 USENIX Annual Technical Conference (USENIXATC 15)*. 2015, pp. 485–497.

[KCS06] Abdullah Konak, David W Coit, and Alice E Smith. „Multi-objective optimization using genetic algorithms: A tutorial“. In: *Reliability Engineering & System Safety* 91.9 (2006), pp. 992–1007. DOI: 10.1016/j.ress.2005.11.018.

[Kha17] Asif Khan. „Key characteristics of a container orchestration platform to enable a modern application“. In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933.

[Kuba] Kubernetes Community. *Building large clusters*. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visited on Nov. 26, 2019).

[Kubb] Kubernetes Community. *Kubernetes IoT Edge Working Group*. URL: https://github.com/kubernetes/community/tree/master/wg-iot-edge (visited on Nov. 26, 2019).

[Kubc] Kubernetes Community. *Scalability Special Interest Group*. URL: https://github.com/kubernetes/community/tree/master/sig-scalability (visited on Nov. 26, 2019).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. „Deep learning“. In: *nature* 521.7553 (2015), p. 436. DOI: 10.1038/nature14539.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. „MNIST handwritten digit database“. In: (1998). URL: http://yann.lecun.com/exdb/mnist/ (visited on Nov. 26, 2019).

[Lei+19] Philipp Leitner et al. „A mixed-method empirical study of Function-as-a-Service software development in industrial practice“. In: *Journal of Systems and Software* 149 (2019), pp. 340–359. DOI: 10.1016/j.jss.2018.12.013.

[Li+17] Li Erran Li et al. „Scaling machine learning as a service“. In: *International Conference on Predictive Applications and APIs*. 2017, pp. 14–29.

[MB17] Garrett McGrath and Paul R Brenner. „Serverless computing: Design, implementation, and performance“. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[MKK15] Roberto Morabito, Jimmy Kjällman, and Miika Komu. „Hypervisors vs. lightweight virtualization: a performance comparison“. In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[MPD+18] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. „An Evaluation of Open Source Serverless Computing Frameworks“. In: *CloudCom*. 2018, pp. 115–120. DOI: 10.1109/CloudCom2018.2018.00033.

[Nas+17] Stefan Nastic et al. „A serverless real-time data analytics platform for edge computing“. In: *IEEE Internet Computing* 21.4 (2017), pp. 64–71. DOI: 10.1109/MIC.2017.2911430.

[ND18] Stefan Nastic and Schahram Dustdar. „Towards Deviceless Edge Computing: Challenges, Design Aspects, and Models for Serverless Paradigm at the Edge“. In: *The Essence of Software Engineering*. Springer, 2018, pp. 121–136. DOI: 10.1007/978-3-319-73897-0_8.

[Ols19] Parmy Olson. *How Sony Sped Up A Factory With These Tiny, $35 Computers*. Mar. 2019. URL: https://www.forbes.com/sites/parmyolson/2019/03/10/how-sony-sped-up-a-factory-with-these-tiny-35-computers/ (visited on Nov. 26, 2019).

[Pin12] Michael Pinedo. *Scheduling*. Vol. 29. Springer, 2012. DOI: 10.1007/978-3-319-26580-3.

[PKC19] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. „An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge“. In: *2019 IEEE World Congress on Services (SERVICES)*. Vol. 2642. IEEE. 2019, pp. 206–211. DOI: 10.1109/SERVICES.2019.00057.

[Ras19a] Alexander Rashed. *Empirical measurements of function placements and executions in a mixed cloud-edge cluster*. Version 1.0. Nov. 2019. DOI: 10.5281/zenodo.3553868. URL: http://doi.org/10.5281/zenodo.3553868.

[Ras19b] Alexander Rashed. *Simulated function placements in predefined mixed cloud-edge clusters*. Version 1.1. Nov. 2019. DOI: 10.5281/zenodo.3538959. URL: https://doi.org/10.5281/zenodo.3538959.

[Ras19c] Alexander Rashed. *Skippy Scheduler - An Optimized Container Scheduler for Serverless Edge Computing*. Version 1.0. Nov. 2019. DOI: 10.5281/zenodo.3553916. URL: https://doi.org/10.5281/zenodo.3553916.

[Rau+19] Thomas Rausch et al. „Towards a Serverless Platform for Edge AI“. In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[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)*. 2019. DOI: 10.1109/IC2E.2019.00022.

[Rez15] Jafar Rezaei. „Best-worst multi-criteria decision-making method“. In: *Omega* 53 (2015), pp. 49–57. DOI: 10.1016/j.omega.2014.11.009.

[Rez16] Jafar Rezaei. „Best-worst multi-criteria decision-making method: Some properties and a linear model“. In: *Omega* 64 (2016), pp. 126–130. DOI: 10.1016/j.omega.2015.12.001.

[Sat+09] Mahadev Satyanarayanan et al. „The case for vm-based cloudlets in mobile computing“. In: *IEEE pervasive Computing* (2009). DOI: 10.1109/MPRV.2009.82.

[Sat17] Mahadev Satyanarayanan. „The emergence of edge computing“. In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[SB14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781107298019.

[Sch+13] Malte Schwarzkopf et al. „Omega: flexible, scalable schedulers for large compute clusters“. In: *SIGOPS European Conference on Computer Systems (EuroSys)*. 2013, pp. 351–364.

[Sco+18] Vincenzo Scoca et al. „Scheduling Latency-Sensitive Applications in Edge Computing“. In: *Closer*. 2018, pp. 158–168. DOI: 10.5220/0006706201580168.

[SD16] Weisong Shi and Schahram Dustdar. „The promise of edge computing“. In: *Computer* 49.5 (2016), pp. 78–81. DOI: 10.1109/MC.2016.145.

[SD19] Jay Shah and Dushyant Dubaria. „Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform“. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2019, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479.

[Shi+16] Weisong Shi et al. „Edge computing: Vision and challenges“. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[Ska+16] Olena Skarlat et al. „Resource provisioning for IoT services in the fog“. In: *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*. IEEE. 2016, pp. 32–39.

[Ska+17a] Olena Skarlat et al. „Optimized IoT service placement in the fog“. In: *Service Oriented Computing and Applications* 11.4 (2017), pp. 427–443. DOI: 10.1007/s11761-017-0219-8.

[Ska+17b] Olena Skarlat et al. „Towards QoS-aware fog service placement“. In: *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*. IEEE. 2017, pp. 89–96. DOI: 10.1109/JIOT.2017.2701408.

[TRA15] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. „Container-based orchestration in cloud: state of the art and challenges“. In: *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. IEEE. 2015, pp. 70–75. DOI: 10.1109/CISIS.2015.35.

[Tru+19] Eddy Truyen et al. „A comprehensive feature comparison study of open-source container orchestration frameworks“. In: *Applied Sciences* 9.5 (2019), p. 931. DOI: 10.3390/app9050931.

[UVK19] Oana-Mihaela Ungureanu, Călin Vlădeanu, and Robert Kooij. „Kubernetes cluster optimization using hybrid shared-state scheduling framework“. In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. ACM. 2019, p. 2. DOI: 10.1145/3341325.3341992.

[VR14] Luis M Vaquero and Luis Rodero-Merino. „Finding your way in the fog: Towards a comprehensive definition of fog computing“. In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014), pp. 27–32. DOI: 10.1145/2677046.2677052.

[Wöb+18] Cecil Wöbker et al. „Fogernetes: Deployment and management of fog computing applications“. In: *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2018, pp. 1–7. DOI: 10.1109/NOMS.2018.8406321.

[Xio+18] Ying Xiong et al. „Extend Cloud to Edge with KubeEdge“. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 373–377. DOI: 10.1109/SEC.2018.00048.