Informatics

Container Scheduling on Heterogeneous Clusters using Machine Learning-based Workload Characterization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Alexander Raith, BSc
Matrikelnummer 01425076

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 15. Februar 2021

Philipp Alexander Raith

Schahram Dustdar

Informatics

Container Scheduling on Heterogeneous Clusters using Machine Learning-based Workload Characterization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Alexander Raith, BSc
Registration Number 01425076

to the Faculty of Informatics

at the TU Wien

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

Vienna, 15th February, 2021

Philipp Alexander Raith

Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Philipp Alexander Raith, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Februar 2021

Philipp Alexander Raith

Danksagung

An dieser Stelle möchte ich mich herzlich bei meinen Betreuern Schahram Dustdar und Mitbetreuer Thomas Rausch bedanken. Vor allem Thomas Rausch möchte ich für die wichtigen Diskussionen und Anregungen danken. Aufgrund ihrer tatkräftigen Unterstützung war ich imstande diese Arbeit zu verfassen.

Dank gilt auch meiner Familie und Freunden, die in schwierigen Zeiten immer zur Stelle waren und aufgrund meiner ständigen Erzählungen und ihr aufmerksames Zuhören sicherlich genauso viel gelernt haben wie ich.

Zu guter Letzt möchte ich bei netidee bedanken, die diese Arbeit mit einem Stipendium gefördert haben.

Kurzfassung

Edge Intelligence ist ein neues Paradigma, welches Nutzern ermöglicht KI-Applikationen zu verwenden, die eine sehr geringe Latenz benötigen. Edge Computing ermöglicht solche Szenarien. Rechenressourcen werden in der Nähe von Nutzern positioniert, um die nötige Latenz zu garantieren. Diese heterogenen Cluster bestehen aus verschiedenen Geräten, die unterschiedliche Fähigkeiten haben. Zu diesen Geräten gehören: CPU/GPU-basierte Geräte bis hin zu anwendungsspezifischen Beschleunigern. Cloud Rechenzentren sind im Vergleich dazu relativ homogen. Diese Heterogenität stellt ein Problem für Anwendungsentwickler da: sie müssen sich für Geräte entscheiden. Serverless Computing kann eine Lösung für dieses Problem darstellen. Dieses Paradigma versteckt die Infrastruktur und Anwendungsentwickler können Funktionen hochladen. Diese werden anschließend vom System automatisch auf Geräten gestartet. Diese beiden Paradgima werden zu Serverless Edge Computing verknüpft. Das Problem ist: Serverless Plattformen verwenden Scheduler, die ursprünglich für homogene Cluster entwickelt wurden. Daher stellen heterogene Cluster eine Herausforderung für diese Scheduler dar.

In dieser Arbeit, präsentieren wir eine Lösung, die mit Hilfe von Anforderungen, von Applikationen, angemessene Geräte findet. Dieser Lösungsversuch wird mit Hilfe von KI-basierter Applikationen-Charakterisierung handhabbar gemacht. Wir verwenden existierende Serverless Plattformen: Kubernetes und OpenFaaS. OpenFaaS ist eine Function-as-a-Service Plattform und verwendet intern Kubernetes, um Anwendung auf Geräten zu platzieren. Der Scheduler von Kubernetes verwendet simple Heuristiken, um Geräte für Anwendungen zu finden. Wir machen den Scheduler aufmerksam auf Applikationen und und schafen das durch drei Erweiterungen. Diese Erweiterungen fokussieren sich auf folgende Punkte: (1) Performance, (2) Ressourcen Staus und (3) das Finden angemessener Geräte für Applikationen. Dies ermöglichen wir durch: (1) umfangreiche Applikationstests auf Geräten bei denen wir Performance und Ressourcenverbrauch messen, (2) eine anschließende Applikations-Charakterisierung und (3) durch Lösen des Problems, Anforderungen für Applikationen zu ermitteln.

Wir evaluieren unsere Arbeit mit Hilfe von Simulationen und drei unterschiedlichen Szenarien. Unsere Anwendungen fokussieren sich auf KI-Anwendungen. Unsere Resultate zeigen, dass wir die Ausführungszeit in Edge Computing Szenarien im Durchschnitt um 33% bis 68% senken. Weiters, wir verringern die Verschlechterung von Ausführungszeit, aufgrund von Ressourcen Staus, um 25% bis 57%.

Abstract

Edge Intelligence is a paradigm that promises to bring highly-responsive AI applications to the end users. Edge computing is the main enabler paradigm for this scenario, where computational resources are pushed from the cloud to the edge of the network. These heterogeneous clusters consist of devices that offer different capabilities, which range from general purpose CPUs, to GPUs, to application specific hardware accelerators. Cloud systems are comparatively homogeneous in terms of computing infrastructure. This heterogeneity poses a problem for users, which have to decide the hardware for their applications. To mitigate this issue, serverless computing may be a solution to this problem. Serverless computing helps abstract the underlying infrastructure from users away—allowing users to simply upload functions and offers the convenience of automatic scaling and pay-per-use cost model. Research proposes the idea of merging both paradigms, resulting in serverless edge computing. The problem is, that current serverless computing platforms use schedulers that were developed for homogeneous clusters. Therefore, heterogeneous clusters pose a challenge for serverless container schedulers to find optimal placements for containers.

In this thesis, we propose a solution that matches application requirements with appropriate node capabilities made tractable with the help of machine-learning based workload characterization. Our approach builds on existing serverless platforms such as Kubernetes and OpenFaaS. OpenFaaS is a Function-as-a-Service platform that uses Kubernetes as its deployment platform. The Kubernetes scheduler uses simple heuristics to schedule containers to cluster nodes. We extend the scheduler to make it workload-aware by adding three scheduling constraints that focus on: (1) performance, (2) preventing resource contention and (3) matching applications with appropriate nodes. We enable these constraints by (1) extensive profiling, (2) subsequent workload characterization and (3) solving the problem of matching applications with appropriate nodes.

We evaluate our approach by running simulations with three different scenarios and focus on AI-based applications. The results show that in edge computing scenarios the Function Execution Time (a key performance indicator) can be reduced by 33% to 68%. Moreover, performance degradation, caused by resource contention, can be reduced by 45% to 57%.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Container orchestration services help Cloud Service Providers manage container deployments. Containers bundle applications as portable images, enabling Platform-as-a-Service (PaaS) providers to remove the burden of managing hardware from users and let them upload images for deployment. This service offers automatic scheduling, autoscaling and self-healing capabilities [8]. Various domains, such as Big Data, scientific or edge computing have adopted this deployment strategy [8]. While data-centers were designed with homogeneity in mind, current application trends show a tendency to favor hardware accelerators in various forms [7, 25, 45]. Domains like Artificial Intelligence provide support for acceleration, speeding up training and inference times[2]. Therefore, large cloud centers are adopting and build heterogeneous clusters [45].

Besides traditional cloud-centric PaaS offerings, new deployment strategies and paradigms are emerging. Serverless computing abstracts the underlying hardware even further away, making the code for a function the unit of deployment. In this thesis we use the definition for Function-as-a-Service (FaaS) and serverless computing described by Jonas et al.[33]: cloud offerings that let the user upload application code, require no administration, scale automatically and bill per usage are serverless computing.

The edge computing paradigm shifts computation resources in near proximity to end-users, fulfilling latency requirements for real-time applications, like Virtual or Augmented Reality. Besides user-facing apps, the growing bandwidth needs posed by the Internet of Things can be reduced to a considerably low amount when utilizing edge devices to preprocess raw data [61].

Serverless computing emerges as promising model for managing and deploying Edge applications [32]. Especially Edge AI, that consists of a highly dynamic environment with focus on AI, is of interest for us and builds the foundation for our evaluation scenarios

[55, 71]. Use-cases comprise different environments, such as smart city [53], cognitive assistance [53, 67] and video analytics [2]. Though it should be noted, that our work is not restricted in terms of applicability and can be used in any FaaS scenario.

Container orchestration platforms, like Google's Kubernetes[1], Apache Mesos[2] or Docker Swarm[3] are used to manage clusters of nodes and provide the functionality necessary to address issues mentioned before, automatic scaling and placement[5].

While data centers become more heterogeneous [7] and even instances with same specifications may run on different hardware [37], heterogeneity in edge computing scenarios is much higher and current container orchestration services have been shown to produce poor function placements in these environments [69].

We conducted experiments that show a substantial increase in execution time between common edge and cloud devices. Our focus lies on AI applications, such as Resnet50 [26] (object classification) and DeepSpeech [23] (speech-to-text).

1.2 Problem Statement

Current container orchestration platforms are not able to autonomously match application requirements and node capabilities when placing services. Users can select CPU or RAM thresholds to prevent resource contentions but performance degradation in multi-tenant environments, where multiple applications run on the same host, are still possible[37]. Developers can tune the Kubernetes scheduling by labeling applications and nodes to influence placements. Labels are used to describe deployed services and hosts, which Kubernetes uses to rate possible hosts. There remain two caveats: first, users and providers have to know which labels to use, which requires knowledge of application requirements and node capabilities. Second, labeling has to be done manually and is not feasible in scenarios of hundred containers.

We propose a solution that solves the problem of matching workloads to their appropriate computing resources in a dynamic and scalable fashion. Further, we tackle resource contention, decrease execution time and focus our work on edge scenarios using serverless computing platforms for deployment.

To that end, we need to solve two specific problems. First, a systematic approach to describe clusters is necessary. The descriptions need to be flexible, because in heterogeneous clusters new types of devices with new capabilities may appear. Second, to prevent resource contention and perform capability matching it is necessary to introduce a workload characterization, which contains multiple resources and should be scalable regarding different types of architectures and hardware. This step builds the foundation for improving the existing scheduler of Kubernetes by making it aware of workloads and node capabilities.

[1] https://kubernetes.io/

[2] http://mesos.apache.org/

[3] https://docs.docker.com/engine/swarm/

1.3 Research Questions

- RQ. 1: What are appropriate methods for workload characterization based on black-box system metrics in serverless edge computing systems?

  Edge computing is characterized by an unprecedented level of computing infrastructure heterogeneity. This heterogeneity leads to a much higher variance in resource consumption and performance, compared to cloud computing, where data center hardware is relatively homogeneous. The differences make it hard to reason about the impact of applications on resource consumption, and potential performance degradation. Platform providers generally have no way of analyzing user code upfront, and cannot rely on users to instrument their code. For effective workload characterization, we therefore require a method that works with black-box system metrics the platform provider has access to.

- RQ. 2: How can we use workload characterization in scheduling of serverless edge functions?

  A main concern of serverless computing platform providers is the placement of functions on computing infrastructure in a way that balances resource usage and application performance. Many different factors can and should be considered during the scheduling process. Providers may focus on lowering power consumption or guaranteeing a certain level of performance. The environment of edge computing makes scheduling even more challenging due to the high level of heterogeneity. By obtaining information about resource requirements of applications on a per node basis, we can aid the scheduler by implementing strategies to make it workload-aware.

- RQ. 3: How can a workload-aware scheduler improve the quality of function placement in serverless edge computing systems?

  Heterogeneous clusters offer different kinds of specialized hardware, capable of performing certain tasks very efficiently. Most devices at the edge are resource-constrained and therefore can experience high performance degradation in multi-tenancy situations. We add awareness regarding performance, resource contention and capabilities to the scheduler. Enabling transparent deployment through serverless platforms and preventing random placements.

1.4 Solution Approach

The aim of this work is to solve the problem of workload-aware scheduling. In other words, to match application requirements and node capabilities, such that we can reduce the function execution time (FET) and prevent performance degradation by making the scheduler workload-aware. Our approach is based on workload characterization, which we achieve through profiling real-world edge computing infrastructure and applying machine learning. We use machine learning to cluster similar applications based on their resource usage. We define an optimization problem with the objective of creating requirements for application clusters, which describe preferred node capabilities. By using clustering we guarantee generalization regarding a growing number of new functions.

The goal of our work is to provide solutions for workload characterization, the subsequent clustering, an optimization approach for the matching node capabilities with application requirements, extensions to our serverless computing platform of choice, and a model to predict performance degradation, which is used in simulations. In the spirit of serverless computing, our resulting solutions work with an industry-proven container orchestration system, acting as a Function-as-a-Service platform in a cloud-to-edge scenario.

To gather data for our workload characterization, we deploy an experiment framework[4] , to profile applications, and a black-box monitoring[5] agent to collect runtime data.

Our overall goal is to dynamically match workload characteristics to the appropriate node capabilities. The matching builds on a systematic description of heterogeneous clusters. Through monitoring data, workload characterization, and mapping capabilities to applications, we influence and improve placement. This results in reduced function execution time and prevention of performance degradation, caused by resource contention. We focus on performance in terms of execution time, which is complementary to the approach of Rausch et al. [58] that focuses on data-locality or proximity.

1.5 Structure

This thesis is structured as follows. Chapter 2 presents the fundamentals of context and techniques used. We motivate our focus on scheduling functions by introducing newly emerging application scenarios. Afterwards, we explain underlying technologies that can realize these services and introduce the core problem of our work. Chapter 3 presents related work. We focus particularly on function placement and container scheduling. Afterwards, in Chapter 4 our approach is explained in detail and each component we develop presented. Before presenting our results in Chapter 6, we explain our evaluation scenario and the simulation settings we set. The results are discussed in Chapter 7. We conclude our work and present future work in Chapter 8

[4] https://github.com/edgerun/galileo

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

CHAPTER 2

Background

This chapter presents the fundamentals necessary to understand our work. We start by introducing the context of our applications and use cases, followed by a promising architecture and our used framework. The section concludes with the problem matching applications with node capabilities and our selected heuristic optimization technique. Edge Intelligence and smart cities offer the possibility to implement visionary applications, such as Cognitive AR and contextual AI [53]. To enable these scenarios, AI pipelines focus on automating the life cycle of these new applications [53]. The current cloud-centered lacks certain requirements to realize these concepts. Real time applications require ultra low latency, which is not possible considering the distance between user and cloud. Further, in the smart city concept sensors are deployed throughout the city. It is expected that sending the raw data to the cloud is not feasible and therefore requires preprocessing at the origin [61]. Edge computing is an emerging paradigm that promises to have all prerequisites. Computing resources are moved into near proximity of users to process data and offer low latencies [61]. These environment are heterogeneous in nature, by using specialized types of devices. To support developers deploying their applications, the serverless model seems promising by providing the right amount of abstraction regarding deployment. Serverless edge computing is the combination and is currently driven by frameworks like Kubernetes and OpenFaaS. Placement of functions is very important and therefore we introduce the problem of matching applications with appropriate computing resources and a heuristic optimization technique.

2.1 Edge Intelligence

The cyber-human evolution is the process of melting our lives with digital information, starting with the internet providing space to store and retrieve data, which we access through smartphones [53]. They are already thoroughly integrated in our day-to-day lives and builds the first step in our cyber-human evolution. User focused technologies such as mixed reality smart glasses, combining Augmented Reality and AI are the next step to connect humans, and the abundance of data from smart cities can make this possible [53]. Further plausible use cases include automatic annotation of shops and restaurants or intelligently giving contextual data about traffic conditions. By setting up compute resources near the users, at the edge, sending data to cloud resources can be skipped and processed locally. This leads to reduced bandwidth requirements, liberates from external resource capacities, mitigates long latencies and enables real time processing.

To realize this, independent AI agents need to be distributed throughout the world, which autonomously collect and process data, offering services in highest quality [53]. The cloud cannot compete with this in terms of processing power and highly contextualized models. To give a perspective to this claim, according to Cisco global data center traffic will reach up to 20 ZB, while they estimate this number for the edge as high as 850 ZB [71]. Gartner estimates that 80% of deployed IoT applications will use AI [71].

Models can be trained in near proximity to data making it high fidelity in their context. For example an object detection model in a flower shop needs different knowledge than one in a hardware store. Context is important, and while current object detection networks, such as Resnet50, have proven to be reliable, it is not feasible to train a single model that generalizes across domains [29].

This complex system requires a transparent connection between cloud and edge resources. Rausch et al. [53] propose a system comprised of a computational fabric and sensing substrate to realize this. The computational fabric, consisting of highly heterogeneous devices for training, monitoring, preprocessing and inference interweaves all the aforementioned components. A sensing substrate, realized by IoT devices, collects information about its environment, which the computing fabric processes and scheduling depends on intelligent resource orchestration. Research considers the serverless computing model as viable approach to tie all resources in the edge/cloud continuum together to make the most out of the hardware [3].

In summary, these infrastructures provide complex insights, exploit locality, deliver yet unseen performance and can bring the power of AI to users. This vision is called Edge intelligence: a closed loop system, in which data is generated by users or sensors, processed in near proximity by edge devices, giving the advantage to exploit local and contextual information, providing better results and sending preprocessed data to the cloud, saving bandwidth and resources [51]. The cloud distributes the initial models, whereas edge nodes fine tune them with contextual information.

Smart cities implement this vision and deployment has already started.

2.1.1 Smart City

The general idea behind smart cities is to distribute nodes throughout the city. One specific concept is called Urban Sensing and revolves around dispersed sensors that emit data upon services can act on [56]. In this setting we have a sensing substrate, which offers different types of data (i.e., air pollution) and a computing fabric, consisting of nodes with different capabilities.

This sensing infrastructure got implemented in multiple cities [65, 11] spreading nodes with cameras and sensors to generate data. A project that is important for our evaluation is the Array of Things located in Chicago [9]. Their urban sensing infrastructure consists of multiple nodes that contain an array of sensors and SBCs. These topologies produce data that can help monitor air pollution, road traffic conditions and offer possibility for video analytic at the edge. Besides hardware and data availability a driving factor of these advances is adoption and development of Artificial Intelligence allowing developers to solve problems years ago were not manageable [35].

The last component, in Rausch et al.'s [53] outline, is the intelligent application orchestration for these scenarios. The concept of AI pipelines is a step towards autonomously managed AI applications, and therefore plausible in these scenarios [55].

2.1.2 AI Pipelines

AI Pipelines fully automate the lifecycle of AI applications. This lifecycle consists of: preprocessing, training and deploying [54]. An analogy is the modern trend of Continuous Integration, which describes a reactive approach to re-deploy applications upon code changes with automatic testing, compiling and deploying [64]. Instead of code changes, concept drifts of the model act as trigger for redeployment. That is the problem of a the degradation of a model's accuracy due to new data and making it necessary to update it [54].

There are already systems available, such as IBM Watson OpenScale[1] and ModelOps [28]. Though they pose the restriction of a cloud centered view. Systems such as Google Cloud IoT Edge [2] are integrating the edge as inference layer, but to accomplish a full integration, resources at the edge must be considered for all operations. Besides the general importance of these AI pipelines they are used in our evaluation and in combination with the Urban Sensing environment act the foundation of our experiment scenario.

These hybrid edge cloud systems would enable aforementioned examples: the cloud is used for base model training, while edge resources learn the contextual information [53]. Before presenting background about our serverless framework of choice we give a brief introduction into neural network designs. Normally, they tend to be large in size but research has shown ways of mitigating this issue. This is especially important in environments with resource-constrained devices.

[1] https://www.ibm.com/cloud/watson-openscale

[2] https://blog.google/products/google-cloud/bringing-intelligence-to-the-edge-with-cloud-iot/

Neural Network Designs

Deep Neural Networks have achieved tremendous success throughout different domains and reaching accuracy never seen before [35]. One caveat of those designs are high memory requirements. Determined by the size of the model itself, but also imposed by working memory requirements for storing intermediate results, reaching sizes of hundred MBs [51]. Ongoing effort is made to decrease these requirements to fit it on even small Edge Devices while keeping the accuracy in mind [10, 52, 13]. A very popular method to achieve memory efficiency, targeted at edge devices, is quantization and is used by Tensorflow TFlite, which aims to combine lower computation requirements and high accuracy [31].

In our evaluation we use both kinds of Tensorflow networks, normal and TFLite, to create a realistic evaluation.

After presenting the context of our applications and the Edge Computing paradigm, we introduce a promising approach to realize these concepts.

2.2 Serverless Edge Computing

The growing Internet of Things with its inherent large amounts of data are substantially responsible for the emerging trend of *edge computing*. Besides these services of analytic nature, user facing applications (i.e., Augmented Reality) require real time processing and low latencies that the push of computing resources to the edge can offer [63]. Figure 2.1 illustrates key differences in invocation and data usage between cloud computing and edge computing. While in the former scenario users talk directly with the cloud, much more data has to be transmitted and latencies may be high due to long distance. On the other hand edge computing solves both problems by pushing nodes in near proximity to users.

This paradigm focuses on shifting computational resources in near proximity of the users to process data at the origin and reduce bandwidth needs, solving problems concerning privacy by anonymization of data before leaving the users and offer resources for low latency applications [61]. The enabler and caveat of this approach are highly heterogeneous clusters. Devices include simple Single Board Computers (SBC), specialized embedded GPU hardware and cloudlets, small factor computers that can be placed anywhere [62]. Heterogeneous clusters offer various possibilities: low latency inference through special hardware accelerators in the size of a coin[3], accelerated training at the origin of data [71]. One downside is the inherent complexity of such an heterogeneous environment, making it hard to reason about optimal placements for applications.

Edge Computing is a symbiosis between applications situated in the cloud, offering a strong backbone to withstand peak workloads, and deployments at the edge to guarantee performance and reduce data flows. Current cloud-centric applications are hosted at

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

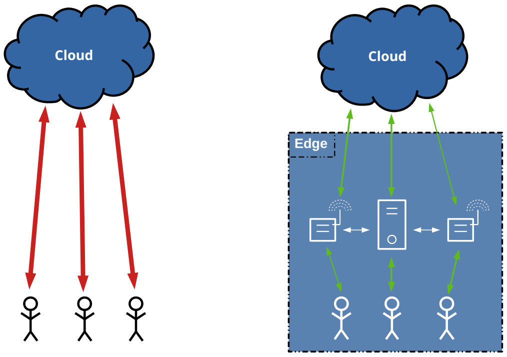


Figure 2.1: Cloud vs Edge Computing

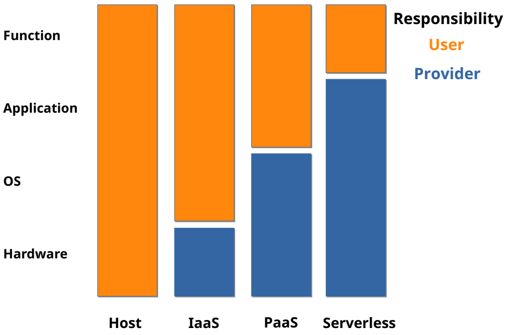


Figure 2.2: Cloud models and responsibilities

Cloud Service Providers). They offer different models, each dictating the level of control users have of the underlying hardware. Figure 2.2 shows the different services we explain in the following and displays user responsibilities, which indicate parts that users have to manually manage. Infrastructure-as-a-Service (IaaS) offerings let users rent bare Virtual Machines with a specified amount of CPU cores, RAM, disk space and possible accelerators like GPU or TPU. Users need to setup and manage everything besides the hardware. Platform-as-a-Service (PaaS) relieves users of OS management and lets them upload their application in a container, which contains all programs for hosting capabilities. Both approaches are cumbersome in the face of edge computing as users need to know which hardware is most suited, and with IaaS they have to manually manage the devices.

The last model removes any involvement of users from the process of deployment and may offer the right amount of abstraction regarding deployment decisions and much needed support for users. *Serverless Computing* has emerged from the adoption of containerized deployments in cloud infrastructures using Container Orchestration Services [4]. This paradigm removes the burden of renting and selecting instances by automatically deploying, scaling and managing applications. We consider the Function-as-a-Service model in this work as Serverless Computing platform [33]: users submit containers that serve a single function, which is called via REST.

In heterogeneous clusters users are faced with many types of nodes, that all may differ in performance and capabilities. The approach of *Serverless Edge Computing* tries to mitigate this issue by employing automatic node selection through Serverless Computing making it suitable for Edge Computing to its full potential [17, 46].

Rausch et al. [58] state three reasons why current state-of-the-art serverless platforms lack substantial compatibility with the edge compute fabric vision, presented in 2.1.

1. Proximity and bandwidth between nodes are not considered, which leads to higher latency times, as distances between compute & storage nodes can have a big effect on the responsiveness.

2. Data management has to be done by hand and leaves platforms in the dark with regards to data locality and storage nodes.

3. Platforms provide limited support for hardware accelerators, important for the vision of a true edge fabric.

Our work focuses on the last point and is therefore complimentary to [58]. We also re-evaluate their work with our newly developed framework to compare advantages and limitations of both approaches but to also take a combined look when fusing them together.

2.2.1 Kubernetes

Kubernetes [4] is a container orchestration service and is de-facto the industry standard. In the following we give a brief introduction into the inner workings of Kubernetes and only highlight elements relevant to our work. It is therefore not complete and kept simple on purpose. A detailed explanation of the scheduler follow in Section 2.3.1.

A container orchestration service manages a cluster of nodes and is responsible for scheduling, autoscaling and self-healing of applications, which are packaged as self-contained containers [8]. The smallest unit of deployment in Kubernetes is called a Pod and groups one more containers. We assume in this work, that every pod contains a single container. This assumption is plausible, as we are considering functions that only serve a single purpose. Devices, used to run the pods, are called nodes. In Kubernetes each pod and node can be described using labels, which are simple key-value pairs. One component of great importance for us is the scheduler. It receives a single pod, for which it finds a suitable node. Fortunately, the scheduling system is very flexible and can be easily extended as we present in Section 2.3.1. As already mentioned, we skimp over details of other components, because it is more important to understand how pods, nodes, labels and the scheduler work together.

To put this system into perspective Figure 2.3 shows a conceptual diagram of the previously described elements and their interactions. The scheduler receives a pod, performs filtering and scoring of feasible nodes, and selects the highest ranked one. Before turning to the scheduling of functions, we introduce serverless computing, including our framework of choice.

2.2.2 Serverless Computing Platforms

Historically seen, was serverless computing made popular by Amazon in 2014 [39, 1]. Google [19], Microsoft [41] and IBM [30] followed Amazon's platform with their own offerings in 2016 [4]. In contrast to PaaS, developers are bound to a certain programming model. The unit of deployment is code that represents a single *stateless* function,

[4] https://kubernetes.io/

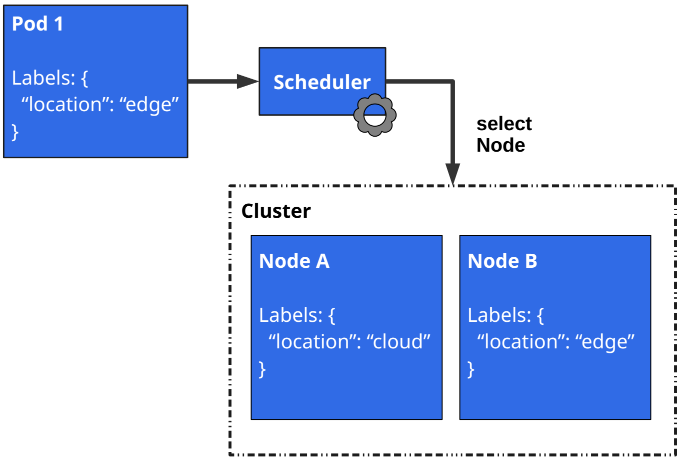


Figure 2.3: Conceptual view of Kubernetes deployments

commonly called via HTTP. Based on the aspect of statelessness and that the code serves only one purpose, platform providers are able to quickly scale the applications. These constraints also represent the main difference to the Platform-as-a-Service model, capable of hosting arbitrary complex and stateful services. Additionally the Function-as-a-Service (FaaS) naming has been established for specifying Serverless computing platforms that focus on the aforementioned type of applications [33]. The change of programming model also leads to a different pricing system. In PaaS it is normal to base this on the rented VM instances for which users are responsible to turn on or off and which may run without any requests. The FaaS system's main selling point is that it scales rapidly and in times of no requests users may pay nothing[3]. Our platform of choice is OpenFaaS, based on the fact that our foundation is build on previous research efforts [58]. Therefore, we focus in the following on explaining relevant internals of this platform.

OpenFaaS

OpenFaaS is an open source Function-as-a-Service framework[5] and uses Kubernetes as execution runtime and deployment platform. Users submit functions as containers to the OpenFaaS gateway, which are translated into pods and submitted to the Kubernetes scheduler. Initial, maximum and minimum numbers of pods are defined in the *Function-Definition*. This entity contains all information about a function and additionally includes: the docker image to pull, a scale factor, a boolean indicating whether *scale to zero* should be deployed and the labels describing a function, Table 2.1 lists all attributes. While

[5] `https://github.com/openfaas/faas`

Attribute	Description
name	function name
image	docker image
labels	key-value pairs describing function
scale min	minimum numbers of replicas
scale max	maximum numbers of replicas
scale zero	whether zero replicas are allowed

Table 2.1: FunctionDefinition

at first glance this level of abstraction seems to be sufficient, we encounter limitations during our work that we highlight in Section 4.4.1.

Figure 2.4 summarizes the important steps that a function has to go through when getting deployed. At first, the user submits a FunctionDefinition to OpenFaaS, the system creates from this definition the necessary number of pods, defined by *scale min*, and submits them to the Kubernetes scheduler.

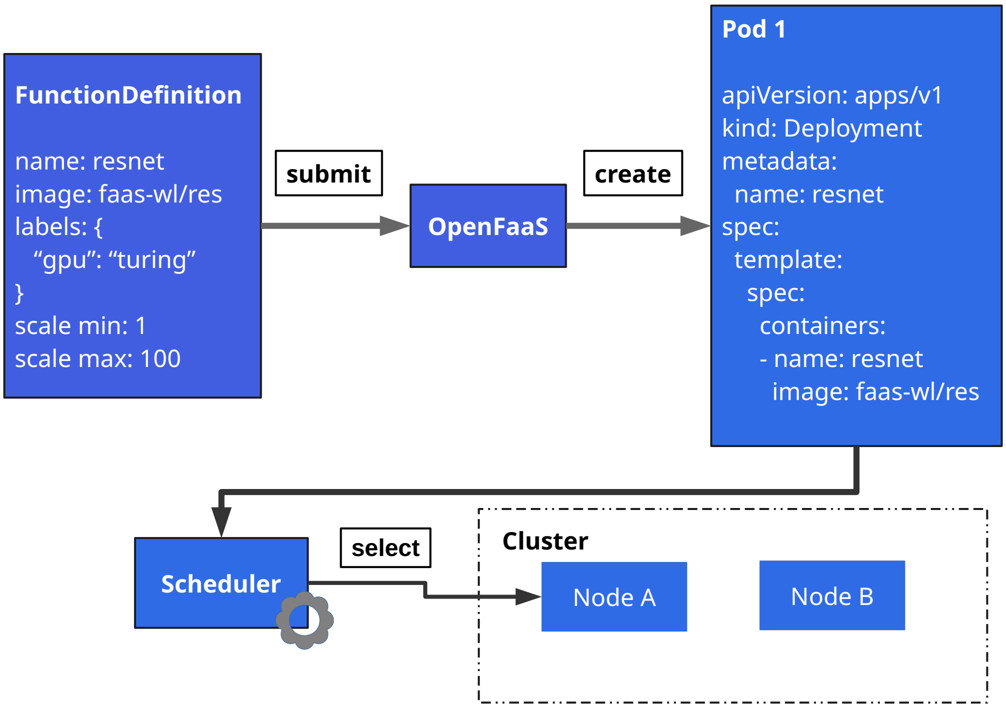


Figure 2.4: Function submission process

Users can choose from various languages, though our choice is Python for all evaluation applications, as it is the predominantly used language in the context of AI applications. Besides choosing a programming language, users need to make another important decision.

OpenFaaS calls function by using runtime environments, called watchdogs.

Watchdogs

An important decision to make when designing functions with OpenFaaS is choosing the watchdog. Watchdogs elevate the boilerplate code necessary for handling invocations. Users can choose from multiple templates, targeted at programming languages, or design their own ones. These templates provide basic networking code, and users only have to implement a handler function, taking the request body as argument and returning the response. In essence, a watchdog implements the actual behavior of handling a request.

The watchdogs are designed to allow different modes of operation. The most important ones for our thesis are the HTTP and Forking types.

At the beginning all modes start a *parent* process, responsible for receiving and forwarding any HTTP calls. In case of the *forking* mode, a new child process is spawned for each invocation. This removes any concurrency issues from the developer but causes overhead by starting a new process. Additionally any initialization steps have to be run repeatedly which can incur major latencies.

The *HTTP* mode mitigates the issue of setting up a new process for each call, by starting an internal HTTP server, which calls the actual user code. In case of Python, our language of choice for all functions, a fixed worker pool is used. While this approach allows users to benefit of caching, they must handle concurrent access and face limitations from Python's thread management. Due to the GIL (Global Interpreter Lock) only one worker executes code at a time and queuing is introduced, halting requests in case no worker is available.

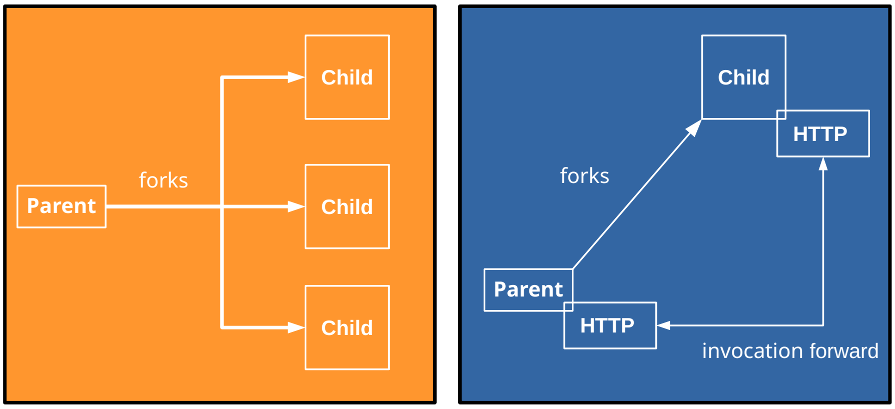


Figure 2.5: Forking and HTTP Watchdogs, based on [49]

Figure 2.5 shows the watchdog modes we use. The left presents the forking mode which spawns for each call a new child, while the HTTP mode spawns a single child and communicates internally via HTTP with it. We discuss in Section 5.3.3 the different modes in the context of our reference applications and highlight our experiences with regard to the FaaS paradigm in general.

This concludes the background with regards to our context and we now focus on scheduling functions.

2.3 Serverless Function Scheduling

The section starts by explaining the Kubernetes scheduler in detail. Afterwards, we introduce an optimization problem that aims to map application requirements to node capabilities. The solution to this problem should support the scheduler to be able to reason in a highly heterogeneous cluster, such as edge computing infrastructures represent. Our chosen heuristic optimization technique is described at the end.

2.3.1 Kubernetes scheduler

The scheduler is built in a highly extensible way, allowing us to easily extend and help it make informed placement decisions. Scheduling is done in an online manner. This means that the scheduler receives a single pod, schedules it and handles the next one —it has no knowledge of future deployments.

The scheduling process consists of two phases: filtering and scoring (hard and soft constraints). The first one removes not feasible nodes, the latter scores them individually and picks the highest scored one. Developers can implement *predicates* to filter nodes and *priorities* to score them. This allows users of Kubernetes to influence the placement process. We present our own predicates and priorities in Section 4.4.3. In the scoring phase a weighted sum over all priorities is created and developers can manually choose the weight of each soft constraint.

While the preceding description has explained the process in an intuitive way, we now introduce a formal definition based on [58].

$\mathcal{S}$ is a set of priority functions $s \in \mathcal{S} : P \times N \mapsto \mathbb{R}$. P represents the pods and N contains all nodes. The scheduler invokes a function $schedule : P \mapsto N$, to retrieve the node a pod should be scheduled on. *schedule* calls for a given pod p and all feasible nodes n every *priority* function, combining the outputs into a weighted sum. This process can be formally expressed as:

$$schedule(p) = argmax_{n \in \mathcal{N}} score(p,n) : \sum_{i=0}^{|\mathcal{S}|} w_i * S_i(p,n)$$

As already indicated, our approach to improve placements is deeply connected to the scheduling process. In essence, we develop three priority functions which influence pod placements to meet our performance and resource oriented goals.

We now introduce an optimization problem to realize a scoring function, which is able to judge the fitness of nodes for an application.

2.3.2 Capability matching

Edge computing infrastructure are characterized by a highly diverse set of devices. Not only are there different types of accelerators, but there is also a big difference in terms of hardware and performance. Kubernetes has a very developer friendly technique to filter and score nodes, which we utilize to integrate the capability matching problem into the decision making. We argue that in our scenario it is necessary to match requirements of functions with the capabilities of nodes. We formulate this as an optimization problem, but first give more details behind the reasoning on why we think it is necessary to consider this problem.

Capability matching is rather abstract and demands a clear specification before solving it. Our interpretation is presented in Section 4.4.2, but the intuition behind it is to match applications with nodes based on their descriptions. In our case, these consist of hardware specifications (i.e., number of CPU cores), but are left open for extensions. Afterwards, we need to find requirements for applications. These are determined through an optimization step and are used in a priority function. In our solution a requirement consists of capabilities, each having a score assigned. A node's favourability increases, in case a capability matches with one of the requirements.

In Chapter 3 we present previous works that solve this problem by manual assignments. In our opinion this may be feasible in cases the following statements hold true:

1. Developers know their application's needs the capabilities of deployed nodes.
2. Services have hard constraints (i.e, software that require certain static resources: accelerators)
3. A homogeneous cluster configuration. In this case all nodes have the same capabilities and therefore placement is irrelevant.
4. Discarding performance interference through co-located applications.

The possible scenarios we consider in this thesis do not fulfill any of these criteria. Transparency is important. By using FaaS, users can submit functions and do not care about the place of execution. While we can assume that developers are aware of possible hard constraints, delicate differences in hardware, that may not be obvious at first, hinder the approach of manual input. Obviously, we can not assume a homogeneous cluster and as we show in Section 4.3.5, performance interference can not be left out. Therefore, we deem it intractable to manually assign labels.

2.3.3 Heuristic Optimization Techniques

Our approach formulates a combinatorial optimization (CO) problem which is NP-hard by a trivial reduction from the Knapsack Problem. We employ a heuristic technique to overcome this issue. Formally, a CO problem $\mathcal{P} = (\mathcal{S}, f)$ is defined by a finite set of solutions $\mathcal{S}$ and an objective function $f\colon \mathcal{S} \mapsto \mathbb{R}^+$, whereas f assigns every solution $s \in \mathcal{S}$ a non-negative value and the goal is to find a solution s^* with minimal costs [6].

2.3.4 Genetic Algorithms

To solve the Capability matching problem, we sue a genetic algorithm to search for optimal solutions.

Genetic algorithms are placed in the category of *Evolutionary Algorithms* (EA), which are based on the idea of natural evolution. These types of optimization techniques consist of three main steps. First, an initial population is created, a set of individuals. Each individual represents a solution for which a fitness score is evaluated by using the objective function f [6].

A termination criteria has to be defined and could be a number of iterations. Further, four operators have to be defined, that will be executed subsequently: selection, recombination, mutation and replacement. All act on the population of the previous result and are intended to imitate natural evolution.

First, parents are selected from the population based on their fitness, afterwards new individuals are derived by most often randomized crossovers and recombinations. The offsprings may mutate afterwards, creating a higher diversity with a random or heuristic character. The last step is to create the population that will be used in the next iteration or serves as the final solution set [6].

CHAPTER 3

Related Work

This chapter presents work related to the main topics of our approach. First, we motivate the context of our problem - function placement in serverless edge computing. Workload characterization builds the foundation of our work and is presented in the next section. Afterwards, we present related work to performance modeling. The next section is dedicated to scheduling approaches in general heterogeneous clusters and focus in the last section on AI pipelines.

3.1 Serverless Edge Computing

Serverless edge computing is an emerging paradigm to deal with the operational challenges of edge computing systems. Aslanpour et al. [3] present the vision and open challenges in this domain. They attribute the missing adaption to the cloud-centric design of current serverless platforms and the heterogeneous edge environment. Because of the unlimited available resources in the cloud, serverless frameworks have not been designed for resource-constrained devices. One open issue addresses the resource inefficiency and need for new lightweight systems as Docker images tend to be heavy. Further, serverless platforms have yet to support different computing platforms [3]. We introduce and implement a concept to mitigate this issue and support multiple computing platforms. The research highlights how many problems have to be solved before fully taking advantage of this paradigm in edge computing scenarios.

There are some frameworks that specifically target a seamless integration of edge and cloud. Approaches range from replacing parts of Kubernetes, to completely new systems. As mentioned, runtime overhead is still high for resource-constrained devices at the edge. An edge-focused all-in-one solution is *faasd*[1], which works with OpenFaaS and acts as

[1] https://github.com/openfaas/faasd

an extension to it. Containerd [14] is used as container runtime to offer a low footprint platform and effectively replaces Kubernetes[2].

Another approach, called FLEDGE, integrates with Kubernetes by placing virtual agents on edge devices and managing proxy pods in the cloud[18]. They claim to significantly lower memory footprint and test their application on a Raspberry Pi 3.

Another approach that aims to create new platform including container orchestration is *tinyfaas* presented by Pfandzelter et al.[50]. It is targeted at edge devices and uses the Docker runtime. While they do present the architecture, including their approach to function handler and the integration of CoAP —a low-resource friendly alternative to HTTP. Details about scheduling or placement is missing.

3.2 Workload Characterization

Dargie et al. [16] aim to characterize workloads based on resource usages in the context of cloud infrastructures. Their goal is to cluster workloads and consolidate them onto servers. Their categorization should allow placing applications that use different resources such that available host capacities are utilized without causing contention. They measure CPU, network, memory usages, read and writes to disks, storage or datastores. Important to them is the consideration of temporal and hidden dependencies between different resources. Tensor decomposition is applied to the recorded data and split into different views, which are analysed. The work focuses on analysis of the views and clustering remains future work. While our approach discards temporal aspects, we believe that our atomic unit of work, a single function, mitigates this.

Nemati et al. [47] present an agent-less monitoring technique, subsequent feature extraction and unsupervised clustering of VMs based on resources. It is similar to our approach as they also use KMeans, but differ substantially in the observed metrics. While we consider basic telemetry, like CPU usage or I/O written and read, they observe features concerning the average and frequencies of waiting times for resources, injection rates and frequency of VM preemption. Their applications cover CPU, network and disk and consist of standard benchmarking tools.

We build our workload characterization and apply machine learning on data gathered from workload profiling.

3.2.1 Workload profiling

Profiling of AI applications on Edge devices has been done by Zhou et al.[70]. They describe an accelerator recommendation approach for edge applications, such as video surveillance. They investigate the performance and power consumption of different devices and estimate the cost and effectiveness for each task per device. We investigate

[2] https://blog.alexellis.io/faasd-for-lightweight-serverless/

this issue further by adding co-located workload on the devices during execution and investigate the performance of training on these devices.

3.3 Performance Modeling

Grohmann et al. [20] investigate the issue of container interference and present a method to predict the inherent degradation to co-location of applications. Their evaluation focuses on key performance indicators, based on quality of service metrics. Services are considered to be structured following the microservice architecture and their resulting model works for heterogeneous applications. Metrics include CPU, memory and I/O throughput.

Han et al. [22] perform benchmarks on Amazon Web Services to investigate the issue of performance degradation on a public cloud service and try to predict the number of co-resident VMs based on performance measurements. This work is in stark contrast to our approach as we predict the factor of performance degradation to strengthen our simulation results. Relevant to our work is their approach of producing and quantifying performance degradation. Their benchmark applications are dedicated to create CPU, disk and network contentions and use ContainerProfiler [3] to profile and characterize resource utilization.

Perseus is a measurement framework to model performance and costs in multi-tenant situations specific to AI model serving. LeMay et al. [36] focus on hosting Neural Networks and sharing GPUs to reduce costs. Their solution provides users with approximations of performance between running the model in different settings, i.e.: multi-tenant GPU sharing or single-tenant CPU accelerated inference.

Moradi et al. [43] are situated in cloud scenarios and present user-level profiling benchmarks to measure contention and learn a function to predict target application performance degradation. In contrast to previously presented works and ours they do not collect resource metrics and build their prediction on that but for each application standardized micro-benchmarks are executed and afterwards the target service. Their benchmarks utilize CPU, memory and disk measuring the number of increments or count the accesses to memory and disk. Applications are called with the same input and they do not investigate the impact of differently sized requests.

3.3.1 Workload profiling

Profiling of AI applications on Edge devices has been done by Zhou et al.[70]. They describe an accelerator recommendation approach for edge applications, such as video surveillance. They investigate the performance and power consumption of different devices and estimate the cost and effectiveness for each task per device. We investigate this issue further by adding co-located workload on the devices during execution and investigate the performance of training on these devices.

[3] `texthttps://github.com/wlloyduw/ContainerProfiler`

3.4 Scheduling

Mao et al.[38] extend Docker Swarmkit to take cluster heterogeneity into account by dynamically labeling the main resource of their applications through constant monitoring and finding placements considering already possible co-located services. Further, they implement a migration strategy which detects hardware contention and migrates the most costly container. While they take heterogeneity into account their resources do not consider GPU as resource and focus on cloud deployments. Joseph et al.[34] consider the Microservice Allocation Problem which maps microservices packaged as containers onto appropriate hosts. They use Reinforcement Learning to learn the mapping, but only take CPU utilization and power consumption into account. Wöbker et al.[68] consider a distributed and heterogeneous environment, as can be observed in edge and fog computing.

There have been efforts to explore the advantages of a sophisticated labeling strategy for resource scheduling. Wöbker et al.[68] assign labels to their nodes and applications per hand and evaluate the resulting scheduling decisions. This approach lacks automatic mapping between application requirements and node capabilities. In contrast to their work, we focus on providing a solution that is capable of inferring requirement-capability mappings.

They recognize the need for matching requirements and capabilities for supporting such environments but do solve this problem only by creating the labels for applications and hosts manually. They do intend to develop this further for dynamic label recognition but to the best of our knowledge no future publication has investigated this issue. Mytilinis et al.[44] have proposed a scheduler which takes different hardware capabilities of each node into account, but is made with Big Data frameworks (i.e., Apache Spark) in mind and therefore expects a task graph as input. A recent survey investigates the problem of smart scheduling in cloud and fog environments, stating that the community is in dire need of smart scheduling approaches that integrate with state-of-the-art orchestration services. Havet et al.[24] combines runtime monitoring of containers to learn their requirements and properties and a scheduler that manages different generations of servers. They are inspired by Garbage Collection, where each generation of server has its own purpose and VMs are migrated continuously to older ones. They integrated their work in Docker Swarm. Santos et al.[60] extend the Kubernetes scheduler with a network-aware scheduling mechanism to support container-based applications in Edge deployments.

3.4.1 Scheduling AI Pipelines

Crankshaw et al. present InferLine, a system for scheduling individual stages of prediction serving pipelines in the cloud [15]. It consists of a low frequency planner responsible for selecting the right hardware for a stage and a high frequency planner that scales the stages. They focus on tight tail latency and balance between performance and costs. Cost is an important factor as they not only consider CPU but also GPU and TPU, which both may drastically more expensive. In contrast to our work InferLine plans the deployment of a pipeline end-to-end, whereas we schedule single functions without knowing their context or associated applications.

The most important related work for is about the Skippy scheduler we use as the Kubernetes scheduler replacement[58]. Rausch et al. present the Skippy, a Python-based re-implementation of the Kubernetes scheduler. Further they provide a trace-driven event-based simulator, *faas-sim*, and priority functions to improve placements of data-driven applications. Skippy can either act as scheduler drop-in replacement in a real cluster or in the context of simulations. Latter is important for us as we evaluate our approach using *faas-sim*. In contrast to Crankshaw et al. [15] focus on Serverless Edge Computing and their test applications represent the stages of a typical AI pipeline: preprocessing, training and inference. Though their approach focuses on data locality and discard performance characteristics of devices. Our work extends this paper in terms of implement priority functions which builds on a different approach to calculate resource contention and adds performance into the process.

CHAPTER 4

Approach

This section describes our approach to improve scheduling decisions in the context of serverless edge applications. First, we present a methodology to model heterogeneous clusters including a quantification of heterogeneity. Second, we introduce our performance model that provides our simulation with realistic data, enables workload characterization and helps us train a performance degradation model. We conclude the chapter with presenting our main contribution that improves container scheduling. We build on existing serverless platforms such as OpenFaaS and Kubernetes. Our approach addresses in particular two shortcomings of these platforms. We introduce first-class-citizen support for functions to use different computing platforms. Second, we enable the scheduler to reason about workload characteristics, allowing it to map workloads to their appropriate computing resources, as well as considering the impact of resource contention on runtime performance.

4.1 Overview

Before explaining our approach, we take a step back to view the problem from a high level perspective. Figure 4.1 shows the general context: users submit their functions to *OpenFaaSExt*, responsible for scaling and deploying of pods. These are submitted to Kubernetes, which is responsible to select a suitable node. All three components are important for us. First, we present an extended version of OpenFaaS - *OpenFaasExt* - that enables developers to ship a function targeting multiple computing platforms. In our use case this allows functions to support multiple accelerators. The second component, Kubernetes, gets extended by adding custom scoring functions. These rely on our workload characterization and capability matching optimization and should improve latencies and prevent resource contention. Finally we introduce a heterogeneity score for clusters and a strategy to describe a node's capabilities.

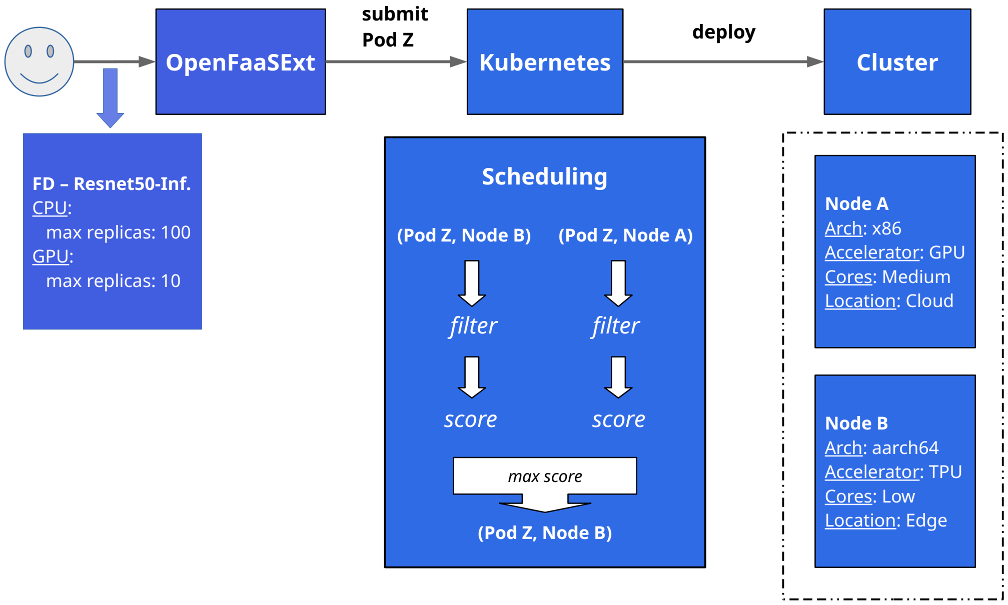


Figure 4.1: Look at the general context

Figure 4.2 gives a detailed look into evaluation pipeline and on which knowledge our proposed priority functions are based on. The first step is to obtain real performance and telemetry data from our testbed nodes. We spawn one or multiple clients that invoke our functions to obtain traces and telemetry. Generating load is done by *galileo*, an open-source experimentation framework [57]. In the first step we record baseline performance and resource usage for each node and function. The performance-oriented priority function is implemented using this data, favoring faster nodes.

The data is also used to characterize workloads, which produces a vector for each node and app. Using this characterization, we can implement a resource-oriented priority function and apply machine learning to cluster functions. The result of this is an assignment for each function to a cluster that is needed to make our capability matching optimization tractable in the face of a high number of functions, as we would otherwise need to solve the capability matching problem for each function. The optimization outputs requirements imposed by each workload group and is used for our capability priority function. These requirements contain a list of node capabilities, each having a value assigned indicating the favourability of this particular capability.

Besides that, we introduce an entropy-based approach to calculate the heterogeneity of a cluster. We use this function in two ways: to determine suitable heterogeneous clusters for our simulations, and is integrated into the capability problem optimization.

Not displayed in Figure 4.2, but important for our approach is performance degradation. To this end, we execute multiple functions at the same time to cause interference and performance degradation. Due to the highly different devices we encounter at the edge

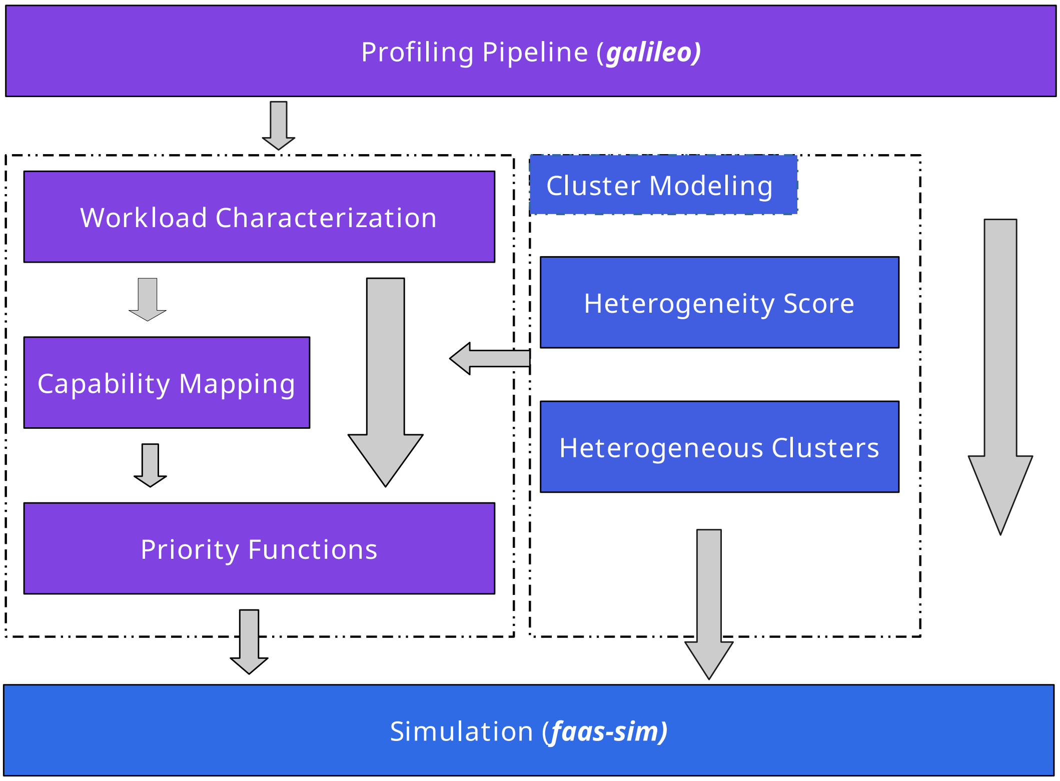


Figure 4.2: Component Overview

we train a model that predicts degradation based on resource usage. The model is used in our simulation to simulate performance loss due to co-located containers.

Using the baseline benchmarks, different compute clusters and our performance degradation model, we implement multiple scenarios in *faas-sim*. We compare different sets of priority functions regarding application placement. In the following we introduce each component in more detail, starting with the modeling of heterogeneous clusters.

4.2 Modeling heterogeneous clusters

Edge computing clusters contain many kinds of nodes, leading to a high diversity of capabilities. We believe that the simple standard strategy to spread applications evenly based on user defined CPU and memory requirements may lead to unexpected and undesirable placements. That's why we consider it important to evaluate our strategy on multiple clusters that differ in the distribution of node types and introduce a score to formally define the heterogeneity. Our approach for this problem is to define a representation of nodes, a subsequent extension to describe a cluster of nodes and our entropy based heterogeneity score function.

As a first step, we introduce a systematic representation of a node, which reflects the equipped hardware, basic information and other characteristics that seem relevant in a diverse scenario like edge computing. Each node n in the cluster has a description d. Descriptions consist of multiple attributes, and each describes a single capability, i.e. the number of CPU cores. We restrict the attributes to nominal ones and therefore have to set bounds for some attributes. The chosen characteristics can be automatically obtained, i.e. with the common Linux tool *lshw*[1]. Subsequent components in our work do not make any assumptions of attribute availability, leading to a very flexible design allowing extensions and modifications. This makes it possible to use different and more sophisticated tools to obtain characteristics from nodes.

Our attributes and their values are shown in Table 4.1. We think that these are sufficient and offer enough flexibility to create diverse descriptions, aiding our heterogeneity score. The bounds for originally continuously attributes are based on our testbed. While most of these attributes are trivial, we want to describe one in more detail. *Location* is the place in which a node is deployed. Due to the geo-distributed nature of edge computing clusters, we enumerate popular ones:

1. *Cloud*: nodes of this type are located in public or private clouds and managed by cloud providers. VM instances may have different capabilities, but are in general homogeneous.

2. *MEC*: an approach to provide services at the edge is undertaken by TELCOM providers through placing nodes at cellphone tower. Developers can use these nodes to deploy their services in near proximity to users.

3. *Mobile*: we believe that smartphones provide enough performance to host certain services on the user devices. Notable efforts are Tensorflow TFLite, which aims to bring inference onto every device and is supported by Apple's iPhone, Android phones and additionally provides a Javascript engine, capable of running in the browser. This leads us to believe that every device can provide host capabilities.

4. *Edge*: this category describes all nodes that are neither assigned to MEC nor Mobile.We introduce such nodes in our evaluation scenario. In general edge computing nodes can be deployed anywhere and therefore we are not able to enumerate every possibility.

After defining the representation for a single node, we introduce a notion for the same concept extended to a cluster of nodes. We call this description *Cluster Configuration* and shows the occurrence for each attribute and value in percentage across all devices. We illustrate this approach with an example. Table 4.2 displays three different types of nodes. A cloud VM equipped with a Xeon processor, high RAM and no accelerator and typical edge devices, such as a GPU-accelerated embedded AI board and a Single Board Computer. We provide in Table 4.3 three configurations with varying node type

[1] `https://linux.die.net/man/1/lshw`

Attribute					
Architecture	x86	arm32	aarch64		
Accelerator	None	GPU	TPU		
Disk	HDD	SSD	NVME	EMMC	SD Card
Location	Cloud	MEC	Edge	Mobile	
Connection	Ethernet	Wifi	Mobile		
CPU	i7	Xeon	ARM		
GPU	Turing	Pascal	Maxwell	Volta	
Bins	*Low*	*Medium*	*High*	*Very High*	
Cores	<= 2	<= 8	<= 32	> 32	
Network	<= 150 Mbps	<= 500 Mbps	<= 1 Gbits	>= 10 Gbits	
CPU MHz	<= 1000 MHz	<= 1200 MHz	<= 1500 MHz	>= 1700 MHz	
GPU MHz	<= 1000 MHz	<= 1200 MHz	<= 1500 MHz	>= 1700 MHz	
Network	<= 150 Mbps	<= 500 Mbps	<= 1 Gbits	>= 10 Gbits	
RAM	<= 2 GB	<= 8 GB	<= 32 GB	> 32 GB	
VRAM	<= 2 GB	<= 8 GB	<= 32 GB	> 32 GB	

Table 4.1: All attributes and associated values, numerical ones are discretized in bins.

Device	*VM*	*Embedded AI*	*SBC*
Arch	x86	aarch64	arm32
CPU	Xeon	ARM	ARM
RAM	High	Medium	Low
Location	Cloud	Edge	Edge
Accelerator	None	GPU	None

Table 4.2: Three different types of nodes and their attributes

distribution. The resulting clusters are presented in Table 4.3. The percentage of all attributes is relative to the total number of devices.

4.2.1 Entropy Score

Based on our cluster representation, we can introduce a quantification method for heterogeneity. We use entropy to define a function that obtains a score, which allows us to compare different configurations.

Let $h : \mathcal{C} \mapsto \mathbb{R}$ be a function that maps a cluster configuration to a real value - the heterogeneity score, with $\mathcal{C}$ being the set of possible configurations. Due to entropy being a function that compares two items, the second cluster is a completely homogeneous one. It describes a one node type, with each attribute having a single associated value that has the probability of one. Which leads our function to yield zero for every homogeneous cluster.

Cluster Configs	A	B	Base
Attribute			
Arch.x86	0.2	0.8	1
Arch.arm32	0.5	0.1	0
Arch.aarch64	0.3	0.1	0
CPU.Xeon	0.2	0.8	1
CPU.ARM	0.8	0.2	0
RAM.Low	0.5	0.1	0
RAM.Medium	0.3	0.1	0
RAM.High	0.2	0.8	1
Location.Cloud	0.2	0.8	1
Location.Edge	0.8	0.2	0
Accelerator.None	0.7	0.9	1
Accelerator.GPU	0.3	0.1	0

Table 4.3: Examples of descriptions for cluster configurations, refer to Table 4.4 for details about node distribution.

Cluster Config	Nodes	% of device type		
		VM	*Embedded AI*	*SBC*
A	100	20	30	50
B	100	80	10	10
Base	100	100	0	0

Table 4.4: Example cluster configurations

Algorithm 4.1: Algorithm for calculating the heterogeneity for a topology configuration.

```
Input : topology
Result: heterogeneity score
entropy_c ← 0
entropy_b ← 0
base ← getHomogenousConfig();
for attribute in attributes do
    $c$ ← topology[attribute] or $1e^{-}22$
    $b$ ← base[attribute] or $1e^{-}22$
    entropy_c ← entropy_c + c * log(c)
    entropy_b ← entropy_b + b * log(b)
end
return entropy_b - entropy_c
```

Algorithm 4.1 shows the implementation of h in pseudo code. Because *log* is not defined on 0, we use a default value of $1e^{-22}$, which we deem small enough to not influence the result. In essence, the function iterates over all attribute values and continuously updates the entropy for each configuration and returns the difference at the end. Note, that the highest obtainable score is 15.99, a cluster configuration in which all available values are present and uniformly distributed.

To give an example of what these scores might look like, we calculate them for the cluster configurations presented in Table 4.3, which return *3.67* for configuration A and *2.60* for B.

After describing the modeling of clusters, we start do describe the ML-based workload characterization approach. We start by introducing our performance modeling, which is integrated into the profiling step in Figure 4.2. And afterwards explain the characterization.

4.3 ML-based workload characterization

In this section we describe our model of performance and resources, followed by an approach to convert time series data into a format, which we can use as an input for our clustering technique of choice. At the end we present a model that can predict performance degradation.

4.3.1 Performance modeling

Our workload characterization is based on real data and therefore we use a benchmarking pipeline to collect information about performance and resource usage. We set up *galileo*, an experimentation framework [57]. It offers workload generation and stores function invocations and telemetry data. Through this streamlined process we can query data for our components from a single place. We explain two important types of data in the following: traces and telemetry.

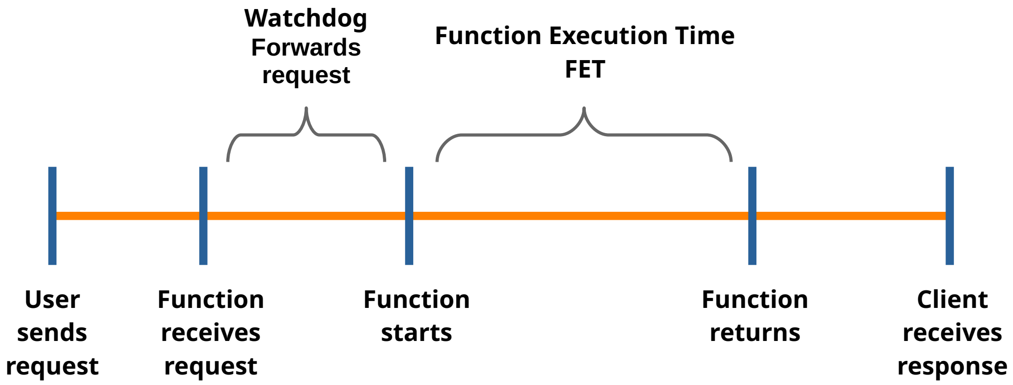

Watchdog Forwards request
Function Execution Time FET
User sends request
Function receives request
Function starts
Function returns
Client receives response

Figure 4.3: A detailed look into a function invocation

Metric	Values	Frequency
CPU Usage	cpu usage in nanoseconds per container	1s
GPU Usage	time spent executing GPU code/usage relative to frequency	1s
Network I/O	total network i/o usage in bytes per container	1s
Block I/O	total block i/o usage in bytes per container	1s
RAM	currently used RAM in kilobytes	1s

Table 4.5: Overview of recorded system metrics

The former is related to function invocations and stores timestamps and durations. Figure 4.3 shows different parts of an invocation and should clarify our definition of *Function Execution Time* (FET) - the duration in which the actual user submitted code is executed. Each call starts with a client sending the request, representing an invocation's start. The next timestamps indicate the server's reception and subsequent forward. Then the function is called, sends the result back, and after the user receives the request, the invocation is finished.

Besides measuring the FET, we also observe various resources, using a lightweight open-source monitor *telemd* [2]. It provides system-wide telemetry and additionally reports cgroup metrics —allowing monitoring on container level. Control groups (cgroups[3]) let users monitor and restrict resources of processes. This includes block I/O, network I/O, CPU and memory. *telemd* gives us the opportunity to track three resources for each container: CPU usage time, total block i/o and total network i/o processed. Relevant system-wide metrics contain memory usage and GPU load. Note that due to the different approaches of common x86 Nvidia devices and the Jetson series, the usage differs in definition: former returns results the time spent executing on the GPU relative to a time frame. While the latter is defined as GPU usage relative to the current frequency [4]. While they are not the same measure, our evaluation and results show that this is irrelevant to our work.

Table 4.5 shows a summary of all recorded telemetry. Every metric is measured once per second.

4.3.2 Baseline Benchmarks

The trace-driven nature of our simulation requires us to collect performance measurements from our devices. We invoke each function on each node multiple times with an interarrival time of one second without any interfering applications. These benchmarks are meant to establish a performance and resource usage baseline. The recorded data is fit onto

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

[3] https://man7.org/linux/man-pages/man7/cgroups.7.html

[4] https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%2520Linux%2520Driver%2520Package%2520Development%2520Guide%2FAppendixTegraStats.html%23

Timestamp	cgrp_blkio	cgrp_cpu	cgrp_net	gpu_util	ram
2020-12-07 12:22:24	0.0	0.024970	34763.0	0.0	3462012.0
2020-12-07 12:22:25	0.0	0.876353	11105269.0	99.6	3491612.0
2020-12-07 12:22:26	0.0	0.024215	34730.0	0.0	3462784.0
2020-12-07 12:22:27	0.0	0.857045	11102174.0	99.7	3493292.0
2020-12-07 12:22:28	0.0	0.067538	35484.0	0.0	3463268.0

Table 4.6: Sample of telemetry time-series data

a log function from which the simulator samples to simulate the invocation. We use the results for our simulation, performance oriented priority function and the workload characterization, which we explain in the following section.

4.3.3 Workload Characterization

This section describes our approach to convert telemetry data into a row vector and the clustering to group similar applications.

Our solution uses telemetry from the baseline benchmarks and uses k-means to cluster the implemented applications. k-means is part of the unsupervised machine learning techniques. They focus on finding patterns in data. The input is not labeled and output interpretation may require domain knowledge and is most often of exploratory nature[21]. Our goal is to find groups of applications that need similar resources. We perform this step to make our optimization problem tractable regarding a growing number of services. The classification can further be used to identify unknown applications with a single trace, such that existing knowledge about other services can be applied to new ones.

Due to the k-means clustering algorithm with Euclidean distance we have to preprocess our data into a vector of fixed length. Further, the distance function is highly sensitive to the value ranges and therefore all attributes should be in the same range to avoid a bias towards larger values. Because we use the resulting vector throughout our work, we have decided to use the k-means and not a clustering technique that is suitable for time-series data. Another positive side effect of our preprocessing is the much more interpretable representation of a function's resource usage.

Table 4.6 shows an excerpt of the recorded data from one benchmark. Columns prefixed with *cgrp* refer to the target service's usage, that is the service we want to record the baseline resource usage and performance. Data with regards to bytes, *cgrp_blkio* and *cgrp_net*, show the total bytes written or read since the last timestamp. *cgrp_cpu* shows the CPU usage time in seconds since the last measurement was taken, the same holds for *gpu_util*. On the other hand *ram* refers to the system's total memory used.

Based on the presented data we calculate different measures that all represent a single trace and therefore mitigates the issue of time series data. We calculate the average resource consumption for one request. This is done by using the invocation's FET. Data

Metric	**Value**	**Scaled**
CPU	0.184	0.184
RAM	0.34	0.34
GPU	0.36	0.36
Block I/O Total	1310.72	0.0
Net I/O Total	599 308.6	0.0005
Block I/O	286.01	
Net I/O	131 431.4	

Table 4.7: Example for workload characterization vector

related values are processed as data rates, which represent the amount of bytes that were written or read per second.

The final workload characterization for a single application and device consists of: Mean CPU & GPU usage, block and net I/O data rates and total amount of bytes written or read and mean memory consumption scaled to the device's maximum capacity.

Table 4.7 shows the resulting characterization from the telemetry presented in Table 4.6.

The result of our workload characterization component is a vector containing resource usage of a single function invocation on one particular node. These vectors are used by two different components: the resource contention-aware priority function and as input for our clustering which we introduce in the next section.

4.3.4 Clustering

The output of our workload characterization per function consists of one vector for each node, which is not suitable for our clustering algorithm. Therefore, we aggregate the vectors by using the mean for each value resulting in a vector of fixed length. Due to the fact that data rates highly depend on the executed device we omit them for the characterization step.

As the results are highly influenced by the size of each attribute, it is necessary to scale Block and Net I/O values. Our approach is to scale them between 0 and 1 using a min-max approach, keeping the original distribution intact. We take the minimum and maximum for both separately and over all applications. The results of scaling for our telemetry data from Table 4.6 can be seen in Table 4.7.

After preprocessing the telemetry data of each function into a single vector it is possible to cluster them using k-means. These groups of applications allow users to recognize possible hidden similarities between functions. In our case, we mainly use the groups to support our optimization problem. Before turning to container scheduling, we describe our implementation of predicting performance degradation.

Service	CPU usage per call
S1	50%
S2	25%
S3	10%
S4	75%

Table 4.8: CPU usages for example services

4.3.5 Performance Degradation

Our approach to performance degradation is realized by employing a Machine Learning model. The model is based on regression, a technique to predict real values from observations [42].

Performance degradation happens in situations where multiple programs compete for the same resources and cause unexpected delays. With the current strategy of using containers for deployment, co-location and multi-tenancy happens all the time. Previous works have investigated the impact of resource contention in public clouds [37]. As our proposed deployment method is container based and multi-tenancy can happen even on the smallest devices, we cannot discard the impact on performance. Therefore, we propose a method to predict the performance degradation based on our workload characterization and extend *faas-sim* to take resources into account when estimating the FET. One caveat of this approach is the need to benchmark every device. Though we are unsure if this could be mitigated in a high fidelity simulator as results show that performance and resource contention varies between devices. It should be noted further that for this problem we omit the total amount of bytes written or read and use the data rates instead.

The problem is formulated as regression task, in which the input corresponds to the current resource utilization over all running containers and our target is the expected service degradation as factor relative to the baseline performance.

This means that a prediction of **1** estimates no degradation, while **2** would increase the FET by 100 per cent.

Input

Our workload characterization offers information about basic resources and GPU usage. Input modeling is built on intuition for the data. While it is plausible to sum up the consumption of each resource, we believe that this hides intricate details about the underlying distribution. In example if two programs are running and needs on a four core machine 2.5 cores and the other 1.5, they will at least have to share one core leading to a reduction of CPU usage time. But for of a single program that uses up all cores it will experience no delay. The sum of CPU usage is in both cases the same.

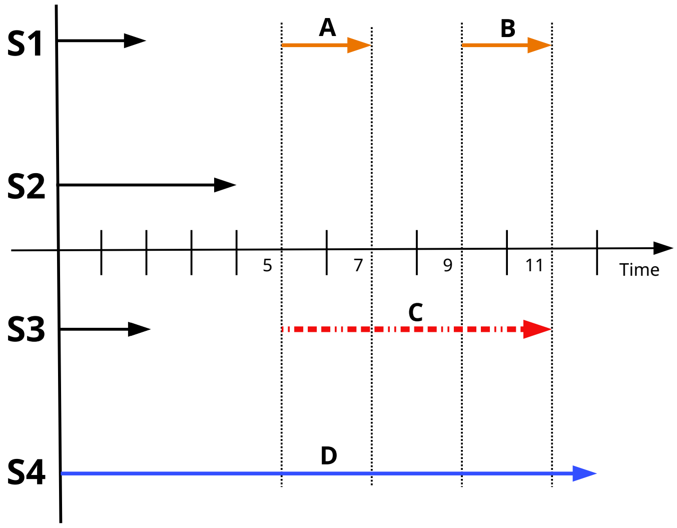


Figure 4.4: Example for calculation of input for the performance degradation model

Therefore, we decide to not only use the sum of each resource but to add various statistical description measures. The final length of the input vector is 34 and includes following measures for each resource (CPU, GPU, network and block I/O): mean, standard deviation, minimum, maximum, 25^{th}, 50^{th} and 75^{th} percentiles, the number of running containers, sum of each resource and the mean memory usage.

To help us illustrate the input calculation we introduce an example. This calculation is used in our simulation and to build our training datasets.

Figure 4.4 shows a timeline on which multiple functions are invoked. To simplify the example we only consider the CPU usage, though this approach applies in the same way to other resources. In our example four functions are deployed: *S1*, *S2*, *S3* and *S4*. The CPU usage for each service is shown in Table 4.8. As already explained, we measure CPU and GPU usages on a per request basis, this allows us to make estimations for calls of different length. For our example we want to calculate the degradation that can happen during the call of C. This one is interfered by A, B and D. For each interfering call we calculate the overlap time and multiply it by the CPU usage for each invocation to estimate the CPU time. A and B overlap for 2, while D does for 7 seconds. According to Table 4.8 this leads us to 1 second of CPU time for each call of A and 5.25 seconds for function D. We also consider the call C, which results in 0.7 seconds. The next step is to sum up the CPU usage time for each service. These sums are used to calculate mean, standard deviation and percentiles. We can apply the same procedure for GPU usage and the network/block I/O data rates.

Training

Our train and test datasets consist, besides the profiling data of our baselines, of benchmarks specifically designed to cause performance degradation. Because we use the same model for each service, we have chosen three characteristic workloads that focus on a single resource: CPU, GPU and block I/O. Further, we use a reference implementation of Resnet50 (object classification) in these experiments. Because every node is capable of hosting only certain amount of services, we have to decide in advance for each node the experiments. In essence, a typical VM instance with several gigabyte of RAM and an off-the shelf x86 CPU will be able to host multiple services and experience no degradation, while a SBC will either fail to host the same services or suffer from severe degradation.

Choosing the right model is hard and we employ TPOT [48] to choose our final model. It uses genetic programming to automatically optimize machine learning pipelines. Basic configuration of this tool consists of providing train and test datasets as well as the number of generations.

This approach proves to be sufficient for our use: predicting performance degradation for known hardware and applications. While the validation scores for all nodes are satisfactory, it has certain limitations. The most obvious one is the lack of differentiation between the services and performance degradation output. This boils down to training with the same input for two different targets. One approach to overcome this issue would be to include the original workload characterization of the desired service, another one could insert the clusters obtained from 4.3.4. Further, it is questionable on how the model performs on unseen services and different inputs. Though the focus of this work is to provide our simulation a model that predicts the scenario applications and therefore investigation is out of scope.

This concludes our performance modeling. We present in the next section our approach to improve container scheduling in heterogeneous clusters using the aforementioned solutions.

4.4 Container scheduling

This section is structured as follows: we explain our extended version of OpenFaaS, the optimization problem we solve and finally our priority functions.

4.4.1 OpenFaaSExt

As already mentioned to realize our vision of dynamically deciding between different accelerator versions of a function, we need to adjust the OpenFaaS architecture to our needs. Per default it is only possible for users to submit a single container image. In our opinion users should provide one or multiple images to accommodate different versions of a function. OpenFaaS lets users submit functions via uploading a *FunctionDefinition* containing the image and other details (see Table 2.1). Our approach is to introduce the concept of a *FunctionDeployment*, which contains all attributes of a *FunctionDefinition* but allows developers to add multiple *FunctionDefinitions*. In essence, we strip the original *FunctionDefinition* from all properties concerning scaling and lift them up into a new type, which basically acts as a container. This introduction enables developers to freely upload many *FunctionDefinitions* for one function and lets the system decide which computing platform to use. The caveat of this approach revolves around the decision of which *FunctionDefinition* the system should deploy. We think that a static ranking is sufficient for our first implementation, though it is plausible to let users dynamically change this ranking - paving the way for a new optimization problem. That means users specify an order of computing platforms. After using all nodes that offer a specific platform, the next one is used. Users can configure on how many nodes per computing platform the function gets deployed. This is done to enable fine-tuning of function deployments and gives users more control over their applications and deployments. The main reason behind this functionality is based on the fact that some computing platforms may be very expensive and users can save money by reducing the number of instances of a certain platform. It should be noted here that we implement this proposed system only in the context of *faas-sim* and do not provide a production-ready application.

This concept introduces first-class-citizen support for different computing platforms and is in our opinion a key requirement to realize the vision of serverless edge computing. Another problem that arises in this context is the high degree of hardware heterogeneity. The next chapter is dedicated to propose a problem formulation that aims to help schedulers find suitable nodes in these kinds of clusters.

4.4.2 Capability Matching Problem

We start with a brief review of the capability matching problem, introduce it formally and show an optimization approach.

Edge computing is characterized by an abundance of different device types, each having their own capabilities. Especially hardware accelerators that speed up AI applications by a significant factor are of interest. Due to the limitation of static ranking, to support multiple computing platforms, we do not influence the choice of accelerator. Choosing a computing platform is therefore out of scope and our focus lies on considering other attributes, such as CPU cores, available memory, network, location and hard drives. The list of capabilities is represented by attributes introduced in Section 4.2. In combination with the serverless deployment model, where placements happen fully automated, a suitable strategy, making informed decisions when placing nodes, is required. The goal is to develop a technique that can match node capabilities with application requirements.

We propose a solution by formulating an optimization problem, which outputs for each application group a requirement setting. These settings are used by our priority function that scores each node-application pair based on their compatibility.

Given a set of functions F, function groups G (determined in Section 4.3.4), a mapping m that assigns each function f to a group g and nodes N, which have a description D.

The input consists of a group g, its associated functions fn and a selection of nodes n. We look for the optimal solution, represented by requirements R, that contains for each possible attribute and value the likelihood of a node having this pair. A possible solution for this problem can be seen later on in Listing 4.1, but was previously shown in a different form, the cluster configuration. Before solving the problem we need to introduce a definition of optimal. In our case we focus on keeping a balance between heterogeneity and performance. Further, to integrate this approach into OpenFaaS and make our priority function aware of this, each deployed application its group's solution assigned.

The problem is similar to 0/1 Knapsack, which tries to put as many weighted objects without exceeding the maximum capacity. In our case items are represented by nodes and their weight is implicitly described their capabilities and performance. Further, our approach does not have a maximum capacity —it is valid to have a solution containing all nodes, and we define it as a minimization problem. This approach allows extensions to include node specific information. For example an objective function can consider the resource contention of specific device types, which may lower the score. It should be noted here that we do not implement such heuristics. To evaluate a solution we define the following heuristic function, which takes a list of selected devices:

$$\begin{gathered}
\text{performance} = \text{mean FET} * \text{performance weight} \\
\text{device ratio} = \text{number of devices in solution/total devices} \\
\text{variety} = ((\text{heterogeneity of selected devices}/16) + \text{device\_ratio}) * \text{variety\_weight} \\
\text{score} = -(\text{performance} + \text{variety})
\end{gathered}$$

The mean FET is for all functions between one and zero, one being fastest. The value is calculated by taking the average over all devices and their mean FET. To prevent any issues that would prefer functions with generally higher response times, we scale each FET to the function's minimum and maximum. This leads to a more abstract mean FET, but weights each function equally. We divide the entropy by 16, the highest possible heterogeneity score, to normalize it. The resulting requirement r is calculated based on the selected devices, i.e., the probability for each attribute and value.

We introduce an example to explain our approach. The cluster is represented by the configuration **B**, presented in Table 4.3. We keep the distribution but assume only ten devices in total. Our application group consists of two functions **A** and **B**. The FET for each device is displayed in Table 4.9. The problem's input, an array of length ten, is displayed as solution in Figure 4.5 (*1:1 representation*). To calculate the score we need to scale the FETs for each device, which is also shown in the table. For simplicity all weights are *1* and the resulting *performance* is *0.675*. Because all devices were selected the *device ratio* is *1* and *heterogeneity* has previously been calculated at *2.60*. The final score is *-1.84* and its solution, the requirements, are the same as shown in 4.3. Extensions or modifications are plausible. It is possible to remove the 1:1 representation and just

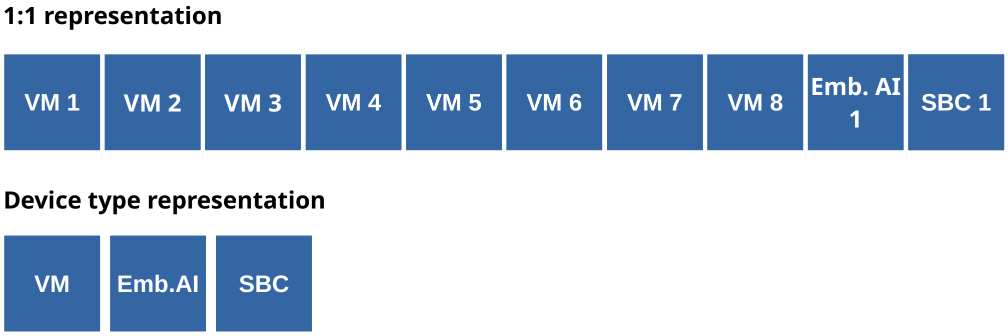


Figure 4.5: Example input of our capability matching problem, inspired by Knapsack 0/1

Device	Function	FET	scaled FET
VM	A	1	1
Emb. AI	A	2	0.5
SBC	A	3	0
VM	B	2	0.5
Emb. AI	B	1	1
SBC	B	3	0

Table 4.9: FET for example application

Setting	Value
Max num. iterations	3000
Population sizes	100
Mutation probability	0.01
Elit ratio	0.01
Crossover probability	0.5
Parents portion	0.3
Crossover type	uniform
Mutation type	uniform by center
Selection type	roulette

Table 4.10: *geneticalgorithm2* settings

take the different types of devices as input, depicted in Figure 4.5. An extension of the fitness function could consider mutable node states, i.e., resource consumption. This would entail a repeated execution of the optimization strategy.

We implement two approaches for our evaluation, which differ in input representation. The first one considers all individual devices, while the second one only uses the device types. In our case the input is in the latter case very small and we therefore solve it by enumeration. Further, using the device type representation, we omit the *device ratio* and only use the heterogeneity score.

As it is trivial to see that this problem is NP-hard, we have decided to use a Genetic Algorithm approach to tackle it. Our implementation uses the open source library **geneticalgorithm2** [5] and apply its default parameters regarding mutations, selections and crossovers. The detailed configuration is displayed in Table 4.10. Further, we only take a look at the parameters of our fitness function and leave the genetic algorithm ones untouched.

After explaining the components relevant to the scheduling optimization steps we have taken, we turn to the scheduler extensions which use: baseline profiling, workload

[5] https://github.com/PasaOpasen/geneticalgorithm2

characterization and capability matching solutions (*requirements*).

4.4.3 Scheduler extensions

As described in Section 2.2.2, the Kubernetes scheduler allows developers to inject custom predicate and priority functions. We develop several of these to add new accelerator/resource-based hard constraints and priority functions that concern themselves with capability matching, resource contention and execution time. In this section we present Kubernetes related concepts and therefore use the pod terminology analogous to previous mentions of a function.

4.4.4 Predicates

Predicates, we deem necessary, filter nodes that: do not have enough memory, do not have the required accelerator, already host a TPU-accelerated pod in case a TPU is required. In case of GPU-based applications we offer two modes: the first one, representing the standard Kubernetes setting, only allows one GPU image per node, the second one retrieves the pods VRAM request and checks if enough is still available. We test in our evaluation only the default Kubernetes setting.

4.4.5 Priorities

We develop three priorities that should improve different aspects of placement decisions.

CapabilityPriority

The first one takes a pod's assigned requirements into consideration and is called *CapabilityPriority*. The calculation, accompanying optimization problem and solution were introduced in Section 4.4.2. Though it is sufficient to view the requirements as labels that give each attribute and each instance (i.e.: attribute: Architecture, instance: X86) a value representing the probability of a node having this pair assigned based on the final solution set of selected to nodes that balances the FET and variety of nodes.

Therefore, the priority function compares the current nodes capabilities with the pods requirements and sums up all probabilities. Algorithm 4.2 shows the soft constraint in pseudo code. This strategy utilizes the mapping of requirements to capabilities and favors nodes with similar characteristics. For easier understanding Listing 4.1 shows a requirement object as JSON. The example shows a strong tendency towards devices equipped with a Turing GPU. Additionally *arm32* devices are not represented in the *architecture* attribute, though they will receive an increase in case other characteristics match.

Algorithm 4.2: CapabilityPriorityFunction

```
Result: Estimation how well suited node and pod are for each other based on
        the pods requirements
Function score:
    Input : pod
    Input : node
    priority ← 0;
    for attribute, instance in node's capabilities do
        if pod.requirements[attribute] is available then
            priority += pod.requirements[attribute][instance] or 0
        end
    end
    return priority;
```

```
{
  "architecture": {
    "x86": 0.5,
    "aarch64": 0.5
  },
  "cores": {
    "MEDIUM": 1
  },
  "accelerator": {
    "None": 0.1,
    "GPU": 0.9
  },
  "gpu_model": {
    "TURING": 0.8,
    "VOLTA": 0.1,
    "MAXWELL": 0.1
  }
}
```

Listing 4.1: Resulting requirements for example

ContentionPriority

Our last priority revolves around the imminent problem of resource contention. Algorithm 4.3 shows the implementation. This function's intention is to score nodes according to their current resource usage and the additional usage from the new pod. For this we use our workload characterizations from Section 4.3.3. In contrast to the clustering step we do not have to aggregate a single vector for each device but can use the original raw data. Another difference is that we use in this step the data rates and discard the total related block and net I/O attributes. Because we know the type of storage and network connection, we can make crude estimates towards the impact of I/O work.

In essence, we sum up the utilization (CPU, GPU, net & block I/O) for each running container and subtract it from the pods resource usages. The function favors nodes with lower resource utilization.

Algorithm 4.3: ContentionPriorityFunction

```
Result: Score nodes according to their current resource usage
Function score:
    Input : pod
    Input : node
    pod_blkio ← pod.get_resource_usages("blkio")
    pod_net ← pod.get_resource_usages("net")
    pod_cpu ← pod.get_resource_usages("cpu")
    pod_gpu ← pod.get_resource_usages("gpu")
    running_blkio ← 0
    running_net ← 0
    running_cpu ← 0
    running_gpu ← 0
    for running_pod in node.pods do
        running_blkio += running_pod.get_resource_usages("blkio")
        running_net += running_pod.get_resource_usages("net")
        running_cpu += running_pod.get_resource_usages("cpu")
        running_gpu += running_pod.get_resource_usages("gpu")
    end
    if running_net > 0 then
        running_net /= node.net_speed
        running_blkio /= node.disk_speed
    end
    pod_net /= node.net_speed
    pod_disk /= node.disk_speed
    pod_usage ← pod_blkio + pod_net + pod_cpu + pod_gpu
    running_usage ← running_blkio + running_net + running_cpu +
      running_gpu
    return pod_usage − running_usage
```

As previously mentioned the data rates represent the bytes a function call needs per second. This allows us to calculate how much time it takes in the best case. Network speed is set in the simulation and disk speed estimates are listed in Table 4.11.

Disk	Speed
NVME	2.5 GB/s
SSD	500 MB/s
HDD	250 MB/s
eMMC	150 MB/s
SD	50 MB/s

Table 4.11: Disk type speed estimations

ExecutionTimePriority

The last priority uses baseline performance of each application and node. In essence, this constraint favors faster nodes and the implementation is fairly simple, see Algorithm 4.4. One thing to note is that, due to favoring nodes with higher scores we negate the FETs. This trick assigns nodes with lower FETs a higher score.

Algorithm 4.4: ExecutionTimePriorityFunction

```
Result: Score nodes according to their FET of the given pod
Function score:
    Input : pod
    Input : node
    fet ← pod.get_mean_fet[node]
    return −fet;
```

CHAPTER 5

Evaluation methodology

This chapter presents our systematic approach to evaluate our findings and artifacts. In Section 5.1 we recap our approach shortly, highlighting important aspects and give an overview of our evaluation. Afterwards, Section **??** describes the simulation settings in detail. The overview includes only for our simulation a detailed description. The other parts: baseline profiling, workload characterization model & clustering, performance degradation and the Capability problem optimization are described in their respective section in Chapter 6.

5.1 Overview

The main goal of this thesis is to evaluate our improvements regarding scheduling of containers in heterogeneous clusters. These are a collection of priority functions, based on workload characterization and Capability matching problem, introduced in 4.4.2. The foundation of our soft constraints and optimization strategy is an approach to characterize and cluster workloads based on resources. Besides that, we introduce and implement an approach for predicting performance degradation into our simulator of choice, *faas-sim*. Detailed settings of our baseline and performance degradation benchmarks are presented in Section 6.1 & 6.3 respectively.

Therefore, the evaluation is split up into multiple parts, starting with the base for all subsequent solutions - the *workload characterization* and clustering of workloads. The former presents the final resource vectors. The evaluation of our approach to cluster functions, based on resource usage, consists of selecting two clustering configurations. We choose a clustering configuration based on intuition and additionally use silhouette, a common cluster validation strategy [59], and select the highest scoring one.

Next, we optimize our optimization strategy parameters on one of our three topologies. We examine runtime characteristics and the objective function's output. Further, we examine the resulting heterogeneity and the impact of our objective function's hyperparameters. As mentioned before, we take two cluster configurations into consideration. For this we optimize and analyze them both separately with regards to the parameters and conduct later on a pre-defined subset of experiments to decide which one is more suitable for our applications and devices.

Due to the large impact on simulation results we validate our approach to predict performance degradation. Because we use TPOT [48], we limit this section to test scores for all our devices.

After validating all of our approaches that build the foundation for our scheduling improvements, we prepare one use case scenario and generate three different heterogeneous clusters. We introduce our experiment setup, which includes scenario, functions, devices and infrastructures. Besides the comparison with default scheduler settings, we include simulations that use the priority functions presented by Rausch et al. [58]. As mentioned in Chapter 3, the foundation of our work is build on [58]. While our priority functions concentrate on performance and resource contention, theirs are mainly concerned with data locality and proximity. In addition, we combine all mentioned priority functions.

For now, we focus on the actual metrics, how we measure success or failure, and highlight key characteristics that build the center of our attention.

Metrics we consider to be of importance are: FET, processed invocations and performance degradation. We show aggregated results over all functions for our scenarios and select two experiments, for which we present more detailed results. As we emphasize the need for considering all devices and to make intelligent decisions when scheduling on Edge nodes, we take a close look into placements. Based on the workload characterization we can estimate the resource usage on each node and present an aggregated look at the general resource usage but also the experienced performance degradation.

This concludes our overview of the evaluation - we continue by explaining our simulation settings in detail, starting with our scenario.

5.2 Scenario

Before explaining our scenario in detail, we motivate our choice by explaining the general context of the applications, what differentiates them to the typical cloud-centered services and new challenges edge computing introduces.

The focus lies on data-intensive serverless edge computing applications [58], specifically on AI Pipelines that utilize all devices connected to an edge network. Further, our pipelines use different deep learning models that also show different resource consumption characteristics. The applications range from object detection models, such as Resnet50 [26], and MobileNets [27], to Speech inference (Deepspeech [23]). For each app, it is plausible that they fit into the context of Edge Intelligence: generating data is done by sensors, i.e. traffic cameras or personal assistants. Preprocessing can be done at the edge, i.e. resizing of images. Their use cases may be highly contextual. For speech inference and object detection the place of use can dictate the importance of words and objects. Taking the example from section 2.1 with the flower and hardware store. In each of both spoken language and objects will stem from different contexts.

In summary we can identify characteristics that differentiates them from typical cloud-centered applications:

1. Heterogeneous landscape: Edge computing in general is considered to be very heterogeneous in hardware, as many small compute units are deployed. The devices can range from IoT sensors, Single Board Computers (SBC), AI hardware for edge systems, cloudlets to cloud servers.

2. Resource contention: shared cloud servers can experience resource contention through co-located containers and virtual machines [37, 22, 40]. Our experiments have shown that performance degradation caused by resource contention is much higher on resource-constrained devices.

3. Accelerators at the edge: AI can benefit by using the right hardware [15]. Cloud operators like Google offer different services, exposing GPUs or TPUs for specific workloads. These types of hardware are also represented at the Edge - the Coral Dev Board provides users a TPU, as USB connected accelerator. Nvidia's offering comprises different devices equipped with AI-focused hardware. Due to their small scale factor they lack performance in comparison to cloud GPUs, but research has shown that locality of compute nodes are critical in data-intensive edge applications [58] and therefore it is important to not neglect them.

4. Tight latency requirements: Virtual Reality and Cognitive Assistance are on the verge of becoming reality, posing strict latency requirements to guarantee flawless execution, which can be only fulfilled by real time processing.

Edge AI applications combine the aforementioned characteristics. Ranging from use of audio and video data, like video analytics [12, 66] or personal assistant services (Amazon Alexa). To large scale operations as seen in Smart City development [53]. In Urban sensing scenarios, nodes consisting of multiple IoT devices are deployed throughout the city, capable of capturing and disseminating various information of their environment.

We use the Urban sensing environment as our motivating scenario and base the resulting cluster from a real world deployment: the Array of Things (AoT), based in Chicago [9]. Catlett et al. deploy Array of Things nodes (AoT), that measure with the help of multiple sensors and host computing capability to run services in-suit. Before explaining the distribution of nodes in our scenario, we present a short overview of our devices. A more detailed description of the devices follows in Section 5.3.1.

The nodes we simulate in our experiments stem from our testbed, containing nine different devices. Table 5.1 shows the specification of each device type. There are four different types that all serve a purpose. Single Board Computers (SBC) are the weakest with regard to computing power have no accelerator. We have two different types of hardware accelerators: GPU and TPU, latter can only be used for inference and GPUs can additionally train a model. The Nvidia Jetson series and the Coral Devboard belong to this type. Cloudlets [62] offer more powerful nodes that can help withstanding peak workloads. VM instances are located in the cloud and have the highest performance.

The aforementioned AoT nodes are distributed over neighborhood areas and have one to three Single Board Computers (SBC) equipped. To support these devices, we assume various cloudlets deployed throughout the city, comprised of Intel NUCs and embedded GPU devices and Edge TPU accelerators. The cells have a shared bandwidth of 10 Gbit/s and are connected to the cloud via a mobile connection that has an up- and download-bandwidth of 250 Mbit/s. Each cell consists of one or up to three SBCs, multiple embedded AI devices and one cloudlet. Because our approach to generate the devices, based on a specification in advance, it is possible that some cells differ in diversity, it may be possible that some do not contain any SBC, AI device or cloudlet. Though this is not of concern for this thesis as we focus on placement decisions and the resulting consequences on performance and resource contention. Besides edge deployments, we make two types of VM instances available which are located in the cloud and differ only in the availability of a GPU. These should help withstand peak workloads and are most appropriate for handling the training of large models.

For our evaluation, it is important to generate multiples settings with a varying number of nodes. Rausch et al. made a tool publicly available that synthesizes distributed systems with a focus on Edge scenarios [56]. Their tool, Ether, is capable of generating topologies, including the possibility to describe different kind of nodes, give them labels, but also has concepts of data stores and has an integrated network simulation. This allows us to generate realistic networks, which in turn can be used to simulate plausible scenarios. This approach lets us evaluate our solution in a meaningful way by considering different capabilities of the network and each device.

By taking the general structure of the Urban sensing scenario as the basis and using Ether, we can create various topologies which range in their node type distribution.

5.3 Experiment Setup

After explaining the general context for applications and device topologies we present detailed explanations. We start by showing the specifications of our testbed, synthesized evaluation scenarios and finish by describing our functions and their associated workload patterns.

Table 5.1: Device type specifications

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

5.3.1 Devices

We have setup a testbed that consists of nodes that offer different capabilities. A detailed description is listed in Table 5.1

The VM instances in our scenarios will be represented by *XeonCpu* and *XeonGpu*. The cloudlets will consist of Intel NUC instances, offering high-performance block I/O due to the NVME. There are three different types of Single Board Computers: Raspberry Pi 3/4 and the RockPi Model 4. To represent the modern TPU technology, we use a Coral DevBoard which is equipped with a Edge TPU - note that this TPU architecture differs significantly from those that are deployed in Cloud settings, i.e. models have to prepared in advance for Edge TPU and Cloud TPU models can not run on this device. Neither is it possible to train on this node, Edge TPUs are specifically made for high speed inference. GPU embedded devices are represented in three different forms and are all build by Nvidia and belong to their embedded AI product line *Jetson*: Nano, TX2 and Xavier NX.

5.3.2 Synthesized Infrastructure

To evaluate our approach we synthesize plausible infrastructures that resemble the Urban sensing setting. Our scenario resembles a Smart City with cells distributed through the town, bandwidth and structure of those have been explained in Section 5.2. The following shows device proportions that are then connected according to our description for purposes of our simulation. We generate different topologies that differ in the distribution of each node type and proportions, including the heterogeneity score, are shown in Table 5.2.

The first scenario, *cloud*, is characterized by having a high number fo *XeonCpu* instances. This scenario presents the beginning of adopting Edge Computing. Few nodes are dispersed at the edge and only 8.2% of the VM instances come equipped with a GPU. The intuition and intention behind this scenario is to offer an evaluation that builds a clear contrast to the targeted environment. Our other two scenarios represent the vision of edge-centric computing. *edge cloudlet* has a high proportion of cloudlets, that do not

Scenario	Coral	Nano	Intel NUC	Xavier NX	RockPi	RPI 3	RPI 4	TX2	XeonCpu	XeonGpu	Het.
cloud	1.0	4.4	2.8	0.8	2.0	5.4	4.6	1.8	69.0	8.2	5.95
edge cloudlet	1.8	9.8	38.8	3.8	3.2	16.0	14.0	1.4	9.6	1.6	6.92
hybrid	8.0	8.4	14.8	1.4	10.6	21.4	18.6	1.6	9.2	6.0	7.38

Table 5.2: Device proportions in percent of evaluation clusters and the resulting heterogeneity score.

Function	Functionality	Language	Watchdog	Processing Mode
resnet50-preprocessing	Scale & resize images	Python 3	HTTP	CPU
resnet50-training	Fine-tuning of Resnet50	Python 3	HTTP	CPU, GPU
resnet50-inference	Object classification	Python 3	HTTP	CPU, GPU
mobilenet-inference	Object classification	Python 3	HTTP	CPU, TPU
speech-inference	Speech-to-text transcription	Python 3	HTTP	CPU, GPU
tf-gpu	Matrix multiplication	Python 3	Forking	GPU
python-pi	Pi calculation	Python 3	Forking	CPU
fio	Random block I/O read/write	Bash	Forking	CPU

Table 5.3: Functions used for benchmarking experiments and simulations

offer any accelerators, while *hybrid* aims to offer all node types with equal chances. To recap, we consider four node types: SBC, embedded AI, cloudlet and VM instances. Further we believe that the proportions of SBC are justified as they tend to be very cheap in contrast to other hardware and therefore should be represented that way. While embedded AI hardware at the moment is much more expensive, though offers substantial performance improvement over SBC nodes.

5.3.3 Functions

The platform of choice for deployments is OpenFaaS and therefore our functions are build accordingly. The basic steps to build a function in this framework consist of choosing a programming language, the Watchdog and providing user written code - the function. We have two sets of functions, the first one contains all and is used for workload characterization and performance degradation. The second one represents AI pipeline functions and is used in our simulations *faas-sim*. We present at first our AI functions and then turn to those that are exclusively used for workload characterization and performance degradation. Besides a description of functionality, we add further information about programming language, container image size, additional network traffic and the selected Watchdog, shown in Table 5.3.

AI pipelines consist in general of different steps, from retrieving and preprocessing the data, to training and finally inference. In the context of Serverless Computing the approach to share data between different functions is to use storage accessible via network. The problem of possible bandwidth congestion and high latencies, caused by slow networks, is posed by the foundation of serverless: stateless functions with automatic provision. Additionally it is possible for users to submit payload via HTTP but this use case may be limited by the fact that binary streaming is not available in OpenFaaS. All AI functions are implemented in Python using Tensorflow. We implement for this thesis one complete pipeline and add two additional inference applications. The Resnet50 model [26] is used to implement a complete pipeline. This model is designed to classify images, i.e.: it may detect the name of a flower. Therefore, the preprocessing step consists of scaling and resizing images to the dimensions 224x224. The basic steps of this function include: downloading, preprocessing and uploading images. Network traffic consists of down- and uploading 28MB total. As can be seen in Table 5.1, some nodes use SD Cards which drastically reduce block I/O performance and therefore execution times are very high. The training step downloads a model and data to train it and finishes with uploading the fine-tuned model, causing a total of 150MB in network traffic. The last step, inference, downloads upon the first request a 100MB large pre-trained model and caches it. User requests include a picture and the result is returned as HTTP response. Training and inference can use CPU or GPU. All functions are implemented with the HTTP mode watchdog. While training and preprocessing can be implemented using the process-per-request approach, it is infeasible for inference as caching is necessary to avoid high latencies. The container image sizes depend on the architecture but range from 700 MB (arm32) to 2 GB (x86, arm64).

MobileNets [27] enables further mobile vision applications and uses Tensorflow TFLite, the runtime for resource-constrained devices, and additionally is optimized for Edge TPU accelerators. Therefore devices need to download in total 4MB, making it very lightweight in contrast to Resnet50. This function's implementation is the same as before: downloading model upon first request, caching it for further requests, retrieving the picture from the request's body and returning results as response. The TFlite runtime reduces the container sizes to 180MB.

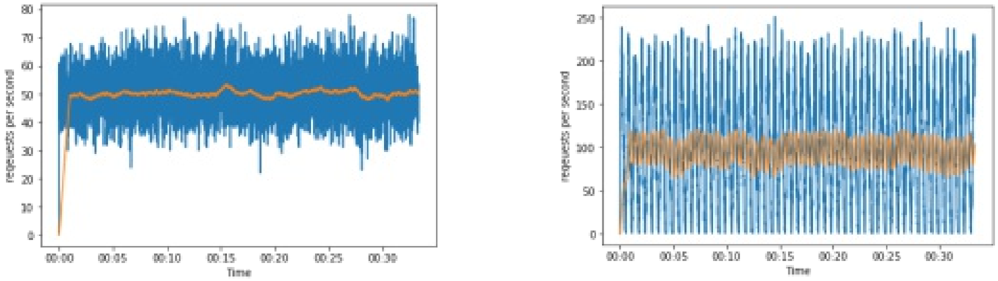
The last inference application differs substantially in terms of use cases as it offers speech-to-text inference. Users send audio files as request and receive the transcription. We use a reference implementation, DeepSpeech [23], which is either run in TFLite or GPU-accelerated mode. The difference in runtime impacts the container sizes: while TFLite images need up to 400MB, the GPU-accelerated version takes up to 1.6 GB of space.

Besides AI-based applications we implement three functions that each focus on a single resource: CPU, block I/O and GPU. These functions are solely used for workload characterization and performance degradation benchmarks. The latter focuses on causing interference when running a task, and is used to create training data for our model. The CPU workload calculates pi up to a user specified precision, block I/O performs random read in write requests, while our GPU-focused function calculates a matrix multiplication.

Table 5.3 shows a comprehensive overview of our functions.

		resnet50-preprocessing	resnet50-training	resnet50-inference	mobilenet-inference	speech-inference
Constant	max rps	1	0.1	50	50	50
Sine	max rps	10	10	100	100	100
	period	75	150	37.5	25	50

Table 5.4: Workload setting parameters



(a) Constant request pattern

(b) Sine request pattern

Figure 5.1: Workload patterns for Resnet50-Inference

5.3.4 Workloads

To simulate user requests we synthesize two different workloads that continually produce load for a fixed duration. Further, we specify vastly different number of invocations for training and preprocessing than for our inference functions. The former tend to be called once every while and inference requests may arrive with a very high frequency. As our functions can be used for image classification, it is safe to assume that they are called frequently with respect to applications such as Virtual Reality or Cognitive Assistance. Our speech inference function is called less often, but fits in the same context of contextual AI.

We synthesize a constant and sinusoidal profile for our evaluation. Therefore simulation parameters are the maximum number of requests per second (rps) and in the case of latter the period. To diversify the simulations further, we use different maximum rps and periods for our functions. Table 5.4 shows the exact settings we execute the simulations. Preprocessing and training have a low number of invocations as they are not called that often in real life and can take a while to complete. Inference requests per second were chosen due to their use cases and it is plausible that in crowded areas requests are higher. We run each experiment five times to ensure consistency in our results and each simulation simulates 33 minutes. The reasoning behind running simulations for a fixed amount of time is to guarantee a consistent workload for all evaluated approaches. Figure 5.1 displays two generated workloads, constant and sine based for Resnet50-Inference. The orange line is based on the average number of requests per minute.

workload		resnet50-training	resnet50-preprocessing	resnet50-inference	mobilenet-inference	speech-inference
constant	minimum	3	3	3	3	3
	maximum	250	125	350	125	125
sine	minimum	1	1	3	3	3
	maximum	250	125	500	500	500

Table 5.5: Scaling settings

5.3.5 Simulation

Further parameters that need to be set are concerned with auto-scaling. The aim of this thesis is not to develop a sophisticated scaling strategy but needs to behave realistic. Therefore we make use of the fact, that our functions use an internal queuing system and can serve up to four concurrent requests. This is the default setting for OpenFaaS Functions that use the HTTP watchdog, but it can be fine-tuned in our simulation. We periodically check, in a fixed time, the median queue length over all applications and calculate how many replicas are needed to get a target queue length across the functions. To prevent over-provision, as applications may need some time to setup, we take starting ones into consideration. The idea of this method is lend from Kubernetes' default auto-scaler [1] that uses resource consumption as metric. All experiments run with the same setting and we target a length of 75 calls and check in an interval of 50 seconds. Further settings are concerned with the minimum and maximum amount of applications running. We set both numbers for each function and differ between the constant and sinusoidal simulations. Table 5.5 shows theses settings for each function and simulation. Parameters were set on a series of benchmarks to avoid bandwidth congestion, that leads the simulation to come to a grinding halt. *faas-sim* internally simulates the network, and deployments, such as *resnet50-inference*, have to download a considerable amount of data. Our reasoning behind the maximum number of functions for training and preprocessing stem from the fact that training may take longer and therefore queues are bound to fill up much faster, which could occupy all GPU-equipped nodes. Minimum scaling decisions are based on the fact that the system may react too slow in the beginning and many requests would get stuck on one node, to prevent this we set the minimum to 3, which has proven effective.

Capability Matching simulations

After evaluating different weights and choosing the best settings, based on the final mean FET and heterogeneity, we run pre-defined simulations. These simulations should help us decide which settings we use for our final simulations. We use our *hybrid* scenario and run our sinusoidal workload profile. To this end, we keep the evaluation simple, concentrating on the average FET and number of finished invocations.

[1] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

CHAPTER 6

Results

This chapter presents the results of our evaluation and is split in multiple steps. First, we execute baseline benchmarks, which provide *faas-sim* with Function Execution Times to sample from. During these experiments we also monitor resource usage. This data is used to (1) simulate resource consumption on each node, (2) predict performance degradation, (3) build the basis for our ML-based workload characterization (4) and help the scheduler make informed decisions. Further, we perform various performance degradation experiments, showing how contention differs across nodes, and gather data for our model training. The trained regressor is used by the simulation to estimate performance degradation, caused by interference through co-located functions. We solve the capability matching problem using heuristic optimization techniques to determine a mapping between workloads and appropriate capabilities. To avoid running the optimization step for each function, we cluster applications based on our telemetry data. This is necessary to make our approach tractable regarding a growing number of deployed unknown functions. The previous steps help us to implement three priority functions, which make the scheduler workload-aware. We compare our solution with priorities from related research [58] and the default scheduler. Our evaluation consists of three heterogeneous clusters, which vary in terms on device types, and two workload patterns.

6.1 Baseline Profiling

Gathering real data from our function and devices is a fundamental step in our approach. This includes the FET and telemetry data. We call each function one hundred times, except for *resnet50-training* which is invoked fifty times. Some functions call initialization code upon the first request, which may result in unexpected long response times. Therefore, each function is called once before the experiment starts, to omit this setup time. Afterwards, we repeatedly call the function with an inter arrival time of one second. The FET represents time needed to execute user submitted code.

Figure 6.1 reports the aggregated mean FET over all devices. Figures 6.2a and 6.2b present an overview on performance per device. In both cases the mean FET for each device is scaled according to the minimum and maximum value for each function and summed up. The sum is divided by the number of functions in the benchmark. We include detailed results in Appendix A.

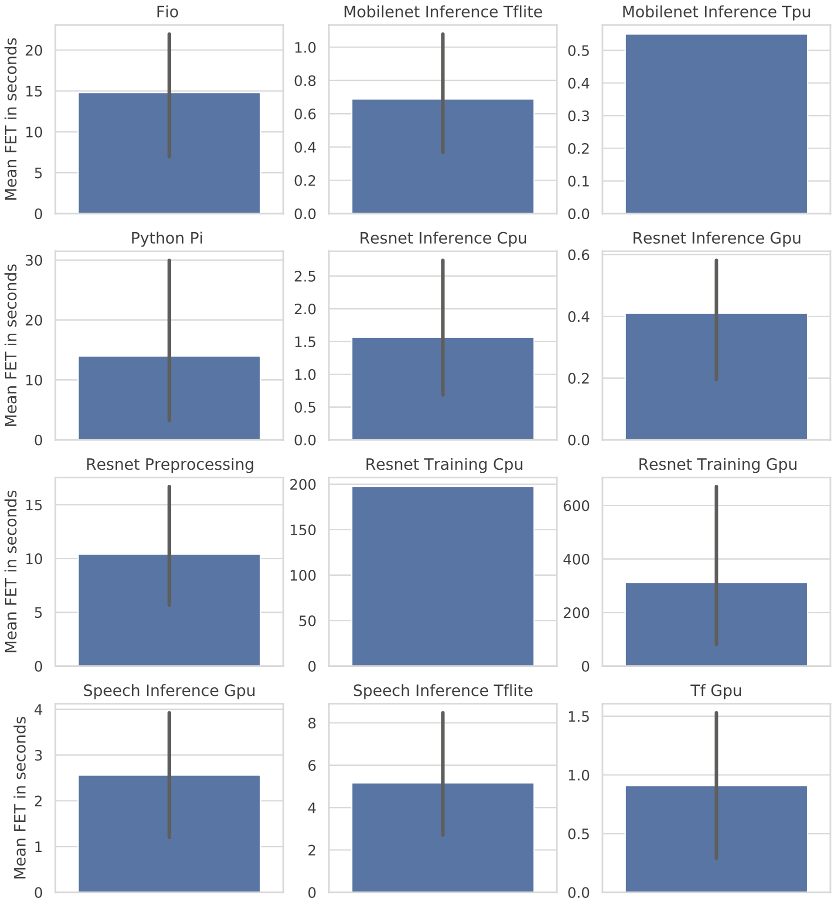
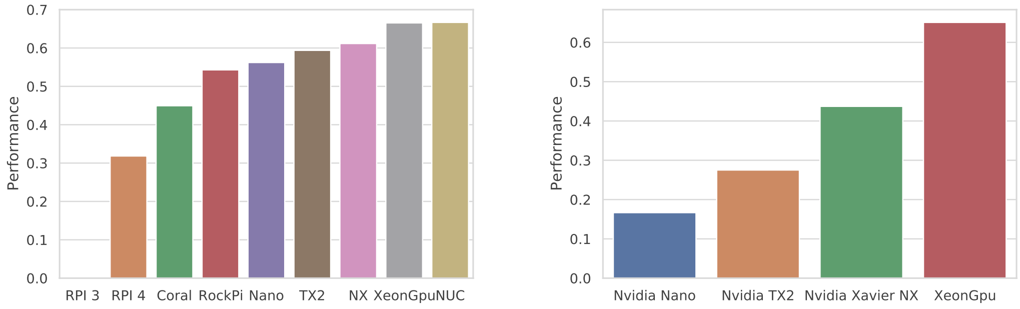


Figure 6.1: Baseline profiling results



(a) CPU Mode functions ranking

(b) GPU Mode functions ranking

Figure 6.2: Performance ranking for CPU & GPU functions

6.2 Workload Characterization

Our ML-based workload characterization uses the telemetry data from our baseline profiling step. The recorded data is a time series and therefore not suitable for our clustering technique, k-means. We reduce this data to a row vector, which represents the resource usage of a single invocation. This allows us to (1) add resource consumption to our simulations, (2) predict performance degradation, (3) cluster functions with k-means and (4) implement priority functions.

A short excerpt of the collected telemetry can be seen in Section 4.3.2 and we show in the following the final resource vectors.

Resources, related to data, i.e. *BLKIO* and *NET*, show the data rates (MB/s) while *BLKIO_TOTAL* and *NET_TOTAL* are the total amount of MB read or written during one call.

Figure 6.3 presents the resource vectors as boxplots including all devices. *Mobilenet Inference TPU* & *Resnet Training CPU* show only a single value, because they were executed on one device, *Coral DevBoard* and *Intel NUC* respectively. Further I/O related columns have been normalized to the range of [0,1], whereas 1 represents largest value encountered and 0 for the smallest one.

We include all of our results in the Appendix B.

6.3 Performance degradation model

This section presents results of our performance degradation experiments and shows test scores of our prediction model.

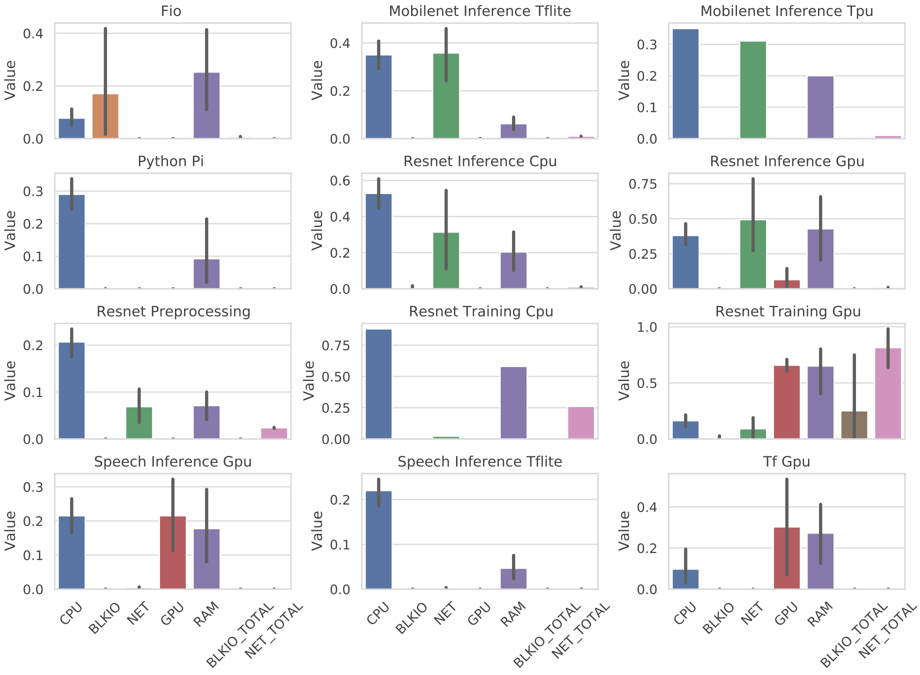


Figure 6.3: Workload characterization aggregated over devices

6.3.1 Performance degradation experiment results

Besides baseline experiments, we execute different workloads to record performance degradation. These experiments produce data which we use to train our model. We execute them on each device with individualized sets of functions. The reason for this is that resource-constrained devices are not capable of hosting the same set of parallel functions as a full-fledged server. During our experiments, devices with very low RAM capacity have proven to be very unstable in multi-tenant situations. This resulted in unexpected function terminations, leading us to determine a suitable set of functions for each device. Most notable was the Coral DevBoard, which was not able to reliable host multiple services. Therefore, we decided to omit this device from our experiments and use in our simulations the trained model of the Raspberry Pi 4 instead.

All benchmarks start one *resnet50-inference* server and receives requests from one or more clients. One or multiple instances of interfering functions are started, which are invoked by possible multiple clients. We start in each experiment only one type of interfering function. These range from I/O (*Fio*), CPU (*Python-Pi*) to GPU-oriented (*TF-Gpu*) functions.

After conducting experiments we calculate the actual performance degradation for each deployed function. This is done by comparing the measured execution time with the baseline performance and calculating the increase of time as factor. Figure 6.4 shows boxplots which display the performance degradation for each device over all experiments. Further, because we additionally calculate the performance degradation of interfering functions (i.e., *Fio*), the boxplots contain the degradation of all functions.

Detailed data on resource contention and performance interference experiments can be found in Appendix C

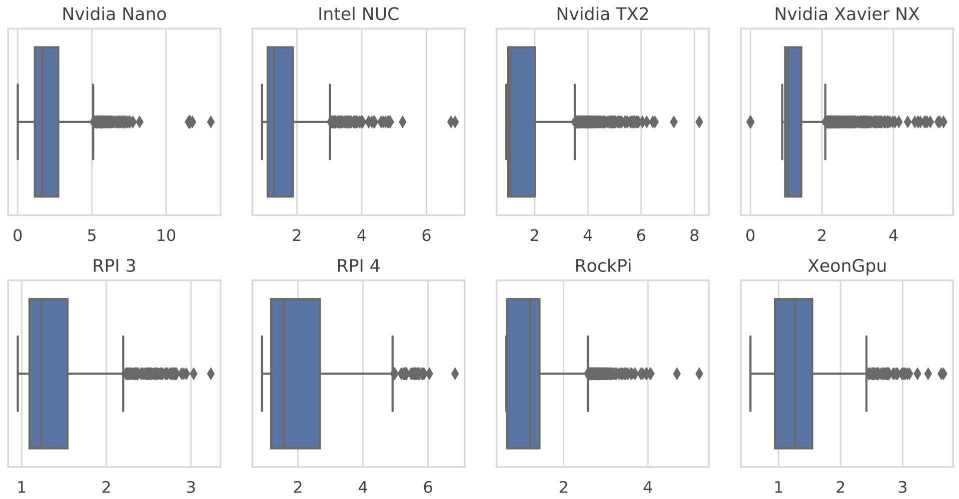


Figure 6.4: Distribution of performance degradation for all devices. Experiment configurations are shown in Tables C.1 and C.2

6.3.2 Training results

The model predicts the performance degradation based on resource usage. The input consists of various description measures calculated for each resource. We use our predetermined resource vectors (see Section 6.2) to prepare our datasets. We resort in preparing the training data to our baseline profiling data and do not use the measured resources during our performance degradation experiments. We justify this approach by highlighting the fact, that our simulation is not able to simulate real resource consumption, and uses our resource vectors to estimate it. Therefore, approach and data used to calculate the input for our prediction model is the same during training and actual usage (in the simulation).

To aid us in finding the right model we use TPOT [48]. As described in Section 4.3.5, TPOT is a library that intelligently finds the best prediction pipeline which is set to evaluate one hundred generations with each having a population size of one thousand, which resulted in 101.000 pipelines and ran for 340 hours. The scoring function has been set to use the negative root mean squared error (RMSE).

Device	RMSE	MAE
Nvidia TX2	0.283963	0.091257
XeonGpu	0.080036	0.035147
RPI 4	0.100866	0.046196
Intel NUC	0.192262	0.059633
Nvidia Nano	0.276058	0.099172
RockPi	0.164287	0.042085
Nvidia Xavier NX	0.173696	0.054399
RPI 3	0.085547	0.025420

Table 6.1: Validation scores for each device

The final pipeline is:

```
ExtraTreesRegressor(CombineDFs(DecisionTreeRegressor(Binarizer(AdaBoostRegressor(
input_matrix, learning_rate=0.001, loss=square, n_estimators=100), threshold=1.0),
    max_depth=10, min_samples_leaf=17, min_samples_split=9), PCA(input_matrix,
    iterated_power=4, svd_solver=randomized)), bootstrap=False,
    max_features=0.8500000000000001, min_samples_leaf=1, min_samples_split=4,
    n_estimators=100)
```

The train and test datasets contained experiments conducted on the *XeonGpu* dataset from our degradation experiments. Besides the performance degradation experiment results, we include all other baseline profiling data too.

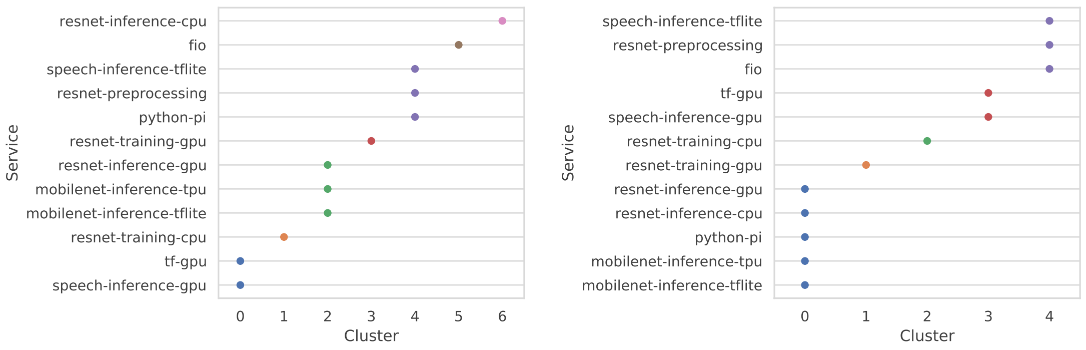
The train-test split is 0.75-0.25 and Table 6.1 shows the test score for each device. We include the mean absolute error (MAE) to interpret the results more easily. The score represents the average error over all predictions and therefore can easily interpreted by looking at the range of values from our dataset (see Figure 6.4). For completeness we include the RMSE scores.

6.4 Workload Clustering

We now present the results of our workload clustering. The goal is to find clusters of similar workloads based on resource consumption. To this end, we use k-means to find a fixed number of clusters.

As described in Section 4.3.3, we use for each function our calculated resource vector. We discard data rates and the average RAM consumption. Further, it is necessary to scale *BLKIO_TOTAL* and *NET_TOTAL* between 0 and 1, as k-means is biased towards larger values. This stems from the euclidean distance function, which is used to calculate the distance between two instances.

We determine two different cluster sizes. The first solution is evaluated manually and



(a) Best subjective configuration

(b) Silhouette score selection

Figure 6.5: Selected cluster configurations

# of Clusters	Silhouette Average
4	0.354243
5	0.386039
6	0.371803
7	0.332041
8	0.262011
9	0.230591
10	0.264676

Table 6.2: Silhouette averages for different number of clusters

based on intuition, i.e. the clustering seems promising. To offer an objective evaluation we use the silhouette score, for which we show various results and choose the highest scoring k. The silhouette score is calculated for each instance and is a measure of similarity to its assigned cluster. It ranges from -1 to 1, whereas higher values indicate a better fitting clustering configuration. By scoring each instance of one cluster, we can take the average to judge the quality of the cluster configuration [59].

In all cases our k ranges between 4 and 10. The cluster configuration, based on subjective intuition, is visible in Figure 6.5a. The silhouette averages are presented in Table 6.2 and the best cluster configuration, according to the silhouette score, is shown in Figure 6.5b.

6.5 Capability Matching Problem Optimization

In this section we show results of solving the capability matching problem, introduced in Section 4.4.2. The goal is to map functions to favorable device capabilities. Our optimization problem formulation has two parameters: a group of functions and a list of devices. Th fitness function of our optimization is based on performance and the set's heterogeneity. Performance is calculated by using results of our baseline profiling experiments. We retrieve each function's mean FET for each device in the selected set. We calculate the average FET across all devices and functions. Before calculating the average, we normalize the FET per function between 0 and 1. This prevents functions with larger values (i,.e. inference vs. training) to dominate the solutions. Solutions (requirements) contain a list of device capabilities, each having the frequency of occurrence assigned. This leads to higher values for capabilities that appear more often in the selected set of devices. The solutions aid our CapabilityMatchingPriority, and enables the scheduler to favor nodes with favorable capabilities. The priority function matches the requirements of the function with a node's capabilities. Which scores nodes higher, that have capabilities that occurred frequently in the final solution set of our optimization.

Our fitness function has two internal weights: performance and variety. While the former should promote solutions that favor the inclusion of nodes with fast FET, the latter increases favourability of having more and different devices regardless of performance.

Further, we evaluate two approaches of input modeling: ***ga*** and ***enum***. While in ***ga***, we can select each individual device, ***enum*** only considers the unique types of devices (i.e, Raspberry Pi 4). We optimize the former by using a genetic algorithm, and can solve the latter by enumeration.

Additionally, we evaluate both cluster configurations (k_5 and k_7) for workload characterization from the previous Section 6.4.

To this end, we (1) perform parameter tuning and (2) execute a small set of simulations. (1) shows the influence of parameters on the performance and heterogeneity, and (2) select approach, weight settings and cluster configuration to use for our priority function in our final simulations.

6.5.1 Parameter Tuning

We show an excerpt of our parameter search in Table 6.3 and 6.4, ***ga*** and ***enum*** respectively. Each table contains the results for both clustering approaches, k_5 and k_7. In each table the three best and worst results are presented. Our implementation performs an optimization step for each identified workload group and tables show average values over all groups. We use the *hybrid* set of devices for this evaluation.

Figures 6.6 and 6.7 delve deeper into the relationship between parameters. As before, values represent the average over all workload groups.

The plots use the variety weights as x-axis and display the resulting mean FET or heterogeneity score. The marker's color represents the associated performance weight.

6.5.2 Simulations

In this section we want to determine approach (*ga*, *enum*), weight setting (*performance* and *variety*) and cluster configuration (k_5 and k_7), to use as support for our *Capability-*

			Sum scores	Mean FET	Het. score	Function	Duration
clustering	p	v					
k_5	0.25	0.75	1.03	0.55	7.62	-1.24	850.67
		1.00	1.03	0.55	7.62	-1.61	928.38
	0.50	0.75	1.03	0.55	7.63	-1.38	825.04
		0.25	1.29	0.82	7.42	-0.69	603.52
	1.00	0.50	1.29	0.82	7.42	-1.37	587.45
	0.75	**0.25**	**1.31**	**0.91**	**6.42**	**-0.90**	**511.37**
k_7	0.25	1.00	1.03	0.55	7.57	-1.6	857.18
		0.75	1.03	0.56	7.57	-1.24	950.29
		0.50	1.03	0.56	7.60	-0.87	852.65
	0.50	0.25	1.33	0.85	7.60	-0.69	644.92
	1.00	0.25	1.34	0.95	6.31	-1.15	638.55
	0.75	**0.25**	**1.36**	**0.92**	**6.92**	**-0.92**	**553.58**

p = performance, v = variety weight

Table 6.3: Best (bottom rows) and worst (top rows) results for the GA approach.

			Sum scores	Mean FET	Het. score	Function	Duration
clustering	p	v					
k_5	0.25	1.00	1.17	0.66	8.25	-0.68	0.65
	1.00	0.25	1.18	0.99	3.16	-1.04	0.66
	0.25	0.50	1.30	0.83	7.45	-0.44	0.64
	0.75	0.75	1.32	0.91	6.52	-0.99	0.64
	1.00	0.75	1.32	0.91	6.51	-1.22	0.66
	1.00	**1.00**	**1.32**	**0.91**	**6.52**	**-1.32**	**0.64**
k_7	0.25	1.00	1.25	0.72	8.50	-0.71	1.04
	1.00	0.25	1.26	0.98	4.49	-1.05	1.00
	0.25	0.75	1.34	0.85	7.92	-0.58	1.01
	0.50	0.50	1.36	0.93	6.89	-0.68	0.97
	0.75	0.75	1.36	0.93	6.89	-1.02	0.98
	1.00	**1.00**	**1.36**	**0.93**	**6.89**	**-1.36**	**0.99**

p = performance, v = variety weight

Table 6.4: Best (bottom rows) and worst (top rows) results for *enum* approach.

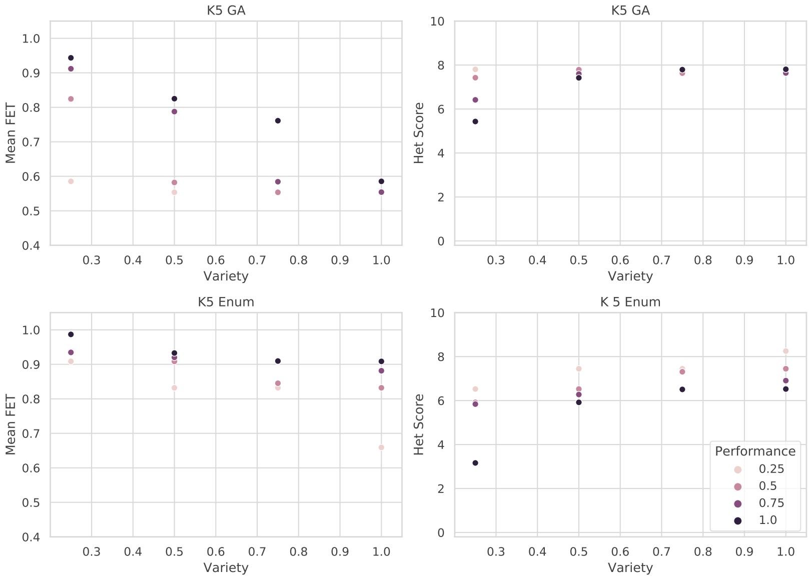


Figure 6.6: Influence of performance on variety tuning for k_5 clustering

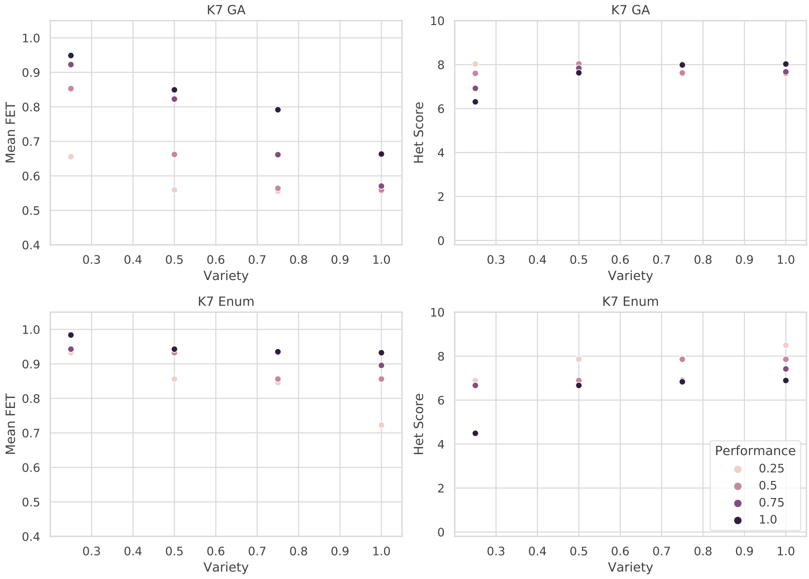


Figure 6.7: Influence of performance on variety tuning for k_7 clustering

Preference	Approach	p	v	k_n
Preference 1	*enum*	1	1	k_5
Preference 2	*enum*	1	1	k_7
Preference 3	*ga*	0.75	0.25	k_5
Preference 4	***ga***	**0.75**	**0.25**	k_7

Table 6.5: Preference definitions

				Performance Deg. $\bar{x}$	Performane Deg. σ	Invocations (scaled)
Approach	p	v	k_n			
enum	1	1	k_5	1.55	0.72	4.91
enum	1	1	k_7	1.48	0.66	4.86
ga	0.75	0.25	k_5	1.46	0.63	4.79
ga	**0.75**	**0.25**	k_7	**1.46**	**0.64**	**4.96**

p = performance, v = variety weight, k_n = clustering configuration

Table 6.6: *Preference* evaluation simulation results.

MatchingPriority.
Our formulation of the capability matching problem results in a list of weighted capabilities to favor during scheduling, called requirements. We solve the problem for each workload group and each function gets its group's solution assigned. The *Capability-MatchingPriority* reads requirements and compares it with the capabilities of the node to score. Therefore, nodes similar to the requirements receive a higher score.

We introduce the term *preference*, which consists of: an approach, weight setting and clustering configuration. There are four *preferences*, which were best in their category during parameter tuning, they are shown in Table 6.5.

We do not present detailed results of our simulations, but focus on deciding the *preference* to use in our simulations. These simulations compare our priority functions, to the scheduler with others. Therefore, we base our decision only on two aggregated values that represent the number of processed invocations and experienced degradation. In each run we simulate 15 minutes. Table 6.6 shows normalized and aggregated values, which represent the mean and standard deviation of performance degradation and the finished number of invocations. The latter is normalized to equally weigh all five functions, resulting in a maximum possible score of 5. We set the devices to *hybrid* and executed the *sine* workload.

6.6 Simulations

In the following we compare four different approaches to support scheduling decisions in Kubernetes. The approaches differ in the set of priority functions. First, we introduce notations that concern approaches, scenarios and workload patterns. Afterwards, we present the results.

6.6.1 Notation

We introduce a notation and naming scheme for each approach and use this in the following section for references.

- ***vanilla***: refers to the scheduler with default settings. Tries to spread functions across the nodes, based on the associated CPU and memory consumption.
- ***ga***: refers to the scheduler modified with our approach and uses the three priority functions presented in Section 4.4.5. They take capability matching, resource contention and performance into account - aiming to utilize edge resources with a focus on more powerful devices but takes resource contention into consideration.
- ***skippy***: refers to the scheduler modified with priority functions introduced in [58]. They focus on placing applications in near proximity to data employ a rudimentary resource contention function. The difference to ours lies in only considering CPU and memory - neglecting GPU, network and disk. Further, CPU and memory requirements are estimated and assigned by users. We use the resource vectors from our workload characterization.
- ***all***: refers to the scheduler using all previous priority functions.

Besides different approaches, there are two types of workload patterns. The settings for each pattern are shown in Table 5.4.

- ***sine***: produces workloads based on a sine wave. Each application has different settings regarding maximum requests per second and period. In this case, all functions have different settings.
- ***constant***: produces constant workload. The only parameter is the maximum requests per second (RPS). Training and Preprocessing receive much less invocations then the inference functions.

Further, there are three scenarios characterized by different device distributions, which are shown in Table 5.2.

- ***cloud***: characterized by a high number of cloud VM instances.
- ***edge cloudlet***: characterized by a high number of cloudlets (Intel NUC), has the least amount of GPU-equipped cloud VM instances.
- ***hybrid***: characterized by a rather balanced distribution over the four node types (SBC, embedded AI, cloudlet, cloud VMs).

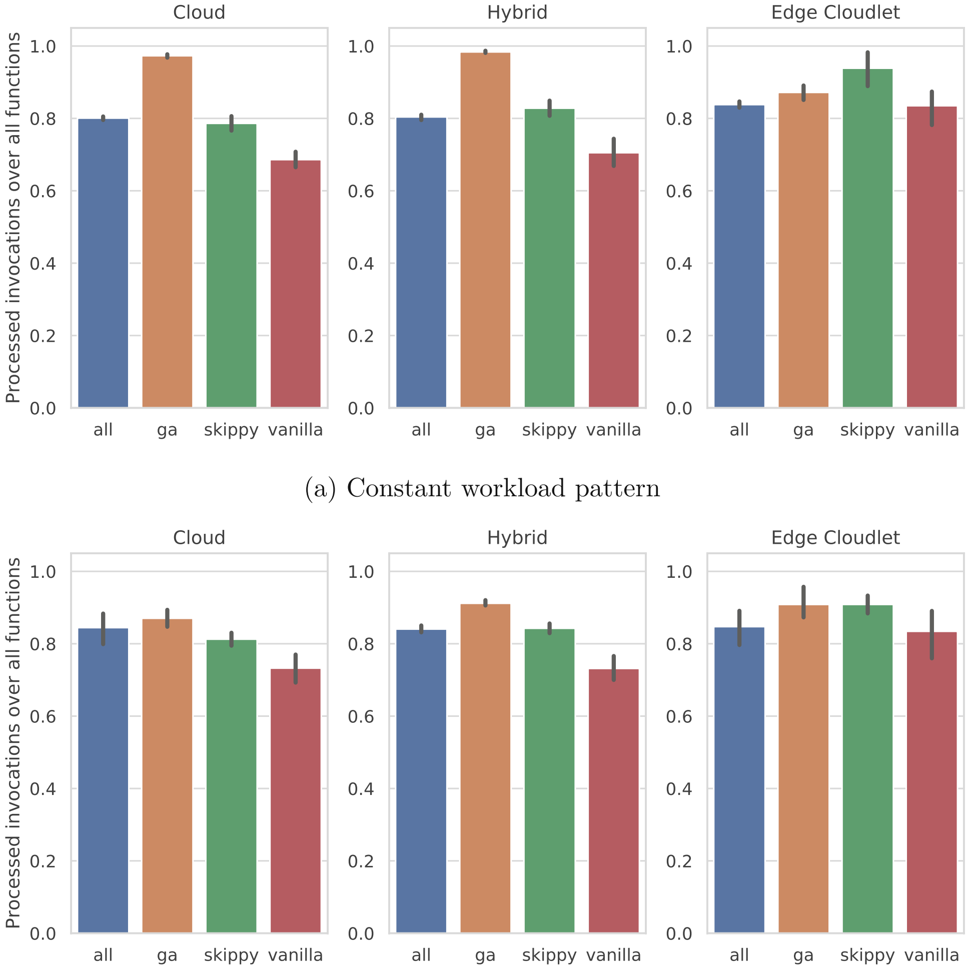
This concludes the notation and we continue by presenting our results. The results focus on three aspects: processed invocations, FET and performance degradation. For each category, we present the aggregated results over five experiments to evaluate consistency of each approach. Further, we show the performance degradation and resource utilization for one specific experiment run and select another experiment, of which we investigate the placement of containers.

A caveat of using all functions, that appear in an AI pipeline, are the different ranges of total invocations and FET. For example, preprocessing and training are called less often than inference functions (5.000 vs. 80.000). Further, FETs of both functions are much higher than for the inference ones (100 seconds vs 1 second). Therefore, we have to aggregate and normalize FET and invocations to compare them in a compact fashion. Normalization is based on min-max scaling, the minimum in all cases is 0 and maximum depends on the metric. For FET, we use the largest FET that occurred during all simulations. Invocations are scaled by taking the maximum number of invocations over all approaches.

We scale each function individually. In case of invocations, this implies that functions with low RPS are not under-represented. As the scaling happens per function, all are weighted equally and inference services, with high RPS, do not overshadow training requests. We normalize invocations for each workload separately. Because FETs are rather similar across all experiments, we choose to scale them equally, making comparison across workloads possible.

6.6.2 Invocations

This section presents the average total number of finished invocations processed during our experiments. Figure 6.8 presents the scaled and aggregated invocations for constant and sine workloads. It is important that the throughput is in all cases relatively similar, this guarantees comparability between approaches. To this end, we pre-compute the inter arrival times for each function and replay them in each simulation.



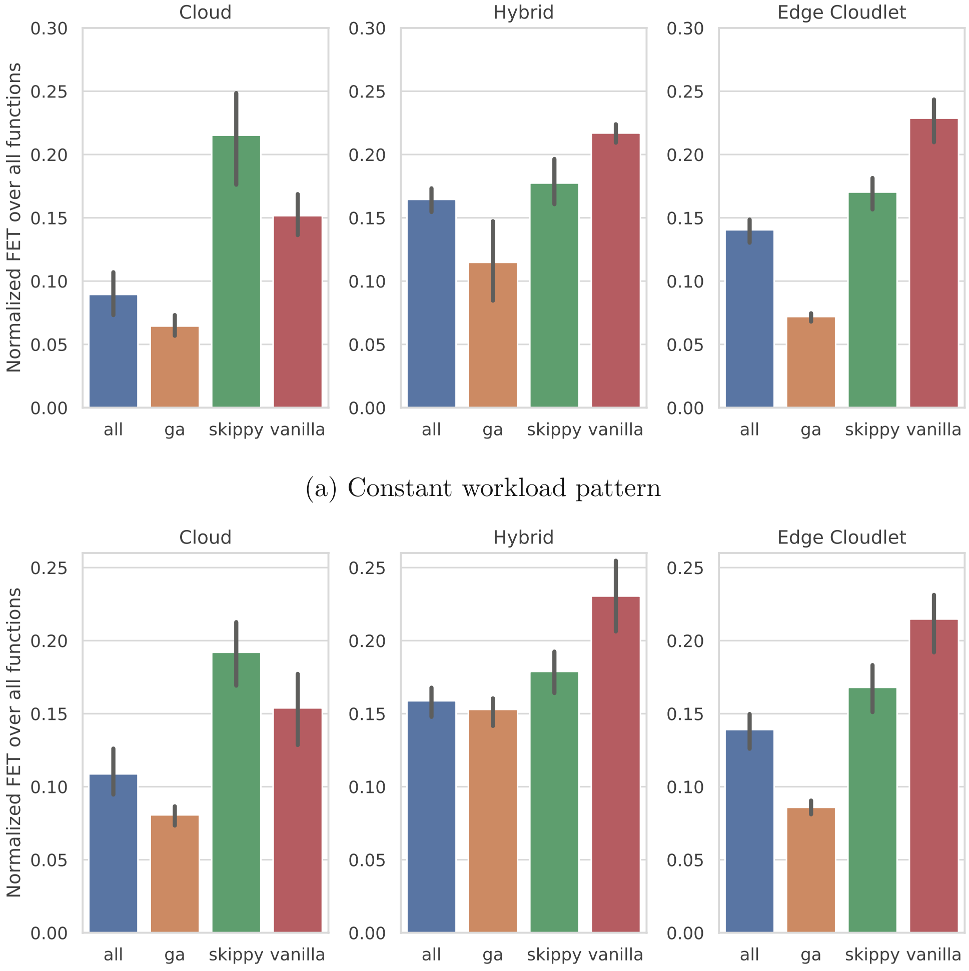
(a) Constant workload pattern

(b) Sine workload pattern

Figure 6.8: Aggregated and normalized processed invocations

6.6.3 FET

This section presents the average FET over all experiment runs in Figure 6.9. Figure 6.9a shows the results for constant workload, and Figure 6.9b for the sine workload. As described before, we apply min-max scaling for each function individually and then take the average over all functions combined. The results are normalized over each scenario and workload to guarantee comparability.



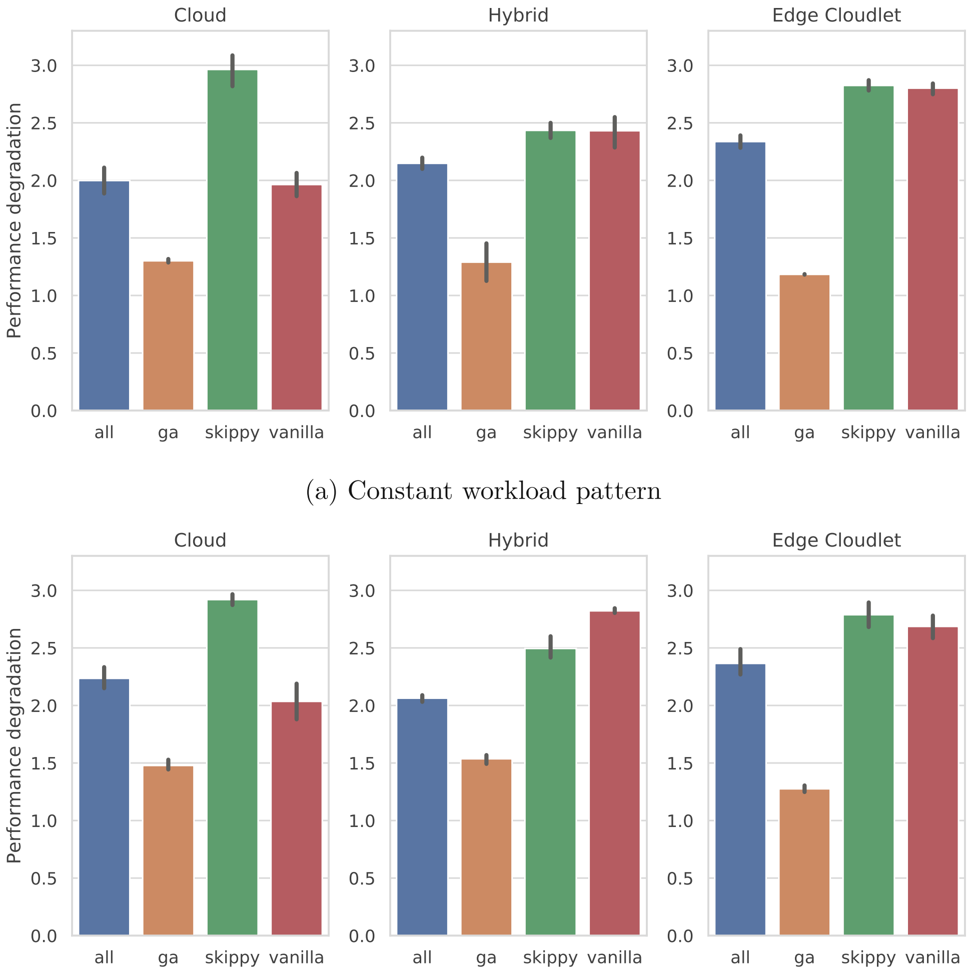
(a) Constant workload pattern

(b) Sinusoidal workload pattern

Figure 6.9: Aggregated and normalized FET results over all experiment runs

6.6.4 Performance Degradation

An important goal of our work is to identify, quantify and prevent performance degradation. Figure 6.10 shows the average performance degradation over all experiment runs, for each workload and scenario. Our trained machine learning model predicts this factor, based on the node's concurrent requests.



(a) Constant workload pattern

(b) Sinusoidal workload pattern

Figure 6.10: Aggregated degradation over all experiment runs

Individual run

Figure 6.11 shows the results of an individual run. The figures illustrate how the resource utilization and the performance degradation behave during runtime depending on which scheduler is used. In contrast to other results, the figure represents exactly one experiment run. We show the average performance degradation over all devices and additionally include the standard deviation. The total resource utilization is the sum of all resources (CPU, GPU, block/network I/O and memory). The calculation is based on our resource vectors, while this represents a single invocation, we think it is useful as an estimation. Each resource is scaled using min-max technique, where minimum and maximum are based on the smallest and largest value logged during our simulations. For each container we take the usage of a single invocation and add the values for each resource together. Further, we sum up all resources and normalize it.

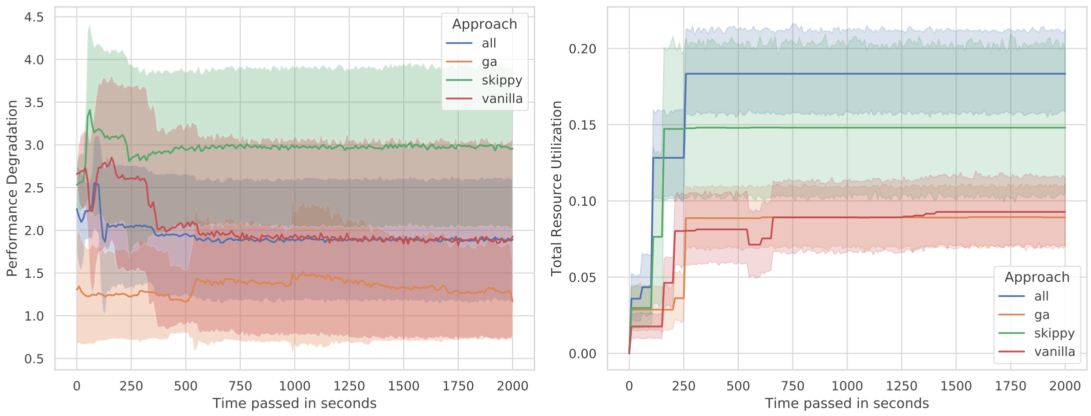


Figure 6.11: Performance degradation and resource utilization over all nodes. Taken from the *cloud* scenario during *constant* workload.

We include for this run also the number of nodes that had on average multiple containers running, the results are shown in Figure 6.12.

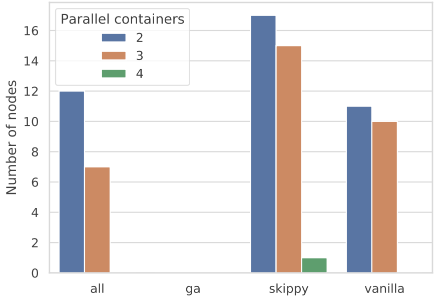


Figure 6.12: Number of nodes that had on average multiple containers running. Taken from the *cloud* scenario during *constant* workload.

Scenario	Workload	FET	Performance Degradation
cloud	constant	56%	33%
	sine	44%	26%
edge cloudlet	constant	68%	57%
	sine	59%	52%
hybrid	constant	47%	46%
	sine	33%	45%

Table 6.7: Decrease of FET and performance degradation by using *ga* instead of *vanilla*

Function	***ga***	***vanilla***
mobilenet-inference	88 307.2	87 820.4
resnet50-inference	95 701.4	92 548.4
resnet50-preprocessing	3179.2	2241.6
resnet50-training	52.0	71.0
speech-inference	98 063.4	71 868.6

Table 6.8: Average number of invocations

6.6.5 *ga* vs. *vanilla*

This section compares *ga* and *vanilla* and presents in Table 6.7 the decrease of FET and performance degradation that occurs when applying *ga* instead of *vanilla*.

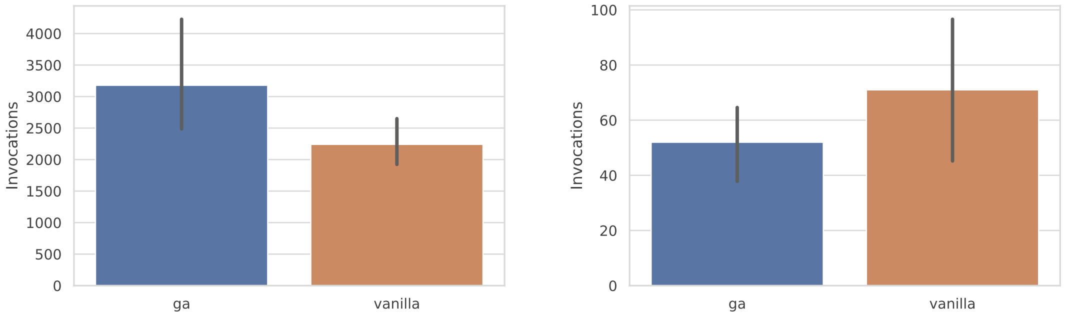
6.6.6 Edge Cloudlet

In this section we present a detail report about actual placements in the *edge cloudlet* scenario. We focus on the *ga* and *vanilla* approach, and the *sine* workload pattern. The reason for this is, that we want to show: why *vanilla* has processed fewer preprocessing invocations, and investigate the low number of processed training invocations in our approach (*ga*). In all cases, we present the average over all experiment runs.

To put things into perspective, we present the average number of invocations per functions in Table 6.8.

While *vanilla* processed less speech-inference invocations, we focus on the the preprocessing and training functions. The reason for this is that the phenomena are related to each other. Before presenting the results, we want to highlight that the deployment ranking dictates for both functions to schedule first the GPU version and afterwards the CPU one. In case of speech-inference, we schedule first the TFlite version (CPU).

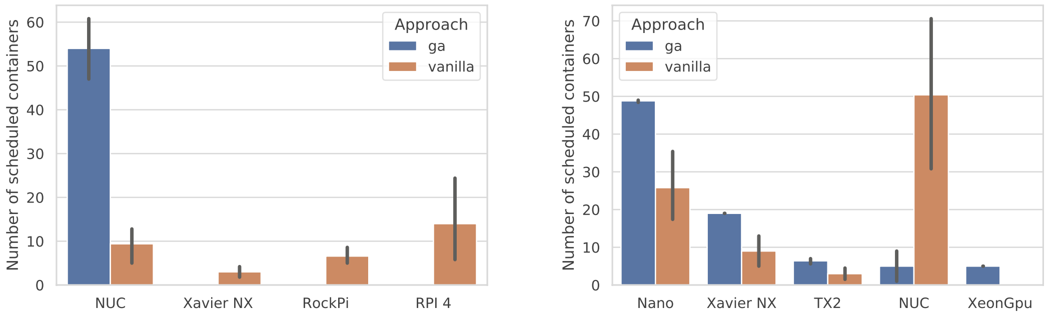
Next, we determine on which nodes the functions were scheduled. Figure 6.14 shows the number of finished deployments per node.



(a) Resnet50-preprocessing

(b) Resnet50-training

Figure 6.13: Number of of invocations



(a) Resnet50-preprocessing

(b) Resnet50-training

Figure 6.14: Resnet50-preprocessing and training containers per node type

Scheduling results lead us to believe that the GPU computing platform is exhausted in the *vanilla* approach and therefore has to use Intel NUC units. Therefore, we investigate the average number of scheduled containers per node and focus on GPU-accelerated functions. While the *speech-inference* function supports GPUs, the scheduler was able place the TFlite function in all cases, and therefore deployment ranking did not dictate to schedule the GPU version. Table 6.9 shows the average number of scheduled GPU-accelerated functions per node, over all experiments.

Table 6.9: Average number of scheduled containers per node and function

Function	Node type	Count
resnet50-inference	Nano	23.4
resnet50-inference	Xavier NX	10.0
resnet50-inference	TX2	4.6
resnet50-inference	XeonGpu	8.0
resnet50-training	Nano	25.8
resnet50-training	Intel NUC	50.4
resnet50-training	Xavier NX	9.0
resnet50-training	TX2	3.0

Table 6.10: *vanilla* approach

Function	Node type	Count
resnet50-inference	XeongGpu	3.0
resnet50-training	Nano	48.8
resnet50-training	Intel NUC	5.0
resnet50-training	Xavier NX	19.0
resnet50-training	TX2	6.4
resnet50-training	XeonGpu	5.0

Table 6.11: *ga* approach

CHAPTER 7

Discussion

In the following we discuss the results presented in Chapter 6. We start with results of our Baseline Profiling experiments and the Workload Characterization. Afterwards, we highlight our experience doing performance degradation benchmarks and using the data to create the prediction model. Before showing Capability Matching Problem optimization results, we discuss our k-means clusters. Both are important for the last part, the simulation. In which we discuss performance, degradation and discuss two experiments in detail.

7.1 Baseline Profiling & Workload Characterization

Baseline benchmarks were executed to obtain the FET and resource consumption per request, without any workload interference or concurrent requests. Figure 6.1 shows the performance aggregated over all devices. The baseline profiling results show a high variance regarding FET. A ranking including all devices is included in Figure 6.2. In general, the devices behave as expected with regards to performance. The Raspberry Pi 3 has a performance score of zero, because in all cases it took the longest to complete a request. The ranking for GPU-focused workloads shows a much higher difference between nodes. The high variance during the Fio workload is explainable by the different disks used in our testbed. Block I/O devices range from relatively slow SD Cards to NVME SSDs. Our results show further the decreased CPU usage time when utilizing GPUs. We perform CPU-based *resnet50-training* on the Intel NUC and use the GPUs of our other devices to run the same training procedure. In case of former the CPU usage is over 75% per request, while in the other case the CPU time decreases to under 20%. Due to the small size of our audio file (90kB) network metrics are lower than for Mobilenet and Resnet services. The reason for high RAM usage during TF-Gpu workloads is owed to the fact that VRAM and RAM are shared on Jetson devices.

We think that the resource vectors seem promising to base our workload characterization on them. We argue that the vectors describe each function accurately. For example, Fio utilizes the most block I/O data rate, while Python-Pi is CPU-only. Further, model training is a task which uses all resources and depicts this accurately. Inference functions utilize CPU, GPU and network, which intuitively is correct.

A limitation to this approach is that we only considered one input for each function. Different inputs could potentially lead to different resource usage of the same function. For example, no block I/O was registered during preprocessing. We attribute this to the small batch size of images, which nodes can store in-memory. In case of larger images this behavior may change, leading the resource vector to display block I/O. The same observation can be made for the *resnet50-training* function, where block I/O varies between devices. Further, *resnet50-inference-gpu* scales the image, and calls the inference engine always with the same image size. Larger images may increase network traffic, but because the image is always of same size, the GPU usage does not increase. This would lead to different resource vectors.

Another thing to note is that resource usage heavily depends on the device. For example, Speech-Inference-GPU resource vectors are very different between devices. This proves that the impact of workload on resources heavily depends on the device.

In the same way as resource usage differs, the FET also depends on the node. Which strengthens our argument, that schedulers need to be workload-aware to make placement decisions, that focus on providing the best performance.

7.2 Performance Degradation

Training and test datasets for learning a performance degradation prediction model are created by executing benchmarks targeted at causing performance. Due to the highly heterogeneous computing infrastructure, we need to adapt experiments based on each device. Most problematic is the Coral DevBoard, for which we use the Raspberry Pi 4 model, as we did not succeed in running any meaningful tests. Besides that, our results show a clear trend that devices have vastly different degradation characteristics. Low-performances, such as the Raspberry Pi 3, have experienced degradation of up to 2000%.

We use an AutoML library [48] to aid us in the search for a model as our focus lies on optimizing scheduling, but it is still important to include degradation in our simulation. The chosen pipeline has proven to be efficient in generalizing over different nodes as the test scores show in Table 6.1.

Results indicate that our input provides sufficient information to learn a predict degradation. A limitation to our approach is that we do not differ between functions. Because performance degradation may differ between functions on the same resource usage, our model's output may be not intuitive or possibly random.

Another caveat of our evaluation is that we do not test unknown functions. Training and test set may share similar data, leading to overfitting.

Further, as described in Section 6.3.2, we use the pre-calculated resource vectors when preparing our training data and not the actual resource usage during degradation experiments.

7.3 Workload Clustering

Clustering of functions is our approach to make the Capability Matching feasible for scenarios in which hundreds of different functions are deployed. This would result in executing the optimization step for each application. Instead, we apply clustering to create a fixed number of sets to which each function is assigned to. The assignment is done by k-means and its input is the workload characterization vector. Because we use different devices we aggregate the characterizations using mean values. Figure 6.5 shows our final assignments. We use the popular silhouette score and intuition to choose two configurations. The former creates five clusters in total. Comparing cluster formations with the individual resource usages reveal that there is a cluster for I/O heavy functions (**4**), CPU oriented tasks (**0**) and GPU applications (**3**). The remaining two clusters contain the two versions of *resnet50-training*. We think that this is a suitable clustering and accurately depicts reality. The subjective choice is made up of seven clusters. The main reason for choosing this clustering is mostly to put the Fio service into its own cluster, as we saw highly variant FET results and believe that it's best to separate it from other functions. Training versions are as before in their own cluster, as well as GPU oriented applications. A caveat of this approach is the fact that we need to introduce for each computing platform new clusters. We need to this, as we perform our capability matching problem based on the workload groups. Because our OpenFaaSExt system deploys static ranking, the scheduler can not actively recommend a computing platform. Therefore, it makes no sense to use the clusters as represented here, but split them further apart. Though the number of groups is still constant, which was the main reason to perform this step.

As the maximum silhouette score is 1, we think that there is room for improvement in this regard. First, we believe that there are too few functions of each type of workload (preprocessing, inference, training). It would make sense to implement multiple functions of each type. This would give us more confidence regarding generalization, though there are already distinct groups of workload recognizable.

Second, a sophisticated analysis of the relationship between execution times and clusters is in order. This analysis would validate if the relative performance across functions and devices is the same. In other words, if we rank devices, based on execution time, over each function in one cluster, is the ranking for each function the same? Perform nodes equally well across functions that we put in group? If not, is there a trend visible? Answering these questions, can aid our scheduling strategy by enabling estimating performance on a node, solely based on resource consumption.

7.4 Capability Matching Optimization

Parameter Tuning

The problem we face and solve is to map functions to appropriate nodes. Functions have requirements and nodes offer capabilities. Our problem formulation produces a list of requirements, which represent the probability of a node having a specific capability. We use two different input representations for the Capability Matching Problem. *ga* includes each device, and *enum* includes only device types. *ga* has the advantage, that its objective function could include runtime information into the optimization (i.e., current resource usage). In clusters with many nodes this may be infeasible. Our fitness function does not consider any runtime information and only balances the score between performance and heterogeneity. The influence both factors have on the solution can be seen in Table 6.3 and 6.4. In the former, results of *ga* are shown. Both approaches reach the same levels of performance and variety. Though, *enum* achieves a better score in case of high variety and low performance weights. The other difference lies in the duration that is used per cluster. Regarding *ga* it is interesting that the weights have an impact on the duration. Results indicate higher performance and lower variety weights result in shorter optimization steps. Further, both approaches favor higher performance weights but differ in the selected variety. *enum* scores highest with both parameters set to 1, while *ga* selects a performance weight of *0.75* and *0.25* as variety.

Further, function scores were re-calculated for each final solution set weights set to 1. This approach seems counterintuitive and not practical for two reasons. First, the re-calculated function score does not represent the final sum of mean FET and Heterogeneity Score reflect, which in some cases even invalidates the ranking. Second, we think that comparing scores is only useful in tuning the hyperparameters of the GA. Our approach to compare solutions based on resulting mean FET and heterogeneity seems more suitable in our situation.

Figures 6.6 and 6.7 present the influence weight and variety parameters have on FET and heterogeneity for k_5 and k_7 respectively. At first we are going to examine each individual clustering, starting with k_5, and conclude with an overall comparison. For each cluster setting we compare the resulting mean FET over all application clusters with regards to variety and performance weight. The goal is to see which approach (*enum* and *ga*) is affected the most by different parameters. This knowledge helps tuning weights as changes are of deterministic nature.

Using k_5 and comparing performance, the *ga* approach is much more susceptible to changes of parameters and performance drastically reduces with higher variety settings. In contrast to that, with *enum* the FET only reduces in case of high variety and lower performance settings, but is stable otherwise. The same behavior is visible regarding heterogeneity score. Parameters influence *ga* much more. In our opinion the outcome of our *ga* approach changes in expected fashion which aides further parameter optimization and makes tuning deterministic. Parameters did not influence *enum* in the same way. We think testing other weight parameters (i.e., variety = 10) is in order and would deepen the understanding of this approach.

k_7 results are similar in terms of effect the parameters have. *ga*'s mean FET changes drastically depending on variety and weight settings, while *enum*' mean FET changes only slightly and parameters do influence the result as much. The same is visible when looking at the heterogeneity score. One interesting observation can be made when comparing k_5 and k_7, mean FET and heterogeneity score differ not much. That means, that the optimization approach performs equally well over both clustering configurations. Further, the results of each optimization step, should in turn aid the scheduler in the same way. We see if this statement holds in the next section.

We extract the four best performing settings for each approach and clustering configuration. This selection competes against each other in Section 7.4 by running actual simulations.

Optimization Simulations

To determine which approach (*ga*, *enum*), weight setting (*performance*, *variety*) and cluster configuration (k_5 and k_7) to use for our final evaluation, we executed four simulations. We introduced the term *preference* to refer to the four sets of parameters, we determined in the previous section. Table 6.5 shows the settings for each preference.

The simulations were executed with the *hybrid* computing cluster and *sine* workload pattern.

Table 6.6 shows the average and standard deviation of performance degradation and additionally the scaled number of processed invocations.

With regards to the number of processed invocations, we can see that our *ga*-k_7 approach leads by a margin of 0.04 in comparison to our *enum*-k_5 approach. As mentioned before, we already suspected that all preferences may perform equally well. While there is a small difference, we think this is a positive outcome, as it shows that we can assume from our optimization results it will perform during the simulation. The same holds true for performance degradation as preferences differ only slightly compared to each other.

While the results seem to indicate that we can already conclude from heterogeneity score and mean FET of each preference, to validate this hypothesis, further experiments have to be conducted.

We conclude the optimizations by selecting *Preference 4*. This preference will be used in the next section to aid the CapabilityMatchingProblem. Before discussing the final simulation results, we want to highlight the fact, that we need to run the capability matching optimization for each of our three device sets. We do no pre-compute the requirements once for each function group, but calculate them for each set of devices (*hybrid*, *edge cloudlet*, *cloud*).

7.5 Simulations

This section is dedicated to discuss the results of our final simulations. We compare the four scheduling approaches described in Section 6.6: *ga*, *vanilla*, *all*, *skippy*.

First, we discuss performance (invocations, FET) and afterwards performance degradation. In case of degradation, we highlight one specific run, in which we show performance degradation and resource consumption as time-series. Afterwards we are going to show some detailed results of one scenario, which helps us explain some unexpected behavior.

7.5.1 Performance

This section is dedicated to performance, in our case the average FET. Figures 6.8 shows the total invocations averaged over all experiment runs, while 6.9 displays the FET.

To guarantee comparability, we pre-recorded five randomly generated workloads. This makes sure all simulations receive the same load. Figure 6.8 shows the total processed number of functions.. In the case of *cloud* and *hybrid*, our approach was able to process the most invocations, *skippy* finished more in the *edge cloudlet* scenario. Especially the *vanilla* approach had a lower number of processed invocations. We take a closer look at the differences between *ga* and *vanilla* in the *edge cloudlet* scenario. One thing to highlight is that the combination of our priority function and *skippy's* did succeed in providing good placements according to the FET and number of processed invocations.

Figure 6.9 shows the average FETs, and how succeeded in providing the lowest FET across all workloads and scenarios.

In comparison to *vanilla*, *ga* was able to decrease the FET between 33% to 68%, depending on the scenario and workload, as presented in Table 6.7.

Results have shown that *ga* scheduled function onto the best performing nodes. For example, *resnet50-inference* was always placed on *XeonGpus*, and Intel NUCs processed *resnet50-preprocessing* invocations. Further, scheduling on the node with lowest FET can negatively impact the number of invocations, because of our static deployment ranking.

In summary, while in some cases (*hybrid*) the FET may vary, in general we observe performance increase when our approach is used. The combination of *skippy* and *ga* performed in most cases equally well as *skippy*. Further, *vanilla* results indicate that only balancing resources has a negative effect on FET and processed invocations. We attribute this to the highly heterogeneous environment and that user assigned CPU and memory requirements may not contain enough knowledge. Because the performance gap between devices is in general very wide (compare our baseline profiling results in Section 6.1), it seems to us that more information is required to guarantee optimal placements regarding performance (FET & processed invocations).

7.5.2 Performance degradation

A goal of our scheduler is to avoid performance degradation caused by resource contention. To measure this, we use the predicted performance degradation factor.

Figures 6.10 shows the average degradation over all experiment runs for both workloads. The results demonstrate that our approach observed the least amount of performance degradation during our simulations. The reason behind this is that our approach simply does not multiple containers onto one node. To explain this behavior we have to consider two things: our scaling policy and placements. Results indicate that by placing *resnet50-inference* on the best performing node, the system did not need to scale this function up. We attribute this to making informed decisions, based on profiling data. Further, results show that our approach never placed more than one container on the same node. Our resource contention preventing priority function may be reason for this. But as we see in the next section, it is hard to reason about scheduling decisions and detailed analysis is required.

In conclusion, the decrease of degradation, in comparison to *vanilla*, ranges on average from 25% to 57%, as presented in Table 6.7. Further, it appears that our priority functions have a positive impact on the scheduling decisions in the *all* approach.

Individual run

Besides looking at the aggregated performance degradation, we examine resource usage and performance degradation over time in detail. Figure 6.11 shows the results of a single run over all approaches. On the left side, the performance degradation can be seen, on the right side the total resource utilization. We show the mean degradation over all devices, including standard deviation. The total resource utilization is calculated by using our resource vectors and is the normalized sum over all resources. Section 6.6.4 includes a detailed explanation.

The scenario is *cloud* and the workload pattern *constant*. We want to highlight a few key insights. First, both metrics visualize the constant workload that all approaches experienced. The Figures prove that this workload can be accurately depicted.

Second, the resource utilization implicitly shows when the system needs to scale up and deploys more containers. From the Figure, we can deduct that the *all* approach more frequently scaled up in the first 250 seconds. Because our workload characterization has shown, that devices differ in resource usage, we can not tell if the *all* approach had more running containers than the others. Further noticeable is that the *ga* approach made a better initial placements, as it was the last one to scale up. This means, that the nodes were able to process requests fast enough, such that scaling was not triggered. To determine which functions were not able to process requests fast enough, further analysis is required.

Third, results of the performance degradation display, that the *ga* approach had experienced less performance degradation than the others. On the other hand, *skippy* experienced the most. Figure 6.12 shows the number of nodes, which had on average multiple containers running. Results show that *ga* had no parallel containers running. And, *skippy* was very prone to put multiple containers on one node. We attribute the high performance degradation to this circumstance. Further, it seems plausible that in the *all* approach, our priority functions were able to positively influence placement by reducing the number of nodes with multiple containers. The same positive effect is visible in the performance degradation.

This examination shows how complex the system is and that scheduling in heterogeneous clusters requires significant amount analysis to reason about behavior.

7.5.3 *ga* vs *vanilla*

In the following we discuss the results presented in Table 6.7.

The Table shows the decrease of FET and performance degradation by using *ga* instead of *vanilla*. Performance Degradation results indicate *vanilla* is closer to *ga* than during the others scenarios. This validates the assumption that resource contention may happen less often than in edge scenarios. In the *edge cloudlet* scenario, our approach had the largest decrease of FET and performance degradation. We think that this may be related to the node distributions, because in this scenario has the lowest number of cloud VM instances. Therefore, the lack of workload-awareness resulted in not optimal placements. Surprising is that in the *hybrid* scenario, during *sine* workload, the difference regarding FET is lowest. Our assumption is, that in this scenario more VM instances and Coral DevBoards are available. Because the deployment ranking of *mobilenet-inference* dictates to use the TPU computing platform first, all approaches put this function automatically on nodes with very low FET.

7.5.4 Edge Cloudlet

In the following, we take a close look at the results of our *edge cloudlet* scenario and concentrate on the *ga* and *vanilla* approach.

To determine functions to examine in detail, we show the number of invocations, presented in 6.8. The functions, which had the largest difference in processed invocations between *vanilla* and *ga* were: *speech-inference*, *resnet50-preprocessing* and *resnet50-training*. We focus on *resnet50-preprocessing* and *resnet50-training*. In the former, *ga* was able to process more invocations than in the latter case. This gives us the ability to showcase positive and negative sides.

Figure 6.13 shows the invocations as barplots to display the variance of function calls. The *ga* approach has a larger variance in case of *resnet50-preprocessing* than *vanilla*. The same behavior is recognizable for the *resnet50-training* function in the *vanilla* approach. We can reason about these observations by examining the actual placements. Figure 6.14 depicts the number of containers for training and preprocessing per node type. Figure 6.14a shows the nodes, which executed *resnet50-preprocessing*. In each case our approached scheduled the *resnet50-preprocessing* function onto *Intel NUC* units. The reason is that our *ExecutionTimePriority* favors this node. *vanilla* scheduled it onto four different types of nodes across all runs, especially *RockPi* and *RPI 4* are not suited. This explains why, *vanilla* experiments processed less preprocessing functions.

The other question to be answered is: why did *ga* process less *resnet50-training* requests? Figure 6.14a shows the number of scheduled containers per node for the *resnet50-training* function. In contrast to *ga*, *vanilla* has scheduled this function much more often onto *Intel NUC* instances. While *ga* has used almost exclusively only GPU-equipped nodes. To explain these placements, we have to highlight the fact that there are two computing platforms available for *resnet50-training*. One supporting GPU, one operating on CPU. Our deployment ranking for this function dictates that first, GPU platforms are used and afterwards, CPU ones. Results show there was a considerably high amount of *resnet50-training* functions scheduled onto the *Intel NUC* node, an explanation is that GPU nodes were occupied. To back this claim, we look at the average scheduled containers per node and per function. Tables 6.10 and 6.11 shows functions that have support for GPU computing platforms. The tables show that *ga* used almost all GPU devices for *resnet50-training*, *vanilla* spread them between *resnet50-inference* and *resnet50-training*.

Therefore, we conclude that the higher number of *resnet50-training* invocations, stems from the reason that GPU nodes were used in the *ga* approach. Which may perform worse than the *Intel NUC*. And due to the fact, that the *vanilla* approach has used the GPU nodes for inference, it had to use the CPU version.

This showcases that the deployment ranking, paired with *ga*, can produce in some cases worse results regarding processed number of invocations.

CHAPTER 8

Conclusion

Emerging concepts accelerate the adaption of heterogeneous computing infrastructure. The edge computing paradigm pushes resources towards users, enabling processing at the origin of data and making it possible to implement applications, such as Cognitive AR. These edge computing infrastructures contain various devices, ranging from Single Board Computers, over specialized embedded hardware to cloudlets and VM instances, located in the cloud. The devices offer different capabilities and range in performance. To help developers deploy their application on edge computing infrastructures, research has proposed to merge the paradigm of edge and serverless computing, resulting in serverless edge computing. Serverless computing allows users to upload containers that contain a single function. This approach has the advantage of rapid automatic scaling, which emphasis the importance of schedulers to make optimal placements regarding execution time and preventing resource contention. Current platforms are incapable of making well-founded scheduling decisions in these scenarios, which results in potentially inefficient placements.

To this end, we propose a strategy, based around workload characterization, to make the scheduler workload-aware. We perform extensive profiling of multiple applications and introduce a systematic representation of heterogeneous clusters to describe node capabilities. We are able to apply machine learning to group applications based on resource consumption. Further, we define a problem that maps applications to appropriate node capabilities. The workload clustering makes our optimization approach tractable regarding a growing number of applications.

We make the scheduler workload-aware by adding priority functions that focus on: execution time, resource contention and capability mapping. Simulation results show that our improvements lead to a reduced Function Execution Time and prevention of resource contention in comparison to previously released location and data-focused priority functions [58] and the default scheduler.

8.1 Research Questions

- **RQ. 1**: What are appropriate methods for workload characterization based on black-box system metrics in serverless edge computing systems?

  Devices in serverless edge computing systems are highly heterogeneous by offering different capabilities, resulting in a wide range of response times. This poses a problem for function scheduling, as current schedulers lack the ability to reason about workloads.

  To this end, we use two kinds of black-box system metrics: static and runtime. First, we define a systematic approach to describe heterogeneous clusters based on automatically retrieved node capabilities. This enables us to formulate a problem which maps workloads to their appropriate nodes based on their capabilities.

  Second, we profile multiple applications extensively on nodes in our testbed and monitor in total five resources during runtime. Based on traces and telemetry data, we create resource vectors for each application and node. The results show an intuitively accurate depiction of resource usage, including each application's impact on different devices. We apply a clustering technique on these vectors to find groups of similar applications. These groups are used to make our capability matching problem feasible regarding new and large amounts of functions.

  A caveat is the discrepancy between computing platforms and workload groups. In our approach it is possible that applications get the same group assigned, but do not share the computing platform. Therefore, we have to split these clusters and group them according to the computing platform in order to perform our capability matching optimization. This limitation is deeply rooted in the approach of unifying multiple computing platforms to provide users the convenience of not caring about their deployment.

  During our evaluation we encountered limitations regarding our system modeling approach. First, some type of devices are equipped with more than one CPU core type. Second, in edge topologies it is common that nodes may be interconnected with each other and therefore have a different bandwidth than they have to the cloud. This is not exclusive to edge deployments but also common in cloud hosted services, where VM instances share a higher bandwidth between each other than they expose to the world. In both cases our approach of describing clusters is not able to depict these characteristics.

  We have shown that our monitoring approach in combination with function-based applications allows to transparently monitor and characterize functions. The capabilities help us match applications with appropriate nodes. Further, we evaluate nine different device types, including only one cloud node and are therefore confident that our solution is feasible in serverless edge computing systems.

- **RQ. 2**: How can we use workload characterization in scheduling of serverless edge functions?

Scheduling functions in a diverse scenario, such as edge computing, can result in unexpected FETs. Our baseline experiments show that nodes can substantially differ in performance. This is not restricted to performance differences between devices, but can be observed on a single device. The reason for this is, that devices may offer different computing platforms (i.e., GPU, TPU, CPU). Another problem we measured is the performance degradation caused by co-located containers. Further, we showed that the degradation highly depends on the node. While cloud VM instances are more robust, resource-constrained nodes at the edge may experience much higher degradation. Therefore, we use our workload characterization to implement three priority functions that make the scheduler workload-aware.

The first priority function matches node capabilities with application requirements. The goal is to favor nodes similar to the preferences of an application. This is possible by using the result of our capability matching problem, which is a list of favorable capabilities. The parameters, used in the optimization step, give control over the focus between performance and variety. To prevent executing the optimization for every function, we use our groups of similar applications.

The second priority function prevents resource contention by using previously obtained workload characteristics, for each node and application. It sums up the current utilization of the node and subtracts it from the new applications total utilization. This favors nodes that have a lower resource usage. While Rausch et al. [58] have implemented a priority function that behaves similar, but only takes CPU and memory into consideration, ours includes other resources (GPU, I/O). Further, their priority function uses CPU and memory requirements assigned by users. We use our workload characterization, which we have available for each node. Our results have shown that each workload has a different impact on resources depending on the node. In their case, they assign only an estimate for CPU and memory, which lacks awareness of heterogeneous environment.

The third priority function uses the baseline profiling data, which consists of the mean FET for each application and node in the cluster. It scores the nodes according to the measured FET and favors faster ones.

During our evaluation we discovered some limitations. Allowing users to upload multiple functions, supporting different computing platforms, introduces the challenge of deciding which one to deploy during runtime. We discovered the same limitation when using Docker. Users can not group multiple computing platforms under one image. Further, this would introduce the same problem, as the Docker runtime would need to autonomously decide which runtime to use. To this end, we use a static deployment ranking, defined by users, to mitigate this issue. As our workload clusters are already separated according to platforms, our optimization strategy does not suggest or change the deployment ranking. Therefore, the system uses in all cases the static deployment ranking. As our results show, this can negatively impact a system's performance regarding processed invocations.

Our resource contention priority disregards differences in resources by using the total over all resources. This may hinder optimal resource usage by discounting the fact that some applications only use certain resources.

Another limitation to our work lies in the evaluation. We have not tested the priority functions individually or examined the importance of each function during the scheduling process. Rausch et al. [58] have shown that it is not a trivial task and heavily depends on the system's structure. For example, the scheduling process is not going to benefit from our capability priority function in a cluster consisting of only a one device type.

- **RQ. 3**: How can a workload-aware scheduler improve the quality of function placement in serverless edge computing systems?

  Our evaluation compares four scheduling pipelines and three compute clusters. We use the default scheduler settings, locality and data-aware priorities [58], our approach, and a combination of all custom priorities. The clusters have different levels of heterogeneity and represent a cloud-centered, hybrid and edge-cloudlet system.

  The results focus on FET and performance degradation, caused by resource contention. In comparison to the default scheduler, we are able to decrease FET on average between 33% and 68%. Degradation reduces on average by 25% to 57%. Further, our approach was able to significantly process more requests on average in three out of six scenarios. While in the most other cases all approaches performed equally. Detailed analysis of experiments has revealed the negative side of our static deployment ranking and that dynamic decisions may lead to more preferable placements.

  Further, due to our performance oriented settings, it is not surprising that our approach chooses the best performing nodes. This was most visible in case of *resnet50-inference*, which our approach scheduled on the fastest node and did not trigger any further scaling.

  This brings us to another caveat: the chosen workload patterns may not spawn enough requests to fully utilize clusters, as we saw in our results.

  The simulations we have performed demonstrated that the current default scheduler is not appropriate and a workload-aware scheduler dramatically improves performance and prevents contention.

  In conclusion, the heterogeneous landscape offers different kinds of specialized hardware, capable of performing certain tasks very efficient. Many devices deployed at the edge suffer from high performance degradation in multi-tenancy situations. We add awareness regarding performance, resource contention and capabilities to the scheduler. Therefore, a workload-aware scheduler can improve the quality of function placement regarding Function Execution Time and performance degradation.

8.2 Future work

This section highlights limitations of our work and proposes goals for future work.

- Extensions to the system modeling has to be done in order to allow more complex device descriptions (i.e., two CPUs equipped).
- We simulate only a single type of OpenFaaS watchdog (HTTP). Other watchdogs have advantages. For example, the process-per-request approach is suitable for long-running tasks.
- Regarding the capability matching, new solution representations can be developed. Of particular interest are ones that represent resource usages of nodes. This leads to periodically running the optimization, resulting in a deepened evaluation of the heuristic optimization technique.
- The performance degradation model can be extended by adding awareness of the service to predict. We propose two approaches to overcome this issue. First, append the specific service to the function, probably leading to very precise estimates. We think the other approach is going generalize better by predicting unknown or new functions and reduced datasets. Instead of the specific function, one could use the workload cluster as discriminating factor.
- Priority weights remained untouched in our scenario and can be subject to further optimizations. Especially a combined look at *skippy*'s and ours appears promising.
- A more detailed analysis of our results can reveal more unexpected behavior. This task is not straightforward in such diverse scenarios. Functions, devices, cluster configurations have to be carefully considered. And all components are complex: multiple computing platforms, different device characteristics and varying node distributions.

APPENDIX A

Baseline Performance

The following table shows the results of all baseline benchmarks we execute. Table contains statistical descriptions of our measured Function Execution Time in seconds. Abbrevations used:

- Resnet Pre.: Resnet50-preprocessing
- Resnet Inf.: Resnet50-inference
- Resnet Tra.: Resnet50-training
- Speech Inf.: Speech-inference
- Coral: Coral Devboard
- TX2: Nvidia TX2
- Xavier NX: Nvidia Xavier NX
- Nano: Nvidia Nano
- Mobilenet Tflite: Mobilenet-inference Tflite
- Mobilenet TPU: Mobilenet-inference TPU

Service	Device	$\bar{x}$	σ	min.	max.	25th	75th	99th
Fio	Intel NUC	1.12	0.07	1.09	1.37	1.09	1.09	1.09
Fio	Nano	19.64	0.73	17.63	21.38	17.69	17.81	17.87
Fio	TX2	4.31	0.12	4.20	4.51	4.20	4.20	4.20
Fio	Xavier NX	13.77	0.34	13.39	14.50	13.39	13.39	13.39
Fio	RPI 3	28.66	5.15	23.95	41.16	23.96	23.98	24.00
Fio	RPI 4	27.81	1.72	23.44	33.21	23.44	23.45	23.46
Fio	RockPi	21.99	0.39	21.46	23.43	21.46	21.46	21.47
Fio	XeonGpu	1.14	0.02	1.12	1.20	1.12	1.12	1.12
Mobilenet Tflite	Coral	0.66	0.01	0.65	0.70	0.65	0.66	0.66
Mobilenet Tflite	Intel NUC	0.28	0.01	0.26	0.31	0.27	0.27	0.27
Mobilenet Tflite	Nano	0.45	0.00	0.44	0.46	0.44	0.44	0.44
Mobilenet Tflite	TX2	0.33	0.00	0.32	0.34	0.32	0.32	0.32
Mobilenet Tflite	Xavier NX	0.33	0.01	0.32	0.35	0.32	0.32	0.32
Mobilenet Tflite	RPI 3	2.07	0.07	1.97	2.31	1.97	1.97	1.97
Mobilenet Tflite	RPI 4	1.28	0.02	1.26	1.34	1.26	1.26	1.26
Mobilenet Tflite	RockPi	0.52	0.07	0.37	0.69	0.38	0.39	0.39
Mobilenet Tflite	XeonGpu	0.28	0.01	0.27	0.32	0.27	0.27	0.27
Mobilenet Tpu	Coral	0.55	0.01	0.53	0.58	0.53	0.53	0.53
Python Pi	Coral	1.82	0.61	1.44	3.03	1.44	1.44	1.44
Python Pi	Intel NUC	0.25	0.00	0.24	0.26	0.24	0.24	0.24
Python Pi	Nano	0.75	0.00	0.74	0.76	0.74	0.74	0.74
Python Pi	TX2	0.55	0.00	0.54	0.56	0.54	0.54	0.54
Python Pi	Xavier NX	0.83	0.01	0.79	0.85	0.79	0.80	0.80
Python Pi	RPI 3	25.67	0.22	25.35	26.30	25.35	25.36	25.37
Python Pi	RPI 4	23.59	0.20	23.26	23.97	23.26	23.26	23.26
Python Pi	RockPi	0.92	0.01	0.90	0.96	0.90	0.90	0.90
Python Pi	XeonGpu	71.59	0.24	71.26	72.22	71.26	71.27	71.27
Resnet Inf. Cpu	Coral	2.18	0.52	1.41	3.64	1.42	1.43	1.44
Resnet Inf. Cpu	Intel NUC	0.16	0.02	0.14	0.31	0.14	0.14	0.14
Resnet Inf. Cpu	Nano	0.93	0.05	0.88	1.32	0.88	0.89	0.89
Resnet Inf. Cpu	TX2	0.74	0.06	0.69	1.14	0.70	0.70	0.70
Resnet Inf. Cpu	Xavier NX	0.51	0.03	0.45	0.60	0.45	0.46	0.46
Resnet Inf. Cpu	RPI 3	5.10	0.44	4.43	5.86	4.43	4.43	4.43
Resnet Inf. Cpu	RPI 4	2.91	0.09	2.77	3.45	2.77	2.77	2.78
Resnet Inf. Cpu	RockPi	1.38	0.12	1.13	1.92	1.13	1.14	1.14
Resnet Inf. Cpu	XeonGpu	0.17	0.03	0.15	0.31	0.15	0.15	0.15
Resnet Inf. Gpu	Nano	0.73	0.12	0.67	1.84	0.67	0.67	0.67
Resnet Inf. Gpu	TX2	0.39	0.04	0.38	0.81	0.38	0.38	0.38
Resnet Inf. Gpu	Xavier NX	0.39	0.04	0.37	0.71	0.37	0.37	0.37
Resnet Inf. Gpu	XeonGpu	0.13	0.02	0.11	0.32	0.11	0.11	0.11

Table A.1: Workload characterization results

Service	Device	$\bar{x}$	σ	min.	max.	25th	75th	99th
Resnet Pre.	Coral	10.51	0.05	10.41	10.66	10.41	10.41	10.42
Resnet Pre.	Intel NUC	2.53	0.03	2.47	2.59	2.47	2.48	2.48
Resnet Pre.	Nano	7.95	0.18	7.85	9.17	7.85	7.85	7.85
Resnet Pre.	TX2	6.39	0.08	6.22	6.59	6.23	6.25	6.25
Resnet Pre.	Xavier NX	6.08	0.13	5.81	6.87	5.82	5.84	5.85
Resnet Pre.	RPI 3	30.48	1.03	29.42	37.48	29.42	29.42	29.42
Resnet Pre.	RPI 4	19.50	0.26	18.99	20.01	19.00	19.02	19.02
Resnet Pre.	RockPi	7.65	0.13	7.35	8.14	7.36	7.38	7.39
Resnet Pre.	XeonGpu	2.66	0.03	2.58	2.74	2.58	2.59	2.59
Resnet Tra. Cpu	Intel NUC	197.45	0.82	196.38	202.84	196.42	196.52	196.56
Resnet Tra. Gpu	Nano	847.17	764.41	475.31	3758.72	475.67	476.38	476.72
Resnet Tra. Gpu	TX2	228.12	2.16	225.43	234.05	225.44	225.45	225.45
Resnet Tra. Gpu	Xavier NX	142.00	1.13	139.36	144.52	139.38	139.41	139.43
Resnet Tra. Gpu	XeonGpu	32.13	0.42	31.53	33.14	31.54	31.54	31.55
Speech Inf. Gpu	Nano	4.54	0.08	4.37	4.69	4.37	4.39	4.39
Speech Inf. Gpu	TX2	3.31	0.03	3.18	3.38	3.19	3.22	3.23
Speech Inf. Gpu	Xavier NX	1.65	0.03	1.59	1.84	1.59	1.59	1.59
Speech Inf. Gpu	XeonGpu	0.75	0.02	0.71	0.79	0.71	0.72	0.72
Speech TFlite	Coral	7.21	0.04	7.14	7.32	7.14	7.14	7.14
Speech TFlite	Intel NUC	1.06	0.01	1.05	1.09	1.05	1.05	1.05
Speech TFlite	Nano	3.89	0.04	3.81	4.03	3.81	3.81	3.82
Speech TFlite	TX2	3.44	0.04	3.36	3.55	3.36	3.37	3.37
Speech TFlite	Xavier NX	2.68	0.05	2.51	2.81	2.51	2.52	2.52
Speech TFlite	RPI 3	16.56	0.31	15.98	17.31	16.00	16.05	16.07
Speech TFlite	RPI 4	6.75	0.05	6.63	6.85	6.63	6.63	6.63
Speech TFlite	RockPi	3.82	0.05	3.66	3.97	3.66	3.66	3.66
Speech TFlite	XeonGpu	1.08	0.01	1.05	1.12	1.06	1.06	1.06
Tf Gpu	Nano	0.21	0.05	0.18	0.37	0.18	0.18	0.18
Tf Gpu	TX2	1.89	0.03	1.87	2.14	1.88	1.88	1.88
Tf Gpu	Xavier NX	1.17	0.01	1.16	1.21	1.16	1.16	1.16
Tf Gpu	XeonGpu	0.37	0.00	0.36	0.38	0.36	0.36	0.36

Table A.2: Workload characterization results

APPENDIX B

Workload Characterization

Besides all performance results, we include additionally the accompanying processed telemetry data. This data is used to simulate resource usage and acts as input for our clustering pipeline. Columns related to data, i.e. *BLKIO* are in megabytes, whereas *BLKIO* and *NET* show the data rates (MB/s) while *BLKIO_TOTAL* and *NET_TOTAL* are the total amount of MB read or written for one call.

Abbrevations used:

- Resnet Pre.: Resnet50-preprocessing
- Resnet Inf.: Resnet50-inference
- Resnet Tra.: Resnet50-training
- Speech Inf.: Speech-inference

Device	Service	CPU	BLKIO	NET	GPU	RAM	i/o tot.	net tot.
Intel Nuc	Fio	0.09	443.47	0.00	0.00	0.03	498.08	0.00
Nvidia Nano	Fio	0.06	6.68	0.00	0.00	0.28	131.07	0.00
Nvidia TX2	Fio	0.12	30.37	0.00	0.00	0.10	131.07	0.00
Nvidia Xavier NX	Fio	0.03	9.51	0.00	0.00	0.13	131.07	0.00
RPI 3	Fio	0.06	2.23	0.00	0.00	0.65	64.02	0.00
RPI 4	Fio	0.06	2.21	0.00	0.00	0.52	61.32	0.00
RockPi	Fio	0.03	4.77	0.00	0.00	0.28	104.86	0.00
XeonGpu	Fio	0.17	105.92	0.00	0.00	0.03	123.27	0.00
Coral DevBoard	Mobilenet TFlite	0.27	0.00	15.98	0.00	0.12	0.00	10.62
Intel Nuc	Mobilenet TFlite	0.21	0.02	38.89	0.00	0.03	0.00	11.03
Nvidia Nano	Mobilenet TFlite	0.46	0.00	22.16	0.00	0.07	0.00	11.04
Nvidia TX2	Mobilenet TFlite	0.49	0.00	32.55	0.00	0.03	0.00	11.01
Nvidia Xavier NX	Mobilenet TFlite	0.27	0.00	26.98	0.00	0.02	0.00	11.04
RPI 3	Mobilenet TFlite	0.30	0.00	5.18	0.00	0.14	0.00	10.72
RPI 4	Mobilenet TFlite	0.39	0.00	8.32	0.00	0.09	0.00	10.75
RockPi	Mobilenet TFlite	0.34	0.00	21.19	0.00	0.04	0.00	10.77
XeonGpu	Mobilenet TFlite	0.42	0.00	33.40	0.00	0.02	0.00	10.85
Coral DevBoard	Mobilenet Tpu	0.35	0.00	19.73	0.00	0.20	0.00	10.81
Coral DevBoard	Python Pi	0.36	0.74	0.00	0.00	0.57	3.48	0.00
Intel Nuc	Python Pi	0.16	0.01	0.00	0.00	0.01	0.00	0.00
Nvidia Nano	Python Pi	0.42	0.00	0.00	0.00	0.03	0.00	0.00
Nvidia TX2	Python Pi	0.37	0.00	0.00	0.00	0.03	0.00	0.00
Nvidia Xavier NX	Python Pi	0.27	0.00	0.00	0.00	0.01	0.00	0.00
RPI 3	Python Pi	0.26	0.00	0.00	0.00	0.07	0.00	0.00
RPI 4	Python Pi	0.26	0.00	0.00	0.00	0.08	0.00	0.00
RockPi	Python Pi	0.26	0.00	0.00	0.00	0.03	0.00	0.00
XeonGpu	Python Pi	0.25	0.00	0.00	0.00	0.00	0.00	0.00
Coral DevBoard	Resnet Inf. Cpu	0.38	22.64	5.10	0.00	0.56	68.92	10.62
Intel Nuc	Resnet Inf. Cpu	0.44	0.04	55.70	0.00	0.05	0.01	11.06
Nvidia Nano	Resnet Inf. Cpu	0.62	0.00	11.93	0.00	0.17	0.00	11.05
Nvidia TX2	Resnet Inf. Cpu	0.64	0.00	14.92	0.00	0.09	0.00	11.06
Nvidia Xavier NX	Resnet Inf. Cpu	0.32	0.00	16.69	0.00	0.08	0.00	10.95
RPI 3	Resnet Inf. Cpu	0.54	0.00	2.06	0.00	0.37	0.01	10.61
RPI 4	Resnet Inf. Cpu	0.62	0.00	2.79	0.00	0.31	0.01	10.75
RockPi	Resnet Inf. Cpu	0.47	0.00	6.60	0.00	0.16	0.00	10.77
XeonGpu	Resnet Inf. Cpu	0.72	0.00	63.57	0.00	0.04	0.00	13.64
Nvidia Nano	Resnet Inf. Gpu	0.31	0.30	14.10	0.18	0.74	32.75	11.34
Nvidia TX2	Resnet Inf. Gpu	0.51	0.00	26.96	0.04	0.43	0.00	10.98
Nvidia Xavier NX	Resnet Inf. Gpu	0.33	0.00	26.65	0.04	0.41	0.00	11.06
XeonGpu	Resnet Inf. Gpu	0.37	0.00	57.66	0.00	0.13	0.00	11.04

Table B.1: Workload characterization results

Device	Service	CPU	BLKIO	NET	GPU	RAM	i/o tot.	net tot.
Coral DevBoard	Resnet Preprocessing	0.24	0.00	2.12	0.00	0.15	0.02	26.20
Intel Nuc	Resnet Preprocessing	0.11	0.00	10.64	0.00	0.02	0.00	27.39
Nvidia Nano	Resnet Preprocessing	0.24	0.00	3.13	0.00	0.08	0.00	26.19
Nvidia TX2	Resnet Preprocessing	0.25	0.00	3.88	0.00	0.03	0.00	26.22
Nvidia Xavier NX	Resnet Preprocessing	0.16	0.00	4.18	0.00	0.02	0.00	26.24
RPI 3	Resnet Preprocessing	0.24	0.00	0.61	0.00	0.13	0.00	20.87
RPI 4	Resnet Preprocessing	0.25	0.00	1.35	0.00	0.09	0.00	26.32
RockPi	Resnet Preprocessing	0.16	0.00	3.21	0.00	0.04	0.00	26.25
XeonGpu	Resnet Preprocessing	0.21	0.00	10.23	0.00	0.08	0.00	27.33
Intel Nuc	Resnet Training Cpu	0.88	0.23	1.42	0.00	0.58	74.20	281.97
Nvidia Nano	Resnet Training Gpu	0.09	16.69	0.45	0.61	0.83	31035.99	1042.25
Nvidia TX2	Resnet Training Gpu	0.18	0.21	1.14	0.56	0.60	55.44	574.59
Nvidia Xavier NX	Resnet Training Gpu	0.14	0.03	5.72	0.51	0.56	32.02	1083.56
XeonGpu	Resnet Training Gpu	0.24	0.01	15.75	0.60	0.28	0.49	829.94
Nvidia Nano	Speech Inf. Gpu	0.18	0.00	0.13	0.36	0.35	0.00	0.60
Nvidia TX2	Speech Inf. Gpu	0.25	0.00	0.15	0.21	0.20	0.00	0.51
Nvidia Xavier NX	Speech Inf. Gpu	0.15	0.00	0.28	0.21	0.12	0.00	0.46
XeonGpu	Speech Inf. Gpu	0.28	0.00	0.51	0.08	0.04	0.00	0.40
Coral DevBoard	Speech Inf. Tflite	0.25	0.00	0.05	0.00	0.09	0.00	0.36
Intel Nuc	Speech Inf. Tflite	0.12	0.00	0.38	0.00	0.01	0.00	0.40
Nvidia Nano	Speech Inf. Tflite	0.25	0.00	0.09	0.00	0.06	0.00	0.37
Nvidia TX2	Speech Inf. Tflite	0.27	0.00	0.11	0.00	0.02	0.00	0.37
Nvidia Xavier NX	Speech Inf. Tflite	0.17	0.00	0.14	0.00	-0.02	0.00	0.37
RPI 3	Speech Inf. Tflite	0.25	0.00	0.02	0.00	0.18	0.00	0.37
RPI 4	Speech Inf. Tflite	0.25	0.00	0.06	0.00	0.14	0.00	0.37
RockPi	Speech Inf. Tflite	0.17	0.00	0.10	0.00	0.03	0.00	0.37
XeonGpu	Speech Inf. Tflite	0.25	0.00	0.35	0.00	0.00	0.00	0.37
Nvidia Nano	Tf Gpu	0.07	0.00	0.02	0.08	0.46	0.00	0.00
Nvidia TX2	Tf Gpu	0.06	0.00	0.00	0.57	0.27	0.00	0.00
Nvidia Xavier NX	Tf Gpu	0.02	0.00	0.00	0.50	0.29	0.00	0.00
XeonGpu	Tf Gpu	0.24	0.00	0.01	0.06	0.07	0.00	0.00

Table B.2: Workload characterization results

APPENDIX C

Performance Degradation Experiments

This chapter and its tables show all experiments we can run successful on our devices, which serve as training datasets for our performance degradation model training.

The interfering function's number of instances and workers is displayed in the first two rows. For example in Table C.1 the first column represents an experiment in which we started one *Resnet50-Inference* function which got requests from a single client and an inter arrival time of one second. The *Python Pi* function interfered with one container client, whereas in this case the inter arrival time is zero.

Tables C.1 and C.2 show the executed benchmarks.

	Python Pi												**Fio**			
Containers	1	3	1	3	1	3	1	3	1	3	1	3	1	2	1	2
Workers	1	1	2	2	3	3	1	1	2	2	3	3	1	1	2	2
Clients	1	1	1	1	1	1	2	2	2	2	2	2	1	1	1	1
Coral DevBoard																
Intel Nuc	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Nvidia Nano	X	X	X	X	X	X	X	X	X	X	X	X	X			X
Nvidia TX2	X	X	X	X	X	X							X	X	X	X
Nvidia Xavier NX	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RPI 3	X	X	X	X			X		X				X			
RPI 4	X	X	X	X	X	X	X	X	X	X	X	X	X			
RockPi	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
XeonGpu	X	X	X	X		X	X	X	X	X			X	X	X	X

Table C.1: Performance degradation experiment configurations

	Resnet50-Inference									**TF-GPU**
Containers	1	1	1	2	2	2	1	1	1	1
Workers	1	2	3	1	2	3	1	2	3	1
Clients	1	1	1	1	1	1	2	2	2	1
Coral DevBoard										
Intel Nuc	X	X	X	X	X	X	X	X	X	X
Nvidia Nano	X	X	X	X	X	X	X	X	X	X
Nvidia TX2	X	X	X	X	X	X	X	X	X	
Nvidia Xavier NX	X	X	X	X	X	X	X	X	X	X
RPI 3	X	X	X	X	X	X				
RPI 4	X	X	X		X					
RockPi	X	X	X	X						
XeonGpu	X	X	X	X	X	X				X

Table C.2: Performance degradation experiment configurations

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Amazon. Aws re:invent 2014 | (mbl202) new launch: Getting started with aws lambda. Online. Accessed 2021-02-12. `"https://youtu.be/UFj27laTWQA"`.

[2] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[3] Mohammad S Aslanpour, Adel N Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: vision and challenges. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.

[4] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[5] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[6] Christian Blum and Günther R Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer, 2016.

[7] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS Netto, et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, 51(5):1–38, 2018.

[8] Emiliano Casalicchio. *Container Orchestration: A Survey*, pages 221–235. Springer International Publishing, Cham, 2019.

[9] Charles E Catlett, Peter H Beckman, Rajesh Sankaran, and Kate Kusiak Galvin. Array of things: a scientific research instrument in the public way: platform design and early lessons learned. In *Proceedings of the 2nd international workshop on science of smart city operations and platforms engineering*, pages 26–33, 2017.

[10] Lukas Cavigelli, Philippe Degen, and Luca Benini. Cbinfer: Change-based inference for convolutional neural networks on video data. In *Proceedings of the 11th International Conference on Distributed Smart Cameras*, pages 1–8, 2017.

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

[12] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, et al. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017.

[13] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[14] containerd. containerd. Online. Accessed 2021-02-12., `https://containerd.io/`.

[15] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[16] Waltenegus Dargie. Identification of resource utilisation patterns in data centers using tensor decomposition. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2019.

[17] 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*, pages 1–1, 2017.

[18] Tom Goethals, Filip DeTurck, and Bruno Volckaert. Extending kubernetes clusters to low-resource edge devices using virtual kubelets. *IEEE Transactions on Cloud Computing*, 2020.

[19] Google. Google cloud functions. Online. Accessed 2021-02-12. `"https://cloud.google.com/functions"`.

[20] Johannes Grohmann, Patrick K Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. Monitorless: Predicting performance degradation in cloud applications with machine learning. In *Proceedings of the 20th international middleware conference*, pages 149–162, 2019.

[21] Gopal K Gupta. *Introduction to data mining with case studies.* PHI Learning Pvt. Ltd., 2014.

[22] X. Han, R. Schooley, D. Mackenzie, O. David, and W. J. Lloyd. Characterizing public cloud resource contention to support virtual machine co-residency prediction. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 162–172, 2020.

[23] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[24] Aurelien Havet, Valerio Schiavoni, Pascal Felber, Maxime Colmant, Romain Rouvoy, and Christof Fetzer. Genpack: A generational scheduler for cloud data centers. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 95–104. IEEE, 2017.

[25] Jiong He, Yao Chen, Tom ZJ Fu, Xin Long, Marianne Winslett, Liang You, and Zhenjie Zhang. Haas: Cloud-based real-time data analytics with heterogeneity-aware scheduling. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1017–1028. IEEE, 2018.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[27] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[28] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. Modelops: Cloud-based lifecycle management for reliable and trusted ai. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 113–120. IEEE, 2019.

[29] Ali R Hurson, Evens Jean, Machigar Ongtang, Xing Gao, Yu Jiao, and Thomas E Potok. Recent advances in mobile agent-oriented applications. *Mobile Intelligence*, pages 106–139, 2010.

[30] IBM. Ibm openwhisk. Online. Accessed 2021-02-12. `"https://www.ibm.com/cloud/functions"`.

[31] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[32] Runyu Jin, Qirui Yang, and Ming Zhao. Is faas suitable for edge computing? USENIX Association, June 2020.

[33] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[34] C. T. Joseph, J. P. Martin, K. Chandrasekaran, and A. Kandasamy. Fuzzy reinforcement learning based microservice allocation in cloud computing environments. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pages 1559–1563, 2019.

[35] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Kathryn Tunyasuvunakool, Olaf Ronneberger, Russ Bates, Augustin Žídek, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Anna Potapenko, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Martin Steinegger, Michalina Pacholska, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. *Fourteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstract Book)*. 2020.

[36] Matthew LeMay, Shijian Li, and Tian Guo. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 66–72. IEEE, 2020.

[37] Wes Lloyd, Shrideep Pallickara, Olaf David, Mazdak Arabi, and Ken Rojas. Mitigating resource contention and heterogeneity in public clouds for scientific modeling services. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–166. IEEE, 2017.

[38] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2017.

[39] Sajee Mathew and J Varia. Overview of amazon web services. *Amazon Whitepapers*, 2014.

[40] Víctor Medel, Carlos Tolón, Unai Arronategui, Rafael Tolosana-Calasanz, José Ángel Bañares, and Omer F. Rana. Client-side scheduling based on application characterization on kubernetes. In Congduc Pham, Jörn Altmann, and José Ángel Bañares, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 162–176, Cham, 2017. Springer International Publishing.

[41] Microsoft. Microsoft azure functions. Online. Accessed 2021-02-12. "https://azure.microsoft.com/en-us/services/functions/".

[42] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. 2018.

[43] Hamidreza Moradi, Wei Wang, Amanda Fernandez, and Dakai Zhu. upredict: A user-level profiler-based predictive framework in multi-tenant clouds. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 73–82. IEEE, 2020.

[44] I. Mytilinis, C. Bitsakos, K. Doka, I. Konstantinou, and N. Koziris. The vision of a heterogenerous scheduler. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 302–307, 2018.

[45] M. Najafi, K. Zhang, M. Sadoghi, and H. Jacobsen. Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1938–1948, 2017.

[46] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[47] Hani Nemati, Seyed Vahid Azhari, and Michel R Dagenais. Host hypervisor trace mining for virtual machine workload characterization. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 102–112. IEEE, 2019.

[48] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[49] openfaas. Openfaas watchdogs. Online. Accessed 2021-02-12. `https://github.com/openfaas/of-watchdog`.

[50] Tobias Pfandzelter and David Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. IEEE, 2020.

[51] George Plastiras, Maria Terzi, Christos Kyrkou, and Theocharis Theocharidcs. Edge intelligence: Challenges and opportunities of near-sensor machine learning applications. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–7. IEEE, 2018.

[52] Thomas B Preußer, Giulio Gambardella, Nicholas Fraser, and Michaela Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 833–838. IEEE, 2018.

[53] Thomas Rausch and Schahram Dustdar. Edge intelligence: The convergence of humans, things, and ai. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 86–96. IEEE, 2019.

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

[55] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge {AI}. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

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

[57] Thomas Rausch, Philipp Raith, Padmanabhan Pillai, and Schahram Dustdar. A system for operating energy-aware cloudlets. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 307–309, 2019.

[58] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259 – 271, 2021.

[59] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[60] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards network-aware resource provisioning in kubernetes for fog computing applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 351–359. IEEE, 2019.

[61] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[62] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[63] Mahadev Satyanarayanan and Nigel Davies. Augmenting cognition through edge computing. *Computer*, 52(7):37–46, 2019.

[64] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

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

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

[67] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 152–165, 2019.

[68] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge. Fogernetes: Deployment and management of fog computing applications. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018.

[69] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289 – 330, 2019.

[70] Xingyu Zhou, Robert Canady, Shunxing Bao, and Aniruddha Gokhale. Cost-effective hardware accelerator recommendation for edge computing. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[71] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.