MSc thesis project

Data Locality Aware Scheduling on a Serverless Edge Platform

Author

Wesley Seubring (s3529797)

Supervisors

Primary supervisor:
Prof. Dr. A. Lazovik

Secondary supervisor:
MSc. M. Hadadian Nejad Yousefi

External supervisor:
Dr. F. Blaauw

4th September 2021

university of groningen

Acknowledgements

First, I would like to thank my supervisors Alexander Lazovik, Mostafa Hadadian Nejad Yousefi and Frank Blaauw for their valuable guidance and feedback throughout this project.

Further, I would like to thank my friends for their support during this project, their feedback and talks have helped me throughout the execution and writing of this project.

Contents

Abstract

Serverless edge computing is a computing architecture that combines on-demand execution and scaling from serverless with the heterogeneous devices from an edge network, extending the platform outside the cloud. Serverless can be a general approach to efficiently utilise the available resources on an edge network without the complexity of orchestration. Serverless platforms abstract away from orchestration, providing (limited to) no control on the execution location of functions. This limitation brings concerns on privacy and bandwidth usage on a serverless edge system.

This work provides an overview of the serverless landscape and proposes a data locality aware scheduler for serverless edge platforms that restricts function execution to nodes within the network of the data. To validate the proposed scheduler and show the performance impact compared to the native Kubernetes scheduler, the scheduler is simulated in a data-intensive application. The results show that moving the execution close to the data reduces bandwidth (cost) and time spent on data movement. In the data-intensive application, the system can provide similar throughput and utilise fewer resources. The scheduler limits the number of resources suitable for scheduling significantly, the limitation causes increased standard deviation of latency over the functions.

The proposed method can be ported to serverless edge platforms utilising the metadata available in the edge network, providing developers with a function level control on the scheduling of functions, constraining the execution to the local network. Compared to existing work, this work provides a hard constraint on data-locality, rather than a soft constraint and is configurable on a function level instead of a system-wide configuration.

Chapter 1

Introduction

Serverless Edge computing is a relatively new computing architecture within the active field of Cloud computing [1]. It combines both serverless and edge computing creating new opportunities and challenges to be explored within the cloud landscape.

Edge computing is an adaptation of cloud computing that facilitates the process from cloud for IoT to distributed cloud on the edge [1]. It is used to support a heterogeneous network of IoT devices, in which edge computing directs computation away from the cloud server to the edge devices. Edge devices that are closer to the user or data, bring several benefits such as reduced latency, reduced bandwidth and fast processing [2, 3].

Serverless platforms manage orchestration and scaling for the application, simplifying the process of deploying code greatly. Serverless computing is upcomming and here to stay and will have an important role in the future of computing [4]. The simplification of serverless comes with the cost of having (limited to) no control of where functions run, which results in an increased risk for privacy, especially in the field of serverless edge computing.

Serverless edge computing is a new concept in cloud computing that combines serverless computing with edge computing [1]. One of the challenges is combining the heterogeneous nature of edge computing with the agnostic execution of serverless functions. To make efficient use of the edge resources, the edge computing platform needs to consider extra information like location, latency, available hardware and data movement, whilst keeping the on-demand execution and scaling of serverless functions and simplify the orchestration. Especially data-intensive applications can benefit from moving the computation close to the data (i.e. on the edge devices) instead of moving the data to the computation [5, 6].

Location of computation is of importance for many companies for factors like bandwidth cost, privacy and law and regulations. There is a growing privacy concern, with new laws being introduced to improve the privacy of users and their personal data, such as the GDPR [7]. These privacy concerns also extend to data-intensive applications [8].

To ensure that sensitive data is not transferred over the network, the scheduling of functions is restricted to enforce execution within the local network of the data. Data movement can be costly in time and bandwidth, especially when multiple nodes are traversed. Limiting placement to the local network could reduce latency and bandwidth cost as the computation is moved to the data.

Currently, there is no data locality aware scheduler for serverless edge that enforces scheduling within the network of the data. This makes serverless edge not suitable for applications that require a level of control on data movement. This work proposes an online data locality aware scheduler that provides this control, by using additional metadata collected from the edge network. The scheduler makes serverless edge available for applications that need control on movement with concerns like cost or privacy. In other applications, the scheduler can also be used to reduce bandwidth and latency, especially for functions with high data requirements.

1.1 Research questions

The goal of this thesis is to design, implement and evaluate a scheduler that supports the enforcement of local execution using a data locality aware scheduler on a serverless edge platform. To ensure this goal, three research questions are answered each extending on the previous research question. The next paragraphs will elaborate on the research questions.

What is the current landscape of serverless computing?

Serverless computing is a growing field, with new literature and platforms being released. An overview of serverless platforms and frameworks with insight into features, differences and limitations to highlight the current state of the art and their ideas in the serverless field.

The literature has several overviews of serverless platforms [9, 10], these overviews mostly focus on the commercial hosted platforms and a few of the frameworks from other literature.

This work performs an initial survey on commercial hosted platforms, self-hosted platforms and frameworks from the literature. The survey will present an overview and a list of categorised features found in the platforms. The differentiation between the features uses the following categories (a) *core* features, which need to be present in a serverless system; (b) *standard* features, that make systems suitable for general use (e.g. fault tolerance), in the view of this work, these are not considered to be essential to call a system serverless; (c) *extra* features, are all other features found that specialise a serverless platform.

How can data-locality aware scheduling be achieved in a serverless edge platform?

Insight in the state of the art and ongoing works showed an open problem for serverless edge platforms. The problem involves scheduling tasks within the network of the data. Scheduling function execution in the network of the data limits data movement and can ensure the privacy of the data.

Raush et al. [6] proposed soft constraints using priority functions for efficient scheduling on the serverless edge system. These priorities include data locality based on download time, but do not enforce the privacy of the data. The soft constraints are also performed on all functions in the serverless platform, which does not allow for fine-grained control and insight knowledge of the developer. Other suggested functions aim to reduce bandwidth cost or startup latency.

A solution is proposed that extends the metadata in the system and allows functions to be individually scheduled with or without the data-locality aware scheduling.

What is the performance impacted by data-locality aware scheduling?

The proposed approach of the previous research question limits the scheduling of the platform. A case study is performed between the Kubernetes[1] scheduler and the data-aware scheduler. In the case study, a benchmark simulation is run using faas-sim[2] for the simulation of the serverless edge platform, which is used to compare the difference in metrics (e.g. latency, throughput, utilisation, etc.) between the Kubernetes baseline and the data-locality aware scheduler.

1.2 Outline

The remainder of the work is structured as follows. Chapter 2 Elaborates on the relative work and the background concepts for this work. In Chapter 3 the relevant tooling and the approach are explained. Next in Chapter 4, an overview is created of the current state of the art in serverless and a categorical list of features of a serverless platform is presented.

A proposal for a data-locality aware scheduler is done in Chapter 5 and Chapter 6 describes

[1]https://kubernetes.io/
[2]https://github.com/edgerun/faas-sim

the experiment to validate the impact on the metrics of the scheduler. Chapter 7 describes the results of the experiment. Next Chapter 8 discusses the points of interest of the results and how they are related to data-locality aware scheduling. Finally, in Chapter 9 this work is concluded and a summary is given.

Chapter 2

Background and Related Work

In 2014 the first commercial serverless platform AWS Lambda was released by Amazon, after two years other serverless platforms start to emerge [11]. All commercial platforms support small stateless short-running functions that are scaled on demand by the platform and billed using a pay-for-usage model. This chapter will further elaborate on serverless and its related topics and will consider relevant works.

2.0.1 Data-intensive applications

Data-intensive applications are applications for the field of data-intensive science (data science as short), which is considered the fourth paradigm of science [3]. In short, a data-intensive application is an application that processes big data. First, we need to consider big data. Much like the definition of big data, there is no fixed number that defines when data is Big data([3] examples are given in the range of terabytes and petabytes, back in 2009), this is dependent on the current progress in the field and the computation power of all systems. More agreed on are the characteristics of these systems. Big data has the four v's: volume, velocity, variety and veracity [2].

Data-intensive applications share the following characteristics:

- The Data size of Big data (as suggested: terabytes or petabytes)
- Limited by data I/O (instead of CPU for computational intensive application).
- Data movement needs to be limited to get good performance.

2.0.2 Data science pipeline

A data science pipeline handles the workflows of a data science system. The workflow of such a system is heterogeneous and can be complex, as a workflow consists of multiple distinct stages of data handling and processing. A workflow generally contains the following steps: pre-processing, training, validation, hyperparameter-tuning and classification. Each of the steps can have specific implementations with its own strengths and limitations.

Looking at the DS pipeline it becomes clear that there are two different pipelines. The first often contains preprocessing, training, tuning, and validation, which is often data-intensive and requires many computation resources and stateful management. The second pipeline is the classification of data using a model, this is less intensive and can be run as a stateless action for input data to classification. In this work, the training pipeline consists of two functions preprocessing and training. The second pipeline in this work is a single function that serves the model to classifie the input.

2.0.3 Edge computing

Edge computing is an adaptation of cloud computing, it extends on cloud computing by directing computational data and services away from the cloud server to edge devices on the edge of a network [12]. In our case the edge devices are servers, actuators and sensors on location closer to the

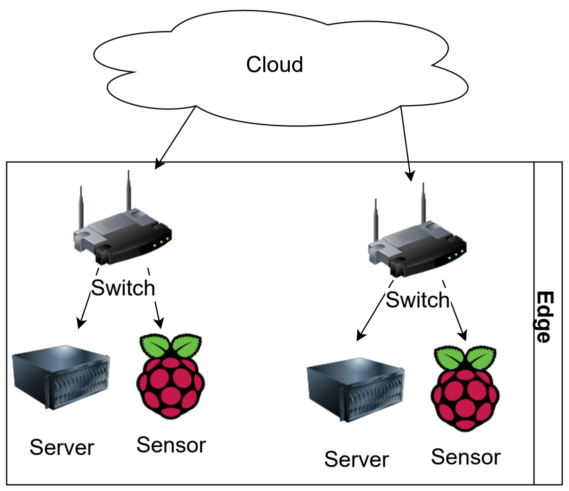

Figure 2.1: Example of edge devices in a edge network

data. An example is shown in Figure 2.1, it shows two networks connected to the cloud.

The edge devices have computational resources available and are located closer to the user. With edge computing, with these resource work can be offloaded from the cloud. Using edge computing has several advantages, reduced latency, reduced bandwidth and fast processing [12, 13]. An example of this would be encoding a video on your mobile device before uploading it to the cloud, this reduces the bandwidth usage and reduces the resources used in the cloud.

Edge computing shares many similarities with cloud computing, as it is an extension that aims to reduce latency by bringing services closer to the user and minimising the load of the cloud. Edge computing also differs from cloud computing in several aspects. The main difference is the location of services, in cloud computing, all services are available via the internet in the cloud and in edge computing, there are also services available on edge networks. Also, edge computing is location-aware and uses a distributed model rather than a centralised model used by the cloud [12].

2.0.4 Serverless computing

Serverless computing or Function as a Service (FaaS) are upcoming computing architectures that allow developers for on-demand execution and scaling functionality defined in cloud functions. This type of architecture allows developers to deploy code, without having to deal with the orchestration. This section gives our definition of serverless computing and elaborates on several key aspects of serverless computing.

2.0.5 Definition

There are many definitions of serverless, most fit only the hosted commercial platforms. To get a good definition of serverless, existing definitions are collected and compared to find the core elements of serverless computing. The following paragraphs give an overview of the existing definition, discussed the likeliness and differences and forms a definition of serverless for this work.

In their white paper [10] the Cloud Native Computing Federation (**CNCF**) defined serverless computing as follows:

> *"Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment"*

It is also mentioned that serverless computing has two involved parties. The first being the developers that develop and upload the function, to the serverless platform with the idea that there is no server and their code is not always running. The second party is the Provider, it deploys the serverless platform for an external or internal customer. This means that a self-hosted system still can be considered serverless, with a clear division between the parties.

In an article on their view on serverless UC Berkley [9], defines serverless computing as follows:

> *"In our definition, for a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage."*

Amazon owns AWS Lambda[1], a commercial FaaS platform. They define serverless computing [14] as follows :

> *"With serverless computing, infrastructure management tasks like capacity provisioning and patching are handled by AWS, so you can focus on only writing code that serves your customers. Serverless services like AWS Lambda come with automatic scaling, built-in high availability, and a pay-for-value billing model."*

In an article on the formal foundation of serverless Jangda et al. [15] defined serverless computing as follows:

> *"Serverless computing, also known as functions as a service, is a new approach to cloud computing that allows programmers to run event-driven functions in the cloud without the need to manage resource allocation or configure the runtime environment."*

They also mention serverless computing to remove orchestration from the developers and handle scaling and payment depending on the required and used resources.

In their paper [16] on the view of FaaS and Serverless architecture, The SPEC cloud group defines serverless as follows:

> *"Specifically, we see serverless cloud architectures as defined by three key characteristics: (i)* ***Granular billing****: the user of a serverless model is charged only when the application is actually executing; (ii) (Almost)* ***no operational logic****: operational logic, such as resource management and auto-scaling, is delegated to the infrastructure, making those concerns of the infrastructure operator; and (iii)* ***Event-Driven****: interactions with serverless applications are designed to be short-lived, allowing the infrastructure to deploy serverless applications to respond to events, when needed."*

The agreement between the definitions is automatic scaling, automatic provisioning and infrastructure management and pay-per-value. With these aspects, serverless computing abstracts away the idea of a server for the developer, as the frameworks handle scaling and orchestration the developer only needs to provide functions and pay the bill based on the used execution time (with no concern of a server).

Looking at all these definitions, this work defines serverless with more aspects. In our definition, a serverless system consists of two parties the developer and the platform. The developer uploads functions to the platform, with no need for orchestration. The platform provides, resources pooling, Elasticity, Optimises resource utilisation and is event-driven.

The definition provides a baseline definition of serverless which we can compare the definition to our own work and existing work, and makes it that decisions can be made on a more formal definition rather than references to previous work.

The main difference is that pay-per-usage is not considered a defining property of serverless computing, as it excludes self-hosted platforms. To fill the gap we introduced Resource utilisation to the definition, as serverless computing tries to make good use of available resources and reducing idle workload, with on-demand execution and scaling of functions. On a hosted platform this results in a pay-per-usage cost and self-hosted results in better utilisation of the available resources (and indirectly the cost). Further Event-driven was added as this is the architecture behind running the cloud functions provided. Finally, the cloud definition of NIST [17] was used to define several properties in the definition.

[1] `https://aws.amazon.com/lambda/`

Table 2.1: Overview of platforms against definition

	Resource pooling	Rapid elasticity	Resource Utilisation	Event-driven	No orchestration
AWS Lambda	✓	✓	✓	✓	✓
IBM functions[3]	✓	✓	✓	✓	✓
Cloud functions[4]	✓	✓	✓	✓	✓
Cirrus	✓	✓	✓	✓	✓
Nuclio[5]	✓	✓	✓	✓	∼
Towards serverless	✓	✓	✓	✓	∼
Crucial	✓	✓	✓	✓	✓
Kubernetes	✓	✓	✓	∼	X
KEBA	✓	✓	✓	✓	X

A checklist is created with the aspect defined by our definition. The checklist items where checked against the commercial serverless providers, serverless frameworks from the literature and a few non-serverless frameworks (e.g. KEDA[2] and Kubernetes). Table 2.1 shows the results of this checklist. Th results shown that all hosted platforms and frameworks that extend on these platforms are serverless, as by the given definition. Some of the custom platforms add orchestration configuration to the functions. This is not required orchestration, but can be used to define constraints on the execution of the function (e.g. run with GPU, or with a specified network).

2.0.6 Serverless functions

Serverless functions are small stateless pieces of code with a single responsibility, that can be executed on demand. The functions have several main characteristics. The first characteristic is ***no operational logic***, this should all be handled by the platform. The second characteristic is that the functions are ***context-agnostic***, as the functions are unaware of where and why they are executed [16]. On the side of the developer, some characteristics need to be considered. Firstly, serverless functions have an ***ephemeral state***. All local data is lost when a function goes idle, therefore the data needs to be serialised and be written to storage over the network. Also, the functions have ***implicit parallel execution***, as there are no guarantees on where each function is executed (even for events coming from the same source). Finally, the functions are guaranteed of ***at-least-once execution***, this can cause functions to be re-invoked and can lead to multiple executions of the same event [15]. The platforms do not resolve these aspects and they are the responsibility of the developers, they need to be aware of handling the situations correctly.

2.0.7 Function replica

A function replica is a container with an image of the environment to run a specific function. The function replica has two states idle or busy, an idle instance is ready to execute a new invocation of its cloud function and when busy it is currently running a cloud function.

The number of function replicas of a specific function on the serverless platform can vary and can even scale down to zero. The platform will handle the scaling of the instances to match demand. When a function replica is started it will be an instance with the required run-time, empty local storage and default runtime environment parameters. Running a function can alter the runtime environment and local storage, to mitigate this the developer of the cloud function should always assume that there is an ephemeral state.

2.0.8 Formal definition of serverless

A formal definition of serverless gives a clear view of what is serverless by explaining the model of a server platform. It reduces the ambiguity and provides the base for a clear scope of the project. The formal definition of the model will be used to define the features of server computing and its

[2] https://keda.sh/

extensions.

The paper by Jangda et al. [15] defined an operational semantics of serverless platforms called λ_λ. Their model λ_λ is divided into two parts, a model of the serverless function replicas and the serverless platform. The λ_λ model defines that functions instances have the following functions:

- A function that initialises the state of the function, where the state is the state of the runtime enviroment VM and the initial runtime state (e.g. JavaScript VM and the initial JavaScript heap).
- A function that receives a request from the serverless platform (e.g. The serverless function is triggered on the platform and the platform starts the execution on one or more function replicas)
- A function that takes internal steps within the function replica, which can produce a call to the platform (Initially only the return value of the function, but this can be extended)

The serverless platform λ_λ keeps track of the collection of function replicas and if they are idle or running and pending requests and responses [15]. On top of this some rules define the subtleties of serverless execution:

- Instance launches of the function replicas are not observable. This is necessary since there can be multiple function replicas started for a single event.
- State reuse during warm starts. A warm instance can be reused for another invocation to speed up the initiation, but the state may be different from the initial state. This means the serverless function needs to assume to have **ephemeral state**
- Function replica termination is not observable. This is due to **implicit parallel** execution and the fault tolerance, allowing multiple instances to run on an event.
- Function replica termination can happen at any time. This can be a failure or to make an instance available for new events.
- Function execution on the function replicas can start any time. Due to cold and warm starts and the lack of observation on the function process, there can be no time guarantees.
- Single response per request. Each event is **executed at least once**, but this can be more than once, therefor the platform needs to ensure only one response is given.

To create a safe serverless function the developer has to consider the following properties of a serverless function:

- There is implicit parallel execution therefore needs to consider an ephemeral state.
- The function is executed at least once therefore needs to be idempotent.

2.0.9 Serverless edge computing

Serverless edge computing is a new paradigm combining the simplicity of serverless with the resources of edge computing. Serverless brings beneficial features to edge computing, such as scale-to-zero, fine-grained scaling (i.e at function level), and event-driven. These match well with the energy-awareness of IoT devices, limited resources of IoT devices and many IoT applications are event-driven [1]. A serverless edge computing platform provides efficient event processing the edge and cloud, the IoT devices will need to follow the FaaS principles and the platform manages the life cycles of an event. It should be noted that serverless edge computing is still not fully adopted in edge computing, as IoT only makes 10% of the serverless use cases [18] and is still in infancy [1]. There are several opportunities and challenges in the field.

2.1 Related work

There is a wide range of literature on serverless, exploring and improving the field. In the following paragraphs, the work related to this work is provided and the relevance and similarities are discussed.

2.1.1 PyWren

Existing serverless platforms have a high entry-level for data scientists and scientific computing users. The elasticity and scale of serverless are out of reach for these users [4]. PyWren [4] is a serverless development platform developed in Python to accommodate ease in development and simplify the use of stateless functions on the AWS Lambda platform. To achieve this PyWren provides solutions to the resource constraints of serverless platforms, allowing storage access (AWS S3) and simplified definition and deployment of map-reduce tasks on the AWS Lambda platform.

PyWren is one of the earlier works that try to simplify the development process of serverless platforms and make them better accessible for developers. PyWren and this work both aim to make serverless more available, whereas PyWren provides a simplified solution to create serverless applications on AWS, this work increases control on schedule to make serverless suitable for more applications and gain some benefits.

2.1.2 Cirrus

The scale and complexity of machine learning workflows make it hard to provision and manage resources. To relieve this burden, Carreira et al. [5, 19] propose Cirrus is a machine learning framework that automates end-to-end management of resources efficiently by leveraging the serverless infrastructure. Cirrus utilises the simplicity of serverless scaling and infrastructure (AWS Lambda) to minimise user effort [5].

Cirrus provides an interactive dashboard to track and manage at a higher application level for the hyperparameter optimisation stage. The framework offers an API for machine learning tasks. The API offers, pre-processing, training and hyperparameter-tuning. The framework has several additional optimisations compared to PyWren. Cirrus, makes uses **data pre-fetching** to reduce idle-time from I/O, **mini-data batches** to run the job efficient in parallel on **lightweight workers** and **custom storage** to reduce latency. The aforementioned optimisations make Cirrus up to 100x faster compared to Pywren [19] and 3.75x faster than an implementation in the traditional ML framework Bösen [20] on a Sparse Logistic Regression job. The implementation allows for stateful management of the machine learning flows, with the intermediate steps split up into many small functions and using a distributed storage as intermediate between functions as no direct communication is available.

The method proposed by this work is similar to Cirrus, both consider data movement of serverless functions. Cirrus aims to improve the time lost by moving the data, with custom storage and batch-fetching, this work looks into moving the computation of the data. Another difference is the scope of functions, Cirrus sees individual steps like pre-processing as a stateful process with many parallel serverless stateful functions, where the functions are constrained by the limitations of the AWS platform (e.g. 15min run-time and limited memory per function). In this work the functions like pre-processing is a single larger function, focusing on the placement of the functions rather than maximum speed-up. This work does not concern the run-time and memory limitations as we assume a Kubernetes/OpenFaas[6] environment. The pre-fetching and mini-bathes could be an interesting extension to this work when combined with a hybrid environment and the data locality constraint.

2.1.3 CRUCIAL

Existing commercial serverless platforms do not support peer to peer communication between functions and the storage is limited to the functions. Applications that require fine-grained support for mutable state and synchronisation, such as machine learning are hard to build on the disaggregation of storage in FaaS [21]. The work of Barcelone-pons et al. proposes CRUCIAL [21], a system to program highly concurrent stateful applications with serverless architectures.

CRUCIAL programming model keeps the simplicity of FaaS and allow porting of multi-threaded algorithms to the serverless environment. The main idea of CRUCIAL is that FaaS resembles

[6]https://www.openfaas.com/

concurrent programming at the scale of a data centre. With this, a distributed shared memory is introduced in CRUCIAL. Compared to an equivalent Spark cluster, CRUCIAL performs in a range from comparable to superior depending on the use case [21].

The similarity between CRUCIAL and this work is that both look at the impact of data movement in a serverless environment. The main difference between the two is that CRUCIAL uses a custom shared memory layer to reduce latency and maintain and share state between functions without intermediate storage, whereas this work considers moving computation to the data to reduce latency but mostly keep data within its local network.

2.1.4 Skippy

Data-intensive applications have a high workload, this in combination with the heterogeneity of devices on an edge system makes it challenging to manage data-intensive applications on edge systems. Serverless computing is a compelling solution for systems of this complexity, but Raush et al. [6] found several limiting factors for their use in the edge setting. Their work proposes a scheduler called Skippy suitable for edge infrastructures. The scheduler can make weighted heuristic trade-offs on aspects of the edge system.

Skippy is built on top of the existing platforms Kubernetes and OpenFaaS. Their work introduces a daemon that runs alongside the nodes and provides metadata to the platform. This metadata includes a bandwidth graph and available hardware resources on the nodes of the system. With the extra metadata provided Skippy introduces the following priorities for scheduling:'

- Latency aware image locality scheduling (prefers, nodes that allow for quick deployment of the images).
- Data locality aware (prefers nodes with a shorter download time of the data)
- Capability priority (prefers nodes with specialised hardware)
- Locality type priority (prefers nodes that are on the edge of the cloud)

The priorities can be weighted and in their work the weights are tuned a certain goal (e.g. reduce bandwidth or reduced latency), using an NSGA-II genetic algorithm. The weights and priorities are validated using the Faas-sim simulator, resulting in a scheduler that supports serverless computing on an edge system.

Comparing the most relevant work of Raush et al. [6] with this work, many similarities can be seen. As both extract metadata from the network for scheduling, Skippy also considers data location as one of the scheduling priorities. The main differences are that this work focuses on the hard constrain of computing location within the sub-network of the data, whereas the data-locality priority of Skippy is a soft constraint and considers bandwidth speed rather than data location. Another difference is that the enforcement of the hard constraint can be set on a function level, allowing developers with fine control over which function to enforce this policy. To conclude Skippy focuses on the overall cost, bandwidth or latency, our work provides a way for developers to ensure privacy and reduce bandwidth for desired functions.

Chapter 3

Methodology

Scheduling in a serverless environment handles placement of functions in an efficient way to utilise the available resources. Scheduling on a serverless edge is harder as the complexity of scheduling increases with new parameters to take into account.

As part of the scheduler, several tools and frameworks are used. This chapter will first explain the method of the initial survey, the used serverless framework and orchestration framework, Skippy scheduler as part are relevant to the proposed simulator, the simulations tools that allow execution of experiments of the serverless framework and finally the experiment.

3.1 Initial Survey of Existing Serverless Platforms

The survey provides an overview of the current state of the commercial platforms and the state of the art, including their features, limitations and benefits.

Existing work provides an overview of some of the existing work [9, 22], but they focus mostly on the commercial serverless platforms. The following paragraphs will elaborate on the process of the initial survey.

First, literature is collected to identify and compare the existing serverless platforms and frameworks. Existing platforms were searched using the following keywords including but not limited to:

- Serverless
- Serverless edge computing
- Hybrid cloud
- Edge computing
- Machine learning
- Orchestration
- Workflow
- Data science

These keywords were run through the following search engines:

- Google Scholar
- Google search

In addition, the snowball method is used to find more related literature. With the snowball method relevant citations are collected from the the initially literature's.

The relevant literature is described and features are extracted and categorised. The differentiation between the features is done using the following categories:

Core features are defined in the formal definition of serverless [15] and need to be present in a platform to consider it serverless.

Standard features are features that are available in all the commercial platforms, that make a serverless platform more suitable for general use.

Extra features are features found in the literature that improve the serverless framework in general or for specific use cases.

3.2 Kubernetes

Kubernetes is an open-source container orchestration system for deployment automation, scaling and container management. Kubernetes is a container run-time, that can run on-premise, hybrid or public cloud. A common use is the deployment of micro-service-based applications and more recently and increasingly FaaS platforms [23].

Figure 3.1 shows a architecture of a Kubernetes cluster. It shows the components of the control plane of Kubernetes. The *kube-apiserver* functions as the front-end of the control panel and exposes the Kubernetes API. The *etcd* component is a consistent high-available key-value storage for the cluster. The *kube-controller-manager* component is responsible for running the controllers for the nodes, jobs, endpoints and Tokens. The *kube-scheduler* component watches for newly created pods with no assignment and schedules them on the best fitting node using hard and soft constraints. Each node also contains components of Kubernetes, these being *kubelet* and *k-proxy*. They are responsible for making sure the Kubernetes pods run correctly and proxy for requests within the Kubernetes, respectively.

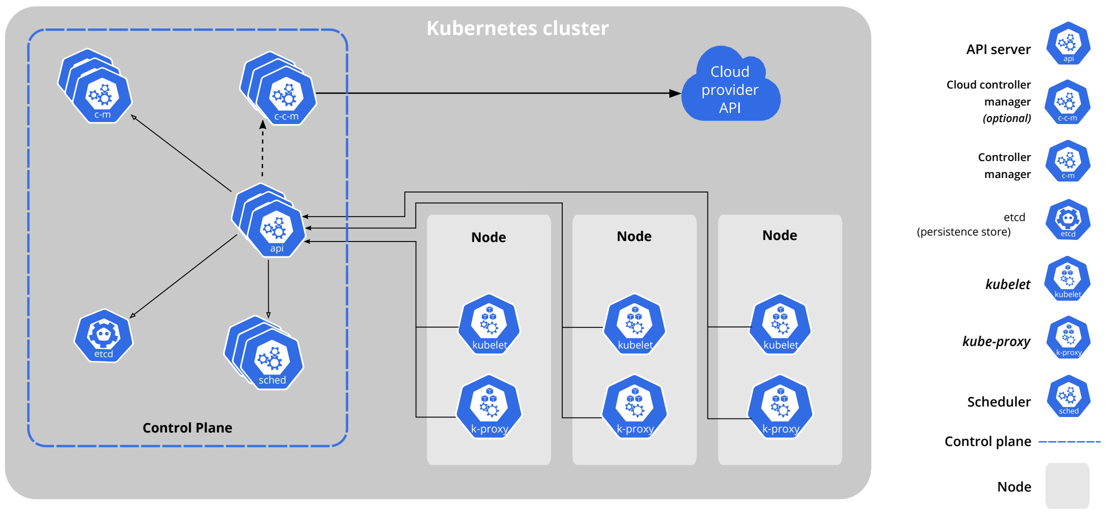

Figure 3.1: Architecture of a Kubernetes cluster [24]

3.2.1 Kubernetes Scheduler

The kube-scheduler is the most relevant part of Kubernetes. The Kubernetes scheduler is an online scheduler that schedules new pods on creation, having no knowledge of future arrivals. It applies a greedy multi categorical decision making (MCDM) strategy which considers both hard and soft constraints. The hard constraints are implemented using *predicate functions*, these functions evaluate the nodes of the cluster and eliminate nodes that do not match the criteria. The soft constraints are priority functions that rank the remaining nodes, the highest-scoring node is then selected for deployment of the pod. The unweighted scheduling is formalised as follows (based on the weighted formalisation of Raush et al. [6]):

$$\text{schedule } (p) = \underset{n \in N}{\arg\max}\, \text{score}(p,n) : \sum_{i=0}^{|S|} S_i(p,n) \tag{3.1}$$

Where S is the set of priority functions $S \in S : P \times N \rightarrow \mathbb{R}$, where P are the domain of the pods and N is the domain of nodes including the metadata. The function performs a mapping from a pod p to node n and does this by selecting the highest scoring node.

The default predicates and priorities of Kubernetes do allow for the data locality aware scheduling, in the remainder of this work we look into a proposed predicate, with corresponding metadata to enforce data locality aware scheduling.

3.3 OpenFaaS

OpenFaaS is an open-source serverless framework built for Kubernetes, it handles both execution and deployment. The functions of OpenFaaS are packaged in Docker[1] containers, these function replicas are deployed via Kubernetes pods alongside a watchdog. The watchdog handles HTTP requests to innovate the execution of the corresponding function. OpenFaaS allows for a scale to zero policy, which removes function replica's after a short idle time. This is used for functions with a low arrival rate, allowing the system to release resources for other deployments.

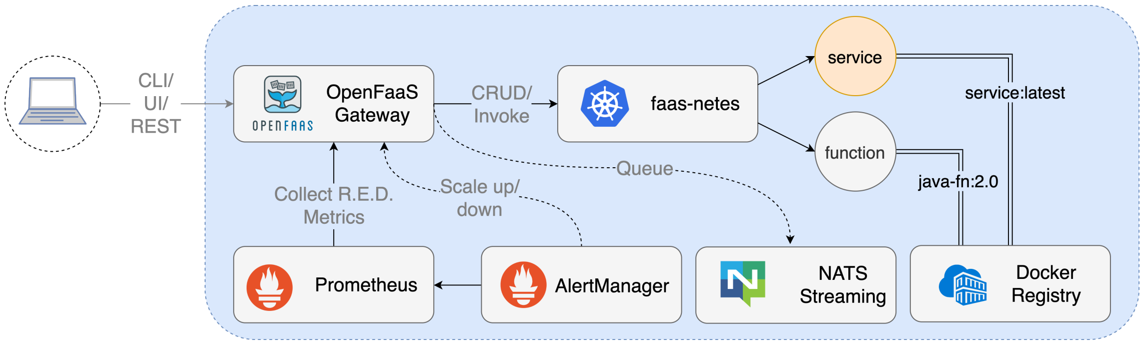

Figure 3.2: Conceptual overview OpenFaaS [25]

Figure 3.2 shows the conceptual workflow of OpenFaaS. The figure shows how OpenFaaS uses Prometheus[2] for system metrics and alerting, NATS[3] for queuing of invocations, Kubernetes for the deployment of the functions and services and Docker for the corresponding latest images.

3.4 Simulator: FaaS-sim

The experiments are run in the FaaS simulator developed by Raush et al. [6, 23]. The simulator emulates an edge environment and simulates the behaviour of a serverless edge platform, which allows for experiments on serverless edge environments. The simulation is still very new and has limited documentation and community, but is used for simulation in a few papers [6, 23, 26, 27].

Figure 3.3 shows an overview of faas-sim. Faas-Sim has consist of multiple components. Ether[4] is used to define the topology of the network and its bandwidth speeds. Simpy[5] the simulation framework used. Simpy provides traceable simulation of processes and asynchronous networking. Furthermore, users can define their own benchmarks and typologies. The user is also able to inject

[1] https://www.docker.com/
[2] https://prometheus.io/
[3] https://nats.io/
[4] https://github.com/edgerun/ether
[5] https://simpy.readthedocs.io/en/latest/

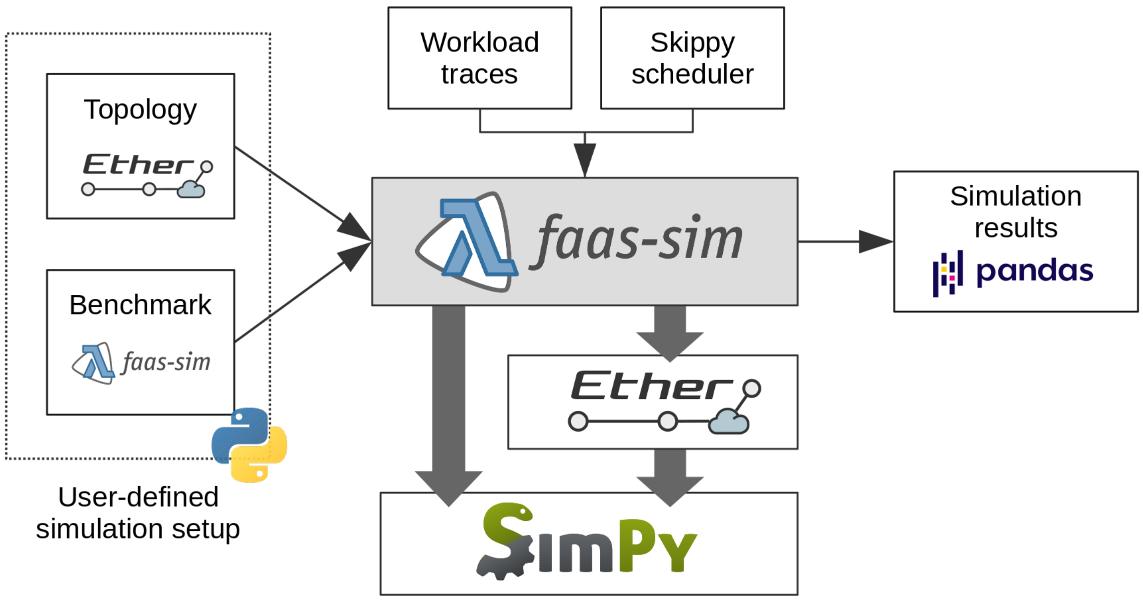

Figure 3.3: Overview of faas-sim [28]

custom elements in the core of faas-sim (e.g. scheduler or the function simulation). The core of the simulator and the Kubernetes scheduler is based on the existing open-source projects OpenFaaS and Kubernetes, to best mimic the behaviour of real FaaS system components (e.g. gateway, scheduler and scaling).

The metrics like function execution time are simulated. To mimic the behaviour the simulator estimates these values by sampling a user provided distribution. With these distributions, the simulator can quickly simulate the behaviour of execution and startup by sampling values and moving the simulation forward in time by the sampled duration.

The result of a simulation is a comprehensive set of logs that gives insight into the simulation behaviour of the serverless edge system. This includes execution time, waiting time, network traffic, utilisation and scaling.

3.5 Experiment

Multiple experiments are performed to validate the proposed scheduling method and investigate the impact on the metrics of the system. The experiments uses the FaaS-sim to simulate a serverless edge platform using the proposed scheduler. The results will be visualised and the significance of the measurements is validated using the t-test.

The experiment runs two scenario's, each consisting of a range of parameter configurations. The configurations are run ten times with different seeds, to show the consistency of the runs.

The workload applied is based on the machine learning workload defined in the work of Raush et al. [6]. A detailed representation of the experiment will be given in chapter 6.

3.6 Metadata

Serverless platforms use additional metadata of the system and its nodes to efficiently utilise an edge system [6]. Skippy [23] get its metadata using a daemon that is deployed along with each node. The metadata provided by Skippy is a network topology with bandwidth information, that stores the limiting download speed between (indirect) connected nodes and the available resources of the machines including specialised hardware.

Additional metadata is also required for data-locality aware scheduling of the functions. Nodes of the system require metadata, defining which data storages are available in the premise.

3.6.1 Data Locality

Data locality meta-data is required to enable a hard constraint on data locality. The locality is computed using the topology and a definition of where the local network ends in relation to the rest of the network.

First, the topology of the system is collected using the Skippy daemon, by extracting the topology from its bandwidth graph. Next, for this work switches are used at the edge between the nodes and the cloud. Therefore, the switch can be used to define the exit of the sub-network. This allows the nodes to be tagged with the storage nodes in their premise, defining their data locality.

Chapter 4

Existing frameworks

The first serverless platform was introduced by Amazon in 2014, many platforms and literature is added since. An overview of the current state of the art provides insight into the current developments and open issues, helping identify open research topics in the field. Furthermore, this chapter focuses on the unique features and limitations of the frameworks.

The structure of this chapter is as follows. First, an overview of serverless platforms is given. Next, the commercial platforms, self-hosted platforms and platform extensions are discussed. Next, the features of the platforms are categorised. Finally, a summary is given of the found results.

4.1 Landscape

Serverless computing is an upcoming topic, many platforms and relevant scientific research have been released in the last years. In this section, an overview of the landscape of serverless computing is given. This work will be limited to the frameworks and platforms:

- AWS Lambda
- Google Cloud Functions[1]
- IBM Cloud [2]
- Microsoft Azure cloud functions[3]
- Apache OpenWhisk[4]
- OpenFaaS
- Nuclio
- Cirrus [5]
- Skippy [23, 6]
- CRUCIAL [21]

The list contains three types of frameworks. First, the commercially hosted FaaS frameworks like AWS Lambda and Google Cloud Functions. Next, self-hosted FaaS frameworks like OpenFaaS, OpenWhisk and Nuclio. Finally, extension on the existing frameworks, like faas-sim and Cirrus that both extend on AWS Lambda. The following sections are split on type and the platform and framework are discussed individually.

[1]https://cloud.google.com/functions
[2]https://cloud.ibm.com/functions/
[3]https://azure.microsoft.com/nl-nl/services/functions/
[4]https://openwhisk.apache.org/

4.1.1 Hosted FaaS platforms

The hosted serverless platforms provide a pay per use serverless platform, where the user is billed for computation time and memory usage of their functions. Hosted platforms handle all infrastructure, which reduces initial cost and or setup complexity. A possible risk, however, is that their is very limited to no control over the location of execution (e.g. AWS lambda allows region specification) and thus data is moved. Generally, the hosted platforms focus on small short running (e.g. maximum of 15 minutes) tasks, with limited memory available (e.g. maximum of 3Gb).

The table Table 4.1 shows an overview of commercial FaaS platforms. Each platform is described in a short section.

Table 4.1: Overview of hosted FaaS platforms

	AWS Lambda	***Google functions***	***IBM functions***	***Azure Functions***
Maximum Run time	15 minutes	9 minutes	10 minutes	10 minutes (default 5 minutes)
Maximum memory size	3Gb	2Gb	2Gb	1.5Gb
delta Accounting Unit	1ms	100ms	100ms	1ms (min 100ms)
Price per accounting unit(512Mb)	$8.30e-9	$9.25e-7	$8.50e-7	$8.00e-9
Price API	$0.2 per 1M requests	$0.4 per 1M request	Free	$0.2 per 1M requests
Function composition	AWS Step functions	GCP Workflows	IBM Compose	Azure DF
Supported languages	Custom runtime support	Node.js, python, Go, Java, .Net	NodeJs, Swift, Java, Python, Ruby, Go, PHP, .Net, Docker	.Net, Js, F#, Java, Powershell core, Python, Typescript

4.1.1.1 AWS Lambda

AWS Lambda currently is the most popular serverless platform with being used in 80% of the serverless use cases [11]. AWS Lambda was the first serverless platform to release in 2014, two year ahead of competitors as Google functions, Azure and IBM.

AWS Lambda supports the longest-running functions at 15 minutes and the highest memory usage of 3Gb among the commercial platforms. Another interesting aspect of AWS Lambda is the use of Firecracker[5] microvm's, which is an open-source technology developed at Amazon that provides lightweight, secure and isolated VM for container or function services. The microvm's allow for the support of custom runtimes in the AWS lambda platform.

4.1.1.2 Google functions

Google functions is Google's implementation of a serverless platform. Google functions is released open source[6] allowing deployment of self-hosted platforms as well as creating custom function frameworks to support other programming languages, or use Docker containers to run other environments on their platform.

[5] https://firecracker-microvm.github.io/
[6] https://github.com/GoogleCloudPlatform/functions-framework-python

Table 4.2: Overview of self-hosted FaaS platforms

	OpenWhisk	***OpenFaaS***	***Nuclio***
Runtime limit	5 minutes	Configurable	Unlimited
Memory limit	512Mb	Configurable	Unlimited
Maximum code size	48Mb	-	unlimited
Maximum invocation	120 per minute	-	-
Maximum triggers	60 per minute	-	-
Supported languages:	.Net, JS, Java, Go, PHP, Python, ruby, swift, docker	Python, Ruby, C#, Node.js, Java, Go, any Docker image	Go/C (native), Python/Java/Shell (via shmem)
Auto Scaling	Yes, always back to zero on idle	Yes, configurable ranges	Yes (mutiple containers and within container), configurable, limited zero scaling
Supported Orchestrators	Kubernetes, Standalone, Docker	Docker Swarm, Kubernetes, any other external provider plugin - in progress - ECS, Hashicorp Nomad, ACS (via Swarm + K8s), Hyper.sh, Rancher Cattle, Cloud Foundry (via bosh), k3b	Docker, Kubernetes, Single binary
GPU- support	No	Yes	Yes

Google functions provides a general serverless platform for small short tasks, via a pay-per-use model.

4.1.1.3 IBM Functions

IBM functions is the serverless platform created by IBM. Next to the commercial platform, IBM released an open-source version of their serverless platform called OpenWhisk. OpenWhisk is not the same as IBM functions, but it is based on the same work.

In this list of hosted serverless providers, IBM functions is the only platform that does not charge for the request of the API gateway, which is beneficial for an application with a huge amount of short-running functions.

4.1.1.4 Azure Functions

Azure function is Microsoft's commercial serverless platform. The platform is open-source[7], supporting deployment on private hardware. Azure functions has a subscription that offers a predictable cost per hour but limits the scaling. Also, Azure functions does not bill memory cost twice when the functions share an instance, which can reduce cost [29].

4.1.2 Self-hosted FaaS platforms

Self-hosted serverless platforms are run on your infrastructure. This can be a cluster on a cloud provider, private hardware or an on-premise cluster. Hosting the application removes the pay-per-use aspect of serverless since the infrastructure is payed separately. This can limit the scaling of the system but can be used to gain better resource utilisation and simplified development and deployment for the developers, once the platform is properly set up [15].

[7]https://github.com/Azure/Azure-Functions

Table 4.2 gives an overview of the platforms. Next, the specific platforms are elaborated in more detail.

4.1.2.1 OpenWhisk

OpenWhisk is an open-source distributed serverless framework. It is developed in house at IBM as their serverless cloud option. Later IBM open-sourced the OpenWhisk project. OpenWhisk orchestrates scaling, servers, and infrastructure using Docker containers. The containers allow users to deploy both locally and in a cloud infrastructure, including Kubernetes, OpenShift[8], Mesos[9] and Compose[10]. This wide range of support is also extended to integration with other services, as OpenWhisk provides a catalogue of packages for the integration of popular services (e.g. Weather, Push-notifications, Slack, Github, etc.).

A key difference between OpenWhisk and other platforms is the limitation on the number of requests. OpenWhisk allows a maximum of 120 invocations per minute and a maximum of 60 triggers per minute within a function namespace. This is a limiting factor for applications with the potential to grow beyond the limit or large scale projects.

4.1.2.2 OpenFaaS

OpenFaaS is an open-source serverless platform, that supports the deployment of functions and microservices to the Kubernetes infrastructure, with the benefits of orchestrated scaling and infrastructure by the platform. OpenFaaS can be deployed on all platforms that support Kubernetes, allowing for both the use on private and public cloud. The OpenFaaS community has a store that allows for sharing and reusing of functions and boilerplate's.

4.1.2.3 Nuclio

Nuclio is an open-source tool for running DS pipelines serverless. The main goal of Nuclio is to simplify the process from code to deployment. Nuclio handles scaling, load balancing and more [30].

Nuclio runs on a Kubernetes cluster locally, cloud or on the edge. The resources available in the cluster are used for the execution of the functions. Nuclio provides the tools to easily deploy on the available resources and scale accordingly. The serverless aspect allows for more optimised utilisation of the resources but does not limit the cost directly.

Nuclio has a high request speed compared to other commercial platforms. Nuclio can scale quickly, by scaling within the function replica's, allowing a single replica to execute multiple invocations this is relaxation of the isolation. This reduces cold start-up and allows Nuclio to share and keep resources (e.g. data binding) alive within the replica for multiple function invocations. Functions do not have to initialise this connection every time, since it is kept available in Nuclio. This should make better use of resources and reduces the cold start time, but makes a pay per use model not fit in the architecture [31].

The next list will provide an overview of the features and properties available in Nuclio:

- GPU support (allows functions to run on GPU devices)
- Flexible config. (e.g. min-max instances of function, required GPU)
- No time limit on functions
- Not pay per use
- Functions workers have a shared state
- Containers are supplied by the developer (also allows custom run environments)
- Runs on provided resources (in Kubernetes)

[8] https://www.openshift.com/
[9] http://mesos.apache.org/
[10] https://www.compose.com/

- No full zero scaling. In the current version at least one pod needs to be active for a function.
- Simplify the development process.
- No limitation on storage/memory usage
- Multi-cloud

4.1.3 Serverless extensions on existing platforms

This section covers frameworks that provide serverless computing, by extending on existing serverless platforms. In many use cases, the out-of-box experience of the commercial serverless platform is not ideal. The extension provide custom solutions for their specific serverless issues and apply them on existing serverless platforms.

4.1.3.1 PyWren

Pywren [4] is a python framework, that integrates Python with AWS lambda. The main goal of PyWren is to support easy deployment of python functions on the AWS Lambda platform for scientists and people from computing.

An example is running map-reduces problems in PyWren. PyWren handles the creation and invocation of the function on AWS Lambda and scales the map-reduce. The scaling provides great speed-up, but a disadvantage is that the cost of serverless execution is higher compared to the specialised framework as Spark[11] or Bösen[12] [21].

4.1.3.2 Cirrus

Cirrus [5] is an ML framework for end-to-end management of data resources, that makes efficient use of the serverless architecture. Cirrus is built on the serverless architecture of AWS Lambda and the S3 storage and uses PyWren. Cirrus provides an interactive dashboard to track and manage at a higher application level for the hyperparameter optimisation stage. The framework offers an API for machine learning tasks. The API offers, pre-processing, training and hyperparameter-tuning.

The framework has several optimisations compared to PyWren. Cirrus, makes uses **data prefetching** to reduce idle-time from I/O, **mini-data batches** to run the job efficient in parallel on **lightweight workers** and **custom storage** to reduce latency. With these optimisations, Cirrus is 100x faster than a Pywren implementation [19] and 3.75x faster than a implementation in the traditional ML framework Bösen [20] on a Sparse Logistic Regression job. A disadvantage over serverless over these traditional ML frameworks is the cost [21].

4.1.3.3 Skippy

The work of [6, 23] present a framework for a serverless platform on Edge AI, that uses both edge and cloud devices for the scheduling of the functions. Their work extends the Kubernetes scheduler with a range of new priority functions. This allows the scheduler to more efficiently deploy on a serverless edge system, the new priority functions considers image latency (i.e. time to get a image running) of the nodes and data locality (i.e. download + upload time of data), capabilities (e.g. prefers a GPU) and locality type (e.g. cloud or edge). The new priorities introduce more constraints on scheduling and increase its complexity. Skippy is build on Kubernetes and also comes with a simulation tool called faas-sim[13], which allows for of simulation of serverless edge computing.

The main difference between Skippy and the others is that it focuses on the edge deployment in combination with the cloud, instead of a full cloud approach. This is an interesting idea as the benefits of serverless seem to fit well to edge computing. Scale-to-zero fits well with l energy-aware IoT and fine-grained scaling supports the vastly heterogeneous requirements and execution environments at the edge [1] property Skippy introduces serverless use in a hybrid environment with edge devices, bringing the simplicity of serverless to a new domain.

[11] https://spark.apache.org/
[12] http://docs.petuum.com/en/latest
[13] https://github.com/edgerun/faas-sim

4.1.3.4 Crucial

Crucial is [21] a serverless framework that focuses on high-concurrent stateful applications on a serverless architecture. Crucial is a Java implementation that makes use of the AWS Lambda platform.

Crucial uses distributed state objects along with annotations to support stateful variables in between the functions to support stateful high-concurrent applications. This allows a function to communicate the state quickly and not having to store it in an S3 bucket as an intermediary between function invocations.

Their experiment compares the performance of Crucial with Spark for both k-means clustering and logistic regression. The results range from comparable to superior performance over Spark. An interesting note on the cost of running, with the current cost of AWS, Crucial is always more expensive than Spark, where crucial running cost is $0.25 per second and Spark $0.15 per second [21].

4.2 Categories of serverless

Serverless platforms are implemented with various features and functionalities. This section, defines the core of serverless, based on the formal definition and show a collection of extra features that are based on frameworks discussed. The features add value to a platform and allows for a more specialised platform. This section will first discuss the core functionalities that are required for a serverless platform. Next, we show the standard features, these are not mandatory but are common and implemented in the mentioned commercial serverless platforms. Finally, extra features that are specialised features for serverless that can benefit certain application use cases.

4.2.1 Core

In the formal definition of serverless [15] the following main functionalities of serverless platform where identified:

- Function replicas **start with an initial state**.
- Function executions are **started by an event/trigger**.
- Serverless **functions are deployed on the platform** by the developer or third-party system.
- Functions replica's can **stop at any time**.
- The platform handles **resource management, allocation and run-time configuration.**
- The serverless function is a **language level abstraction from cloud computing**.
- Function execution **starts on idle instances**, and **reuse their state** (this can be different than the initial state).

4.2.2 Standard

The core of serverless is missing features that make it suitable for general use, such as fault tolerance. The following list shows the standard features:

- Serverless function is **executed at least once**.
- The serverless functions run with **implicit parallel execution**.
- **Full isolation** on the execution of a serverless function.

4.2.3 Extra

This section contains a list of specialized features found in the frameworks and literature

- **Distributed shared object layer**, which provides a mutable state that is shared between the function replicas. The data is provided by in-memory storage deployed alongside the serverless platform [21].
- **Relaxed isolation** of the serverless functions, e.g. multiple instances on the same run-time environment. However strong isolation is the basis for multi-tendency, fault isolation and security [22].
- **Sharing data bindings**(e.g. sharing connection string, object, database, etc.) within multiple executions on a function replica, overhead is reduced as data is reused in the next function executions [30]. Reusing these data bindings can increase the chance of faulty execution and introduces race conditions and lock if not used as read-only outside of the function replica initialisation.
- **Function versioning** allows for the execution of a specific version of a serverless function. This can be used for testing new functions or for backwards compatibility [30].
- **Detect likelihood of function failure** and negate the risk by starting multiple instances [15].
- **Function replica scheduling**, with more complex constraints (e.g. run with specialised hardware or with an isolation level). The more constraints, the harder it becomes for the platform to achieve high resource utilisation [22, 23].
- **Simple user-agnostic scheduling** provides a basis for short-startup times and high resource utilisation [22].
- **Relaxation of disaggregation**, Instead of shipping the data from to the computation is delegated to a stateful storage tier, to improve performance. This negatively impacts elasticity in a cloud environment [22].
- **Custom storage** (optimised storage), provides a speedup in as latency is reduced from 10-100ms (AWS S3 storage) to as low as 300μms [5]. Creating specialised storage to a problem instead of generalised storage can reduce the latency, thus increasing the function throughput.
- **Data pre-fetching** to function replicas. Moving data to the function before execution can speed the function run-time, as data does not need to be serialised and is downloaded during the execution of the function. This requires fast accessible storage and predictable data usage [5].
- **Policy based function composition**. Function composition allows chaining of multiple functions with given triggers and thresholds(e.g. receiving a given accuracy for an ML model), policies given in serverless functions allow for more control on the execution (e.g. run within a given network or inject an ML model) [23].
- **Servermix**, Combine serverfull (e.g. scheduler, coordinator and parameter servers) and serverless services, can increase performance significantly [22].

4.3 Summary

In this chapter, an overview of the landscape is provided. The overview shows the differences between the frameworks in practice. The chapters showed a range of hosted platforms and self-hosted platforms, some general-purpose (e.g. OpenFaaS and AWS lambda) and others specialised(e.g. Nuclio for data science). Next, the elaborations provide insight into the distinguishable features, limitations and benefits of the frameworks. These features are then combined in a set of serverless categories. The categories provide us with a clear list of functionalities that either must be, should or could be part of a serverless framework. This overview provides a good overview of ideas in the existing work and the state of the art.

With these elements, the landscape of serverless frameworks is provided and a clear set of features is distilled from the landscape increases, providing insight into the state of the art and introduces interesting starting points for future research.

Chapter 5

Data locality aware scheduling of data-intensive functions on a serverless edge platform

The last chapter introduces the idea of relaxation of disaggregation as a specialisation feature of serverless. An example of this relaxation is shipping the computation to the data. Limiting computation to data location will limit the elasticity of a serverless system, but also has benefits for certain use cases.

Data-intensive applications have to handle large amounts of data, the size of data is directly connected to the bandwidth and time required to move the data. Another growing concern in data-intensive applications is the privacy of personal data. Scheduling on a serverless edge platform does not offer control on the location of execution for functions, thus not having control on bandwidth, cost and network traffic in general.

A scheduler that can restrict scheduling to nodes within the local network of the data, offers more control to execution on a serverless platform. When handled on a function level developers can isolate functions with high bandwidth requirements or sensitive data, by constraining execution to the local network. Executing in a local network reduces bandwidth usage and keep data within the network. This chapter will propose an implementation for a data locality aware scheduler for a serverless edge platform. First, the problem is motivated and defined. Followed by an elaborated description of data-aware scheduling. Finally, a proposal for implementation is made, that supports data-aware scheduling on a serverless edge platform.

5.1 Problem statement

Data-intensive applications with large amounts of data being moved or handle sensitive data that need to stay with premise are unsuitable for serverless edge platforms, due to a lack of control on deployment location.

A serverless platform handles the orchestration of the functions, including deployment location. From a developers perspective, no mechanism provides guarantees on the scheduling locations.

Existing work has proposed a scheduler [23] that considers multiple parameters from the serverless edge environment to improve the efficiency of scheduling. The scheduler uses a set of weighted priority functions to steer the goal of the scheduler. The priority functions include a data locality priority, which considers the time needed to move data from the source to the nodes. The goals can be minimising uplink/downlink bandwidth, resource utilisation, cost or execution time, by fitting the weights of the priority functions.

The scheduler does not provide guarantees on the execution location of functions, as it uses soft constraints for scheduling. Also, the priority functions are applied on all functions, making it hard to fit a system with different goals between the functions (e.g. minimizing uplink/downlink for pre-processing and minimising execution time for inference).

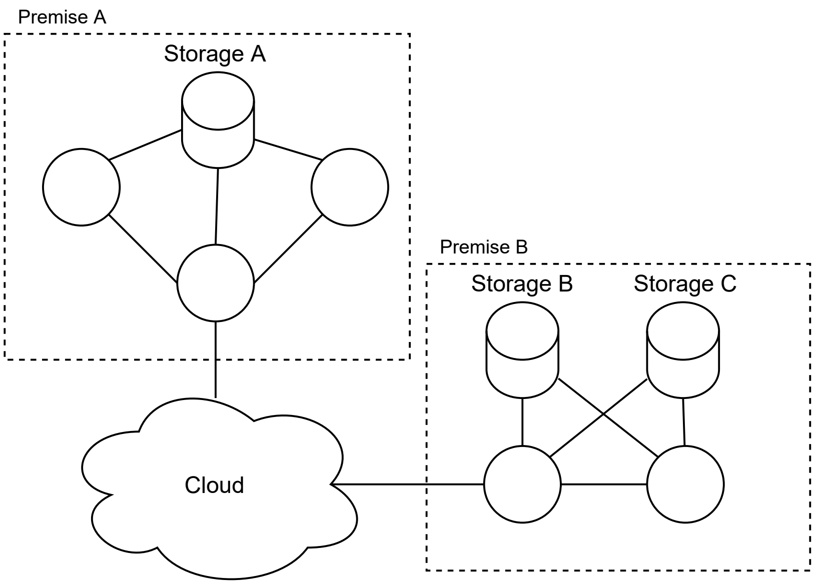

Figure 5.1: Overview serverless edge system

5.2 Data locality aware scheduling

For this work, data locality aware scheduling is considered the scheduling of functions within the sub-network of where the data resides. Figure 5.1 shows an overview of a system with nodes in two premises and the cloud. An example, let f be a function that requires data from storage A. Using data locality aware scheduling the scheduler is a constraint to deploy function f on the nodes in premise A. Another example, let g be a function that requires data, which is stored in storage A and Storage C. The data locality aware scheduler can use all nodes in Premise A and B for scheduling of function g.

The scheduler will enforce these hard constraints using additional metadata and deploy function replica's on nodes within a premise where the data resides, guaranteeing that all invocations are performed within the premise of the data. In the following paragraphs, the process from function development to function executions is described in more detail.

5.2.1 Tagging of the functions

The first step is tagging the functions to indicate that it needs to enforce data-aware scheduling for the function. A tag provides the developer function level control scheduling and can be set in the code of the function. Listing 1 shows an example function with the data-aware tag and also the data consumes and produces a tag for the required data, which is used to determine the locality. This work simulates the function execution, in which the internal implementation of the functions is not considered. To provide the tag to the simulations, it is directly set in the metadata of the functions, skipping the step of collecting metadata out of the function definitions. Other works propose [23] and implement [21] a similar tagging system.

5.2.2 Enriching the system with metadata

The scheduler needs additional metadata to support the data locality aware scheduling. The first metadata that is required is the topology of the system network. The topology is extracted from

Listing 1 Example of tagging

```
from sklearn.svm import SVC

@data.consumes("bucket:train_data")
@data.produces("bucket:model")
@scheduling.data_aware(True)
def handle(req, data):
    clf = SVC()
    clf.fit(data.data, data.target)
    return clf
```

the bandwidth graph available via the Skippy-daemon. The topology shows the nodes, types and connections, this provides insight into the relative location between the nodes and sub-networks can be identified.

The next metadata is the storage nodes that are present in the premise of a node. Each node knows which storages are within its premise to support scheduling when a tag is provided. The data is extracted using the topology, all the storage nodes are iterated and all the nodes in its premise store the name of the storage in their metadata. This allows for filtering by the scheduler.

5.2.3 Reducing the nodes for scheduling

Next, the feasible nodes for scheduling are reduced to the nodes within the premise of the data. The scheduler starts with evaluating the requested function replica for a data-aware tag. When the tag is present the scheduler gets a list of storage nodes that contain the required data. Next, the metadata of the nodes is evaluated and reduced based on the presence of the storage nodes in its premise. The scheduling will proceed as normal with the remaining nodes within the premise(s) of the storage node(s).

5.2.4 Invocation

The incoming invocations of functions are scheduled on the deployed functions replica's. If a function is tagged as data-aware, then the function replica will be placed on-premise, as a result, all invocation will be performed within the premise of the data. The incoming invocation then executes the functions on the replicate, keeping data within the local network.

5.3 Proposal Data locality aware scheduling

In the last section, an overview of data-aware scheduling was given, with a description of how the system should behave.

This section provides a proposal of the algorithms that support data-aware scheduling on a serverless edge platform.

First, the environment of the suggested proposal is elaborated. Next, pseudo-code of the enriching of the metadata with locality; filtering of the nodes on locality and scheduling is provided.

5.3.1 Run-time environment

For the run-time environment, and OpenFaaS implementation on Kubernetes is used. Both are well known, Apache 2.0 licensed and MIT licensed, respectively and are open-source platforms [27], which allows for adjustments of code and redistribution. The FaaS-sim simulator and Skippy is also built on this Kubernetes and OpenFaaS allowing this work, extend on their work and validate data-aware scheduling using the simulator.

The modifications suggested by this work are limited to the scheduler and the metadata in Kubernetes. The idea of data-aware scheduling could be ported to other schedulers given that the system can be provided with the required metadata.

5.3.1.1 Enrichment of metadata with locality

The enrichment of the metadata required both the topology of the network and the storage nodes need to be identified. This work assumes that the topology is available, the simulation uses a manual defined topology. For the identification of the storage nodes, it is assumed that every node has a running daemon that can identify its resources and enrich the Kubernetes metadata. In the case of the simulation, this is by the Skippy-daemon.

With this information, the metadata of all nodes in the same premise as the storage nodes is extended with the information which storage nodes are on the premise.

Algorithm 1 LabelNodesWithLocality (Labelling split from searching over the network)

```
1: procedure labelNodesWithScope(storageNodes)
2:     for storageNode in storageNodes do
3:         nodesInPremise ← get all nodes in same premise as the storageNode (where nodeInPremise ≠ storageNode)
4:         for node in nodesInPremise do
5:             node.labels.scope $\overset{+}{\leftarrow}$ storageNode.name
6:         end for
7:     end for
8: end procedure
```

The algorithm 1 shows pseudo-code for the algorithm used to compute the local storage nodes for the nodes. The algorithm loops overall storage nodes and adds its name to the data scope of all connected nodes. Connected nodes are nodes directly connected to the switch within the premise (This is sufficient for the used topology).

5.3.1.2 Filter nodes

Now that each node has a list of storage nodes that are within its local network, the nodes can filtered to nodes that have a direct connection to the required storage. By only selecting nodes that are on-premise of one of the storage nodes containing the data, it is assured that the scheduling is within the premise.
The algorithm 2 shows the pseudo-code of the filtering of nodes. First, the required data bucket is collected from the pod, next, all storage nodes containing this data are collected. Finally, there is a loop over all nodes to see if the node is within the sub-network of the required storage node.

5.3.1.3 Scheduler

The scheduler used in the experiment is based on the python implementation of the Kubernetes scheduler in FaaS-sim [6] and is modified to support the data-aware scheduling.

The algorithm 3 shows the pseudo-code of the scheduler. The scheduler first reduces all nodes to the nodes within the data locality of the pod. Next, the nodes are filtered to the nodes that match the set predicates. Finally, the remaining nodes are scored on a set of given priority functions for the Greedy MCDM strategy, the node with the highest combined score is selected and the pod will be scheduled on the suggested node.

5.4 Summary

In this chapter, the concept of data locality aware scheduling is shown. Starting with the general idea, gradually going into more detail and proposing a set of three functions to support scheduling with a hard constraint on the execution within the premise of the data. These are the following

Algorithm 2 filterNodesOnDataLocality

```
    procedure filterNodesOnDataLocality(nodes, pod, clusterContext)
2:      if pos has label data-aware then
            $requiredBucket \leftarrow$ get bucket required by pod
4:          $storageNodes \leftarrow$ get all storageNodes containing data bucket
            $filteredNodes \leftarrow empty\ list$
6:          for node in nodes do
                if node has localStorages then
8:                  for localNode in localStorages do
                        if localNode in storageNodes then
10:                         $filteredNodes \leftarrow localNode$
                        end if
12:                 end for
                end if
14:         end for
         return filteredNodes
        end if
         return nodes
16: end procedure
```

Algorithm 3 Scheduler

```
1:  procedure Schedule(pod, clusterContext, predicates, priorities)
2:      $nodes \leftarrow allNodes(clusterContext)$
3:      $nodes \leftarrow filterNodesOnDataLocality(nodes, pod, clusterContext)$
4:      $nodes \leftarrow feasibleNodes(nodes, pod, predicates)$
5:      $suggestedHost \leftarrow Max(scoreNodes(nodes, priorities))$
6:      if $suggestedHost$ is not Null then
7:          $placePod(pod, sugestedHost)$
         return SchedulingResults(suggestedHost, nodes)
8:      end if
9:  end procedure
```

three steps: (a) enrich metadata with storage nodes in its premise ; (b) filter nodes in the network, to nodes within premise(s) of the required data and (c) schedule the pod of the function replica on one of the remaining nodes.

Chapter 6

Experiment

Last chapter proposed an implementation of data locality aware scheduling on a serverless edge computing platform. In this chapter an experiment is defined, that validates the proposed scheduler compared to the Kubernetes scheduler in the Faas-sim environment.

This chapter will describe the topology of nodes and there bandwidths, the workload, the configurable parameters of the benchmark (e.g. interval of arrival, function execution simulation and scaling of replica's), scenario's and gathering of results.

6.1 Topology

The topology defines the composition of nodes within a network. The topology of the benchmark considers five types of nodes. (i) **Cloud** nodes are VMs running in the cloud. (ii) **Raspbery pi's** (RPI) nodes are single board computers that can be used for reading sensor data or controlling actuators. (iii) **Nvidia Jetson** also called tegra nodes in this work, are single-board computers with GPU acceleration. (iv) **NUC** nodes are small form factory computers with desktop capabilities. (v) **Storage** nodes contain the data of the system, these nodes do not use computing capabilities and storage limits are not capped. The specification of the computation nodes are shown in Table 6.1

Device	**CPU**	**Memory**	**Architecture**
Cloud	4 cores	7.79GiB	amd64
Raspberry Pi 3B+ (RPI)	4 cores	975.62MiB	arm
NVidia Jetson TX2 (Tegra)	4 cores	7.67GiB	arm64
Intel NUC	4 cores	16GiB	amd64

Table 6.1: Node devices

The structure of the topology in the benchmark is shown in Figure 6.1. It shows the network that consists of a cloud with multiple cloud nodes and multiple premises with configurable number of tegra, nuc and RPI nodes. The cloud and each of the premises is connected over the internet via switches and the switches are connected to all devices within the premise. The bandwidth configuration used in the benchmark is 1Gb/s connection for all connections. It is assumed that the premise has LAN connection and the connection to the internet is a business internet service provider. Moving data between premises or cloud is still twice as slow within the premise, as the data needs to make an additional hop between the switches.

6.2 Workload

The workload of the experiment simulates a machine learning pipeline, consisting of a preprocessing, training and serving function forming a pipeline. The workload is reused from [23], based on a test bed they created distributions for startup time and execution and the resource usage for the images. Also it is built in FaaS-sim and the pipelines highlight different types of images within

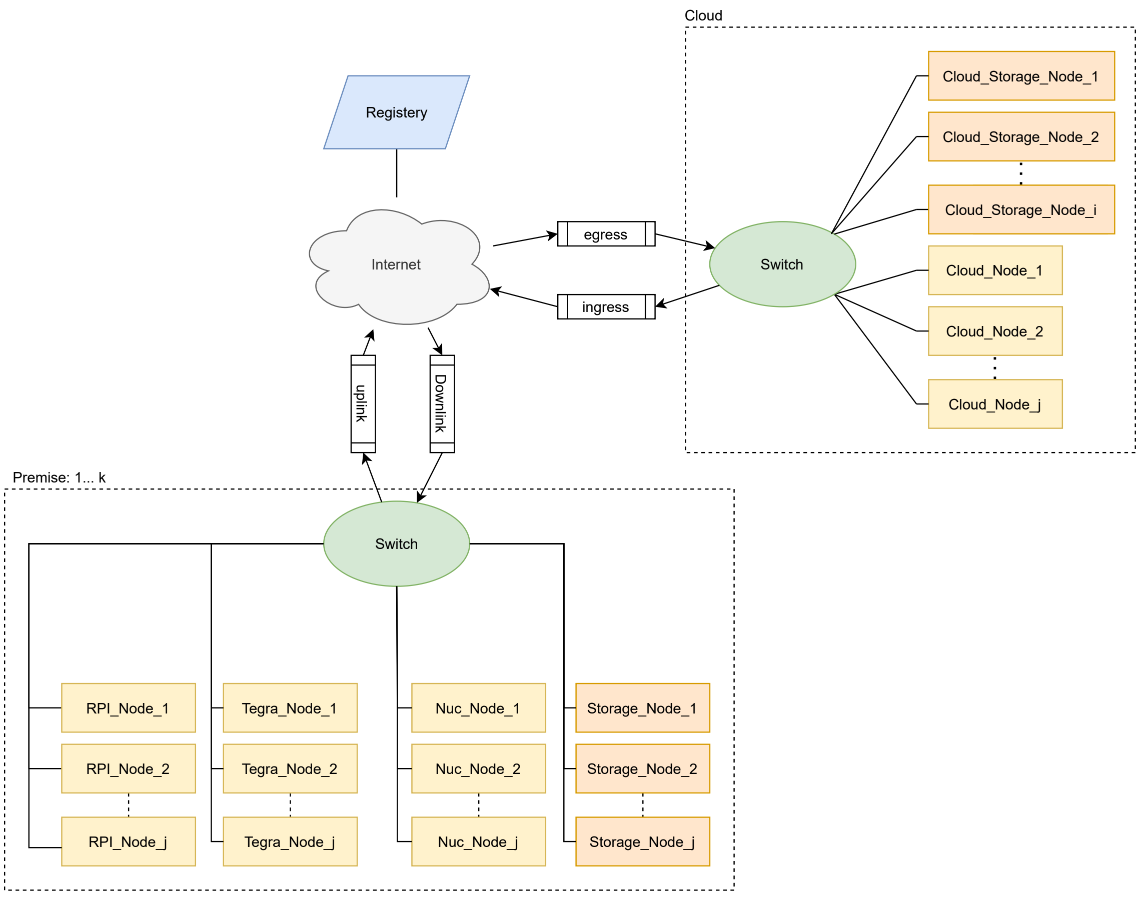

Figure 6.1: Topology of the water company use case

a data-intensive system.

The execution time and startup time distributions from the test bed are shows in appendix A. Other parameters like image size, required download and upload file sizes and the image architecture are manually set in the implementation (see Table 6.2). The workload is a set of distributions and data that simulates machine learning pipelines, based on a real world benchmark.

Image name	**Task**	**Size**			**Download**	**Upload**	**Cpu**	**Memory**
		amd64	**arm**	**arm64**				
ml-wf-1-pre	pre-processing	533Mb	466Mb	540Mb	12MiB	209MiB	100m	100MiB
ml-wf-2-train	training	550Mb	519Mb	594Mb	209MiB	1500KiB	100m	1GiB
ml-wf-3-serve	serving and inference	589Mb	512Mb	591Mb	1500Ki	-	100m	200MiB

Table 6.2: Specification of images used in the work load

6.3 Benchmark

In this sections the elements of a benchmark are defined and elaborated. First the configurable parameters of a benchmark are described. Secondly the interval of arrival is explained and the chosen configuration for the benchmark. Next a overview is given of how the data is distributed over the nodes, during the benchmark. Followed by an explanation of the function simulator, that emulates the behaviour of the execution of functions on a function replica. The last paragraphs elaborates how the scaling of the functions is handled in the benchmark.

6.4 Benchmark parameters

The benchmark is configurable through multiple parameters. These parameters tune the benchmark and allow multiple scenario's to be explored. The parameters of the benchmark include:

- Scheduler (e.g. Kuberneter or Data locality aware)
- Simulation duration
- Request per seconds for the serving functions
- Percentage of functions that is tagged as data locality aware. (only impacts data locality aware scheduler)
- Amount of nodes on premise, per type
- Amount of premises
- Amount of pipelines deployed (set of pre-processing, training and serving images).
- Seed

6.4.1 Interval of arrival

The interval of arrival is the interval between invocation of a function. The interval of rate of the serving function can be defined in the benchmark parameters. The interval values of preprocessing and training are fixed values they are not invoked by the end users, but a lower interval by the developer or system. The base value for preprocessing is set at once every second and training is set for once every 10 seconds in simulation time.

The simulation uses the values as a base for an exponential variant distribution to sample the interval of which the functions are invoked in the simulation. This allows for up and down scaling to occur, since the interval rate is not a constant value.

6.4.2 Storage initialisation

Each function requires data to be downloaded and uploaded to a storage bucket. The data location of these buckets is used for filtering the nodes on data locality. In order to have a fair distribution over the premises and the cloud storage nodes, the storage buckets are uniformly distribution over the storage nodes. Next each pipeline is assigned one bucket.

6.4.3 Function execution simulation

The function execution simulator defines the behaviour of function invocation and setup in a function replica for the simulation. Each function replica can handle one function invocation at a time. The invocation handles execution including downloading (non cached data) and uploading the results. On invocation a lock is required to enforce that there are no concurrent executions within the replica, with the lock acquired the invocation start. First the downloading of data is simulated, next a execution time is sample from the distribution and the execution time is simulated. Finally the uploading of data is simulated and the lock is released. The function replica's of the serving image download and caches the model on replica setup, reducing the redundant downloads.

6.4.4 Scaling of functions

The simulator supports multiple variants of scaling out of the box and supports custom scaling implementations. The benchmark uses the existing queue average scaling. This scaling method does not require a predefined fixed number of throughput's per functions deployment, offering dynamic scaling to the system. The scaling looks at the average number of queued invocations over replicas execution a unique function. When the average queue is to small or to big, the system tries to scale to get the targeted queue size. As an example, there is one queue A and A aims for a queue size of 10. When queue size of A exceeds 10 an new replica is deployed, immediately reducing the average to 5 and allowing new invocation to be handled by both replica's. With two replica's it can be that the average queue size goes below a threshold, causing the system to scale down one non active replica. A downside of this approach is that latency is directly impacted by queue size as the system scales to match the queue size, where $waiting\ time \approx queue\ length\ *\ execution\ time$.

6.5 Setup

A set of experiments will run to the performance of data locality aware scheduling, using the defined topology and benchmark. In this section the experiment scenario's are described and the benchmark parameters of the experiment are defined. Figure 6.2 show the topology used for the runs and maintaining the structure shown in Figure 6.1

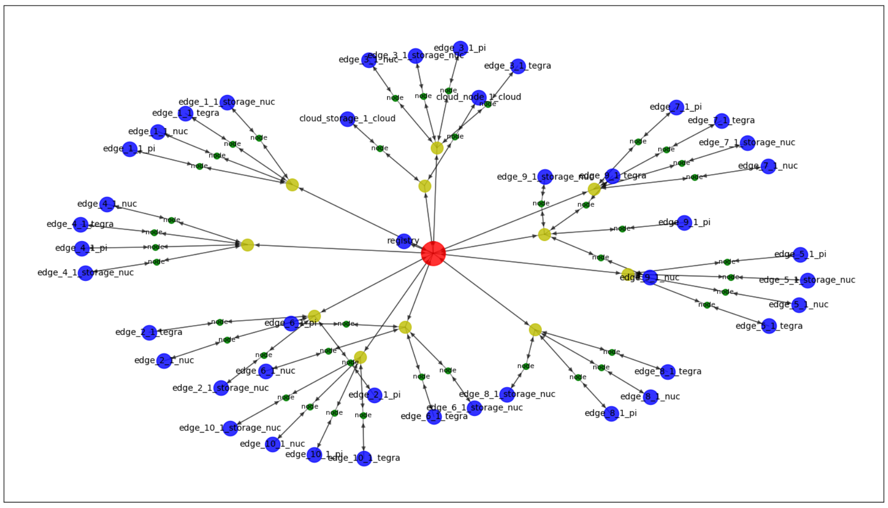

Figure 6.2: Topology of simulations

6.5.1 Scenario 1: Data locality aware scheduling

This scenario looks at the impact of data locality aware scheduling. The scenario explores how a different percentage of tagged function impact the performance compared to the Kubernetes scheduler. Table 6.3 shows the benchmark parameters for this scenario.

Percentage data locality aware tagged functions	[50, 100]
Amount of premises	10
Amount of nodes in a premise	4 (1 RPI, 1 Nuc, 1 Tegra, 1 storage)
Amount of nodes in the cloud	1 cloud and 1 storage
Serving rate per second	1.5
Amount of pipelines	10
Simulations duration	1000
Seeds	[1,2, ..., 10]

Table 6.3: Benchmark parameters scenario 1: Data locality aware

6.5.2 Scenario 2: Ratio deployments

The second scenario looks at the scheduler with different levels of saturation. The scenario runs an increasing number of serving invocations, making the system scale serving replica's to match the load. This scenario's is to show how data locality aware scheduling performs relative to the saturation of the network. All benchmark parameters are shown in Table 6.3

Percentage of functions tagged data locality aware	100
Amount of premises	10
Amount of nodes in a premise	4 (1 RPI, 1 Nuc, 1 Tegra, 1 Storage)
Amount of nodes in the cloud	1 cloud and 1 storage
Rate per second	[1, 1.5,...,6]
Amount of pipelines	10
Simulations duration	1000
Seeds	[1,2, ..., 10]

Table 6.4: Benchmark parameters scenario 2: Saturation with data locality aware

6.5.3 Run Environment

All the experiments are run using the Peregrine cluster[1] of the RUG. This way all experiments are run in a controlled environment and can be conducted within reasonable time. This experiment uses the standard nodes of the Peregrine cluster, these nodes have 28 cores, using two 3.5Ghz Intel Xeon E5 2680v3 CPUs and have 128Gb RAM. The experiments are submitted as jobs to the cluster, using the Slurm workload manager[2].

6.6 Gathering of results

The simulations of the benchmark results in extended set logs. The logs are collected and are analysed and plotted in the results. The following list gives an overview of logs produces by the simulation:

- Execution time
- Function execution flow
- Function deployment
- Functions
- Invocations
- Bandwidth usage
- Scale

[1]https://www.rug.nl/society-business/centre-for-information-technology/research/services/hpc/facilities/peregrine-hpc-cluster

[2]https://slurm.schedmd.com

- Scheduling
- Utilisation

Chapter 7

Results

The last chapter explained the experiment setup and its parameter. The experiments were run and the results are shown in this chapter.

The results of both scenarios are elaborated. The scenario compares the data locality aware scheduler against the Kubernetes scheduler, with both all and half of the functions marked for scheduling with the data locality aware scheduler.

7.1 Scenario 1: Data locality aware scheduling

The first results are based on ten runs with the same configuration. In the following figures, the box plots show the mean of ten runs with a solid line and the standard deviation is shown by a transparent ribbon. The following figures show the results of the scenario and are elaborated with a short description.

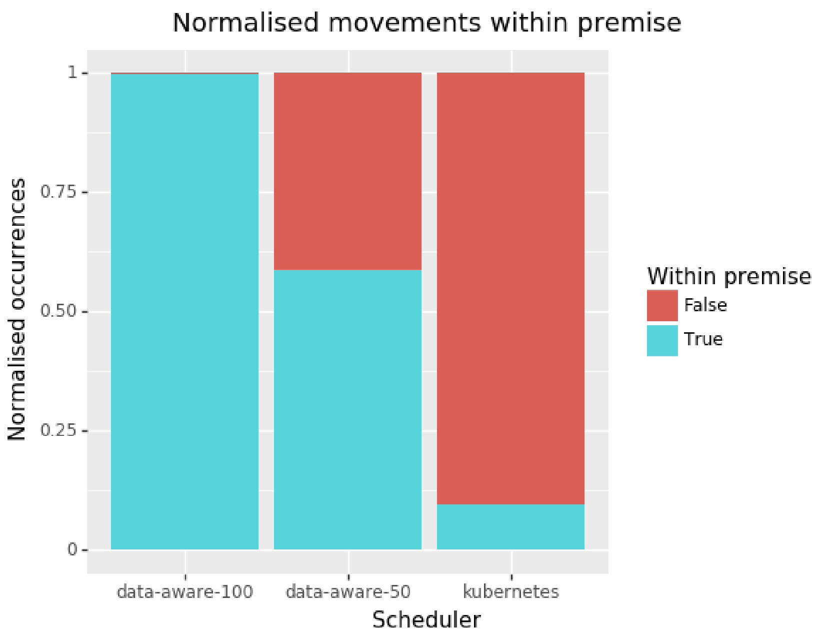

Figure 7.1: Normalised movements that go in or out of the premise

Figure 7.1 shows the normalised number of movements that stayed within the premise of the data. It shows that the data locality aware scheduler only has a few movements outside its premise, this are the downloads of the images of the functions. With 50% of the functions marked, the movements within the premise are reduced to +-55% and for the default Kubernetes scheduler, only +-10% of the movements are within the premise.

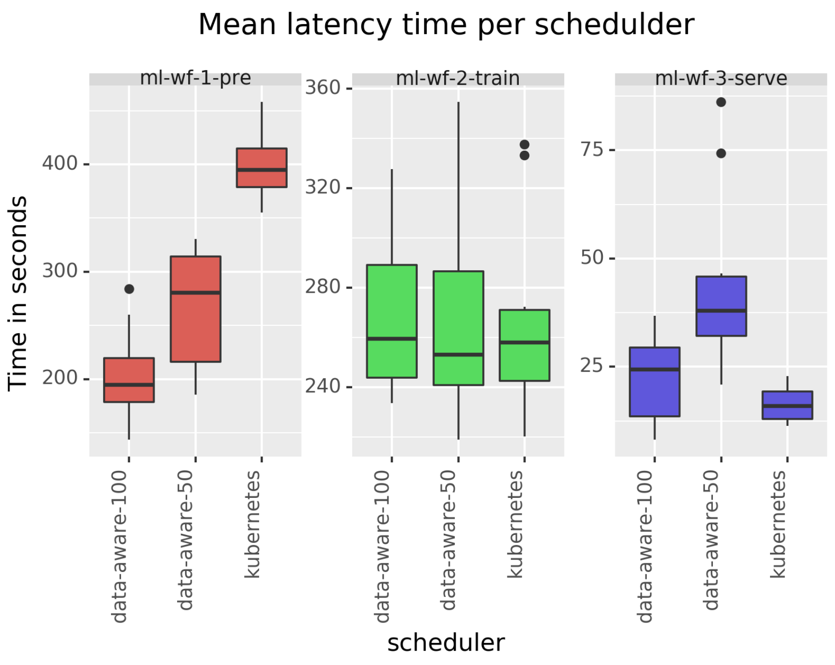

Figure 7.2: Mean waiting time for function execution per image type

Figure 7.2 shows the latency of the schedulers for each of the functions. Latency is the time from invocation to response, in this figure, this is equal to the sum of function execution time and waiting time. The waiting times are way larger than the execution time and therefore the latency is mostly determined by the waiting time. To get a good idea of how both schedulers compare these values need to be inspected separately.

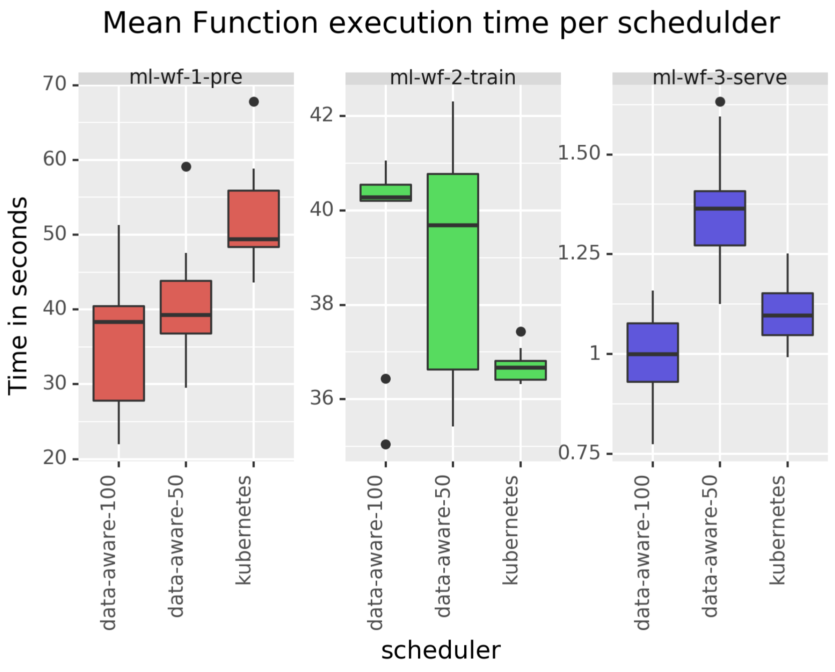

Figure 7.3: Function execution of different image types

Figure 7.3 shows the function execution time per image. Looking at the execution times of the preprocessing and serving functions, the data locality aware scheduler with all functions tagged outperforms both Kubernetes and data locality for 50% of the functions.

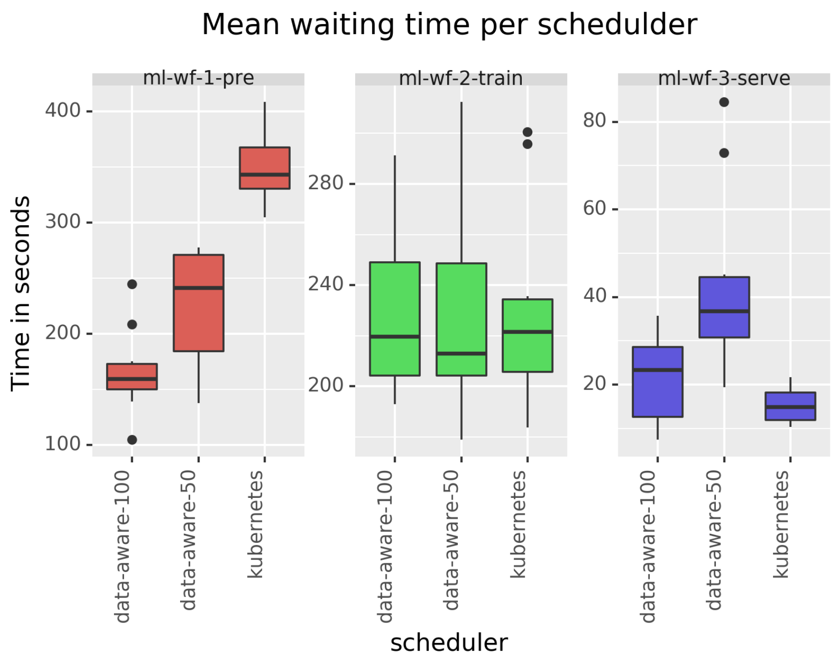

Figure 7.4: Box plot mean waiting times for the images

Figure 7.4 shows the waiting time of function invocation for the different image types. The data locality aware scheduler performs significantly better than the Kubernetes scheduler on the preprocessing functions. Looking at the serving functions it can be seen that the Kubernetes has lower waiting times.

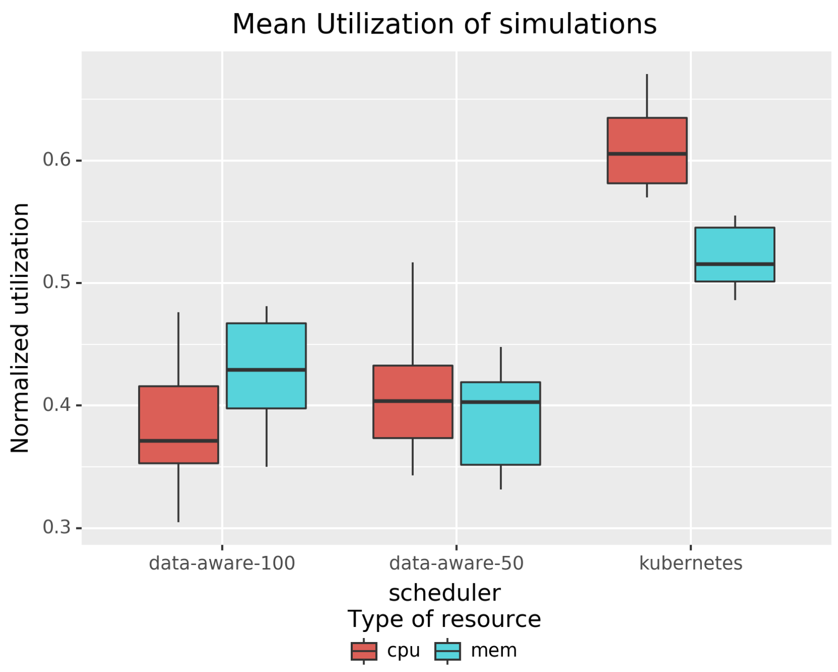

Figure 7.5: Mean utilisation for function execution per image type

	p-value		
	Pre-processing	Training	Serving
Function execution time	**4.061e-4**	**7.975e-5**	0.221
Function waiting time	**4.888e-10**	0.907	**1.331e-3**
Function latency	**1.056e-10**	0.959	**1.373e-3**

Table 7.1: p-values per image types between data locality aware scheduler with all functions marked and the Kubernetes scheduler

Figure 7.5 shows a boxplot of the mean utilisation over the simulation runs. Fully restricting all functions seems to reduce the utilisation of the entire system. Also with 50% of the functions tagged the utilisation drop significantly compared to the Kubernetes scheduler.

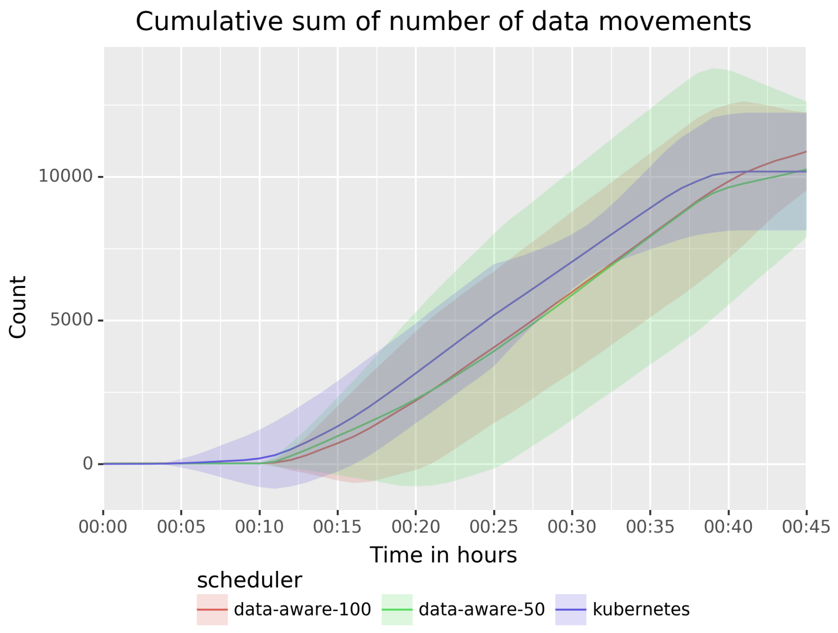

Figure 7.6: Mean waiting time for function execution per image type

Table 7.1 shows the p-values as a result of performing a t-test between the Kubernetes scheduler and the data locality aware scheduler with all nodes marked for constraint scheduling.

7.2 Rate of arrival

The second scenario runs for a range of parameters exploring the behaviour of a saturated. The results compare the data locality aware scheduler where all functions are tagged and the default Kubernetes scheduler. The results plot the rate per second of the serving image against the measurements. The solid line indicates the mean and the transparent ribbon indicates the standard deviation of the runs. Figure 7.7 shows the mean function execution times for each of the image types. The execution time includes downloading the required data, the sample execution time of the function, and uploading the result.

As the image shows, Kubernetes has a lower standard deviation for all of the images. When looking at the pre-processing image shows that the data-locality aware scheduler outperforms the Kubernetes scheduler. The Kubernetes scheduler performs better on training functions than the data-locality scheduler, with a smaller standard deviation. For the serving images both means are similar, where the Kubernetes scheduler start better, but quickly align with the same downward

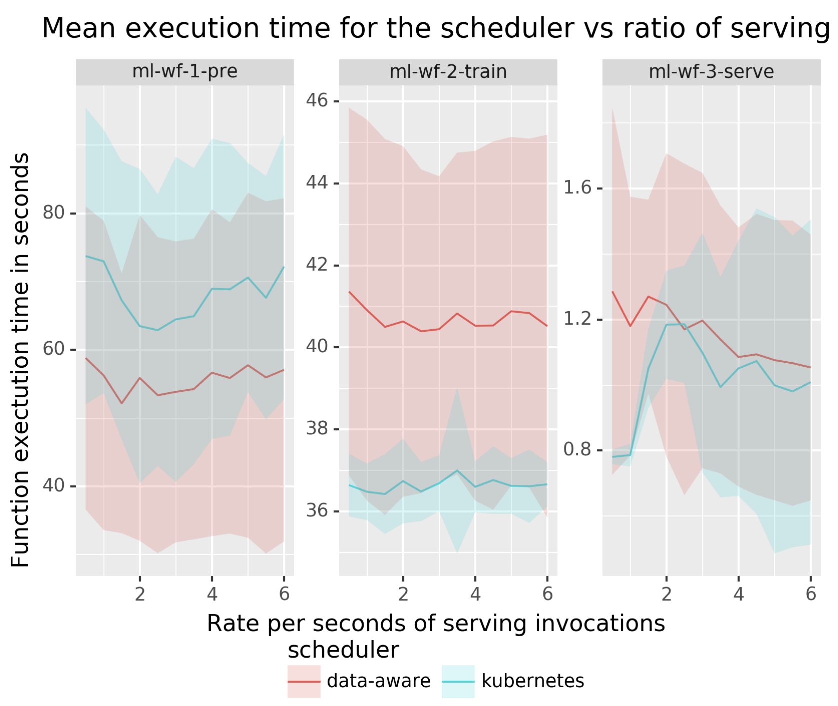

Figure 7.7: The mean function execution time of the images plotted against the rate per second of serving invocations over 10 runs of the simulation

trend as the data locality aware scheduler. Also, the standard deviation increased as the serving invocation rate increases.

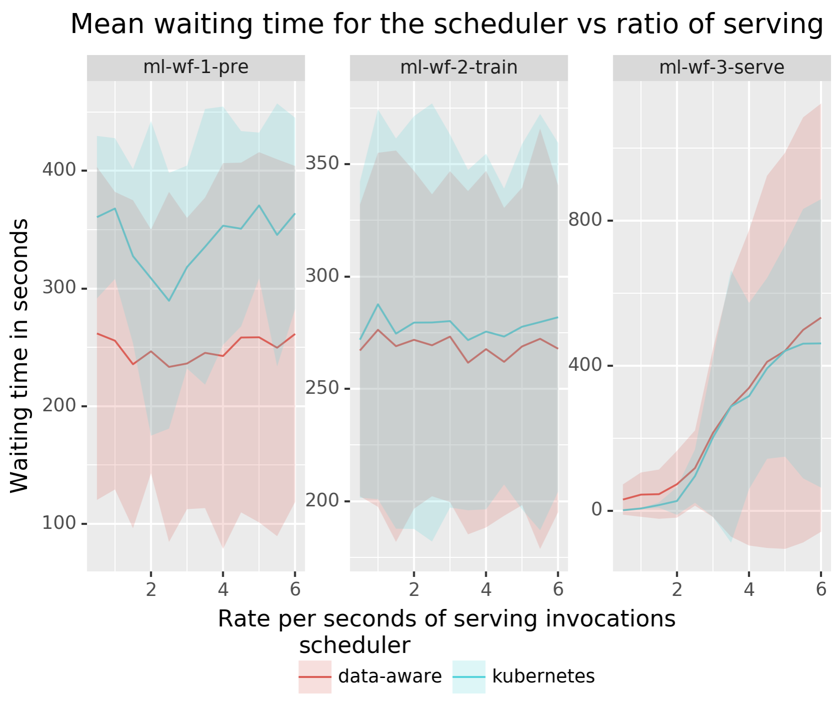

Figure 7.8: Mean waiting time for function execution per image type

Figure 7.8 shows the waiting time for the different functions. The waiting time is the measured time from the function arriving in the queue till the start of the execution. The queue size threshold is set to five for training and preprocessing and ten for serving. This results in the system aiming for a waiting time similar to 5x or 10x the function execution time.

The first thing that the figure shows is significant waiting time for the pre-processing and training image, both having a mean waiting time of over three minutes. Next, the pre-processing images have a lower waiting time with the data-locality aware scheduler and a longer but more predictable waiting time for the Kubernetes scheduler. For the training images, both schedulers perform very similarly, with the data-locality aware scheduler having a little lower waiting times. Next for the serving image waiting times increases as the number of invocations per second grow and the Kubernetes scheduler has a lower standard deviation.

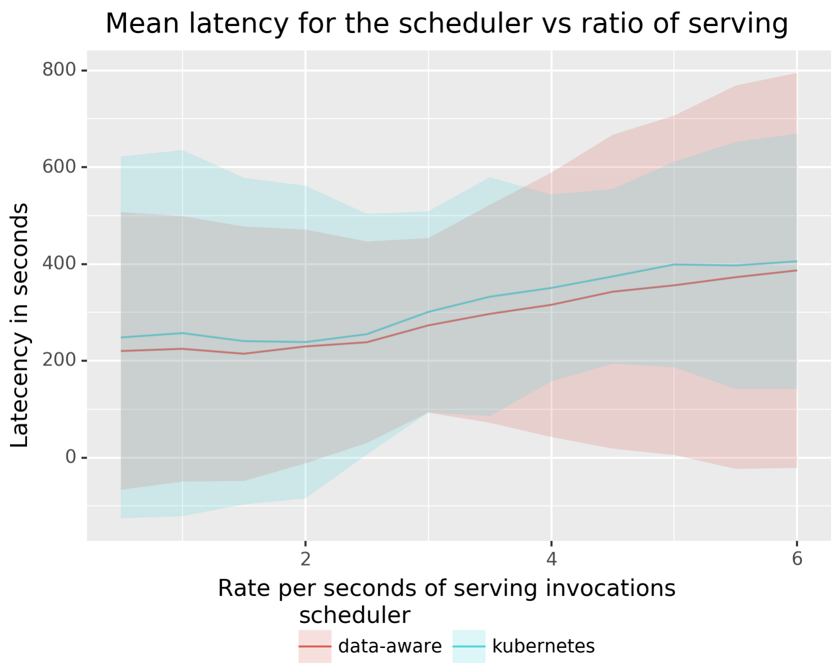

Figure 7.9: Mean latency for the over ten simulations

Figure 7.9 shows the latency for the different images types. The latency is the time when a function enters the queue of a replica until completion of execution, in other words, function execution time and function waiting time combined.

The results are almost identical to the function waiting time, as this value is significantly larger than the execution time. A clear thing that shows is that the user has to wait longer and longer for a response of serving tasks as the system becomes more saturated by the serving instances.

Figure 7.10 shows the number of data movements per image type over a simulation. The measurement counts the number of movements made in the system, where intermediate hops are not considered. Preprocessing and training both have two movements per invocation, as they download upload data for each execution. Serving only moves data at setup as it caches the model and has no upload. Another movement to consider is the initial image download in the setup of the function replicas.

In both pre-processing and training, it shows that on average the data-locality aware scheduler can perform more movements. Serving shows that movement goes down as the number of invocations scale.

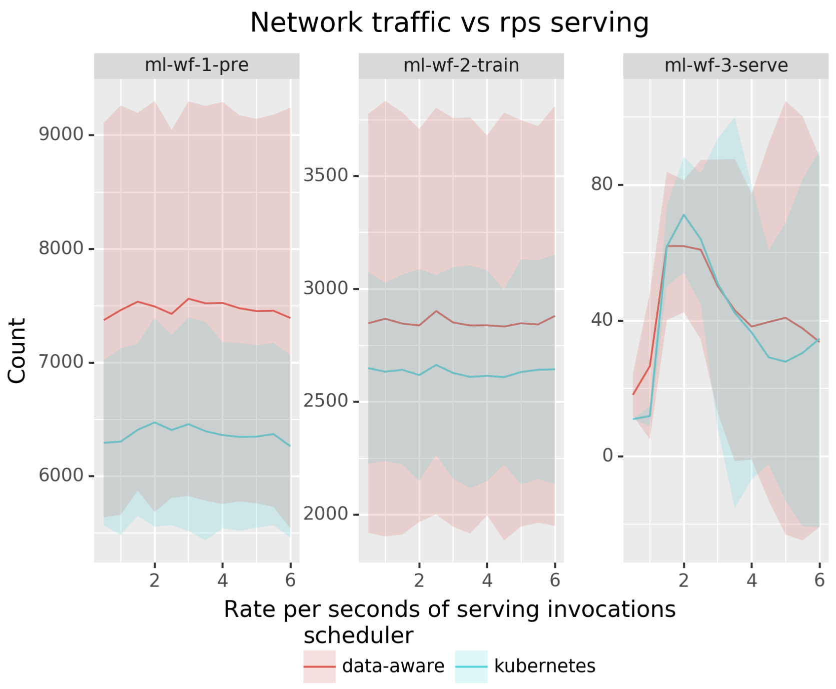

Figure 7.10: Number of data movement per image type

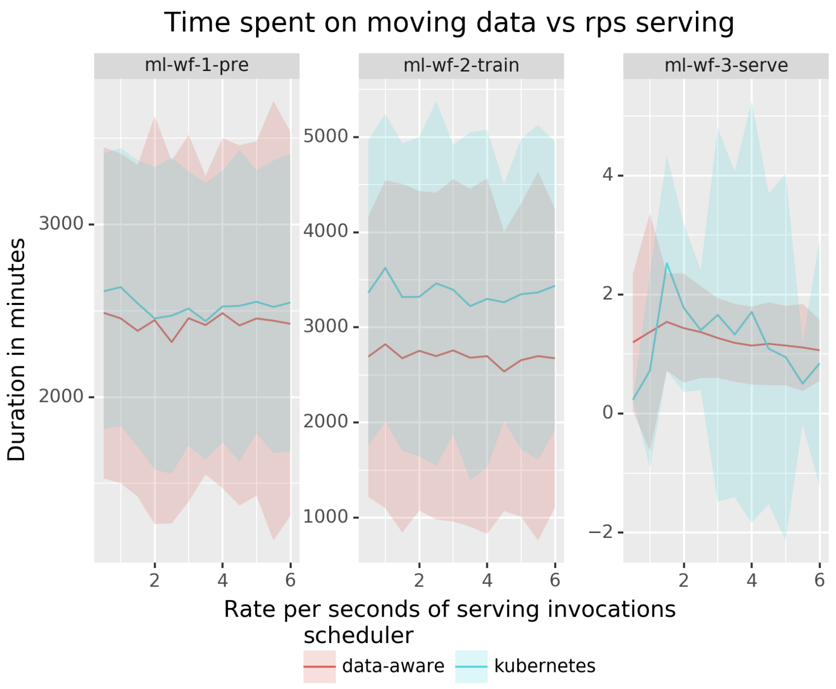

Figure 7.11: Duration spend moving data for the image times

Figure 7.11 shows the time spend on data movement for each of the image types. The duration considers both upload and download time. Serving uses cache, therefore only moves data on the creation of the function replica, for both the model and required images.

First the pre-processing images, it can be seen that both schedulers spend similar time moving data. Next training, the data locality aware scheduler spends less time moving data, but both

	p-value		
	Pre-processing	Training	Serving
Function execution time	**3.823e-21**	**1.644e-84**	**1.401e-08**
Function waiting time	**7.617e-36**	0.044	0.289
Function latency	**1.826e-36**	0.280	0.286
Network traffic count	**9.791e-47**	**7.227e-10**	0.225
Network traffic bytes	**1.424e-41**	**4.724e-33**	**1.509e-26**
Network traffic duration	0.051	**1.488e-12**	0.847

Table 7.2: p-value between the data-aware scheduler and Kubernetes scheduler

follow a similar spread. Looking at serving, it can be seen that data movement with the data locality aware scheduler has a lower standard deviation.

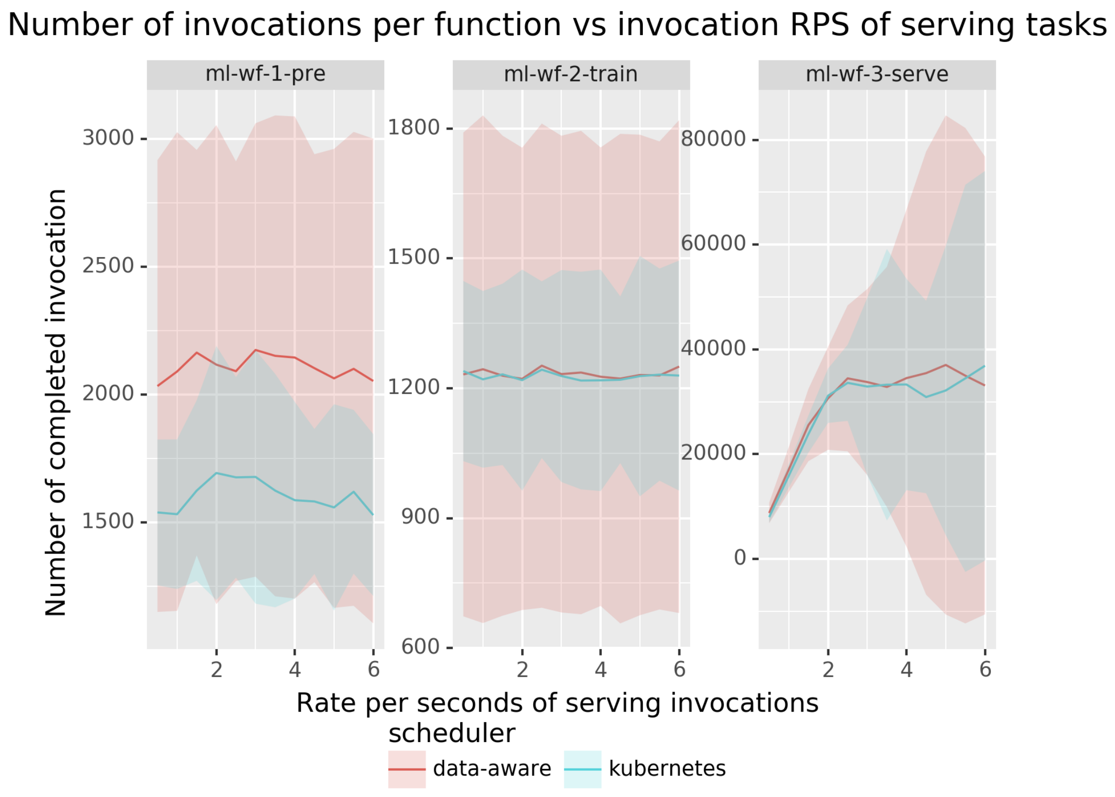

Figure 7.12: Invocation of the functions

Figure 7.12 shows the number of processed invocations for each of the image types. Overall the figure shows a higher standard deviation over all images types for the data locality aware scheduler. Both schedulers follow the same average trend of invocations for the training and serving images. For the pre-processing, the data locality aware scheduler handles more invocations on average per simulation run.

Figure 7.13 shows the mean utilisation of the memory and the CPU over all nodes in the system. It shows how the Kubernetes scheduler can scale to higher utilisation the the data locality aware scheduler.

Table 7.2 shows the p-values as a result of performing a t-test between the Kubernetes scheduler and the data locality aware scheduler with all nodes marked for constraint scheduling.

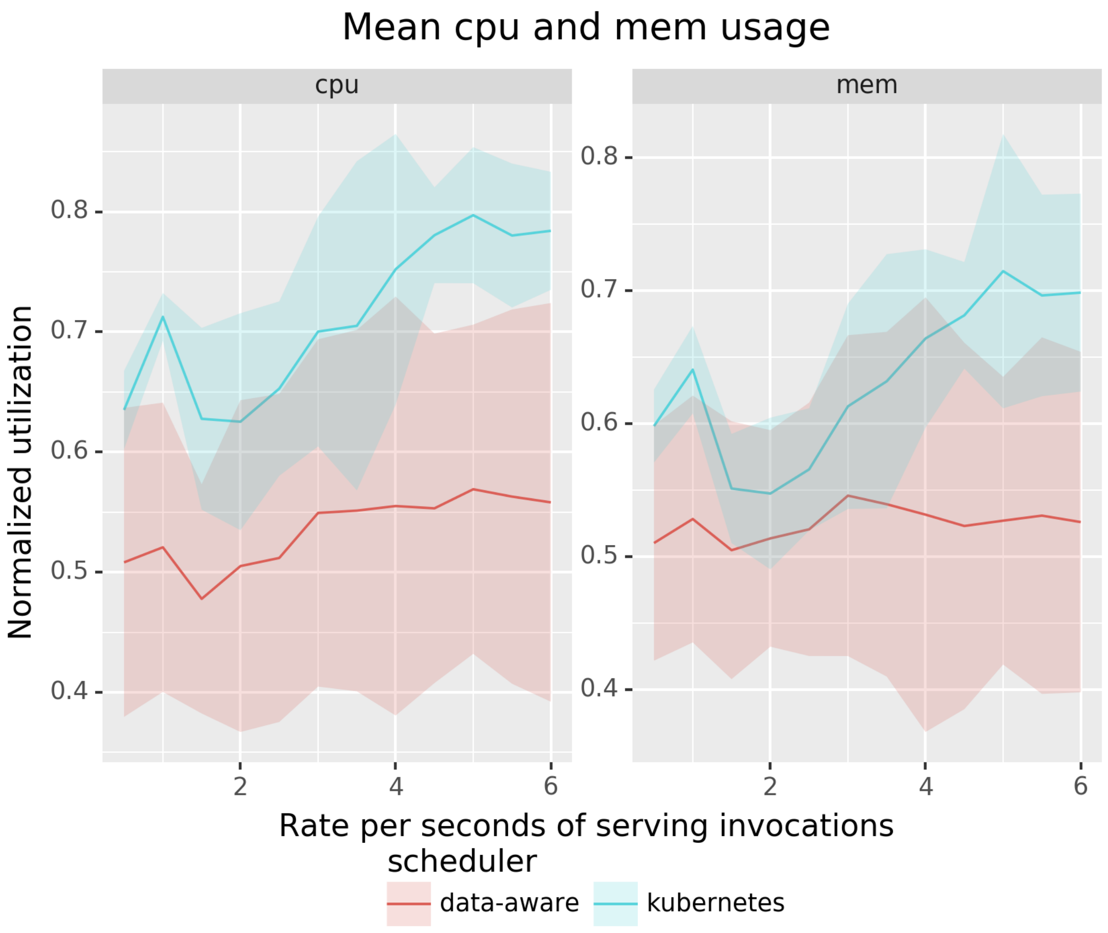

Figure 7.13: Utilisation of CPU and memory for the scheduler

Chapter 8

Discussion

This work looked at the serverless landscape and proposes a data locality aware scheduler. The scheduler is validated using a benchmark in a serverless edge simulation and the results were visualised to provide insight into the data.

8.1 Initial Survey

The initial survey has two results. Firstly, an overview of the serverless platforms and frameworks is provided. The overview focuses on literature relative to the data-intensive tasks in the serverless (edge) platforms. Secondly, a list of categorised features of serverless platforms is created.

In the first part, an overview of ten frameworks and platforms on the landscape of serverless an insight into the current state of the art of serverless computing. The overview considers hosted platforms, self-hosted platforms and platform extensions, including general-purpose platforms (e.g. AWS Lambda and OpenFaaS) and specialised platforms (e.g. Nuclio and Skippy).

The second part of the initial survey provides an overview and categorises the features available in serverless computing. The list illustrates the features found within the literature, defining core features, standard features and specialised features. The features of the existing field lead to the idea of data locality aware scheduling on a serverless edge platform.

8.2 Proposal Data Locality Aware Scheduling

The primary problem the data locality aware scheduler solves is lack of control on function execution location in respect to the data location. This raises concerns for privacy and bandwidth usage. The scheduler introduces a tagging method, that enforces placement within the premise of the data (see Figure 7.1). This can reduce bandwidth over the network and keeping the data private within the premise.

Looking more in-depth at the movements in and out premise (see Figure 7.1), it can be seen that even with all functions tagged there is a small marge of movements outside the premise, these movements are for downloading of the images from the container registry. As the image only needs to be downloaded once per node, an interesting extension could be a mechanism that first pulls images from local nodes instead of the registry at the cloud. The proposed tagging method allows developers to apply the scheduling on a function level, allowing them to handpick functions that require large amounts of data or have sensitive information and not limiting the entire serverless edge platform.

The performance of the system is impacted by the data locality aware scheduler. The number of possible target nodes are significantly reduced. When all functions are marked for locality aware scheduling the performance of the system seems to have limited impact, as some images execute faster and others slightly slower. With half of the function tagged for locality aware scheduling a greater impact is visible in the waiting times and execution times (see Figure 7.3 and Figure 7.4).

Likely this difference is due to the uniform distribution of data across all premises. In the case where all functions are scheduled with the data locality aware scheduler, the workload is evenly distributed across the nodes and thus local to the required data. In the 50% case, the distribution is sub optimal, as resources local to the data will host functions replicas that require data from other premises, reducing the availability of resources local to the data.

Figure 7.4 and Figure 7.8 show a high waiting time. The waiting time is the limiting factor in the latency to the user. The reason that the waiting time is so high is due to the current implementation of the function simulation and the selected scaling method. First, the scaling method looks at the average number of invocations queued in the function replicas and scales the system to match the threshold. Secondly, the current implementation of function simulation allows one function to be executed at once. This results in the waiting time being as long as the execution time multiplied by the threshold queue length for the function. In the case of the experiment, this is set to five for training and pre-processing and ten for serving, causing the system to aim for a relatively high waiting time.

The final thing to consider in the results is the structure of the network and the available speeds. The scenario used considers local networks in which every node is directly connected to the switch, and all nodes have a good connection speed within the local network and to the cloud. When both the network topology and bandwidth speeds are sub optimal, the impact of moving the data increases. This makes it more likely to benefit from the data locality aware scheduler.

8.3 Performance impact data locality aware scheduling

This section states the finding on performance impact in the results of the simulation. The following metrics are considered: function execution time, function waiting time, bandwidth usage out of the premise, bandwidth cost, number of invocations handled and utilisation.

Looking at all metrics, it shows that the metrics of pre-processing and train function replicas are not significantly influenced by the scaling of the serving function replicas. This is likely because of the relatively stable workload of the pre-processing and training functions, especially in combination with a an theshold queue size of five invocations. This keeps the same function replicas active during the simulation and not gives the scaling of serving replicas no chance to interfere with their replica placement.

For the following results it should be considered that the data is uniformly distributed, which fits well to a scenario with all the functions being scheduled using the data locality aware scheduler, as the workload is evenly distributed over the premises.

8.3.1 Functions Execution Time

Scheduling within the premise of the data significantly reduces the execution time for pre-processing images and increases the execution time for both training and serving (see Figure 7.3 and Figure 7.7). Pre-processing and training both move quite similar amounts of data. The main difference is that the training function supports GPU acceleration, allowing training to achieve to execute faster on a node with a GPU available. Considering that time spend moving data (see Figure 7.11) is lower and the number of invocations (see Figure 7.12) is higher for the data locality aware scheduler, thus data locality aware scheduling reduces the average time spend moving data during the execution for both pre-processing and training. The Kubernetes scheduler gets the speed-up elsewhere, the most likely explanation is that the Kubernetes scheduler is less restricted in placement and therefore schedules on a node with GPU more consistently.

The serving execution time reduces as the system becomes more saturated with replica's (see Figure 7.7). Serving replicas caches the model on startup and therefore each replica only download the model once, with an increasing number of invocations in the queue (see Figure 7.8) and less fluctuation in scaling, less replica startup and thus movements have to be made per invocation (see Figure 7.10).

8.3.2 Function waiting time

In waiting time there are two aspects (see Figure 7.4 and Figure 7.8). First, the waiting time of serving increases rapidly as the network does not have the required resource to support to this extend.

Secondly, the waiting time of the data locality aware scheduler is significantly lower than that of Kubernetes except for the training function (see Figure 7.4). The reduction in waiting time is due to the scaling mechanism being based on a queue threshold. The function execution time is lower and so will the waiting time reduce respectively (see Figure 7.7). For training the Kubernetes scheduler has a better execution time, this also maps to the waiting time.

8.3.3 Utilisation

The utilisation of the system is significantly impacted by the limitations on scheduling (see Figure 7.5). The Kubernetes scheduler has more resources available for scheduling functions, compared to the data locality aware scheduler. It can schedule on the entire system and thus scale further, however for functions with more data the less optimal placements is slower (see Figure 7.7).

8.3.4 Throughput

For throughput, the number of processed invocations is considered (Figure 7.12). The throughput shows that data locality aware scheduling causes a large standard deviation and thus handle a less predictable number of invocations.

The only image performing different between the schedulers is pre-processing. With the reduced function execution time and thus waiting time, the pre-processing replica's can handle more invocation during the run of the simulation. Supporting the point that functions with high data movement benefit most of the data locality aware scheduling in performance.

An interesting observation is that both methods reach similar throughput even with a significant difference in utilisation of resources (see Figure 7.5). Indicating the data locality aware scheduling can handle more computing on limited resources. Especially the pre-processing image contributed to this as the time spent moving the data is reduced significantly (see Figure 7.11), which resulted in lower function execution time (see Figure 7.7) and thus more invocations on the available resources.

8.3.5 Bandwidth

One of the main goals of data locality aware scheduling is to reduce bandwidth usage. Scheduling nodes on local nodes on-premise reduces the number of intermediate steps data needs to make to get to its destination. Moreover, scheduling on nodes with the required data can remove the need to transfer data over the network, reducing the cost.

The data locality aware scheduler can move more data in the span of the simulation (see Figure 7.10). Combining this with the fact that fewer intermediate hops are made by the data locality aware scheduler to reach its target destination, as the movement out of a premise has two of movements as in premise (see Figure 7.1) . This shows that the data locality aware scheduler moves less data on average.

Due to the ability of data locality aware scheduling, functions can be limited to only allow scheduling on-premise of the data. This reduces bandwidth cost to transfer data via the internet and cloud to only the bandwidth usage for downloading the image. As mentioned earlier, the data locality aware scheduling prevents data to leave the premise (see Figure 7.1)

8.3.5.1 Cost

Bandwidth cost increases with the amount of data transferred. As a result of data locality aware scheduling, the data usage can be limited to only transferring images (see Figure 7.1). Resulting in a cost reduction of roughly the same amount, varying on volume size and step costs

One of the advantages of data locality aware scheduling is the utilisation of domain knowledge or system monitoring information to achieve optimal scheduling decisions, to target functions that have a large bandwidth usage system or for privacy concerns.

8.4 Limitations

In this section known limitations of the simulation and experiment are discussed.

The simulated workload of serving is to much for the simulated network. function waiting time rapidly increases with an increase of invocation rate (see Figure 7.8). This type of load may not be representative of the actual workload especially with the limitation on available resources. By also using other workloads, it can help to see how the system behaves when more down and up-scaling occurs. An option would be by providing generating the invocation following a sinusoid pattern.

The simulation allows only scheduling of function replicas that can fit when using all there possible required resources on the pod, preventing the simulation from using more resources than available. A more complex approach would be to allow placement over resource limit and apply a degradation model to simulate the performance interference between the competing resources. The experiment is only run on a simulation and not a real testbed. The distributions used for the images are based on values of a real testbed, but the data locality aware scheduling is not tested in a real-world scenario. Performing a real testbed scenario could be a good addition that would help support this work.

The network used in the simulation is relatively simple, fast and static. All wired connections are 1Gb and all nodes are directly connected to the network and internet via a switch. This network is sufficient for showing the working of the data locality aware scheduling and giving general insight into the impact, but in future work, it can be interesting to extend both the scheduling as the complexity of the network. A few examples: (a) Nested networks without direct connections to storage or access points; (b) dynamic graph and meta-data; (c) Topology with slower a wider range of bandwidth speeds on edge devices.

Chapter 9

Conclusion

This work first presents an overview of serverless platforms and frameworks. Followed by the definition of a method for data locality aware scheduling of functions in a serverless edge platform. The scheduler is validated on a simulation that runs data-intensive workload on a serverless edge platform based on Kubernetes and OpenFaaS. In the next paragraphs, each of the research questions will be concluded.

The first research question is as follows: **What is the current landscape of data-intensive serverless computing?** To answer this question chapter 4 gives an overview of (self-hosted) serverless platforms and frameworks, with each a short description elaborating their limitations and benefits. The features found are then grouped in categories. The categories provide a clear list of functionalities that either are must be, should or could be part of a serverless framework. This overview provides a good overview of ideas in the existing work and the state of the art. Especially the could features show the current interests and state of the art. Several works looked at data movement as a limiting factor in serverless computing and suggest custom storage's, data pre-fetching and other scheduling constraints. The other questions in this work explores a new approach for scheduling, extending on some of the found features.

The second research question was **How can data-locality aware scheduling be achieved in a serverless edge platform?** Data-locality aware scheduling is achieved by constraining the scheduler to new meta-data (e.g. local storage's and bucket locations) extracted from the network graph of the system. The proposed method support function level control to enable data locality aware scheduling. The proposed scheduler is validated using a simulation of a serverless edge platform based on Kubernetes and OpenFaaS. The evaluation shows that data movement in and out of premises is reduced to the required imaged for the function replica. This in combination with fine-grained control allows the developers to choose to mark a function as default or be enforce a hard constraint on its scheduling location, allowing developers the choice for what function handle sensitive data or have high data requirements. Effectively reducing bandwidth usage and increasing privacy.

The third research question extends on the previous research question and looks at the impact on the performance of adding a hard constraint on data locality. The research question: **What is the performance impacted by data-locality aware scheduling?** A comparison of metric between the Kubernetes scheduler and the data locality aware scheduler shows several interesting points. (a) Function execution time is reduced when movement of large files is required (e.g. preprocessing), restricting date movement reduces time spend on movement thus execution time. (b) Function execution time becomes less predictable in a system partially using data locality aware scheduling, resources available can be in use be default functions, causing deployment on sub optimal nodes within the local network. (c) Resources are can be used more efficiently, the data locality aware scheduler can achieve similar throughput with fewer resources utilised, by spending less time on data movement. This assumes that the data is uniformly distributed on the network. (d) Bandwidth usage of a function can be limited to stay within the local network and thus reduce the cost of data movement over the cloud.

Overall it showed scheduling with function level control, which prevents data to be moved out of the premise for the specified functions. This keeps sensitive data local and reduces bandwidth usage over the cloud. In the case of large data movement, it even can improve performance.

9.1 Future Work

This work showed data locality aware scheduling in a simulation environment, in the discussion several points were mentioned that can be done in future work.

Moving the scheduler out of the simulation to a real testbed and validating its performance, with this the meta-data needs to be acquired using from the network as well as parsing of the tags to check if it needs to be scheduled within the premise of the data. This will result in a deployable prototype.

Improving the benchmark and simulation, by including performance degradation model, topology with higher depth of nodes, different workload patterns (e.g. sinusoid) and non-uniform data distribution. These aspects can provide better insight into when to apply data locality aware scheduling and are a better base benchmark for other simulations.

Extending on the idea of a more efficient scheduler for online scheduling in a serverless edge system, the data locality ware scheduler could be combined with other priority functions or offloading strategies.

Bibliography

[1] 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*, ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Wo L Chang, Nancy Grady, et al. Nist big data interoperability framework: Volume 1, big data definitions. Technical report, 2015.

[3] Tony Hey, Stewart Tansley, and Kristin M Tolle. Jim gray on escience: a transformed scientific method.

[4] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451, 2017.

[5] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

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

[7] Data protection in the eu. https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en, Jun 2021.

[8] Arash Nourian and Muthucumaru Maheswaran. Privacy and security requirements of data intensive computing in clouds. In *Handbook of data intensive computing*, pages 501–518. Springer, 2011.

[9] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[10] S. Allen, C. Aniszczyk, C. Arimura, B. Browning, L. Calcote, A. Chaudhry, D. Davis, L. Fourie, A. Gulli, Y. Haviv, D. Krook, O. Nissan-Messing, C. Munns, K. Owens, M. Peek, and C. Zhang. Cncf-wg serverless whitepaper v1.0. 2018.

[11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2020.

[12] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[13] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[14] Amazon. https://aws.amazon.com/serverless/.

[15] Abhinav Jangda, Donald Pinckney, Samuel Baxter, Breanna Devore-McDonald, Joseph Spitzer, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *CoRR*, abs/1902.05870, 2019.

[16] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4, 2017.

[17] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, USA, 2011.

[18] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *CoRR*, abs/2008.11110, 2020.

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

[20] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394, 2015.

[21] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, pages 41–54, 2019.

[22] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. Servermix: Tradeoffs and challenges of serverless data analytics. *CoRR*, abs/1907.11465, 2019.

[23] 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.

[24] Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/, Mar 2021.

[25] OpenFaaS. Stack. https://docs.openfaas.com/architecture/stack/.

[26] Philipp Alexander Raith. *Container Scheduling on Heterogeneous Clusters using Machine Learning-based Workload Characterization.* PhD thesis, Wien, 2021.

[27] Alexander Rashed. *Optimized container scheduling for serverless edge computing.* PhD thesis, Wien, 2019.

[28] Thomas Raush. edgerun/faas-sim. https://github.com/edgerun/faas-sim.

[29] Microsoft. Documentatie voor azure functions. https://docs.microsoft.com/nl-nl/azure/azure-functions/.

[30] Iguazio. Nuclio documentation. https://nuclio.io/docs/latest/concepts/architecture/.

[31] Comparing nuclio and aws lambda. https://theburningmonk.com/2019/04/comparing-nuclio-and-aws-lambda/.

Appendix A: Function distributions

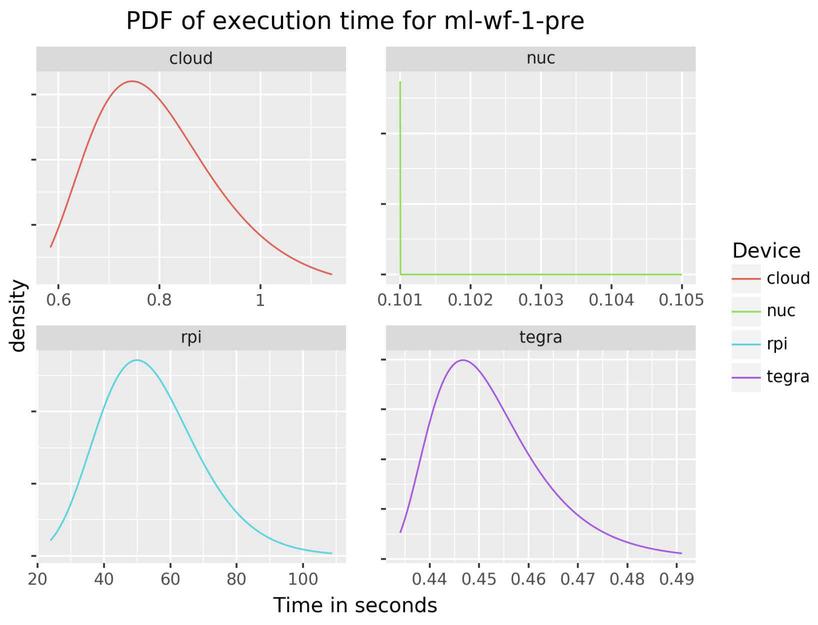

Figure 1: Distributions of execution time for pre-processing image

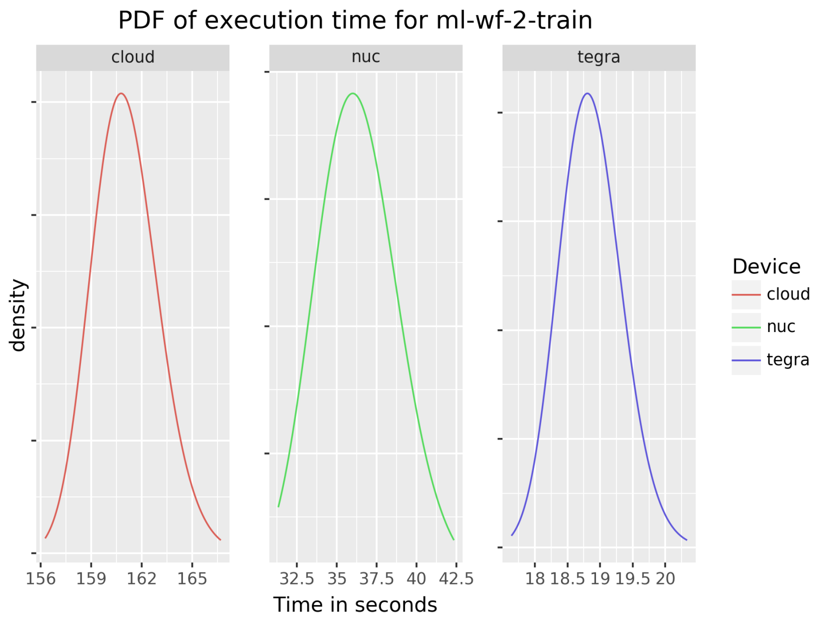

Figure 2: Distributions of execution time for training image

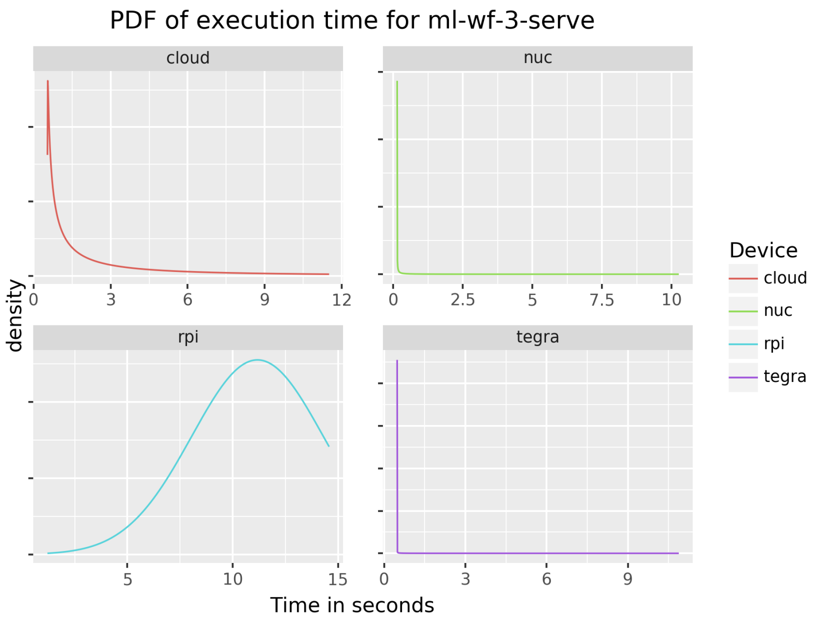

Figure 3: Distributions of execution time for serving image

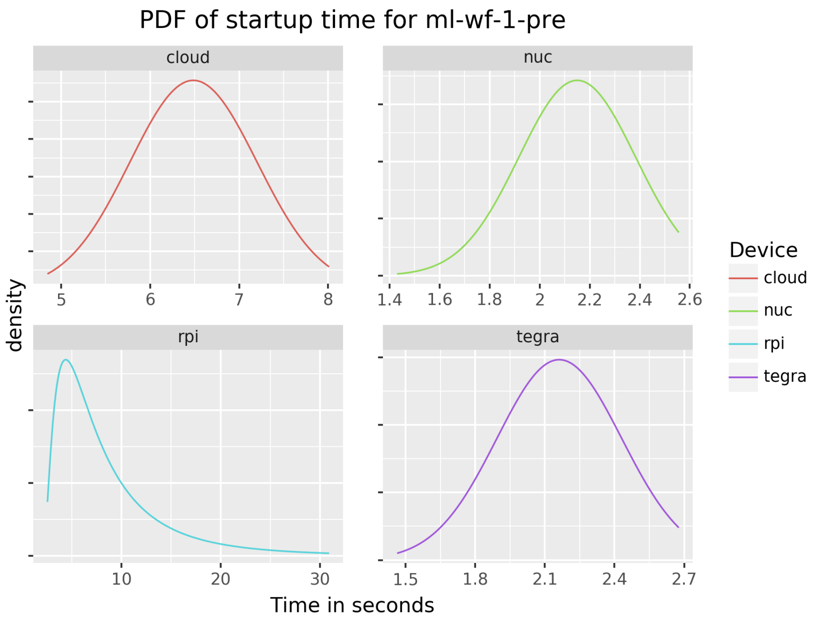

Figure 4: Distributions of startup time for pre-processing image

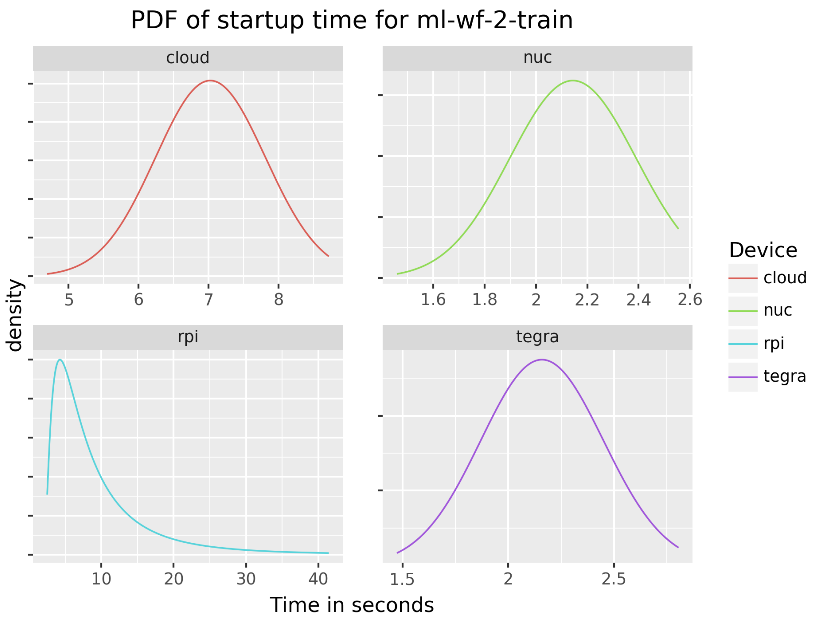

Figure 5: Distributions of startup time for training image

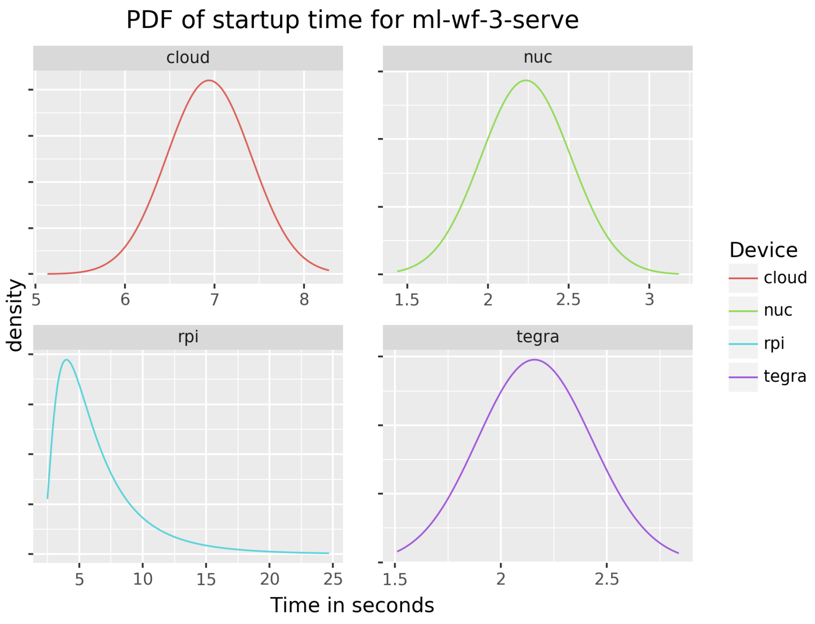

Figure 6: Distributions of startup time for serving image