Informatics

Data Locality-Aware Scheduling for Serverless Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Cynthia Kenia Arcanjo Marcelino, BSc
Matrikelnummer 01529611

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 19. August 2021

Cynthia Kenia Arcanjo Marcelino	Schahram Dustdar

Informatics

Data Locality-Aware Scheduling for Serverless Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Cynthia Kenia Arcanjo Marcelino, BSc
Registration Number 01529611

to the Faculty of Informatics

at the TU Wien

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

Vienna, 19th August, 2021

Cynthia Kenia Arcanjo Marcelino   Schahram Dustdar

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

Erklärung zur Verfassung der Arbeit

Cynthia Kenia Arcanjo Marcelino, BSc

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

Wien, 19. August 2021

Cynthia Kenia Arcanjo Marcelino

Acknowledgements

First, I would like to thank Professor Dr. Schahram Dustdar for giving me this opportunity. To my co-advisor, Dr. Thomas Rausch, I would like to emphasize how important it was to have you guiding me through this path. I wish I could say that pursuing a technical career is equally challenging for all of us, but I wouldn't be lying only to you; I would also be lying to myself. Nevertheless, I am happy to say that this was one of the few times I did not feel the treatment difference for being a woman. Thank you for your support and for being an ally in this equality fight. To my friend, Peter Vasil, thank you for the constant technical discussions with and without a beer; I am happy I have the opportunity to learn something new with you every day. To my life partner, Agnes Poks, I would like to thank you for the RaspberryPi's sponsorship. I will be forever grateful for your infinite patience and your daily support. Thank you for always being by my side; I could never have done this without you. I want to extend these acknowledgments outside of the technical area, every thesis has a story behind it, and I feel many people helped me achieve this. Thank you to Julia Ehgartner and Theresa Gwiggner, who helped me integrate and feel like I have a family away from home. To the sister that life has given me, Dr. Tainá Almeida, I want to thank you for the researching tips and for being present at all times when I need you.

The path until here was not easy, but the word easy was never in my vocabulary, and last but not least, I would like to thank my parents, who taught me never to give up. Father and Mother, I know your path was not easy; I recognize how much you had to sacrifice so that I can be here today. Both children of farmers who never had the opportunity to study. Mom, I know the effort and sacrifices you had to go through to finish school. I know that your dream was to go to university and that you only managed to get into college when we were grown up. For me, it was an honor to enter college at the same time as you. I will always be grateful for every penny counted so that I could take an English course. Dad, you are my hero; even though you didn't have the opportunity to study, you made an effort to provide a way for your children to be able to. Father and Mother, thank you so much for always working hard for us to have the opportunity you never had. Mom, I know that your dream continued in an academic career, but unfortunately, life didn't allow it. I just wanted to tell you that I will forget everything you sacrificed so that I could be here today. I will be forever grateful for all this. This diploma may even have my name printed on it, but it belongs to both of you.

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao Professor Dr. Schahram Dustdar por me dar esta oportunidade. Ao meu co-orientador, Dr. Thomas Rausch, gostaria de enfatizar a importância de ter você me guiando por esse caminho. Eu gostaria de poder dizer que seguir uma carreira técnica é igualmente desafiador para todos nós, mas eu não estaria mentindo apenas para você, eu também estaria mentindo para mim mesmo. No entanto, fico feliz em dizer que essa foi uma das poucas vezes em que não senti a diferença de tratamento por ser mulher. Obrigada pelo seu apoio e por ser um aliado nesta luta pela igualdade. Peter Vasil, obrigada pelas constantes discussões técnicas com e sem cerveja; fico feliz por ter a oportunidade de aprender algo novo com você todos os dias. À minha companheira, Agnes Poks, gostaria de agradecer pelo patrocínio das RaspberryPi's. Quero que saiba que serei eternamente grata por sua infinita paciência e seu apoio diário. Obrigada por estar sempre ao meu lado; eu não poderia ter feito isso sem você. À Julia Ehgartner e Theresa Gwiggner que me ajudaram a me integrar e a sentir que tenho uma família longe de casa. À irmã que a vida me deu, Dra. Tainá Almeida, quero agradecer pelas dicas de pesquisa e por estar presente em todos os momentos quando eu preciso.

O caminho até aqui não foi fácil, mas a palavra fácil nunca fez parte do meu vocabulário e, por último e mais importante, gostaria de agradecer aos meus pais e minha familia, que me ensinaram a nunca desistir. Pai e mãe, eu sei que o caminho de vocês não foi fácil, eu reconheço o quanto vocês tiveram que sacrificar para que eu possa estar aqui hoje. Ambos filhos de agricultores que nunca tiveram oportunidade para estudar. Mãe, eu sei o esforço e os sacrifícios que você teve que passar para conseguir terminar o ensino médio, eu sei que teu sonho era cursar uma universidade e que você só conseguiu entrar na faculdade quando teus filhos já estavam grande o suficiente, para mim foi uma honra entrar na faculdade ao mesmo tempo que você. Eu serei sempre grata a cada centavo contado para que eu pudesse fazer um curso de inglês. Pai, você é meu herói, mesmo não tendo oportunidade para estudar, você se esforçou para proporcionar um meio para que seus filhos pudessem. Pai e mãe muito obrigada por vocês sempre se esforçarem para que nós tivéssemos a oportunidade que vocês nunca tiveram. Mãe eu sei que teu sonho era ter continuado na carreira acadêmica, mas infelizmente a vida não permitiu. Eu só queria dizer-lhes que em nenhum momento me esquecerei de tudo que vocês sacrificaram para que eu pudesse estar aqui hoje. Eu serei eternamente grata por tudo isso. Este diploma pode até ter meu nome impresso, mas ele pertence a vocês dois.

Kurzfassung

Das vernetzte Zeitalter, in dem Milliarden von Geräten Echtzeitdaten sammeln und verarbeiten, erfordert Anpassungen in der aktuellen Technologieinfrastruktur, um die gesammelten Daten zu verarbeiten. Cloud Computing wurde durch sein On-Demand-Geschäftsmodell bekannt, welches Flexibilität und Skalierbarkeit durch wenige Klicks verspricht. Dennoch kann der hohe Kommunikationsaufwand und Datenaustausch zwischen Geräten mit geringeren Rechenressourcen und den Cloud-Diensten zu hohen Latenzen und finanzielle Kosten führen. Eine Lösung dafür bietet Edge Computing, welches die Datenverarbeitung von der Cloud in das Edge-Netzwerk verlagert. Edge Computing verwendet die Ressourcen von Endgeräten, um Echtzeitdatenverarbeitung zu ermöglichen. Edge Computing erzeugt jedoch neue Herausforderungen, zum Beispiel beim platzieren von Funktionen, da Geräte heterogene und begrenzte Rechenkapazitäten aufweisen. Edge Computing systeme basieren häufig auf Container-Management-Tools wie Kubernetes, um Funktionen im Cluster zu verteilen und eine Überlastung der Ressourcen zu vermeiden. Diese Orchestrierungs-Tools sind nur bedingt in der Lage, die Ressourcen der Geräte an die unterschiedlichen Workload-Anforderungen anzupassen. Um diese Einschränkungen zu überwinden, konzentrieren sich mehrere Untersuchungen auf die Identifizierung spezieller Gerätefähigkeiten und der Nutzung dieser Informationen zum optimalen Scheduling der Arbeitslast. Obwohl solche Scheduler die Geräte- und Netzwerkauslastung erheblich verbessern, indem sie die Characteristiker der Workloads mit den Ressourcen der Geräte, wie beispielsweise vorhandener GPU Beschleunigern abgleichen, bestehen nach wie vor Herausforderungen beim platzieren der Funktionen basierend auf der Datenlokalität.

Daher präsentieren wir in dieser Diplomarbeit eine Erweiterung eines Orchestrierungs-Tools-Schedulers mit der Berücksichtigung von Datenlokalität. Um dies zu erreichen, haben wir (1) einen Speicherindex mit den Metadaten der im Cluster vorhandenen Datein und (2) ein Netzwerküberwachungs-Tool entwickelt, das einen Echtzeit-Verfügbarkeits-Bandbreitengraphen bereitstellt. Zusätzlich, erweitern wir (3) den Kubernetes Skippy Scheduler mit einem Datenlokalitäts-Feature. Resultierend daraus, kann der Scheduler den Bandbreitengraphen zwischen Nodes und Speicherindex in seinem Service-Placement-Scheduling-Prozess verwenden. Da sich die Netzwerkauslastung zur Laufzeit nach dem Scheduling dynamisch verhält, haben wir außerdem (4) ein Framework eingeführt, um den kürzesten Weg für eine Dateiübertragung während der Laufzeit der Serverless-

Funktionsausführung zu identifizieren. Unser Lösungsansatz fügt Datenlokalität während der Platzierung und Laufzeit von Serverless-Funktion hinzu.

Unsere experimente zeigen, dass die Berücksichtigung der Datenlokalität die Ausführungszeit von Serverless-Funktionen um bis zu 40% verbessert. Unser Framework priorisiert Dateiübertragungen in Edge-Netzwerken, was zu nahezu doppelt so viel Edge-Netzwerkverkehr führt. Folglich verringert die Berücksichtigung der Datenlokalität die Ein- und Ausgänge des Cloud-Netzwerkverkehrs erheblich. Die Verteilung des Netzwerkverkehrs basierend auf der Verfügbarkeit von Edge-Ressourcen reduziert die finanziellen Kosten mit Cloud-Diensten um bis zu 85% im Vergleich zu Lösungen ohne Datenlokalität.

Abstract

The connected era in which billions of devices collect and process real-time data demands adjustments in the current technology infrastructure to process the collected data. Cloud computing introduced an on-demand business model, which enabled flexibility and scalability a few clicks away. Nevertheless, the constant communication and data exchange between low computational resource devices and cloud services may lead to high latency and financial costs with cloud resources. Thus, edge computing emerged, shifting the data processing from the cloud to the edge network. Edge computing leverages container orchestration tools such as Kubernetes to distribute functions across the cluster according to the devices' resources. These orchestration tools have limitations to match devices' capabilities with different workload requirements. Although new schedulers emerge to improve the workload by matching workload with devices' capabilities such as video acceleration, they still struggle with function placement based on data locality. Data-intensive workloads can profit from edge network proximity and data-locality awareness to improve latency and bandwidth usage. Additionally, when the data processing is closer to its source, the data can be processed using edge resources, decreasing latency, bandwidth usage, and avoiding additional financial costs with cloud resources.

Therefore, we propose in this thesis a data-locality enhancement for a container orchestration scheduler. To achieve that, we create (1) a storage index containing the file's metadata and (2) a network monitoring tool to provide a real-time availability bandwidth graph. Further, we introduce (3) a data-locality functionality on the Kubernetes Skippy Scheduler. As a result, the scheduler can use the bandwidth graph between nodes and storage index in its service placement scheduling process. Additionally, as the network usage may differ between scheduling and runtime, we introduce (4) a framework to identify the shortest route for a file transfer during the serverless function execution runtime. Thus, our proposed solution adds data locality during the serverless function placement and runtime.

Our experiments show data locality-aware scheduling improves the function execution time up to 40%. Our framework prioritizes file transfers on edge networks, leading to nearly twice as much edge network traffic. Consequently, the data locality-aware scheduling decreases the ingress and egress of cloud network traffic significantly. The network traffic distribution based on edge resources availability reduces the financial costs with cloud services up to 85% compared to solutions without data locality.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Edge computing is the terminology that describes a new paradigm where the processing is placed close to its data producer on the edge of the network. According to [1], edge network is populated with heterogeneous devices such as mobile, sensors, cloudlets, and data centers that produce data and process computational tasks. Different from cloud networks that present unlimited computational resources, there is a heterogeneous devices infrastructure composed of low processing capabilities such as sensors and mobiles in the edge network.

The increase of Internet of Things (IoT) devices in people's daily routine brought many questions and challenges about workload distribution on limited computational resources such as edge systems. Every IoT device collects a large amount of data that needs to be processed either on the device or the edge or cloud network. When the data is transferred and processed in the cloud, the processing relies on unlimited infrastructure resources. However, the use of cloud resources may lead to high latency and increased financial costs. Edge computing shifts the processing closer to the data source. As the edge network is closer to the end device, data-intensive workflows can profit from this proximity to improve latency and bandwidth usage. Additionally, when the data processing is closer to where it is produced, the data can be processed using edge resources, decreasing the latency and avoiding cloud resources and additional financial costs. This data processing shift from the cloud to the edge network opened up an opportunity for new execution models to emerge [2].

Serverless computing has become a popular model for deploying applications because it enables the developer to deploy short-lived functions which work as standalone services [2]. To control and manage the deployments of these functions, Function-as-a-Service

(FaaS) platforms often leverage container orchestration systems like Kubernetes (K8s)[1]. In a heterogeneous device infrastructure presented by edge computing, it is a challenge to distribute computing tasks to have a balanced infrastructure. Therefore, the use of container-management systems is essential since it considers the devices' available resources to allocate functions effectively and closer to the end devices [3]. To control service placement in a heterogeneous infrastructure, systems like Kubernetes depend on a smart scheduling mechanism able to identify the edge network infrastructure and make decisions based on devices' capabilities such as Random-access Memory (RAM) and Central Processing Unit (CPU) [4].

In a serverless-edge-computing environment with a data-intensive workload, there is an issue with transferring information effectively without sending data back and forth when data is spread between cloud and edge storage. Regardless of where the ping-pong data flow is happening, either device-edge-cloud or device-edge-device, the latency increases significantly in proportion to the amount of data. Additionally, data-intense workloads may lead to high inter-network traffic causing network traffic overhead when processed on edge. If the data-intense workloads are processed on the cloud, they can profit from high-speed LAN in data centers; nevertheless, the use of cloud data centers may increase financial costs. Thus, during the function's deployment, data locality is essential in serverless edge computing since responses are faster according to the function location and the distance between the function and data. Therefore, function and data placement are critical elements to be evaluated during function scheduling [5].

1.2 Problem Statement

As an example, we consider a typical machine learning (ML) workflow scenario where the data that need to be processed are collected from cameras and stored in a cloud storage service such as Amazon Web Services (AWS) Simple Storage Service (S3)[2]. Taking into consideration that in a common ML workflow, the data needs to be preprocessed, trained, and the ML model evaluated and deployed. If we have an environment where part of the ML flow is executed on edge and part is executed on the cloud, for single end-to-end workflow execution, this data and model need to be transferred multiple times. In this case, by the end of our workflow execution, we will have an intensive data transfer scenario between edge and cloud where every ML step needs to download its input and upload its output [6]. If the ML flow is completely executed on the cloud, it might incur high financial costs and latency. On the other hand, a flow mainly executed on the edge benefits from the short distance and lower financial costs. However, the increase of data processing on edge result in inter-network traffic, which may debilitate the workflow.

Therefore, data locality awareness, bandwidth, and processing capabilities are essential for an edge-cloud scheduling system. A scheduling system with such characteristics can distribute the tasks on edge and the cloud according to its availability and data location.

[1]https://kubernetes.io
[2]https://aws.amazon.com/s3

In this scenario, we can observe four main challenges to be considered in data-intensive edge-cloud workflows as follows:

Service Placement It is a known obstacle in edge computing [7]. In a serverless-edge environment, the scheduler performs the function placement based on devices' and workload attributes. A data-locality-aware scheduler can improve the flow by predicting which node can quickly access a file based on bandwidth availability between node and storage. Thus, functions can be placed closer to their input data. By placing the function closer to its input data, we decrease the data traffic in the network and consequently latency.

Runtime Data Transfer During a request execution, the serverless function needs to transfer data from a storage node. In an environment where data-locality awareness is not provided, the data would be transferred from any random storage node. Considering that the nodes can be placed in different data centers, the flow can be improved by identifying where the data is located and what the network status and consequently transfer the necessary data from an optimal storage [8].

Data Index In a container-managed environment, the scheduler controls when and where the functions will be placed. In data intensive scenarios, Rausch et al. [9] have shown that system scalability and data throughput can be improved if the scheduler considers functions' input and output data. In order to provide data-locality, either during scheduling or runtime, the framework needs to have access to information about existing data in the cluster. Thus, it is necessary to create a data index which can be quickly accessible whenever necessary.

Bandwidth Graph The bandwidth graph is a data structure that stores the available bandwidth between nodes in the network. Data-locality leverages the network state knowledge to find the shortest route between two nodes. However, authors in [10] mention network monitoring as an issue in edge computing. Generally, precise monitoring creates an additional overhead in an already limited network. Thus, we consider lightweight solutions to generate a bandwidth graph that reflects the network traffic availability between the nodes in the cluster.

1.3 Research Questions

The research and implementation is this thesis are based on the following research questions:

RQ1: Which components are necessary to enable data-locality-aware scheduling in state-of-the-art container schedulers?

In a container-based system, scheduling is a crucial factor for balanced resource usage in the cluster. Container management systems available in the development community offer useful scheduling features such as CPU and RAM resource-based filtering and scoring. However, in a data-intense exchange scenario with limited computing resources like serverless edge computing, the scheduler needs extensive recognition to improve the running environment and consequently the scheduling mechanism. Rausch et al. present the Skippy Scheduler[9, 11]. Skippy is a custom Kubernetes scheduler that scans the edge infrastructure and makes decisions based on edge devices and workload. Besides computing resource capacity, Skippy is capable of identifying specific workflow characteristics like Graphics Processing Unit (GPU) and devices' locality like edge and cloud. However, Skippy does not consider data locality, which is crucial in data-intensive scenarios like serverless edge computing. To solve the data-locality problem during scheduling, we propose a Skippy enhancement to identify existing data on the storage nodes in the cluster and use data placement and network state as attributes during the scheduling decision process.

We propose to extend Skippy to recognize files used by the serverless function and schedule these new serverless functions by the file location in the cluster. The nodes which host data files are referred to as storage nodes throughout this thesis. As an example, if we say $functionA$ reads $fileA$ and this $fileA$ exists in multiple storage instances across the cluster like $storageNodeA$, $storageNodeB$ and $storageNodeC$. Our Skippy Data Scheduler should be able to identify which candidate host node has better bandwidth availability to either $storageNodeA$, $storageNodeB$ or $storageNodeC$. Our Skippy Data Scheduler should score the candidate nodes based on their capability to access the requested file in a storage node during the scheduling process. To achieve that, we propose additional components that provide (1) a data index with metadata of all files present in the storage nodes in the cluster. To provide the bandwidth availability between the nodes in the cluster, we propose a network monitoring tool that creates (2) a bandwidth graph of all nodes and storage nodes in the cluster. During the serverless function deployment, the user adds Kubernetes labels that identify the files necessary. Skippy Data Scheduler reads these file labels and accesses the data index to query the storage nodes which contain the requested file. Finally, the scheduler scores the candidate host nodes based on their bandwidth availability to access the requested file in a certain storage node. By the end of the scheduling process, the node with the highest score hosts the serverless function.

RQ1.1: what is the performance impact of providing data storage information to the scheduler?

Data-locality solutions that depend on file metadata have the challenge of low-latency data access. If a scheduler needs to know where the data is placed across the network, it is inefficient to search for the data during its scheduling time. On that account, we suggest a solution that identifies the existing data on the storage instances and stores this metadata in a *key/value* store, which can be accessed during scheduling and runtime. This solution creates a data index that works as a lookup mechanism to the existing data in the storage instances present in the network. To interact with the scheduler, we propose an independent component responsible for creating and updating the data index. This data component makes the data index available by storing the data index in a *key/value* caching system. Once the data index is stored in a caching system, it can be retrieved by the scheduler. We hypothesize that this approach allows low-latency access to the metadata information during scheduling time and runtime.

RQ1.2: What is the tradeoff for providing inter-node bandwidth information to the scheduler?

Network monitoring plays an important role in enabling data locality awareness. Existing solutions like *iperf*[3] deliver precise network metrics. However, such solutions add overhead to the network, which can be considered intrusive. In data-intense workloads where the network is already overloaded with constant transfers between device-edge-cloud networks, an additional load is not an option. To overcome this challenge, we propose two non-intrusive networking monitoring solutions to aid the data-locality process without adding extra network load.

We create a bandwidth graph between the storage nodes and devices present in the edge and cloud networks. In the bandwidth graph, $G = (V, E)$ where V stands for the vertices and represents the devices while E is the edge that connects the vertices. In our bandwidth graph, the edge graph E is represented by the link availability between the nodes in the network. Once the bandwidth graph is created, it can be low-latency accessed by the Skippy Data Scheduler or Skippy Data Software Development Kit (SDK) to support during the decision-making process.

We propose a network monitoring solution that collects metrics from the edge devices and creates an estimated bandwidth graph based on these metrics. We hypothesize that this approach allows efficient bandwidth graph creation without adding extra network overhead. However, we hypothesize that our bandwidth graph does not have the accuracy as intrusive tools. Therefore, our bandwidth graph might not have the same accuracy as intrusive tools, but it provides an overview of the current network usage. We consider the accuracy a tradeoff to have a lightweight bandwidth graph.

RQ2: What additional runtime mechanisms are needed by a serverless system that performs data locality-aware scheduling?

Scenarios like the ML workflow example described above, which require multiple storage data transfers, are challenging for data-locality solutions. The scheduler can improve the workflow by placing the function on a node close to its input data. However, whenever a serverless function is triggered, there is another decision about which storage to use during

[3]https://iperf.fr

runtime. As an example, when a *functionX* is placed on *nodeA*, during its execution *functionX* needs to identify the best storage to download or upload the data necessary for its processing during runtime. Since the network usage might quickly change, this decision should not be taken only during scheduling time. Therefore, we need a runtime solution that supports the scheduler in its data-locality mechanism. To address this challenge, we introduce a python library SDK which automatically downloads and uploads data to the storage. The SDK reinforce the data-locality decision taken by the scheduler. Nevertheless, the SDK still has the autonomy to identify network load and to make different decisions in case of environmental changes.

1.4 Solution Approach

In this thesis, we present a data-locality scheduler and a data-locality runtime SDK. We propose a network monitoring tool that provides bandwidth availability between the cluster nodes to support the decision-making process during scheduling and runtime. Additionally, we also create a data index containing the file's metadata from the storage nodes. As already described in the previous section, in data-intensive serverless edge computing scenarios, the data transfer between functions may result in high inter-network traffic and cause network congestion and low data throughput. This process can be faster and more efficient if the framework knows where the data is located and the current network traffic usage. In other words, data-intensive serverless scenarios can profits from data proximity when a function is placed closer to the data instead of using remote data storage centers [6].

To solve the data-locality for function placement, we propose to extend the Skippy scheduler presented in [11, 9]. By consulting the requested data in a data lookup search mechanism provided by the data index. Additionally, the scheduler uses the bandwidth graph, which provides an overview of the current network usage and its availability between the devices in the network. Further, the scheduler can score the cluster nodes according to the data index and bandwidth graph.

As a runtime solution, we propose a SDK to resolve the requests during the function execution. Once the function is placed on *nodeA*, it still needs to download and upload data during its execution. In case the data is replicated across several storage nodes in the network, the function still needs to be able to find the shortest path. Similar to the data locality in the scheduler, our proposed SDK uses the current bandwidth graph and low latency data index to identify the best storage node to be connected with.

To support this decision-making process during scheduling and runtime, there are two key factors: data index and bandwidth graph. To build a fast data index lookup mechanism, we need to scan files and storage in the network constantly. Once this information is accessible, we can build a data index and store it in a *key/value* store. Throughout this thesis, we refer to the data index as storage index. The second essential mechanism in our data-locality framework is the bandwidth graph. To find the shortest route between nodes A and B, we need to know the network status at an exact instant in time. To solve that, we build a network monitoring solution that uses non-intrusive techniques to overview the current network usage.

To evaluate the project, we create a testbed that runs our solution end-to-end. Additionally, we use a simulator to obtain the results of our scheduler when run in a cluster with a large number of devices.

1.5 Structure

The remainder of this thesis is structured as follows. In the current Chapter, we introduce the problem and propose a solution for it. Chapter 2 presents basic concepts necessary to understand the content of the thesis. Further in Chapter 3, we show similar approaches for some of the problems. Chapter 4 gives a high-level overview of the system and its components. In Chapter 5, we detailed how the scheduler works, which modifications and improvements are implemented. In Chapter 6, we continue to describe how the framework manages the metadata efficiently to have low latency access to it and consequently to speed up the decision-making process. Chapter 7 shows how we obtain the network metrics necessary to build a reliable bandwidth graph without intrusive network monitoring methods. In Chapter 8 we detail tests and experiments performed. The results show whether the solutions proposed are feasible or not. Finally, in Chapter 9, we conclude the thesis and present bottlenecks and possible continuation of the project.

CHAPTER 2

Fundamentals

In this chapter, we introduce the background necessary to understand the project. We give an overview of serverless edge computing and its main applications in the current market. Additionally, we present the advantages and challenges of the serverless edge computing approach.

2.1 Serverless Edge Computing

Cloud computing introduced new business models which enable unlimited computing resources upon request. It delivers fast and on-demand services like computer power, cloud storage, databases, and network services. The growth of IoT introduced a new scenario to cloud computing. The increase of low capability devices and its intense data transfer workflow created a scenario in which a large amount of data produced by these devices was constantly moved back and forth from edge and cloud [12]. Edge computing suggested the leverage of end devices' capabilities to enable fast processing [13].

The adoption of serverless edge computing provided a fast, scalable way to produce and process data close to its data source [14]. Serverless edge computing enables low latency, fast processing, and low bandwidth usage for applications that rely on data collections and analysis such as ML. Data intense workflow profits from the proximity between data source and data center, leading to a higher performance since processing and storage are also closer to the end device. However, its "pay-per-use" cloud-designed business model became financially challenging for mixed edge-cloud infrastructure with data-intensive workflows [15]. In [16, 17], the authors summarize how the costs to run a FaaS system like AWS Lambda increase drastically according to storage access, function duration, the number of invocations, and allocation of computational resources like RAM and CPU.

Serverless edge computing offers low latency for real-time applications. As an example, in a traffic control environment where cameras and sensors collect real-time traffic data, the system needs to react as fast as possible whenever necessary. In case of an emergency, e.g., ambulance or fire alarm, the traffic must respond immediately. Thus, the processing closer to the device enables low latency responses. Additionally, it also avoids unnecessary cloud data transfer hence, avoiding bandwidth overhead [18, 19].

2.1.1 Serverless Computing

According to [4], serverless computing is a new business model in which the developer has to take care of as little as possible of server maintenance. Serverless computing emerged with properties like event-driven and FaaS computing that fits in the edge computing infrastructure with simple tasks execution [20].

Using the characteristics of cloud computing, serverless computing allows users to scale their application's resources up and down depending on workflows' requirements. As the name already suggests, there is no need to maintain any server, virtual machine (VM), containers, or any infrastructure resources. Once a function is deployed on a specific platform, it can be increased or decreased depending on the necessity of each application [21]. However, as an execution model that hosts short-lived functions and do not do any data management, serverless computing presents data management obstacles with services that produce and process large amount of data. Thus, edge computing enables serverless computing to process the produced data closer and faster without constantly sending the data to the cloud [22].

FaaS presents an ability to be easily and quickly modified. It simplifies the complicated deployment process for the developer, thus providing the developer with more time to focus on programming tasks instead of operational ones. The simplified deployment and low maintenance are also influencing FaaS to become widely used in the edge computing topology [23]. As IoT devices and sensors present on the edge network have limited computational power, thus the workflow can profit more from the proximity with the data source if its task requirements do not require extensive computational resources.

2.1.2 Serverless Edge Computing Architecture

Fig. 2.1 shows an overview of a typical serverless edge computing architecture. The top layer represents a standard cloud infrastructure that offers on-demand scalable infrastructure and services. The intermediate layer illustrates the edge network. This middle layer represents the processing nodes and data centers closer to the devices. The bottom layer represents end-user client devices which often collect a large amount of data that needs to be processed and stored. In a typical serverless edge architecture, events are triggered by devices and processed on the edge or cloud network. The utilization of edge nodes' resources capabilities reduces cloud financial costs, and it increases latency. Additionally, edge computing includes similar cloud solutions such as storage and other processing services. Although the processing shift from cloud to edge network decreases cloud costs, it may lead to provisioning challenges due to the heavy utilization of its scarce computational power [24].

Although most of serverless edge computing platforms work with three layers as base architecture, the architecture details may differ according to the platform hosts like the cloud providers such as AWS Greengrass[1], Azure IoT Edge[2] or even the open sources like OpenWhiskey[3], Kubeless[4], Nuclio[5] and OpenFaaS[6] [25].

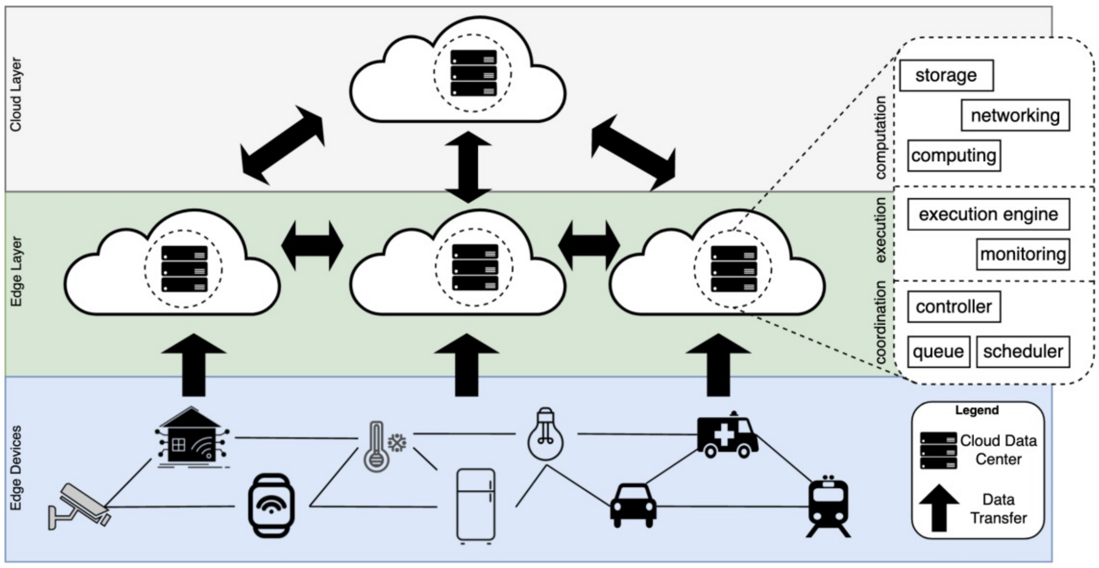


Figure 2.1: Serverless Edge Computing Architecture

As displayed in Fig. 2.1, an edge orchestration system is composed by computation, execution and coordination [14]. Scheduling is part of the coordination mechanism in serverless edge computing. In an orchestration system, the coordination is composed mainly by three functionalities: queue, controller and scheduler. Every incoming message arrives in *queue*, the *controller* validates the messages in the *queue*. Once the message is verified and the current scheduler is not overloaded, the message is taken by the *scheduler* which filters, scores and binds the message in a cluster node [14]. The scheduling process is explained in detail in Chapter 5.

2.1.3 Application Scenarios

Serverless edge computing is becoming popular among scenarios that present heterogeneous devices constellation and data-intensive workflows. The proximity between the processing center and data source is enabling low latency and real-time processing scenarios. As an example of its current usage, we can describe typical serverless edge

[1]https://aws.amazon.com/greengrass/
[2]https://azure.microsoft.com/en-us/services/iot-edge/
[3]https://openwhisk.apache.org/
[4]https://kubeless.io/
[5]https://nuclio.io/
[6]https://www.openfaas.com/

computing scenarios. The scenarios described below leverage the proximity to the data source to enable fast processing. Serverless edge computing provides the means to real-time processing close to the end user. As data-intense producers, the scenarios listed below would profit from our data-locality solution described in the Chapter 4 to place the functions close to the data, which leads to faster execution time and lower bandwidth usage.

Edge Intelligence

Edge intelligence refers to systems where edge nodes have the autonomy to orchestrate and distribute the tasks between themselves. Edge intelligent systems do not only produce and collect data but also react upon them efficiently [26]. Compared to typical edge cloud architecture where the data is transferred with minimal processing on edge nodes, edge intelligence brings advantages such as low latency and energy saving due to its proximity to the end device. It also provides scalability; an intelligent task distribution induces idle nodes to share tasks with currently overloaded nodes. Furthermore, in an intelligent edge system, the orchestration mechanism, which can identify its surrounding properties, can profit maximum from it. Once there is an edge overhead, tasks can be easily shifted to the cloud. This guarantees high availability and scalability [27]. Edge-intelligent systems can improve their advantages such as low latency and task distribution with a data-locality scheduling solution described in Chapter 5. The data-locality awareness provided during scheduling helps the edge intelligent systems better task distribution, leading to better results like low latency and energy saving.

Machine Learning

The rise of IoT and the large amount of data collected by sensors on the edge network is pushing the adoption of ML. ML allows the optimization of an application according to incoming analyzed data According to [28], a ML workflow consists basically in 1) pre-process the collected raw data 2) train raw data 3) evaluate and deploy trained model.

ML is a data driven approach used to teach applications how to perform tasks like classification, prediction and recognition [29]. In serverless edge computing, ML is being largely adopted due to the task functionalities previously described. In [30], the authors present an innovative cognitive assistance concept. Data collected in sensors around an urban area assist cyclists to predict cars on the next street even if they are still not visible to the human eye. The authors also affirm the key role of edge computing in such scenarios. In Chapter 8, we describe a typical ML workflow to evaluate this project. The experiments use a ML workflow to show the benefits of data-locality awareness during serverless edge function placement.

Smart City

More and more cities are converging on the smart city concept. A smart city scenario might include smart traffic control, smart parking, or any other urban area which uses end devices such as sensors to collect data. The collected data can be used for real-time processing and environment monitoring. Serverless edge computing enables the data collection from sensors around the city to be processed on edge data centers. The edge processing allows the applications to react according to the environment's information quickly. As an example, traffic lights might promptly respond to an accident or any other emergency case, or bikers can predict cars coming even when they are still not visible to the human eyes preventing accidents [31].

Industry 4.0

Modern factories use serverless edge computing to improve their production line. In [32], the authors introduce an interconnected and intelligent production line for a manufacturing company. The production line is composed of robots which collect data through sensors and cameras interconnected to the edge nodes. The IRobot production line is enforced by edge computing's ability to process data produced by the robots. Edge processing allows the robots to quickly act upon environmental change and dynamically auto reconfigure.

Mobile Edge Cloud

Mobile edge cloud is an emerging technology that enables interconnection between devices as in a home network. For example, in a smart home, the devices can communicate through a home network. In a mobile network, this is not possible since, in the mobile network, every device is isolated in the network. A conventional solution is to send the data to a cloud center, where all the devices have access. However, the increase of cloud traffic data leads to high bandwidth, latency, and high financial costs. Mobile edge computing connects stationary devices to a local area network facilitating the intercommunication between the devices [33]. In [34], the authors highlight the struggles of the current mobile cloud edge computing due to high latency and low-speed uplink and downlink. The ultra-high availability and ultra-low latency promised by 5G presents a new range of serverless edge scenarios such as immediate traffic reaction in connected cars on a highway in case of an accident or a real-time response for health monitoring solutions.

2.2 Scheduling Data Intensive Workloads

As data-intensive scenarios increase, schedulers are targeting the challenges faced by heterogeneous workloads in serverless edge computing. In [9], authors enumerate these technical challenges as follows:

- heterogeneous environment and workload: an edge cloud network is composed of multiple nodes which present different processing capabilities. Additionally, executed functions contain different resource necessities. An efficient scheduling tool should analyze what the environment offers and match with functions' requirements.

- locality: data might be locality sensitive e.g. some scenarios might require data to be processed next to the data consumer.

- latency: certain workloads require fast response. The process directly on the edge node reduces the latency by avoiding unnecessary cloud connections.

- bandwidth: constant data transfer between edge nodes intensifies overhead on the network. Regardless of cloud or edge, intense network traffic can congest the network and impact on the application's performance. Hence, to decrease the network traffic, the scheduler must know function's required data in order to favor the bandwidth during task and data placement.

2.2.1 Kubernetes

As stated in [4], serverless computing is generally adopted in a distributed landscape where the serverless functions are placed on different cluster devices. For this scheduling to happen, serverless computing relies on a framework that can facilitate and orchestrate the deployment and management of functions. Kubernetes is the open-source orchestration tool used in this thesis as described in Chapter 5. It provides flexibility and scalability for users to deploy applications at scale, it contains components which manage the three domains *coordination*, *execution* and *computation*.

As displayed in Fig. 2.2, Kubernetes has a main component control plane which acts in the three layers coordination, execution and computation. K8s control-plane is composed of four separated components: api-server, kube-scheduler, kube-control-manager and etcd. The api-server is responsible to *control* the incoming flow. It allows incoming pods from the *queue* to the *scheduler*. Kube-scheduler places pods in worker nodes according to pod's requirements and cluster environment while kube-controller-manager *monitors* the cluster status and it *stores* this information in etcd[7]. Furthermore, in each cluster worker there are three additional components to complete Kubernetes architecture: kube-proxy, kubelet and container-runtime. Kube-proxy is a *networking* tool responsible for forwarding incoming Hypertext Transfer Protocol (HTTP) requests to a specific pod inside a cluster node. Kubelet *execution engine* identifies pods assigned to itself, it pulls the container images if necessary and it starts a pod's container. The container-runtime is a third-part *computing* application necessary to be previously installed in order for Kubernetes to run the container. As containers leverage host computation resources, the container-runtime application is also responsible for resource allocation. Every running

[7]https://etcd.io

container allocates real host resources in order to prevent over provisioning on a specific worker. For this project, we have used Docker[8] as a container runtime application.

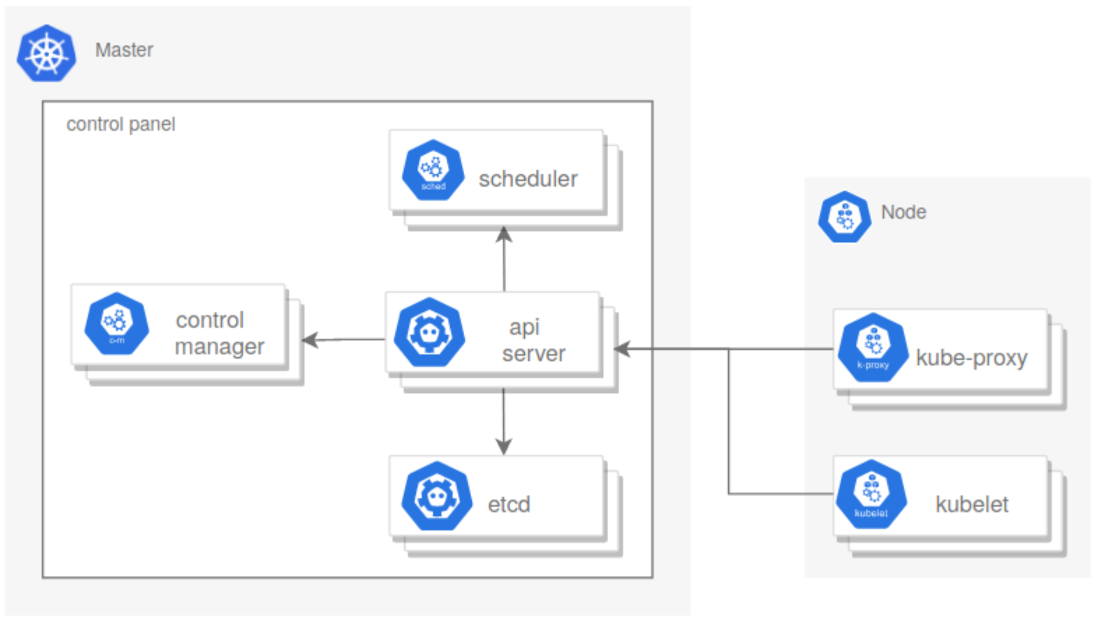


Figure 2.2: Kubernetes Control Panel

The Kubernetes scheduler's decision-making process is based on a sum of scores defined by priorities. The framework also allows the possibility to use another scheduler or even run it parallel with another one. Along with a scheduler, Kubernetes[1] also provide a kube-proxy which is responsible to forward the incoming requests to the least loaded worker that is able to execute that request. Details about Kubernetes mechanism and its scheduling process are further discussed in Chapter 5.

2.2.2 Scheduling Problem

The proposed solution described in the previous chapter extends the Skippy scheduler by implementing functionalities to address the data locality, data movement tradeoff, bandwidth, and proximity between the nodes. To achieve that, we proposed to create scheduler priorities for the existing Skippy scheduler. Chapter 5 explains how Skippy Data scheduler addresses the data locality problem. Nevertheless, to understand how the scheduling process works, we need to analyze the scheduling problem.

The Skippy scheduler assigns new pods to a node based on scores calculated by priority functions. In [9], authors describe this problem as $s \in S : P \times N \rightarrow \mathbb{R}$. In which S denotes a set of priorities, P denotes the set of pods awaiting scheduling and N is the domain of nodes. The Function *schedule* $P \rightarrow N$ selects the node by calling function

[8]https://www.docker.com/

score for each pod and for each node. Each of these priorities matches a pod's requirement to a node's ability to fulfil these requirements. After the score has been calculated, the node which best fulfils the pod's requirement presents the highest score which is then multiplied by its priority weight. The node which has the maximum sum of priorities' score is elected to host the pod as displayed in Eq. (2.1).

$$schedule(p) = arg \max_{n \in N} \sum_{i=0}^{|S|} \omega_i \cdot S_i(p, n) \tag{2.1}$$

As kubernetes weights ω_i every priority equally to 1, Skippy scheduler offers a weight value setup which can tune the priorities according to the workflow's necessity. Considering a heterogeneous workload, priorities can have different importance e.g. in a intense data scenario, data locality priority might have higher impact on the flow performance than computational resource priority while in a intense CPU based workflow, a bandwidth priority is not so relevant. Therefore, the optimized scheduler does not only calculate priorities score but it also supports weight values to match the workloads' requirement [9].

In [35], the authors present a algorithm complexity for the problem. Overall, the scheduling complexity depends on each priority function's specific implementation. However, if we dismiss this fact, we can consider the minimum scheduler complexity. In our context, if there are a number of incoming pods p and n nodes in the cluster, during the scheduling an incoming pod needs to check priorities s times, for all nodes at least once. Considering that in our case we have 7 priorities + 1 overall sum calculation, we can say for our scheduling algorithm $s = 8$. Therefore, our scheduling algorithm needs $p \times n \times 8$ which can be written as the algorithm's complexity of $O(p \times n)$.

2.3 Data Sharing at the Edge

In a typical serverless edge computing workflow, the data is shared over the network as displayed in figure 2.3. More specifically, data can be shared between the edge architecture layers as *edge* $\leftrightarrow$ *edge*, *edge* $\leftrightarrow$ *cloud* and *cloud* $\leftrightarrow$ *cloud*.

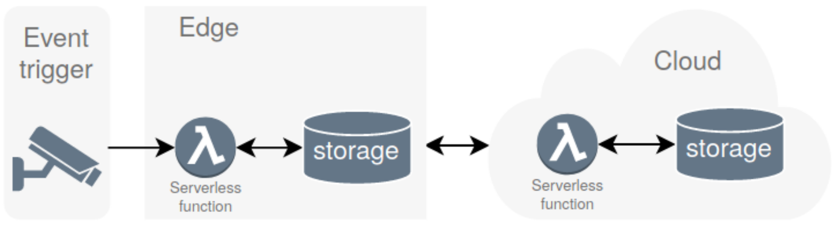


Figure 2.3: Serverless Function Workflow

Data sharing across the network is a necessary tradeoff to build web applications. The growth of real-time data producers at edge computing is pushing new solutions to improve data transfers between edge and cloud network [36]. Data-locality awareness solutions like Hadoop offers a MapReduce technique to locate the data. To schedule its jobs efficiently, Hadoop first identifies the files across the cluster. Once Hadoop is aware of the data, it distributes the tasks according to the data location [37]. Even though Hadoop has a data-locality awareness, it does not consider a heterogeneous environment, making the state-of-the-art Hadoop unsuitable for serverless edge computing.

In an attempt to overcome the intense data sharing bottleneck between edge networks, solutions propose approaches like data indexing [38]. Our solution described in Chapter 4 profits from a data index to quickly locate the data in the cluster and consequently decrease the latency and bandwidth usage. Our data management mechanism is described in Chapter 6.

2.3.1 Data Index Complexity

Data indexing is one of the available approaches to enable fast information access to data-intense serverless edge computing. In this technique, the knowledge about the storage environment is collected and stored to be accessible when necessary; this is often referred to as data locality. In [39], the author describes *full-index*, *central-index* and *Data Hashing Tables (DHT)* as approaches to keep the data index up to date in a data-intense workload scenario.

In a *full-index*, every node at the edge network keeps a copy of the storage nodes and their metadata. As every edge node keeps a full-index copy, this approach profits from the quick index access. However, it has drawbacks regarding scalability. It is hard to keep every node's full-index copy synchronized since the update needs to be broadcast to fellow nodes in every change. Besides, the constant information transfer to maintain the full index might add additional overhead on the network [40]. A *central-index* relies on a central server which keeps the index up to date. A central node responsible for a single task suffers the single-point-of-failure principle, which leads to performance and fault-tolerance drawbacks [41]. At the same time, DHT systems rely on a key-value lookup mechanism to retrieve data information. In [42], authors present a DHT algorithm to locate the node which holds a piece of information quickly. It assigns a key to data and nodes; thus, the node which contains key k calls its successor, the successor is defined by the closest node id to k.

In our proposed solution, we use a third-party application Redis[9] which allows a distributed *key/value* store mapped across the cluster and it handles a key-value lookup search. Our data index is built upon content from the storage nodes; we referred to it as "storage index" throughout this thesis. The detailed storage index construction and maintenance are discussed further in chapter 6.

[9]https://redis.io

CHAPTER 3

Related Work

This chapter gives an overview of solutions existent in the research community. Although we investigate these challenges like service problem placement and scheduling from the serverless-edge-computing view, many of these issues have already been studied and discussed for other scenarios. This chapter relates existent approaches for the challenges we are addressing in this thesis. It is structured as Section 3.1 where we target related research involving task distributions in general, Section 3.1.1 where we discuss approaches, especially for an edge environment. Following, we show in Section 3.2 proposed resolutions in different types of scenarios and how DHT,in Section 3.2.1 , plays a crucial role in this problem. In Section 3.2.2, we show how temporary disk storage can improve workflow performance. Further, Section 3.3 describes research that collects network status in limited computational ecosystems and how these metrics decrease bandwidth overhead.

3.1 Service Placement Problem

Edge-computing topology is commonly known as an intensive data producer, and in some cases, it needs to process large amounts of data. The service distribution also needs to consider this factor to choose the feasible host for the task execution. In [43], related researchers solve the problem in different ways according to their specific needs. In [44] authors use a multi-component application where the tasks are placed according to the ecosystem's components while in [45] authors focus on workload. The task placement should match each use case to reduce the overhead communication between the services and achieve high performance. According to [46], the job scheduling should also be responsible for other essential tasks like prioritization, capacity management, failure recovery, and job completion.

3.1.1 Scheduling at the Edge

Rausch et al. present skippy in [9]. A Kubernetes scheduler can make decisions based not only on typical CPU and RAM usage but also on specific node attributes such as GPU. Skippy brings an essential aspect for workflows such as ML where accelerators play an important role. A detailed review of Skippy's approach is explained in Chapter 5.

A similar approach is shown in [35] where a multi-objective optimized algorithm for container scheduling is developed. In an attempt to minimize the scheduling tasks processing time, the developers use a $NP - complete$ problem as optimization criteria. The scheduling algorithm is evaluated in the container orchestration system Docker Swarm[1]. Its decision-process mechanism scores incoming containers based on CPU and RAM utilization, image transfer from container registry, nodes which match containers' requirements, and clustering of containers which favors related containers to the same node. According to the authors, a multi-object algorithm performs with a complexity of $O(c \times n)$ where c is a set of containers while n is a set of nodes.

Following a different methodology, *KaiS* [47] propose a learning-based scheduler designed for a cloud-edge environment that aims to reduce incoming request processing time in the long term. To achieve that, the authors propose a coordinated multi-agent actor-critic algorithm for its request dispatching. Additionally, *KaiS* applies a graph neural network for its orchestration and decision-making process. Results show that this scheduling framework reduces the system processing rate by approximately 14%.

The edge scheduler in [48] focuses on the current network 5G. According to the authors, the 5G scenario introduces new network challenges as the edge nodes are connected to the internet. Still, they are not in the same edge network and have different subnets that mean they need to communicate via public IP. As the edge nodes are placed in various isolated networks, it is necessary to enable public Internet Protocol (IP) in each edge node to enable cross-communication between themselves. The custom Kubernetes scheduler addresses computational resources like CPU and RAM. However, this solution does not consider data-locality awareness in the edge-cloud scenario.

3.2 Data Placement

The idea presented in [49] goes beyond function placement. In this paper, data and function locations are equally important in an intensive data scenario. The authors say that the best approach for data storage in an edge network is to store the data closer to the execution node. This strategy decreases the data traffic and speeds up the service execution.

To avoid the overhead of sending a large amount of data to the cloud repeatedly, [50] created an optimization algorithm that allows a node in the edge network to share its resources and act as a micro data center. Considering a homogeneous edge-cloud

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

infrastructure where services can be executed either in the edge or in the cloud, once there is an incoming request, the algorithm can decide the shortest path for that request, accounting for data and function location cloud available resources. In [51], the authors used a similar approach; they created storage units in the edge network. Thus, devices could access the data quicker since there was no need for remote cloud requests.

In [52], it is noticed that accessing remote data storage or caches during task scheduling affects the performance significantly. Therefore, it developed a heuristic optimized schedule system for data access. They use the input data and node's resources to calculate possible paths for a specific task. Once the paths are known, it is possible to identify the shortest route and the first and the exit node. Whereas in [53], the authors reduced the bandwidth usage considerably by using a cache system in the edge network before sending the video packets to the long-distance cloud centers.

Another distributed data placement is proposed by Hadoop[2]. Its Hadoop Distributed File System (HDFS) strategy enables a data file system equally distributed across the cluster. Due to its MapReduce technique, Hadoop provides high efficiency and high availability in its lookup search [54]. MapReduce programming model takes as input a set of *key/value* populated by each cluster node containing every data piece stored. It reduces to a *key/value* single data list only, which allows quick access to the data [55]. Although it presents an efficient data placement mechanism, it still presents drawbacks. Its default implementation is designed for a homogeneous infrastructure where all the nodes have the same storage capacity. Nevertheless, researchers [54, 56] are addressing these problems by creating special Hadoop solutions for heterogeneous clusters and context characteristics awareness.

3.2.1 Distributed Hash Table

DHT is mentioned in [57] as a solution for storage and quick access to the data through the network. However, the author also points out that most of the DHT methods are designed to distribute the data equally within the network, which might not be the case of serverless-edge computing, since in this case, the data can be present only on a few nodes.

A Round-Hashing method is proposed in [58] for data storage on distributed servers solution, which might be a suitable option for the problem stated in this proposal. An ideal hashing algorithm should be available during deployment and runtime. Thus, the scheduler and the API can quickly access it.

Another caching solution for request routing is proposed in [59]. The authors present an algorithm able to improve the latency for content access of incoming requests. A caching method that can store data and paths was developed. Once the paths are stored in the cache, the framework can use this information to predict and allocate data accordingly. Consequently, the following requests will be executed faster.

[2]https://hadoop.apache.org

The use of caching systems in an intensive data environment avoids the overhead of resources like bandwidth. It decreases the traffic volume on the network and consequently reduces the latency. Nevertheless, every use case has a different requirement, and therefore it is necessary to analyze every situation carefully to find the best-caching strategy [60].

3.2.2 Ephemeral Storage

Often in a serverless-edge-computing environment such as ML, the data needs to be read-only once, e.g., pre-processed data [61]. This only-once read data is referred to as ephemeral data. In [62], authors showed that it is profitable to use local storage as ephemeral storage to avoid high latency to read and write a file that will only read once. After the data is used, the framework deletes the data. This mechanism takes advantage of local resources at their maximum, avoids overhead in the network, and significantly reduces network throughput.

In [63] the authors innovate in its in-memory caching solution designed specifically for data-intensive serverless scenarios. According to the authors, large object in-memory storage degrades the caching application as it consumes a significant amount of ram and it overloads the network. As in serverless computing, there are many situations that use ephemeral data. InfiniCache proposes mitigation of this problem by intelligently identifying no-longer-used data and evicting them. The disposal of ephemeral data reduces the financial costs since it reduces data storage in cloud solutions like AWS ElastiCache[3]. Additionally, it provides an intelligent backup methodology in which every serverless function is responsible for its data backup, which offers high availability within the cluster nodes.

3.3 Network Monitoring

Network monitoring is an important factor in a distributed systems ecosystem. The authors in [64] show how one can build a reliable bandwidth graph using intrusive methods such as *iperf*[4]. In a venture for a reliable result, intrusive tools transfer data on the maximum network capacity during the monitoring execution. Despite its accuracy, such methods add additional load on the network and might not be an option in a small time window or an already heavily used environment. Therefore, [64] presents a mixed solution of intrusive and non-intrusive methods. It uses effective data transfers to calculate the network capacity, which is later used as bandwidth metrics. Thus, the framework can obtain an average between intrusive precise data and non-intrusive, less precise data. Additionally, it proves that it is more profitable for decision-making to have a bandwidth graph built from reliable and estimated data than to have completely reliable data but with an overloaded network.

[3]https://aws.amazon.com/elasticache
[4]https://iperf.fr

Another solution [65] proposes a self-reporting network monitoring system. The telemetry application introduces a mechanism where the packets report the network speed during packet transportation in the network. As the packet report adds extra load on the network, the authors proposed an orchestration framework where a network telemetry scheduler can identify and select which node and route should be currently monitoring. This monitoring system provides reliable results without affecting the bandwidth.

CHAPTER 4

System Overview

The goal of this thesis is to add data-locality awareness to the existing skippy Scheduler. We develop a system that identifies the files in the storage in the Kubernetes cluster and creates a storage index. Additionally, our system monitors the network traffic to generate a bandwidth graph representing the network availability. To support the scheduling decision, we reinforce the data-locality awareness during function runtime. This chapter presents the core components of this system.

In Section 4.1, we list the components used in our system and give a brief explanation of how the components play together. Section 4.2 introduces our data-locality scheduler. In Section 4.3, we show how the storage index is created. Further in Section 4.4, we explain the storage node prediction during scheduling and runtime. Section 4.5 gives a brief overview of how we create the bandwidth graph. In Section 4.6, we detail how to use Skippy Data SDK to make data-locality decisions during runtime. Finally, in Section 4.7, we show the function deployment mechanisms of our system.

4.1 Overview

As this project is an extension of Skippy scheduler developed in [9, 11], it is important to keep consistency between the tools used in the two projects. Hence, we selected Kubernetes[1] as Container Orchestration (CO) platform, Skippy Data, described in Chapter 5, as scheduler, OpenFaas[2] as serverless framework and MinIO[3] as storage system. Additionally, we use Redis[4] as key/value store and Telemd[5] as system metrics collector. Furthermore, we introduce new components developed during this thesis.

[1]https://kubernetes.io
[2]https://github.com/openfaas
[3]https://min.io
[4]https://redis.io
[5]https://github.com/edgerun/telemd

Skippy-cli assists the function deployment, *Skippy-network* creates and updates the bandwidth graph and Skippy Data SDK responsible for data-locality prediction during runtime.

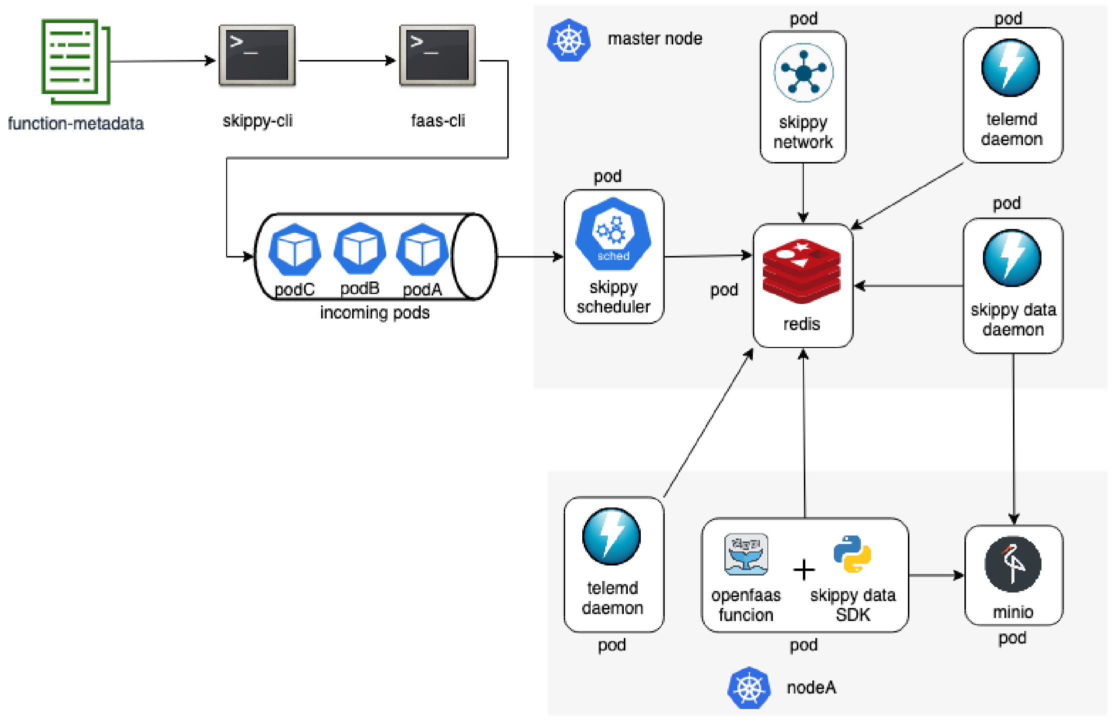


Figure 4.1: System Overview

In Fig. 4.1, we can see how the components interact in the complete solution. The tools *telemd daemon*, *skippy data daemon* and *skippy network* work asynchronously. Telemd collects the nodes' network metrics and stores them in the Redis system. Skippy network fetches the network metrics stored in Redis, generates the bandwidth graph, and stores the bandwidth graph in Redis. Skippy Data Daemon scans the MinIO storage nodes for files' metadata and creates the storage index. To initiate the data-locality decision process, first, the function is deployed with the assistance of our deployment tool Skippy-cli, described in Section 4.7.1. Skippy-cli assigns the Kubernetes labels necessary for data-locality scheduling and deploys the function via Faas-cli. Faas-cli creates a Kubernetes deployment which sends the new incoming pods to the scheduling queue. Skippy Data scheduler identifies new incoming pod and starts the scheduling process. At the data-locality priority from the Skippy Data Scheduler, the priority queries the storage index and bandwidth graph. Then, the scheduler finds the closest feasible node to the storage node, which contains the functions' files to host the incoming pod. During execution time, our Skippy Data SDK automatically downloads the files necessary for the function from the closest storage node. By the end of the function execution, the SDK finds the closest storage node again and automatically uploads the produced file.

4.2 Scheduler

Kubernetes is the tool used in this project for containerized workload management. Kubernetes presents advantages relevant for this project like the ability to run an external custom scheduler, FaaS integration, and active developments. Kubernetes also provides a scheduling workflow that has the decision-making capability to place FaaS functions on the node, which has more computational resources available [66].

This thesis deals with three Kubernetes schedulers: Kubernetes' default scheduler, the Skippy scheduler, and our scheduler extension called Skippy Data Scheduler. Skippy scheduler identifies the cluster node's attributes like hardware acceleration and resource usage to match workflow's requirements. When this resource knowledge is shared across the cluster, the scheduler can better identify incoming requests and make scheduling decisions based on each resource's node characteristics. However, to profit from intensive data transfer scenarios, it is necessary to learn where workflow data is stored to make scheduling decisions. We consider the execution of serverless function as workflow and workflow data, the files consumed or produced by a serverless function. The storage mechanism is explained further in Section 4.3.

The Skippy Data Scheduler is an extension of the Skippy Scheduler. Specifically, our scheduler adds data-locality awareness to Skippy's awareness of node characteristics and workload requirements. Our Skippy Data scheduler is designed as a framework able to identify in the environment additional characteristics related to the data workflow like network Input/Output (I/O) metrics and file metadata. Skippy Data Scheduler is further discussed in Chapter 5.

We refer to every node participant in the Kubernetes cluster simply as *node* for any device participant in the cluster or *feasible node* for devices able to host a specific incoming pod e.g. if a pod requires 1 Gigabytes (GB) RAM, a *feasible node* refers to node which can offer this amount of RAM. On the other hand, a docker container is wrapped in a Kubernetes pod. A Pod is the smallest deployable entity that must contain at least one container.

4.3 Storage Index

Storage plays an important role in this project. Since the project is designed for an infrastructure with limited resource availability, the storage tools must be reliable and efficient independently of its current environment. Furthermore, we assume that any data content replication across the network is handled by the chosen storage framework.

Towards solutions that satisfy data-locality requirements for the two types of storage used in this project, we decide to use MinIO for file storage and Redis as key/value store for metadata storage. Both tools are deployed in a local Kubernetes setup and accessible within the cluster.

To create and maintain the storage index used in this thesis, we use a dedicated component that asynchronously and regularly checks the MinIO instances for files update and stores the storage index in our Redis key/value store. The details about the storage index is described in Chapter 6.

4.3.1 Skippy Data Daemon

The Skippy Data daemon is responsible for creating the storage index. It identifies every file present on MinIO and places the file's metadata in a Redis key-value cache. The daemon constantly runs on the cluster and periodically searches all the storage nodes and collects information necessary to keep the storage index Section 6.2 up to date. The Skippy data daemon connects to every MinIO storage node and reads every bucket and file stat. Once all the files are known, the daemon can build and update the storage index accordingly. Since there is no specific need for the daemon to run in every node, thus we can deploy Skippy Data Daemon using Kubernetes default scheduler and priorities.

Besides the storage index, Skippy Data Daemon also stores secondary data which supports Skippy Data SDK in its decision-making process. This data is described below as:

- **Storage nodes**: The storage index provides a tree with data properties and relations like name, size, bucket and node. However, in order to directly connect to this node, the SDK needs to be aware of its address. Therefore, Skippy Data Daemon runs periodically and collects the HTTP address of each node e.g. `http://10.244.2.41:9000`.
- **Locality type**: Each node has a locality type which can be either *edge* or *cloud*. This property helps identify where the node is located. Once the framework is aware of its location, the SDK can take decisions based on the node's location. The locality type is the Kubernetes label which is assigned by Skippy Daemon as described in Section 5.3.1.

4.4 Storage Node Prediction

Our data-locality solution leverages information from the bandwidth graph and storage index to predict the closest storage node to transfer the necessary file for the serverless function. The storage node prediction happens during two moments, as described below.

4.4.1 Select Node at Scheduling

The Skippy Data scheduler uses a Data Locality Priority to predict the closest node to host an incoming pod. The data locality priority is responsible for searching every feasible node that can transfer the workflow data quicker from a specific storage node. To achieve that, the Data Locality Priority learns from the incoming pod, which "consume" and "produce" files the function needs. Once the file metadata is known, the priority queries the storage index to find which storage nodes contain those specific files. Finally, the scheduler searches a feasible node with higher bandwidth available to transfer the functions' files from a storage node option. The prediction of storage node during scheduling time is part of the Skippy Scheduler, and it is detailed in Section 5.4.3.

4.4.2 Select Node at Runtime

To enforce the Skippy Data scheduler's data-locality decision, we need to ensure the prediction of the storage node during the function runtime. The runtime storage node prediction is necessary because the bandwidth graph can change between scheduling and runtime. Thus, we created a tool Skippy Data SDK that predicts the storage during the runtime. In a similar decision process as the scheduler, when the function is executed, the SDK queries the storage index to fetch the functions' file metadata. Then, the SDK searches the closest storage node option according to the bandwidth graph. The storage node prediction during runtime is detailed in Skippy Data SDK in Section 6.4. Further in Section 4.6, we show how to use Skippy Data SDK to enable the storage node prediction during function runtime.

4.5 Bandwidth Graph

To have an overview of the networks' current usage, we need to create a bandwidth graph with current availability between the nodes. In order to achieve this, we need to have the correspondent network metrics from each node. Once we have the metrics we need to create the bandwidth graph. The bandwidth graph represents the network bandwidth availability between every node and storage node present in the cluster.

To generate a bandwidth graph without adding extra overhead on the edge network, we propose two non-intrusive approaches: "Estimated Bandwidth Graph" and "Precise Bandwidth Graph". Both approaches generate a bandwidth in a JSON format as displayed in Listing 4.1. The following subsections give an overview of the tools used to collect and create the bandwidth graph.

```json
{
"nodeA": {"storageNodeA": 4958842.4, "storageNodeB": 19651766.4},
"nodeB": {"storageNodeA": 25221414.0, "storageNodeB": 5048774.7},
"nodeC": {"storageNodeA": 1626606.8, "storageNodeB": 2810665.8},
"nodeD": {"storageNodeA": 17612750.1, "storageNodeB": 1368385.8}
}
```

Listing 4.1: Bandwidth Graph Example in Bytes/sec

Telemd

To collect node's real time metrics, we have used the *edgerunio*[6] telemetry system Telemd. This tool presents off-the-shelf metrics regarding node resource usage. It also provides the network's input and output rates in bytes/s. As the framework needs input from every device in the network, Telemd needs to run as a daemon on every node on the cluster. Metrics collected by the tool are stored in Redis. Although the metrics collected by this tool provides very good insight from the current network state, it is still not enough to build the bandwidth graph. Therefore during this project development, we enhance Telemd with new metrics which are described in Chapter 7.

Skippy-network

Skippy-network is a standalone application that reads collected network metrics by Telemd and calculates the bandwidth graph. Skippy-network works as a daemon application continuously analyzing Telemd data to provide a current overview of the network overload. It provides two different approaches to generate the bandwidth graph: "Precise Bandwidth Graph" and "Estimated Bandwidth Graph". The "Precise Bandwidth Graph" approach uses the download and uploads effective speed to generate the graph. In contrast, "Estimated Bandwidth Graph" reads Linux kernel configuration properties and Telemd metrics to produce the graph. Details and implementations are discussed further in Chapter 7.

4.6 Skippy Data SDK

Once the serverless function is placed near the storage node, it still needs to take the decision during runtime. At the request time, there needs to be a framework or tool which decides that *storageA* is closer than *storageB*. Following these requirements, Skippy Data SDK is introduced as a Skippy library which considers the storage environment and its metadata to make runtime decisions.

This python library enables the user to transfer the data from the closest node during request time. The data transfer can be specified in levels such as download and upload or even single or multiple file transfers. Skippy Data SDK is further discussed in Section 6.4.

4.6.1 Decorators

A FaaS function can invoke the SDK by Python decorators as shown below. The Skippy configuration file enables the user to consume and produce more than a single file at once. The python decorators read the file's metadata from the Skippy config file added in the OpenFaas function. The configuration file is the same one used by Skippy-cli, discussed in Section 4.7.1, for the function deployment as described in Listing 4.2.

[6]https://edgerun.io

```
@consume()
@produce()
def handle(request, consumed_data=None):
    produced_data = produce(consumed_data)
    return produced_data
```

Additionally to the configuration file, the user can also use the decorator without Skippy configuration file. The decorator accepts Uniform Resource Name (urn) as shown below.

```
@consume("mybucketc:myfilec")
@produce("mybucketp:myfilep")
def handle(request, consumed_data=None):
    produced_data = produce(consumed_data)
    return produced_data
```

4.7 Function Deployment

To deploy our Skippy labels, we use Skippy-cli. As serverless framework to be used in this thesis, we choose OpenFaaS due to its simplicity, flexibility, and previous research [11, 9]. Additionally, in our framework, Faas-cli is used by Skippy-cli, described in last Section 4.7.1, to deploy the serverless functions.

4.7.1 Skippy-cli

Skippy-cli is a command line tool used to deploy OpenFaas functions. Metadata identification is essential for the decision-making process, if no file metadata is present during the deployment, data locality cannot be achieved neither in the scheduling nor during execution time. Therefore, Skippy-cli is introduced as an encapsulated OpenFaas command line interface (CLI) which reads a file containing properties associated to file metadata and specific function's attributes like chain name. Fig. 4.2 specifies how the Skippy participates in the function deployment in combination with Kubernetes and OpenFaas.

Yaml Configuration File

Skippy-cli enhances the deployment process by reading a configuration file and translating these properties into Kubernetes pod labels. These properties are defined in yaml[7] format as described in Listing 4.2. To be recognized and parsed by the CLI, the configuration must be named `skippy.yml` and be placed in the function's directory, e.g., `~/my-function/skippy.yml`. The placement of the configuration file inside the

[7]https://yaml.org

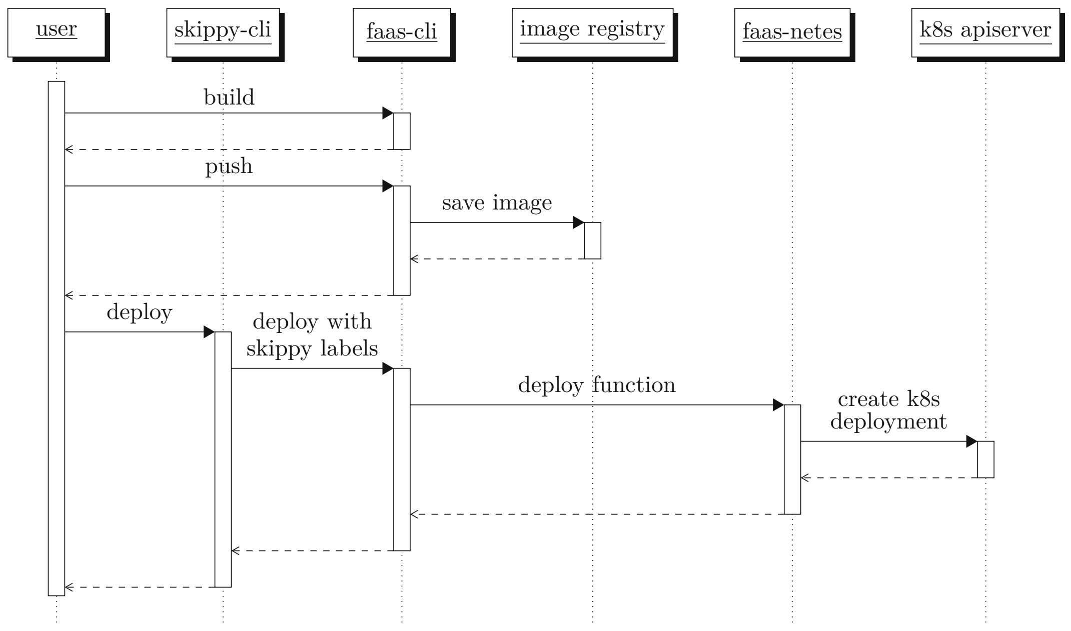


Figure 4.2: Skippy Function Deployment

function's directory enables the build process to pack this file inside the docker image, which makes it accessible at runtime. As the Yaml file contains the file's metadata such as consume/produce bucket and filename, it provides this information at runtime, which is used by Skippy data SDK in its storage node prediction described in Section 6.4.

```
data:
  consume:
    - bucketone.data-preproceesing.csv
    - buckettwo.additionaldata.csv
  produce:
    - producebucket.trainedmodel.npy
chain:
  function: skippychain
```

Listing 4.2: Skippy Yaml Configuration

Specifically, Skippy-cli is a CLI that identifies two properties from the yaml configuration file: file metadata and chain function. Regarding file's metadata, one can specify multiple properties in consume and produce for download and upload respectively. Due to alphanumeric limitations in underlying *faas-cli*[8] system, Skippy CLI reads file' metadata

[8]https://github.com/openfaas/faas-cli

in the format *bucket.filename.fileformat*. The yaml configuration file is translated as Kubernetes labels as displayed below.

```
skippy.io.data.consume=bucketone.data-preproceesing.csv
skippy.io.data.consume=buckettwo.additionaldata.csv
skippy.io.data.consume=producebucket.trainedmodel.npy
skippy.io.chain.function=skippychain
```

Command Line Tool

Once installed, Skippy-cli provides one option `deploy` which reads OpenFaas function file. It can be triggered by the command below.

```
$ skippy deploy my-function.yml
```

4.7.2 OpenFaas & Faas-cli

Following the FaaS principle, OpenFaas allows users to deploy serverless functions into the Kubernetes framework. OpenFaas enables users to deploy a function using minimum configuration. If any additional configuration is necessary, it can be adjusted in the configuration file that contains the function's information.

OpenFaas provides a simple and flexible process to create and deploy serverless functions. Furthermore, OpenFaas provides additional tools like command line tools, Graphic User Interface (GUI) and monitoring tools. To enable the serverless function triggering, the framework uses a watchdog. The watchdog is a small HTTP server deployed with every OpenFaas function. The framework abstracts this task from the user, it automatically deploys a watchdog and attaches the function to it. Once the function is triggered, the watchdog receives the incoming requests, executes the function and forwards its output back to the user.

Faas-cli[9] provides options to assist the OpenFaas function deployment in the Kubernetes cluster. The CLI is an additional tool developed by OpenFaas. It is an option for users that are more comfortable with terminals instead of user interfaces. Additionally, the framework also provides an user interface which allows users to deploy and remove functions accordingly.

[9]https://github.com/openfaas/faas-cli

CHAPTER 5

Skippy Data Scheduler

In the first part of this chapter, we make an overview in Section 5.1. Following in Section 5.2 we describe a typical Kubernetes scheduler architecture that is the core of our Skippy Scheduler's implementation. In Section 5.3, we present the first version of Skippy Scheduler, which can identify different devices' resource and workflow characteristics such as GPU. Finally, in Section 5.4, we present Skippy Data Scheduler, an enhanced version of skippy designed to address data locality in data-intensive workloads.

5.1 Overview

The Skippy Data Scheduler is an enhanced version of Skippy. Skippy is a custom Kubernetes scheduler that enables serverless edge computing. Skippy adds priority functions that target edge computing characteristics, such as edge and cloud locality or node capabilities, to address workload and infrastructure heterogeneity. At scheduling, Skippy can identify specific workflow requirements and match them with the current cluster infrastructure such as image awareness or device edge and cloud locality [11].

5.2 Kubernetes Scheduler

To understand how Skippy Data Scheduler, described in Section 5.4, works, we need to look at Kubernetes Scheduler's architecture. Skippy Scheduler and its data-locality extension, Skippy Data Scheduler, use similar stages as Kubernetes default's scheduler during their scheduling process. According to its implementation[1], Kubernetes' default scheduler is in charge of node selection for the incoming pods as can be seen in Fig. 5.1. The selection process happens in four stages, as described below.

[1]https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/#kube-scheduler-implementation

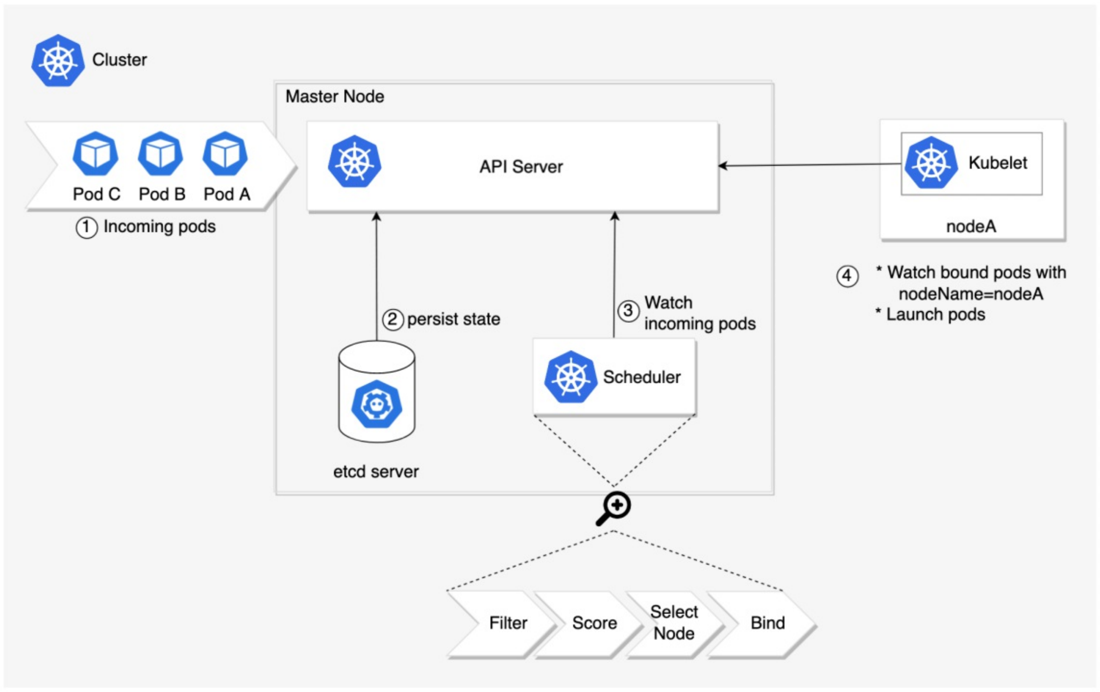


Figure 5.1: Kubernetes Scheduler Overview

5.2.1 Predicates Filter

The predicates should exclude the nodes which do not match the pod's expectations. This stage can be composed of one or more *Predicates*, and each of these predicates analyzes a different requirement. As an example, one incoming pod might require 2 GB of RAM. During this stage, the scheduler should eliminate the nodes which cannot offer this amount of RAM. Predicates work as a filtering mechanism to exclude nodes that do not match the pod's demands. Besides Kubernetes generic predicates, Skippy presents one custom predicate *PodFitsResourcesPred*. Skippy Data Scheduler uses the predicates filter to select only nodes which can host an incoming pod. Although the filtering process does not directly affect the Data Locality Priority, the predicates are essential to eliminate unqualified nodes from the priority scoring process.

PodFitsResourcesPred

This predicate checks if the nodes have requested CPU and RAM. In case there is no value assigned, skippy uses default values of 0.1 for CPU and 200Megabytes (MB). This behavior is also present on Kubernetes Default Scheduler Section 5.2. The scoring system assists the scheduler in balancing loads between the nodes and filtering out nodes overloaded.

5.2.2 Priority

The scoring process defines weights for the feasible nodes according to a specific capability. In this stage, the scheduler grades all priorities for all nodes. A priority can be a simple check if the node has a specific label e.g. *FunctionChainPriority* or can also determine complex checks like *DataLocalityPriority*. Regardless of its implementation, the priorities are a key factor for the node selection and, consequently, scheduling decision.

Balanced Resource Priority

This priority is similar to *PodFitsResourcesPred*, it considers CPU and RAM usage for its scoring. Since *PodFitsResourcesPred* predicate already excluded out-of-capacity nodes in the previous stage, during this priority all feasible nodes will be scored and load distributed. This scoring is achieved by the fraction $requestedCPU\ /allocatableCPU$ and $requestedRAM\ /allocatableRAM$ for CPU and RAM respectively.

Image Locality Priority

The priority favors nodes that already have the Docker Image locally stored. This image node favoring avoids multiple image downloads from the container registry. The pod leverages the existent Docker Image on the node and consequently faster startup time.

5.2.3 Select Node & Bind

In this stage, the scheduler picks the best-weighted node from the previous step. The node which accumulates more points will move to the next phase to be bound to the pod.

After node selection, the binding happens between the scheduler and the Application Programming Interface (API) server. Once the scheduler selects the node, it will send this information to the Kubernetes API server, which will save its state and bind this node to the incoming pod.

5.3 Skippy Scheduler

Skippy uses the four stages of Kubernetes scheduling described in previous Section 5.2. This section explains Skippy's specific priorities that enable scheduling according to workload and device characteristics such as edge and cloud locality or video accelerators presence. Skippy Data Scheduler is a data-locality extension applied on Skippy Scheduler. Thus we need to show Skippy's implementation to understand how these schedulers address edge infrastructure and the workload during its scheduling process. Next, Skippy Data Scheduler in Section 5.4 details our specific data-locality mechanism. However, Skippy Data Scheduler also utilizes every Skippy predicate and priority described in this section.

Skippy, at its core, was designed to analyze the cluster's available resources. Skippy scheduler stands out as a resource scheduler that also addresses the workload. Additionally, its workflow's orientation design allows the scheduler to match nodes with workflow requirements such as video accelerators.

5.3.1 Skippy Daemon

The Skippy Daemon is a service that runs continuously and monitors nodes on the cluster. Skippy Data Scheduler uses this service to identify and label nodes according to their locality edge or cloud. These labels are used during scheduling by the priority *DataLocalityPriority* detailed in Section 5.4.3. The daemon identifies node capabilities such as GPU presence or locality type values like *edge* and *cloud*. As every node presents different characteristics, the Skippy Daemon periodically collects this data. If no locality value is present, it adds the label *locality.skippy.io/type=<edge>*. Additionally, it includes also label *capability.skippy.io/<nvidia-gpu|nvidia-cuda>*. Specific labels related to the GPU support the scheduler to prioritize nodes that provide this feature. In the same principle, the Skippy Scheduler uses locality labels later to identify edge nodes and consequently decrease cloud resource usage and costs.

5.3.2 Priorities

The Kubernetes scheduler is composed of many priorities, which score the nodes according to their ability to fit the pod's requirement. By the end of the scoring process, the scheduler selects a winner node to host the incoming pod. Skippy enables different scoring parameters using multiple priorities, as is described in each subsection below.

Locality Type Priority

Locality type is one of Skippy's priorities. This priority identifies nodes situated on the edge network. The locality information is essential for the scheduler; it participates directly in decreasing financial costs as edge nodes have lesser economic costs than cloud-hosted nodes. Nodes on the edge network have preference over nodes on the cloud, thus minimizing cloud services fee such as AWS.

Latency Aware Image Locality Priority

The initial implementation aims to identify nodes with faster downlink connections thus can transfer their Docker Image faster from the container registry. This priority scores every node according to its connection and transfers time. Since skippy did not support real-time network monitoring information, the priority works with a static assumed bandwidth graph.

Data Locality Priority

Skippy [11] introduced a primary data locality priority. The incoming pods that contained labels specifying the data size could be prioritized based on an assumed bandwidth graph provided during startup application. Since this was not enough to mitigate the problem in a data-intensive scenario, this priority is re-designed in our skippy data scheduler 5.4.

Capability Priority

Workflows that produce a large amount of data and process this data close to data produced, such as the typical serverless-edge-computing scenario, motivates this thesis. For such scenarios, the identification of specific components such as GPU is a crucial factor for the scheduling decision. Skippy shows an improvement in the total execution time of jobs scheduled on nodes that matches workflow's specific hardware requirements like GPU.

5.4 Skippy Data Scheduler

Skippy Data Scheduler is an enhanced version of its first implementation described in section 5.3. The new scheduler presents a data-locality feature that can identify nodes and storage nodes according to their distance and data transfer capacity. It also identifies multiple functions that produce and consume data from each other. In this case, the scheduler aims to speed the execution time by leveraging nodes' resources. Figure 5.2 shows how the sequence diagram works since the user deploys a new function until the best node is selected and bound by the scheduler.

5.4.1 Data Cluster Context

Skippy context provides a cluster context that loads the scheduler information and facilitates its maintenance and extension. Skippy Data Scheduler presents an additional context *DataClusterContext* that loads the storage index at scheduler startup. Fig. 5.3 shows the context relation between each other. Chapter 6 details how Skippy Data Daemon scans the cluster and creates the storage index, which is constantly updated and stored in a key-value caching system.

5.4.2 Simulation Cluster Context

The Simulation Cluster Context, introduced in [11], enables the evaluation of this project. Our testbed does not provide enough resources to evaluate the scheduler on a big scale, e.g., 5000 devices. Thus, we use a Simulation Cluster Context during our evaluation. Generally, the Simulation Cluster Context in combination with Faas-sim[2] allows the user to simulate every cluster method without a Kubernetes cluster running as shown in Fig. 5.3. During its startup in the *get_storage_index*, the Simulation Cluster Context loads the storage index described in Chapter 6. This context triggers the scheduling mechanism on *place_pod_on_node* which later triggers our Data Locality Priority. Additionally, during scheduling, in *get_dl_bandwidth*, the Simulation Cluster Context queries the bandwidth

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

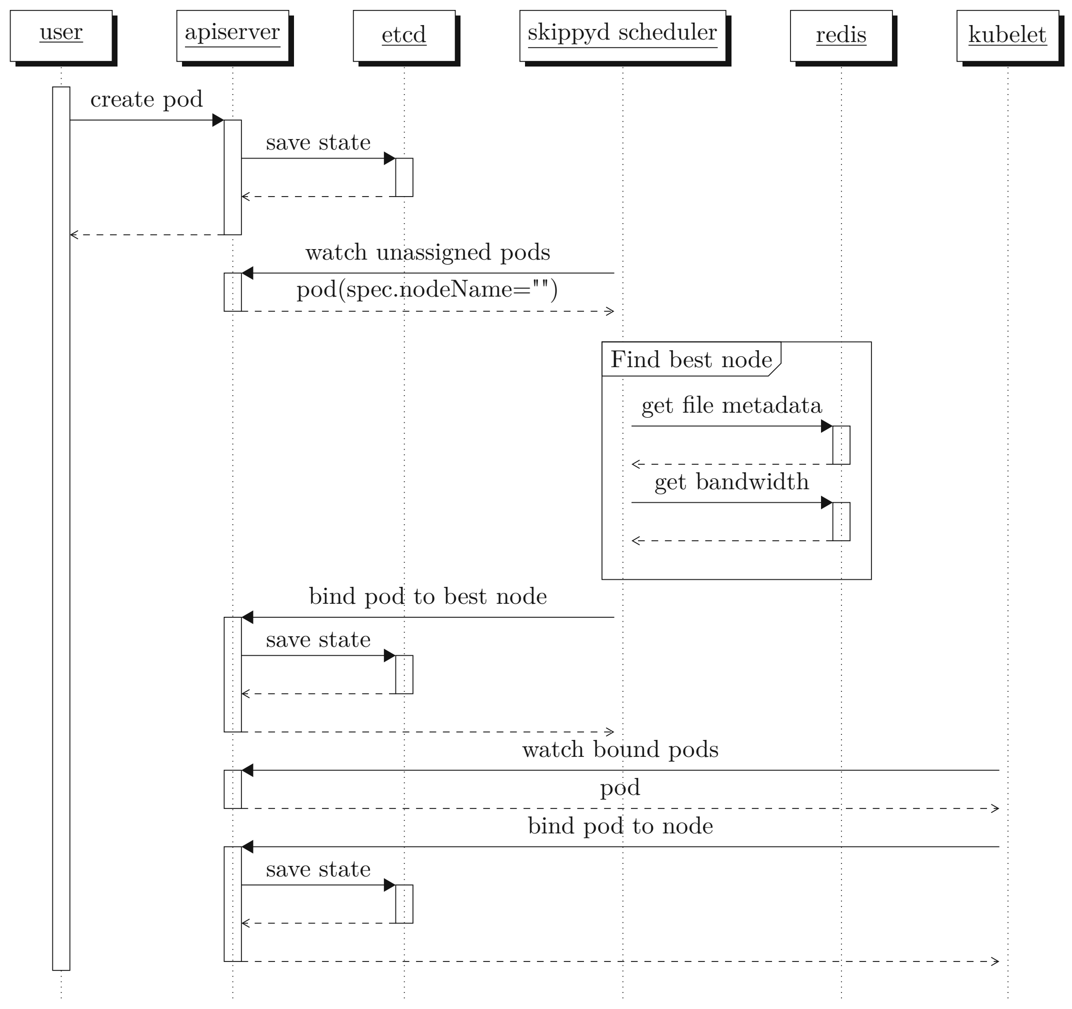


Figure 5.2: Skippy Data Scheduler

graph generated by our Skippy Network described in Chapter 7. Chapter 8 details the evaluation process. Additionally, it shows how Skippy Data Scheduler performs in large clusters like with large numbers of nodes.

5.4.3 Priorities

The critical factor for Skippy Data Scheduler is to find the best node which matches data-locality characteristics for a given function. The best node is calculated by the sum of each priority described below. By the end of priorities execution, the scheduler selects the node with a higher sum score. To achieve data locality in the Skippy Data Scheduler, we created a new *Function Chain Priority*. Additionally, we modified the existing *Latency Aware Image Locality Priority* and *Data Locality Priority*.

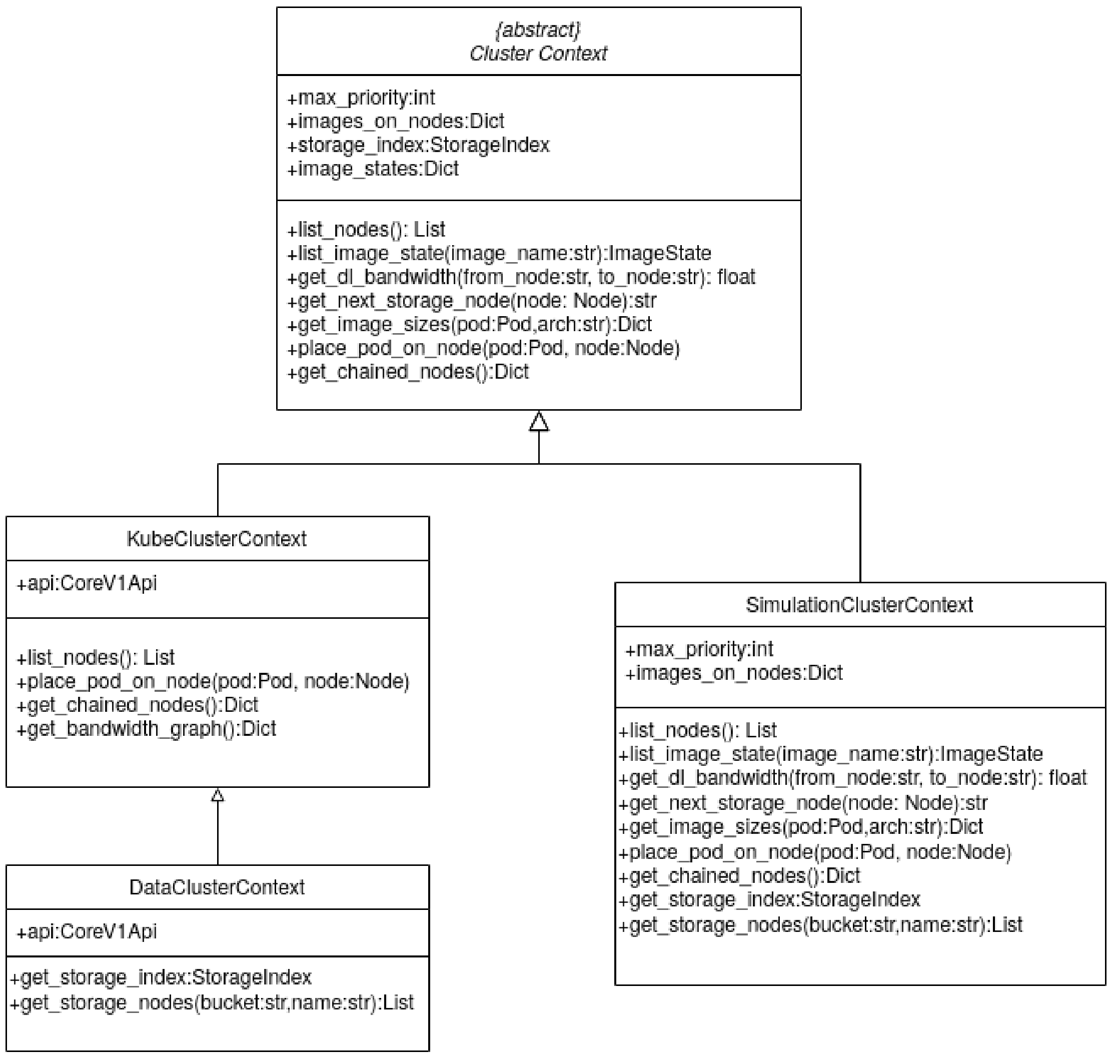

{abstract}
Cluster Context
+max_priority:int
+images_on_nodes:Dict
+storage_index:StorageIndex
+image_states:Dict
+list_nodes(): List
+list_image_state(image_name:str):ImageState
+get_dl_bandwidth(from_node:str, to_node:str): float
+get_next_storage_node(node: Node):str
+get_image_sizes(pod:Pod,arch:str):Dict
+place_pod_on_node(pod:Pod, node:Node)
+get_chained_nodes():Dict
KubeClusterContext
+api:CoreV1Api
+list_nodes(): List
+place_pod_on_node(pod:Pod, node:Node)
+get_chained_nodes():Dict
+get_bandwidth_graph():Dict
DataClusterContext
+api:CoreV1Api
+get_storage_index:StorageIndex
+get_storage_nodes(bucket:str,name:str):List
SimulationClusterContext
+max_priority:int
+images_on_nodes:Dict
+list_nodes(): List
+list_image_state(image_name:str):ImageState
+get_dl_bandwidth(from_node:str, to_node:str): float
+get_next_storage_node(node: Node):str
+get_image_sizes(pod:Pod,arch:str):Dict
+place_pod_on_node(pod:Pod, node:Node)
+get_chained_nodes():Dict
+get_storage_index:StorageIndex
+get_storage_nodes(bucket:str,name:str):List

Figure 5.3: Skippy Cluster Context

Latency Aware Image Locality Priority

As already described in the previous section, this priority scores the nodes according to their network availability and link speed. Skippy Scheduler scores every node available in the cluster accordingly. Nodes with higher-speed connections present higher weights, while lower-speed connections have lower weights. Due to the network monitoring solution presented in chapter 7, Skippy Data Scheduler enhances this priority with real-time network information. Since real-time bandwidth graph data is part of the scoring process, the priority can provide more reliable weights.

Data Locality Priority

Data Locality Priority follows the Kubernetes general scoring principle. That means, Data Locality Priority is composed of two externally available functions *score* and *normalize*. The score function, shown in 5.1 as pseudo-code, is responsible for identification and scoring. At the same time, the *normalize* method translates the score into a global weight which can be later summed with other priorities' results. Function *normalize* was reused from Skippy's homonym priority [11], thus omitted from the algorithm 5.1.

As one can see in the algorithm 5.1, the priority receives a node and a pod as input. The pod is an instance of an incoming pod that contains labels that identify which data file this pod needs to transfer during its execution. The node is a potential candidate for the pod's scheduling. The scheduler is in charge of triggering this priority for every possible node and selects a candidate node that presents the best results.

This priority calculates the necessary amount of time to transfer the requested data from the storage nodes. The result is obtained by the sum of the required download and upload time. The scheduler needs quick access to the bandwidth graph and the storage index as it relies on these elements to make correct decisions. It is crucial to have this information up-to-date and accessible whenever necessary.

When this priority is triggered, it first searches for the file's metadata in the storage index. The storage index contains information about files existent in the storage nodes. If the required data by an incoming pod is not present in a node candidate, the Data Locality Priority scores the node with a minimum value of 0. Chapter 6 describes the process to create and maintain the storage index. It also details how this information is accessible at a pod's deployment time by the scheduler and at a function's runtime by the Skippy SDK 6.4. The second key factor for this priority execution is the network state. Chapter 7 explains how Skippy Network creates the Bandwidth Graph and keeps it accessible for the scheduler at any time. The bandwidth graph represents the current network status; it provides the available network speed between the source and the target node. Once the priority is awake of which storage nodes contain the data, it fetches the network availability between node candidate and storage. In possession of network availability and file metadata, the priority can do a simple math dividing file size by bandwidth as displayed in 5.1.

Function Chain Priority

In a data-intensive workflow, one function's input might be the output produced by another function, e.g., *functionB* receives as input the data produced by *functionA*. However, as *functionA* was still not executed, thus this input does not exist. Therefore, the Data Locality Priority 5.4.3 is not able to score based on the data present on the storage nodes. These functions are called *chained functions*, and the scoring for these cases needs to happen based on the chain instead of the data input.

This priority enables chained functions to be prioritized closer to each other by iden-

Algorithm 5.1: DataLocalityPriority

```
   Result: Score time necessary to transfer data described in the labels
 1 Function score:
       Input : node
       Input : pod
 2     time ← 0;
 3     time $\xleftarrow{+}$ calculateDownloadTime(pod,node) ;
 4     time $\xleftarrow{+}$ calculateEstimatedUploadTime(pod,node) ;
 5     return time;
 6 Function calculateDownloadTime:
       Input : node
       Input : pod
 7     time ← 0;
 8     files ← list of files of pod's skippy.io.data.consume label;
 9     dataItems ← list of files metadata object from StorageIndex;
10     for dataItem in dataItems do
11         storageNodes ← getStorageNodes(dataItem.bucket,dataItem.fileName);
12         maxBandwidth ← 0;
13         for storageNode in storageNodes do
14             bandwidth ← getBandwidth(node,storageNode);
15             if bandwidth > maxBandwidth then
16                 maxBandwidth ← bandwidth ;
17         time $\xleftarrow{+}$ $\frac{\text{dataItem.size}}{maxBandwidth}$;
18     return time;
19 Function calculateEstimatedUploadTime:
       Input : node
       Input : pod
20     time ← 0;
21     files ← list of files of pod's skippy.io.data.produce label ;
22     dataItems ← list of files metadata object from StorageIndex;
23     fileSizeSum ← sum of download files size;
24     for dataItem in dataItems do
25         storageNodes ← getStorageNodes(dataItem.bucket);
26         maxBandwidth ← 0;
27         for storageNode in storageNodes do
28             bandwidth ← getBandwidth(node,storageNode) ;
29             if bandwidth > maxBandwidth then
30                 maxBandwidth ← bandwidth ;
31         time $\xleftarrow{+}$ $\frac{\text{fileSizeSum}}{maxBandwidth}$;
32     return time;
```

tifying which chain it belongs. Chained function recognition is possible due to Kubernetes pods labels which is added by Skippy-CLI 4.7.1 via Skippy YAML configuration *chain.function=<chain-name>* as displayed in listing 4.2. By placing chains on the same node, functions leverage direct access to the temporary storage, thus spare the network transfer of its input data. The temporary storage is later detailed in the subsection 6.4.1.

Once the priority is triggered, it searches pods in the cluster that belong to the same chain, and it scores a specific node based on the presence or absence of request chained function as shown in pseudo-code 5.2. Function *normalize* is responsible for translating the scores. As in every priority function, the score normalization converts this score into a weight between 1 - 10. Later the scheduler sums up with remaining priorities weights. The *normalize* function is recycled from Skippy [11], thus omitted from the algorithm 5.2.

Algorithm 5.2: FunctionChainPriority

```
  Result: Score chained functions in a node
1 Function score:
      Input : node
      Input : pod
2     score ← 0;
3     if pod has 'skippy.io.chain.function/chain-name' label then
4         score $\xleftarrow{+}$ 1 ;
5     return score;
```

5.5 OpenFaaS Modifications

As described in Chapter 4, Skippy Data Scheduler uses OpenFaaS as a serverless computing framework to deploy the FaaS functions. To trigger our custom scheduler described throughout this chapter, we modified few parameters in the OpenFaaS framework as described below.

Scheduler Assignment

OpenFaas is currently not designed to support an additional Kubernetes scheduler. In other words, the OpenFaas functions will by default always be deployed with the property *scheduler:default-scheduler*. This property assures the functions to be always scheduled by kubernetes default's scheduler. In order to deploy OpenFaas functions with skippy data scheduler, we modified this OpenFaas deployment property from *scheduler:default-scheduler* to *scheduler:skippy-scheduler*.

Hostname Environment Variable

As described in Skippy Data SDK storage node prediction Section 6.4.2 finds the best storage node to either download or upload a specific file. However, this prediction only happens if the SDK is aware of which node the functions are running on as its route source, then the SDK can use the hostname as route source to query a target storage node that has more bandwidth available in the bandwidth graph. Since OpenFaas does not enable environment variables from its function properties, we slightly modified *faas-netes* component to retrieve the node's hostname from the Kubernetes configuration and include this as the pod's environment variable.

Persistent Volume Claim

To be able to mount volume shares in the Kubernetes deployment, one needs to create Persistent Volumes which can be claimed by a specific Kubernetes service account user [67]. However, this Kubernetes feature is not available through OpenFaas build and deploy process, thus we created a cluster Persistent Volume (PV) and Persistent Volume Claim (PVC) dedicated for skippy functions. In the *faas-netes*[3] component, we have bound the OpenFaas function to the skippy PVC. This feature is described in ephemeral storage in Section 6.4.

Docker Template Store

OpenFaas provides function templates[4] which take care of basic configuration such as Dockerfile for image build and specific language script. As our experiments are executed in an ARMv7 architecture testbed, it is required to build docker images specifically for this architecture and therefore we modified its Dockerfile specification to match our arm testbed. Additionally, we have included special libraries necessary for our ML test workflow as shown in Section 8.1.2.

[3]https://github.com/openfaas/faas-netes
[4]https://github.com/openfaas/templates

CHAPTER 6

Data Management

In this chapter, we generally explain how the file's metadata is handled. In Section 6.1, we briefly overview why data management is necessary for our framework. Section 6.2 describes the data structure used to keep the file's metadata and the components used as storage in the cluster. Further, in Section 6.3 we explain the process to create and update the metadata information. Additionally, we explain when this data is used during scheduling and runtime function execution. Section 6.4 introduces our component Skippy Data SDK that provides data-locality awareness during runtime.

6.1 Overview

To enable metadata information access for the scheduling decision process described in Chapter 5, we need a lookup system that allows a fast way to access the file's metadata existent in the cluster. During scheduling, the decision-making process profits from quick to the data index, which means it does not need to manually request every file's metadata information. If the Skippy Data Scheduler has access to a data index, it can quickly access files' information metadata leading to a faster decision-making process.

To manage the files within the cluster, we use MinIO for the data storage and Redis as a cache framework. Both tools are deployed in the Kubernetes cluster, which means both Redis and MinIO can communicate with any other pod also present in the cluster. Figure 6.1 shows a simple overview from the storage framework in the Kubernetes cluster.

To achieve a fast and efficient lookup, we keep the data in MinIO storage and its metadata in key-value Redis cache. Furthermore, we created a data structure to facilitate metadata access.

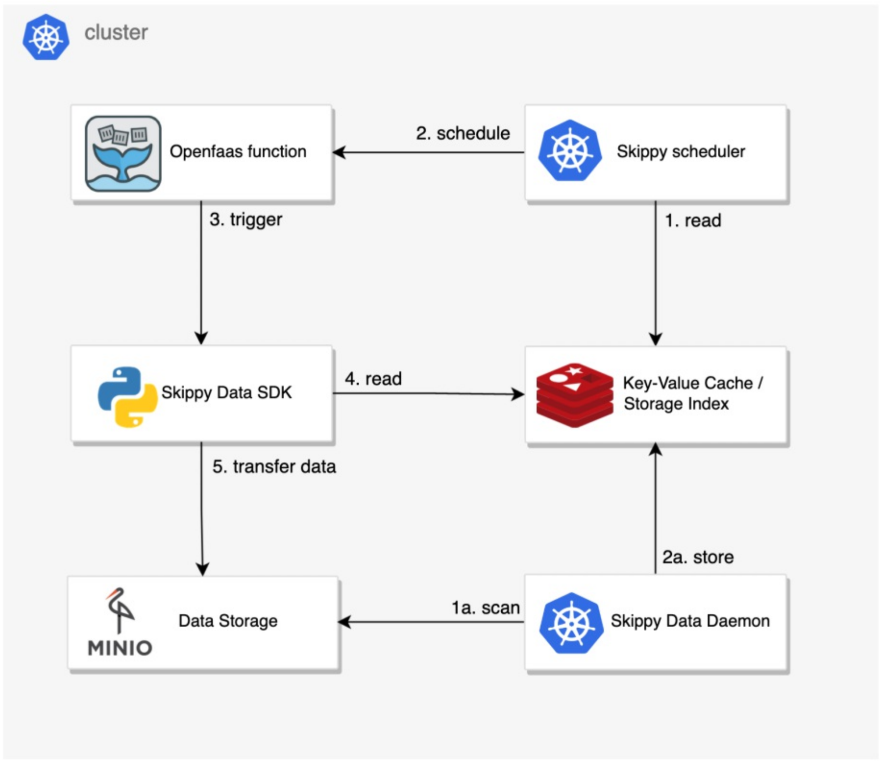


Figure 6.1: Data Management Framework Overview

6.2 Storage Index

Storage Index is a data structure that provides metadata information for all the available files stored in the MinIO buckets. We can define it as a data structure that stores file metadata and supplies this information whenever necessary.

The data index can be accessed either via in-memory storage described in Section 6.3.1 or via key-value cache described in Section 6.3.2. It ensures complete metadata scanning of the files existent in the storage nodes. Through the structure listed below, it is possible to know file properties like name and size. Additionally, the storage index supports a tree structure that contains metadata from each file present in the storage nodes. Once this storage mapping is fully read, the lookup mechanisms detailed in 6.3 delivers the information. Figure 6.2 displays Storage Index model class.

6.2.1 Skippy Data Daemon

As introduced in Section 4.3.1, Skippy Data Daemon is the component responsible for creating and updating the Storage Index. The daemon keeps track of data items on a set of storage nodes in the cluster. Skippy Data Daemon is an independent component that automatically recognizes new MinIO pods and scans the files present on the MinIO

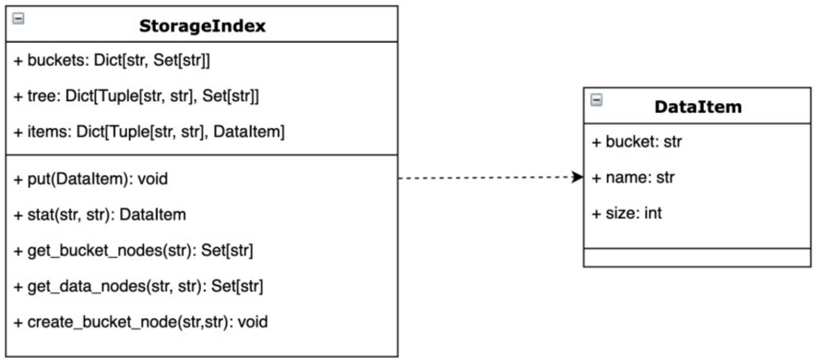

StorageIndex
+ buckets: Dict[str, Set[str]]
+ tree: Dict[Tuple[str, str], Set[str]]
+ items: Dict[Tuple[str, str], DataItem]
+ put(DataItem): void
+ stat(str, str): DataItem
+ get_bucket_nodes(str): Set[str]
+ get_data_nodes(str, str): Set[str]
+ create_bucket_node(str,str): void
DataItem
+ bucket: str
+ name: str
+ size: int

Figure 6.2: Storage Index Model Class

buckets. To find the information necessary for the data structure shown in Fig. 6.2, Skippy Data Daemon requests every file stats via MinIO API[1]. Thus, the storage index creation and update time vary according to the disk I/O, the number of storage nodes and the number of files in the buckets. Once the storage index is up-to-date, Skippy Data Daemon stores it on Redis key-value cache, which can be later retrieved by Skippy Data Scheduler or Skippy Data SDK.

Skippy Data Daemon can be deployed in the Kubernetes cluster as described in Listing 6.1.As it recognizes any MinIO in the cluster, the daemon can run on a single node only. Once deployed in the cluster, Skippy Data Daemon updates the Storage Index every 5 minutes. However, the update time can be adjusted as necessary.

6.2.2 MinIO

As a data centered solution for edge computing, Skippy Data Scheduler needs a lightweight solution which provides flexibility but it is still compatible with cloud computing services like AWS S3[2]. Therefore we opted for MinIO as our file storage tool. This storage holds every file consumed and produced by the functions. MinIO is an open-source project which provides high performance object storage. Besides the ability to run on edge and cloud environments, MinIO is also compatible with S3.

As the official version do not support arm architecture which is a common used architecture in the low processing devices present in edge network such as *RaspberryPi*[3] Single Board Computer (SBC), we use a community arm version of MinIO which is called *MinIO-Multiarch*[4].

[1]https://docs.min.io/docs/python-client-api-reference.html#stat_object
[2]https://aws.amazon.com/s3
[3]https://www.raspberrypi.org
[4]https://github.com/jessestuart/minio-multiarch

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: skippy-data-daemon
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: skippy-data-daemon
  template:
    metadata:
      labels:
        app: skippy-data-daemon
    spec:
      serviceAccountName: skippy-data-daemon
      nodeSelector:
      containers:
      - name: skippy-data-daemon
        image: keniack/skippy-data-daemon:0.5
        imagePullPolicy: Always
        ports:
        - containerPort: 5002
        env:
        - name: redis_host
          value: 10.107.29.82
        - name: MINIO_AC
          value: "myuser"
        - name: MINIO_SC
          value: "mypassword"
```

Listing 6.1: Skippy Data Daemon Deployment Specification

6.3 Metadata Handling

The data index is stored in an in-memory cache and key-value cache-store. The scheduler always tries to fetch the information from the scheduler's in-memory cache. In case file metadata is not present in the first layer, the scheduler retrieves this information from the second level key-value cache, as is detailed in the following subsections. Figure 6.3 shows how the storage index lookup works with different levels.

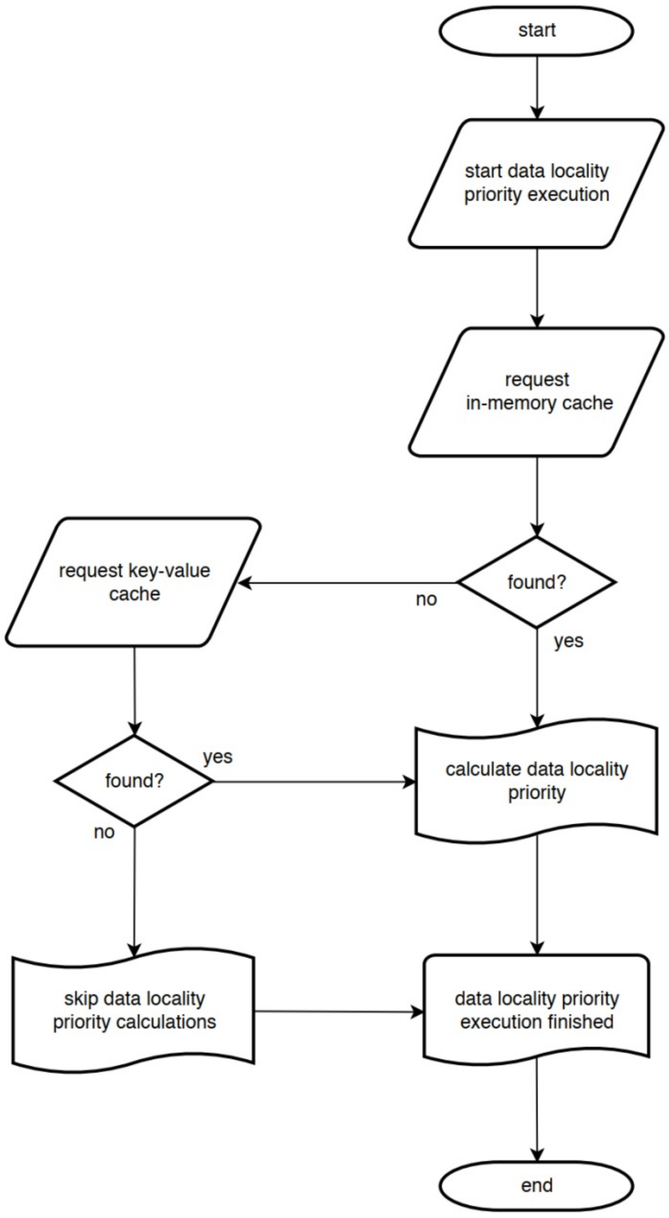

start
start data locality priority execution
request in-memory cache
found?
no
yes
request key-value cache
found?
yes
no
calculate data locality priority
skip data locality priority calculations
data locality priority execution finished
end

Figure 6.3: Storage Index Cache Layers Lookup

6.3.1 First level: In-Memory Storage

The first level of metadata storage refers to the in-memory storage on the *Skippy Data Scheduler*. The Skippy Data Scheduler should be able to access file metadata for the decision-making process quickly. The file metadata is necessary to prevent the Skippy Data Scheduler from searching all MinIO pods manually during scheduling time, leading to slower scheduling. The in-memory cache holds the storage index 6.2 instance, and it works as a first level cache. That means whenever the scheduler receives an incoming pod, data locality priority requests first the in-memory cache and retrieves the metadata from it. The storage index in the in-memory cache is updated in the following situations:

Scheduler startup Skippy scheduler loads the storage index directly from the cache. Once the scheduler has the storage index in memory, it will continually search for the file metadata at first.

Scheduling execution The update here happens on the occasion of non-existing information in the first layer as described above. During scheduling, more precisely at data locality priority execution, the scheduler requests a storage index update from the key-value cache. In the event of missing metadata information in both layers in-memory and key-value store, the scheduler ignores the Data Locality Priority execution. Section 5.3.2 details the Data Locality Priority.

6.3.2 Second level: Key-Value Cache

The second storage necessary for our skippy data-locality solution is an in-memory storage. In this storage we keep key information like file metadata available at runtime. This in-memory storage is heavily used throughout the many use cases described for skippy either for file metadata or also to store network metrics and bandwidth graph. Therefore, the necessity of a tool which is able to handle the intense data flow.

The second-level cache is a separated service that needs to be accessed via the network in our solution. Due to its high throughput of approximately GET 81000 per second and SET 110000 per second[5], we select Redis due to its fast key store access. Redis can be configured as a single storage node or as a distributed cache mechanism. The multiple configurations enable flexibility for the framework and also scalability if necessary. There is another component, Skippy Data Daemon, feeding this information to the Redis cache.

Redis is an open-source in-memory data store known in the community which also provides extensive support. The following Chapter 6 describe in detail how the file's metadata is collected and stored in a key-value format in Redis. Additionally, Redis stores network metrics which are used to build the bandwidth graph. This approach is explained in Chapter 7.

[5]https://redis.io/topics/benchmarks

6.4 Skippy Data SDK

Skippy Data SDK is a Python library that enables data locality in a serverless function. Generally, the SDK hides the complexity of the locality-aware data management system, which is necessary for serverless computing. It gives the developer simple storage APIs @*consume* and @*produce* but performs the operations with data-locality awareness. Data-locality awareness is beneficial when implementing serverless functions. Designed to download and upload files necessary, the SDK presents a feature to identify the best storage node where the file transfer should happen. Similar to the algorithm used in *skippy scheduler's* data-locality priority described in 5.3.2, before any file transfer, the SDK calculates which storage node be connected to.

The SDK reads a configuration file provided in the function as suggested in [6]. The tool is responsible for automatically downloading all the data necessary for the function's execution. Once the function execution is complete, the SDK uploads the produced data to the MinIO storage. Primarily, the Skippy Data SDK provides three main features, which are specified below. The decorators 4.6.1 trigger one of the file transfers either via storage node prediction 6.4.2 or via ephemeral storage 6.4.1.

6.4.1 Ephemeral Storage

Serverless functions typically store data in S3 buckets from cloud data centers. However, in edge computing, the constant cloud transfers might lead to high latency and high financial costs. On the other hand, data centers present on the edge network may struggle with limited resources. Considering that in an intense data transfer scenario, it might occur to have data only read and write once. Chained functions such as ML present many steps to achieve its final result. The data produced between the flow is only processed once and can be discarded afterward. As an example, we mention our ML workflow, which presents 3 steps: 1-pre-process, 2-train and 3-evaluate. The data produced by *1-pre-process* function is only read once by *2-train*, further *3-eval* reads the data produced by *1-pre-process* and *2-train*. This cycle repeats for every trained model. The trained model is the final target data, but all the data produced in the middle of the ML workflow already completed its life cycle, and it is no longer necessary. This one-time read/write data is called ephemeral data. However, this approach presents a drawback. In an edge computing environment, resources are limited. If disk I/O is already highly used at the moment of execution, it might present performance issues that can affect function's Task execution time (TET).

Often few functions are scheduled in the same node. The scheduling decision depends on all the priorities calculations executed by the scheduler described in previous **??**. To adjust to ephemeral data workflows, thus decreasing even more bandwidth usage, we store the produced data in temporary storage in the machine. In this temporary storage where the faas-function is executing, any future faas-function which belongs to the same chain can access the file directly from the temporary disk storage without transferring the file via the network.

Consume At the moment the function is executed, Skippy Data SDK searches first the temporary storage if the data is found there, the SDK reads the file directly from the disk. If the file is not present on the disk, the SDK transfers from one of the MinIO storage nodes. Figure 6.4 shows how the consumed data source is decided.

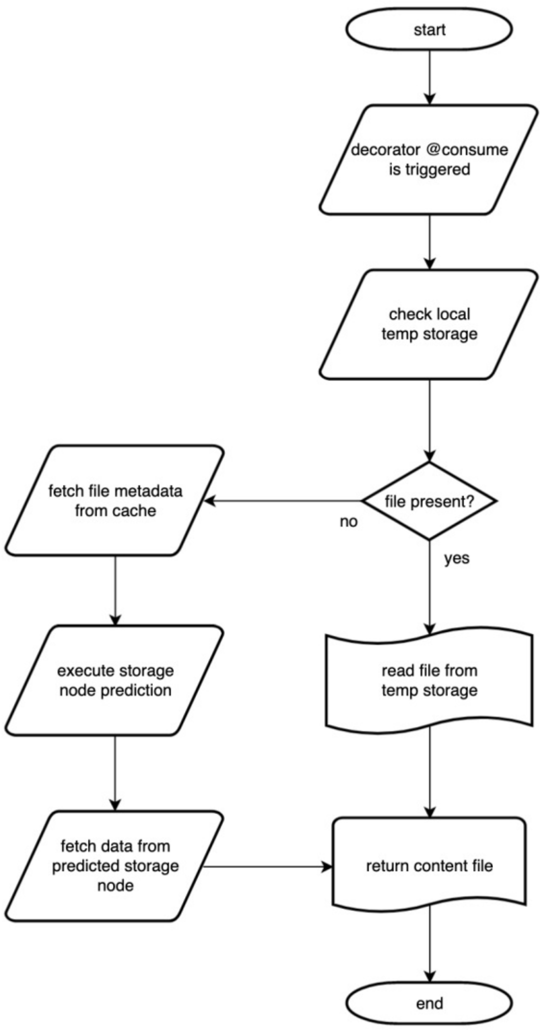

start
decorator @consume is triggered
check local temp storage
file present?
no
yes
fetch file metadata from cache
execute storage node prediction
fetch data from predicted storage node
read file from temp storage
return content file
end

Figure 6.4: Ephemeral Storage Flowchart

Produce The produced data is always duplicated in the temporary storage and the MinIO storage. At execution time, the faas-function which will consume this file might not even exist; the SDK is not sure if it will be deployed on this node or not, thus, the duplication in both storages.

The temporary storage has the goal to avoid unnecessary network transfers for ephemeral data. Thus, it contains data duplication from MinIO. In our scenario, we assume the data will be produced and consumed within a short period. Hence, the temporary storage is cleaned up regularly.

6.4.2 Storage Node Prediction

Files are also transferred from and to MinIO buckets which are placed in storage nodes within the cluster. In a serverless-edge environment, it is possible to have more than one MinIO storage instance. To transfer the files efficiently, the SDK needs to be aware of where each file's location, and in case of multiple instances, it needs to decide to which storage node should connect. Figure 6.5 details how the SDK execution flow happens.

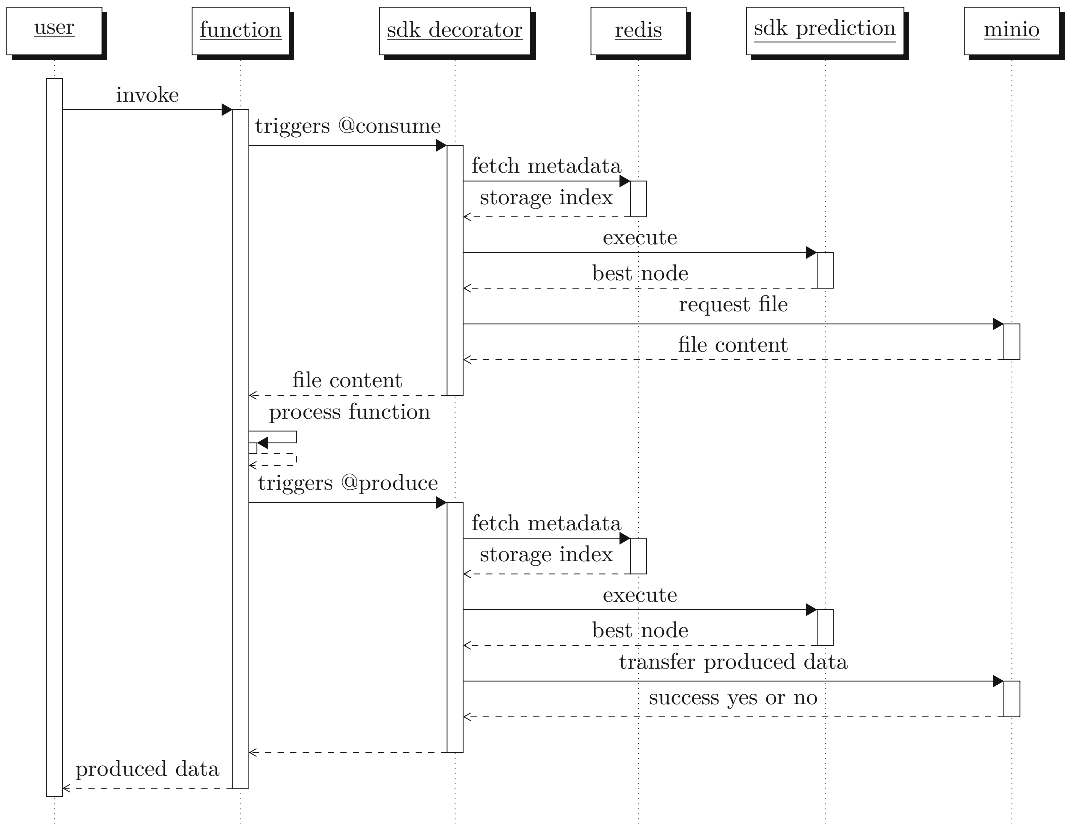


Figure 6.5: Skippy Data SDK

Metadata Retrieval

Skippy Data SDK relies on the metadata for its Storage Node Prediction calculations. Thus, we ensure the metadata is always available. Once the function's execution triggers the prediction, the SDK fetches the storage index 6.2 from Redis cache. As explained in Section 6.2, the storage index contains metadata from each file on the storage node. Thus, the storage index size depends on the number of files on the storage nodes. The

SDK requests each metadata by its key instead of the complete index download. Due to high throughput from Redis[6], the Skippy Data SDK can request one storage index key less than one millisecond. The metadata request is detailed in Section 6.4.2.

As this execution happens at runtime in a short-lived function, in-memory storage 6.3.1 does not apply, since after the execution, the function's Kubernetes pod might be destroyed and its in-memory data lost. Hence, the SDK only deals with key-value cache 6.3.2 to retrieve the metadata necessary for the storage node prediction.

Prediction

As soon as the library already has the file metadata, it calculates the closest node by executing the prediction algorithm. To keep consistency between scheduling and runtime, the SDK uses a similar approach applied in the Data Locality Priority algorithm as the scheduler detailed in Algorithm 5.1. Although there are many similarities between Skippy Data Scheduler and Skippy Data SDK, the Storage Node Prediction presents different characteristics from the Scheduler since the SDK searches for one specific storage node during runtime while the Skippy Data Scheduler searches all possible storage nodes. At the same time, the Data Locality Algorithm 5.1 scores all the storage nodes which contain the files. The storage node prediction finds the storage node which includes the file and presents higher bandwidth available. The detailed storage node prediction is displayed in Algorithm 6.1.

If this is the case, the SDK identifies and handles every single file independently. That means for each consumed or produced file, the SDK will predict the best storage node to transfer the file content.

Locality Priority

As in Skippy Data Scheduler described in Chapter 5, Skippy Data SDK prioritizes nodes on the edge network over nodes on the cloud. That means nodes on the cloud will only be accessed if there is no device on the edge that provides the same service or data. In the edge environment, the SDK searches for the best node, which provides better network traffic availability as displayed in pseudo-code 6.1.

[6]https://redis.io/topics/benchmarks

Algorithm 6.1: StorageNodePrediction

```
Result: Best storage node to transfer the requested file
Function predictBestStorageNode:
    Input : hostname
    Input : dataItem
    localities ← ["edge","cloud"];
    bestNode ← "";
    for locality in localities do
        storageNodes ←
         getStorageNodes(dataItem.bucket,dataItem.fileName,locality);
        for storageNode in storageNodes do
            bandwidth ← getBandwidth(node,storageNode);
            if bandwidth > maxBandwidth then
                maxBandwidth ← bandwidth ;
                bestNode ← storageNode ;
        if bestNode then
            break;
    return bestNode;
```

CHAPTER 7

Network Monitoring

In this chapter, we explain how network monitoring affects our Skippy Data Scheduler decisions. Additionally, we introduce network monitoring solutions used in our framework. Section 7.1 presents the problem and a proposed solution. Section 7.2 explains which metrics are necessary for our solution and which system we use to collect these metrics. In Section 7.3, we describe the bandwidth graph and propose two non-intrusive network monitoring approaches. Further, in Section 7.4, we compare the two proposed monitoring approaches. Additionally, we show the advantages and disadvantages of each technique. To conclude this chapter, we present in Section 7.5 Skippy Network, the network monitoring component responsible for creating and maintaining the bandwidth graph.

7.1 Overview

In our system's architecture presented in Chapter 4, the bandwidth graph enables the decision-making process explained in Chapter 5. Skippy Data Scheduler leverages the bandwidth graph knowledge to make scheduling decisions based on the current network availability. Further, the Skippy Data SDK, described in Section 6.4, uses the bandwidth graph to find the best storage node according to the bandwidth graph availability during runtime.

To achieve a reliable bandwidth graph without extra network overhead, we propose solutions with non-intrusive methods. Our proposed components collect metrics and generate the bandwidth graph reflecting the network traffic at one specific moment. Fig. 7.1 shows how these components work in our current system setup. The details are explained throughout this chapter.

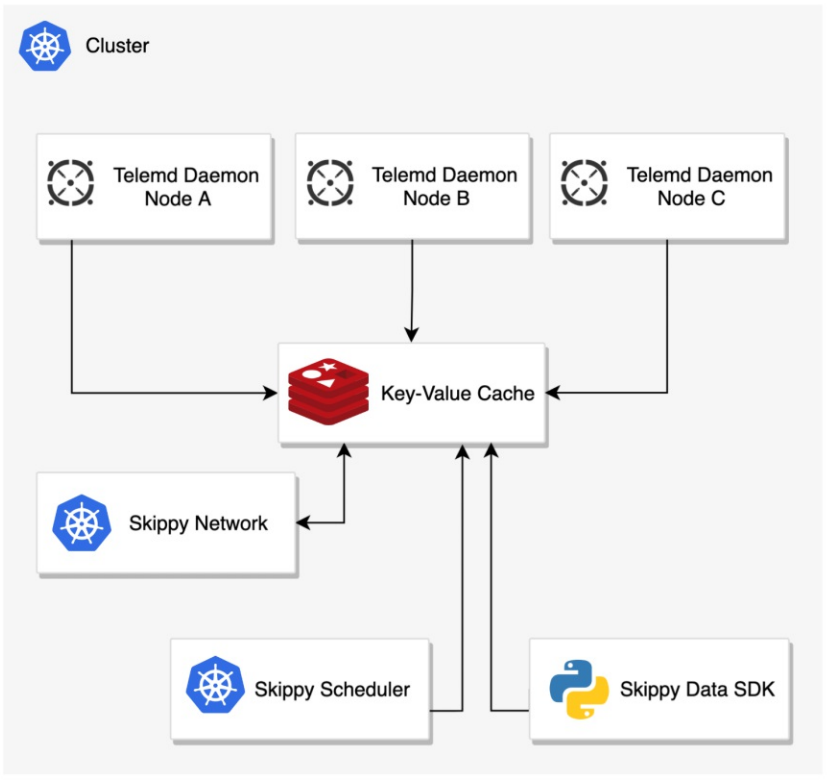


Figure 7.1: Network Framework Overview

7.2 Telemetry

According to [68], a telemetry system is fundamental for an environment where resources are limited. A serverless-edge-computing scenario struggles with limited computing resources. Additionally, there is a large amount of data being produced and transferred across the network. Hence, it is necessary to learn the network usage, and telemetry systems play an essential role in this path. Network data is a fundamental piece to have an accurate bandwidth graph and thus make reliable and precise scheduling and runtime decisions. To obtain system metrics such as network data, we choose to use the already existing telemetry system Telemd[1]. Telemd is a telemetry system that provides fine-grained system data. Telemd stores its data on Redis caching system, which in this specific case also needs to be running and available on the cluster.

7.2.1 Telemd: Ouf-of-box Metrics

Telemd has system-ready metrics about a machine and its usage such as CPU, RAM, disk and frequency. Most important for this monitoring, Telemd provides an overview of

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

the node's network I/O rates in real-time. This metric enables our system to estimate network usage at one specific moment.

Rx and Tx

Rx represents the amount of bytes received by a specific network interface while Tx is the amount of bytes sent by a particular network interface [69]. The Linux kernel stores these statistics in the paths

```
sys/class/net/<interface>/statistics/rx_bytes
sys/class/net/<interface>/statistics/tx_bytes
```

for `rx` and `tx` respectively.

Since one device can have many network interfaces, *telemd*1 provides `rx` and `tx` metrics for every interface present on the machine. As the linux statistics data does not provide any time frame, Telemd1 calculate `rx` and `tx` in Bytes/s for a random interface e.g. `eth0` as follows:

```
start=/sys/class/net/eth0/statistics/rx_bytes
sleep 1s
end =/sys/class/net/eth0/statistics/rx_bytes
rx_bytes_per_second = end - start
```

The same approach is used to obtain `tx` in Bytes/s.

7.2.2 Telemd: Add-on Metrics

In addition to the provided metrics, we slightly modified the tool to supply also the following metrics:

Active Network Device

The Skippy framework needs to obtain statistics from the correct network interface. The metrics collection is only possible if one knows the primary network interface, in other words, the interface with the highest route priority. Therefore, this metric is an essential addition to the telemetry system. This is obtained via the Linux package `route` CLI as displayed below.

```
route | awk 'NR==3{print $8}'
```

It is crucial to notice that for this metric, we assume the node is not connected to any virtual interface such as Virtual Private Network (VPN) and network bridge.

Netspeed

Netspeed represents the connectivity speed of the device. This metric only tells the maximum speed the machine can achieve in a specific network interface. It considers only the active network interface as described above. In this context, it was possible to provide connectivity speed for two types of network connection:

- **Ethernet** Metrics read from `/sys/class/net/<interface>/speed`.
- **Wi-Fi** This metrics is obtained via linux package command line:

```
$ iw dev <interface> link
```

7.2.3 Telemd Daemon Deployment

Since Telemd collects data from one specific machine only, we need to ensure that Telemed will run automatically on every node. This is achieved by using the Kubernetes daemon feature. The Telemd Kubernetes daemon deployment is defined in Listing 7.1.

7.3 Bandwidth Graph

According to [70], the bandwidth graph is the result of G=(V,E) where V represents the vertices, in our case, the nodes, while E stands for edges or communication between nodes. The bandwidth graph G displays the current network status, its utilization, and its availability. The throughput β available on a certain node A is described in [35] where φ and ψ represent input and output network throughput respectively while δ stands for max network availability. This is displayed in Eq. (7.1).

$$\beta_A = \delta_A - (\varphi_A + \psi_A) \tag{7.1}$$

As stated in [64], the bandwidth graph β is the minimum available bandwidth between source node A and target node B bandwidth capacity as displayed in Eq. (7.2). Thus, we propose two non-intrusive methods described in the following sections.

$$\beta_{A,B} = \min(\beta_A, \beta_B) \tag{7.2}$$

Many of the solutions on the market use intrusive methods such as *iperf* which provide very accurate rates, but it also adds significant overhead to the network. In serverless edge computing, the constant data transfers overload the network; thus, we choose to avoid intrusive methods not to add extra load during metrics collection.

```yaml
apiVersion: apps/v1
kind: DaemonSet
...
spec:
...
  template:
    metadata:
      labels:
        name: skippy-telemd
    spec:
      hostNetwork: true
      tolerations:
      - operator: Exists
        effect: NoSchedule
      containers:
      - name: telemd
        image: keniack/telemd:0.2
        imagePullPolicy: Always
        securityContext:
          privileged: false
          capabilities:
            add: ["NET_ADMIN"]
        env:
        - name: telemd_redis_host
          value: <redis-cluster-ip>
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
```

Listing 7.1: Telemd Daemon Deployment Specification

7.3.1 Estimated Bandwidth Graph

In this approach, we use *netspeed* to calculate the available network speed. Since the *netspeed* metric only represents the maximum speed that a specific network interface can achieve, thus, the bandwidth graph created upon this metric is not accurate but only estimated.

The estimated bandwidth graph is built upon three metrics: `speed`, `rx` and `tx`. These metrics are collected by Telemd1 as described in the previous section. As shown in Algorithm 7.1, the graph is built by calculating netspeed minus I/O network interface rates. Although not precise, the bandwidth graph provides a substantial overview of

Algorithm 7.1: EstimatedBandwidthGraph

```
   Result: Available bandwidth between nodes in the cluster
1  Function getBandwidthGraph:
       Input : nodes
2      edgeCapacity ← 0;
3      bandwidthGraph ← Dict[string][string];
4      for source in nodes do
5          for target in nodes do
6              if source != target then
7                  edgeCapacity ← calculateEdgeCapacity(source,target) ;
8                  bandwidthGraph[source][target]← edgeCapacity;
9      return bandwidthGraph;
10 Function calculateEdgeCapacity:
       Input : sourceNode
       Input : targetNode
11     sourceCapacity ← calculateNodeCapacity(sourceNode);
12     targetCapacity ← calculateNodeCapacity(targetNode);
13     return min(sourceCapacity,targetCapacity);
14 Function calculateNodeCapacity:
       Input : node
15     nodeCapacity ← 0;
16     link ← get Node Link from telemd;
17     rx ← get Node Rx from telemd;
18     tx ← get Node Tx from telemd;
19     nodeCapacity ← link - (rx + tx);
20     return nodeCapacity;
```

the network status without the overhead from intrusive methods. In the experiments Section 8.4, it is possible to see how the bandwidth graph varies according to the network rate changes.

7.3.2 Precise Bandwidth Graph

Another approach to generating the bandwidth graph is to collect download and upload rates of function executions. As soon as an OpenFaaS event triggers the function, it will download the files necessary for the function's input from a storage node. By the end of its execution, it uploads the produced file also to a storage node. The storage nodes chosen by download and upload might not necessarily be the same one. The decision process is detailed described in Chapter 6. Algorithm 7.2 explains how *skippy network* uses download and upload to generate a more precise bandwidth graph.

Algorithm 7.2: PreciseBandwidthGraph

```
   Result: Stores speed rate from node to storage node
1  Function downloadFile:
       Input : fileMetada
       Input : storageNode
2      start ← time.now();
3      download File From Node;
4      end ← time.now();
5      bandwidthGraph[hostname][storageNode] ← $\frac{\text{fileMetada.size}}{end-start}$;
6      store bandwidth in redis;
7  Function uploadFile:
       Input : fileMetada
       Input : storageNode
8      start ← time.now();
9      upload File From Node;
10     end ← time.now();
11     bandwidthGraph[hostname][storageNode] ← $\frac{\text{fileMetada.size}}{end-start}$;
12     store bandwidth in redis;
```

This technique provides an accurate overview of the network at a specific moment. However, it might affect *Skippy Data Scheduler's* initial phase described in Chapter 5. Given that first it is necessary to upload or download to calculate the rates, this will directly affect *Data Locality Priority* Section 5.3.2 which initially will not have any bandwidth data for its calculations. Thus, *Data Locality Priority* will only start its prioritization after enough OpenFaaS functions have been executed and consequently, download and upload rates used for the bandwidth graph collected.

This approach only generates partial bandwidth graphs as downloads and uploads only happen between nodes and storage nodes. In such cases, Skippy Data SDK, described in Section 6.4, can not evaluate the network rates because there is no file transfer between the nodes. However, the partial bandwidth graph is sufficient for the use cases described in this project. The decision-making process focuses on the search for the bandwidth for file transfers. Therefore, this procedure is entirely valid for our use case, even though it does not generate a complete graph.

7.4 Comparison of Approaches

Each of these methodologies has specific characteristics, and both approaches provide satisfactory results. The Table 7.1 lists the main differences between *Estimated Bandwidth Graph* and *Precise Bandwidth Graph* as well as advantages and disadvantages of each method.

	Estimated Bandwidth Graph	Precise Bandwidth Graph
Advantages	• Non-intrusive. • Full availability. • Full network graph. • It considers every ongoing network communication.	• Non-intrusive. • Accurate network rates.
Disadvantages	• It uses network's interface maximum communication speed. Therefore, it does not provide accurate bandwidth availability.	• Partial availability, it will only be fully available when enough download and upload requests have been executed. • Network overload might differ from the previous collected rate which can mislead the decision-making processes. • Partial network graph between nodes and storage nodes only.

Table 7.1: Bandwidth Graph Approach Comparison

7.5 Skippy Network

Considering that the network monitoring is entirely independent of the other components like *Skippy Data Scheduler* detailed in Chapter 5, we created a new tool called *Skippy Network*. This tool is responsible for bandwidth graph generation. To do that, it reads the network metrics collected by Telemd. Once *Skippy Network* generates the graph, it stores the graph in the caching system Redis. To deploy skippy-netwwork in the cluster, we created a *kubernetes* deployment defined in Listing 7.2.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: skippy-network
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: skippy-network
  template:
    metadata:
      labels:
        app: skippy-network
    spec:
      nodeSelector:
      containers:
      - name: skippy-network
        image: keniack/skippy-network:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 5002
        env:
        - name: redis_host
          value: <redisIp>
        - name: registry_link
          value: "10"
        - name: cloud_link
          value: "20"
```

Listing 7.2: Skippy Network Deployment Specification

CHAPTER 8

Evaluation

In this thesis, we develop a data-locality scheduler enhancement to Skippy Scheduler described in Chapter 5. To achieve that, we created a storage index containing the file's metadata, described in Chapter 6. Additionally, we created a network monitoring tool to provide real-time bandwidth availability, described in Chapter 7. Furthermore, as the network usage may differ between scheduling and runtime, we introduced a framework to identify the shortest route for a file transfer during the serverless function execution runtime, described in Section 6.4.

In this chapter, we present the results obtained throughout this project. In Section 8.1, we present the testbed and the simulation environment where we collect results for the evaluation. In the simulation environment, we describe different scenarios applied in this evaluation. Further, empirical measurements give an overview of experiments and evaluation indicators of our Skippy Data Scheduler and Skippy Data SDK. The final sections 8.2, 8.3 and 8.4 describe performed experiments and detailed analysis from Skippy Data Scheduler, Function Runtime and Network Monitoring respectively.

8.1 Methodology

This section describes the environment in which we evaluate the framework, the test workflow used, and which indicators we use as a baseline to evaluate our framework solution.

8.1.1 Environment

To evaluate our framework developed in this thesis, we perform experiments in two different environments: a simulation and a testbed of real edge computing hardware. The testbed assists the proper configuration and engineering problems during the development, while the simulations outline issues in different scenarios like urban sensing vs. industrial

IoT. Additionally, a simulation allows us to use a different number of devices in the cluster, such as 100 or 1000 nodes.

Testbed

To test our framework end-to-end, we created a testbed shown in Figure 8.1, where we could reproduce actual use cases scenarios and achieve results as close as possible to a real production environment. Our testbed contains heterogeneous devices characteristics that enable the developed framework to schedule using each node's characteristics. Therefore, a heterogeneous testbed was composed of different SBC. The SBC technical specification can be seen in the Table 8.1. A testbed setup used is available in our git repository[1].

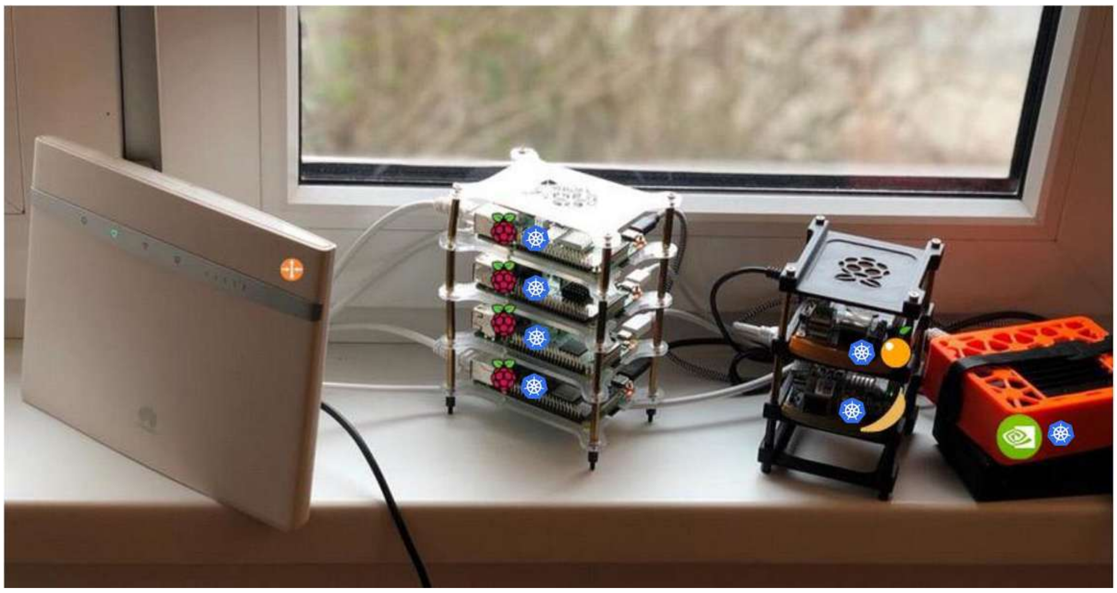


Figure 8.1: Testbed

SBC	Arch	Name	CPU	Accelerator	RAM	Ethernet	Storage
Nvidia Nano	aarch64	jambo	4x Cortex-A57 @1.43GHz	128-core Maxwell GPU	2GB	1 Gbps	SD Card
RPI 4B	arm32	ananas	4x Cortex-A72 @1.5GHz	N/A	4GB	1 Gbps	SD Card
RPI 4B	arm32	papaya	4x Cortex-A72 @1.5GHz	N/A	8GB	1 Gbps	SD Card
Banana Pi-M3	arm32	banana	8x Cortex-A7 @1.8 GHz	N/A	2GB	1 Gbps	SD Card
RPI 3B+	arm32	acerola	4x Cortex-A53 @1.4GHz	N/A	1GB	1 Gbps	SD Card
RPI 3B	arm32	guava	4x Cortex-A53 @1.2GHz	N/A	1GB	100 Mbps	SD Card
OrangePi PC	arm32	orange	4x Cortex-A7 @1.6GHz	N/A	1GB	100 Mbps	SD Card

Table 8.1: Testbed Device Specification

[1]https://github.com/keniack/testbed

Simulation

Running the experiments on the testbed provides an overview of the end-to-end flow. However, we would like to evaluate the project in different scenarios with different devices setup. Therefore, we need to simulate other clusters scenarios and devices setup. For the simulations we use *edgerun faas-sim*[2] which allows us a detailed configuration environment. Among the many features available in the simulator, we created test suites using the necessary characteristics of this project which is discussed in Section 8.1.3. The heterogeneous devices environment is formed as described in the Table 8.2 similar approach used in [71].

Device	Arch	CPU	Accelerator	RAM	Storage
VM	x86	4x Core 2 @3GHz	N/A	8GB	HDD
XeonCPU	x86	4x Xeon E-2224 @3.44GHz	N/A	8GB	HDD
Intel Nuc	x86	4x Intel i5 @2.2 GHz	N/A	16GB	NVME
Nvidia Nano	aarch64	4x Cortex-A57 @1.43GHz	128-core Maxwell GPU	2GB	SD Card
Nvidia TX2	aarch64	4x Cortex-A57 @2GHz	256-core Pascal GPU	2GB	eMMC
Coral DevBoard	aarch64	4x Cortex-A53 @1.43GHz	Google Edge TPU	1GB	eMMC
RockPi	aarch64	2x Cortex-A72, 4x Cortex-A53	N/A	2GB	SD Card
RPI 4	arm32	4x Cortex-A72 @1.5GHz	N/A	1GB	SD Card
RPI 3	arm32	4x Cortex-A53 @1.4GHz	N/A	1GB	SD Card

Table 8.2: Simulation Device Specification

Scenarios

To obtain close results to a realistic environment, we choose two different scenarios with different infrastructure configurations to evaluate our project. In both scenarios, we randomly assign around 10 % of total devices as storage nodes. As the scenarios are populated with different specifications and localities, the storage nodes may have different storage capabilities. We place the storage nodes in the edge and cloud network to analyze the impact of network traffic on the edge and cloud scenarios. This hybrid edge and cloud network gives us an overview of the different approaches used in this project. Additionally, it shows how we can improve the network traffic and the financial costs. We used a similar network connection configuration for both scenarios as our implementation is focused on network traffic and data exchange between edge and cloud environments. Thus, we can compare our data-locality mechanism performance on different topologies.

The multiple scenarios are available via Ether[3]. Faas-sim [4] leverages from Ether the ability to set up different topology configurations to obtain results close to a real system. The scenarios are described below. Our evaluation scenarios are influenced by Skippy experiments described in [9].

[2]https://github.com/edgerun/faas-sim
[3]https://github.com/edgerun/ether
[4]https://github.com/edgerun/faas-sim

S1: Smart City As the adoption of the smart city concept increases, solutions are arising from real-time data processing on the edge [31]. The urban sense idea shows sensors attached to IoT gateways which process the data in cloudlets [72]. Following the principle presented in [9], we assume a city populated with sensor nodes and cloudlets. Sensor nodes are represented by SBCs and have limited processing capacity while cloudlets are characterized by Intel NUC and Intel Xeon. To analyze the performance on edge and cloud networks, we also introduce cloud VM. Additionally, we also add VM in the cloud network as part of the ecosystem. From the topology point of view, we assume that each city district is one edge network, and the storage nodes are spread across the city. Table 8.3 displays a detailed setup of our smart city scenario.

Device	Locality	Population %	LAN	Internet
VM	cloud	10	1Gbps	200 Mbps
XeonCPU	edge	10	1Gbps	100 Mbps
Intel Nuc	edge	10	1Gbps	200 Mbps
Coral DevBoard	edge	20	100Mbps	50 Mbps
RockPi	edge	12	100Mbps	50 Mbps
RPI 4	edge	12	100Mbps	50 Mbps
RPI 3	edge	18	100Mbps	50 Mbps

Table 8.3: S1: Smart City Device Constellation

S2: Industry 4.0 The fourth industrial revolution is quickly changing manufacturing companies. The idea of smart manufacturing is bringing the edge network to the factories' production line [24]. In this scenario, we assume a web of factories where each factory represents one edge network. Each factory is composed of provider-managed on-premises cloud centers. Our industrial simulation is composed of SBCs, Intel Xeon, and Intel NUC on the edge while VMs are placed in the cloud as can be seen in table 8.4.

Device	Locality	Population %	LAN	Internet
VM	cloud	44	1Gbps	100 Mbps
XeonCPU	edge	10	1Gbps	100 Mbps
Intel Nuc	edge	10	1Gbps	200 Mbps
Nvidia TX2	edge	9	100Mbps	50 Mbps
RockPi	edge	9	100Mbps	50 Mbps
RPI 4	edge	9	100Mbps	50 Mbps
RPI 3	edge	9	100Mbps	50 Mbps

Table 8.4: S2:Industry 4.0 Device Constellation

8.1.2 Test Workflow

To simulate a data intense scenario, we designed a *scikit-learn*[5] python workflow to simulate a ML scenario. The workflow is composed of three functions as shown in

[5]https://scikit-learn.org

Fig. 8.2.

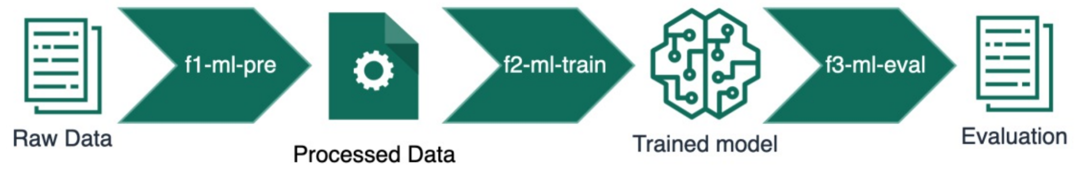


Figure 8.2: Test ML Workflow

Each of these functions consume and produce data which are transferred from a MinIO storage node. Fig. 8.3 shows how functions communicate with the storage nodes in order to transfer the data.

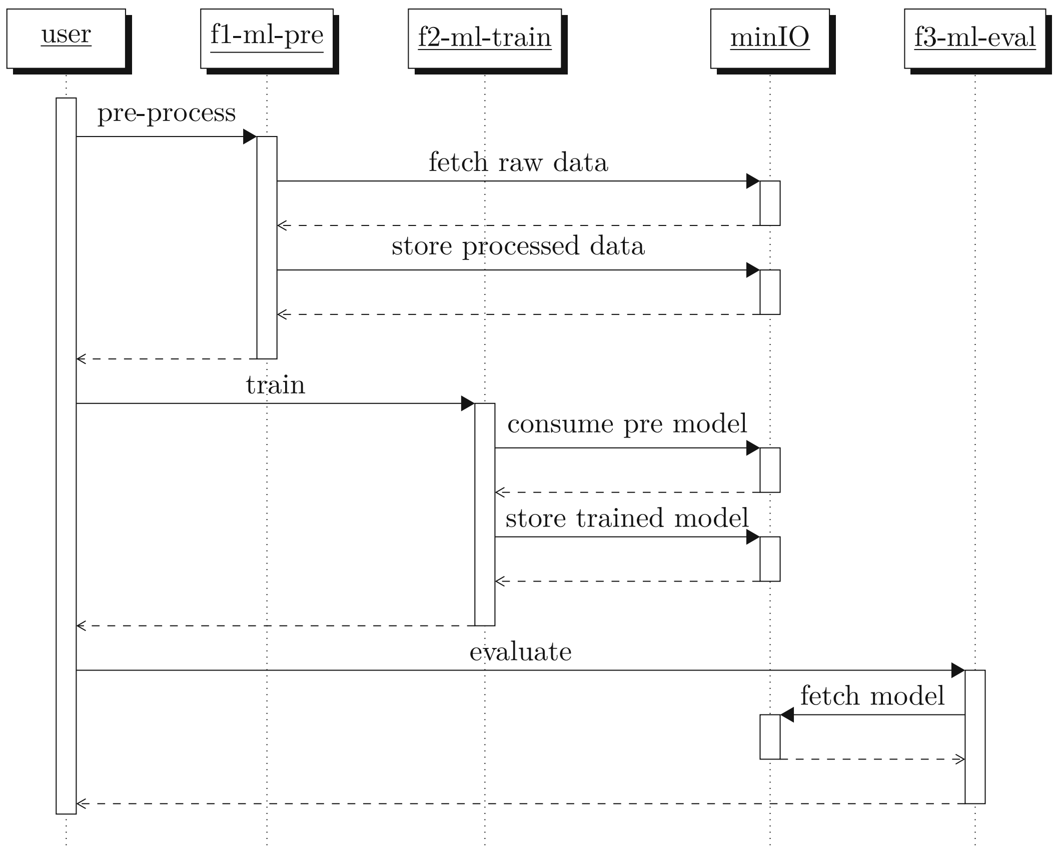


Figure 8.3: Machine Learning Test Flow

Function f1-ml-pre loads the raw data and treats the data to be processed. This function reads csv files from the stored nodes and its result should be uploaded again.

```
data:
  consume:
    - pre-processed.rawdata.csv
  produce:
    - trained-models.pre-model.npy
```

Listing 8.1: Function f1-ml-pre skippy.yml

Function f2-ml-train is a training function. f2-ml-train has as input the processed data produced by previous function. This data is trained and returned as a result to be uploaded.

```
data:
  consume:
    - trained-models.pre-model.npy
  produce:
    - trained-models.model.npy
```

Listing 8.2: Function f2-ml-train skippy.yml

Function f3-ml-eval is the last step, it downloads the model created by f2-ml-train function and evaluates how precise this model is and further serves the model. As this is the last step and the model is already stored, this function returns the evaluation results but does not produce any data.

```
data:
  consume:
    - trained-models.pre-model.npy
```

Listing 8.3: Function f3-ml-eval skippy.yml

In our set of tests, we use the different data files size for each function as displayed in Table 8.5.

8.1.3 Empirical Measurements

This project deals with a solution that acts in two different execution times in the workflow: scheduling and runtime. To evaluate the project, we defined different indicators

Function Name	Consume	Produce	Image Size
ml-f1-pre	230 MB	230 MB	300M MB
ml-f2-train	850 MB	850 MB	320M MB
ml-f3-eval	100 MB	n/a	320M MB

Table 8.5: Test Workflow Data Size

to be measured during the experiments as shown in the “Expected Results” column in Table 8.6. These indicators provide an insight to conclude whether the developed tools present satisfactory results applicable in real systems. The following subsections define the expected results of each experiment.

Experiment	Setup	Expected Results
Skippy Scheduler: **Placement Quality**	1100 nodes; 50 pods placement; 100 requests	Increased Function execution time (FET), maximization edge traffic and minimization of egress and ingress in the cloud network traffic.
Skippy Scheduler: **Scheduling Latency**	1100 nodes; 500 pods placement	At least half of the scheduling **latency** from the default Skippy implementation.*
Skippy Scheduler: **Scalability**	5000 nodes; 500 pods placement	At least half of the scheduling **throughput** from the default Skippy implementation.*
Function Runtime - Skippy Data SDK: **FET**	1100 nodes; 50 pods placement; 100 requests	Increased FET.
Function Runtime - Skippy Data SDK: **Network Traffic**	1100 nodes; 50 pods placement; 100 requests	Maximization of edge network traffic and minimization of egress and ingress in the cloud network traffic.
Function Runtime - Skippy Data SDK: **Financial Costs**	1100 nodes; 50 pods placement; 30000 monthly requests	Decrease of financial costs with cloud services.

* Given the increased complexity of the scheduler due to data locality implementation, we consider this loss a tradeoff to achieve data locality.

Table 8.6: Experiments Summary

Skippy Scheduler

To analyze our data-locality enhanced Skippy, we compare it with its default behavior to conclude whether our solution is efficient for a real case application. The results present the efficacy of the designed system architecture discussed in Chapter 4 as in our data-locality solution; the scheduler is dependent on other components, e.g., *Skippy Data Daemon* which keeps the storage index up to date. The efficiency of our solution is the outcome of the analysis between the components described as follows:

- **Default Skippy**: we use Skippy default implementation [11, 9] as a base measurement point. Skippy's default implementation gives an overview of whether our development is efficient enough for real case scenarios.
- **Skippy Data**: This is our default scheduler implementation with data locality related priorities (*DataLocalityPriority,FunctionChainPriority*) described in chapter 5. Additionally, the scheduler uses the storage index described in Chapter 6. The default storage index keeps a tree of metadata files where storageNode, bucket and fileItem is quickly accessible during scheduling time. In this default implementation, during scheduling time, *DataLocalityPriority* Section 5.3.2 resolves every download/upload file by searching in the fastest StorageNode available.
- **Optimized Skippy Data**: This is an improved performance feature for the storage index. As in the default implementation, the scheduler still needs to access the bandwidth graph and find the best storage node, increasing the latency in overloaded cases. Thus, we created an asynchronous full storage index where all the feasible nodes are already previously calculated and stored. In other words, we create a full index matrix beforehand $[feasibleNode, bucket, item] = storageNode$ where the result storageNode already represents the fastest storage Node available in the network. The stored matrix results in low latency storage index access during scheduling time without any additional search necessary. As the "full storage index" matches all the feasible nodes available in the cluster, it has high latency during its creation than the default *storage index* Chapter 6. However, it improves the scheduler's latency and scalability, as can be seen in the following experiments in Section 8.2.

Every tool is measured according to Skippy's default implementation. As the components have specific roles, we consider the following key results displayed. The placement quality, performance, and scalability measure the scheduling efficiency. An efficient scheduler should consider incoming pods and total worker nodes present in the cluster.

Placement Quality Function placement plays a crucial role in this solution. We consider a good service placement if our scheduler placement improves the function execution time. Besides scheduling, our solution also includes a SDK component, which is responsible for storage node prediction during runtime. However, to measure the placement quality, we do not consider the runtime SDK. Hence, we can directly compare the function time runtime according to the function placement during scheduling.

Scheduling Latency The scheduling latency gives the scheduler a performance indicator. We want to see how the scheduler performs with different numbers of pods in the queue in these experiments. The algorithm's complexity reflects in the scheduler's performance. As discussed in Section 2.2.2, the scheduling complexity depends on the complexity of each priority. Considering that the bandwidth graph is directly accessed, default Skippy has a Data Locality complexity of $O(1)$ [11]. As our implementation resolves every file, our data locality implementation presents a complexity of $O(f)$ where f is the total number of files consumed and produced by the serverless function. This performance is measured by the total scheduling time in the experiments.

Scalability We want to ensure that the changes implemented in the Skippy scheduler present a good scheduling throughput in a highly-populated cluster. We consider a good scheduling throughput if Skippy Data Scheduler shows half of the scheduling throughput from its default Skippy implementation. As we know, our data-locality priorities increase the scheduler complexity; a lower scheduling throughput is a tradeoff to achieve the data locality. The experiments aim to show the degradation of Skippy Data pods placement per second according to the number of nodes present in the cluster. Furthermore, it also displays how the full index in a *Optimized Skippy Data* can improve the scalability of the project and whether such an index is feasible or not. In the experiments, we use Kubernetes maximum worker nodes number of 5000 in a single cluster [73].

Function Runtime: Skippy Data SDK

Once the function is placed in a pod by the scheduler, the function still needs to download/upload the necessary data from one of the storage nodes. Thus, function runtime experiments are a key factor in evaluating whether proposed runtime solutions are considered enough for a data-intensive edge-cloud environment. As the function runtime is handled by our Skippy Data SDK 6.4, we evaluate whether the Skippy Data SDK improves the function execution time to the scheduling data-locality decisions. This evaluation is measured by considering the following runtime solutions as described below.

- **Prediction** is a Skippy feature where the SDK predicts the best storage node for the data to be transferred either via download or upload. Due to Skippy Data SDK edge priority, storage nodes on edge have higher priority even when they present lower network availability. The consumed data is not present in the machine; thus, the function must retrieve the data via network transfer from one of MinIO buckets present in the storage nodes. The storage node prediction is described in Section 6.4.2.
- **Random** storage node assignment is the default function behavior. Random storage node assignment also discards edge and cloud locality; the experiment picks the first storage which holds the metadata necessary for the file transfer.
- **Ephemeral** refers to the temporary disk storage used in the pod where the function is placed. The function leverages from its disk to quickly access the data without

transferring the data from the storage node via the network. The ephemeral data storage is a solution to store the data in the current node temporarily; thus, we do not recommend using it as storage. As an example if $functionA$ and $functionB$ are hosted in the same worker $nodeX$ and $functionB$ consumes data produced by $functionA$. Whenever $functionA$ is executed, it stores its produced data in temporary disk storage in $nodeX$. Later during $functionB$ execution, the consumed data for this function is already present in $nodeX$, and there is no network transfer necessary. The detailed runtime functionality is explained in subsection 6.4.1 while subsection 32 explains how the scheduler handles chained functions so that the functions can avoid data transfer by leveraging time from the ephemeral storage.

Function Execution Time In this set of experiments, we want to show how the Skippy data SDK performs in all different solutions described below. We also show results from the different scenarios S1: Smart City and S2: Industry 4.0 described in 8.1.1. The function is already placed by the scheduler, which has no longer influences the runtime. Once the serverless function is triggered, it needs to download the necessary data, and by the end of the function execution, it needs to upload produced data. These experiments aim to evaluate whether the storage node prediction and ephemeral storage can improve the function execution time.

Network Traffic One of the successful results of our project is to ensure that in a data-intensive scenario, the edge network is efficiently maximized. In contrast, the ingress and egress of the cloud network are minimized. The optimization of network traffic avoids network overload while decreasing financial costs with cloud resources. In the experiments, we show how the Skippy Data SDK acts to increase the edge network traffic, thus bringing processing closer to the end devices. To avoid network overload, the SDK balances the overload due to our network monitoring tools which provide access to an updated bandwidth graph.

Financial Costs Following the network experiments, we estimated monthly costs assume daily requests are executed. For this section, we assume the cloud resources are hosted by external providers, e.g., AWS,Google Cloud Platform (GCP) and Azure. In cases where we simulate on-premises cloud, e.g., S2: Industry 4.0, we assume a provider manages the premises cloud platform; thus, it generates financial costs. In the experiments, we assume that the cloud resources generate additional economic costs, regardless of on-premises or provider-hosted. For these experiments we use AWS[6] prices as baseline. According to the AWS pricing model, the financial costs generated by these experiments come mostly from AWS S3 storage and AWS egress network traffic.

Network Monitoring The experiments for this section aim to evaluate the development tool described in Chapter 7. As Faas-sim uses Ether to simulate different topologies

[6]https://aws.amazon.com/s3/pricing/

which can be applied in real scenarios, we choose to evaluate the network monitoring by experiments in our testbed described in Table 8.1. The small setup of a testbed assists the deep understanding of each proposed approach. In this section we discuss the difference between *EstimatedBandwidthGraph* and *PreciseBandwidthGraph* and in which situations they are better suitable for.

8.2 Skippy Data Scheduler

This section describes the experiments performed to evaluate our data locality implementation of the Skippy scheduler. We use Skippy default's implementation as a reference point to be measured and compared. The scheduler should provide **Placement Quality** by improving FET, it should have a similar or better **Performance** than Skippy default's implementation, and it should display **Scalability**. That means it should efficiently handle heavily populated node clusters.

8.2.1 Placement Quality

In this section, we want to analyze the scheduler's placement quality. All the results shown in Fig. 8.4 are executed in a cluster containing 1100 worker nodes or feasible nodes and scaled to 50 pods placement. Additionally, the experiments are performed in two different scenarios $S1$ and $S2$ as described in Section 8.1.1. In these experiments, we want to analyze whether data locality during scheduling affects the FET during runtime. The topology of total feasible nodes created for each scenario is randomly created following the specification described in the previous section.

Our proposed solution also contains a framework that resolves the data locality problem at runtime. Nevertheless, to analyze the solo impact of the scheduler, we need to provide the same runtime behavior to be able to compare the measured schedulers. Thus, during runtime, the function uses the first storage node available, which contains the file disregarding bandwidth availability or edge/cloud locality. The runtime solution (Skippy Data SDK) is further discussed in a different set of experiments in the next section.

Default Skippy In this test use case, we ignore the *DataLocalityPriority*. This scheduling uses mainly capacity and resource priorities to place the function. Skippy priorities are described in detail in Chapter 5. Even though the same functions are performed in $S1$ and $S2$, we notice a slightly lower FET in $S2$. This faster execution is due to the different devices' clusters which form $S2$. In $S2$, we have devices with higher capacity than $S1$, which results in lower FET as can be seen in all the experiments displayed in 8.4. Furthermore, we also notice a lower FET in each scenario no matter which function is executed. Figure 8.5 shows how the scheduling affects the network usage during its runtime. This scheduler has no data locality; thus, the network traffic depends on the random choice of storage nodes to transfer the files during execution time. As described in Table 8.4 $S2$ has a higher number of cloud devices than $S1$ (Table 8.3) which explains why $S2$ has higher cloud traffic than $S1$.

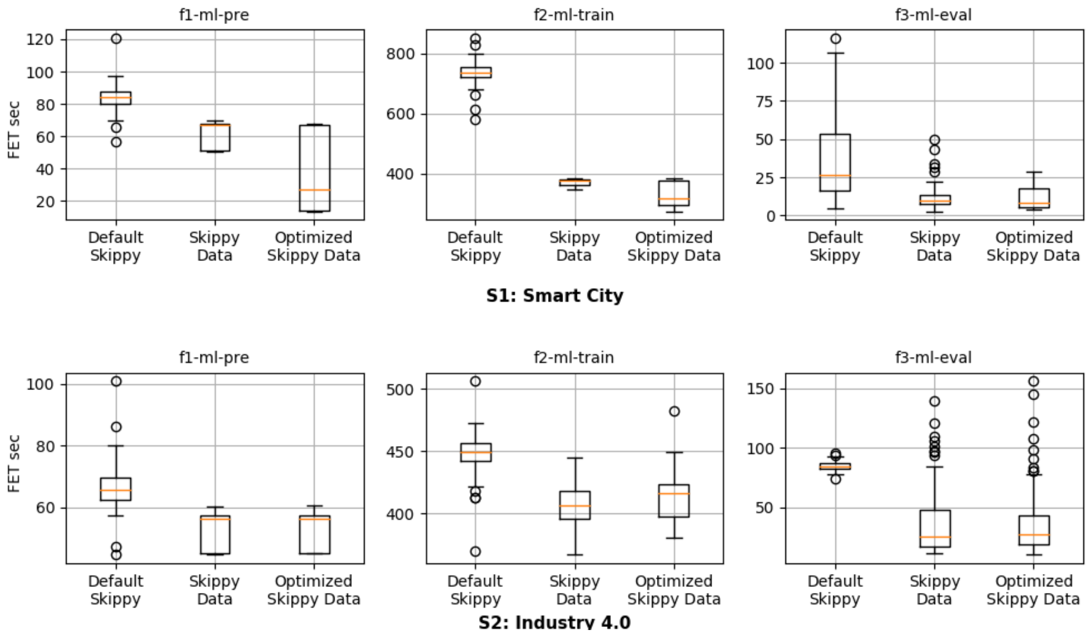


Figure 8.4: Skippy Scheduling Placement Quality

Skippy Data refers to the default data locality implementation described in Section 5.3.2. In comparison with *Default Skippy*, we notice some impact of this scheduling decision with/without data locality awareness. As an example, if we take a look at $S1$ and $S2$ device specification (Table 8.3 and Table 8.4 respectively), in a data-intensive environment the function placement has a major impact. In every scenario, data transfers are directly affected if the function is placed in a node with a link connection of 100 Megabits per second (Mbps) or 1 Gigabits per second (Gbps). A lower data transfer leads to lower FET. This effect can be seen in $S1$ and $S2$ for all the functions in the experiment of Fig. 8.4. The network traffic is shown in Fig. 8.5. As it is possible to notice, edge network usage increases with simultaneous requests while cloud network usage presents insignificant amounts.

Optimized Skippy Data is a scheduler's storage index improvement. It should achieve the same data locality decision as *Skippy Data* but in a better scheduler time execution. It is possible to see in every scenario in Fig. 8.4 that this optimized scheduler barely affects the FET. The same happens for the network usage displayed in Fig. 8.5. As we can see, the optimization reaches similar results as *Skippy Data*, edge network usage increases in both scenarios while cloud usage remains at low values.

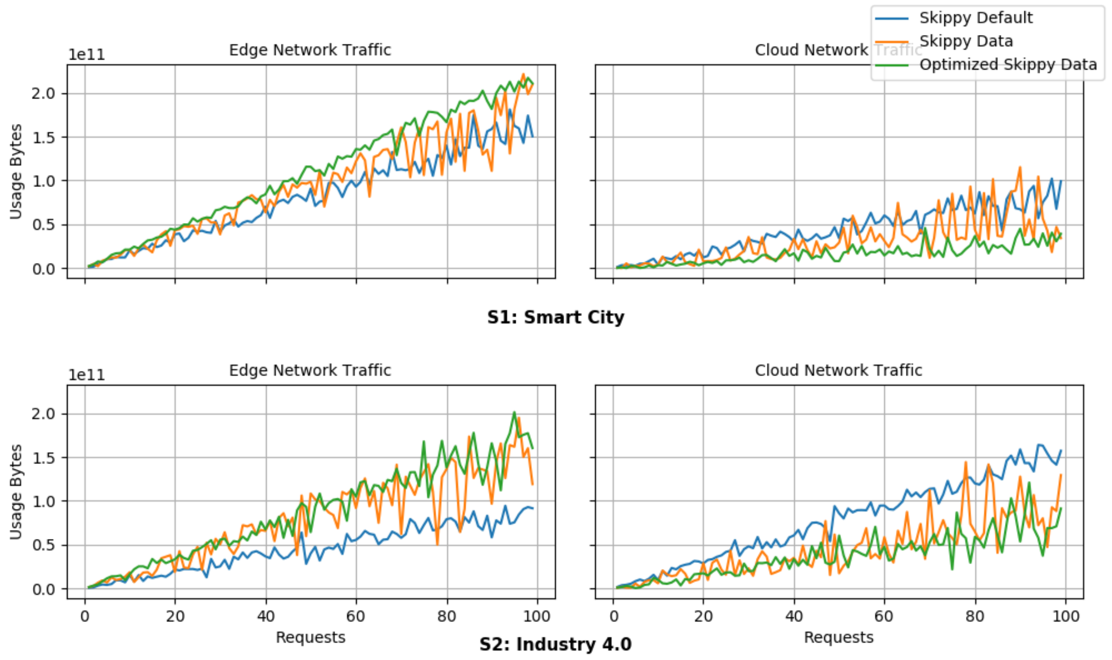


Figure 8.5: Skippy Scheduler Network Usage

Outcome

Considering the scheduling results of these experiments, we can observe that data-locality scheduling improves total FET in every scenario discussed. As “Optimized Skippy Scheduler” is a performance improvement for “Skippy Data”, there is no effect during function runtime execution. This optimization aims to provide the scheduler with file metadata to enable a faster decision-making process. The optimization results can be seen in Fig. 8.4 as both data locality scheduler approaches display similar results.

8.2.2 Scheduling Latency

As described in Section 8.1.3, the scheduling complexity is determined by the number of nodes and the runtime complexity of the priority functions. Thus, in this section, we aim to analyze the scheduling latency of our data-locality scheduler, also referred to as *Skippy Data* throughout this thesis. Like in the previous section, Skippy default is used as a quality reference. We aim to analyze how long the scheduler takes to process the number of pods in the queue. The entire scheduling mechanism is explained in Chapter 5. As in the previous section, we utilize a cluster with 1100 feasible nodes. The x shows the number of pods placed in the scheduling queue, and the axis y displays the time necessary to place the function pods. Our analysis considers high y axis values high latency and low y axis values low latency.

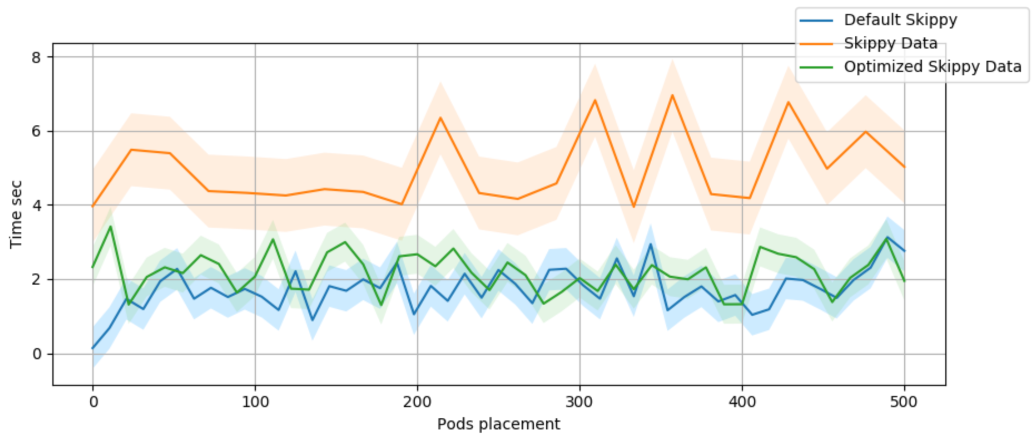


Figure 8.6: Skippy Scheduling Time

Default Skippy is represented by the blue line in Fig. 8.6. In Fig. 8.6, the default implementation shows a stable scheduling time around 2s for total scheduling and moves slowly towards 4s by 500 pods placement. Due to the absence of Data Locality in default Skippy implementation, this scheduler displays the fastest scheduling time among the schedulers analyzed.

Skippy Data shows the highest scheduling time among the schedulers in Fig. 8.6. When compared with the other schedulers in the figure, Skippy Data presents a poor performance. This is explained by *DataLocalityPriority* implementation described in Section 5.3.2. The storage index offers low latency access to storageNode and metadata files as it is explained in Chapter 6. However, the scheduler still needs to resolve a data file list by searching the storageNode, which contains each file on this list and fetching the fastest one in the bandwidth graph. As this algorithm is executed during scheduling time, this additional calculation is reflected in a nearly 100% increase in the total scheduling time in comparison with Skippy default, as can be seen in Fig. 8.6.

Optimized Skippy Data is an attempt to improve Skippy Data performance and scalability. As the calculations necessary for the data locality are unavoidable, we improve the data locality by doing these calculations beforehand in an asynchronous process. As described in Section 8.1.3, we create a full-index matrix by storing *feasibleNode,bucket,fileName] = storageNode*. Once this is pre-calculated, the full-index can be quickly accessed during scheduling time resulting in a faster total scheduling time. This improvement is represented in Fig. 8.6 by the blue line, which displays significantly better performance than Skippy data and slightly worse performance than default Skippy.

Outcome

As data locality adds additional processing during the scheduling time, it is expected that Skippy Data Scheduler presents a higher latency than the default Skippy scheduler. However, experiments show that we can minimize this difference by using a full storage index as it is described in “Optimized Skippy Data” in Section 8.1.3. Data-locality calculations during scheduling cannot be avoided, but low latency access to the file metadata can improve the scheduling process. This low latency access leads to the scheduler’s performance improvement.

8.2.3 Scalability

In this subsection, we want to see how our scheduler implementations degrade when additional feasible nodes are included in the cluster. Axis x represents the number of feasible nodes in the cluster, while y axis shows how many pods per second can be scheduled given n feasible nodes in the cluster. As our schedulers are built on top of the Kubernetes framework, we limit this set of tests to the maximum number of nodes accepted by Kubernetes of 5000 feasible nodes. Additionally, we use 500 pods placement of each function described in Section 8.1.2 [73].

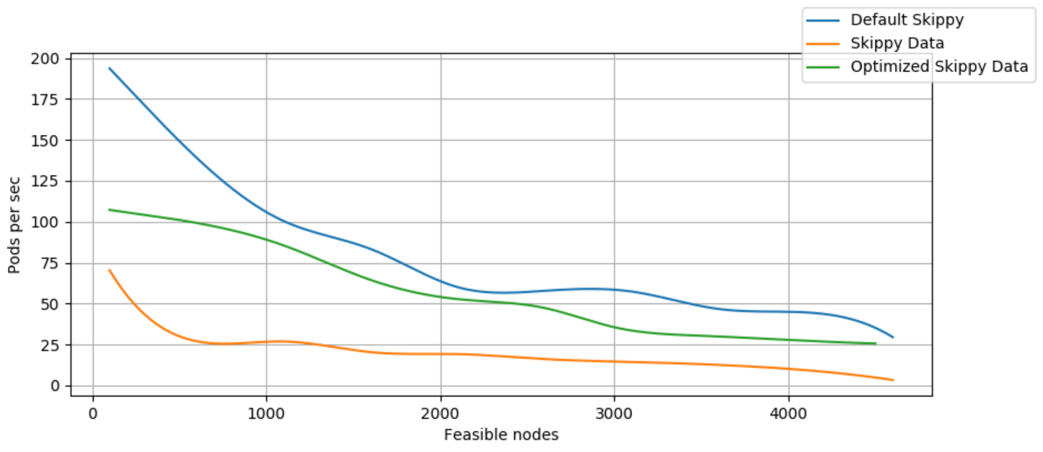


Figure 8.7: Skippy Scheduler Scalability

Default Skippy Displayed by the blue line in Fig. 8.7. Due to the absence of a data locality in Skippy default, its tests unsurprisingly reveal a higher scheduling throughput given a low number of feasible nodes in the cluster. It shows a difference of almost 125 pods per second close to $x(0)$ between Default Skippy and Skippy Data. However, by the $x(5000)$, we see this difference dropping to nearly 30 pods per second.

Skippy Data Shown in orange in Fig. 8.7. As already mentioned during the discussion in the previous Section 8.2.2, the Skippy Scheduler searches the storageNode with the best bandwidth availability for every consumed and produced data file. This searching algorithm leads to the lowest pods scheduling throughput represented in y axis even when the cluster is not highly populated. Skippy Data shows up to 65% lower throughput if compared to Skippy Scheduler. As we can observe, Skippy Data can schedule around 75 pods per second in a cluster with around $x(100)$ feasible nodes, and it converges to close to around five pods per second with x at its maximum.

Optimized Skippy Data Displayed in green line in Fig. 8.7. This optimization enables low latency access to the storage index information without any further calculations necessary, it shows higher rates than Skippy Data but still lower rates than default Skippy. While Skippy Data shows up 65% scheduling lower throughput, the optimized Skippy shows a lower throughput up to 45% compared to Skippy Scheduler.

Outcome

Overall, Default Skippy presents better scheduling throughput with low numbers of feasible nodes in the cluster. A higher scheduling throughput in Skippy Data Scheduler is due to the increased complexity of the scoring functions necessary to achieve data locality. We also notice that all measured schedulers converge between 10 and 30 pods per second towards the maximum feasible nodes present in the cluster.

8.3 Function Runtime: Skippy Data SDK

In this section, we evaluate Skippy Data SDK features and its results according to different scenarios described in Section 8.1.1 as is explained below. Skippy Data SDK implementation is responsible for resolving consumed and produced data at runtime when the serverless functions are triggered. It provides two main features: storage node prediction and ephemeral storage. Their detailed implementations are described in Section 6.4. To give an overview, we discuss the features from three points of view: Function Execution Time, Network Traffic, and Financial Costs. Additionally, we consider positive results if they decrease the total **FET**, use **network traffic** efficiently by decreasing cloud traffic while increasing edge traffic when possible and consequently decreasing **financial costs** with cloud resources. In the experiments for this section, we use a cluster of 1100 feasible nodes, 50 pods placement using Skippy Data Scheduler for the function scheduling placement. We use random storage node assignment as the baseline for comparison. Additionally, the test workflow uses data sizes as it is described in Table 8.5.

8.3.1 Function Execution Time

In Fig. 8.8 it is possible to compare the test workflow FET between the Skippy Data SDK features: storage node prediction and ephemeral disk storage. As a reference measuring point, we use random storage node assignment. That means if function f1-ml-pre is placed in nodeA, the SDK can **predict** what the best storage node to transfer the data from node A, read the data from the **ephemeral** storage if present or pick a **random** storage node to transfer data is. This set of tests aims to analyze whether ephemeral and storage node prediction can improve the function execution time compared to a random storage node assignment. We execute 100 simultaneous requests with the faas-sim simulator and analyze the results to create a range of results.

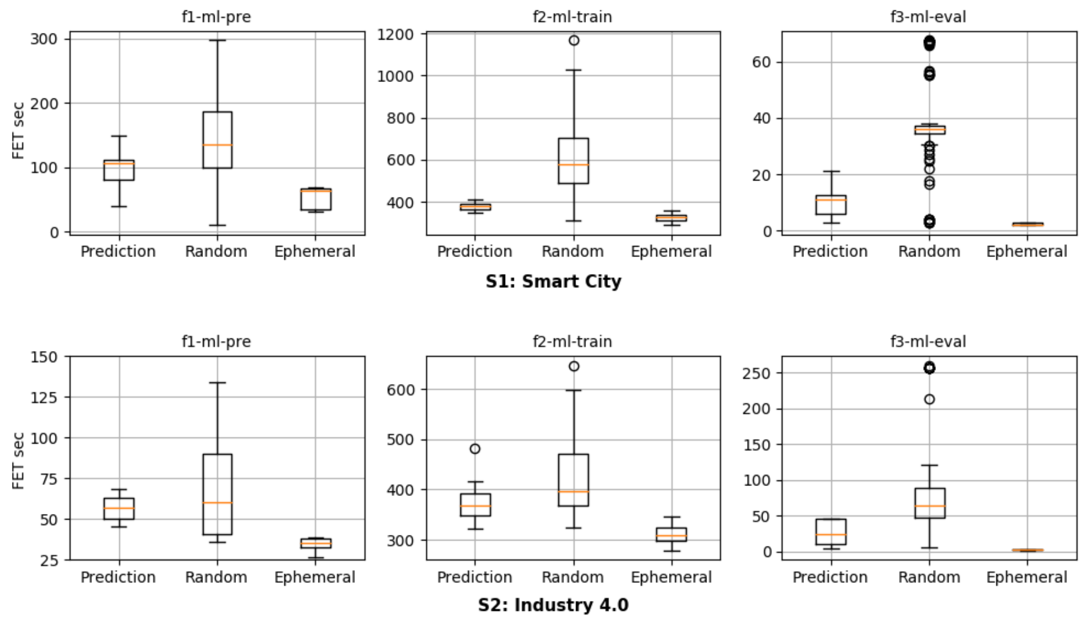


Figure 8.8: Function Execution Time

Prediction is Skippy data SDK's default behaviour. As described in Section 6.4, it assigns a storage node based on the network availability and locality. The SDK picks the storage nodes with the highest network availability placed on the edge network. StorageNodes on the cloud network are picked when there is no storage node on the edge that holds the requested metadata. As it is possible to see in Fig. 8.8 in both scenarios $S1$ and $S2$, the storage node prediction based on the bandwidth graph availability significantly improves the total function execution time. In every test case, prediction shows a decrease in execution. In some cases like *S1:f2-ml-train* nearly 30% less or even a decrease up to 50% of its FET like in *S2:f3-ml-eval*.

Random storage node refers to the first storage node found. Fig. 8.8 shows how the FET is affected when there is no network knowledge during requests execution. As displayed in the figure, in every test case for both scenarios, transferring the data from function nodeA from a random storage node assignment presents a higher average FET in comparison with other approaches in the Fig. 8.8.

Ephemeral Storage For this experiment, we enable ephemeral storage in Skippy SDK and analyze its results with other data transferring approaches. Fig. 8.8 shows how the different storage approaches perform according to the specific function. This approach is designed for chained functions where *functionB* consumes the data produced by *functionA*. Thus if both functions are placed on the same node, they can be privileged by the low latency access to the disk storage without the necessity of a network transfer. Additionally, a scenario where *f1-ml-pre* uses ephemeral storage is most likely not to happen as this is the first function to be executed. However, we assume that for *f1-ml-pre* the consumed data is already present in ephemeral storage in the disk where the function is placed. As in the ephemeral storage, there is disk reading instead of network transfer; we simulate the data transfer by using the disk speed according to the device specification displayed in Table 8.2. In both scenarios, we notice a significant decrease of FET. In *f3-ml-eval* for both scenarios, the use of ephemeral storage leads to nearly 10x faster execution or 90% of FET improvement.

Outcome

Regardless of which scenario is analyzed, in Fig. 8.8 the introduction of data-locality during runtime accelerates the total FET. The use of network availability knowledge combined with *StotrageIndex* information result in a significant improvement for the runtime execution. As the experiments show, data transfers from a predicted storage node improve up to 50% of the total FET. Additional experiments involving the ephemeral storage can speed up to 90% or 10x faster FET.

8.3.2 Network Traffic

Fig. 8.9 shows how the network traffic behaves between the storage prediction, randomly storage node assignment, and ephemeral storage. This experiment is designed to investigate whether the storage node prediction and ephemeral storage can reduce cloud network traffic, by minimizing the data exchange between cloud and edge andm consequently reduce the financial costs of network traffic. For this set of experiments, we assume the cloud transfers are over the Internet, or if they are hosted on-premises, they are provider-managed. Thus, in the Fig. 8.9 cloud network traffic also includes Internet traffic data. We used a cluster with 1100 worker nodes where each function is scaled up to 50 in both scenarios $S1$ and $S2$. Data size for the transfers is specified in Table 8.5.

In the plots shown in Fig. 8.9 the x axis shows the number of simultaneously requests of functions *f1-ml-pre*,*f2-ml-train* and *f3-ml-eval* equally executed, while axis y shows the sum amount of I/O data exchanged. The usage bytes displayed in axis y are collected from the node which hosts the function. In this setup, we want to analyze how this function node trades data with storage nodes on the edge and cloud network.

Edge Network Traffic The x axis represents the number of requests and y axis the sum of data bytes I/O in Fig. 8.9. In the Figure we see a linear increase in the bytes

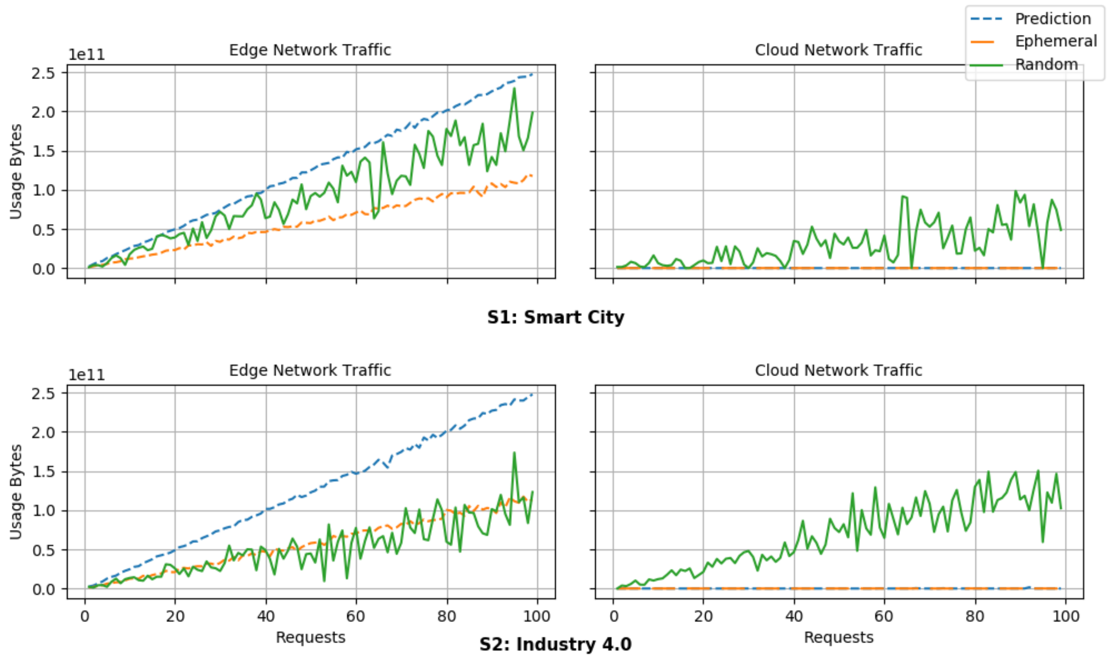


Figure 8.9: Function Runtime Network Traffic

usage. Due to the edge locality preference in the storage node prediction described in Section 6.4.2, both scenarios $S1$ and $S2$ predicted storage networks show similar results. The same happens for ephemeral storage, as the number of requests increases in axis x, ephemeral shows lower network usage as in this approach, the data is read directly from a temporary disk. The biggest difference between $S1$ and $S2$ is seen in random node assignments. This happens due to the scenarios locality distribution of 80% and 60% for $S1$ and $S2$ as can be seen in Table 8.3 and Table 8.4 respectively. As in $S1$ there is relatively higher edge network traffic even during random storage assignment, while in $S2$, the same set of tests produces less edge traffic. As there is a higher number of edge nodes on $S1$ there is a higher probability that random assignments are also placed on the edge, thus generating similar network traffic as the storage node prediction. The opposite situation happens in $S2$; since fewer devices are on edge, the network traffic is lower even in the random assignment.

Cloud Network Traffic In both scenarios in Fig. 8.9, it is possible to see a significant decrease in a cloud network traffic. The cloud network traffic is close to zero in a predicted storage node choice, which also happens for ephemeral storage. In random storage node assignment, we can see a timid increase in $S1$ as in this scenario; there are only 20% of devices on the cloud. On the other hand, a higher increase of cloud network traffic in $S2$ as 40% of its devices is placed on the cloud.

Outcome

As it is possible to see in the Fig. 8.9, Skippy Data SDK favors the edge network traffic. In all scenarios, there is a maximization of edge traffic while minimizing cloud network traffic. An efficient traffic shift from cloud to edge network reduces the financial costs and brings the processing closer to the end device, reducing latency. However, to ensure efficient network traffic, we need to ensure that edge traffic increases do not cause network overhead. To prevent network overloading, we use a network monitoring solution described in Chapter 7. Specific network experiments are further discussed in Section 8.4.

8.3.3 Financial Costs

The efficient usage of the network does not only accelerate the functions FET, but it also decreases the financial costs with cloud resources. As we want to simulate a real environment, our random storage node assignment can be transferred either from edge or cloud-like a similar real edge-cloud scenario in this set of experiments. To have an overview of this financial difference, we use the sum of transferred data necessary for one request one for each function *f1-ml-pre*, *f2-ml-train* and *f3-ml-eval*. Additionally, we assume 30 000 monthly requests are executed and estimate the financial costs to maintain this workflow using the approaches measured in network traffic according to AWS[7] prices.

According to AWS calculator[8], S3 have prices as 0.0245 USD per GB in the S3 storage and 0.09 USD per GB access. Hence, we can estimate financial costs as displayed in Fig. 8.10.

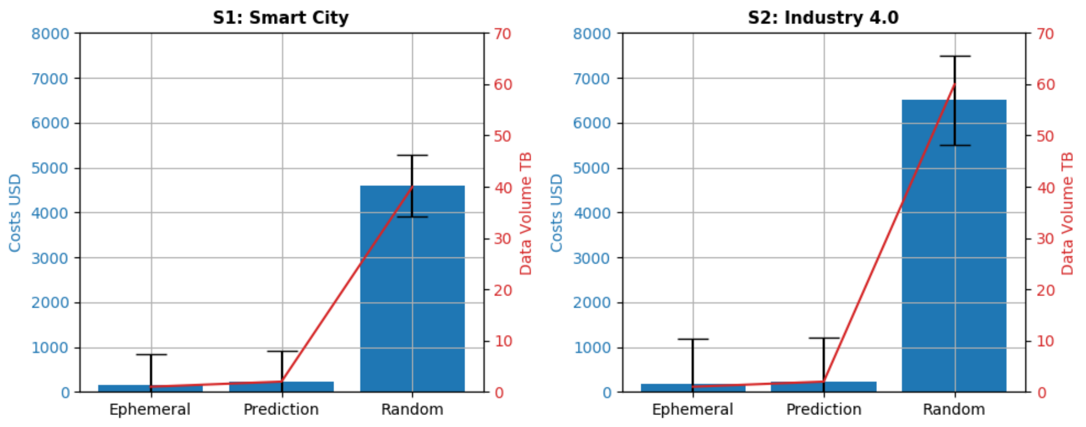


Figure 8.10: Financial Cost Estimation

The low number of 20% devices on the cloud shows a timid difference between the approaches in $S1$ while ephemeral and prediction keeps around 200 USD; the random

[7]https://aws.amazon.com/s3/pricing/
[8]https://calculator.aws/#/createCalculator/S3

assignment pushes this value to over 4000 USD. As in $S2$ there is a higher number of devices placed on the cloud; its random storage node assignment makes the financial costs to over 6500 USD while a predicted storage node shows a value of 200 USD similar to ephemeral storage approach. Overall prediction and ephemeral features display identical values in both environments. This financial similarity is the result of a decrease in cloud network traffic. In both scenarios, the ephemeral and prediction approaches maximizes the edge network traffic, leading to a reduction in cloud traffic and financial costs. Specifically, our Skippy Data SDK combined with Skippy Data Scheduler show a reduction up to 60% in $S1$ and nearly 85% in $S2$ of its financial costs with cloud providers.

8.4 Network Monitoring

In this section, we discuss the approaches presented in Chapter 7. Additionally, we show experiments for each of these approaches and an analysis of whether the feature satisfies the requirements for the project. We conclude by analyzing in which situations each methodology is best suitable.

Network Topology

For this implementation, we considered that one node uses its primary network interface for its Kubernetes communication. In other words, if a node is connected via *Ethernet* and *Wifi*, the highest route is also the one used for Kubernetes. In this case, the bandwidth graph is built reading metrics from `eth0` interface since *ethernet* has higher priority than $wlan0$ according to default routing configuration.

In an attempt to simulate a real-world system, we split our single cluster into three subgroups, as shown in Fig. 8.11. Each subgroup represents a different network that can present its settings, such as different max speeds. This composition was used as the main base to construct the test cases for the following described experiments.

8.4.1 Intrusive Bandwidth Graph

To evaluate our bandwidth graph, we ran intrusive methods which give precise results. Hence, we can compare the intrusive and non-intrusive methodologies, then establish how effective and reliable the non-intrusive approaches are.

From the many tools available in the open-source community, we picked *iperf* since it is already known for its accuracy. According to [74], the tool provides two protocols that could potentially be used for our network monitoring: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). In the default TCP behavior, *iperf* uses its Maximum Transmission Unit (MTU); in other words, the client sends as much data as possible to the server. In contrast, in the default UDP implementation, the client can set the bandwidth that can be helpful for latency measurement. Since the maximum bandwidth capacity is the metric necessary for experiments, we used the TCP protocol to get an accurate overview of the network. However, the accuracy comes at a high price;

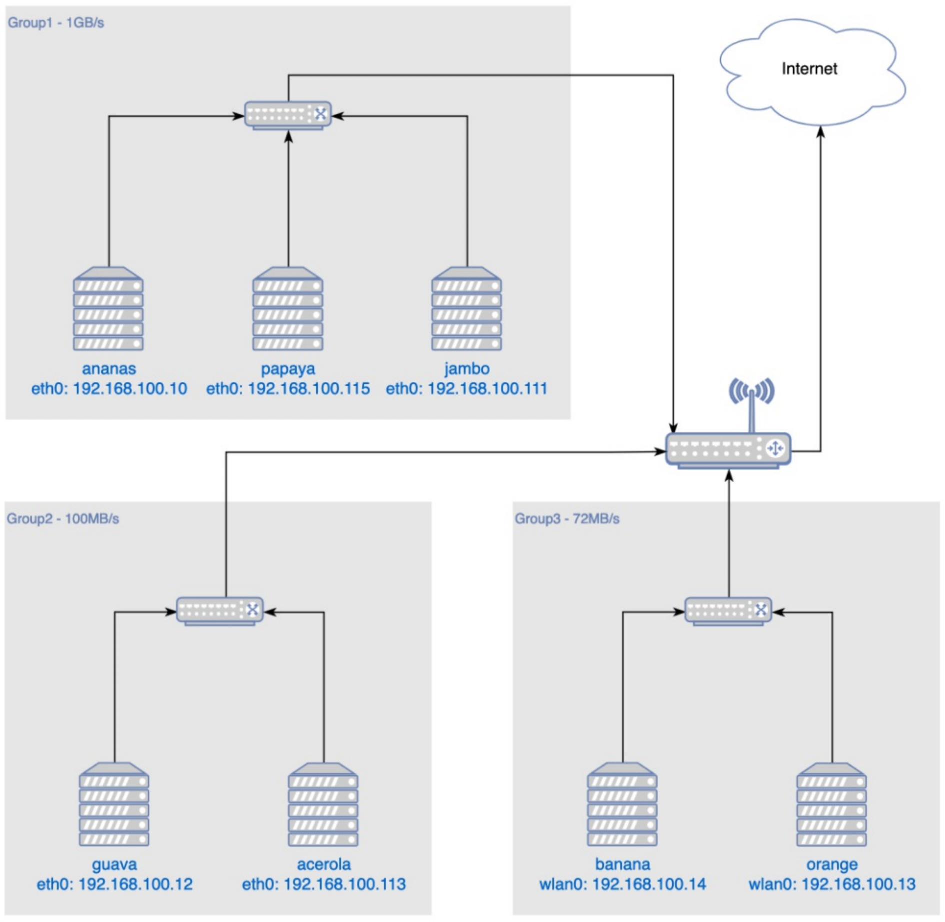


Figure 8.11: Network Topology

the tool pushes the bandwidth transfer at its maximum, which means any other transfer happening at the same time will be directly affected.

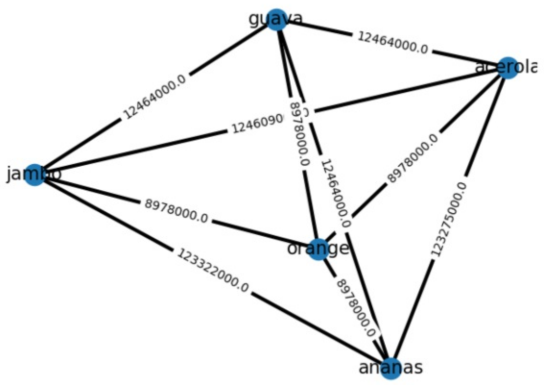
8.4.2 Estimated Bandwidth Graph

To see the Algorithm 7.1 performance, at first, we generated the graph following the network topology from the previous section. Fig. 8.12a shows available bandwidth between nodes in an idle state. We consider an idle network when there are only the necessary network connections, such as internal Linux communication or Kubernetes system communication. We generate a graph in the network where every node has a speed link between each other. In this methodology, we observe that the rates displayed are very close to the maximum node's speed link capacity, leading to false network availability since the other approach results rarely come close to the maximum link speed. The bandwidth graph can also be fetched in JSON format as displayed in Listing 8.4:

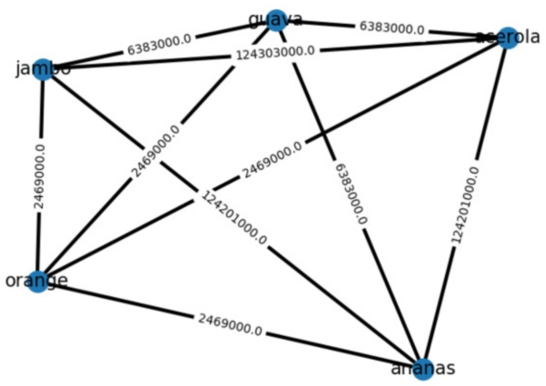
```
{
"jambo":{"guava":12464000.0,"orange":8978000.0,"acerola":123322000.0,
        "ananas":123322000.0,
"guava":{"jambo":12464000.0,"orange":8978000.0,"acerola":12464000.0,
        "ananas":12464000.0},
"orange":{"jambo":8978000.0,"guava":8978000.0,"acerola":8978000.0,
        "ananas":8978000.0},
"acerola":{"jambo":123322000.0,"guava":12464000.0,"orange":8978000.0,
        "ananas":123322000.0},
"ananas":{"jambo":123322000.0,"guava":12464000.0,"orange":8978000.0,
         "acerola":123322000.0}
}
```

Listing 8.4: Idle Network Json in Bytes/sec

Following the first experiment, we started one *scp*[9] network transfer at 6 Megabytes per second (MBps) rate between nodes orange and guava. Fig. 8.12b shows nodes *orange* and *guava* availability decreased not only between themselves but also with any other connected node. The figure also shows that connections between nodes that were not affected by the transfer remained unchanged. To have a better overview of the network, we included only five devices from the available testbed described in Table 8.1.



(a) Idle Network



(b) Busy Network

Figure 8.12: Estimated Bandwidth Graph

[9]http://www.scpwiki.com

8.4.3 Precise Bandwidth Graph

Fig. 8.13 shows what the bandwidth looks like when it is built with the latest transfer values like download and upload. In this experiment, we spawn as many pods as necessary to obtain the full bandwidth graph. We noticed that this might lead to nodes exhaustion before a full bandwidth graph is generated. Hence, we optimized the flow by temporally disabling scheduling on the nodes with bandwidth rates collected. The bandwidth can be represented in JSON as shown in Listing 8.5.

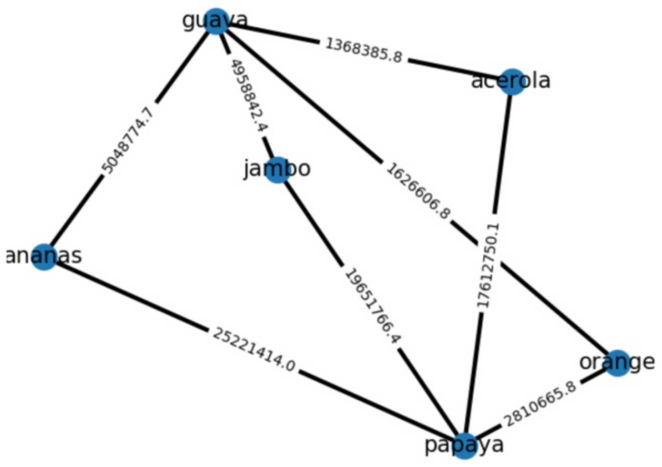


Figure 8.13: Precise Bandwidth Graph

```json
{
"jambo": {"guava": 4958842.4, "papaya": 19651766.4},
"ananas": {"papaya": 25221414.0, "guava": 5048774.7},
"orange": {"guava": 1626606.8, "papaya": 2810665.8},
"acerola": {"papaya": 17612750.1, "guava": 1368385.8}
}
```

Listing 8.5: Precise Bandwidth Graph Json in Bytes/sec

The bandwidth graph is generated with collected rates from previous data transfers such as download and upload from pods to storage nodes in this approach. In this experiment, we used six nodes from the available nodes described in the Table 8.1. This setup presents nodes *papaya* and *guava* as storage nodes and the remaining four nodes as feasible nodes able to run the pods.

It is important to emphasize that this methodology does not consider ongoing network traffic. In parallel network transfers on the nodes outside of the Kubernetes context during the collection of the rates, the precise bandwidth will have a lower value than its full capacity. The term *full capacity* refers to the maximum bandwidth possible to achieve in an idle network with only Kubernetes and the system's network traffic. As the bandwidth graph already displays the network status at one specific moment, it was unnecessary to perform load tests.

8.4.4 Outcome

To understand the difference between the results from the different methodologies, we selected four routes and directly compared values obtained from each approach as it is displayed in Table 8.7. The rates from estimated and precise bandwidth graphs are converted into Mbps. The tables show that discrepancy between results may differ significantly. We observe that the higher the maximum connection speed, the more significant differences between the three approaches. On the other hand, the results show lower contrast between the approaches for the lower connection speed groups.

Routes	Max Route Speed	Intrusive *iperf*	Estimated	Precise
ananas - papaya	1Gbps	570 Mpbs	992 Mpbs	201 Mbps
ananas - guava	100Mbps	94.1 Mpbs	99 Mpbs	40 Mbps
jambo - guava	100Mbps	93.7 Mpbs	99 Mpbs	39 Mbps
jambo - papaya	1Gbps	890 Mpbs	994 Mpbs	157 Mbps
orange - guava	72Mbps	50 Mpbs	71 Mpbs	13 Mbps
orange - papaya	72Mbps	50 Mpbs	71 Mpbs	22 Mbps
acerola - guava	100Mbps	94 Mpbs	99 Mpbs	10 Mbps
acerola - papaya	1Gbps	285 Mpbs	994 Mpbs	140 Mbps

Table 8.7: Bandwidth Monitoring Comparison

Analyzing the conducted experiments and the results displayed by approaches, we learned the following:

- **Intrusive iperf** it is a well-known tool in the open-source community and provides very accurate results. However, it adds an extra overhead in the network, which might decrease the performance in our specific case. Since our discussed scenarios rely on a stable and fast network for constant file transfers, it is not the best option in these situations.
- **Estimated Bandwidth Graph** presented reliable results when considering the ongoing transfers in the network even outside of the cluster scope. Nevertheless, it proposes too high rates compared to the other two approaches, and after experiments conducted, these rates differ from what it is possible to achieve. Thus, we recommend this solution for a light file transfer environment since it also measures network I/O outside of the cluster scope, and it is constantly updating itself without relying on previous data. This feature will always provide an up-to-date network overview regardless of past data load.

- **Precise Bandwidth Graph** shows the actual transfer speed achieved regardless of what intrusive monitoring is displayed. This approach performs well in a heavy file transfers environment since its decision relies on the latest transfer rate without invasive techniques. Thus, if transfers are long apart, the network might differ significantly from when the rate was collected. This fact might wrongly induce the scheduler.

Taking into consideration a data-intensive serverless-edge scenario, we consider **Precise Bandwidth Graph** most effective from our monitoring solutions. This approach reports the maximum achieved transfer rate, which will be used as a metric for the next scheduling decision. Furthermore, in our circumstances where there is a constant data load on the network and a short time distance, there is no significant network usage difference between the rate collected and the function execution.

CHAPTER 9

Conclusion

In this chapter, we conclude by summarizing all the contributions that led to the solution. In the following section, we discuss the research questions and the results obtained by this project, and how this thesis contributes to the research community. To conclude, we describe the main challenges faced and what could be done to continue and improve this project in future work.

9.1 Contributions

To achieve an end-to-end solution, we carefully analyzed new components and modifications on existing components. The contribution for each component is described as follows:

Skippy Data Scheduler is an extension of the existing custom Kubernetes scheduler presented in [11]. The first version of Skippy was able to identify specific cluster properties, such as CPU, RAM, GPU and resource utilization. The Skippy Data Scheduler adds data locality to Skippy default implementation. The data-locality scheduling enables Skippy to improve the FET in data-intensive scenarios, e.g., in ML workflows. Our results show that the data-locality feature in Skippy Data Scheduler leads to a 40% lower FET. Additionally, it prioritizes file transfers on edge, leading to nearly duplicated edge network traffic and consequently cloud traffic decrease. The network traffic distribution based on edge resources availability reduces the financial costs with cloud services up to 85% compared to solutions without data locality.

Skippy CLI is a deployment tool used to add Skippy labels to an OpenFaas function deployment. Skippy-cli is built on the top of Faas-cli. It reads the properties provided in the configuration file `skippy.yml` file, and it sends this information to Faas-cli in Kubernetes label format.

Skippy Data Daemon is the component in charge of collecting the file's metadata and storing it in Redis *key/value* store. The daemon executes a full scan periodically in all MinIO pods existent in the cluster. Additionally, Skippy Data Daemon builds a storage index which is later available for other components like Skippy Data Scheduler and Skippy Data SDK.

Skippy Data SDK is a python library that consumes and produces files necessary for OpenFaaS function execution. Thanks to the *Storage Node Prediction* feature described in Section 6.4.2, the Skippy Data SDK decides which storage has the largest bandwidth available to transfer the files. Additionally, our python SDK provides an ephemeral storage feature that determines whether the file is read from local temporary storage if present or if it should transfer the file from the MinIO storage.

Skippy Network is responsible for the network monitoring in the cluster. Based on metrics collected, Skippy Network builds bandwidth graph displaying current network usage. This bandwidth graph is used by Skippy Data Scheduler and Skippy Data SDK to perform decisions. Skippy Network is essential for the decision-making process during scheduling and runtime. It provides routes between nodes in the cluster that enables the Skippy Data Scheduler and Skippy Data SDK to make decisions based on the largest bandwidth availability.

Telemd is an open-source telemetry system part of *edgerun*[1] platform. Our main contribution to this project was the metric addition *netspeed* which displays the maximum speed connection of a device. This metric was added in the main project, while the additional metrics described in Section 7.2 were only used for the development and evaluation of this project.

9.2 Research Questions

Based on the evaluation of the thesis, we can answer the research questions as follows:

RQ1: Which components are necessary to enable data-locality-aware scheduling in state-of-the-art container schedulers?

To provide scheduler scoring based on data locality, we use (1) a storage index, which enables a low latency lookup search to the file metadata existent in the cluster. Additionally, we fetch (2) the bandwidth graph containing the current network usage. Once the scheduler was aware of both file metadata and the available bandwidth between cluster nodes, the scheduler scores based on the time necessary to transfer the files for each possible node in the cluster.

[1]https://edgerun.io

The combination of data-locality awareness, such as storage index and bandwidth graph, enables our scheduler to improve the total FET by 40%. Experiments in Chapter 8 corroborate that data-locality awareness improves function's execution time. Additionally, our scheduler can maximize edge networks and minimize cloud traffic efficiently. The priorities mechanism allows users to favor specific attributes such as edge and cloud locality according to the workload's requirements. In our described scenarios, the Skippy Data Scheduler increases edge traffic up to 100% compared with default Skippy implementations while cloud network traffic is decreased to close to nominal values.

RQ1.1: what is the performance impact of providing data storage information to the scheduler?

To enable decision-making based on data locality, we need first to know where in the cluster the data is stored. In our solution, we created a storage index containing file metadata information. Our Skippy Data Daemon runs constantly scanning MinIO storage nodes deployed in the cluster. The application asynchronously updates its information in a *key/value* caching store. This approach improves the decision-making process since it enables the Skippy Data Scheduler and Skippy Data SDK to a low latency access to the metadata information.

In experiments in Chapter 8, we showed that data locality could reduce the FET by 40%, increase the edge network traffic, and reduces up to 85% of the financial costs with cloud resources. However, Section 8.2.2 shows that the data locality priority increases the latency by nearly 100% compared to the Skippy default scheduler. Further, Section 8.2.3 displays a throughput degradation up to 65%. To minimize the latency and throughput degradation, we present a full storage index in "Optimized Skippy Data" in the experiments. The full storage index improves the data locality scheduling by only increasing up to 50% of latency and up to 45% of scheduling throughput compared to Skippy default. Nevertheless, we consider the scheduling latency and throughput degradation a tradeoff to achieve data locality.

RQ1.2: What is the tradeoff for providing inter-node bandwidth information to the scheduler?

To monitor the network usage and provide a bandwidth graph between the nodes in the cluster, we proposed two non-intrusive network monitoring techniques which give a bandwidth usage overview.

In the first proposed solution, referred as *EstimatedBandwidthGraph* in Section 7.3.1, we created the bandwidth graph by collecting information from the nodes in the network using *edgerun telemd*[2]. The key metrics used in this technique are the Ethernet connection speed and network I/O rates. Based on this network information, we can estimate the available connection speed on a node by decreasing its current network I/O usage from its max speed. Once this is calculated for every device present in the network, we can

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

create a bandwidth by checking the minimum value between two devices. As the use of maximum speed connection information might not give detailed insight from the device network connection at a certain moment, we considered this an “estimated” bandwidth graph.

In the second proposed solution in Section 7.3.2, we used the data transfers to collect the metrics necessary between the source node and target node where the data is stored. As our python Skippy Data SDK is responsible for downloading and uploading the data from the node that hosts the serverless function to a specific storage node, we use these data transfers to calculate the maximum transfer speed in this network route ($nodeA - StorageNodeA$). As these metrics provide the speed transfer availability in a specific moment in the network, we called it *PreciseBandwidthGraph*.

Both of our approaches we proposed non-intrusive bandwidth graphs. In the first approach is more appropriate for light data transfer scenarios since metrics are constantly collected to calculate an up-to-date bandwidth. It delivers an estimation of bandwidth usage in real-time. Meanwhile, as the second approach relies on previous transfer data to create the bandwidth graph, it displayed promising results in a workflow scenario with constant file transfers.

In our proposed framework, we avoid the usage of accurate networking monitoring tools such as *iperf*[3]. Therefore, we created two non-intrusive approaches, which give us an overview of the current network usage. In Section 8.4, accurate *iperf* shows 570 Mbps, *EstimatedBandwidthGraph* and *PreciseBandwidthGraph* show 992 Mbps and 201 Mbps respectively. As described in Chapter 7, the non-intrusive approaches are not accurate as *iperf*. Still, in data-intense scenarios, accurate tools add extra overhead to the already-overloaded edge network. As our framework does not need accurate information but only an overview of the network usage, we opt for non-intrusive methods to create a lightweight bandwidth graph. We consider the accuracy a tradeoff to provide the scheduler a bandwidth graph with an overview of the current network usage information.

RQ2: What additional runtime mechanisms are needed by a serverless system that performs data locality-aware scheduling?

As discussed in chapter 8, data-locality during scheduling time can improve the workflow characteristics such as FET and network usage. However, in an intense data transfer scenario, system attributes and workload can quickly change in a short time window. We analyze these system attributes during scheduling and runtime. To solve data locality during execution time, we proposed a Skippy Data SDK described in Section 6.4 able to consider data locality once the serverless function is executed.

Skippy Data SDK downloads and uploads necessary files for the function execution, which means it exempts developers from extra tasks to transfer the files. The Storage Node Prediction feature optimizes the files transfers. This feature uses the same data-locality

[3]https://iperf.fr/

principle used during scheduling time. Our python library can identify the shortest route between the node which hosts the executed function and potential storage node options based on storage index and bandwidth graph. Our experiments displayed that the additional runtime data locality calculations speed up the data transfer and consequently the total FET. Additionally, the framework efficiently distributes network traffic between edge and cloud. It prioritizes edge storage cloud storage which leads to less financial costs with cloud provider services.

The data-locality approach during scheduling and runtime results in up to 40% faster execution time. Additionally, it nearly duplicates the edge network traffic while decreasing the cloud network to insignificant numbers. Furthermore, the cloud traffic reduction leads to a reduction of up to 85% of financial costs with cloud services.

9.3 Challenges

For this thesis, we intended to create an environment as heterogeneous as possible. Therefore, we use different types of SBC in our testbed as described in Section 8.1.1. The different types of SBC brought configuration challenges that required executing some specific tasks for specific boards. As an example, *BananaPi*[4] supported Linux Kernel has flags that prevent Docker installation. It was necessary to build the Kernel with all the flags necessary for the framework tools installation to include BananaPi in our Kubernetes cluster.

9.4 Future work

The results obtained in this thesis show an improvement of data-intensive workflows in limited resource capacity, such as serverless edge computing. We want to highlight additional areas where data locality can be important for serverless edge computing.

9.4.1 Data Locality Awareness in the Load Balancer

Kubernetes provides an internal component *kube-proxy* responsible for forwarding requests to the specific pods in the node. Once an incoming request arrives at the Kubernetes Load Balancer (LB) it is only forwarded to the pod respecting internal Kubernetes rules. To enable a data-locality load balancing, we suggest adding data-locality awareness in *kube-proxy*. Thus, textitkube-proxy can improve the workflow as it can also take part in the decision-making process, hence optimizing the workflow. The data locality during load balancing is another improvement during function execution.

9.4.2 Function Runtime Weighted Priorities

According to [46], the scheduler is responsible for function placement and other essential jobs like prioritization, capacity management, failure recovery, and task completion.

[4]http://www.banana-pi.org/m3.html

However, in serverless environments with constant data transferred between devices with different characteristics such as data location, storage type, and storage locality, there is still a lack of attention to resolving this heterogeneity during runtime. We can improve the function execution by providing similar scheduler priority calculations based on score points. Once a serverless function is triggered, the scheduler has no longer influence, yet there are decisions to be taken like which storage node to use for transfer or pick storage from edge or cloud. Our implemented Skippy Data SDK resolves the data locality during runtime. It does find the shortest path favoring edge storage. However, a heterogeneous workflow in a heterogeneous environment can leverage the ability to tune the different features like weighted priorities according to its needs as it happens during scheduling time.

9.4.3 Scheduling on Distributed Multi-Cluster

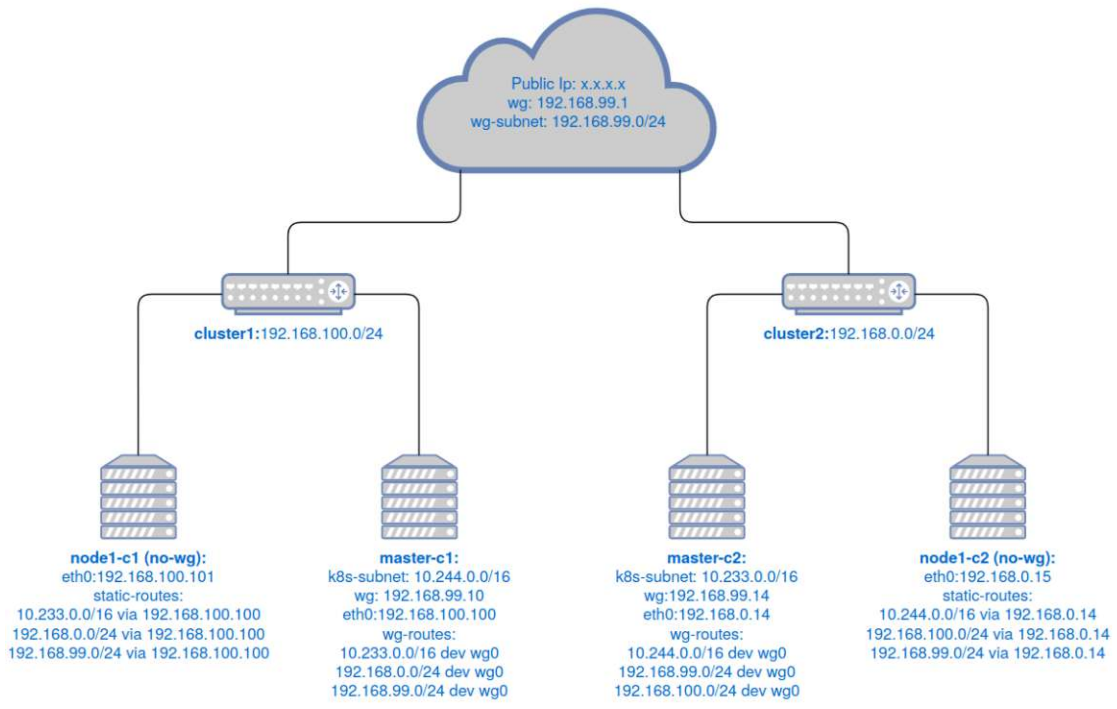


Figure 9.1: High Availability Private Clusters Setup

Fig. 9.1 shows how multiple private clusters communicate between themselves. In order to achieve high availability of the private clusters, we used *Wireguard*[5] installed on one Virtual Private Server (VPS) and on the master node of each cluster, described in the Fig. 9.1 as master-c1 and master-c2.

[5]https://www.wireguard.com

Throughout this thesis, we consider that the clusters can communicate and transfer the information as necessary. Since each cluster only hosts its information, the foreign cluster setup is completely ignored at the scheduling moment. If the clusters share the data between themselves, perhaps the scheduler can consider the alien nodes during scheduling time.

APPENDIX A

Skippy Data Scheduler Logs

In the listing A.1, one can see detailed a testbed pod scheduling logs for DataLocalityPriority and FunctionChainPriority.

```
DEBUG|There's a new pod to schedule: skippy-test-f8c49d477-nsbg5
DEBUG|Received a new pod to schedule: skippy-test-f8c49d477-nsbg5
DEBUG|Pod skippy-test-f8c49d477-nsbg5 requests 100 / 209715200. Available on
    node acerola: 4000 / 1021714432.Passed: True
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node acerola / PodFitsResourcesPred:
    Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node acerola /
    CheckNodeLabelPresencePred: Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 requests 100 / 209715200. Available on
    node ananas: 4000 / 4066054144.Passed: True
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node ananas / PodFitsResourcesPred:
    Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node ananas /
    CheckNodeLabelPresencePred: Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 requests 100 / 209715200. Available on
    node guava: 4000 / 1021718528.Passed: True
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node guava / PodFitsResourcesPred:
    Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node guava /
    CheckNodeLabelPresencePred: Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 requests 100 / 209715200. Available on
    node orange: 4000 / 1048010752.Passed: True
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node orange / PodFitsResourcesPred:
    Passed
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / Node orange /
    CheckNodeLabelPresencePred: Passed
DEBUG|ResourcePriority: Calculating score for skippy-test-f8c49d477-nsbg5 on
    acerola
DEBUG|ResourcePriority: Calculating score for skippy-test-f8c49d477-nsbg5 on
    ananas
```

```
DEBUG|ResourcePriority: Calculating score for skippy-test-f8c49d477-nsbg5 on
    guava
DEBUG|ResourcePriority: Calculating score for skippy-test-f8c49d477-nsbg5 on
    orange
16.01.2021 12:13:12.1610539992|INFO|TIME EXECUTION for (1.0, <core.priorities
    .BalancedResourcePriority object at 0xb54b8e30>) 941 microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    BalancedResourcePriority'>: [8.0, 9.0, 8.0, 8.0]
10.244.0.1 - - [16/Jan/2021 12:13:12] "GET / HTTP/1.1" 200 -
DEBUG|Docker images:{'creator': 676243, 'id': 131699505, 'image_id': None, '
    images': [{'architecture': 'arm', 'features': '', 'variant': None, '
    digest': 'sha256:
    d5447d985cb6ad5823683eec73dc08709e7d1cc83e16b01885dd619b3d3726d7', 'os':
    'linux', 'os_features': '', 'os_version': None, 'size': 49150162, 'status
    ': 'active', 'last_pulled': '2021-01-08T22:10:51.180722Z', 'last_pushed':
     '2021-01-07T12:12:32.859215Z'}], 'last_updated': '2021-01-07T12
    :12:32.859215Z', 'last_updater': 676243, 'last_updater_username': '
    keniack', 'name': 'latest', 'repository': 10591726, 'full_size':
    49150162, 'v2': True, 'tag_status': 'active', 'tag_last_pulled':
    '2021-01-08T22:10:51.180722Z', 'tag_last_pushed': '2021-01-07T12
    :12:32.859215Z'}
16.01.2021 12:13:14.1610539994|INFO|TIME EXECUTION for (1.0, <core.priorities
    .LatencyAwareImageLocalityPriority object at 0xb54b8e50>) 429279
    microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    LatencyAwareImageLocalityPriority'>: [10.0, 10.0, 10.0, 10.0]
16.01.2021 12:13:14.1610539994|INFO|TIME EXECUTION for (1.0, <core.priorities
    .LocalityTypePriority object at 0xb54b8e70>) 43 microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    LocalityTypePriority'>: [10.0, 10.0, 10.0, 10.0]
DEBUG|----- Calculating DOWNLOAD estimated time-------
DEBUG|File [test/minio.csv] present in nodes ["guava", "acerola"]
DEBUG|From [acerola] to [acerola] the same. Time = 0
16.01.2021 12:13:14.1610539994|WARNING|MinioClientException: The specified
    key does not exist.
DEBUG|File [faas/file_120M.txt] present in nodes ["guava"]
DEBUG|From [acerola] to ['guava'] [faas/file_120M.txt] calculating transfer
    size 133333337 bytes
DEBUG|From [acerola] to [guava] bandwidth is 12464000.0 bytes/s
DEBUG|From [acerola] best storage found [guava] to transfer [faas/file_120M.
    txt]. Estimated time = 10s
DEBUG|From [acerola] DOWNLOAD total estimated time for all file transfers (['
    test/minio.csv', 'faas/file_120M.txt']) is 10 s
DEBUG|----- end DOWNLOAD calculations -------
DEBUG|----- Calculating UPLOAD estimated time-------
DEBUG|From [acerola] estimated produced file size is 400000006 bytes
DEBUG|File not present. bucket present in nodes {'guava'}
DEBUG|From [acerola] to ['guava'] [myproduce1/None] calculating transfer size
     400000006 bytes
DEBUG|From [acerola] to [guava] bandwidth is 12464000.0 bytes/s
DEBUG|From [acerola] best storage found [guava] to transfer [myproduce1/].
    Estimated time = 32s
```

```
DEBUG|From [acerola] UPLOAD total estimated time for all files to (['
    myproduce1']) is 32 s
DEBUG|----- end UPLOAD calculations -------
DEBUG|From [acerola] total file transfers 42s
DEBUG|----- Calculating DOWNLOAD estimated time-------
DEBUG|File [test/minio.csv] present in nodes ["guava", "acerola"]
DEBUG|From [ananas] to ['guava', 'acerola'] [test/minio.csv] calculating
    transfer size 266666669 bytes
DEBUG|From [ananas] to [guava] bandwidth is 12464000.0 bytes/s
DEBUG|From [ananas] to [acerola] bandwidth is 124524000.0 bytes/s
DEBUG|From [ananas] best storage found [acerola] to transfer [test/minio.csv
    ]. Estimated time = 2s
DEBUG|File [faas/file_120M.txt] present in nodes ["guava"]
DEBUG|From [ananas] to ['guava'] [faas/file_120M.txt] calculating transfer
    size 133333337 bytes
DEBUG|From [ananas] to [guava] bandwidth is 12464000.0 bytes/s
DEBUG|From [ananas] best storage found [guava] to transfer [faas/file_120M.
    txt]. Estimated time = 10s
DEBUG|From [ananas] DOWNLOAD total estimated time for all file transfers (['
    test/minio.csv', 'faas/file_120M.txt']) is 12 s
DEBUG|----- end DOWNLOAD calculations -------
DEBUG|----- Calculating UPLOAD estimated time-------
DEBUG|From [ananas] estimated produced file size is 400000006 bytes
DEBUG|File not present. bucket present in nodes {'guava'}
DEBUG|From [ananas] to ['guava'] [myproduce1/None] calculating transfer size
    400000006 bytes
DEBUG|From [ananas] to [guava] bandwidth is 12464000.0 bytes/s
DEBUG|From [ananas] best storage found [guava] to transfer [myproduce1/].
    Estimated time = 32s
DEBUG|From [ananas] UPLOAD total estimated time for all files to (['
    myproduce1']) is 32 s
DEBUG|----- end UPLOAD calculations -------
DEBUG|From [ananas] total file transfers 44s
DEBUG|----- Calculating DOWNLOAD estimated time-------
DEBUG|File [test/minio.csv] present in nodes ["guava", "acerola"]
DEBUG|From [guava] to [guava] the same. Time = 0
DEBUG|---------------------------------
DEBUG|File [faas/file_120M.txt] present in nodes ["guava"]
DEBUG|From [guava] to [guava] the same. Time = 0
DEBUG|From [guava] DOWNLOAD total estimated time for all file transfers (['
    test/minio.csv', 'faas/file_120M.txt']) is 0 s
DEBUG|----- end DOWNLOAD calculations -------
DEBUG|----- Calculating UPLOAD estimated time-------
DEBUG|From [guava] estimated produced file size is 400000006 bytes
DEBUG|File not present. bucket present in nodes {'guava'}
DEBUG|From [guava] to [guava] the same. Time = 0
DEBUG|From [guava] UPLOAD total estimated time for all files to (['myproduce1
    ']) is 0 s
DEBUG|----- end UPLOAD calculations -------
DEBUG|From [guava] total file transfers 0s
DEBUG|----- Calculating DOWNLOAD estimated time-------
DEBUG|File [test/minio.csv] present in nodes ["guava", "acerola"]
```

```
DEBUG|From [orange] to ['guava', 'acerola'] [test/minio.csv] calculating
    transfer size 266666669 bytes
DEBUG|From [orange] to [guava] bandwidth is 8993000.0 bytes/s
DEBUG|From [orange] to [acerola] bandwidth is 8993000.0 bytes/s
DEBUG|From [orange] best storage found [guava] to transfer [test/minio.csv].
    Estimated time = 29s
DEBUG|File [faas/file_120M.txt] present in nodes ["guava"]
DEBUG|From [orange] to ['guava'] [faas/file_120M.txt] calculating transfer
    size 133333337 bytes
DEBUG|From [orange] to [guava] bandwidth is 8993000.0 bytes/s
DEBUG|From [orange] best storage found [guava] to transfer [faas/file_120M.
    txt]. Estimated time = 14s
DEBUG|From [orange] DOWNLOAD total estimated time for all file transfers (['
    test/minio.csv', 'faas/file_120M.txt']) is 43 s
DEBUG|----- end DOWNLOAD calculations -------
DEBUG|----- Calculating UPLOAD estimated time-------
DEBUG|From [orange] estimated produced file size is 400000006 bytes
DEBUG|File not present. bucket present in nodes {'guava'}
DEBUG|From [orange] to ['guava'] [myproduce1/None] calculating transfer size
    400000006 bytes
DEBUG|From [orange] to [guava] bandwidth is 8993000.0 bytes/s
DEBUG|From [orange] best storage found [guava] to transfer [myproduce1/].
    Estimated time = 44s
DEBUG|From [orange] UPLOAD total estimated time for all files to (['
    myproduce1']) is 44 s
DEBUG|----- end UPLOAD calculations -------
DEBUG|From [orange] total file transfers 87s
16.01.2021 12:13:14.1610539994|INFO|TIME EXECUTION for (1.0, <core.priorities
    .DataLocalityPriority object at 0xb54b8e90>) 253148 microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    DataLocalityPriority'>: [5.0, 4.0, 10.0, 0.0]
16.01.2021 12:13:14.1610539994|INFO|TIME EXECUTION for (1.0, <core.priorities
    .FunctionChainPriority object at 0xb54b8eb0>) 182 microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    FunctionChainPriority'>: [0.0, 0.0, 10.0, 0.0]
16.01.2021 12:13:14.1610539994|INFO|TIME EXECUTION for (1.0, <core.priorities
    .CapabilityPriority object at 0xb54b8ed0>) 409 microseconds
DEBUG|Pod skippy-test-f8c49d477-nsbg5 / <class 'core.priorities.
    CapabilityPriority'>: [0.0, 0.0, 0.0, 0.0]
DEBUG|Node scores: [(acerola, 33.0), (ananas, 33.0), (guava, 48.0), (orange,
    28.0)]
16.01.2021 12:13:14.1610539994|INFO|Creating namespaced binding: Pod skippy-
    test-f8c49d477-nsbg5 on Node guava
DEBUG|response body: {"kind":"Status","apiVersion":"v1","metadata":{},"status
    ":"Success","code":201}
DEBUG|Found best node. Remaining allocatable resources after scheduling:
    Capacity(CPU: 3900 Memory: 812003328)
DEBUG|Pod yielded SchedulingResult(suggested_host=guava, feasible_nodes=4,
    needed_images=['keniack/skippy-test:latest'])
```

Listing A.1: Testbed Skippy Data Logs

APPENDIX B

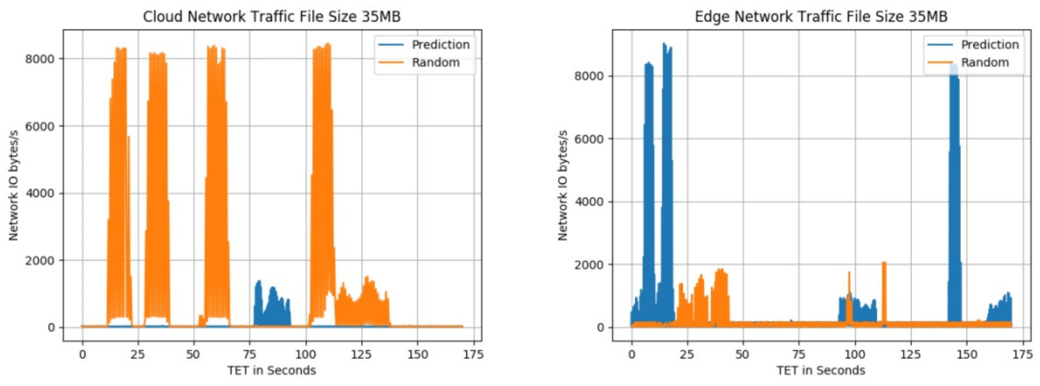
Additional Experiments

In the following pictures, we can see experiments performed in the testbed and additional simulation experiments. Figure B.1 displays a network usage overview between storage node prediction and random storage node assignment. Additionally, it considers two random scenarios *a*) most random storage nodes are placed on the cloud, and thus it generates high financial costs. In *b*) random storage nodes are placed at the edge network, generating lower financial costs. In these experiments, we imply that every edge device is not provider-managed while cloud devices are managed by services like AWS, Azure or GCP.

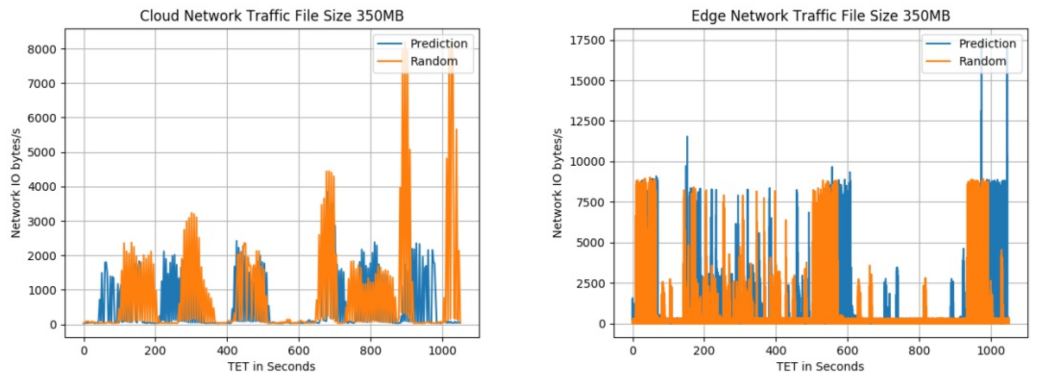
Figure B.2 shows how the use of ephemeral storage impacts computational resources like disk and ram; even when data is transferred from the network storage, there is similar resource usage on the testbed.

In figure B.3, we can see a simulation of a TET in axis y. The plot shows how execution time varies according simultaneously executed requests displayed in axis x.

Further, figure B.4 displays a 3D overview from the skippy scheduler scalability.



(a) Worst Financial Scenario



(b) Best Financial Scenario

Figure B.1: Testbed Skippy SDK Network traffic

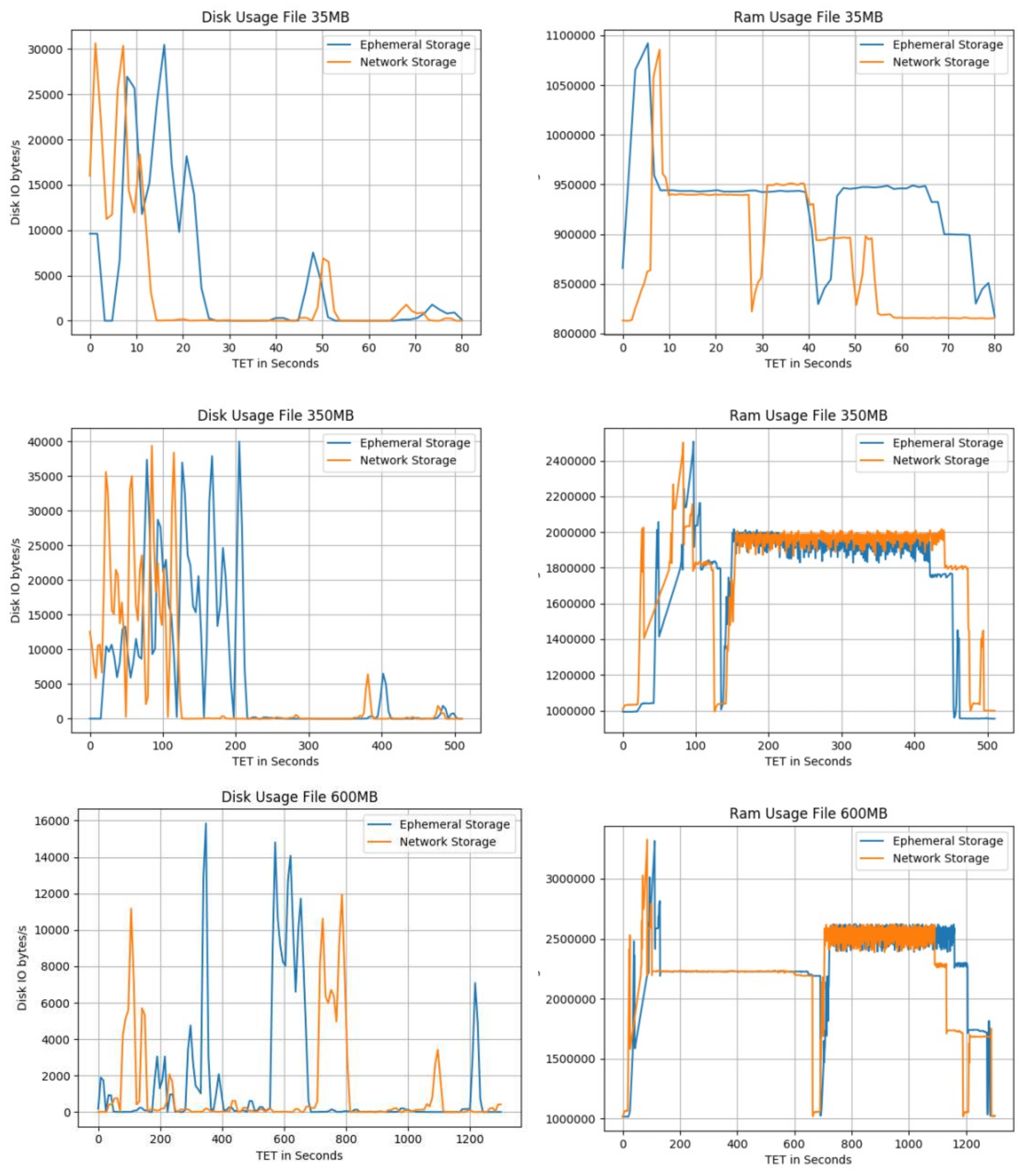


Figure B.2: Testbed Resource Usage: Function Runtime

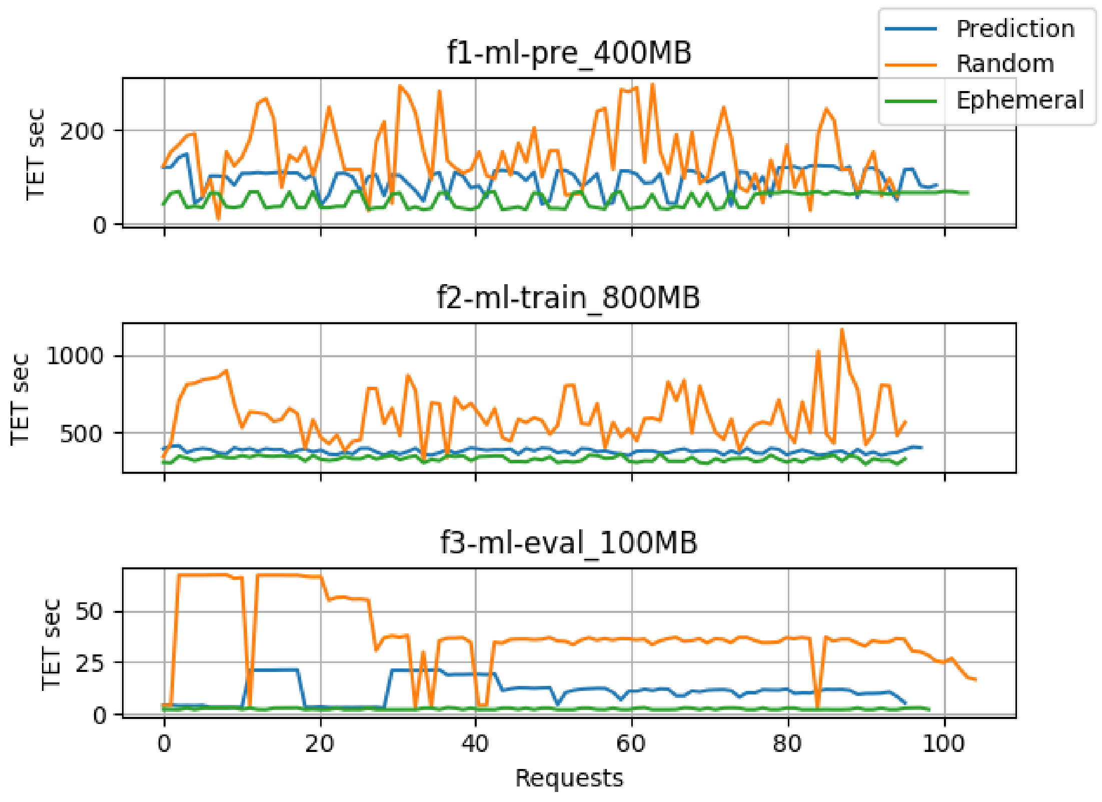
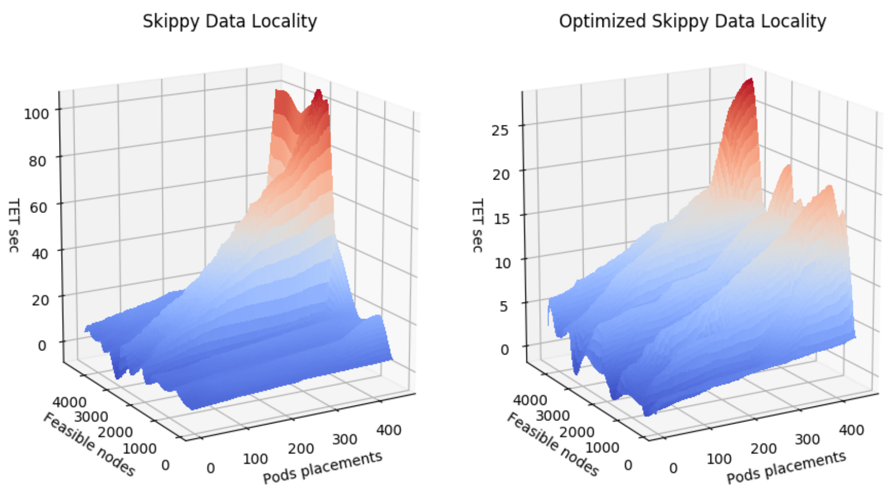


Figure B.3: Skippy Data SDK: Function Execution Time



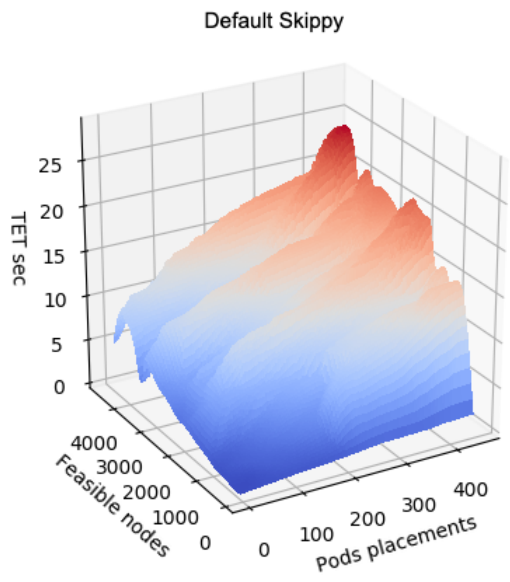


Figure B.4: Skippy Scheduler Scalability 3D

APPENDIX C

Telemd Modifications

Listing C.1 shows the changes which were necessary to *telemd*. It shows the modifications to include activeDevice, netspeed and hostIp metrics

```
commit a873ea9403ea0902e771470e2cbfabf4326686de
Author: Cynthia Marcelino <keniack@gmail.com>
Date:   Wed Jan 13 23:25:21 2021 +0100
    Add hostIp to info node
diff --git a/internal/telemd/cfg.go b/internal/telemd/cfg.go
index 5f80cca..6de9b25 100644
--- a/internal/telemd/cfg.go
+++ b/internal/telemd/cfg.go
@@ -1,10 +1,12 @@
 package telemd
 import (
+ "errors"
+ "net"
@@ -175,3 +177,26 @@ func execCommand(args string) (string, error) {
    return strings.TrimSpace(string(output)), nil
  }
 }
+
+func getInterfaceIpv4Addr(interfaceName string) (addr string, err error) {
+ var (
+   ief      *net.Interface
+   addrs    []net.Addr
+   ipv4Addr net.IP
+ )
+ if ief, err = net.InterfaceByName(interfaceName); err != nil { // get
    interface
+   return
+ }
+ if addrs, err = ief.Addrs(); err != nil { // get addresses
+   return
+ }
+ for _, addr := range addrs { // get ipv4 address
+   if ipv4Addr = addr.(*net.IPNet).IP.To4(); ipv4Addr != nil {
+     break
+   }
+ }
+ if ipv4Addr == nil {
+   return "", errors.New(fmt.Sprintf("interface %s don't have an ipv4 address\n", interfaceName))
+ }
+ return ipv4Addr.String(), nil
+}
diff --git a/internal/telemd/info.go b/internal/telemd/info.go
index 7150fbc..9b8921f 100644
--- a/internal/telemd/info.go
+++ b/internal/telemd/info.go
@@ -17,6 +17,7 @@ type NodeInfo struct {
  NetDevice string
+ HostIp    string
 }
 func (info NodeInfo) Print() {
@@ -29,6 +30,7 @@ func (info NodeInfo) Print() {
  fmt.Println("netDevice: ", info.NetDevice)
+ fmt.Println("hostIp: ", info.HostIp)
 }
 func SysInfo() NodeInfo {
@@ -73,6 +75,13 @@ func ReadSysInfo(info *NodeInfo) {
  } else {
-   log.Println("error reading network device info", err)
+   log.Println("error reading active device info", err)
  }
+
+ if hostIp, err := getInterfaceIpv4Addr(info.NetDevice); err == nil {
+   info.HostIp = hostIp
+ } else {
+   log.Println("error reading hostIp info", err)
+ }
+
 }
diff --git a/internal/telemd/redis.go b/internal/telemd/redis.go
index be164b1..bfe1d45 100644
--- a/internal/telemd/redis.go
+++ b/internal/telemd/redis.go
@@ -95,6 +95,7 @@ func WriteNodeInfo(client *redis.Client, nodeName string, info NodeInfo) error {
  multi.HSet(key, "net", strings.Join(info.Net, " "))
  multi.HSet(key, "netspeed", info.NetSpeed)
  multi.HSet(key, "netdevice", info.NetDevice)
+ multi.HSet(key, "hostIp", info.HostIp)

  _, err := multi.Exec()
  return err
commit 2de1ba9276189a6379ac8a8c9e1abb202a6dc5e9
Author: Cynthia Marcelino <keniack@gmail.com>
```

```
Date:   Fri Nov 13 00:27:55 2020 +0100
    #10 find net device
diff --git a/internal/telemd/cfg.go b/internal/telemd/cfg.go
index c1a397f..0ed54d6 100644
--- a/internal/telemd/cfg.go
+++ b/internal/telemd/cfg.go
@@ -135,31 +135,36 @@ func blockDevices() []string {
 }
 func netSpeed() string {
- devices := networkDevices()
+ activeNetDevice := findActiveNetDevice()
+ wirelessPath := "/sys/class/net/" + activeNetDevice + "/wireless"
+ if fileDirExists(wirelessPath) {
+   return findWifiSpeed(activeNetDevice)
+ }else {
+   path := "/sys/class/net/" + activeNetDevice + "/speed"
+   speed, err := readFirstLine(path)
+   check(err)
+   return speed;
  }
  return ""
 }
+func findActiveNetDevice() string {
+ args := "route | awk 'NR==3{print $8}'"
+ return execCommand(args)
+}
-func exec_command(device string) string {
+func findWifiSpeed(device string) string {
  args := "iw dev "+device+" link | awk -F '[ ]' '/tx bitrate:/{print $3}'"
+ speed := execCommand(args)
+ value, _ := strconv.ParseFloat(speed,32)
+ return fmt.Sprint(int(value))
+}
+func execCommand(args string) string {
  cmd := exec.Command("sh","-c", args)
  if output,err := cmd.Output(); err!= nil {
-   log.Printf( "Error fetching wifi bitrate: %s",err)
+   log.Printf( "Error executing command: %s",err)
  }else{
    log.Printf( "wifi bitrate: %s",output)
-   str_output := strings.TrimSpace(string(output))
-   value, _ := strconv.ParseFloat(str_output,32)
-   return fmt.Sprint(int(value))
+   return strings.TrimSpace(string(output))
  }
  return ""
 }
diff --git a/internal/telemd/fileio.go b/internal/telemd/fileio.go
index 6074a0a..d35a4d3 100644
--- a/internal/telemd/fileio.go
+++ b/internal/telemd/fileio.go
@@ -68,3 +68,11 @@ func readLineAndParseInt(path string) (int64, error) {
  }
  return strconv.ParseInt(line, 10, 64)
 }
+
+func fileDirExists(filename string) bool {
+ _, err := os.Stat(filename)
+ if os.IsNotExist(err) {
+   return false
+ }
+ return true
+}
\ No newline at end of file
commit 3ebb7f3f38bb5369aeccbf10441e569e54ce09aa
Author: Cynthia Marcelino <keniack@gmail.com>
Date:   Thu Nov 12 23:09:00 2020 +0100
    #10 add wifi net speed
diff --git a/internal/telemd/cfg.go b/internal/telemd/cfg.go
index ad92e7a..c1a397f 100644
--- a/internal/telemd/cfg.go
+++ b/internal/telemd/cfg.go
@@ -1,10 +1,13 @@
 package telemd
 import (
+  "fmt"
   "github.com/edgerun/telemd/internal/env"
   "io/ioutil"
   "log"
   "os"
+  "os/exec"
+  "strconv"
   "strings"
   "time"
 )
@@ -131,7 +134,7 @@ func blockDevices() []string {
  })
 }

-func ethSpeed() string {
+func netSpeed() string {
  devices := networkDevices()
  for _, dev:= range devices {
    if strings.HasPrefix(dev, "e"){
@@ -139,7 +142,24 @@ func ethSpeed() string {
      speed, err := readFirstLine(path)
      check(err)
      return speed;
+    } else if strings.HasPrefix(dev, "w") {
+      speed := exec_command(dev)
+      return speed;
    }
  }
  return ""
 }
+
+func exec_command(device string) string {
+ args := "iw dev "+device+" link | awk -F '[ ]' '/tx bitrate:/{print $3}'"
+ cmd := exec.Command("sh","-c", args)
+ if output,err := cmd.Output(); err!= nil {
+   log.Printf( "Error fetching wifi bitrate: %s",err)
+ }else{
+   log.Printf( "wifi bitrate: %s",output)
+   str_output := strings.TrimSpace(string(output))
+   value, _ := strconv.ParseFloat(str_output,32)
+   return fmt.Sprint(int(value))
+ }
+ return ""
+}
diff --git a/internal/telemd/info.go b/internal/telemd/info.go
index 8305234..f9aac60 100644
--- a/internal/telemd/info.go
+++ b/internal/telemd/info.go
@@ -14,7 +14,7 @@ type NodeInfo struct {
  Hostname string
- EthSpeed string
+ NetSpeed string
 }

 func (info NodeInfo) Print() {
@@ -25,7 +25,7 @@ func (info NodeInfo) Print() {
  fmt.Println("Hostname: ", info.Hostname)
- fmt.Println("EthSpeed: ", info.EthSpeed)
+ fmt.Println("netSpeed: ", info.NetSpeed)
 }

 func SysInfo() NodeInfo {
@@ -63,7 +63,7 @@ func ReadSysInfo(info *NodeInfo) error {
    return err
  }
  info.Hostname = hostname
- info.EthSpeed = ethSpeed()
+ info.NetSpeed = netSpeed()

  return nil
 }
diff --git a/internal/telemd/redis.go b/internal/telemd/redis.go
index 13b7cdf..f9671a7 100644
--- a/internal/telemd/redis.go
+++ b/internal/telemd/redis.go
@@ -93,7 +93,7 @@ func WriteNodeInfo(client *redis.Client, nodeName string,
    info NodeInfo) error {
  multi.HSet(key, "net", strings.Join(info.Net, " "))
- multi.HSet(key, "ethspeed", info.EthSpeed)
+ multi.HSet(key, "netspeed", info.NetSpeed)
  _, err := multi.Exec()
  return err
```

Listing C.1: telemd patch to add new metrics

APPENDIX D

Openfaas Modifications

Listing D.1 shows *openfaas* changes necessary to . It shows the modifications done in *openfaas*

```
diff --git a/pkg/handlers/deploy.go b/pkg/handlers/deploy.go
index 54cfd9ca..6f5c8d85 100644
--- a/pkg/handlers/deploy.go
+++ b/pkg/handlers/deploy.go
@@ -116,11 +116,11 @@ func MakeDeployHandler(functionNamespace string,
    factory k8s.FunctionFactory) ht
     }
     log.Printf("Service created: %s.%s\n", request.Service, namespace)
-
     w.WriteHeader(http.StatusAccepted)
   }
 }
+
 func makeDeploymentSpec(request types.FunctionDeployment, existingSecrets
     map[string]*apiv1.Secret, factory k8s.FunctionFactory) (*appsv1.
     Deployment, error) {
  envVars := buildEnvVars(&request)
@@ -173,7 +173,6 @@ func makeDeploymentSpec(request types.FunctionDeployment,
     existingSecrets map[st
  }
  enableServiceLinks := false
- allowPrivilegeEscalation := false
  deploymentSpec := &appsv1.Deployment{
    ObjectMeta: metav1.ObjectMeta{
@@ -212,8 +211,24 @@ func makeDeploymentSpec(request types.FunctionDeployment
    , existingSecrets map[st
        },
        Spec: apiv1.PodSpec{
          NodeSelector: nodeSelector,
-         Containers: []apiv1.Container{
+         SchedulerName: "skippy-scheduler",
+         Volumes: []corev1.Volume{
            {
+             Name: "openfaas-local-storage",
+             VolumeSource : corev1.VolumeSource{
+               PersistentVolumeClaim: &corev1.
    PersistentVolumeClaimVolumeSource{
+                 ClaimName: "openfaas-skippy-pvc",
+               },
+             },
+           },
+         },
+         Containers: []apiv1.Container{
+           { VolumeMounts: []corev1.VolumeMount{
+               {
+                 Name: "openfaas-local-storage",
+                 MountPath:"/openfaas-local-storage",
+               },
+             },
              Name:  request.Service,
              Image: request.Image,
              Ports: []apiv1.ContainerPort{
@@ -229,8 +244,7 @@ func makeDeploymentSpec(request types.FunctionDeployment,
     existingSecrets map[st
              LivenessProbe:   probes.Liveness,
              ReadinessProbe:  probes.Readiness,
              SecurityContext: &corev1.SecurityContext{
-               ReadOnlyRootFilesystem:   &request.ReadOnlyRootFilesystem,
-               AllowPrivilegeEscalation: &allowPrivilegeEscalation,
+               ReadOnlyRootFilesystem: &request.ReadOnlyRootFilesystem,
              },
            },
          },
@@ -312,6 +326,15 @@ func buildEnvVars(request *types.FunctionDeployment) []
    corev1.EnvVar {
    })
  }
+ envVars = append(envVars, corev1.EnvVar{
+   Name:  "node",
+   ValueFrom: &corev1.EnvVarSource{
+     FieldRef: &corev1.ObjectFieldSelector{
+       FieldPath: "spec.nodeName",
+     },
+   },
+ })
  for k, v := range request.EnvVars {
    envVars = append(envVars, corev1.EnvVar{
      Name:  k,
diff --git a/yaml_armhf/skippy-storage.yml b/yaml_armhf/skippy-storage.yml
new file mode 100644
index 00000000..290a0ab8
--- /dev/null
+++ b/yaml_armhf/skippy-storage.yml
@@ -0,0 +1,28 @@
```

```
+kind: PersistentVolume
+apiVersion: v1
+metadata:
+  name: openfaas-skippy-pv
+  labels:
+    openfaas.storage: local
+spec:
+  capacity:
+    storage: 10Gi
+  accessModes:
+    - ReadWriteMany
+  hostPath:
+    path: "/openfaas-local-storage"
+---
+apiVersion: v1
+kind: PersistentVolumeClaim
+metadata:
+  name: openfaas-skippy-pvc
+  namespace: openfaas-fn
+  labels:
+    openfaas.storage: local
+spec:
+  accessModes:
+    - ReadWriteMany
+  resources:
+    requests:
+      storage: 10Gi
+
```

Listing D.1: openfaas patch to add new metrics

Acronyms

API	Application Programming Interface. 37, 49, 53, 124
AWS	Amazon Web Services. 2, 9, 11, 22, 38, 49, 78, 88, 107, 124
CLI	Command Line Interface. 31–33, 61, 95, 124
CO	Container Orchestration. 25, 124
CPU	Central Processing Unit. 2, 4, 9, 16, 20, 36, 37, 60, 95, 124
DHT	Data Hashing Tables. 17, 19, 21, 124
FaaS	Function-as-a-Service. 1, 9, 10, 27, 30, 33, 44, 124
FET	Function Execution Time. 75, 79–81, 84–86, 88, 95, 97–99, 124
GB	Gigabytes. 27, 36, 88, 124
Gbps	Gigabits per second. 80, 124
GCP	Google Cloud Platform. 78, 107, 124
GPU	Graphics Processing Unit. 4, 20, 35, 38, 39, 95, 124
GUI	Graphic User Interface. 33, 124
HDFS	Hadoop Distributed File System. 21, 124
HTTP	Hypertext Transfer Protocol. 14, 28, 33, 124
I/O	Input/Output. 27, 49, 53, 61, 63, 86, 93, 97, 124
IoT	Internet of Things. 1, 9–12, 70, 72, 124
IP	Internet Protocol. 20, 124
K8s	Kubernetes. 2, 14, 124

LB	Load Balancer. 99, 124
MB	Megabytes. 36, 75, 124
MBps	Megabytes per second. 91, 124
Mbps	Megabits per second. 80, 93, 98, 124
MCDM	multi-criteria decision making. 124
ML	Machine Learning. 2, 5, 9, 12, 20, 45, 53, 72, 73, 95, 124
MTU	Maximum Transmission Unit. 89, 124
PV	Persistent Volume. 45, 124
PVC	Persistent Volume Claim. 45, 124
RAM	Random-access Memory. 2, 4, 9, 20, 27, 36, 37, 60, 95, 124
S3	Simple Storage Service. 2, 49, 53, 78, 88, 124
SBC	Single Board Computer. 49, 70, 72, 99, 124
SDK	Software Development Kit. 5, 6, 25, 26, 28–30, 32, 42, 45, 47, 49, 53–56, 59, 65, 69, 75–79, 84–86, 88, 89, 96–98, 100, 124
TCP	Transmission Control Protocol. 89, 124
TET	Task Execution Time. 53, 107, 124
UDP	User Datagram Protocol. 89, 124
urn	Uniform Resource Name. 31, 124
VM	Virtual Machine. 10, 72, 124
VPN	Virtual Private Network. 61, 124
VPS	Virtual Private Server. 100, 124

Bibliography

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

[2] Bin Cheng, Jonathan Fürst, Gürkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive IoT services. *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35, 2019.

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

[4] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. Understanding open source serverless platforms. *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19*, 2019.

[5] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[6] 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)*, Renton, WA, July 2019. USENIX Association.

[7] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, and S. Schulte. A framework for optimization, service placement, and runtime operation in the fog. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 164–173, Dec 2018.

[8] V. Farhadi, F. Mehmeti, T. He, T. L. Porta, H. Khamfroush, S. Wang, and K. S. Chan. Service placement and request scheduling for data-intensive applications in edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1279–1287, 2019.

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

[10] W. Yoo and A. Sim. Network bandwidth utilization forecast model on high bandwidth networks. In *2015 International Conference on Computing, Networking and Communications (ICNC)*, pages 494–498, 2015.

[11] Alexander Rashed. Optimized container scheduling for serverless edge computing. Master's thesis, Technische Universität Wien, 2019.

[12] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49:78–81, 05 2016.

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

[14] Mohammad Sadegh Aslanpour, Adel 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. 02 2021.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, 04 2010.

[16] Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32:615–632, 10 2020.

[17] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312, 2018.

[18] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[19] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019.

[20] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service(faas) in industry and research. *CoRR*, abs/1708.08028, 2017.

[21] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2658–2659, 2017.

[22] M. McGrath and P. Brenner. Serverless computing: Design, implementation, and performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017.

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

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

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

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

[27] D. Xu, Tong Li, Y. Li, Xiang Su, Sasu Tarkoma, Tao Jiang, J. Crowcroft, and Pan Hui. Edge intelligence: Architectures, challenges, and applications. *arXiv: Networking and Internet Architecture*, 2020.

[28] Thomas Rausch, Waldemar Hummer, and Vinod Muthusamy. Pipesim: Trace-driven simulation of large-scale AI operations platforms. *CoRR*, abs/2006.12587, 2020.

[29] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A review of machine learning and iot in smart transportation. *Future Internet*, 11(4), 2019.

[30] Thomas Rausch, Waldemar Hummer, Christian Stippel, Silvio Vasiljevic, Schahram Dustdar, Carmine Elvezio, and Katharina Krösl. Towards a platform for smart city-scale cognitive assistance applications. In n.n., editor, *IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW) 2021*, 2021.

[31] C. Catlett, P. Beckman, Rajesh Sankaran, and Kate Kusiak Galvin. Array of things: a scientific research instrument in the public way: platform design and early lessons learned. *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, 2017.

[32] Long Hu, Yiming Miao, Gaoxiang Wu, Mohammad Hassan, and Iztok Humar. irobot-factory: An intelligent robot factory based on cognitive manufacturing and edge computing. *Future Generation Computer Systems*, 90, 08 2018.

[33] Sayed Chhattan Shah. Mobile edge cloud: Opportunities and challenges. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1572–1577, 2017.

[34] Haojun Huang, Wang Miao, Geyong Min, and Chunbo Luo. *Mobile Edge Computing for the 5G Internet of Things*, pages 143–161. 05 2019.

[35] Bo Liu, Pengfei Li, Weiwei Lin, Na Shu, Yin Li, and Victor Chang. A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, 22:1–12, 12 2018.

[36] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. Sharing and caring of data at the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[37] Balasundaram Vengadeswaran. Core – an optimal data placement strategy in hadoop for data intentitive applications based on cohesion relation. *Computer Systems Science and Engineering*, 34(1):47–60, 2019.

[38] Apostolos Papageorgiou, Bin Cheng, and Ernö Kovacs. Real-time data reduction at the network edge of internet-of-things systems. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 284–291, 2015.

[39] Junjie Xie, Chen Qian, Deke Guo, Minmei Wang, Shouqian Shi, and Honghui Chen. Efficient indexing mechanism for unstructured data sharing systems in edge computing. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 820–828, 2019.

[40] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[41] B Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *27th International Conference on Very Large Data Bases (VLDB 2001)*, September 2001. This is a shortened version; see the extended version for full details.

[42] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, page 149–160, New York, NY, USA, 2001. Association for Computing Machinery.

[43] T. He, H. Khamfroush, S. Wang, T. La Porta, and S. Stein. It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 365–375, July 2018.

[44] S. Wang, M. Zafer, and K. K. Leung. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[45] H. Tan, Z. Han, X. Li, and F. C. M. Lau. Online job dispatching and scheduling in edge-clouds. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[46] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[47] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor C. M. Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. *CoRR*, abs/2101.06582, 2021.

[48] Michael Ogbuachi, Anna Reale, Peter Suskovics, and Benedek Kovacs. Context-aware kubernetes scheduler for edge-native applications on 5g. *Journal of Communications Software and Systems*, 16:85, 04 2020.

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

[50] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. Towards qos-aware fog service placement. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 89–96, 2017.

[51] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu. Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. *IEEE Transactions on Computers*, 65(12):3702–3712, 2016.

[52] Muhammad Khurram Bhatti, Isil Oz, Sarah Amin, Maria Mushtaq, Umer Farooq, Konstantin Popov, and Mats Brorsson. Locality-aware task scheduling for homogeneous parallel computing systems. *Computing*, 100:557–595, 2017.

[53] Atakan Aral, Ivona Brandic, Rafael Brundo Uriarte, Rocco De Nicola, and Vincenzo Scoca. Addressing application latency requirements through edge scheduling. *Journal of Grid Computing*, 11 2019.

[54] Chia-Wei Lee, Kuang-Yu Hsieh, Sun-Yuan Hsieh, and Hung-Chang Hsiao. A dynamic data placement strategy for hadoop in heterogeneous environments. *Big Data Research*, 1:14–22, 2014. Special Issue on Scalable Computing for Big Data.

[55] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[56] Minho Bae, Sangho Yeo, Gyudong Park, and Sangyoon Oh. Novel data-placement scheme for improving the data locality of hadoop in heterogeneous environments: Na. *Concurrency and Computation: Practice and Experience*, page e5752, 03 2020.

[57] Luiz Angelo Steffenel and Manuele Kirsch Pinheiro. Improving data locality in p2p-based fog computing platforms. *Procedia Computer Science*, 141:72 – 79, 2018.

[58] Roberto Grossi and Luca Versari. Round-Hashing for Data Storage: Distributed Servers and External-Memory Tables. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 43:1–43:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[59] M. Dehghan, A. Seetharam, B. Jiang, T. He, T. Salonidis, J. Kurose, D. Towsley, and R. Sitaraman. On the complexity of optimal routing and content caching in heterogeneous networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 936–944, April 2015.

[60] S. Borst, V. Gupta, and A. Walid. Distributed caching algorithms for content distribution networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[61] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, July 2018. USENIX Association.

[62] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[63] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[64] Stefano Forti, Marco Gaglianese, and Antonio Brogi. Lightweight self-organising distributed monitoring of fog infrastructures. *Future Generation Computer Systems*, 114:605–618, 2021.

[65] István Pelle, Francesco Paolucci, Balázs Sonkoly, and Filippo Cugini. Latency-sensitive edge/cloud serverless dynamic deployment over telemetry-based packet-optical network. *IEEE Journal on Selected Areas in Communications*, PP:1–1, 03 2021.

[66] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, 2017.

[67] Persistent volumes | kubernetes. `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`. Accessed: 2021-04-04.

[68] Abdulkadir Karaagac, Eli De Poorter, and Jeroen Hoebeke. In-band network telemetry in industrial wireless sensor networks. *IEEE Transactions on Network and Service Management*, PP:1–1, 10 2019.

[69] The linux kernel. `https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-net-statistics`. Accessed: 2021-02-12.

[70] A. Hassidim, D. Raz, M. Segalov, and A. Shaqed. Network utilization: The flow view. In *2013 Proceedings IEEE INFOCOM*, pages 1429–1437, 2013.

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

[72] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, 10 2009.

[73] Kubernetes. Considerations for large clusters. `https://kubernetes.io/docs/setup/best-practices/cluster-large/`, May 2021.

[74] Iperf - the ultimate speed test tool for tcp, udp and sctp. `https://iperf.fr/iperf-doc.php`. Accessed: 2021-03-09.