POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

From the Edge to the Cloud, A New Approach To Hybrid Computing

Master of Science in
Computer Science and Engineering

Authors:
Leonardo Barilani
Giampietro Fabrizio Bonaccorsi

Students ID: 945167, 945198
Advisor: Alessandro Margara
Co-advisors: Gianpaolo Cugola
Academic Year: 2022-23

Abstract

This thesis presents a comprehensive study on the design, implementation, and evaluation of a stateful Function-as-a-Service (FaaS) framework tailored for edge computing environments. Edge computing has emerged as a promising paradigm to address the latency and bandwidth constraints of centralized cloud computing by bringing computation closer to the end users. However, existing FaaS frameworks primarily focus on stateless functions, limiting their applicability for complex edge computing applications that require state management.

To address this limitation, we propose a framework compatible with OpenFaaS, a popular open-source FaaS platform, that incorporates stateful capabilities and automatic offloading of function execution to higher nodes in the infrastructure hierarchy. We draw inspiration from Cloudflare's Durable Objects[43] to enable efficient session management, data consistency, and seamless scalability on the edge.

The thesis begins with a comprehensive review of existing FaaS frameworks and platforms, critically analyzing their methodologies, results, and weaknesses. Real-world applications and case studies are examined to assess the impact of stateful functions on efficiency, latency, and user experience. Based on this analysis, we justify the need for a stateful FaaS framework for edge computing.

The design and implementation of our modified OpenFaaS framework are presented, highlighting the integration of stateful capabilities and automatic offloading. We discuss the architectural enhancements, leveraging containerization with Docker and orchestration with Kubernetes to achieve portability and scalability. We also describe the incorporation of session management and data migration mechanisms to ensure seamless transition and optimal resource utilization.

To evaluate the performance and efficiency of our framework, extensive experiments and benchmarking are conducted and through the obtained results we demonstrate the advantages of our stateful FaaS solution in real-world edge computing scenarios.

Keywords: Faas, Serverless, Edge, Kubernetes, Cloud, Stateful, Offloading

Abstract in lingua italiana

Questa tesi presenta uno studio completo sulla progettazione, l'implementazione e la valutazione di un framework Function-as-a-Service (FaaS) stateful adattato agli ambienti di edge computing. L'edge computing ha dimostrato di essere un paradigma promettente per affrontare i vincoli di latenza e consumo di banda del cloud computing centralizzato, avvicinando la computazione agli utenti finali. Tuttavia, i framework FaaS esistenti si concentrano principalmente su funzioni stateless, limitando la loro applicabilità a casistiche complesse di edge computing che richiedono la gestione dello stato.

Per ovviare a questa limitazione, proponiamo un framework compatibile con OpenFaaS, una popolare piattaforma FaaS open-source, che incorpora la gestione dello stato sottoforma di sessioni e l'offloading automatico dell'esecuzione ai nodi superiori della gerarchia dell'infrastruttura. Ci siamo ispirati ai Durable Objects di Cloudflare [43] per consentire una gestione efficiente delle sessioni, la coerenza dei dati e la scalabilità sull'edge.

La tesi inizia con una revisione completa dei framework e delle piattaforme FaaS esistenti, analizzando le loro metodologie, i risultati e le debolezze. Vengono esaminate applicazioni e casi di studio del mondo reale per valutare l'impatto delle funzioni stateful su efficienza, latenza ed esperienza dell'utente. Sulla base di questa analisi, si giustifica la necessità di un framework FaaS stateful per l'edge computing.

Vengono presentati il progetto e l'implementazione del nostro framework, evidenziando l'integrazione delle funzionalità stateful e dell'offloading automatico. Si discutono i miglioramenti architetturali, sfruttando la containerizzazione con Docker e l'orchestrazione con Kubernetes per ottenere portabilità e scalabilità. Descriviamo inoltre l'incorporazione di meccanismi di gestione delle sessioni e di migrazione dei dati per garantire una transizione fluida e un utilizzo ottimale delle risorse.

Per valutare le prestazioni e l'efficienza del nostro framework, sono stati condotti svariati esperimenti e benchmark che dimostrano i vantaggi della nostra soluzione FaaS stateful in scenari di edge computing reali.

Parole chiave: Faas, Serverless, Edge, Kubernetes, Cloud, Stateful, Offloading

Contents

1 | Introduction

In the rapidly evolving landscape of technology and data-driven industries, the concept of **Edge Computing** has emerged as a transformative paradigm. As the world becomes increasingly interconnected and reliant on real-time data processing, traditional centralized **Cloud Computing** infrastructures face *limitations in terms of latency, bandwidth, and privacy.*

1.1. Definition and Core Principles of Edge Computing

At its core, **Edge Computing** involves *bringing computation and data storage closer to the location where it is needed*, rather than relying on a centralized cloud infrastructure. Unlike traditional Cloud Computing, where data is sent to a remote server for processing, Edge Computing involves processing data at or near the edge of the network, typically on devices such as routers, gateways, or cloudlets. This proximity to data sources enables faster processing and reduced latency.

1.2. Key Trends and Drivers

There are several trends that are driving the adoption and growth of Edge Computing. First and foremost is the explosive growth of the **Internet of Things** (**IoT**), where an *increasing number of connected devices generate massive amounts of data that need to be processed* and analyzed in real-time. *Edge Computing provides a practical solution* by enabling data processing and analysis at the network edge, minimizing the need for data transfer to distant cloud servers. Additionally, the rise of autonomous vehicles, smart cities, and industrial automation further emphasizes the need for low latency and real-time decision-making, which can be effectively addressed through Edge Computing. Just two years ago, the research firm Gartner predicted that *By 2025, 75% of enterprise data will be generated and processed at "the edge."* [41]

1.3. The Big Data Problem

In recent years, the proliferation of **Internet of Things** (**IoT**) devices has led to a surge in data generation, *posing significant challenges to conventional data centers in terms of bandwidth and computational capacity*. Historically, cloud-centric solutions have been the predominant architectural choice due to the limited computational capabilities of edge devices, rendering them impractical for resource-intensive computational tasks. Moreover, managing data flows within a centralized data center has been more straightforward compared to managing a vast distributed heterogeneous system on a global scale.
However, the landscape has evolved as edge devices, including **IoT** devices and smartphones, *have progressively gained more computational power*, aided further by the introduction of **5G** networks, enabling greater data transmission capacities. Consequently, it has become evident that in the foreseeable future, Cloud Computing will remain the sole viable architectural approach for effectively managing the vast volumes of data generated.

1.4. Challenges and Opportunities

While Edge Computing holds immense promise, it also presents several challenges that need to be addressed for its widespread adoption, such as managing a **distributed computing infrastructure**, ensuring **data security and privacy**, dealing with **heterogeneous edge devices** and **connectivity issues** while developing efficient edge algorithms and architectures.
Many of the challenges about deploying on the edge have been identified by *Nick Barcet, Senior Director of Technology Strategy at Red Hat*:

> First is the ability to scale. Edge deployments can range from hundreds to hundreds of thousands of nodes and clusters that need to be managed in locations where there may be minimal to no IT staff at all. There has to be a centralized way to deploy and manage them [...].
> The second challenge is that edge deployments can vary greatly, which can make it hard for a single vendor to build an entire edge stack. Organizations need to ensure interoperability within a multi-vendor hardware and software environment.
> Lastly, we're thinking about consistency. It's almost impossible to manage all these deployments if they don't share a secure control plane via automation, management and orchestration. This is where the hybrid cloud is key, because you'll want to manage your entire infrastructure in the same way and create

> an environment where you can consistently develop an application once and deploy it anywhere. [22]

However, the challenges also present opportunities for innovation and growth. Companies are investing in Edge Computing solutions, developing edge-specific hardware and software, and exploring novel approaches to address the unique requirements of edge environments.

1.5. Scope of this Thesis

The purpose of this thesis is to **investigate the current state of the art in Edge Computing**, focusing on the development on the edge and the management of edge resources. By identifying gaps in the existing research landscape, **our goal is to propose innovative solutions and enhancements** that address these gaps and contribute to advancing the field of Edge Computing, by solving the three main challenges identified in the previous section.

1.6. Research Methodology

The research methodology adopted in this thesis involved a systematic approach to address the challenges and requirements of developing a stateful FaaS framework for Edge Computing. The key steps of the research methodology can be summarized as follows:

1.6.1. Literature Review and State-of-the-Art Analysis

The research began with an extensive literature review and analysis of the state-of-the-art in **Edge** and **Fog Computing**. The focus was on understanding the current landscape of hybrid infrastructure management and deployment, specifically the integration of edge and cloud resources. This analysis provided insights into the existing research trends, methodologies, and identified the gaps and limitations that need to be addressed.

1.6.2. Review of Available Frameworks

A comprehensive review of publicly available frameworks in the field of Edge Computing, stateful logic on the edge, and **Function-as-a-Service (FaaS)** was conducted. This review aimed to identify existing solutions and frameworks provided by the industry, assessing their capabilities and limitations. **By understanding the strengths and weaknesses of these frameworks**, it became possible to determine what was miss-

ing and what improvements were needed to develop a robust and efficient stateful FaaS framework.

1.6.3. Design of a Novel Solution

Based on the requirements and insights gathered from the literature review and framework analysis, a novel problem solution was designed. The design process focused on ensuring **efficient management of stateful functions**, **automatic offloading** to higher nodes in the infrastructure hierarchy, and **integration of edge and cloud** resources for seamless operation.

1.6.4. Evaluation of the Solution

To evaluate the effectiveness and performance of the proposed solution, a prototype implementation was developed. Additionally, simulations were conducted to assess the behavior and scalability of the stateful FaaS framework in various scenarios. The evaluation process involved measuring key metrics such as **latency** and **bandwidth utilization**.

2 | Edge Computing Today

2.1. Preliminaries

In this chapter, we will explore the **current landscape of Edge Computing** and its relevance in modern computing architectures. We will review recent papers, particularly focusing on the proceedings of the "**IEEE International Conference on Fog and Edge Computing** (**ICFEC**)" of recent years to understand the **development**, **management**, and **deployment** of Edge Computing applications. Our goal is to identify the gaps in existing research and propose a framework that addresses these gaps by combining the benefits of Edge Computing and **FaaS** (**Function as a Service**) frameworks.

We will also critically analyze the methodologies, results, and potential weaknesses of the reviewed papers, providing a comparative evaluation of different approaches. Additionally, we will discuss emerging trends in Edge Computing and highlight areas that require further research and exploration. Real-world applications and case studies will be examined to understand the practical utilization of Edge Computing concepts and their impact on **efficiency**, **latency**, and **user experience**.

2.1.1. Edge Computing Background

Edge Computing is a paradigm where computing and storage nodes are located at the Internet's edge, close to mobile devices or **IoT** sensors. Unlike traditional centralized models where computational power is concentrated in on-premise data centers, Edge Computing enables processing data closer to where it is generated. The concept of Edge Computing originated from **Content Delivery Networks** (**CDN**) introduced in the late 1990s to **enhance web performance**. **CDNs** utilized edge machines to **cache frequently requested content**, reducing bandwidth usage and improving latency. Edge Computing builds upon the **CDN** concept, extending it to a geo-distributed environment.

2.2. Why Edge Computing

Edge Computing is gaining significance due to its unique advantages over traditional centralized models. By processing data closer to its source, **Edge Computing enables faster and larger-scale processing**, leading to real-time action and enhanced user experience. This approach addresses the limitations of Cloud-centric architectures, which may struggle to handle the increasing data volumes and requirements for low latency [22].

Several factors drive the need for Edge Computing [15]:

- **Data Scale**: The exponential growth of data surpasses the capacity of **Cloud** infrastructure alone.
- **5G and IoT**: The emergence of 5G and the proliferation of **IoT** devices enable the widespread adoption of edge technologies.
- **Latency Constraints**: Latency-sensitive applications require processing closer to the data source to ensure faster response times.
- **Efficiency and Resource Optimization**: Edge Computing reduces the impact on centralized resources, preventing future scalability issues in the Cloud.

2.3. Advantages of Edge Computing

Edge Computing offers several advantages over centralized Cloud-centric models:

- **Reduced Latency**: Processing data closer to the source minimizes network latency and improves response times for time-critical applications.
- **Bandwidth Optimization**: Edge Computing minimizes the need to transfer large volumes of data to the Cloud, reducing bandwidth requirements and associated costs.
- **Resource Proximity**: Edge nodes are geographically distributed, enabling data processing near the point of generation and reducing reliance on distant Cloud resources.
- **Improved Privacy**: Edge Computing facilitates compliance with data privacy regulations by keeping data locally within the edge network.

2.4. Disadvantages of Edge Computing

While Edge Computing offers significant advantages, it also comes with certain limitations:

- **Limited Computational Power and Resources**: Edge nodes typically have lower computational power, limited memory, and lower persistence capabilities compared to Cloud data centers.
- **Hardware Reliability and Battery Dependence**: Edge devices may rely on less reliable hardware, and their operation is often dependent on battery life.
- **Fault Tolerance Challenges**: Implementing fault-tolerant mechanisms in edge environments can be challenging due to resource constraints and distributed nature.

2.5. Low Latency Driver

Low latency is a critical factor driving the adoption of Edge Computing. The benefits of low latency are particularly noticeable in the following scenarios:

1. **Microfunctions with Execution Time Comparable to Latency**: For small microfunctions, where the execution time is comparable to the latency, reducing latency becomes essential. This becomes more evident when multiple message exchanges occur between edge devices and the **Cloud**, nullifying the advantage given by the proximity to the end user.
2. **Real-Time and Fast Response Requirements**: Applications that require real-time and near-instantaneous data processing can greatly benefit from Edge Computing, as it reduces the time between data generation and processing.
3. **Geographically Distributed Data**: Edge Computing allows data to be processed and stored close to its geographical origin, ensuring compliance with regulations such as GDPR and reducing data transfer latency.
4. **Autonomous Distributed Systems**: Edge Computing enables the implementation of logic between network components without relying on the **Cloud**. This is particularly relevant for autonomous distributed systems, such as **IoT** networks, where devices exchange data among themselves.

2.6. Management of Scarcity of Edge Resources

Edge Computing introduces the challenge of managing scarce resources, which distinguishes it from traditional core-centric infrastructures.

As highlighted in the paper "**Efficient Edge Storage Management Based on Near Real-Time Forecasts**" [29] presented at the (**ICFEC**) in 2017, data analytics on the edge requires historical data for accurate analysis. *"Since storage capacities on the Edge are limited, we are faced with a challenge to balance the quantity of data stored with the quality of near real-time decisions"*. For this reason, they presented a solution that dynamically finds a trade-off between providing high forecast accuracy necessary for efficient real-time decisions, and minimizing the amount of data stored in the space-limited storage. We will try to face this challenge by **implementing a framework capable of relocating data** to keep them as near to the edge as possible, and move them to the other nodes whenever needed.

2.7. Serverless on the Edge

Serverless computing has emerged as an abstraction that frees developers from infrastructure management concerns, allowing them to focus solely on business logic implementation. The Serverless model works effectively on the **Cloud**, where homogeneous and readily available resources facilitate immediate scalability and cost-efficiency.

Nastic et al. in their article "A Serverless Real-Time Data Analytics Platform for Edge Computing" [32] shed light on the limitations of current approaches for data analytics on the edge, which often require developers to employ ad hoc solutions tailored to the available infrastructure. They propose an architecture that abstracts the complexity of edge infrastructure, enabling data processing and analytics while providing the benefits of Serverless computing. Key concepts of their architecture include:

- **Local and Global Views**: The edge focuses on local views, while the **Cloud** supports global views, allowing developers to define function behavior and data processing logic without being burdened by infrastructure complexities.
- **Function Wrapper Layer**: A layer that manages user-provided functions, wrapping them in executable artifacts such as containers and facilitating their deployment.
- **Orchestration Layer**: Responsible for determining the most suitable node (**Cloud** or edge) for executing an analytics function, optimizing for reduced network latency.

- **Runtime Layer**: Manages the provisioning, deployment, scheduling, and execution of functions, adapting to varying resource requirements.
- **Implicit State Management**: The function wrapper layer handles state replication and migration transparently, providing controlled access to a function's state through an exposed API.

The paradigm shift triggered by Serverless computing, with its stateless, pay-per-use, event-driven, and distributed nature, is crucial for enabling the adoption of edge compute, where similar principles apply. Building upon these concepts, we aim to develop a Serverless platform for Edge Computing.

2.8. Stateful FaaS on the Edge

While **Function as a Service** (**FaaS**) is well-suited for executing stateless functions, supporting **Stateful** constructs and distributed transactions across multiple functions remains a challenge. Access to remote state introduces increased latency and network traffic, limiting the effectiveness of FaaS in Edge Computing scenarios. Thus, it becomes necessary to implement a **Stateful** FaaS Framework.

This problem was well described in the paper "**Fado**: FaaS functions and data orchestrator for multiple Serverless edge-Cloud clusters" [38] presented in 2022 IEEE 6th International Conference on Fog and Edge Computing (**ICFEC**):

> The FaaS platform is responsible for deploying and facilitating resources to the application functions. The benefit of not having to manage the infrastructure also comes with some challenges, one of them being the function placement. To achieve good performance, the infrastructure should be able and willing to co-locate particular code and data physically. This is often best achieved by shipping code to data, rather than the approach of pulling data to code. When using **FaaS**, the **Cloud** provider is responsible for scheduling the workload. However, data placement is not considered, leading to a non-optimal performance. The function's execution time varies based on the location of its accessing data. A local data access will be faster than remote data access, which has an overhead of network latency. The data-access variations cause differences in performance between the identical function deployments for the same Serverless compute platform. Latency-sensitive tasks, such as media streaming or complex distributed machine learning calculations, are thus not well suited to the current **FaaS** model.

In their paper, **the researchers proposed a solution based on replicating data on a distributed database over the same edge nodes where functions are deployed**. In this way they just had to implement a load balancer in front of an existing **FaaS** infrastructure and track the position of replicas containing data from sessions contained in request headers, in order to redirect incoming requests to nodes who already had local replicas of data. They **did not have actual control on where data where stored**, but instead they tried to redirect incoming requests to nodes with local replicas available whenever possible.

We instead aim to deliver an enhanced **solution that optimizes the allocation and migration of data**, ensuring their proximity to the client at all times.

A previous attempt to address the challenges of stateful FaaS on the edge was conducted by Dennis Motta in his Master's Thesis titled "**Location-aware and stateful serverless computing on the edge**" [30]. Motta's work shares similarities with ours in terms of implementing stateful functionality using a lightweight in-memory database like Redis. However, there is a notable difference between his approach and ours regarding the management of limited capabilities of edge nodes.

In Motta's framework, **the focus was on propagating data to upper nodes in order to delegate more resource-intensive tasks**, such as data aggregation and computation, to those nodes. He designed the framework in a way that **developers were responsible for implementing the logic to manage and process data** at different levels of the infrastructure. In contrast, **our approach aims to dynamically reallocate both data and code execution in response to runtime conditions**, by taking into account factors such as resource saturation and the number of incoming parallel requests to make intelligent decisions regarding data and code placement.

2.9. Exchange of Metadata for Session Tracking in a P2P Fashion

Efficient **resource discovery** and service provisioning within Edge Computing environments **often rely on the exchange of metadata in a peer-to-peer (P2P) manner**. In such systems, devices located in proximity form an edge neighborhood and collaborate by exchanging information about available resources, enabling local service discovery for **IoT** applications [31].

In our framework, **we adopt a similar approach** to enable session tracking and management. Instead of centralizing metadata and control, we encourage **edge nodes to**

exchange information about their available resources and actively participate in the discovery and provisioning of services. This decentralized approach allows for scalability, fault tolerance, and efficient resource utilization within the edge network.

2.10. Distributed Transactions on Serverless Stateful Functions

Distributed transactions across Serverless Stateful functions present challenges in terms of managing state, ensuring consistency, and coordinating transactions among functions. While FaaS platforms excel in executing stateless functions, **supporting transactional workflows remains an open problem**.

To address this limitation, **our framework aims to support distributed transactions with proper isolation and consistency** guarantees. We draw inspiration from the solution proposed in the paper "Distributed transactions on Serverless Stateful functions" [13], presented at the **15th ACM International Conference on Distributed and Event-Based System**. They designed their programming model on top of Apache Flink StateFun, in order to leverage Flink's exactly-once processing and state management guarantee.

To address the limitations of using Flink on the edge due to its computational overhead, we will design and implement a custom lock mechanism. This mechanism will **ensure client-centric consistency among function executions and provide an "at most once" execution guarantee**, minimizing the risk of data corruption.

2.11. Fog Architecture and Hierarchical Resource Representation

Fog computing architectures leverage compute **nodes distributed between the Cloud and smart things** to process **IoT** data close to its source. These architectures often adopt **hierarchical structures**, where compute nodes are organized in layers. Cloud-based resources reside at the top of the hierarchy, while edge nodes form the middle layer, and **IoT** devices operate at the bottom [26].

The hierarchical architecture allows for **efficient data processing**, **reduced latency**, and **higher bandwidth capacity** within the edge environment. Quantitative results from a comparative analysis of architectures by Karagiannis et al. [26] indicate that hierarchical architectures exhibit lower communication latency and improved bandwidth

utilization compared to flat architectures. This finding emphasizes the benefits of adopting a hierarchical representation in our framework.

2.12. Migration of Edge Services

The migration of edge services, particularly in geo-distributed environments, is a common operation driven by factors such as moving data sources or changes in the network infrastructure. However, traditional container orchestration systems like Kubernetes were designed for centralized Cloud environments, **lacking proper support for efficient edge service migration**. To address this gap, recent research proposes frameworks specifically tailored for edge service migration.

One recent approach proposed in "**Stateful Function as a Service at the edge**" [36] was to make the **developer implement two different versions of the same function**, one to be executed on the edge, and one on the Cloud. Their framework was then in charge of **migrating the local state to the Cloud whenever needed**, and redirect the client consequently. A better approach for pod relocation was proposed on the paper "**Good Shepherds Care for Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes**"[24]: a **pod migration technique** for Kubernetes that enables transparent migration while minimizing downtime. This technique involves checkpointing the container's memory state and network connections, then resuming the saved state in the new pod.

These **existing solutions lack flexibility** as they either completely migrate the state to the Cloud or distribute it across edge nodes without considering the infrastructure's topology. Additionally, they do not aim to achieve an optimal allocation of resources across edge nodes. **We aim to provide a Stateful FaaS platform that dynamically adjusts to environmental changes** and seamlessly **migrates both data and computation** to the nearest edge node to the client. By prioritizing edge resources and utilizing the Cloud as a last resort when local resources are insufficient, we would be able to minimize latency and network traffic in order to achieve optimal performances.

2.13. Conclusion

In this chapter, we explored the current state of Edge Computing and its significance in today's technological landscape. We highlighted the advantages of Edge Computing, such as reduced latency, bandwidth optimization, improved privacy, and resource proximity. Additionally, we discussed the challenges associated with Edge Computing, including

limited resources, hardware reliability, fault tolerance, and the need for effective resource management.

We also discussed the emergence of Serverless computing on the edge, focusing on the concepts of Serverless architectures, **Stateful FaaS**, metadata exchange for session tracking, distributed transactions, hierarchical resource representation, and edge service migration. These areas of research will be essential for the development of our proposed framework.

2.13.1. Functionalities to Implement in Our Framework

Based on the insights gained from the discussed papers, we have identified the following functionalities to implement in our framework:

1. **Data and computation locality**: Our framework will aim to keep data close to computation, minimizing latency and bandwidth constraints.
2. **Efficient resource utilization**: We will develop resource management and optimization techniques to effectively utilize edge resources, considering their limited computational power, memory, and reliability.
3. **Serverless platform for Edge Computing**: Following the Serverless model, we will provide a platform that abstracts the underlying infrastructure complexities, allowing developers to focus solely on defining function behavior and data processing logic.
4. **Stateful function support**: We will enable Stateful functions on the edge, allowing them to access and manipulate data stored locally within the edge network. Our approach will consider data replication and migration to ensure consistency and mitigate data corruption risks.
5. **Peer-to-peer session tracking**: Our framework will utilize a decentralized approach for session tracking, allowing edge nodes to exchange metadata and collaborate in discovering and sharing available resources.
6. **Hierarchical representation of resources**: We will adopt a hierarchical architecture for our framework, with Cloud nodes at the top and low-power edge nodes at the bottom, enabling efficient resource utilization and data processing in edge environments.
7. **Migration and resource relocation**: We will develop a mechanism for transparent migration and resource relocation, allowing dynamic movement of functions or containers within the edge network while minimizing service downtime and main-

taining uninterrupted client access.

By incorporating these functionalities into our framework, we aim to address the gaps in current research, enable efficient Edge Computing, and enhance the user experience in real-world applications.

3 | Use Cases

3.1. Real-World Applications and Case Studies

A Stateful FaaS platform with automatic **offloading** to higher nodes in the infrastructure hierarchy has various real-world applications and can bring significant benefits to different domains. In this chapter we show a few examples found by analyzing other scientific papers.

3.1.1. Smart Cities

In a smart city context, a Stateful FaaS platform can **support real-time data processing and decision-making**. For instance, in a traffic management system, edge devices such as traffic sensors and cameras can generate data that needs to be processed locally for immediate actions. It enables the deployment of traffic analysis functions on edge nodes, ensuring low **latency** and efficient resource utilization. Additionally, the platform can automatically **offload intensive computational tasks to higher nodes**, such as aggregating data from multiple edge nodes and performing long-term traffic pattern analysis in the cloud. [20]

3.1.2. Industrial Internet of Things (IIoT)

In industrial settings, a Stateful FaaS platform can **facilitate real-time monitoring and control of manufacturing processes**. Edge devices, such as sensors and **Programmable Logic Controllers** (**PLCs**), can generate data that requires immediate processing and response. By deploying Stateful functions on edge nodes, the platform enables **low-latency data processing**, enabling real-time decision-making and reducing reliance on cloud connectivity. Furthermore, the automatic **offloading** capability allows resource-intensive tasks, such as complex analytics and predictive maintenance, to be performed in the cloud. [19]

3.1.3. Augmented Reality (AR) and Virtual Reality (VR)

AR and VR applications often require real-time data processing and low-latency interactions. With a Stateful FaaS platform, edge devices such as **AR/VR headsets can leverage local processing capabilities** to achieve **low latency** and provide immersive user experiences. For example, in a collaborative AR environment where multiple users interact in real-time, the platform can enable distributed processing and data synchronization among edge nodes, while **offloading computationally intensive tasks**, such as **rendering complex graphics**, to higher nodes for enhanced performance. [12]

3.1.4. Healthcare

In healthcare applications, a Stateful FaaS platform can support **real-time monitoring and analysis of patient data**. Edge devices, such as wearable sensors and medical devices, can collect and process patient data locally, ensuring **privacy** and reducing **latency** for critical health monitoring tasks. The platform enables the deployment of Stateful functions on edge nodes for **continuous monitoring**, **real-time alerts**, and **immediate responses**. Additionally, resource-intensive tasks, such as advanced data analytics and long-term health trend analysis, can be offloaded to higher nodes for comprehensive patient care. [26] [37]

3.1.5. Retail and E-commerce

In the retail and e-commerce industry, a Stateful FaaS platform can support **real-time inventory management** and **personalized customer experiences**. Edge devices, such as smart shelves and beacons, can **collect data on product availability** and **customer preferences**. By deploying Stateful functions on edge nodes, the platform enables efficient inventory tracking, real-time stock replenishment, and personalized recommendations for customers. Resource-intensive tasks, such as **demand forecasting and advanced analytics**, can be **offloaded** to higher nodes to optimize inventory management and improve customer satisfaction. [33]

3.1.6. Smart Grid and Energy Management

In the context of smart grids and energy management, the Stateful FaaS platform can enable **real-time monitoring and control of energy generation and consumption**. Edge devices, such as smart meters and sensors, can collect data on energy usage and environmental conditions. By deploying Stateful functions on edge nodes, **the platform**

facilitates local energy optimization, **load balancing**, and **fault detection**. Additionally, it can automatically **offload energy forecasting and demand-response algorithms** to higher nodes for efficient energy management and grid stability. [26] [44]

3.1.7. Logistics and Supply Chain Management

In logistics and supply chain management, the Stateful FaaS platform can **support real-time tracking and optimization of goods movement**. Edge devices, such as **RFID tags** and **GPS trackers**, can **provide data on shipment status and location**. By deploying Stateful functions on edge nodes, the platform enables **efficient route planning**, **real-time inventory visibility**, and **proactive issue resolution**. **Complex logistics optimization algorithms and supply chain analytics can be offloaded** to higher nodes for enhanced efficiency and decision-making. [39]

3.1.8. Environmental Monitoring and Conservation

In environmental monitoring and conservation efforts, the Stateful FaaS platform can **facilitate real-time data analysis and proactive interventions**. Edge devices, such as sensors and drones, can collect data on air quality, water quality, and wildlife habitats. By deploying Stateful functions on edge nodes, **the platform enables localized data processing**, **early detection of environmental threats**, and timely responses to preserve ecosystems. Resource-intensive tasks, such as **species population modeling and environmental impact assessments, can be offloaded** to higher nodes for comprehensive analysis and planning. [26] [45]

3.1.9. Speech-to-Text and Cloud AI Integration

The Stateful FaaS platform enables **speech-to-text conversion on the edge** and seamless integration with cloud-based AI services. By deploying Stateful functions on edge nodes, real-time speech recognition and transcription can be performed locally. The **converted text can then be offloaded** to higher nodes, such as the cloud, to **leverage advanced language understanding and response generation capabilities**. This integration facilitates interactive voice-controlled applications with **reduced latency** and enhanced **privacy**, making it suitable for **voice assistants**, **IVR systems**, and **voice-enabled applications** in various domains. [46]

These additional use cases highlight the versatility of a Stateful FaaS platform in various industries and domains. By combining the benefits of Stateful function management,

automatic offloading, and **hierarchical resource management**, the platform can enable efficient and intelligent edge computing solutions tailored to specific application requirements.

4 | Existing Solutions

In this chapter, we will explore the current **FaaS** frameworks and platforms available in the industry for **serverless** computation on the edge. We will critically analyze their methodologies, results, and potential weaknesses and justify the choice of creating a modified version of **OpenFaaS** for deploying **stateful** functions on the edge with automatic offloading.

4.1. Hosted Serverless Edge Paltforms

In this section, we will discuss the **serverless** frameworks offered by industry leaders for performing edge computations and evaluate their **stateful** support capabilities.

4.1.1. AWS Lambda@Edge

AWS Lambda@Edge allows running **serverless** code on the edge network of AWS. By leveraging the global infrastructure of **Amazon CloudFront**, Lambda@Edge enables the execution of functions at various edge locations. This reduces **latency** and enhances the responsiveness of applications. However, in terms of stateful support, Lambda@Edge primarily **focuses on caching**, **stateless processing**, and **content delivery use cases**. While it offers the capability to store and retrieve data from CloudFront's cache, it **lacks comprehensive stateful capabilities** required for more complex applications that involve frequent write operations and session management. [3]

4.1.2. Akamai EdgeWorkers

Akamai EdgeWorkers, similar to Cloudflare Workers, provides **serverless computation capabilities on the edge**. Akamai offers **stateful** support through the **Akamai EdgeKV**, a **global key-value database** with eventual consistent writes. EdgeKV allows developers to **store and retrieve data at the edge**, enabling stateful operations. However, like other key-value stores, it is more **suitable for applications with a higher number of reads than writes**. [2]

4.1.3. Appfleet

Appfleet enables developers to **deploy long-running containers in multiple locations globally**, aiming to run services closer to users. While Appfleet is not yet considered a comprehensive edge network due to its **limited number of locations**, it offers the potential for future expansion. However, as of now, Appfleet's stateful support is not specifically focused on edge computing. [5]

4.1.4. Cloudflare Workers

Cloudflare Workers is a serverless platform that enables the **execution of code across Cloudflare's global edge network**. Workers are designed to be highly efficient, with low cold start times and resource limitations (e.g., CPU time and memory). Cloudflare provides **stateful support through their Cloudflare Workers KV and Cloudflare Durable Objects**.

Cloudflare Workers KV is a **global key-value data store** that allows storing and retrieving data from edge locations. It provides low-latency access to data and can be used for various stateful use cases. However, Workers KV's eventual consistency model makes it more suitable for applications with frequent reads but infrequent writes. [10]

Cloudflare Durable Objects offer a more robust stateful solution. Durable Objects allow developers to create **stateful objects with methods and properties that can be accessed and manipulated across multiple requests**. These objects are stored and executed on Cloudflare's edge network, providing **low-latency access and scalability**. Durable Objects are particularly useful for applications that require maintaining stateful sessions or managing complex workflows. The ability to store and update state within Durable Objects provides a foundation for building sophisticated edge applications that require stateful computations. [9]

4.2. Solutions Summary

The existing frameworks for **serverless** edge computing offer varying degrees of **stateful support**. Cloudflare Workers, with their Cloudflare Workers KV and Cloudflare Durable Objects, provide robust **stateful** solutions that enable the storage and management of data on the edge. These offerings have proven to be valuable in real-world applications, addressing the limitations of purely stateless edge computing.

In contrast, other frameworks, such as **AWS Lambda@Edge** and **Akamai EdgeWork-**

ers, primarily focus on caching, stateless, and forwarding use cases. While they provide some stateful support through key-value databases, their **capabilities are limited** compared to Cloudflare's offerings.

Considering the available frameworks and their strengths and weaknesses, it is evident that **stateful support plays a crucial role** in enabling more complex and efficient edge computing applications. The ability to **maintain state and manage sessions on the edge can significantly impact latency, user experience, and application performance**.

The **existing frameworks** for serverless edge computing **lack robust stateful support** for use cases involving frequent write operations. While caching, stateless, forwarding, and infrequent-write scenarios can be fulfilled, more **complex applications requiring intensive write operations are not well supported**.

Considering the limitations of the available frameworks and the need for efficient solutions, it is evident that researchers should focus on studying systems capable of on-demand edge computing without the reliance on long-running containers.

Given the absence of comprehensive stateful support in current frameworks, we sought to devise a new solution to address the identified shortcomings.

4.3. FaaS Platforms

FaaS platforms serve as the foundation for building serverless architectures. In this section, we explore some popular FaaS platforms and assess their suitability for our stateful edge computing requirements

4.3.1. Apache OpenWhisk

Apache OpenWhisk is an open-source **FaaS** platform that **supports multiple programming languages** and provides a **scalable infrastructure** for executing functions. Its architecture includes components like a document-oriented database (**CouchDB**) and a messaging platform (**Kafka**) for processing requests. While OpenWhisk offers **scalability**, its resource requirements and **complexity make it unsuitable for edge computing**. A lighter version called **Lean OpenWhisk** was developed to address these limitations but **has been discontinued**, leaving a gap in lightweight edge computing platforms based on OpenWhisk. [4]

4.3.2. Fission

Fission is a well-supported **FaaS platform** that operates on top of **Kubernetes**, offering **low cold-start latency** and **auto-scaling capabilities**. It supports multiple languages and employs a stateless router component for handling HTTP requests. Fission's architecture includes an executor component responsible for starting function pods that fetch the function information from Kubernetes Custom Resource Definitions. Although Fission provides an efficient container-based approach for deploying functions, it **does not natively support stateful functions**. [18]

4.3.3. OpenFaaS

OpenFaaS is an open-source FaaS platform known for its **scalability** and **extensibility**. It supports a **wide range of programming languages** and can be **deployed on Kubernetes**. OpenFaaS offers two flavors: "**faas**" for scalability and "**faasd**" for running on hardware-limited devices. The faas flavor provides horizontal scalability using Kubernetes and **Prometheus** for auto-scaling. The faasd flavor, designed for resource-constrained environments, **does not support horizontal scaling**. OpenFaaS serves as a flexible and widely adopted FaaS platform, making it a **suitable base for implementing stateful edge computing** capabilities. [6]

4.3.4. Other Platforms

There are several other **FaaS** platforms such as **OpenLambda**, **Fn Project**, and **Qinling**, which have either been discontinued or have limited adoption. While they may have provided interesting concepts and ideas, they are not extensively used in the industry or actively maintained, making them **less suitable for our stateful edge computing** requirements.

4.4. Addressing Challenges: Towards a Stateful FaaS Framework for Edge Computing

The **existing FaaS** platforms and frameworks, although offering valuable insights and functionalities, **lack comprehensive support for stateful operations**, automatic **scaling**, and **edge computing** requirements. Lightweight **FaaS** frameworks with native **stateful** capabilities for efficient session management and automatic scaling on the edge are currently not prevalent in the industry. Additionally, hybrid solutions that seam-

lessly manage both cloud and edge environments while maintaining a unified programming model and infrastructure abstraction are missing.

To address these gaps, **we have chosen to develop a modified version of OpenFaaS**, leveraging the lightweight nature of edge computing, **incorporating stateful support** inspired by **Cloudflare's Durable Objects**, and implementing **automatic scaling** mechanisms. This approach allows us to provide developers with a robust and efficient **framework for deploying stateful functions** on the edge while maintaining the advantages of popular FaaS platforms.

In the subsequent chapters, we will delve into the **architecture**, **implementation**, and functionalities of our **stateful FaaS framework**. We will discuss in detail the incorporation of **session management**, **automatic scaling**, and the impact of these features on **efficiency**, **latency**, and **user experience**. Furthermore, we will explore potential optimizations, additional functionalities, and future directions for the framework.

5 | Design of the Solution

To fulfill the identified requirements derived from our research, we have designed and implemented a comprehensive **framework that facilitates the development and deployment of stateful functions**. Our framework enables developers to seamlessly write and deploy stateful functions across a network infrastructure consisting of a **hierarchical tree of nodes**, extending from edge nodes to the cloud (located at the root of the tree). This architecture facilitates **automatic session offloading to higher-level nodes or onloading to lower-level nodes**, based on the availability of resources.

Below is a comprehensive list of the features offered:

- Write **stateful functions** with the following characteristics:
  - **Blocking** functions with **read/write permissions** on sessions
  - **Non blocking** functions with **read-only permissions** on sessions
- **Define a network** of nodes
- **Deploy stateful functions** on the defined network
- **Sessions hold key-value data** and key-list data
- **Sessions** can be set to be **automatically deleted**
- **Automatic offload/onload** based on resource availability
- **User-defined offload/onload logic** by writing a custom function and leveraging our internally exposed offload/onload API

To address the implementation of the aforementioned features, we need to consider the following aspects:

- How to handle incoming connections
- How to implement the concept of a session
- When to trigger offload/onload

- How to manage onload/offload
- Which sessions have to be migrated
- How to manage consistency of data during migration
- How to manage consistency of data during parallel accesses

5.1. The Session

A session is a bucket of data associated with a SESSION-ID, which is included as a mandatory header in each client request.

Sessions are **stored in an in-memory key-value database**, where each entry can be either raw data stored as a string or an array of strings.

To ensure consistency when multiple accesses occur simultaneously to the same session (either by invoking the same function twice or by invoking two different functions with the same SESSION-ID header), or when a function gets interrupted during its execution (due to a connection error or an execution timeout expired) we have adopted a **client-centric consistency model**. This model incorporates a **locking mechanism to prevent concurrent access by non-pure functions to the same session**. Additionally, leveraging the atomic operation capabilities of REDIS within a transaction, **all write operations performed during the execution of a single function are temporarily cached**. These writes are then **atomically committed** at the end of the function's execution, ensuring that either all writes are completed or none of them are.

In the subsequent sections, we will provide further details on how we implemented the client-centric consistency model to achieve "at-most-once" execution. We will also discuss how clients can resend the same request in case of failure, without the request being executed twice.

5.1.1. Storage

Focusing on the objective of achieving fast execution and low latencies, we have implemented the memory layer using REDIS, an **in-memory database**. REDIS provides fast access times and the option to automatically store logs on disk for data restoration in case of failure. By default, this optional feature is disabled in our framework, as we assume that edge nodes may not have access to permanent storage. However, if necessary, it can be easily enabled.

Furthermore, we assume that if permanent storage is ever required, traditional cloud-based solutions can be utilized. In such cases, data can be stored asynchronously, eliminating the need to connect to the cloud during function execution, thereby avoiding unnecessary delays in processing requests. For instance, a chrono-function could be employed to periodically send and store data in the cloud while deleting the local copy to free up memory on the REDIS instance.

It is important to note that the data access interface we provide is not strictly tied to the use of REDIS. Any other database, such as CockroachDB, can be used as long as the same interface is re-implemented to support it.

The interface offered by our framework to access the data can be found in section 6.2.2.

5.2. Data Consistency

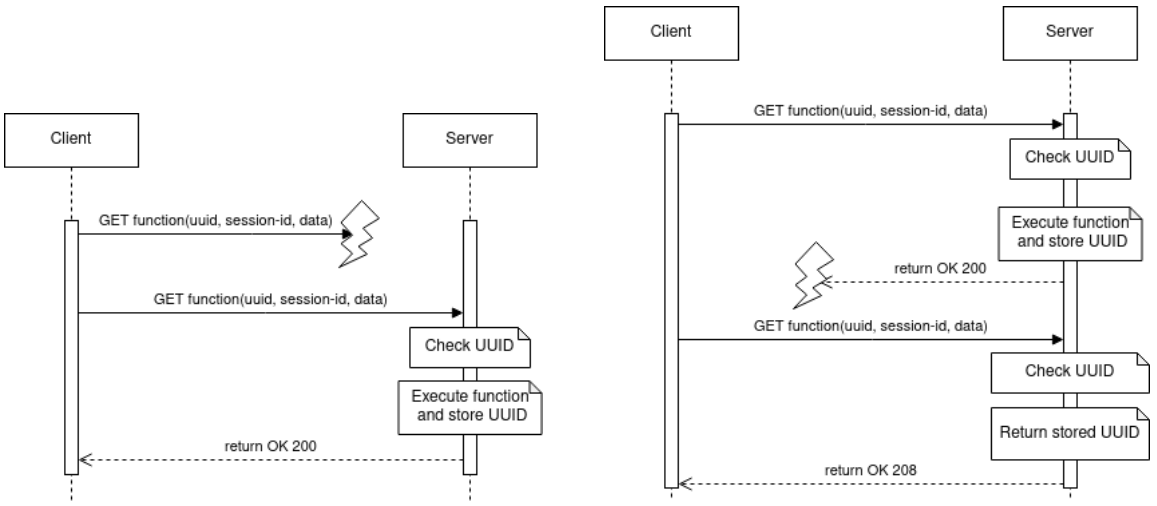
Before diving into the details on how we secured data consistency, it is worth mentioning that the contents of this section 5.2 only applies to blocking functions, except for the first sub section that discuss client centric consistency through the use of client generated UUIDs, which is a feature used by both blocking functions and non blocking functions.

5.2.1. Client Centric Consistency

As mentioned earlier, we have implemented a **client-centric consistency model to guarantee that requests are executed at most once**, ensuring data consistency from the client's perspective. This is accomplished by **requiring the client to generate a unique identifier**, specifically a UUIDv4 [28], for each request.

By assigning a unique identifier to each request, we can implement a **server-side mechanism that prevents duplicate execution of requests**. If a client does not receive a successful response (e.g., due to a timer expiration or a connection issue), it can simply reissue an equal request while retaining the same UUID. The server will attempt to execute the request only if it has never been completed before. Otherwise, it will respond with an HTTP 208 status code. Here is a scheme of the aforementioned mechanism:

1. The client generates a request, associates a unique id to it and sends it
2. The server receives the request and discards it if the id was previously stored, responding with a 208 http status code
3. If the id was not stored, the server elaborates the request, stores the id and sends a "success" response to the client



(a) Request packet lost. (b) Response packet lost.

Figure 5.1: Examples of how the requests are handled in the case of packets losses by means of client centric consistency.

Figure 5.1a illustrates the scenario where a request is lost. The client sends a request to the server, but the request never reaches the server. When the client's connection times out, it sends another request with the exact same parameters, including the same UUID, session ID, and function parameters. Upon receiving the request, the server checks the UUID and does not find it in the server's UUID cache. As a result, the function is executed, and a 200 response is sent back to the client.

In Figure 5.1b, the scenario depicted is when the response is lost. The first request is successfully received by the server, and the UUID is checked (not found in the UUID cache). The function is then executed, and the response is sent back to the client. However, due to connection issues, the response is lost. After the client's connection times out, it sends a new request to the server. The server checks the UUID and finds it in the UUID cache, indicating that the request has already been executed in the past. In response, the server sends a 208 response to inform the client that the request has been previously processed.

5.2.2. Session Consistency and Atomic Writes

In order to guarantee consistency of the sessions among multiple requests with the same UUID, we have adopted an approach similar to the one used by Cloudfare with Durable Objects [42]. Before the actual execution of a function, the associated session is retrieved from Redis. The name of the session must be provided as a SESSION-ID header. If the session does not exist in Redis, a new empty session is created.

During function execution, all read and write operations are performed on a local cache, resulting in significantly faster response times compared to contacting Redis for each request. **At the end of the execution, all write operations are atomically executed on Redis**. The UUID of the request gets saved as "completed" and a 200 OK response is returned to the client. This ensures that the **write operations are either executed altogether or not at all**.

The same functionality is offered for pure functions, with the only difference being that any writes made to the cache are flushed and lost at the end of the execution.

Thanks to this functionality, we can effectively prevent inconsistencies in various scenarios, including:

- **Partial writes**: If a function fails in the middle of its execution, it may only complete a subset of the intended write operations, leading to an inconsistent session state.
- **Multiple executions**: In cases where a function executes correctly but fails to return a 200 response to the client (either due to a connection problem or an expired execution timer), the client may reissue the same request. Multiple executions of the same function for the same request can potentially result in an inconsistent session from the client's perspective.

5.2.3. Parallel Access Consistency

The previously described mechanisms work effectively when only one function at a time accesses a particular session. However, in scenarios involving **multiple clients** or a single client sending multiple parallel requests, there is a **possibility of encountering consistency issues**.

Consider this scenario: a request 'A' gets processed before a request 'B'. When 'A' is successfully completed, including its write operations, while 'B' is still in progress, request 'B' may utilize values that have been modified by request 'A', resulting in inconsistent write operations for 'B'.

To address this challenge and ensure consistency, additional measures need to be implemented, such as locking mechanisms or transactional operations. These mechanisms can prevent concurrent access to the same session, guaranteeing the integrity and consistency of session data even in the presence of multiple parallel requests.

5.3. Session Lock

To address the challenge of consistency in the presence of concurrent requests, we have integrated a locking mechanism into our framework:

When a request is received, a lock is automatically created and associated with the SESSION-ID present in the request header. This process is carried out atomically using a LUA script and the lock is stored in REDIS.

If a lock is already present for the same session, the function immediately terminates and returns an HTTP 503 error, indicating that the resource is temporarily unavailable. This **ensures that only one function can access the session at a time**.

At the end of the function execution, the lock is released, allowing other functions to access the session. Simultaneously, all the write operations are performed atomically, ensuring the integrity of the session data.

5.3.1. Failover Implementation with TTL

While the simple locking mechanism we previously described is effective, it has one potential drawback. If the execution of a **function is interrupted for any reason**, such as a system failure or a timer expiration, the **lock associated with the session may not be released**, resulting in a deadlock situation.

To address this issue, we have **enhanced the locking mechanism by introducing a timeout feature**. The timeout duration is set to match the execution timeout of the function. If the lock is not released within the specified timeout period by the executing function, it will be automatically released.

In cases where a lock times out before the completion of the function execution, the function automatically fails, and an HTTP 423 "LOCKED" error response is returned. This ensures that the lock is released in a timely manner and **prevents potential deadlocks in the system**.

By incorporating this timeout feature, we enhance the reliability and resilience of our locking mechanism, mitigating the risk of deadlocks caused by unexpected interruptions during function execution.

5.3.2. Unique Lock Id

One remaining challenge to address is **how to determine if the lock being unlocked is the same lock that was initially acquired**. Due to the introduction of lock expiration, it is possible for another function to acquire the lock before the first function completes.

Consider the following scenario involving three function executions:

- Function A: starts execution and the lock expires, but the function itself does not terminate
- Function B: starts execution after Function A's lock has expired and terminates before Function A
- Function C: starts execution after Function B terminates. However, before Function C can complete, Function A terminates, "stealing" the lock from Function C

This scenario demonstrates how an inconsistent state can still occur.

To overcome this issue, we implemented a solution suggested by Redis on his official documentation [1]. We **associate a random value with each lock** when the session is locked. This random value is **generated at the start of function execution** and is retained until the function terminates. In order **to unlock the session, the function must provide the randomly generated value** that was associated with the lock at the beginning of its execution. **If the provided lock value matches the value associated with the lock, it indicates that no timeout has expired**, and the in-memory session can be updated and the lock released.

However, if the lock value provided by the function differs from the value associated with the lock (indicating that the lock expired and another function took over the session), or if there is no lock present on the session (indicating that the lock expired), the function cannot update the in-memory session or release the lock. In such cases, the function terminates without modifying the session and returns an HTTP 423 error.

By implementing this mechanism, we can **ensure that only the function that acquired the lock can update the in-memory session**, preventing inconsistencies caused by parallel execution of the same function or lock expiration.

5.4. Metadata

The session's metadata is a collection of data that describes various properties of the session. This metadata is always transferred along with the session itself and serves multiple purposes, such as keeping track of the original proprietary node and the Last Access Timestamp. They can also be used to implement a LRU mechanism when choosing which session to offload.

The following fields are included in the session's metadata:

- **Proprietary Location**: This field indicates the location that was initially contacted by the client and served as the session's original proprietary node. It is essential for session tracking and request forwarding, as explained in Section 5.6.4. This information is also necessary for onload requests, as the upper node must determine which local session to migrate to the lower node making the onload request. However, the migration should only occur along the path between the proprietary node and the root node within the same subtree.
- **Current Location**: This field denotes the current location of the session. It is used by the proprietary location to track the session's location and redirect clients accordingly. As detailed in Section 5.6, this field is updated when offloading or onloading operations occur.
- **Timestamp Creation**: This field stores the timestamp indicating when the session was created. The timestamp follows the ISO-8601 format [23].
- **Timestamp Last Access**: This field records the timestamp of the most recent access to the session. It is updated with each read or write operation on the session. The garbage collector component, described in Section 5.7, uses this field to determine how old is the session and make decisions regarding its deletion.
- **Requests' Id**: This field contains the unique identifiers of successfully processed requests associated with the session. As explained in Section 5.2.2, a new ID is added to this field each time a request is successfully processed.

All the fields are created when the session is first initialized on a node and are deleted when the session expires, along with the session's data, as described in the previous Section 5.7.

5.5. Network Structure

As showed in chapter 3, among the problems that we have to deal with when working with edge devices, we have to **manage a high number of devices distributed on a vast geographical surface**. In contrast, cloud technologies often relies on a few very well-defined nodes. Additionally, edge devices are spatially very close to the clients, while the cloud is practically always far from clients; the same applies if we use the network hops metric.

With the observations that we just made, we can come to the conclusion that **a network with such features can be represented with a tree structure**: the leafs are the multitude of edge devices, while the root is the cloud. The intermediate nodes represents edge devices with increasing computational power and eventually other server/cloud instances. This tree representation is supported also by the fact that if we remove the intermediate nodes and leave only the leafs and the root, we still have a tree structure and it is the structure that current networks adopt, so increasing the number of levels of the tree can be considered the natural next step.

The benefits that the tree structure provides is to **allow the developer to deploy functions easily**: this is achieved by laying an additional abstraction on top of the tree, which is a hierarchy divided by levels. After the developer defined the entire network, the deployment of functions is performed by specifying on which level the function has to be deployed. This functionality is explained with more details about how to use it in section A.6.

5.6. Offloading and Onloading

The core functionality of our framework is the automatic offloading and onloading of sessions across the global infrastructure. This mechanism maximizes resource utilization and ensures efficient utilization of available resources while maintaining transparency for both function developers and clients.

The key objectives of this feature are:

- **Move sessions up the hierarchy (offload) or down the hierarchy (onload)**: The framework dynamically determines the optimal placement of sessions, either moving them to higher-level nodes for offloading or to lower-level nodes for onloading, based on resource availability and workload distribution.

- **Guarantee data consistency during migration**: The offloading and onloading processes are designed to ensure that data remains consistent throughout the migration.
- **Transparent redirection for clients**: The migration process is transparent to clients using the functions. Clients are automatically redirected to the appropriate node where the session resides, without the need for explicit client intervention. This redirection enables smooth and uninterrupted usage of the framework's functions.

To achieve the mentioned objectives, the framework will be designed such that the following operations will be executed in order for both offload and onload functionalities:

1. Nodes express the need for an offload or the availability for an onload and thus negotiate to decide if a session will be migrated
2. Once the session is determined, both nodes create a local lock on the session to render it unavailable
3. The migration is performed
4. The locks are released and the requests are redirected to the appropriate node

5.6.1. Offloading

The offloading process involves moving one or more sessions from a lower-level node to an upper-level node in the global infrastructure. This process serves two purposes: it **frees up resources on lower-level nodes** (typically edge nodes with limited memory and computational power) and leverages the more powerful nodes at higher levels of the infrastructure, thereby potentially **improving function execution performance**. Although there may be a slight increase in latency due to the increased distance between the computing node and the client, the benefits outweigh this trade-off.

It is important to note that our framework focuses on moving sessions and redirecting clients to the appropriate node, while the horizontal scaling of function replicas within each node is delegated to Kubernetes itself. To minimize cold starts after an offload to an upper-level node, a minimum number of replicas (typically one) can be set on those nodes. Alternatively, to free up resources on underutilized upper nodes, the minimum replicas can be scaled down to zero, albeit with a longer cold start time in the event of an offload.

The actual implementation details of the migration process will be further explained in section 5.6.5.

The challenges associated with this migration process are similar to those discussed in the data consistency section (Section 5.2). To **ensure session integrity during migration**, we can **employ the same lock mechanism** described in Section 5.2.3 to the session that is being migrated. The lock needs to be applied to the session in both lower and upper node, for the following reasons:

- **Lock on the session in the lower node**: The lower node requires the lock to reject incoming requests for the session being offloaded, thereby ensuring that the session remains consistent during the migration process.
- **Lock on the session in the upper node**: The upper node needs the lock to protect the session in case of a past onload operation. Even after the session has been offloaded, some requests may still be received by the upper node that previously hosted the session. Note that the lock can be applied even if the node has no metadata and no data of the session, because the lock itself it is an independent piece of data, as showed in section 5.3.2.

Lastly, we need to address the scenario where the upper node may be overwhelmed and might reject the offloading request. The rejecting features is triggered with a policy that is explained in section 5.6.3.

- **Ignoring the issue**: One approach is to ignore the issue and force the upper node to accept the offloading request. In this case, the upper node would then trigger another offload to an even higher-level node.
- **Redirecting the offload**: A preferred solution is to virtually reject the offload and redirect the offloading request to a higher-level node in the hierarchy. This approach avoids the overhead of an additional migration (or multiple migrations) with minimal logic implementation. This is the approach that we implemented. Eventually, the Offload Request would be accepted, either by an upper level node, or by the cloud (which we assume has "unlimited" storage and computational power and always accept any offload).

The following figures provide an insight of the offloading process, including the internal mechanisms involved and the implications for clients accessing the offloaded sessions.

In Figure 5.2, we present an overview of the offloading process, specifically focusing on the scenario where a node rejects the offloading. The process begins with a Kubernetes cron invocation on the Offload Trigger pod, described in section 5.6.3, which is responsible for managing offloads. The Offload Trigger pod enumerates the sessions in its associated node (referred to as the Child Node) and selects a session for offloading based on its

internal policies. Once the session is chosen, the Offload Trigger pod invokes the ‘force-offload‘ function on the node, passing the session ID as a parameter to signal the need for offloading. The node locks the session for the reasons mentioned earlier and calls the ‘offload-session‘ function on the parent node, along with the session ID, to request offloading. In this scenario, the parent node rejects the offloading request due to being overwhelmed and forwards the offload request to the grandparent node with the same parameters. Finally, the grandparent node accepts the offload, locks the session, and initiates the migration process between the grandparent node and the child node. The details of the migration process are discussed in Section 5.6.5. Once the migration is complete, the session is unlocked, and the grandparent node is ready to handle requests redirected to it. Connections are closed accordingly, and the session is also unlocked on the child node, allowing clients to receive a redirect message to the grandparent node.

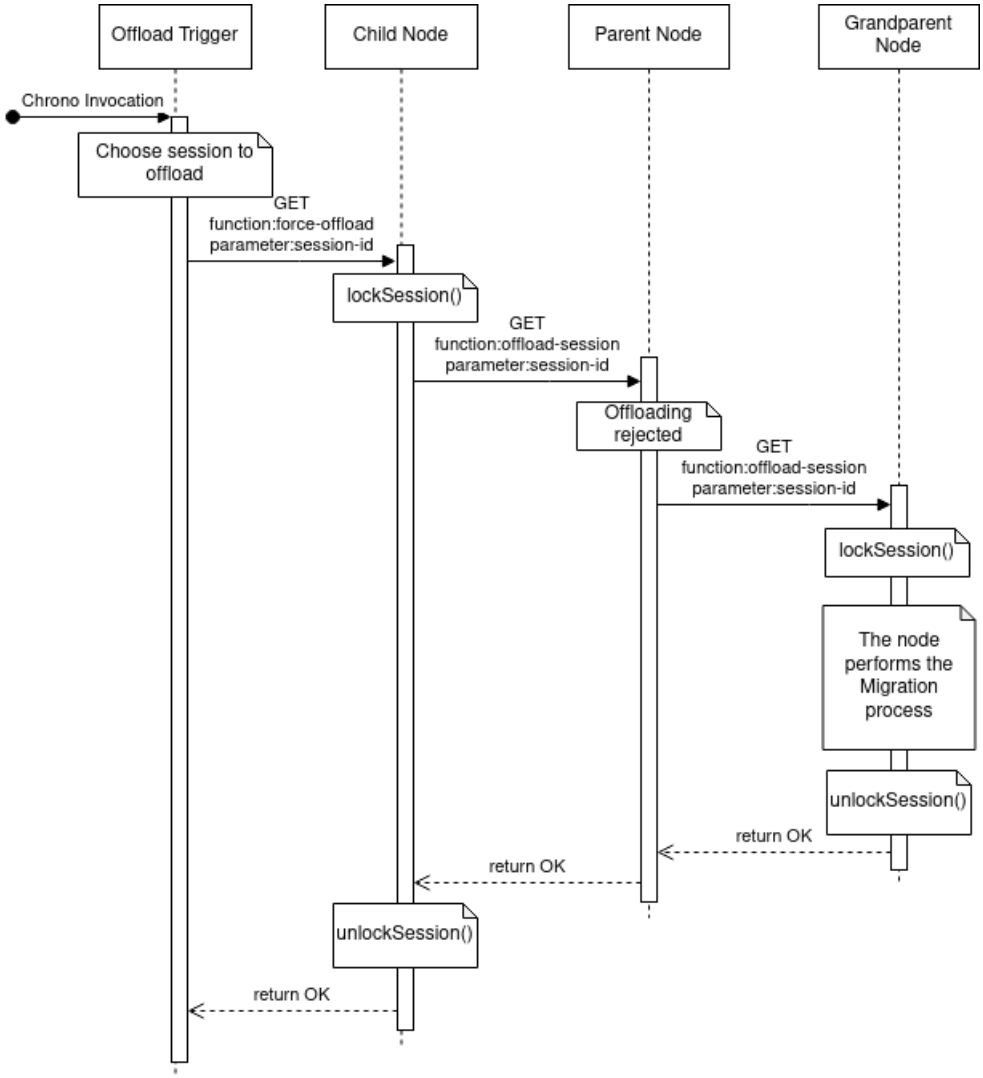


Figure 5.2: Multi level offloading.

Figure 5.3 illustrates the impact of offloading from a client's perspective. When an offload is performed from an edge node (the original proprietary of the session) to the cloud, the client initially requests access to the session. The edge node detects that the session has been offloaded and responds with a 307 message containing the new location of the session. The client can cache this information and subsequently communicate directly with the cloud for each subsequent request, where the function will be executed.

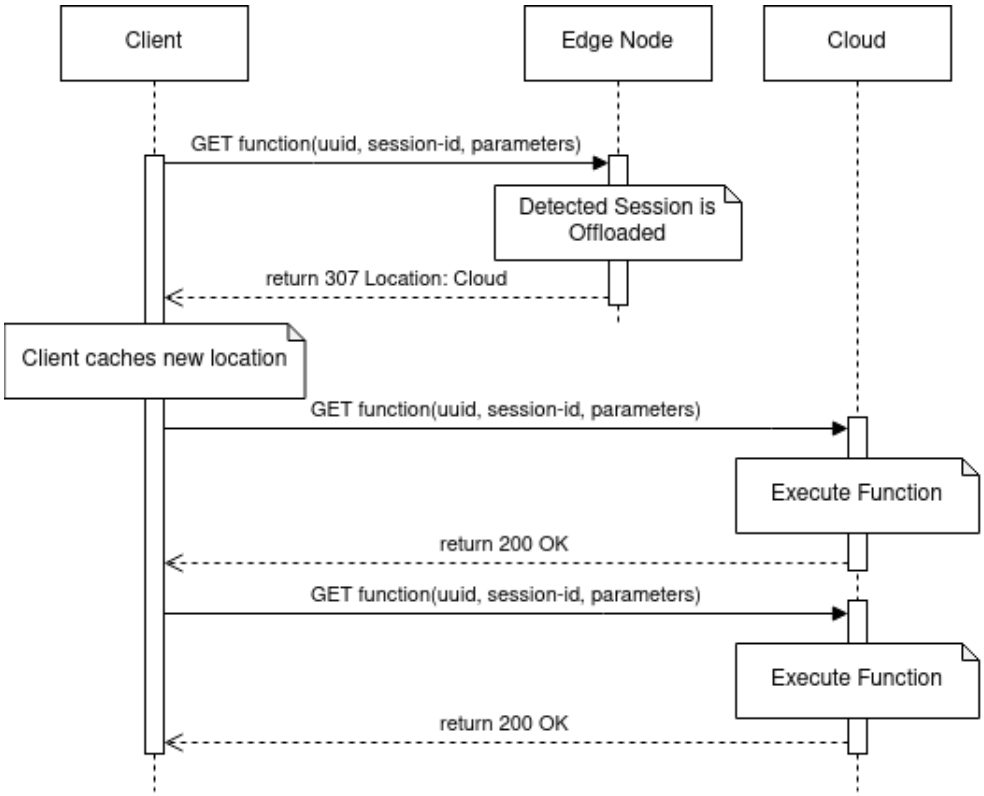


Figure 5.3: Point of view of a client after an offload.

5.6.2. Onloading

The onloading protocol closely resembles the offloading protocol, with the key difference being that sessions are migrated from an upper node to a node in the levels below. This means that while the onloading process is **triggered by the lower node**, the **decision of which session to onload is the responsibility of the upper node**.

To accommodate this difference in the onloading protocol, we have divided the protocol into three sequential steps:

1. Request of a session from the lower node to the upper node: The lower node sends a request to the upper node to obtain a session. The response from the upper node can either be the session's metadata (indicating that the session is locked by the upper node) or "no session found" if there are no candidate sessions available for onload.
2. Migration of the session's data: If a suitable session is found, the session's data is migrated from the upper node to the lower node. This ensures that the session is now available and accessible at the lower node for further processing.
3. Request to unlock the session: Once the session data has been successfully migrated, the lower node sends a request to the upper node to unlock the session. This allows the lower node to make modifications to the session and serve incoming requests from clients.

5.6.3. Triggering Offloading and Onloading

To trigger offloading or onloading operations, it is sufficient to call the internal HTTP APIs defined in our framework. The decision to provide these APIs through an HTTP interface is driven by the need to decouple the act of offloading/onloading a session from the decision of when to perform the operations. This decoupling brings two main benefits:

- **Enhanced modularity**: By decoupling the offloading/onloading mechanism, we achieve better resource management and utilization. This modular approach allows for more flexibility and adaptability in managing sessions and resources within the framework.
- **Custom triggering logic**: The HTTP interface enables developers to implement their own triggering logic based on specific requirements. This flexibility empowers developers to design offloading/onloading strategies that align with their application's unique needs and resource utilization patterns.

In our testing and experimentation, we have utilized an OpenFaaS function as the trigger for offloading and onloading operations. This function is called periodically and its task is to retrieve memory consumption statistics and initiate offloading/onloading based on the defined policies. We have two specific policies that regulate offloads and onloads:

- **Offload policy**: If the memory consumption exceeds the `offload_top_threshold`, one or more offloading operations are performed to alleviate resource usage. Offloads cease when the memory consumption falls below the `offload_bottom_threshold`.
- **Onload policy**: If the memory consumption falls below the `onload_threshold`, a single onload operation is triggered to balance resource distribution and optimize resource utilization.

By employing these policies and leveraging the HTTP APIs, our framework provides a flexible and configurable approach to trigger offloading and onloading operations based on real-time memory consumption statistics.

5.6.4. Session Tracking and Request Forwarding

In the context of session movement between nodes, a **mechanism to locate sessions is crucial** to enable users to access their desired sessions. Several solutions have been evaluated to address this requirement:

- **Centralized Sessions Locator**: The simplest approach involves creating a centralized component that is independent of any level or node in the hierarchy. This component is contacted whenever a node in the hierarchy or an external client needs to access a session whose location is unknown. The centralized component provides the current session location information, which can be cached until invalidated by a "session migrated to another node" message from the node that previously hosted the session. While this method is straightforward to develop, it has significant drawbacks. It does not scale well with an increasing number of clients and nodes, which is a concern in edge computing scenarios where the number of devices can grow exponentially. Additionally, the centralized component becomes a single point of failure, which is not desirable in a distributed system. Moreover, the approach can introduce extra network latency due to the distance between the locator and some nodes.
- **Distributed Sessions Locators**: Another approach builds upon the centralized locator by addressing its issues. Instead of having a single centralized locator, the hierarchy can be subdivided into zones, and each zone is assigned its own Sessions

Locator. This distributed approach eliminates the single point of failure, but it introduces the responsibility of accurately setting up locators and assigning nodes to them.

- **Proprietary Nodes**: An alternative modification to the previous approach involves removing the concept of locators entirely. Instead, each node becomes the "proprietary" of the sessions created by users on that node. With this approach, the **responsibility of session tracking is distributed among the nodes**, eliminating the need for a centralized component or locators. Whenever a session is migrated, the receiving node updates the session's metadata to indicate its new proprietary. This ensures that every node hosting the session knows its current proprietary. This **fine-grained approach removes the burden from developers and maintains a distributed architecture** with no global single point of failure.

In our framework, **we have adopted the proprietary nodes approach**, designating the first node contacted by the user as the "proprietary" of the sessions. This node becomes responsible for tracking all session migrations associated with that particular session.

5.6.5. Migration

With the offloading and onloading processes defined, we can now discuss the actual implementation of session migration between nodes.

A migration is the process of moving the data and metadata of a session between two nodes, regardless of the direction in the hierarchy.

In our framework, we have implemented a custom protocol based on HTTP message passing between nodes to facilitate the exchange of sessions. Alternatively, we could have utilized the Redis `MIGRATE` command integrated into Redis itself. However, we made the deliberate choice to separate the logic of the framework from the specific implementation of data storage. **This separation allows for better modularity and the potential for future migrations to other databases without impacting the core framework**.

To perform the session exchange, we need to execute two main operations:

1. **Copy data and metadata**: The first operation involves copying both the session's data and its associated metadata from the source node to the destination node. This ensures that all relevant information is transferred accurately during the migration process.

2. **Update proprietary information**: Once the data and metadata have been successfully transferred, the proprietary node of the session needs to be updated with the current location of the migrated session. This information is crucial for subsequent session accesses and ensures that the session is properly routed to the appropriate node.

By implementing this session migration protocol, **our framework enables seamless movement of sessions between nodes**, ensuring **consistent data transfer** and **updated session location** information.

Figure 5.4 illustrates the messages exchanged during the migration process, which occurs after both nodes involved have locked the session and before they unlock it. In this context, the "New Session Node" refers to the upper node in the offloading scenario and the lower node in the onloading scenario. Conversely, the "Current Session Node" refers to the lower node in the offloading scenario and the upper node in the onloading scenario. The "Proprietary Node" is independent of the scenarios and can be either the "New Session Node," the "Current Session Node," or another node.

1. The "New Session Node" updates the session's metadata to indicate that it will become the new current proprietary of the session.
2. The "New Session Node" initiates the migration process by calling the `migrate-session` function on the "Current Session Node" to request the session data identified by the session ID.
3. The "Current Session Node" updates its session's metadata to no longer be the current session node and deletes the session data after transmitting it to the "New Session Node."
4. Upon receiving the response, the "New Session Node" updates its session's data with the received session data.
5. Finally, the "New Session Node" sends an update message to the proprietary node by invoking the `update-session` function on the proprietary node. This update ensures that the proprietary node can modify the current location to redirect clients appropriately.

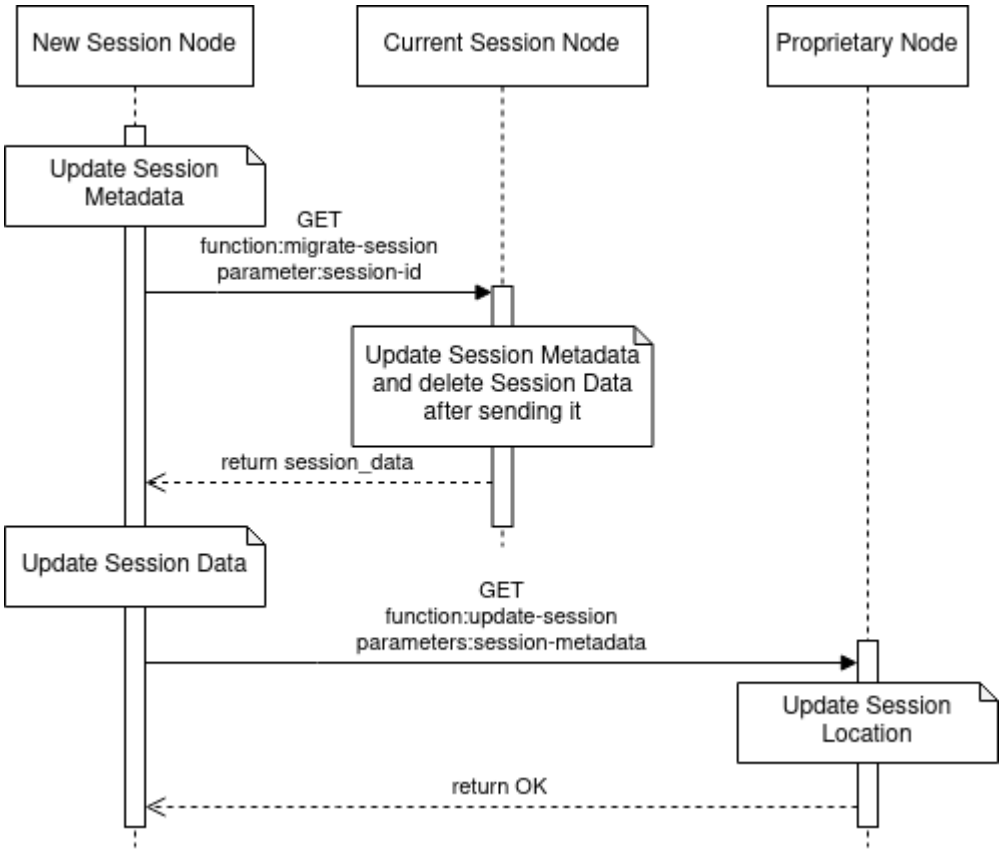


Figure 5.4: Migrate process.

Indeed, it is important to note that after the migration process is concluded, **some metadata is retained on the source node**. This metadata includes all session-related information, except for the request IDs. The purpose of retaining this metadata is to **ensure that clients can be properly redirected** to the proprietary node when they request the session and subsequently to the new current node.

5.7. Garbage Collection

Given that our framework is specifically designed for **devices with limited resources, including memory**, we have addressed the need to **free unused memory on these devices**. The unused memory include sessions' data and metadata. To solve this problem, we evaluated two approaches:

- **Explicit collection (calling a method)**: This approach would require the developer to call an API exposed by our framework. However, it places a significant responsibility on the developer to implement the logic for memory collection. This approach may not be ideal, as it adds complexity and burden to the developer.

- **Implicit collection (setting an expiration timeout)**: Instead of relying on the developer to implement **memory collection logic**, this approach requires the developer to set a timeout value in our framework settings. This **value represents the maximum duration of inactivity for a session**. In other words, if the difference between the "last access time" of a session and the current time exceeds the expiration time, the session is considered expired and can be eliminated. To check and delete expired sessions, our custom Garbage Collector is invoked periodically as a chrono-function.

By adopting the implicit collection approach, we **reduce the burden on the developer and enable automatic elimination of expired sessions** based on the specified timeout value. This approach is more suitable for our framework, as it helps optimize memory usage on resource-constrained devices.

In the **case of an offload**, where the session is migrated from the current node to a new node, the **responsibility for garbage collection lies with the current node that held the session**. If the session reaches its expiration time, the **current node is responsible for deleting the session** and its associated data. The current node should also **notify the original proprietary node** of the deletion of the session. This notification allows the proprietary node to safely delete the metadata associated with the session, **ensuring consistency and preventing any potential conflicts** or inconsistencies in the system.

6 | Prototype Implementation and Real-world Applications

This chapter presents the prototype of our framework, highlighting its key components and functionalities. The prototype demonstrates the practical implementation of our proposed stateful FaaS framework for edge computing scenarios.

6.1. Declaring the Infrastructure

To deploy the network hierarchy, we define the tree structure using the JavaScript Object Notation (JSON) format [8]. The hierarchy declaration consists of two main sections:

- **AreaTypesIdentifiers**: This section declares the names of each level in the tree, starting from the root. These identifiers are necessary for specifying levels in the deployer. For example, the first level can be named "country," the second level as "city," and the third level as "district." The lowest level always consists of edge nodes, while the upper level represents the root of the hierarchy, which is typically the cloud. Intermediate levels represent cloudlets or other nodes with computational power between the cloud and the edge. An example of area type identifiers declaration is shown in Listing 6.1

Listing 6.1 Area Types Identifiers declaration example

```
"areaTypesIdentifiers":[
    "country",
    "city",
    "district"
]
```

- **Hierarchy**: This section declares the actual hierarchy and is composed of a JSON object. Each node in the hierarchy is defined using the following structure:
  - *areaName*: The name of the area or node.

- *mainLocation*: An object containing the necessary information for nodes to connect with each other, including the OpenFaaS gateway, OpenFaaS password, Redis host, Redis port, and Redis password.
- *areas*: An array of nested infrastructure JSON objects representing the child nodes of the current node in the level below.

An example of the hierarchy declaration is shown in Listing 6.2.

Listing 6.2 Area declaration example

```
{
    "areaName":"<name>",
    "mainLocation":{
        "openfaas_gateway":"http://<ip>:<port>",
        "openfaas_password":"<password>",
        "redis_host":"<host>",
        "redis_port":6379,
        "redis_password":"<password>"
    },
    "areas":[
    ]
}
```

A complete example of a hierarchy with two levels and three nodes is shown in Listing 6.3.

Listing 6.3 Hierarchy example

```
{ "areaTypesIdentifiers":[
    "country",
    "city" ],
  "hierarchy":[ {
      "areaName":"italy",
      "mainLocation":{
        "openfaas_gateway":"http://172.18.0.5:31112",
        "openfaas_password":"lKVC4Htf4V",
        "redis_host":
          "my-openfaas-redis-master.openfaas-fn.svc.cluster.local",
        "redis_port":6379,
        "redis_password":"vdfverhyrt" },
      "areas":[ {
          "areaName":"milan",
          "mainLocation":{
            "openfaas_gateway":"http://172.18.0.2:31112",
            "openfaas_password":"brgfngtyhrg",
            "redis_host":
              "my-openfaas-redis-master.openfaas-fn.svc.cluster.local",
            "redis_port":6379,
            "redis_password":"mnjutyrbg" },
          "areas":[ ] }, {
          "areaName":"rome",
          "mainLocation":{
            "openfaas_gateway":"http://172.18.0.6:31112",
            "openfaas_password":"sdcvdbfe",
            "redis_host":
              "my-openfaas-redis-master.openfaas-fn.svc.cluster.local",
            "redis_port":6379,
            "redis_password":"wefsdcv" },
          "areas":[ ] } ] } ] }
```

6.2. Creating a Function

To allow developers to write functions for our framework, we have developed a **Java API**. After considering various approaches, we decided to create our own template for Java functions. While OpenFaaS natively supports a Java 11 template and allows developers to implement their own template if needed, we opted to develop our template for two main reasons.

First, we wanted to **upgrade the template from Java 11 to Java 17**, upgrade Gradle from version 6 to version 9, and update outdated dependencies marked as insecure by Gradle itself. Second, we needed to **include new exposed APIs for storage access**, allowing the addition and retrieval of data associated with the session invoked.

From a developer's perspective, our new template offers the same functionalities as the older one, **ensuring backward compatibility**. It is worth noting that our framework and its offloading capabilities can be utilized by developers using languages other than Java. In such cases, there would be no need to change the components already deployed in the cluster. Only the template would need to be reimplemented to expose the same APIs shown in Sections 6.2.2 (`com.openfaas.function.api.EdgeDB`),
6.2.3 (`com.openfaas.function.api.Offloadable`) and
6.2.4 (`com.openfaas.function.api.NonblockingOffloadable`).

6.2.1. Building a Function

To declare a function, we leverage the OpenFaaS `stack.yaml` configuration file feature. This powerful feature allows programmers to deploy an OpenFaaS function compatible with our framework. The essential metadata required include the function's name, programming language, function handler, and Docker image reference. An example of the basic version of the configuration file is shown in Listing 6.4.

Listing 6.4 `stack.yaml` example

```
version: 1.0
provider:
  name: openfaas
functions:
  <function-openfaas-name>:
    lang: java17
    handler: ./function-handler
    image: docker-reference/function-container-name:container-version
```

OpenFaaS also provides additional features worth mentioning in the context of our thesis:

- **Replicas**: It is possible to specify the minimum and maximum number of replicas for a function using labels. For example:

```
labels:
  com.openfaas.scale.min: 1
  com.openfaas.scale.max: 1
```

- Managing Resources: it is possible to specify the maximum quantity of resources that an instance of the function can use. In the following example, we are limiting the function to use a maximum of 40Mb of RAM and 1/10 of an Intel Hyperthread core:

```
limits:
  memory: 40Mi
  cpu: 100m
```

Additional information on declaring functions using the OpenFaaS standard can be found in the official documentation [35].

6.2.2. Stateful Support

Our API offers various methods to manipulate session data, which are located in the `EdgeAPI` class. These methods are shown in Listing 6.5.

Listing 6.5 Session manipulation methods

```
public static String get(String key);
public static void set(String key, String value);
public static List<String> getList(String key);
public static void setList(String key, List<String> list);
public static void delete(String key);
```

It is important to note that the methods for storing or deleting data in the session do not perform the operations immediately. As explained in Section 5.2.2, the data is written atomically to Redis after the user function has terminated. For non-blocking functions, which only have read permissions, changes made using write APIs are only local to the execution of the function for the current request.

The effects of these methods are as follows:

- `public static String get(String key)`: This method reads the value stored in the session with the given `key` identifier and returns it.

- `public static void set(String key, String value)`: This method stores the string value `value` in the session using the string `key` as the identifier.
- `public static List<String> getList(String key)`: This method reads the list stored in the session with the given `key` identifier and returns it.
- `public static void setList(String key, List<String> list)`: This method stores the contents of the `list` variable in the session using the string `key` as the identifier.
- `public static void delete(String key)`: This method deletes the data stored in Redis with the given `key` identifier.

For an example of using these methods in a use case, refer to Section 6.3.

Additionally, our API provides methods for developers to interact with other nodes in the hierarchy. These methods are available in the `EdgeInfrastructureUtils` class, as shown in Listing 6.6.

Listing 6.6 Node interaction methods from `EdgeInfrastructureUtils` class

```
public static String getParentHost(String locationChild);
public static String getParentLocationId(String locationChild);
public static String getGateway(String name);
public static List<String> getLocationsSubTree(String root);
public static List<String> getLocationsFromNodeToLevel(String nodeName,
↪  String level);
public static Infrastructure getInfrastructure();
```

These methods provide information without modifying the session state or node configuration. The methods have the following effects:

- `public static String getCurrentVirtualLocation()`
  Returns the proprietary location of the session being accessed during the current function execution.
- `public static String getParentHost(String locationChild)`
  Returns the value of the `openfaas_gateway` field for the parent node of the node with `locationChild` as the `areaName` value.
- `public static String getParentLocationId(String locationChild)`
  Returns the value of the `areaName` field for the parent node of the node with `locationChild` as the `areaName` value.

- `public static String getGateway(String name)`
  Returns the value of the `openfaas_gateway` field for the node with `name` as the `areaName` value.
- `public static List<String> getLocationsSubTree(String root)`
  Returns a list of strings containing the `areaName` fields of all nodes in the sub-tree of the hierarchy rooted at the specified `root` node. The root node is included in the list.
- `public static List<String> getLocationsFromNodeToLevel(String nodeName, String level)`
  Returns a list of strings containing the `areaName` fields of all nodes in the path from the specified `nodeName` node up to the specified `level`. Both the `nodeName` node and the node at the `level` are included in the list.
- `public static Infrastructure getInfrastructure()`
  Provides the declaration of the entire network hierarchy in case the previous helper functions are not sufficient for the developer's needs.

6.2.3. Blocking Function

To **create a function that can be offloaded to other nodes and has read and write permissions on sessions**, developers can implement the `Offloadable` interface and the `HandleOffload` method. An example of such a function is shown in Listing 6.7.

Listing 6.7 Offloadable function example

```
package com.openfaas.function;

import com.openfaas.function.api.EdgeDB;
import com.openfaas.function.api.Offloadable;
import com.openfaas.function.utils.EdgeInfrastructureUtils;
import com.openfaas.model.*;

public class Handler extends Offloadable {

    public IResponse HandleOffload(IRequest req) {
        Response res = new Response();
        res.setBody("Hello, world!");

        return res;
    }
}
```

This function can read and write session data using the APIs shown in Section 6.2.2.

For an example of using a blocking function in a use case, refer to Section 6.3.1.

6.2.4. Non-blocking Function

To **create a function that can be offloaded to other nodes, but is non-blocking and only has read permissions on sessions**, developers can implement the `NonBlockingOffloadable` interface and the `HandleNonBlockingOffload` method. The code for such a function is similar to that of a blocking function, but with the aforementioned modifications. An example is shown in Listing 6.8.

Listing 6.8 Non Blocking Offloadable function example

```
package com.openfaas.function;

import com.openfaas.function.api.EdgeDB;
import com.openfaas.function.api.NonBlockingOffloadable;
import com.openfaas.function.utils.EdgeInfrastructureUtils;
import com.openfaas.model.*;

public class Handler extends NonBlockingOffloadable {

    public IResponse HandleNonBlockingOffload(IRequest req) {
        Response res = new Response();
        res.setBody("Hello, world!");

        return res;
    }
}
```

This function can read session data using the APIs shown in Section 6.2.2.

For an example of using a non-blocking function in a use case, refer to Section 6.3.2.

6.3. Example of Use Case: Content Delivery Network (CDN)

In this use case, we demonstrate how our framework can be used to implement a potential Content Delivery Network (CDN) for delivering videos. The implementation requires defining procedures for uploading and downloading content from the CDN.

6.3.1. Uploading Content to the CDN

To upload content to the CDN, we can leverage the blocking function functionality of our framework, which provides write permissions. The function receives a `POST` request with the file in the request body and the file name in the URL query. It then writes the file to memory. The code for this function is shown in Listing 6.9.

6.3.2. Downloading Content from the CDN

To download content from the CDN, we can leverage the non-blocking function functionality of our framework. The function receives a `GET` request with the file name in the URL query and reads the file from memory. The code for this function is shown in Listing 6.10.

These functions demonstrate how our framework can be used to implement a simple CDN for content delivery. Developers can extend and customize these functions as needed to suit their specific requirements.

6.4. Example of Use Case: Speech-to-Text

Another use case that showcases the potential of our framework is speech-to-text processing. By leveraging an edge infrastructure distributed geographically, it is possible to **perform speech-to-text conversions to interact with AI systems like ChatGPT**, reducing the reliance on cloud resources and **minimizing the overall bandwidth usage towards the cloud**. This improvement can be observed in the graph shown in section 7.4.1, which compares the product of traffic and distance between scenarios with and without the edge infrastructure.

By offloading speech-to-text processing to edge nodes, the **speech data no longer needs to be sent to the cloud** for processing, reducing the overall bandwidth requirements and improving response times. This use case demonstrates how our framework can effectively support such distributed computing scenarios. In 6.11 we show how it can be implemented:

Listing 6.11 Speech-to-text Example

```
package com.openfaas.function;

import com.openfaas.function.api.EdgeDB;
import com.openfaas.function.api.Offloadable;
import com.openfaas.function.utils.Logger;
import com.openfaas.model.*;

import java.util.ArrayList;

public class Handler extends Offloadable {

    public IResponse HandleOffload(IRequest req) {
        Response res = new Response();
        if (req.getBody() == null) {
            res.setBody("{\"message\":\"400 Missing body in the
            ↪  request\", \"statusCode\":400}");
            res.setStatusCode(400);
            return res;
        }
        String fileWav = req.getBody();

        String prompt = Transcriber.speechToText(fileWav);
        List<String> history = EdgeDB.getList("history");
        if (history == null)
            history = new ArrayList<>();
        String aiResponse = ChatGPT.ask(prompt, history);
        history.add(prompt);
        EdgeDB.setList("history", history);

        res.setBody("{\"message\":" + aiResponse + ",
        ↪  \"statusCode\":200}");
        res.setStatusCode(200);
        return res;
    }
}
```

Listing 6.9 Upload Function CDN

```
package com.openfaas.function;

import com.openfaas.function.api.EdgeDB;
import com.openfaas.function.api.Offloadable;
import com.openfaas.model.IRequest;
import com.openfaas.model.IResponse;
import com.openfaas.model.Response;

public class Handler extends Offloadable {
    public IResponse HandleOffload(IRequest req) {
        Response res = new Response();

        String requestedFile = req.getQuery().get("file");
        if(requestedFile == null) {
            String message = "{\"statusCode\":\"400\",\"message\":\"400
             ↪  Missing query parameter 'file' in url\"}";
            System.out.println(message);
            res.setBody(message);
            res.setStatusCode(400);
            return res;
        }

        String body = req.getBody();
        if(body == null) {
            String message = "{\"statusCode\":\"400\",\"message\":\"400
             ↪  Missing body in http message\"}";
            System.out.println(message);
            res.setBody(message);
            res.setStatusCode(400);
            return res;
        }

        EdgeDB.set(requestedFile, body);
        System.out.println("Successfully uploaded file to EdgeDB class");
        res.setStatusCode(200);
        return res;
    }
}
```

Listing 6.10 Download Function CDN

```
package com.openfaas.function;

import com.openfaas.function.api.EdgeDB;
import com.openfaas.function.api.NonBlockingOffloadable;
import com.openfaas.model.IRequest;
import com.openfaas.model.IResponse;
import com.openfaas.model.Response;

public class Handler extends NonBlockingOffloadable {
    public IResponse HandleNonBlockingOffload(IRequest req) {
        Response res = new Response();
        res.setContentType("video/mp4");
        res.setHeader("Access-Control-Allow-Origin", "*");
        res.setHeader("Access-Control-Allow-Headers", "Origin,
        ↪  X-Requested-With, Content-Type, Accept, X-session,
        ↪  X-request-id");

        String requestedFile = req.getQuery().get("file");
        if(requestedFile == null) {
            String message = "{\"statusCode\":\"400\",\"message\":\"400
            ↪  Missing query parameter 'file' in url\"}";
            System.out.println(message);
            res.setBody(message);
            res.setStatusCode(400);
            return res;
        }

        String file = EdgeDB.get(requestedFile);
        if(file == null) {
            String message = "{\"statusCode\":\"404\",\"message\":\"404
            ↪  File '" + requestedFile + "' not found\"}";
            System.out.println(message);
            res.setBody(message);
            res.setStatusCode(404);
            return res;
        }

        System.out.println("Successfully returned file '" + requestedFile
        ↪  + "'");
        res.setStatusCode(200);
        res.setBody(file);
        return res;
    }
}
```

7 | Experimental Evaluation

This chapter presents a comprehensive analysis of the experimental evaluation conducted to assess the effectiveness, efficiency, and usability of our framework. We have achieved our original goals by conducting benchmarks to evaluate the performance and testing real use cases. The evaluation includes various metrics and scenarios to demonstrate the accomplishments and contributions of our framework in the field of Edge Computing.

7.1. Original Goals

As outlined in Section 1.4, our framework aimed to achieve three main goals:

- **Scalability**: Develop a deployment approach that can effectively manage an infrastructure consisting of hundreds to thousands of nodes, even in locations with limited or no IT staff, while maintaining centralized control and management.
- **Interoperability**: Ensure compatibility within a multi-vendor hardware and software environment, enabling integration with various components and technologies.
- **Consistency**: Develop an environment where applications can be developed once and deployed anywhere, allowing developers to write code without worrying about the underlying infrastructure.

To achieve these goals, we leveraged existing frameworks, particularly Kubernetes and OpenFaaS. These frameworks provide capabilities for resource management, deployment across different hardware and software through containerization, and automatic horizontal scaling. However, as these frameworks were originally designed for the cloud, we enhanced their capabilities and introduced new functionalities to support edge computing. These enhancements include offering a session API to enable stateful functions and managing vertical offloading. A complete list of functionalities is provided in Chapter 5.

In the following sections, we present benchmark results to demonstrate the benefits obtained through offloading logic and to verify that the performance achieved aligns with expectations. We also showcase real use cases implemented using our framework.

7.2. Technical Setup

The experimental evaluation was conducted using the following machines:

- Three laptops with the following specs:
  - CPU: Intel Core i7 @ 2.60 GHz
  - Memory: 16GB
  - Location: Milan (Italy)
- A Google Cloud instance was used with the following specs:
  - CPU: Intel Xeon @ 2.20 GHz (4 vCPUs)
  - Memory: 16GB
  - Location: `us-central1-a` (Iowa, USA)

To estimate the bandwidth between a node in Milan and the Google Cloud instance, we used the `iperf` command-line tool, which provided an estimate of 161 Mbits/sec in download and 163 Mbits/sec in upload.

The experiments were conducted in two settings:

- **LAN Experiment**: In this setting, a laptop was used to generate client requests, while the other two laptops served as an edge node and a mid-level node. The Google Cloud instance was not used.
- **Cloud Experiment**: In this setting, a laptop was used to generate client requests, another laptop served as an edge node, and the Google Cloud instance served as a cloud node. The height of the tree hierarchy was always 2.

Unless otherwise specified, the OpenFaaS functions were limited to 2 cores per function and 2GB of memory per function, emulating the resource constraints of typical edge hardware such as Raspberry Pi devices.

Python 3 scripts were used to perform the experiments, utilizing the `requests` library to generate client requests and the `matplotlib` library to generate graphs. The timing for performance measurements was taken immediately before the start and immediately after the end of each request.

In order to simulate different scenarios and evaluate the performance of our framework, various tests were conducted using different types of functions. Here is a breakdown of the function types used and their purposes in the testing:

- **Empty Function**: This function was used to benchmark the latencies and costs introduced by our framework without being impacted by the computational cost of the function itself. It simply returns an HTTP 200 response.
- **Non-blocking Function** (**CDN Download**): This function was used to test the performance of handling multiple parallel accesses to the same node within the same session. The function's purpose is to allow users to download a requested file and stream it if it's a video file. This test helps evaluate the efficiency of our framework in handling concurrent read-only requests.
- **Blocking Function**:
  - **CDN Upload**: This function was used to test the performance of blocking functions, which have both read and write permissions and lock the session during their execution. The purpose of the CDN Upload function is to save the content of the body received into a Redis instance and associate that content with the body contained in the Session header. This test allows us to assess the performance of our framework in handling write-intensive tasks and concurrent requests that require session locking.
  - **Speech to Text**: This function serves as an example of a computationally expensive function. It translates speech to text and stores the result in the history associated with the Session header. This test demonstrates the performance of our framework in executing complex computational tasks and managing the associated session data.

7.3. Performance Benchmarks

To assess the performance of our framework, several benchmarks were conducted:

- **Average Redirect Time**: This benchmark evaluated the time taken to redirect requests to the new location of an offloaded session.
- **Average Offloaded Response Time**: This benchmark measured the increase in latency when a request was offloaded to a node farther away from the client.
- **Average Offload Time**: This benchmark evaluated the time required to offload a session from one node to another.

7.3.1. Average Redirect Time

The benchmark for average redirect time was conducted to measure the duration it takes for requests to be redirected to the new location of an offloaded session. In this setup, a client initiates contact with a node where the session has recently been migrated. The client's request is then redirected to the new node, resulting in a redirect cost. By averaging the time of 1000 redirected requests, we assessed the redirect time. Notably, the redirect time is noticeable only after the first redirect, as subsequent requests can communicate directly with the new node without additional redirection.

The experiment was conducted in two scenarios: a local network setting and a cloud network setting. In the local network setting, the redirect time was measured between two nodes in a LAN. In the cloud network setting, the redirect time was measured between a local node and a cloud node using an Ethernet and fiber optic connection.

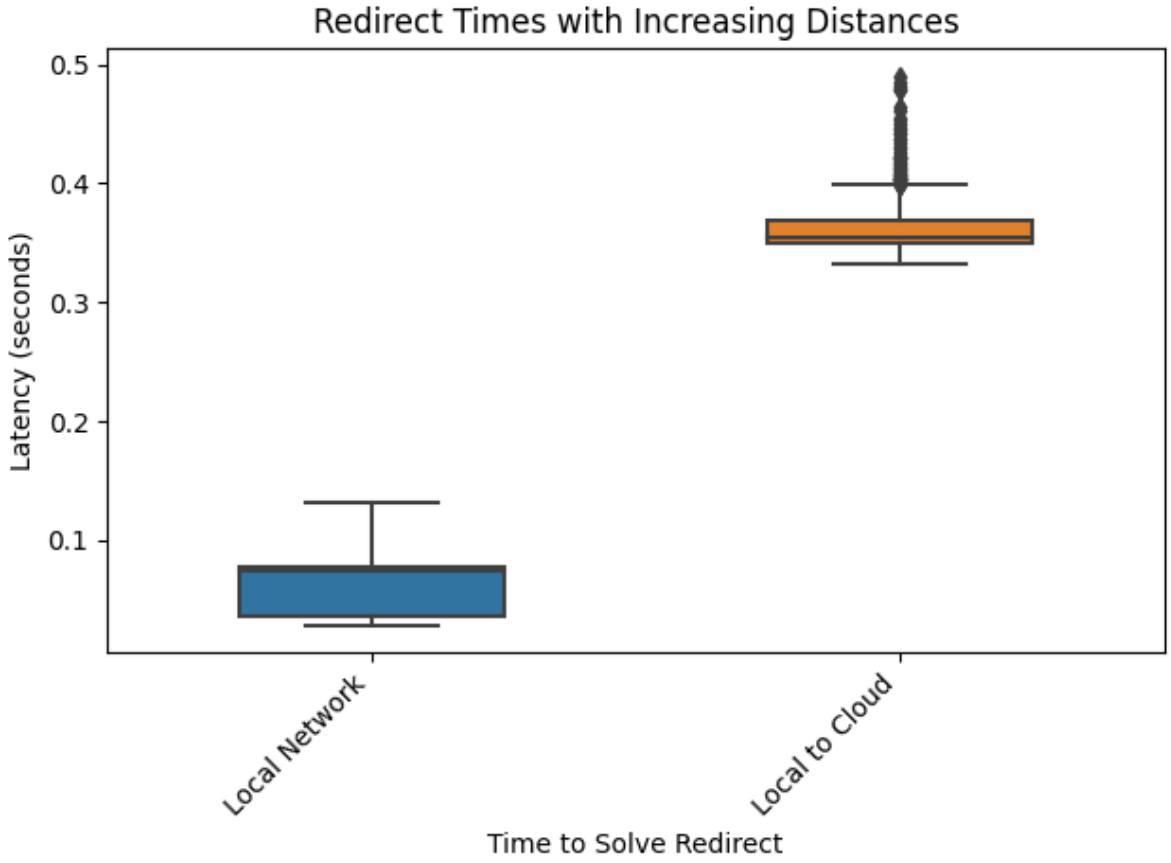


Figure 7.1: Average response time that a redirect takes to complete locally.

Figure 7.1 presents the results of the average redirect time benchmark. The "Local Network" experiment shows the redirect time between two nodes in a LAN, while the "Local

to Cloud" experiment shows the redirect time between a local node and a cloud node. The graphs provide insights into the time taken for redirects in different network settings.

7.3.2. Average Offloaded Response Time

This section provides a detailed examination of the average offloaded response time, a crucial determinant of user experience. The concept is explored using two comprehensive experiments.

In the first experiment, a client repetitively sends the same request for a duration of 10 seconds, sequentially; no requests rate is imposed to the client, when it receives a response a new requests is immediately issued. As soon at it receives a response, it send the consecutive request. Initially, the exchange of messages occurs between the client and the leaf node. However, after exactly 3 seconds into the experiment, an offload is triggered, causing the client to be redirected to the cloud node. The request contains a 1MB message to a blocking function, exemplified here by our Content Delivery Network (CDN) upload function. The function writes the content of the body of the received message into a session.

The response time at the onset of the experiment, during the initial message exchanges, is minimal. This period is then followed by several "503 Session not available" messages, caused by the unavailability of the session during the migration process. Subsequently, following the offload, an increase in latency, and consequently response time, is observed as the exchange of messages resumes. The duration for each request is noted from just before the request is sent to immediately after the response is received.

This experiment was conducted in an optimal environment, with only one client connecting to the network, thus leading to a low response time. In this context, offloading to the cloud would, only serve to increase the latency.

Figure 7.2 visually demonstrates the client's perspective on the impact of an offload.

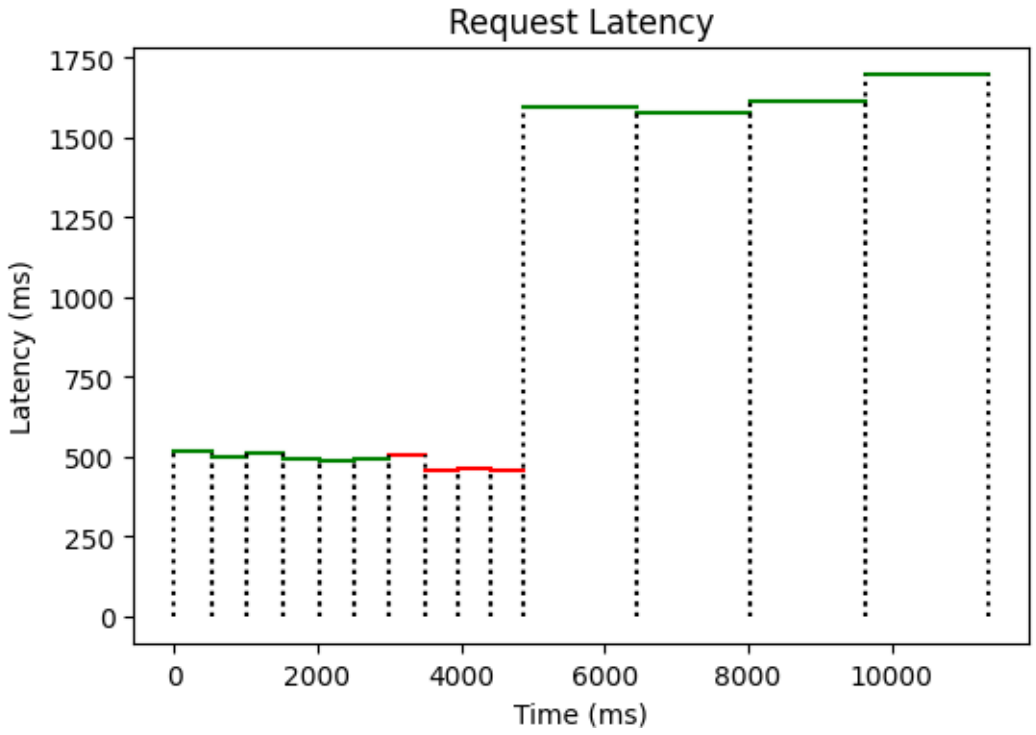


Figure 7.2: Timeline showing the impact of an offload from a client point of view

The second experiment demonstrates how latency might fluctuate in a scenario involving a varying number of concurrent client accesses to the same node. Similar to the previous experiment, a client sends a continuous cycle of identical requests to the same node; again, no rate of requests is imposed, but a new request is immediately issued after a response is received. However, throughout the experiment, the number of additional clients connecting concurrently to the same node dynamically increases (from 0 to 30, sequentially added between the 40th and 140th seconds).

The graph shows that the average response time is affected by this increased load, up until the point where the offload is triggered. After this, the session that the client is accessing is moved to an upper node (at second 140th the offload is triggered), with superior computational capabilities. This results in a lower, more stable response time, despite the latency being higher compared to the edge node. The graph effectively demonstrates the effectiveness of our framework in maintaining a low response time and adaptively responding to changing environmental conditions.

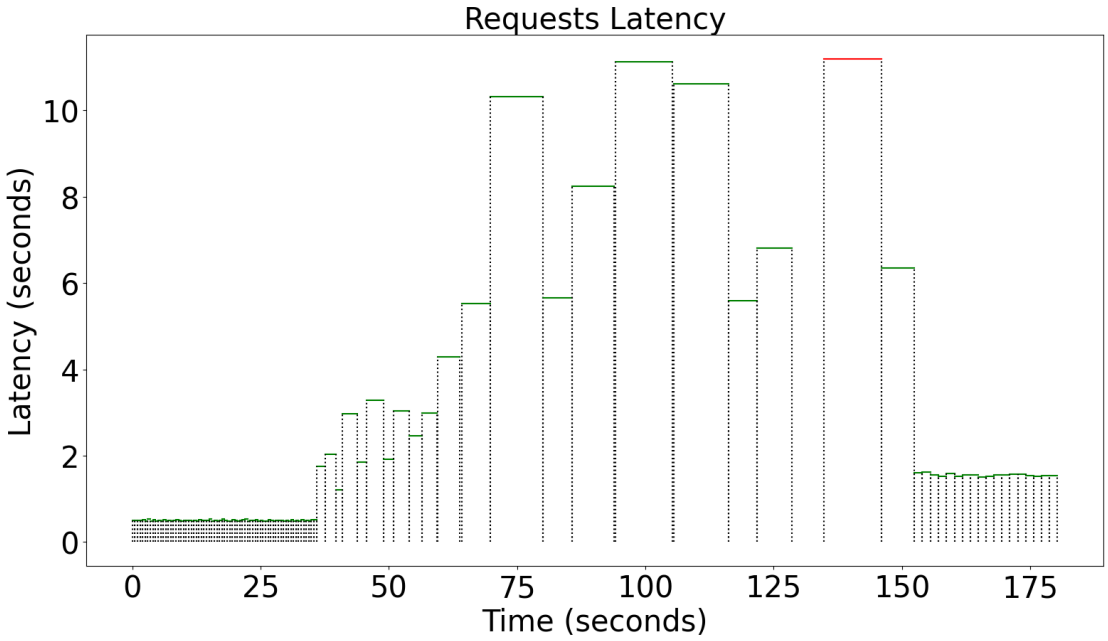


Figure 7.3: Experiment conducted to show the impact of an increasing number of clients

7.3.3. Average Offload Time

In this section, we investigate the average time it takes for a session to be offloaded from one node to another, an important metric referred to as the offload time benchmark. This benchmark's core goal is to assess the offloading process's speed and efficiency, two key factors that affect overall system performance.

An average of 200 offloads for each test size was used to estimate the offloading impact.

The offload time is captured from the standpoint of the internal component handling the data migration. The timer starts immediately before the offload request is sent and concludes immediately after receiving the "migration ended" response. It should be noted that the downtime experienced by the client as a result of offloading is visible only to functions with write permissions on sessions, as the session is write-locked during the migration process. Read-Only functions, such as "CDN download," do not experience downtime during migration, although the local session is deleted at the end of it. In this situation, if the function executes a read operation before termination, it returns an error, and the client is then redirected to the new node, resulting in minimal downtime.

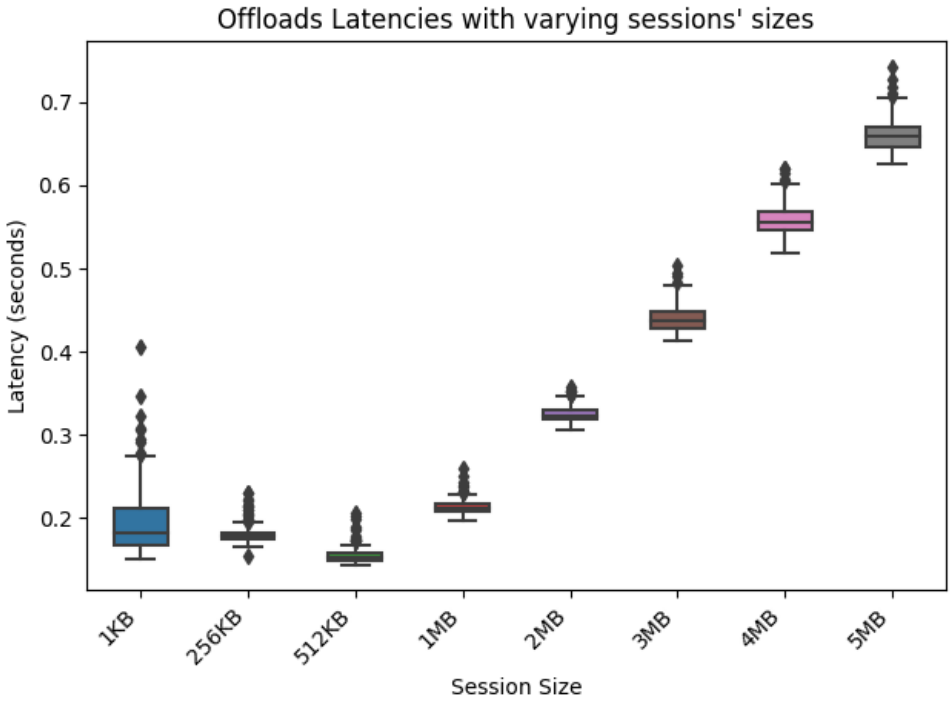


Figure 7.4: Experiment conducted to evaluate the average offloading time in a local setting

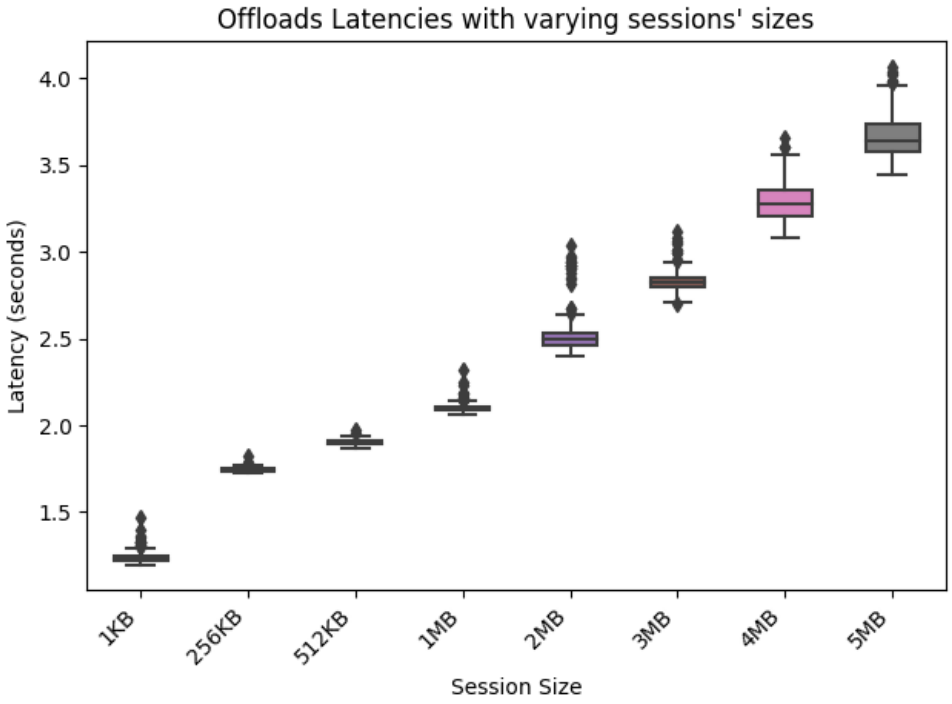


Figure 7.5: Experiment conducted to evaluate the average offloading time in a cloud setting

Figures 7.4 and 7.5 showcase box-and-whisker plots which illustrate the distribution of offloading times across three quartiles. The "box" encapsulates the interquartile range (IQR), stretching from the first quartile (25th percentile) to the third quartile (75th percentile), thereby illustrating the middle 50

These diagrams represent the temporal cost of migrating sessions of different sizes between nodes. We conducted this experiment in two settings: the first between two local nodes (Figure 7.4), and the second involving a local node and a cloud node (Figure 7.5).

From the local network experiment, we observe that the offloading process has a base time cost of about 200 milliseconds for sessions smaller than 1 Megabyte, with an additional 100 milliseconds required for every extra Megabyte. On the other hand, the experiment involving the local and the cloud node reveals a baseline time cost of approximately 2 seconds for sessions under 1 Megabyte, which then increases by an additional 400 milliseconds per Megabyte.

7.4. Performance in real use cases

The preceding section outlined the generic performance metrics of our framework during the offload process. In this section, we aim to demonstrate the framework's ability in handling multiple sessions and concurrent access in real-world scenarios.

Our framework is designed to circumvent the limitations inherent in edge computing. As the number of parallel connections rises, the average response time also increases proportionally. When the edge node's computational or bandwidth limits are reached, the average response time approaches infinity as clients' connections start to encounter indefinite waiting periods. Typically, this situation would be managed in the cloud by dynamically augmenting the resources dedicated to the task to accommodate the influx of connections. Our approach achieves a similar outcome by leveraging the resources available within our edge infrastructure. By offloading a portion of the session and evenly distributing connections across the available nodes, we can accommodate more clients while maintaining a low average response time. Clients subjected to offloading may experience heightened latency due to the increased distance to nodes. However, this loss is frequently balanced by the gain in computational power. By making judicious choices about which sessions to offload, we can prioritize the offloading of computationally intensive functions while striving to keep latency-sensitive sessions local to the leaf edge nodes. This approach allows us to strike an ideal balance between minimal edge latency and fast cloud computation.

Our framework can be utilized in various scenarios, many of which were proposed in Chapter 3.

While much of the existing literature on offloading functions to the edge focuses on methodologies, innovative approaches, and benchmarks to evaluate the feasibility of these methods, not many real-world use cases have been implemented or tested. This is primarily because the key objective of a FaaS framework on the edge is not to create new use cases or solve new problems. Instead, the goal is to extend the vast array of existing use cases currently deployed on the cloud, using existing FaaS frameworks, on the edge.

For this reason, our testing prioritizes benchmarking real scenarios with diverse settings, including different hardware capabilities and varying numbers of concurrent client access. By repeating the same tests across these varied scenarios, we aim to demonstrate the resilience and robustness of our framework in addressing the potential challenges an edge node may encounter. Rather than implementing a multitude of different functions or use cases—which would not provide significant insight into the capabilities of our framework—we have chosen to concentrate on implementing a select few use cases and testing them rigorously.

7.4.1. Reduction in total bandwidth used

This section aims to explain the gains in terms of bandwidth used that is possible to achieve by adopting our framework. For this demonstration, we monitored the bandwidth consumption in the Speech-To-Text use case described in section 6.4. The following assumptions were made for the purpose of this analysis:

- The edge node is located in Milan.
- The cloud is located in Iowa (USA) resulting in a geographical distance of approximately 7600 Km between the edge and the cloud.
- The average audio prompt it's a wav file of 20 seconds, that corresponds to 1.764 MB
- The typical history of text prompts amounts to a size of 0.5 MB.
- The edge node receives an average of 1000 requests per seconds

Our experiment calculated the bandwidth-distance product for both the hybrid and standalone solutions. The results are displayed in Figure 7.6.

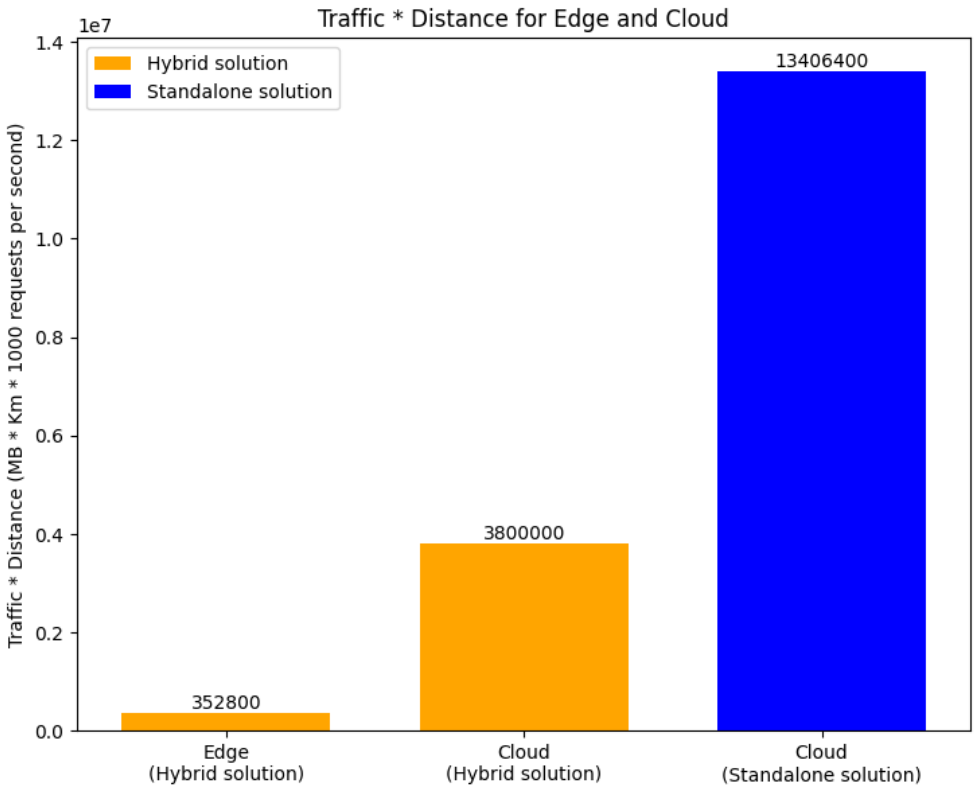


Figure 7.6: Traffic * Distance comparison between Hybrid Solution and Standalone Solution

The graph in Figure 7.6 vividly illustrates the gains previously discussed. In the Standalone Solution, only the cloud is utilized. Consequently, a substantial amount of bandwidth is consumed in transmitting the audio files to the cloud for conversion into text, subsequent storage with the history, and finally, use as input prompts. In contrast, the Hybrid Solution forwards the audio files to the edge, where the speech-to-text computation is performed. Only the prompt and the history are then transmitted to the cloud. This approach implies that the bandwidth consumed by the sizable audio files is significantly less compared to the Standalone Solution. Although the text prompts do require some bandwidth, the impact is noticeably smaller compared to the Standalone Solution.

In simple terms, our Hybrid Solution offers significant benefits in terms of bandwidth-distance product, a critical metric for assessing network efficiency. By reducing the amount of data transmitted over a long distance (from the edge to the cloud), we achieve lower total bandwidth usage, thereby improving the overall efficiency of the network.

7.4.2. Average Response Time for Concurrent Accesses

Our framework is designed to mitigate the constraints inherent in edge computing. As observed in Dennis Motta's thesis "Location-aware and stateful serverless computing on the edge" [30] (a recent example of a stateful FaaS framework implementation), the average response time naturally increases with the increase in parallel clients until it reaches a threshold. This threshold is contingent on the specific characteristics of the executed function and the technical specifications of the node under consideration. Once the node's resources are exhausted, the average response time tends to infinity, resulting in unserved or indefinitely delayed client requests. In such scenarios, cloud solutions can take advantage of their virtually limitless resources and autoscaling capabilities to dynamically reallocate resources for the task, ensuring continued service with minimal latency.

Through the next tests, we aim to demonstrate that our innovative approach to edge computing can effectively manage a real use case. By replacing horizontal scaling in the cloud with vertical offloading on the edge, we aim to optimize network resource utilization and maintain low client response times.

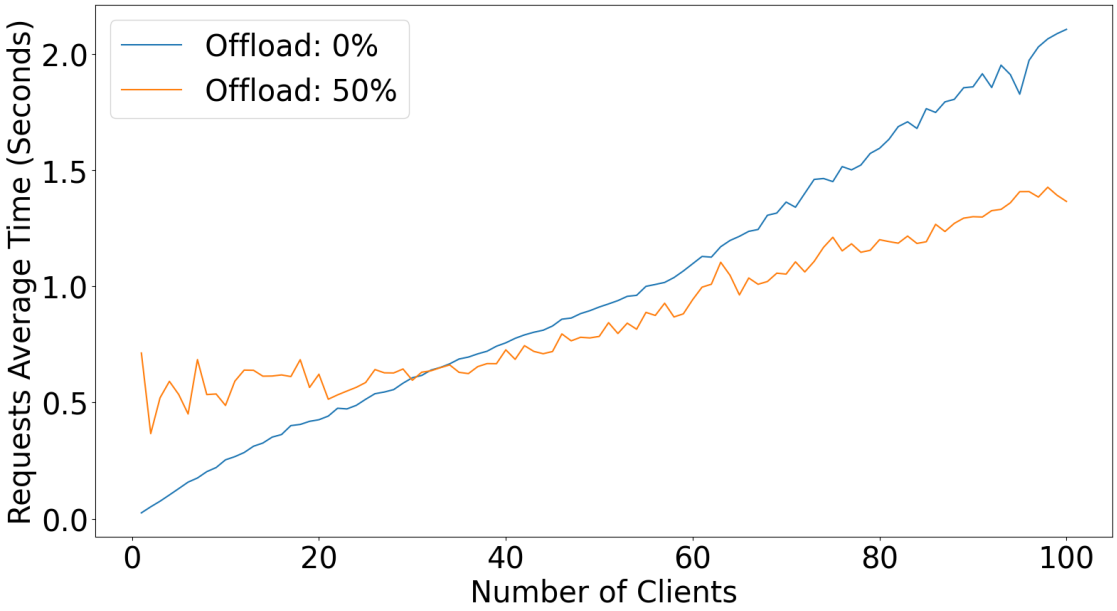


Figure 7.7: Experiments conducted to evaluate the average response time by increasing clients for the CDN Download function

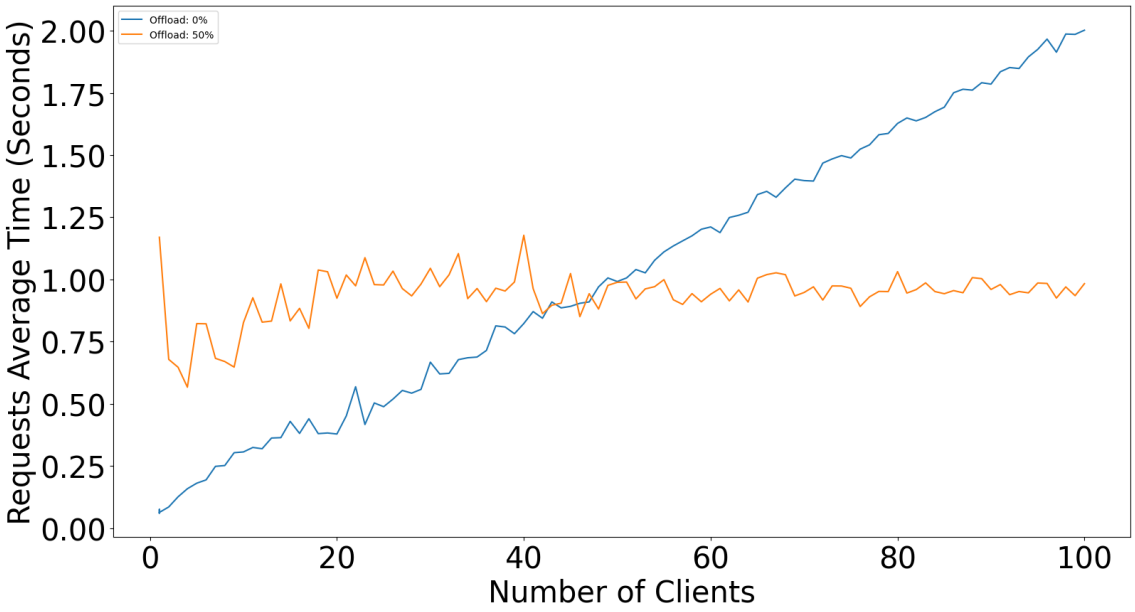


Figure 7.8: Experiments conducted to evaluate the average response time by increasing clients for the CDN Upload function

We conducted two experiments to generate these graphs, using the "CDN upload"7.8 function as a blocking function example and the "CDN download"7.7 function as a non-blocking function example. The leaf node was set up in a LAN and the parent node was hosted on a cloud server in America.

To conduct the tests, we used a local node and a cloud instance. The cloud instance maintained a fixed amount of 5 replicas for the function. The edge instance had a single fixed replica for the function, and it was limited to the use of a single CPU. We conducted multiple simulations with an increasing number of clients. Each client made 10 requests per simulation, and we averaged the response time per request obtained by each client across the 10 requests. The clients generated the requests one after another without imposing a rate on them, but just by issuing one request after another, similarly to how a stress test is created. In the "50% Offloads" experiment half the sessions are already offloaded before the start of it, so no offload is performed during the experiment.

In these graphs, the blue line represents the average response time on a single node, without any offloading, while increasing the number of parallel clients connected to it. The orange lines depict a simulation wherein half of the sessions are consistently offloaded to an upper node. As evident, by distributing connections evenly between two nodes (by offloading half of the sessions to an upper node), we can serve a larger number of clients

while maintaining a lower response time. Initially, the response time in case of an offload is higher due to increased latency. Hence, offloading sessions when there are fewer connected clients doesn't make sense, as the edge is perfectly capable of managing all the clients independently while maintaining low latencies. The intersection point marks the stage where offloading begins to be beneficial. As such, our framework only initiates offloading when computational limits are reached.

7.4.3. Efficient Choice of Offloading

The purpose of this experiment was to demonstrate that for a computationally intensive function, such as the speech-to-text function, the latency increase incurred by offloading is counterbalanced by the reduction in execution time.

To conduct this experiment, we utilized a local node and a cloud node. To simulate computational strain on the edge node, we constrained the function to effectively use half a CPU by setting both the OpenFaaS parameters `limits:cpu` and `requests:cpu` to 500m. Request timings were measured from the client's perspective, beginning just prior to sending the request and ending immediately after receiving the response. The experiment involved a client trying to access the speech-to-text function, initially located on the edge node, for a duration of 4 minutes. Exactly after two client requests, an offload was triggered, making the function on the edge node unavailable. Once the offload was complete, the client was redirected to the cloud.

As depicted in Figure 7.9, our framework can proficiently manage not only low latency functions typically deployed on the edge, but also efficiently handle the deployment of computationally demanding functions and dynamically offload them when necessary. This strategy aligns perfectly with the three primary objectives introduced earlier: our framework is capable of automatically managing the deployment of any type of function, in any environment, by placing functions on the leaves of a tree-like infrastructure consisting of edge, cloudlets, and cloud at the top. This arrangement strikes an optimal balance between the requirement for low latency achieved through edge computation and the need for rapid computation achieved through execution on upper nodes.

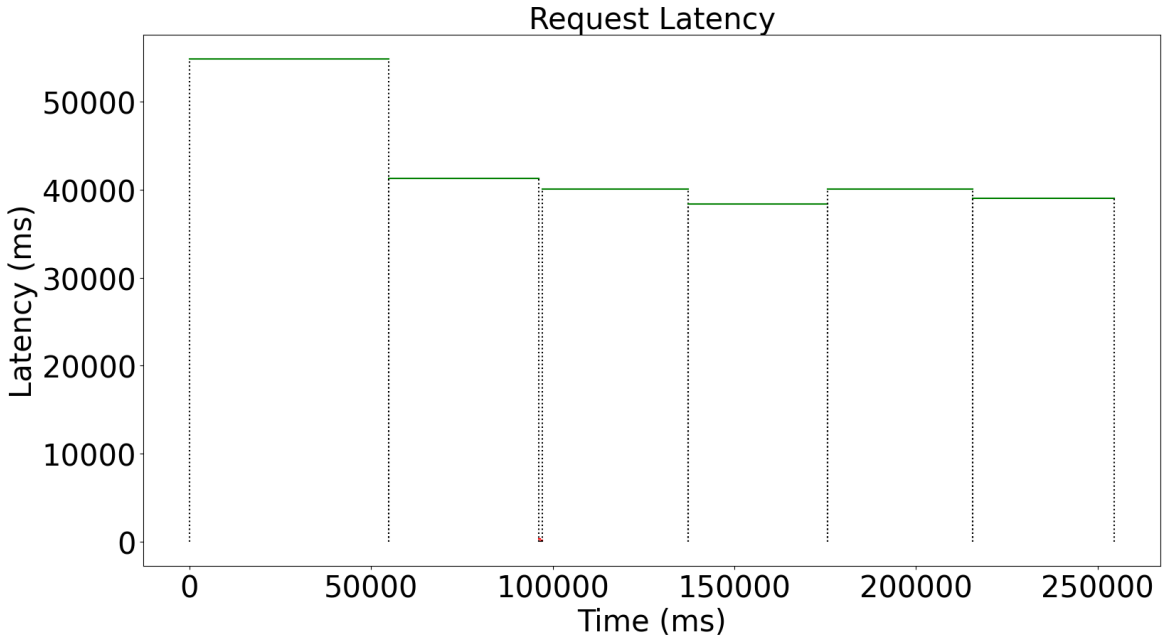


Figure 7.9: Experiment conducted to evaluate the impact of the offload on a computational intensive function

8 | Conclusion And Future Works

8.1. Conclusion

In this master thesis, we have proposed and developed a framework for location-aware and stateful serverless computing on the edge. The framework aims to address the challenges and limitations of edge computing by leveraging existing frameworks such as Kubernetes and OpenFaaS, while introducing new functionalities specific to edge environments.

Throughout the thesis, we have defined the objectives of the framework, which include scalability, interoperability, and consistency. We have achieved these goals by implementing a centralized approach to deploying and managing the infrastructure, ensuring compatibility within a multi-vendor environment, and creating an environment where applications can be developed once and deployed anywhere.

The experimental evaluation conducted on the framework has demonstrated its effectiveness, efficiency, and usability. We have conducted benchmarks to evaluate performance metrics such as redirect time, offloaded response time, and offload time. The results of these benchmarks have validated the capabilities of the framework in handling offloading, managing session data, and optimizing network resource utilization. Additionally, real use cases have been implemented and tested, showcasing the framework's ability to handle various scenarios and maintain low response times even with increasing client accesses.

The achievements of this research provide a significant contribution to the field of edge computing. By combining the advantages of edge and cloud computing, our framework offers an optimized solution for deploying serverless functions in edge environments. It allows for efficient resource utilization, reduces bandwidth usage, and ensures low-latency execution of functions.

8.2. Future Work

While the developed framework has achieved the proposed goals and demonstrated promising results, there are several avenues for future work to further enhance and expand the

research conducted in this master thesis. The following areas can be explored for future improvements:

1. **Flexible Infrastructure:** One potential area for improvement is to implement a more flexible infrastructure. Instead of a fixed tree-like structure, a generic graph of nodes could be employed. This would allow for more dynamic and adaptive deployment of functions based on varying edge conditions.

2. **Metrics-based Offloading:** Implementing various other metrics and logics to trigger offloading based on metrics taken from Prometheus could enhance the offloading decision-making process. This would enable the framework to dynamically adapt to changing environmental conditions and optimize resource allocation accordingly.

3. **Geographic Relocation Offload Mechanism:** Introducing a geographic relocation offload mechanism could ensure that the edge node is always the nearest to the current user. This would further improve latency and response times by minimizing network distance and reducing the need for frequent offloading between nodes.

4. **Exploring Alternative Programming Languages and Platforms:** While OpenFaaS has been utilized as the chosen platform in this research due to its lightweight nature, exploring alternative programming languages and platforms such as Fission could provide additional functionalities and address potential limitations, such as premium features in OpenFaaS that require payment.

5. **Distributed Load Balancer:** Implementing a distributed load balancer in front of the framework could act as a DNS for clients, ensuring redirection to the geographically nearest available node. This would improve load distribution and further optimize response times by minimizing network latency.

6. **Improved Custom Logic for Offloading and Onloading:** Enhancing the custom logic for selecting which sessions to offload and onload could enable better optimization based on specific requirements and scenarios. For instance, in latency-sensitive scenarios with multiple deployed functions, prioritizing the offloading of sessions associated with less critical functionalities while maintaining low-latency for more critical functions could be achieved.

7. **Optimized Kubernetes Deployment:** To reduce the impact of Kubernetes

   on computing performance, exploring the use of one Kubernetes cluster for each geographical position could be beneficial. This approach would allow for centralizing core Kubernetes components on a single node responsible for managing Kubernetes

logic, while using other nodes as worker nodes. This would result in reduced computation overhead on the worker nodes.

8. **Further Explorations:** Various other aspects can be explored to enhance the framework, including optimizing resource utilization, improving fault tolerance, enhancing security mechanisms, and exploring novel edge computing architectures.

By addressing these areas of future work, the framework developed in this master thesis can be further improved and extended, paving the way for more efficient and effective location-aware and stateful serverless computing on the edge.

A | Appendix A

All the code that was written until now, including both the framework and the tests, are available on GitHub [7].

The purpose of the next section is to show how to use it

A.1. Running the Prototype

We will now show all the prerequisites needed and a quick tutorial on how to setup everything:

A.2. Environment Prerequisites

The following applications and Command Line Interface programs are needed to setup the framework:

- Ubuntu 22.04 [40] or other compatible Linux distributions equipped with `bash` shell and `apt` package manager
- Docker Engine [14]
- Kubernetes CLI (`kubectl`) [27]
- K3d CLI and environment (`k3d`) [25]
- Helm Package Manager (`helm`) [21]
- OpenFaaS CLI (`faas-cli`) [16]

The following git repositories will be used:

- Edge Offloading Framework [7]
- Java17 OpenFaaS Template [17]

To ease the installation process and the usage of our framework we provide several scripts in the "Edge Offloading Framework" repository. The bulk of the scripts are contained in

the folder `Scripts`. Among them, these are the essential ones to start using our prototype:

- `first-setup.sh`: installs all the Environment Prerequisites listed before. It optionally adds auto-completion scripts to the environment for the aforementioned tools. (To use auto-completion run "sudo apt install bash-completion"). It also adds local aliases refering to the scripts in the `Scripts` folder; refer to `aliases.sh` to see all the aliases
- `install-registry.sh`: deploy a local docker registry hosted in K3d to allow much faster deploying time when developing and testing OpenFaaS functions locally. Our OpenFaas Yaml is already configured to pull images from the local registry instead of using the default online Docker Hub.
- `setup-clusters.sh`: setup a K3D cluster locally with a single node in it. This script will deploy a minimal K3D cluster, with a OpenFaas instance and a single-node Redis instance. The name of the node created inside the cluster will be `k3d-<my-cluster-name>`

A.3. Deploying a Node

The first step to create a node, is to create a cluster: as mentioned in section A.2, the framework provides a script to create a cluster with a node in it.

```
./setup-clusters.sh <cluster-name>
```

After the cluster is created, we can see the status of all the pods by running the following command:

```
kubectl get pods -A --context k3d-<cluster-name>
```

Once all the pods are in the `Ready` status, you can proceed with the creation of the other clusters.

A.4. Hierarchy Json

Once all desired cluster are created, you have to describe the network using the hierarchy JSON file to be able to deploy functions on them using our deployer. How to describe the hierarchy is explained in section 6.1. You can either deploy all of the clusters on the same computer, or on different machines. If you decide to use clusters local to the same computer, set the "openfaas_gateway" ip to the local ip of each cluster. You can obtain it by calling "kubectl get nodes -o jsonpath=".items[0].status.addresses[0].address".

Listing A.1 Compacted hierarchy example

```
{ "areaTypesIdentifiers":[
    "country",
    "city",
    "district" ],
  "hierarchy":[ {
      "areaName":"italy",
      "mainLocation":{ },
      "areas":[ {
          "areaName":"milan",
          "mainLocation":{ },
          "areas":[ ] }, {
          "areaName":"rome",
          "mainLocation":{ },
          "areas":[ ] } ] } ] }
```

As it is shown in example 6.3, the configuration file can grow quite large, even with just 3 nodes. To ease the writing of it, we integrated in the deployer an autofill feature for the fields of the `mainLocation` object that are often redundant and have default values. To use this feature, it is sufficient to not specify the field in the configuration file (this is usefull only for test purposes, as it autofills IPs of multiple local cluster deployed with k3d). An example of configuration file that uses the autofill feature is the snippet of code A.1.

The fields are automatically filled with the following policies:

- `openfaas_gateway`: this field is filled with the output of the following command:
  `kubectl get nodes -o jsonpath='{.items[0].status.addresses[0].address}'`
  `--context=<areaName>`
  To work properly the local `kubectl` environment needs to be able to describe the `areaName` node
- `openfaas_password`: this field is filled with the string `"autofilled-password"`
- `redis_host`: this field is filled with `my-openfaas-redis-master.openfaas-fn.svc.cluster.local`
- `redis_port`: this field is filled with 6379 which is the default Redis Port
- `redis_password`: this field is filled with `autofilled-password`

A.5. Deploying the Framework Components

After the network is described, it is possible to deploy the `session-offloading-manager`, which is the main component of our framework, which takes care of collecting expired sessions, performs offloading/onloading operations and related tasks. Follow these steps to deploy it:

1. Open a terminal and move to the folder `FaaS/openfaas-offloading-session`
2. Download the OpenFaaS Java17 template: execute the script `../template-java17.sh` to be sure to have the Java17 template installed
3. Build and push the docker image of `session-offloading-manager` to the registry by running the following command:
   `faas-cli up --skip-deploy --filter session-offloading-manager`
4. Finally, deploy the function using our deployer: for ease of use, you can execute the script

   `Faas/openfaas-offloading-sessions/deploy-session-offloading-manager.sh`

   The script requires two arguments: the first argument is the name of the root of the hierarchy, while the second argument is the level where you want to deploy the pod. You have to execute this script for all the levels of your hierarchy to have the framework deployed in all your nodes. More about of to use the deployer can be found in section A.6

A.6. Deploying a Function

The deployer is written in Java17, so a compatible environment must be set up to use it. The syntax to run it is the following:

```
java -jar $SCRIPTS_PATH/edge-deployer.jar <command> <deploy-options>
```

The command field is mandatory. A comprehensive list of commands can be found in section A.6.1, while the deployment options are listed in section A.6.2.

A.6.1. Deployer Usage

The deployer can be used to deploy functions more easily using the hierarchy configuration file. It is equipped with the following commands:

- `check-infrastructure`: checks that the provided JSON infrastructure file is properly formatted
- `display-infrastructure`: display the provided infrastructure file in a compacted way to highlight the structure of the hierarchy. Only area names are displayed
- `deploy`: deploy the specified function. More on this command in section A.6.2
- `help`: displays the text showed in listing A.2

Listing A.2 Deployer help message

```
Available commands:

check-infrastructure <path to infrastructure file>

display-infrastructure <path to infrastructure file>

deploy <function name> <path to infrastructure file>
    --inEvery <areaTypeIdentifier>: In which area type to deploy
        the function. If not specified the function is
        deployed to the lowest level.
    --inAreas <area>: The name of the areas in which to deploy
        the function. If not specified the function is
        deployed everywhere.
    --exceptIn <area>: The name of the areas in which to NOT deploy
        the function.
    --faas-cli <faas-cli deploy compatible parameter>: The argument
        of this parameter will directly passed to faas-cli deploy.
        See https://github.com/openfaas/faas-cli for more info.
        You can specify this parameter multiple times.

help
```

A.6.2. Deployer Options

The deployer command has several options to help the developer specify in which locations to deploy the function. The options are the following:

- `--inEvery`: this option has to be followed by an area-type identifier. It allows the developer to specify that the function has to be deployed only on the specified level. If this option is not present, the lowest level is used as default value
- `--inArea`: this option has to be followed by an area name. It allows the developer to specify that the function has to be deployed only in the specific sub-tree that has

the specified area name as root. If this option is not present, the root of the entire hierarchy is used as default value

- `--exceptIn`: this option has to be followed by an area name. It allows the developer to specify that the function must not be deployed in the specific sub-tree that has the specified area name as root. This option overrides the `--inArea` option

Additionally, since the deployer relies on the official OpenFaaS CLI to deploy functions, we added the option `--faas-cli "<command line argument>"` to allow the developer to specify additional arguments for the OpenFaaS CLI deployer.

A.7. Writing a Function

The process to write a function is very similar to the current OpenFaaS standard:

1. Download the function template by executing the following command

   ```
   faas-cli template pull
   ↪ https://github.com/Fabrizzz/openfaas-java17-template
   ```

2. Create a new function by executing

   ```
   faas-cli new function-name --lang java17
   ```

3. Edit the `stack.yaml` configuration file as explained in section 6.2.1 if needed

4. Include our `EdgeDb` framework as a standard library:
   just create a local maven repository to include in your project: you can either compile it yourself using
   mintinlinetextgradle publishToMavenLocal, or you can just copy the folder from `FaaS/template/java17/libs/com/openfaas/EdgeDb`
   contained in the framework repository to your function with the following path `<your-function-root>/libs/com/openfaas/EdgeDb`

5. Edit repositories in `build.gradle`: add the following line to the repositories paragraph:

   ```
   maven {
       name = 'localrepo'
       url = layout.buildDirectory.dir("<your-function-root>/libs")
   }
   ```

   or, in case you built the library yourself, just include your local maven repo:

```
repositories {
    mavenLocal()
}
```

6. Edit dependencies in `build.gradle`: add the following line to the dependencies paragraph:
   `implementation ('com.openfaas:EdgeDb:2.0.0')`

7. Now the developer is ready to write functions compatible with our framework. To write a blocking function that has read and write permissions on sessions, see section 6.2.3. To write a non-blocking function that has only read permissions on sessions, see section 6.2.4. To see how we enable stateful support, see section 6.2.2

A.7.1. Deploy a function

Once the function is defined, we can build it and push it to the docker repository. To achieve this we have to execute the following command:

```
faas-cli up --skip-deploy --filter <function-name>
```

The deploy of the function is skipped in this command since it will be handled by our custom deployer, which is explained in section A.6.

A.8. Calling a Function

To call a function, the procedure is identical with the way an OpenFaaS function would be called with some minor additions.

As stated in the OpenFaaS documentation [34], an invocation can be performed by means of sending an HTTP request to the proper URL. Here is an example of call by using the popular `curl` [11] command line tool:

```
curl -X POST http://<openfaas-gateway>/function/<function-name> -d
↪  '<request-body>'
```

If we would execute the previous line to call a function created with our framework, we would receive this response:

```
300 Header X-session is not present
```

This happens because, as the error message tells, an header was missing from the request. The following headers are mandatory when calling a function created with our framework:

- `X-session`: as the name suggests this field must contain the name of the session that we want the function to interact with. If the session does not exists, it will be created
- `X-session-request-id`: this field must contain a unique identifier as showed in section 5.2.1 of type UUIDv4. Of course if the request does not terminate correctly or if no response is received, a request with the same UUID can be sent to ensure the execution of it

The `curl` command should look like this to account for the mandatory headers:

```
curl -X POST http://<openfaas-gateway>/function/<function-name> -H
↪  'X-session: <my-session-name>' -H 'X-session-request-id:
↪  <generated-uuidv4>' -d '<request-body>'
```

Finally, it is important to also accept redirects which plays a key role in our framework to resolve offloads with the `-L` option:

```
curl -L -X POST http://<openfaas-gateway>/function/<function-name> -H
↪  'X-session: <my-session-name>' -H 'X-session-request-id:
↪  <generated-uuidv4>' -d '<request-body>'
```

With the last example we are using the framework as intended.

B | Appendix B

Here we show technical details of the internal implementation and internal API exposed to components of our framework:

B.1. Redis

Redis implements multiple logically independent databases. We used them to store different informations:

- Configuration Data (Redis DB 0): Contains configuration data local to the node for our framework to function stored in a key-value fashion.
  - **offloading:** Can contain **accept** or **reject** and refers to the current offloading policy as explained in Section 5.6.3.
  - **offload_top_threshold:** Must contain an integer that refers to the number of bytes that the Redis pod has to start to trigger an offload.
  - **offload_bottom_threshold:** Must contain an integer that refers to the number of bytes that the Redis pod has to store to stop the offloading of sessions.
  - **sessions_locks_expiration_time:** Must contain an integer that refers to the number of seconds that locks applied on sessions on this node have before expiring and unlocking the session.
  - **sessions_data_expiration_time:** Must contain an integer that refers to the number of seconds that a session on this node has before expiring and getting collected.
  - **onload_threshold:** Must contain an integer that refers to the number of bytes that the Redis pod has to store to start the onloading of sessions.
- Metadata of Each Session (Redis DB 1): Contains the metadata of each session, except the request IDs.

  - **TIMESTAMP_CREATION:** The timestamp of the creation of the session in ISO 8601 format.
  - **CURRENT_LOCATION:** The current location.
  - **PROPRIETARY_LOCATION:** The proprietary location.
  - **TIMESTAMP_LAST_ACCESS:** The timestamp of the last access to the session in ISO 8601 format.

- Data of Sessions (Redis DB 2): Contains the data of the sessions, which is accessible with the session ID. The data is stored with the keys provided by the users' functions.
- Locks of Sessions (Redis DB 3): Contains the locks of the sessions. The key is the session ID, and the value is the randomly generated value.
- Request IDs of Sessions (Redis DB 4): Contains the request IDs of the sessions. The key is the session ID, and the value is the set containing the request IDs associated with the session ID.

B.2. Session Offloading Manager Component

The final component that we have to describe is the Session Offloading Manager. This is a core component that deals with the following tasks:

- Migration of the sessions including offloads procedures and onloads procedures
- Triggers to start Offloads and Onloads process
- Garbage Collection
- Setting of internal configuration and debug functions

This component is usable through its HTTP API. It is built from an OpenFaaS function, so to access it, the URL has to be in the following format:

```
http://[gateway:port]/function/[function_name]?[function_query_parameters]
```

The HTTP API are showed in the following list:

- `session-offloading-manager?command=force-offload`
  This function expect the `X-forced-session` HTTP header to be present in the request, otherwise a 400 error is thrown. It is used to start the offloading of the specified session. It is called by `offload-trigger`.

- `session-offloading-manager?command=force-onload`
  This function does not require any argument. It is used to start the onload of one session from the upper nodes in the hierarchy. It returns a 400 error code if there was no session available to onload on the node, or 200 if a session was onloaded. It is called by `offload-trigger`.

- `session-offloading-manager?command=set-offload-status&status=<status>`
  This function requires a query parameter `status` with the possible values of `accept` or `reject`. It can be used to change the internal configuration of the offloading policy of a node. It returns a 400 error if the request is missing the query parameter or the value in the query parameter is unexpected, while on success it returns 200.

- `session-offloading-manager?command=offload-session`
  This function requires the HTTP header `X-session-token` to be present in the request, containing a JSON representation of the session metadata. The JSON is an object with the following fields: `session`, `proprietaryLocation`, `currentLocation`, `timestampCreation` and `timestampLastAccess`. This function is called by `force-offload` and perform the negotiation to accept the offload request and thus start the migration process or to reject the offload request and thus forward the request to the upper node. It always returns 200.

- `session-offloading-manager?command=onload-session?action=<action>`
  This function requires an HTTP header `X-onload-location` to specify which location is requesting the onload and a query parameter **action** that can assume the values of `get-session` or `release-session`. Additionally, if the action is `release-session`, two additional query parameters are necessary, namely `session` and `random-value`. If any parameter is missing or a value is unexpected, the function returns a 400 error. The action `get-session` is used in the first step of the onloading protocol, before the migration, to negotiate the session. The action `release-session` is used in the last step of the onloading protocol, after the migration, to release the lock on the onloaded session and allow redirects.

- `session-offloading-manager-update-session?command=update-session`
  This function requires the HTTP header `X-session-token` in the request. It is called on the proprietary nodes to update the location metadata after a session has been migrated. If the request is malformed, the session does not exist on the node or the location is wrong, a 400 error is returned, otherwise a 200 status code is returned.

- `session-offloading-manager-migrate-session?command=migrate-session`

This function requires two query parameters: `session` and `data-type`. It is used by nodes to actually move sessions' data and metadata (requests ids) from the receiving node, to the requesting node. The `data-type` parameter can be `sessionData` to transport data or `requestIds` to transport the sessions' ids. If the request is malformed it returns 400, otherwise it returns 200.

- `session-offloading-manager?command=offload-trigger`
  This function takes not parameters in input. It starts the offloading and onloading processes as shown in section 5.6.3. It always returns 200 as status code, while in the body of the response an integer: `-1` if a session was onloaded, `0` if no session was moved or a positive integer if one or more session were offloaded. The integer represents the number of bytes freed by offloading the sessions.

- `session-offloading-manager-garbage-collector?command=garbage-collector`
  This function requires a query parameter `deletePolicy`. If the value specified is `expiration`, then the function starts the garbage collection process by deleting all the session that are expired, checking their `last_access_timestamp` field. If the policy specified is `forced`, then another query is required, which is `sessionId`. This last action is used to directly remove all the data and metadata of a session without waiting for the expiration. If the requests are malformed a 400 error is thrown, otherwise a 200 status code is returned along with the number of sessions delete in the body of the response.

- `session-offloading-manager?command=test-function`
  This function is an helper function to help the developers test their frameworks. It requires two query parameters, `type` to specify what kind of data to test and `value` which is the session-id of the session to test. The parameter `type` can assume the values `configuration` or `sessionMetadata` or `sessionMetadataLocations` or `sessionData` or `session`. It returns a 400 error if the request is malformed, otherwise a 200 status code along with the requested data in the body of the response.

It is worth noting that of all these APIs a developer can directly call `force-offload` and `force-onload` to implement its own offloading and onloading policies.

Bibliography

[1] Distributed Locks with Redis, July 2023. URL https://redis.io/docs/manual/patterns/distributed-locks. [Online; accessed 3. Jul. 2023].

[2] Akamai Technologies. Akamai EdgeWorkers. https://www.akamai.com/us/en/products/web-performance/edgeworkers, 2023. Accessed: July 2023.

[3] Amazon Web Services. AWS Lambda@Edge. https://aws.amazon.com/lambda/edge/, 2023. Accessed: July 2023.

[4] Apache Software Foundation. Apache OpenWhisk. https://openwhisk.apache.org/, 2023. Accessed: July 2023.

[5] appfleet. Appfleet. https://appfleet.com/, 2023. Accessed: July 2023.

[6] O. Authors. Introduction - OpenFaaS, June 2023. URL https://docs.openfaas.com. [Online; accessed 29. Jun. 2023].

[7] L. Barilani and G. Fabrizio Bonaccorsi. Edge Offloading Framework, June 2023. URL https://github.com/leonardobarilani/edge-computing-thesis. [Online; accessed 15. Jun. 2023].

[8] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, Dec. 2017. URL https://www.rfc-editor.org/info/rfc8259.

[9] Cloudflare. Cloudflare Durable Objects. https://developers.cloudflare.com/workers/learning/how-durable-objects-works, 2023. Accessed: July 2023.

[10] Cloudflare. Cloudflare Workers. https://developers.cloudflare.com/workers, 2023. Accessed: July 2023.

[11] curl. curl, June 2023. URL https://curl.se. [Online; accessed 15. Jun. 2023].

[12] X. Dai, Z. Yu, X. Ma, and X. Wang. Offloading virtual reality rendering tasks in cloud-assisted mobile augmented reality applications. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[13] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, DEBS '21, page 31–42, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385558. doi: 10.1145/3465480.3466920. URL https://doi.org/10.1145/3465480.3466920.

[14] docker. Docker Engine 24.0 release notes, June 2023. URL https://docs.docker.com/engine/release-notes/24.0. [Online; accessed 14. Jun. 2023].

[15] B. Ellerby. Why Serverless will enable the Edge Computing Revolution. *Medium*, Apr. 2021. URL https://medium.com/serverless-transformation/why-serverless-will-enable-the-edge-computing-revolution-4f52f3f8a7b0.

[16] faascli. Installation - OpenFaaS, June 2023. URL https://docs.openfaas.com/cli/install. [Online; accessed 15. Jun. 2023].

[17] G. Fabrizio Bonaccorsi. Java17 OpenFaaS Template, June 2023. URL https://github.com/Fabrizzz/openfaas-java17-template. [Online; accessed 15. Jun. 2023].

[18] Fission. Fission. https://fission.io/, 2023. Accessed: July 2023.

[19] M. Franke, T. Rausch, T. Weis, and B. Mitschang. Stateful edge computing: Offloading and management of stateful applications. *IEEE Transactions on Network and Service Management*, 15(4):1436–1449, 2018.

[20] Y. Guo, J. Yang, S. Wang, and H. Wang. Smart traffic management with edge computing. In *2021 IEEE 2nd International Conference on Control, Automation and Artificial Intelligence (ICCAAI)*, pages 333–338. IEEE, 2021.

[21] helm. Installing Helm, June 2023. URL https://helm.sh/docs/intro/install. [Online; accessed 15. Jun. 2023].

[22] M. Hines. What does the future hold for edge computing?, 2020. URL https://builtin.com/cloud-computing/future-edge-computing.

[23] ISO Central Secretary. Date and time — representations for information interchange — part 1: Basic rules. ISO 8601-1:2019, 2019. URL https://www.iso.org/standard/70907.html.

[24] P. S. Junior, D. Miorandi, and G. Pierre. Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes. In *2022 IEEE 6th Interna-*

tional Conference on Fog and Edge Computing (ICFEC), pages 26–33, 2022. doi: 10.1109/ICFEC54809.2022.00011.

[25] k3d. k3d, May 2023. URL `https://k3d.io/v5.4.7`. [Online; accessed 15. Jun. 2023].

[26] V. Karagiannis and S. Schulte. Comparison of alternative architectures in fog computing. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 19–28, 2020. doi: 10.1109/ICFEC50348.2020.00010.

[27] kubectl. Install and Set Up kubectl on Linux, May 2023. URL `https://kubernetes.io/docs/tasks/tools/install-kubectl-linux`. [Online; accessed 15. Jun. 2023].

[28] P. J. Leach, R. Salz, and M. H. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005. URL `https://www.rfc-editor.org/info/rfc4122`.

[29] I. Lujic, V. De Maio, and I. Brandic. Efficient edge storage management based on near real-time forecasts. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 21–30, 2017. doi: 10.1109/ICFEC.2017.9.

[30] D. Motta. Location-aware and stateful serverless computing on the edge, Dec. 2021. URL `http://hdl.handle.net/10589/182795`. [Online; accessed 2. Jul. 2023].

[31] I. Murturi, C. Avasalcai, C. Tsigkanos, and S. Dustdar. Edge-to-edge resource discovery using metadata replication. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6, 2019. doi: 10.1109/CFEC.2019.8733149.

[32] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21:64–71, 01 2017. doi: 10.1109/MIC.2017.2911430.

[33] D. T. Nguyen, N.-T. Pham, and D.-K. Hwang. Intelligent edge computing for internet of things: A survey. *Journal of Network and Computer Applications*, 135:27–42, 2019.

[34] OpenFaaS. Invocations - OpenFaaS, June 2023. URL `https://docs.openfaas.com/architecture/invocations`. [Online; accessed 15. Jun. 2023].

[35] openfaasyaml. Openfaas yaml file reference, 2023. URL `https://docs.openfaas.com/reference/yaml/`.

[36] C. Puliafito, C. Cicconetti, M. Conti, E. Mingozzi, and A. Passarella. Stateful func-

tion as a service at the edge. *Computer*, 55(9):54–64, 2022. doi: 10.1109/MC.2021.3138690.

[37] I. C. Rau, R. Goyal, and H. Singh. Edge computing for the internet of things in healthcare: A comprehensive survey. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*, pages 189–196. IEEE, 2019.

[38] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 17–25, May 2022. doi: 10.1109/ICFEC54809.2022.00010.

[39] J. Song, Y. Wang, C. Fan, Y. Niu, and N. N. Xiong. Stateful event-driven offloading in edge computing for the internet of things. *IEEE Transactions on Network Science and Engineering*, 2021.

[40] ubuntu. Jammy Jellyfish Release Notes, Apr. 2023. URL `https://discourse.ubuntu.com/t/jammy-jellyfish-release-notes/24668/1`. [Online; accessed 14. Jun. 2023].

[41] R. van der Meulen. What edge computing means for infrastructure and operations leaders, 2018. URL `https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders`.

[42] K. Varda. Durable objects: Easy, fast, correct, 2021. URL `https://blog.cloudflare.com/durable-objects-easy-fast-correct-choose-three/`.

[43] K. Varda. Workers Durable Objects Beta: A New Approach to Stateful Serverless. *Cloudflare Blog*, Feb. 2023. URL `https://blog.cloudflare.com/introducing-workers-durable-objects`.

[44] Z. Wen, Y. Shi, S. Guo, and C. Wang. Data analytics and decision making in smart grid. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 145–150. IEEE, 2018.

[45] W. Yang, F. Li, H. Jiang, and Q. Zhang. Edge computing based on internet of things: A survey. *IEEE Access*, 7:114144–114156, 2019.

[46] Y. Zhang, S. Wang, H. Zhang, Z. Zhao, Z. Cui, and W. Zhang. Speech recognition and understanding at the edge. In *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pages 1–5. IEEE, 2020.

List of source codes

List of Figures

List of Symbols

API	Application Programming Interface
BGP	Border Gateway Protocol
CDN	Content Delivery Network
CLI	Command Line Interface
FaaS	Function-as-a-Service
GB	Gigabyte
GPS	Global Positioning System
HTTP	HyperText Transfer Protocol
ICFEC	IEEE International Conference on Fog and Edge Computing
ID	Identifier
IDE	Integrated Development Environment
IP	Internet Protocol
IoT	Internet-of-Things
JSON	JavaScript Object Notation
KB	Kilobyte
MB	Megabyte
OS	Operating System
RAM	Random Access Memory
RQ	Research Questions
TTL	Time-To-Live
UI	User interface
URL	Uniform Resource Locator
VM	Virtual Machine

Acknowledgements

Special thanks to Prof. Alessandro Margara and Prof. Gianpaolo Cugola for the professionalism and dedication with which they guided this project.

We would also like to thanks our families and our friends that supported us through our academic path.

Last but not least, we want to remind that this thesis has also been possible thanks to the resources provided by Politecnico di Milano and to the knowledge acquired at this outstanding University.