Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Edgeless: Design and Implementation of Serverless Computing at the Edge for Performance in Precision Agriculture

Author: Edgardo J. Reinoso Campos

1st supervisor:	Dr. Ir. Animesh Trivedi
daily supervisor:	Matthijs Jansen
2nd reader:	Dr. Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 7, 2023

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

The development of computing has evolved to smaller devices, like smartphones and wearables. These devices generate data at an unprecedented rate, requiring to process data much faster and closer to where it is generated. Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed. However, the limitation of this paradigm is processing high amount of data on individual edge devices, due to constraint of computational resources available. Serverless is a modern execution model for building, deploying and scaling applications, without managing about the infrastructure. Therefore, the focus of this thesis becomes to build a system that uses serverless functions deployed at the edge, coined as Edgeless. deploying serverless functions at the edge becomes the main focus of this thesis through a system, Edgeless. By performing a thorough evaluation using serverless functions deployed at an edge device, this thesis unravels whether the benefits of cloud scalability can be translated to the edge. As part of this evaluation, a real use case in the domain of Smart Farming is implemented to benchmark the system. In order to carry such evaluation, two type of workloads were considered, sequential and concurrent. During the experiment, Lambda functions and Docker containers were deployed in both cloud and edge environment. The idea is to explore whether cloud Lambda function scalability provides the same throughput benefits at the edge. The scaling capabilities of Lambda are attributed to the initialization of concurrent execution environments. From the results of the experiment, cloud Lambda functions yield higher throughput in concurrent workloads than in sequential. However, at the edge, Lambda functions do not have the same concurrency capabilities, resulting in a lower throughput than Docker containers.

Acknowledgements

This hardeous work has been accomplished not only by the author, but also by many other people that helped directly or indirectly along the way.

First and foremost, I would like to thank my daily supervisor, ***Matthijs Jansen***, for bringing me under his wing in this long year process of exploration and research. He has, without a doubt, been the most influential person during the development of this master thesis. His knowledge and guidance helped me overcome many challenges, particularly when it came to structuring the thesis, finding research questions, and designing the experiments for the evaluation.

Second, I would like to specially thanks ***Thijs Bieling*** and ***Berend de Klerk***, both co-founders of Plense Technologies. They have supported and believed in me since the day I started working on this project, and they were both key mentors and guiders during the thesis. Without their collaboration, this master thesis would not have been completed with such a passion and motivation.

Third, I would like to thank ***Eric Obeysekare***, for being my academic advisor during my transition to this wonderful masters. He has certainly pushed me to do my absolute best during the initial phase of my master's degree.

Fourth, I would like to thank ***Myli Batkeviciute***, for being a close friend who provided emotional support during difficult times. She has been one of the few people who helped me stay focused and keep faith in adverse circumstances.

Last and most importantly, I would like to thank my parents, ***Luis Reinoso*** and ***Graciela Campos*** for being exceptional and wonderful individuals who have given me their support throughout my studies. To my sisters, who were always there unconditionally for emotional support, no matter the time.

Contents

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**

1.1 Context 1
1.2 Problem Statement 2
1.3 Research Questions 3
1.4 Methodologies 4
1.5 Contributions 4
1.6 Plagiarism 5
1.7 Structure 6

2 Background **7**

2.1 Cloud 7
2.1.1 Grid Computing 7
2.1.2 Virtualization (Hypervisor) 8
2.1.3 Service on Demand 8
2.1.4 Benefits of Cloud Computing 9
2.1.5 Shortcomings in Cloud Computing 10
2.2 Edge 11
2.2.1 Evolution of Mobile Networks 11
2.2.2 Mobile Cloud and Edge Computing 12
2.2.3 Classification of Edge Computing 13
2.2.4 Benefits of Edge Computing 15
2.2.5 Shortcomings in Edge Computing 16
2.3 Serverless 16
2.3.1 Technical Specifications 17

2.3.2 Anatomy of a Serverless Function 18
2.3.3 Lifecycle of a Serverless Function 20
2.3.4 Use Cases for Serverless Functions 21
2.3.5 Shortcomings in Serverless Computing 22

3 Design of Edgeless: a Serverless System at the Edge 23
3.1 Agriculture Workload: Signal Processing 23
3.2 Edgeless Requirements for Agriculture System 25
3.2.1 Functional Requirements 25
3.2.2 Non-Functional Requirements 27
3.3 High-Level Overview of Agriculture System 27
3.4 Components of the Agriculture System 29
3.4.1 Design of Edgeless Engine: Greengrass 30
3.4.2 Design of Edgeless Function: Lambda 31
3.4.3 Design of Edgeless Storage: S3 32
3.5 Design Process and Alternatives 32

4 Implementation of Edgeless: Processing Engine at the Edge 35
4.1 Implementation of Edgeless Engine: Greengrass 35
4.1.1 High-Level Overview of Greengrass 35
4.1.2 Component Lifecycle Management 38
4.1.3 AWS Managed Greengrass Components 39
4.1.4 Custom Greengrass Components 40
4.2 Implementation of Edgeless Function: Lambda 43
4.2.1 Workload Execution: Sequential and Concurrent 43
4.2.2 Invocations Methods: Synchronous and Asynchronous 44
4.3 Implementation of Edgeless Storage: S3 46
4.3.1 S3 High-Availability: Replication and Versioning 47
4.3.2 S3 Storage Classes 48

5 Evaluation: Setup and Deployment 51
5.1 Experiment Setup 51
5.1.1 Simulation Setup For Capturing Audio Files 51
5.1.2 Experiment Setup in Cloud Environment 52
5.1.3 Experiment Setup in Edge Environment 55
5.2 Experiment Deployment 57

5.2.1 Infrastructure as Code (IaC) . . . 57
5.2.2 Serverless Application Model (SAM) . . . 58
5.2.3 Experiment Pipeline in Cloud Deployment . . . 59
5.2.4 Experiment Pipeline in Edge Deployment . . . 61

6 Evaluation: Design and Experiments 63
6.1 Experiment Design . . . 63
6.1.1 Throughput and Concurrency . . . 64
6.1.2 Experiment Workload Design . . . 64
6.2 Experiment Results . . . 65
6.2.1 Cloud Experiment Results (RQ1) . . . 65
6.2.2 Edge Experiment Results (RQ2) . . . 72

7 Limitations 77
7.1 Design Limitations: Vendor Lock-In . . . 77
7.2 Implementation Limitations: Greengrass and Lambda . . . 77
7.2.1 Greengrass Component Updates . . . 78
7.2.2 Edge Lambda Function Deployment . . . 78
7.3 Experiment Limitations: Multi-thread MQTT Messages. . . . 78

8 Related Work 79
8.1 Edge and Agriculture . . . 79
8.1.1 FarmBeats: An IoT platform for data-driven agriculture (1) . . . 79
8.1.2 Ubiquitous Sensor Network in Precision Agriculture (2) . . . 81
8.1.3 A Game Theoretic Analysis for Cooperative Smart Farming (3) . . . 82
8.2 Edge and Serverless . . . 83
8.2.1 Serverless management for fog computing framework (4) . . . 83
8.2.2 Serverless in Machine Learning Applications at the Edge (5) . . . 85
8.2.3 A Serverless Real-Time Data Analytics for Edge Computing (6) . . . 85

9 Conclusion 87
9.1 Main Contributions . . . 87
9.2 Future Work . . . 89
9.2.1 Energy Consumption . . . 89
9.2.2 Network Latency . . . 90

CONTENTS

A Artifact Appendix **91**

A.1 Abstract . . . 91

A.2 Checklist . . . 91

A.3 Description . . . 92

A.3.1 How to access . . . 92

A.3.2 AWS Managed Component Dependencies . . . 92

A.3.3 Custom Managed Components . . . 92

A.4 Installation . . . 93

A.4.1 Greengrass Core Software . . . 93

A.4.2 Web Server Setup . . . 93

A.4.3 Component Deployment . . . 94

A.4.4 Lambda Function Update . . . 94

A.5 Experiment Workflow . . . 95

A.5.1 Cloud Experiment Workflow . . . 95

A.5.2 Edge Experiment Workflow . . . 96

A.6 Evaluation and expected results . . . 96

A.6.1 Cloud Evaluation Results . . . 96

A.6.2 Edge Evaluation Results . . . 96

References **97**

List of Figures

1.1 Thesis main contributions along with Research Questions: (1) Edgeless design and implementation, (2) Cloud serverless performance, (3) Edge serverless performance, (4) Evaluation of network protocols latency. 5

2.1 Landscape of Computing Paradigms (7) 14
2.2 Reference Architecture of Serverless (8). 18
2.3 Lifecycle of a Serverless Function (8). 20

3.1 Plense Technologies Microphone Sensor Setup (9). 24
3.2 Functional requirements composition for Serverless Management (SM) and Serverless Function (SF). 25
3.3 Design of Edgeless: serverless functions deployed at the edge for processing in-flight data coming from microphone sensors. 28
3.4 Components of Edgeless Ecosystem applied in Precise Agriculture. 29

4.1 AWS IoT Greengrass device is composed by: (1) Core device, (2) Client device, (3) Greengrass Components, (4) Deployments, (5) Greengrass core software. 36
4.2 Greengrass component lifecycle processes 37
4.3 Greengrass core device component decomposition. 40
4.4 Sequential executions with respect to the incoming requests, when a single execution environment handles two requests one after the other. (10). . . . 43
4.5 Concurrent executions with respect to the incoming requests, when multiple requests are done simultaneously with various execution environments working (10). 44

4.6 Lambda invocation methods: Synchronous, where a dedicated channel is established between the client and function. Asynchronous, where the client does not wait for a response from the function 45
4.7 Additional destinations where events can go after being processed by a Lambda function. 46
4.8 S3 data replication process on geographically separated availability zones. . 47
4.9 S3 versioning on bucket, on (1) create and update object and (2) read object. 48

5.1 Setup for evaluating serverless performance in the cloud 53
5.2 Setup for evaluating serverless performance at the edge 55
5.3 Internal process for deploying resources in AWS using CloudFormation . . . 59
5.4 Deployment arrangement for cloud experiment setup using SAM 60

6.1 Total execution time for **Cloud Sequential Workloads (W1)** with 100 requests from three execution environments: 1. Lambda and Docker (2. cold start and 3. warm start). 66
6.2 Time per execution for each individual request. 100 requests are executed twice by different runtimes, Lambda and Docker Containers. Furthermore, it is visible the dimension of cold start and how much of an impact it may have in the first execution request. 67
6.3 Process for scheduling a Docker container in ECS, taking on average between 50 and 60 seconds from the Provisioning phase to the Stopped phase. 67
6.4 Total execution time for **Cloud Sequential Workloads (W1)** with several of requests (100, 300, 500, 700 and 1000) from two runtimes: Lambda and Docker containers (warm). 68
6.5 Total execution time for **Cloud Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) invoking concurrent Lambda function. 69
6.6 Execution environment provisioned by Lambda when invoked concurrently. 70
6.7 Total execution time for **Cloud Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) using two runtime: Lambda and Docker containers (warm). 70
6.8 Lambda and Docker memory utilization with concurrent workloads. 71
6.9 Total execution time for for **Edge Sequential Workloads (W1)** with several requests (100, 300, 500, 700 and 1000) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container). 72

6.10 Total execution time for **Edge Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container). 74

8.1 FarmBeats architecture for edge processing in drones and other IoT devices (1). 80
8.2 Architecture from Ubiquitous Sensor Network in Precision Agriculture (2). . 81
8.3 Enter Caption 82
8.4 Enter Caption (4). 84
8.5 High level architecture of cloud and edge data analytics platform (6). 86

LIST OF FIGURES

List of Tables

3.1 Alternatives considered during the design process for Edgeless. 34

4.1 Comparison among different S3 storage classes (11) 49

5.1 Compute services setup for serverless performance experiment in the cloud. 54
5.2 Raspberry Pi specifications used during the experiment of serverless performance. 56

6.1 (W1) Fixed number of sensors and several number of requests - Sequential. 65
6.2 (W2) Fixed number of requests and several number of sensors - Concurrent. 65
6.3 Total execution time (seconds) for **Cloud Sequential Workloads (W1)** from three execution environment: Lambda, Docker (cold start and warm start). 66
6.4 Total execution time (seconds) for **Cloud Sequential Workloads (W1)** with several requests (100, 300, 500, 700 and 1000). 68
6.5 Total execution time (seconds) for **Cloud Concurrent Workloads (W2)** with 100 requests in parallel with several sensors -threads- (5, 10, 20, 25, 50). 71
6.6 Total execution time (seconds) for **Edge Sequential Workloads (W1)** with requests (100, 300, 500, 700 and 1000) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container). 73
6.7 Total execution time (seconds) for **Edge Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container). 74

LIST OF TABLES

1

Introduction

Today, our society is relying on data more than ever. Smartphones, wearables and other sensing devices are constantly capturing data from our everyday routine. Data has become notoriously beneficial for understanding our behaviors to improve our daily performance. Athletics, for example, can use smartwatches and other sensing devices to optimize their performance, identify areas for improvement, and prevent injuries. Similarly, self-driving cars rely on various sensors to make decisions about speed or directions.

Technology is rapidly evolving to a more data-driven ecosystem, where sensing devices play a key role in innovating many industries today. As a result, edge computing has become notably important with this surge of sensors and microprocessors. This enables data processing to be done closer to where data is captured. The benefit is lower latency and faster performance. Therefore, today, more than ever, edge computing has a great growth projection as our society ventures in the world of Internet of Things (IoT).

1.1 Context

The main idea of edge computing involves processing and analyzing data closer to the source, rather than sending it all to data centers, which may be far away from where this data originates. In general, this computing paradigm helps in reducing latency, which is the time taken for data to travel back and forth from the source to the destination. Furthermore, it also helps improve system performance, which is a key when making real-time decisions in certain domains, e.g. self-driving cars, as discussed earlier. Lastly, data protection also increases, as sensitive information is not transmitted over the network. All in all, edge computing has brought many benefits that require further research.

Over the past decade, edge computing has evolved significantly with the proliferation of sensors and microprocessing devices. Today, over 17.08 billion IoT devices [1] are being used across many industries to capture meaningful data relevant to business success. As the number of devices increases, the need for more efficient and faster computational processes also arises. Therefore, this has led to the development of more advanced edge computing technologies, that are capable of processing data that is generated from sensors.

Agriculture is one of the domains in which edge computing has shined in the last decades. This rapid adoption has been mostly due to the drawbacks that traditional agricultural practices, e.g. inefficient use of resources such as water and fertilizers, or inability to respond to environmental changes. As a result, these shortcomings opened the gates for more efficient and accurate processes through the use of IoT devices. Consequently, this has led to an astonishing adoption in sensors, with more than 225 million devices deployed in agricultural by 2024 (12). Therefore, processing at the edge is the key for this success.

1.2 Problem Statement

Edge computing can still present some fundamental shortcomings when it comes to device-to-device communication in widespread area, resource utilization and energy efficiency.

Network Communication. Device-to-device communication is a challenge in areas with limited network connectivity or wide open spaces. Although processing at the edge improves the network latency, the issue arises when network connectivity is low, since there is no way of transferring data. This limitation can specially be seen in agricultural fields, with wide open areas that affect reliability of the network.

Limited Utilization. This poses limitations for edge devices due to their lower computational capacity for processing heavy workloads. Therefore, running applications at the edge becomes more restrictive when compared to the cloud. Furthermore, as the number of sensors increases, processing data with limiting resources can quickly become a bottleneck, leading to a degradation in system performance.

Energy Consumption. Lastly, energy consumption is a concern when computing workloads in smaller devices. Some edge devices rely on batteries to properly incoming process data from sensors. Therefore, applying heavy computation, such as image recognition or weather prediction, can deplete the battery of the edge device.

[1] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

The focus of this thesis will revolve in how to implement more efficient computational models for better utilization of resources at the edge. Hence, the aim of designing and implementing a serverless system at the edge is the scope of this focus.

What is Serverless? It is an efficient computational model that allows applications to be developed and deployed without the need to manage any servers or infrastructure. It can automatically scale resources up and down depending on the application's demand.

1.3 Research Questions

Although there are clear benefits of serverless applications deployed in the cloud, specially on scaling and managing, the question really becomes whether these same benefits can be applied at the edge. Therefore, the purpose of this thesis is to explore the performance of serverless functions at the edge. The following two research questions will initiate this scientific study that leads to the main contributions in this master thesis.

RQ1: Cloud

What is the **scalability performance** of **Lambda functions** deployed in the **AWS cloud** with **sequential and concurrent workloads**?

One of the benefits of adopting serverless computing is autoscaling. This capability pose great potential for handling any number of requests at a determined point in time, by dynamically increasing or decreasing the computational capacity. Therefore, this research question presents an evaluation of performance of serverless functions. The idea is to understand how this autoscaling mechanism works. Hence, the answer to this question would be beneficial for creating a performance metric. Such a metrics would be useful to understand whether these same scalability benefits can be extended to the edge.

RQ2: Edge

What is the **scalability performance** of **Lambda functions** deployed in a **Raspberry Pi 4** with **sequential and concurrent workloads**?

Considering the results obtained from the first research questions, this question explores the capabilities of extending such scalable benefits to the edge. Hence, the focus of this research question is experimenting and evaluating how the autoscaling of serverless functions works when deployed at the edge. Even though functions might be bounded by the computational capacity of the edge device, evaluating such performance benefits is useful

for designing scalable systems at the edge. The idea is not for comparing the results with cloud, but rather understanding whether scaling happens similarly at the edge.

1.4 Methodologies

A comprehensive research methodology is applied with the goal of pursuing systematic investigation and findings on how edge devices can be optimized in agricultural fields.

1. **M1 Quantitative research:**An extensive literature survey was conducted (separate from this work) to investigate the different serverless function optimizations that can be done, in the cloud and then later at the edge. Furthermore, this survey helped in finding the requirements for edge systems in the agricultural field. Besides, it contributed to the exploration of scientific literature for this master thesis.

2. **M2 Design, abstraction, prototyping:** Edgeless was designed and implemented to process audio files during the development of this master thesis. The system uses serverless functions as compute power for classifying and reducing signals taken from a microphone sensor. First the system was developed and deployed in the cloud, and then it was deployed in a Raspberry Pi (RPi) 4.

3. **M3 Experimental research (Cloud):** The system was benchmarked with methods of experimental research. In this particular case, audio files serving as inputs were gathered from microphone sensors deployed in greenhouses. Such files were provided by Plense Technologies, a company for which collaboration was key during the execution of this work. Initially, the experiments started with a smaller sample set of 10 audio files, but then it was later scaled to 100 files.

4. **M3 Experimental research (Edge):** Similarly, the system was deployed and benchmarked at the edge, in the Raspberry Pi. The same audio files were used as inputs, to mimic the setup and execution than in the previous experiment. The difference is that the serverless functions were executed locally in the device, as opposed to a data center in the cloud. The experiment also started with a smaller sample set (10 files), to then scaled to a bigger sample (100 files).

1.5 Contributions

When addressing the research questions previously discussed, there have been several contributions made throughout the development of this master thesis:

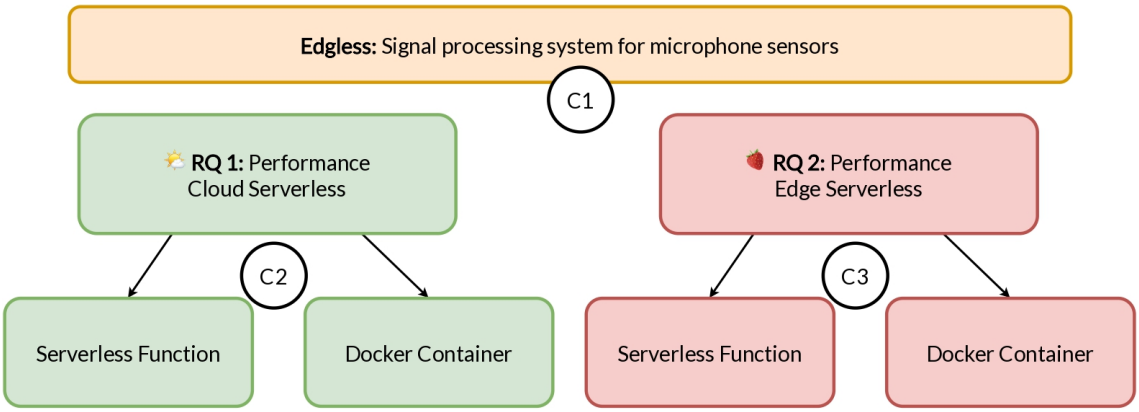

Figure 1.1: Thesis main contributions along with Research Questions: (1) Edgeless design and implementation, (2) Cloud serverless performance, (3) Edge serverless performance, (4) Evaluation of network protocols latency.

C1: Design, implementation and deployment of Edgeless, a signal processing system that captures data from plants using microphone sensors, developed by Plense Technologies. These sensors are deployed in greenhouses, where they constantly monitor plants to determine optimal conditions for crop vegetation, e.g. water level, temperature.

C2: A cloud implementation from Edgeless was first done for processing audio signals. In the evaluations, two environments are deployed, one with a serverless backend, and one with Docker containers. Experiments were executed to analyze the throughput, request per seconds (RPS), from these two environments. The idea is to understand the performance benefits from serveless functions when compared to containers.

C3: An edge implementation from Edgeless was later done for processing audio files in the RPi. Similar, two execution methods are done, one with serverless functions and another process running in bare metal. Consequently, the aim is to study performance of serverless functions at the edge. In detail, evaluating whether scaling functionalities are similar have the same benefits in the edge as in the cloud.

Figure 1.1 presents a visual representation of the main contributions addressed in this master thesis. Furthermore, it shows how contributions are mapped to research questions.

1.6 Plagiarism

I confirm that all material present in this report, unless explicitly stated, is the result of my own efforts. No parts of this report are copied from other sources unless credited

and properly cited. The work has also not been submitted elsewhere for assessment. I understand that plagiarism is a serious issue and should be dealt with if found.

1.7 Structure

This thesis has been structured in 8 distinguished sections, starting with the one being read and following the next chapters.

Chapter 2. Background. Provides the necessary information to understand more about the three different domains in master work: Edge, Serverless and Agriculture.

Chapter 3. Design. Dives deeper into the design and architecture of the signal processing system, analyzing the functional and non-functional requirements. Furthermore, this section presents a high level overview of how serverless and edge intertwined.

Chapter 4. Implementation. Focus more on the implementation of the system, looking into specific components that were used while developing the system. Furthermore, detail explanation is given about the algorithm for processing audio files in functions.

Chapter 5-6. Evaluations. Presents the evaluation methods for both research questions. The performance evaluations will be divided into two, one for the cloud and one for the edge. Furthermore, this section presents the results and important discussions.

Chapter 7. Related work. Goes into details on some of the related work that have already been done both in academia and industry. This section will present useful implementations that were highly considered as inspirations for this master thesis.

Chapter 8. Limitations. Presents some of the shortcomings that were considered during the experiment, for both cloud and edge. Furthermore, explanations are given for possible ways to overcome these challenges, should there be more time.

Chapter 9. Conclusion. Summarizes the main contribution, reinforcing the findings and results about the research questions. It suggests the direction for future work on how serverless can be further researched at the edge, in terms of network and energy.

2

Background

This chapter presents the evolution of computing for understanding the need of serverless at the edge. It is divided into three sections, cloud, edge and serverless. Section 2.1 discusses cloud computing, primarily the origins, adoption, main benefits and shortcomings. Section 2.2 introduces edge computing, with a particular focus on the development of networks, the main benefits and shortcomings. Section 2.3 presents serverless computing, along with its technical specifications, anatomy, use cases and shortcomings. The goal is to provide some conceptual knowledge on these topics.

2.1 Cloud

The fundamental concept of cloud computing can be dated back to the times of the ARPANET (1960s), where researchers explored the idea of time-sharing with multiple users accessing a single computer simultaneously (13). This permitted the foundation of resource-sharing, which would eventually become an essential pillar of the cloud. As a result, distributed computing emerged, with the primary goal of having geographically dispersed computers working together as a single, integrated system. Therefore, the development of grid was the first inception into the world of cloud computing.

2.1.1 Grid Computing

The rise of grid computing addressed the need for vast computational power by integrating computers in separate parts of the world into a cohesive infrastructure (14). Therefore, this paradigm extended the principles of resource sharing to a global scale. However, there were still challenges in regard to resource management and isolation, since the grid was operated by multiple organizations and required specialized knowledge (15). Such issues

were later addressed in the era of cloud computing, where virtualization really played a key role in resource isolation.

2.1.2 Virtualization (Hypervisor)

The key initial factor for the success of cloud computing was the surge of virtualization. Traditional approaches to computing infrastructure were inherently very limiting to the specific hardware configurations. Consequently, resource abstraction emerged as a crucial imperative to overcome such challenges. Therefore, virtualization is a technology that abstracts the physical hardware, decoupling computing resources from the underlying machine (16). Eventually, this technology would be developed in Virtual Machines (VMs), as it encapsulates operating systems and applications within a virtualized environment.

Hypervisors

The heart of virtualization lies on hypervisors. Hypervisors are the orchestrator for this abstraction layer, which provides the means to create, manage and operate multiple VMs on a single physical server (17). Furthermore, hypervisors enabled the coexistence of diverse operating systems and applications, fostering resource isolation and security. Therefore, a single physical server could host multiple VMs, each operating independently and unaware of the others. This marked an outstanding shift in resource utilization, reducing the need for underutilized servers in data centers . As a result, utility computing surged as the core of cloud for service on demand.

2.1.3 Service on Demand

Utility computing is a fundamental concept in the development of cloud. It focuses on making computing services available on demand, shattering the traditional model of fixed and dedicated computing infrastructure. As a result, this gave birth to the vanguard of cloud service providers. Amazon Web Services (AWS) [1] was the pioneer in this innovative model, dating back to 2006 (18). Following AWS, other providers such as, Google Cloud Platform (GPC) [2], Microsoft Azure [3], IBM Cloud [4], entered the cloud computing market.

As the popularity of the cloud usage increased, so did the different kinds of services offered. Therefore, services such as storage, database, and later on machine learning, started to become more widely available to the public. This allowed many companies to design

[1] https://aws.amazon.com/
[2] https://cloud.google.com/?hl=en
[3] https://azure.microsoft.com/en-us
[4] https://www.ibm.com/cloud

complex distributed systems without having specialized expertise in the infrastructure. Hence, the terms Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) came into existence.

- **Infrastructure as a Service (IaaS) (19).** Computing service that provides virtualized resources by renting specialized hardware, such as servers, storage, and networking components. This avoids the complexity and cost of owning and maintaining physical servers and infrastructure. Furthermore, it gives control over the operating systems and applications, while letting providers manage the underlying hardware.
- **Platform as a Service (PaaS) (20).** Computing service that provides a platform for developing, running and managing applications without dealing with the complexity of building and maintaining the underlying infrastructure. Furthermore, it allows developers to focus on building application logic, while letting the platform, managed by providers, handle aspects such as scalability, security, and maintenance.
- **Software as a Service (SaaS) (21).** Computing service that delivers software applications on subscription basis. This model allows users to access applications hosted on cloud services instead of installing and maintaining the application on their own servers. As a result, SaaS providers manage everything from application security and performance to updates and maintenance, while delivering the service.

2.1.4 Benefits of Cloud Computing

This proliferation in cloud services permitted businesses to scale their IT infrastructure much quicker and without trouble, from both an operational and cost perspective. Therefore, some of the benefits of cloud can be defined in the following categories:

Scalability on demand. This is the ability to swiftly adapt computing resources to changing demands and workloads. As a result, this is an invaluable virtue for businesses, since scalability allows them to meet their required capabilities during peak periods (22). Therefore, cloud computing allows to seamlessly scale computing resources upwards, ensuring that applications remain responsive and reliable. Conversely, during quieter periods, these resources can be scaled down, preventing unnecessary costs by avoiding the maintenance of idle hardware.

Cost-efficiency on utilization. The pay-as-you-go model has revolutionized the computing industry. Instead of costing with significant upfront investments, it is now

possible to pay only for the resources and services that are actually consumed. This has been a transformative shift from a capital expenditure (CapEx) to an operational expenditure (OpEx) (23). Furthermore, adopting cloud reduces the need for an extensive in-house IT team, since the service provider handles the maintenance, updates and security of the underlying infrastructure.

Agility in deployment. Rapid deployment of applications and services is no longer a time-consuming endeavor. Furthermore, this agility is particularly evident in its support for DevOps practices, which usually brings collaboration between development and operations teams for new software releases to applications (24). Additionally, continuous integration and continuous deployment (CI/CD) have become the frontier of this agility, delivering high-quality software at an unprecedented pace (25). Therefore, this synergy has ensured quick adaptation to changes in market demands.

Disaster recovery solutions. Cloud computing has ushered in a new era of disaster preparedness, offering unparalleled advantages in terms of flexibility, reliability and accessibility. One of the ways for achieving this is by creating redundant backups (26). Therefore, in the event of a disaster, this would not affect the availability of the data since it would be geographically distributed across different servers, in potentially separate data centers. Backups can then be restored, minimizing the downtime that can occur, while also resuming operations with minimal disruption.

2.1.5 Shortcomings in Cloud Computing

So far, cloud computing has been discussed as the most widely adopted paradigm in our era, massivizing computer systems like never before. Nonetheless, cloud has still some noticeable shortcomings that have not been yet addressed. Therefore, it is important to highlight these issues to understand how other paradigms, like edge, can be complementary.

Latency in data transfer. This relates to the time it takes for data to travel from its source to its destination, often referred to communication or network delay. This can be attributed to the physical distance data must traverse or network congestions. While cloud computing has revolutionized the way information is accessed and processed, the issue of latency continues to cast a shadow over certain applications (27) and use cases, from online gaming experiences marred by lag to financial transactions that demand split-second precision.

High energy consumption. The substantial power requirements of cloud data centers have raised concerns about their environmental impact and long-term sustainability. The sprawling infrastructure of data centers demands vast amounts of electrical power to operate and maintain optimal temperature conditions (28). Data centers run around the clock, supporting a myriad of services and applications. The consequence of this continuous operation is a significant carbon footprint.

Vendor lock-in. Limited control over the infrastructure presents significant disadvantages that resonate across various dimensions in cloud adoption. One of these is the phenomenon of vendor lock-in (29), which is when organizations become heavily reliant on a particular cloud provider's services and tools. The lack of portability can stifle ability to respond to changing business needs or to leverage competitive pricing with other cloud providers.

Security and integrity. This has been a concern when discussing cloud adoption. Users must trust that cloud providers implement robust security measures to protect their data. Furthermore, users may face difficulties in ensuring compliance with specific regulations when limited control over infrastructure where data resides (30). Lastly, unique security needs may find it challenging to implement custom security measures in a shared cloud environment.

2.2 Edge

Edge computing became an important paradigm when Internet of Things (IoT) emerged in the late 1990s and early 2000s (31). As sensors and other devices generated more data, there was a need for processing and analyzing this data closer to where it was generated. Therefore, edge computing came as a compliment of cloud, laying on the same foundations of distributed computing. However, due to the development of networks and microchips, edge computing has become a prominent paradigm that requires further research.

2.2.1 Evolution of Mobile Networks

The evolution of networks has powered edge devices to become more efficient in transmitting data. From the beginning of GSM to the speed of 5G, mobile networks have gone through a transformative speed revolution, which ultimately benefits in reducing the latency between the sender and receiver. The following networks can be described as the evolution of mobile communication.

Global System for Mobile Communications (GSM) - 1990. It was a widely used second-generation (2G) cellular network technology that modernized mobile communications. It was developed to provide digital voice and data services for mobile devices (32). GSM adopted circuit-switching for voice calls, an innovative mechanism for data transfer, which reached a maximum transmission of 9.6 Kbps.

General Packet Radio Service (GPRS) - 1994. It was an enhancement of GSM that introduced packet-switched data services, enabling faster and more efficient data communication (33). As a result, this allowed multiple users to share the same radio channel simultaneously, making more efficient use of network resources. Furthermore, it reached data transmission rates up to 115 Kbps, greater than its predecessor.

Third Generation (3G) - 2000. It was designed to provide faster data transmission, enhanced multimedia capabilities, and improved voice communication compared to its predecessor (34). Therefore, it increased data transmission speeds, allowing users to access the internet, download files, and stream media content more quickly. Furthermore, it provided data transmission rates from 200 Kbps to 384 Kbps.

Fourth Generation (4G/LTE) - 2009. This represented a substantial leap forward in terms of speed and capacity for data transmission. It offered a significantly faster data transmission rate, from between 100 Mbps to 1 Gbps. Furthermore, it contributed to the improvement in latency and multimedia support, enabling higher quality video streaming and calls as well as better online gaming experience (35).

Fifth Generation (5G) - 2015. This is the current mobile network technology used today. It can remarkably deliver speeds in the gigabits per second, allowing rapid downloads, seamless streaming of high-definition content, and real-time communication. Furthermore, it introduces network slicing, improving the network performance by segmentation it in different resources and ensuring optimal delivery speed (36).

2.2.2 Mobile Cloud and Edge Computing

The convergence between cloud and edge computing with mobile networks have allowed new concepts to flourish. The development of these new fields have played a very significant improvement in the performance of systems response time.

Mobile Cloud Computing (MCC)

This paradigm combines mobile and cloud computing to provide seamless, scalable, and resource-rich services and applications. As systems became more complex, mobile devices were limited by their processing power, storage capacity and software capabilities. Therefore, these devices can leverage the vast computational resources and storage from cloud servers. Computation offload became an important concept within MCC.

- **Computation Offloading.** This is a procedure that migrates resource intensive computation from mobile devices to more powerful servers in the cloud (37). There are two different directions: (1) *Edge to Cloud:* involves offloading resources from mobile devices to cloud services. (2) *Cloud to Edge:* entails offloading computation from cloud services to mobile devices. The goal of offloading is to improve the performance, efficiency, or resource utilization of the system.

Mobile Edge Computing (MEC)

This paradigm brings cloud computing capabilities closer to the user by placing processing and storage resources at the network's edge. The primary goal of MEC is to reduce latency, by processing data locally or within user's proximity, enabling real time applications and services (38). Content Delivery Networks (CDNs) is an example where MEC leverage computational capabilities is deployed at the edge of the mobile network.

- **Content Delivery Networks (CDNs).** Servers that are strategically deployed at the edge of cellular networks, reducing the distance between content and end-users. This proximity minimizes latency in delivering content, making it ideal for latency-sensitive applications like video streaming (39). Furthermore, it leverages edge caching to store frequently accessed content locally. This reduces the need for repeated data retrieval from remote servers, speeding up response time.

2.2.3 Classification of Edge Computing

Edge computing has diversified its definition depending on where data is generated, captured and processed. Therefore, there can be different classification for where computation can be performed. Figure 2.1 shows the landscape of computing paradigm, and how they are segmented into the different layers (7), which are presented as follows:

Mist computing. It is the conglomeration of IoT devices that provide data gathering from sensors and other devices. Through microcontrollers and microchips embedded

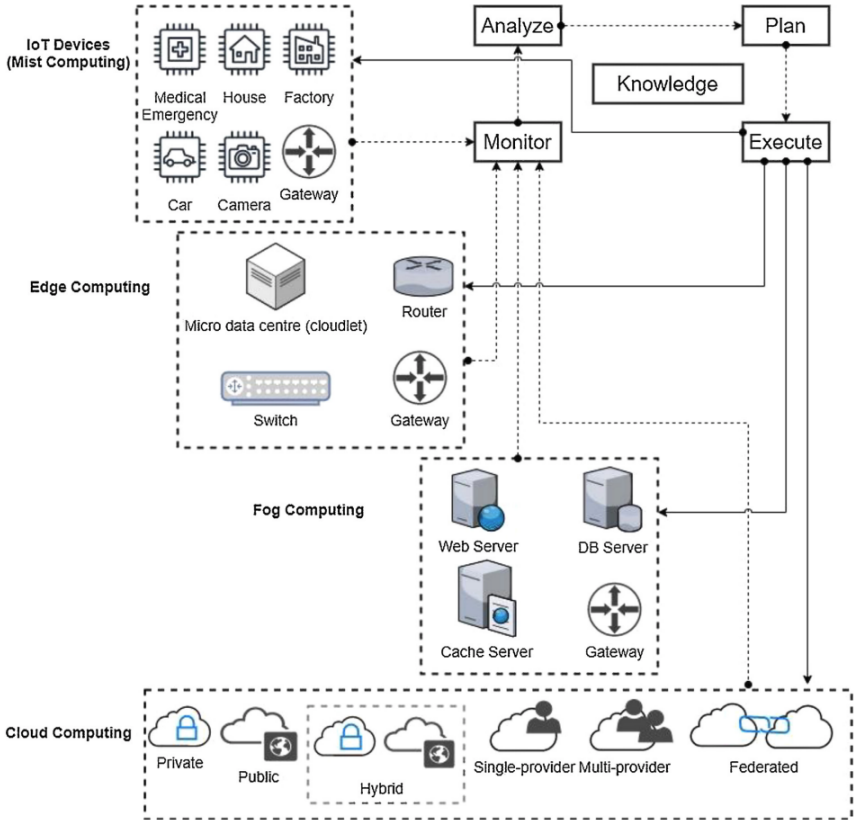

Figure 2.1: Landscape of Computing Paradigms (7)

in devices, mist computing enables local decision-making and real-time data processing, since it is the closest to where data is generated. Therefore, small computational workloads are executed on sensors embedded in a physical device (40).

Edge computing. It is a group of servers and microprocessors that are between the cloud and the mist layer. The aim of this layer is to minimize the latency by applied heavier computational workloads that are offloaded from sensors and mist devices. Furthermore, this layer can be responsible for managing traffic between the IoT sensing devices and the cloud.

Fog computing. It is the group of surrogate servers that extend the cloud computing capabilities to the edge of the network. However, it differs from cloud due to its geographical distribution and support for mobility (41). Furthermore, it is often organized into tiers, where lower fog nodes would be closer to sensors. While higher-tier nodes are closer to the cloud to handle more complex computation.

2.2.4 Benefits of Edge Computing

The development in improving network latency, and the advancement in more powerful microchips, have brought tremendous benefits to edge computing. Therefore, some of the benefits of edge computing can be defined in the following categories:

Latency and bandwidth improvements. Since computation is done closer to where data is generated, this minimizes the potential latency that can occur due to data transfer from an edge device to the cloud (42). As a result, this instantaneous response is optimal for some applications that require millisecond processing, such as autonomous vehicles and drones. Furthermore, doing computation at the edge also minimizes the need to transmit vast amounts of data, since it can be processed locally at the device, conserving bandwidth.

Security and privacy tighten. Edge computing helps with data security, since it does not have to be transferred over the network. As a rule of thumb, anything that is transfer over a network can be overheard by eavesdroppers. In other words, by applying local computation in edge devices, the integrity and privacy of the data is protected, unless malicious attackers purposefully target the device (43). As a result, reducing the surface attack can be very beneficial for certain applications that have very strict requirements with data privacy.

Reliability in network disruption. Edge computing systems can continue to function even if there are connection issues in the network (44). Unlike the cloud, where a sender needs to have a network connectivity in order to receive their processed requests, edge computing can apply local processing, making it optimal for applications in remote locations, such as farms or greenhouses, or mission-critical environments, like autonomous vehicles in highways. Once connection is re-established again, edge devices can resume their communication with other endpoints.

Real-time insights from data. Edge computing has made a significant impact in data analytics, helping decision-making processes across various domains based on live updates from sensors (45). Consider sensors that are deployed in agricultural fields. They constantly capture data about plants, e.g. water level, temperature, irrigation, etc. Therefore, by processing data locally, systems can immediately identify the optimal condition for vegetation and growth, thus taking proper action at the moment of capturing data. This makes a tremendous impact in yield production.

2.2.5 Shortcomings in Edge Computing

One key point to distinguish is that edge is not the replacement of cloud. Rather, cloud and edge should both coexist in the same ecosystem, in order to delivery a high Quality of Service (QoS) that systems require. So far, edge computing has been presented as a great compliment of cloud, but there are still certain shortcomings that need to be discussed.

Limited computational capabilities. Many edge devices have limited computational resources. These devices may have slower CPUs, less memory, and limited storage space (46). Therefore, this can pose constraints on the types of applications that can be run at the edge. For instance, machine learning inference, video processing, or complex data analytics, may struggle to run effectively on resource-constrained edge devices due to the substantial processing power and memory required.

Insufficient energy capacity. Edge devices are bounded (and somewhat limited) by their power capacity (47). As a result, certain workloads can quickly deplete the battery of devices, leading to a shorter operational lifespan, which ultimately degrade performance. Furthermore, this may lead to a higher maintenance cost, as batteries may often have to be replaced. This is even a more plausible problem when edge devices are deployed in remote areas, such as agricultural fields.

Data synchronization among devices. Edges devices can operate independently of each other, creating a difficult environment for synchronizing their workloads (48). As a result, this poses a great challenge for distributed systems built at the edge, since consistency of data and workloads have to be taken in considerations as part of the requirements. Delays and inconsistencies in this synchronization can lead to incorrect decision-making, impacting operational efficiency and accuracy.

2.3 Serverless

Function as a Service (FaaS) emerges as the serverless computing solution that allows developers to focus on building and deploying applications, without managing about the underlying infrastructure. As applications became more complex, so did the need for more lightweight and portable environments. Even though containers made a great innovative leap in packaging and executing code much simpler, there was still some infrastructure management skills required for provisioning and deployment. Therefore, some of the main benefits from FaaS are automatic scaling, ease of management and deployment.

2.3.1 Technical Specifications

There are certain technical points that have made FaaS provides widely adopted when building systems. Here, some of these points are discussed, making emphasis on only those which will be useful for understanding subsequent chapters.

Stateless. FaaS is designed to operate without retaining any state or data between invocation. Stateless functions are independent units of code that do not retain any information from previous executions. Each time a stateless function is invoked, unless previously invoked, it starts with a clean slate, processes the incoming event or trigger, performs its task, and returns a result. Therefore, it doesn't store session data or user information between requests.

Performance. FaaS is designed to provide a high level of performance through automatic scaling. FaaS platforms automatically handle the scaling of functions based on the incoming load. **Concurrency** is at the core of this scalability feature. It refers to the number of concurrent execution required for processing a determined number of requests in parallel. As a result, this helps performance by processing multiple incoming requests simultaneously with different execution environments.

Isolation. FaaS is designed with the same principle of virtualization, isolation among of resources in a shared pool of computation. When a function is invoked, an execution environment is provisioned for processing requests. This environment runs independently and isolated from other processes, guaranteeing fairness and predictable performance in the allocated resources. Thus, this prevents resource contention and interference between executions.

Asynchronous. Serverless functions are inherently event-driven, meaning they are designed to respond to events or triggers. Asynchronous processing enables these functions to handle events without immediate response. Message queues are commonly used for implementing asynchronous processing in serverless applications. Events are placed in a queue, and one or more serverless functions consume and process these events at their own pace. This decouples event producers from consumers, improving system reliability and scalability.

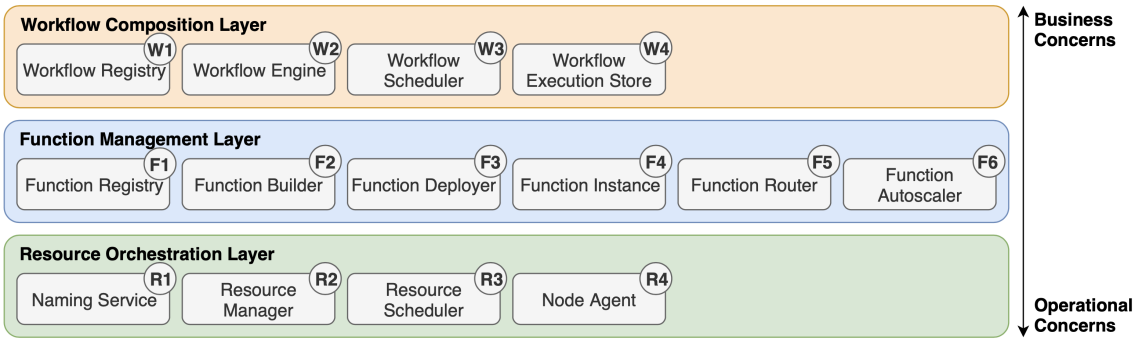

Figure 2.2: Reference Architecture of Serverless (8).

2.3.2 Anatomy of a Serverless Function

This section provides the anatomy of FaaS, which highlights the underlying processes that are required for serverless platforms to execute code. Therefore, in order to analyze these components more in details, a reference architecture is provided (8).

Resource Orchestration Layer

This layer is responsible for the management of physical resources of a cluster of machines. The components in this layer manage the operational lifecycle of the containers or VMs, which are consolidated on physical resources. This can be separated from a FaaS platform as a lower abstraction layer since it delegates resource managements to more mature systems. This layer consist of the following components:

1. **Naming Service:** provides cluster-wide unique and consistent naming to resources. This allows components to identify each other.

2. **Resource Manager:** manages the available resources of cluster-nodes through node agents, ensuring that the state of the resources (eventually) conforms with the desired state. It also monitors resources for changes, and executes actions if needed.

3. **Resource Scheduler:** determines which actions are needed to ensure that the current state of the resources converges toward the desired state of the resources. Schedulers decide where jobs should be deployed depending on the resource capacity.

4. **Node Agent:** is deployed on each node in the cluster. It monitors the local resources, communicates with the Resource Manager by executing instructions it receives.

Function Management Layer

The Function Management Layer contains the core components responsible for the (operational) lifecycle of individual FaaS functions: deploying function instances, executing functions triggered by events, and elastically scaling functions. This layer concerns about the management of arbitrary functions, as opposed to management of resources, like the previous one. This layer consists of the following components:

1. **Function Registry:** serves as a (local or remote) repository of functions. In practice, this registry is often further split into a function metadata store, for low-latency look-ups of function metadata, and a function store, which contains the binaries of the function (the function-code).

2. **Function Builder:** turns function sources into deployable functions. There are certain processes that can transform a function before being deployed. Some of these processes can attribute to compiling, when a function's code is being updated, and validating, when the code actually is tested for error correctness.

3. **Function Deployer:** ensures a function instance (see next component) is deployed. The function deployer combines the configuration stored in the function registry, the parameters supplied by the requester, and other factors into a decision when deploying. Though it decides how the function instance should be deployed, the deployment of the function instance itself is delegated to the resource orchestration.

4. **Function Instance:** is a self-contained worker typically a container—capable of handling function executions. For scalability, discussed previously, a function can have multiple concurrent function instances.

5. **Function Router:** routes incoming requests or events to the correct function instance. If no function instance is available, the function router queues the events to await the deployment of new instances.

6. **Function Autoscaler:** monitors the demand and supply of resources, and elastically scales the number of function instances— adding or removing instances as needed.

Workflow Management Layer

This layer is responsible for handling the management of inter-function state. It can be responsible for two concerns, state management and interdependent functions. Similar to the Resource Orchestration, this layer has a separate category deployed from the FaaS platform. It can be seen as the higher level abstraction that orchestrate the different workflows for functions to work in conjunction. This layer consists of the following components:

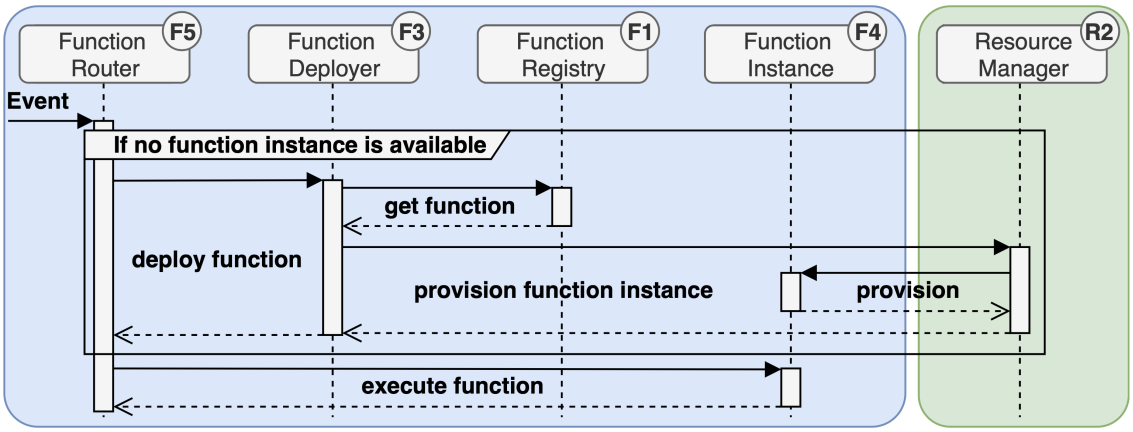

Figure 2.3: Lifecycle of a Serverless Function (8).

1. **Workflow Registry:** serves as a (local or remote) repository of workflows. To be admitted to this registry, the workflow typically requires validation and compilation of its individual tasks.

2. **Workflow Engine:** is responsible for monitoring workflow executions. It takes the appropriate action based on decisions from the Workflow Scheduler.

3. **Workflow Scheduler:** decides which functions to execute when. It makes these decisions based on current state of the workflow execution and historical data.

4. **Workflow Execution Storage:** ensures the persistence of data of workflow executions. A database holds the state of the workflow to ensure the reliability executions.

2.3.3 Lifecycle of a Serverless Function

FaaS platforms tend to undergo through a series of processes for packaging, building and deploying functions in a dynamic and scalable way. The reference architecture, from the previous section, would b used here to describe the behavior of serverless functions.

During the execution of a serverless function, some of the components previously discussed work together to build the execution environment, which would handle the incoming requests. Therefore, FaaS inherently involves the function deployment, as function instances scale up and down depending on the number of requests.

1. An event, such as an HTTP request, arrives at a Function Router, which triggers the deployment of a Function Instance. If no available of function instance is available, then a new one would be initiated. The time between initiation and handling the

request would be coined as the cold start, which is the delay for the platform to stand a new Function Instance.

2. The Function Deployer fetches the function (metadata) from the Function Registry to decide the appropriate configuration of the Function Instance. It then tasks the resource manager with ensuring that a function instance with the appropriate configuration is deployed.

3. Once the function instance is fully deployed, the event can be passed to it to start the function execution. If a function instance is already available for handling the execution, the expensive cold start process can be avoided. Therefore, the function router directs the event to the existing function instance, be it directly over RPC, or using a message queue.

2.3.4 Use Cases for Serverless Functions

Serverless computing can be used for a variety of applications due to its lightweight and scalable benefits. However, determining whether is FaaS suitable can be only driven by the requirements of the system. As a result, there are a couple of use cases where FaaS is particularly beneficial for the scalability, availability and cost-efficiency of the system.

Event Driven

Serverless computing is very well-suited for event-driven applications due to its inherent design and characteristics, such as scalability, cost efficiency, isolation and asynchronous processing, discussed in previous section. Functions can be triggered by events, such as HTTP requests, database changes, file uploads, among others. When an event matches a predefined trigger, the associated function is executed. This reactive execution model is highly efficient, as functions are only invoked when needed. Event-driven applications may experience sudden spikes in event volume. Therefore, serverless platforms are particularly beneficial for these use cases because they can scale to accommodate these bursts, ensuring that no events are dropped or delayed during peak loads.

Web Services and API

Serverless computing also provides many benefits for building web applications and APIs due to enhance development, scalability, cost-efficiency, and operational simplicity of the system. Systems have evolved from single monolith to distributed services working together cohesively. Microservice is an uprising architectural pattern that has gained a lot

of popularity, partly due to the development of serverless computing. The core idea of the pattern is that a function is only responsible for a single API endpoint or functionality. The benefit is the decoupling of business logics, which brings more productivity and efficiency not only when building systems, but also when deploying and scaling them. Systems are able to scale more granular, avoiding the unnecessary scale up from unneeded services.

IoT Applications

Serverless computing is inherently event-driven, making it perfect to handle events from IoT devices as they occur. Therefore, serverless functions can react instantly to IoT events, enabling real-time decision-making and automation. Scalability also plays a big role in IoT applications, due to the high influx of data generated by sensors. Therefore, serverless platforms can excel in providing optimal performance, adapting to changing data volumes and traffic patterns. When events flood in, functions automatically scale out. During quieter periods, resources are efficiently scaled down, reducing operational costs.

2.3.5 Shortcomings in Serverless Computing

So far in this section, the benefits of serverless computing have been clearly specified, as well as the use cases and the advantages for edge deployment. However, serverless computing still faces some issues that are relevant to mention as part of this master thesis.

Cold Starts. Functions experience a delay when they are invoked for the first time or after a period of inactivity. During a cold start, the platform needs to initialize resources, causing latency. This delay can severely impact real-time applications.

Limited Execution Time. Functions tend to have a maximum execution time limit imposed by the platform, often ranging from a few seconds to a few minutes. This can be critical for long-running task applications such as video transcoding.

Resource Limitations. Functions are limited by the amount of CPU and Memory that can be allocated. As serverless platforms abstract infrastructure management, there are often limitations for accessing specific hardware underlying components.

3

Design of Edgeless: a Serverless System at the Edge

This chapter presents a design of Edgeless ecosystem in Precision Agriculture. The chapter is organized as follows: Section 3.1 defines the application workloads and the specific domain in Precise Agriculture for which this master thesis is based on. Section 3.2 provides detail requirements for the design of this agricultural system. Section 3.3 presents a high level design of the system, taking into considerations the (non)functional requirements gathered. Section 3.4, a more detail view of the system is presented, specifying the components. Section 3.5 explains the process for design, and some alternatives considered.

3.1 Agriculture Workload: Signal Processing

Agriculture is one of the industries that have rapidly adopted sensors to modernize manual processes. Smart Farming is an uprising field due to the rapid development of edge computing and sensors. This enabled better and more precise measurements for treating crops. As a result, **Precision Agriculture** is a subdomain, within Smart Farming, which concerns about data driven approaches for real-time monitoring of the fields.

Edge computing can provide tremendous benefits in various applications in Precision Agriculture. For example, crop management can be monitored with cameras and sensors, thus getting the live status of crop's health. Therefore, the main goal for edge in precision agriculture is to create autonomous, innovative, and automated growing practices that can be driven by data collected from plants and their surrounding environment.

For this master thesis, a collaboration with *Plense Technologies* [1], a startup from TU

[1] https://plense.tech/

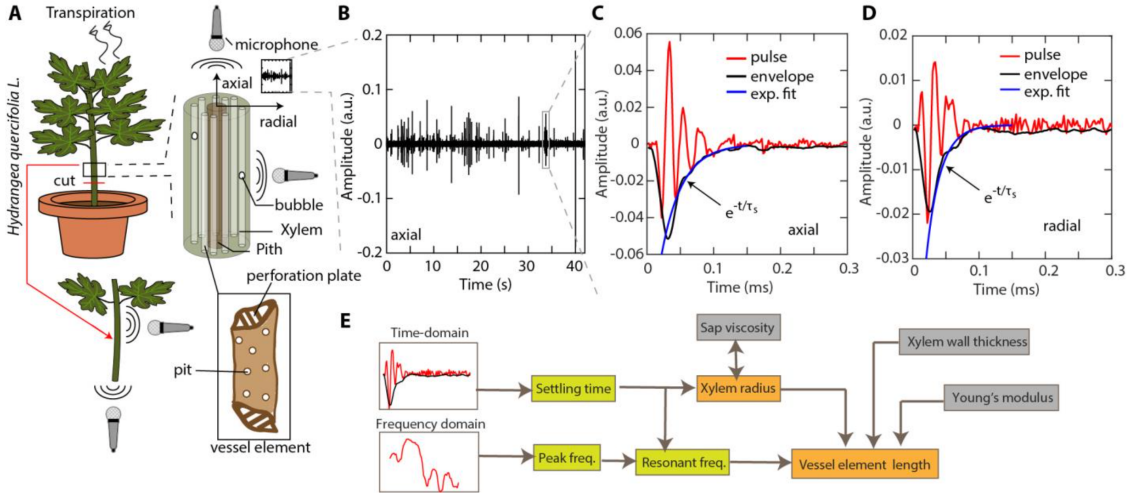

Figure 3.1: Plense Technologies Microphone Sensor Setup (9).

Delft, was made to design, implement and evaluate a scalable and reliable system for processing signal data. Plense focuses on monitoring plants through the use of ultrasound sensors (9). The main purpose of these sensors is to capture acoustic wave generated from plants in order to understand their current health state.

In essence, acoustic waves are recorded once there is an excitation from the plant. This excitation can be related to periods of respiration or drought that plants experience during day or night. Therefore, certain features can be extracted from these acoustic waves that can provide details such as water content in the plant, or irrigation levels. Although the in-depth knowledge is out of the scope of this work, a general understanding is required, specially when gathering requirements for building the system and performing experiments.

Figure 3.1 describes the setup for gathering these acoustic waves generated by the plants.

- **(A)** Recording of ultrasound pulses are captured from the plants along the vertical and horizontal directions. From the figure, the zoom-in part represents a view of the stem, showing the composition inside the xylem vessels. Bubbles seeded in the vessels trigger the emission of ultrasound pulses.

- **(B)** An example is shown of a raw time-domain data for ultrasound recorded on a stem. Time t = 0 represents the start of the recording, which occurs after 5 minutes of drought period experienced by the plant.

- **(C)**, **(D)** A zoomed-in time-domain example is given of these ultrasound pulses.

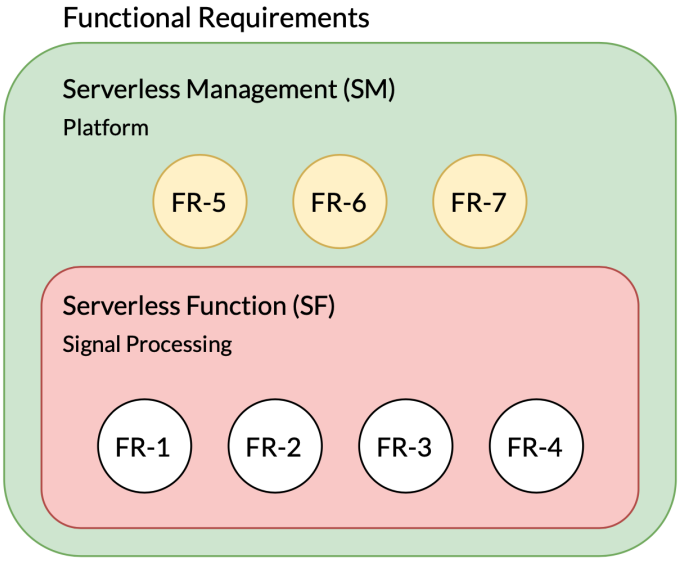

Figure 3.2: Functional requirements composition for Serverless Management (SM) and Serverless Function (SF).

Black curves represent the amplitude envelope, and blue curves represent the exponential fit of the envelope decay.

- **(E)** Shows a schematic flow-chart illustrating these steps. The settling time and peak frequencies are obtained from the time-domain and frequency-domain wave forms. The resonant frequency is obtained from peak frequency and settling time. Using these, xylem radius and xylem vessel element length are extracted, features which are relevant for understanding plant structure and health.

3.2 Edgeless Requirements for Agriculture System

3.2.1 Functional Requirements

These requirements are specifications that describe the functions and features a system, software application, or product must have. They define what the system is supposed to do and provide a clear understanding of the expected behavior (49).

From a functional standpoint, these requirements can be classified in two buckets, as shown in Figure 3.2. *Serverless function requirements (SF)* are the functionalities to perform the signal processing inside the execution environment. *Serverless management requirements (SM)* are the functionalities to perform outside the execution environment.

While the former deals with processing the data, the latter deals with monitoring, logging and deploying functions to execute.

FR-1: Reading signal from sensors (SF). Microphone sensors will be constantly sending data about plants to the Raspberry Pi. The system should be able to read the incoming data sent from these sensors. The format of the data is still unknown, partly due to the development of the sensors. However, for the purposes of the thesis, the data to be analyzed will have an audio format, i.e. wave files.

FR-2: Classifying incoming signal (SF). After reading the signal, the system should be able to classify it according to the following categories: burst, natural emission, poor signal-to-noise ratio, and further inspection. Burst signals can be defined as an induced wavelength by the speaker. Natural emission signals are directly coming from the plants. Poor signal-to-noise ratio signals are those which capture background noises. Lastly, further inspection signals are those who require extra analysis.

FR-3: Reducing incoming signal (SF). After classifying, the system should be able to reduce the size of the signal by the average pulse from the audio file. This reduction would aid when transferring the data, as the original signal ranges from 800 KB to 1.5 MB. As a result, taking the average of all the peaks would still provide meaningful information about the signal, while the reducing the size of data transfer.

FR-4: Outputting processed signal (SF). The reduced signal is then outputted into a JSON string, along with other information such as type of signal, sample rates and number of frames. The system should then be able to make communicate with another service where this data can be persisted. For the purposes of this thesis, an S3 bucket is used as persistence storage, using the standard HTTP Boto3 [1] library.

FR-5: Ease in function deployment (SM). The system should be able to create, update and delete serverless functions from the device in which they are deployed. This process is not as intuitive as in the cloud (depending on the provider), where functions can be easily edited and deployed. The complexity is managing the proper versions from serverless functions, which are deployed to the edge environment.

FR-6: Logging and notification mechanism (SM). The system should be able to provide some logging messages, which includes information and error messages.

[1] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

FR-7: Dependency requirement management (SM). The system should be able to handle dependencies and other necessary components that are required for the serverless function to run and execute. This functionality requires the platform to properly set up the environment for the serverless function, meaning building and deploying the function that process the data from sensors.

3.2.2 Non-Functional Requirements

Non-functional requirements define the overall quality attributes, constraints, and external factors that a system must adhere to. Unlike functional requirements, which describe specific behaviors and functions of the system, non-functional requirements describe how the system performs those functions.

NFR-1: Ensure performance during peak times. The system should be able to provide an optimal response time when processing signals. This needs to be particularly the case when the number of sensors increase. Therefore, the system should process data at the specified time, no matter how many sensors are deployed in the edge environment.

NFR-2: Rapid scalability for handling data. The system should be able to handle bursts of data during certain times of the day. Microphone sensors records and send data about plants every five minutes. As a result, the serverless functions should scale to meet the demand. This scalability is present horizontally, where functions spin up more execution environments in concurrently for processing incoming signals.

NFR-3: Data availability for reading. Data captured from the microphone sensor is essential for understanding plant behavior and growth. Therefore, this data should be available 99.99% of the times when it is needed to be retrieved. Older data can be archived since it will not be accessed so often, saving some cost on storage. However, archived data should be retrieved within 2 minutes of being called.

3.3 High-Level Overview of Agriculture System

This section presents the design of Edgeless, applied to the specific use case for the agriculture system. At its core, there are three major environments that compose this system, mainly Endpoint, Edge and Cloud. The requirements specified in previous section can be reflected into the design of the overall system. Most importantly, the serverless function

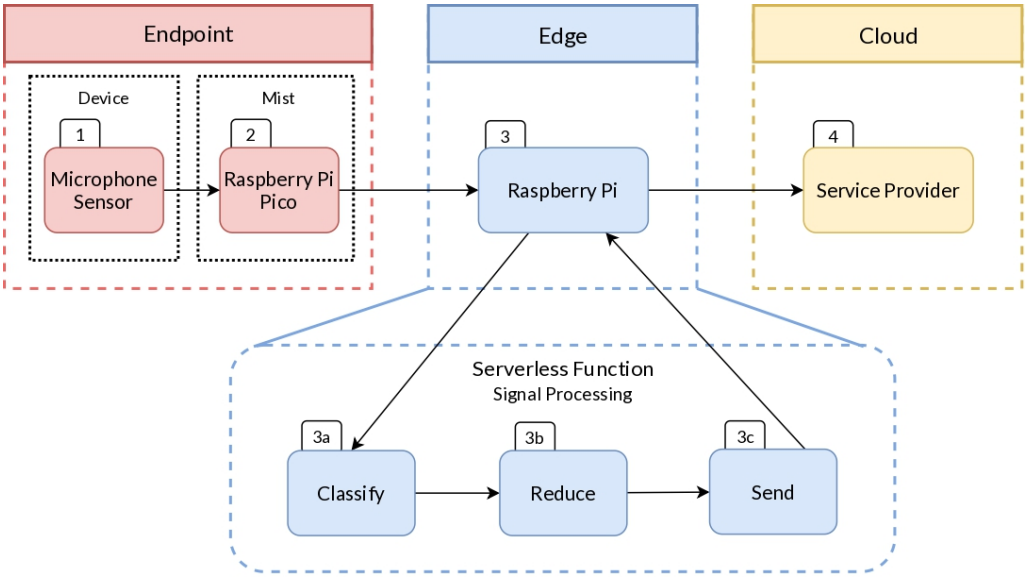

Figure 3.3: Design of Edgeless: serverless functions deployed at the edge for processing in-flight data coming from microphone sensors.

design is presented separate as integration from the edge. This is where the focus of the master thesis will lay.

Figure 3.3 depicts the designed system. The endpoint layer is mainly decomposed in two segments, the device and the mist (discussed in section 2.2). (1) The **Microphone Sensor** is the device where the data is generated. As mentioned in the previous section, the data is captured by a microphone that is clamped onto the stem of a plant. This is an audio data that contain meaningful information about the plant vegetation. Then, the (2) **Micro-controller** receives the signal from sensors, and averages the pulses of the signal as a compression mechanism before transmitting data. For this thesis, the selected hardware is a Raspberry Pi Pico W [1]. After, the (3) **Micro-processor** receives the preprocessed signal at the edge layer. Although right now this design reflects a one-to-one relationship between the endpoint and the edge environment, this setup is scalable for a one-to-many relationship. Therefore, there are potentially multiple Picos that are constantly sending data to the *micro-processor*. For this thesis, the device used for edge processing is a Raspberry Pi 4 Model B [2]. Additionally, this edge device will be executing the serverless function as data comes from the endpoint layer. (3a) **Classify** operation is the first step inside the processing pipeline, it satisfies *FR-2*. Then, (3b) **Reduce** operation will take part by applying further compression for a more compact data transmission; further, this

[1] https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html
[2] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

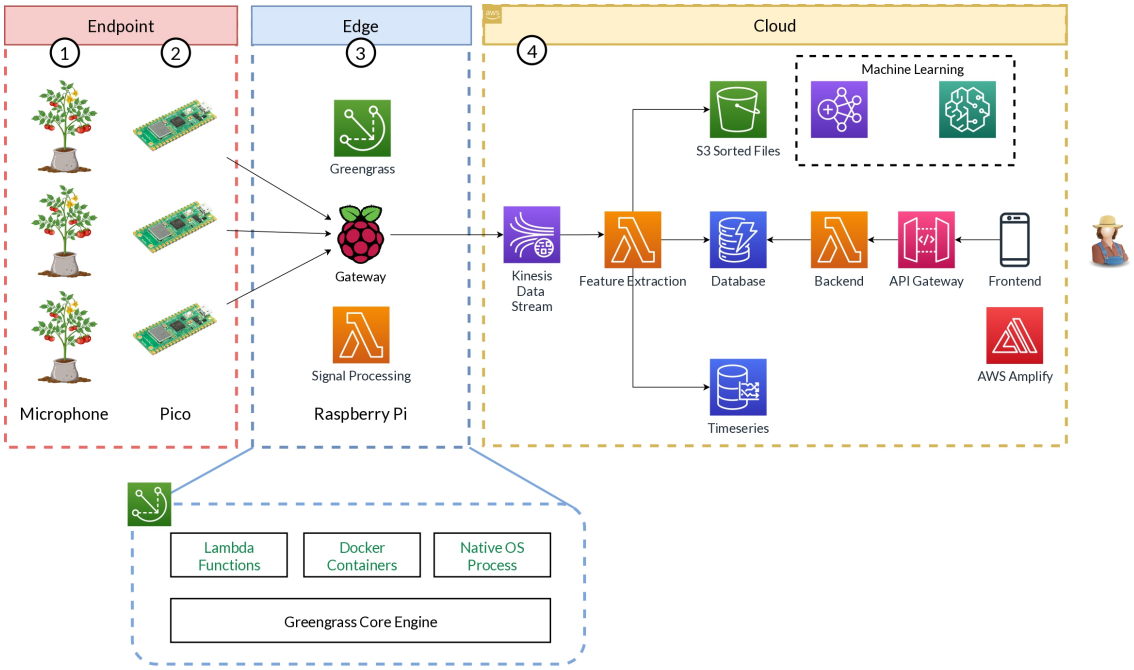

Figure 3.4: Components of Edgeless Ecosystem applied in Precise Agriculture.

satisfies *FR-3*. Lastly, (3c) **Send** operation will be in charge of sending the processed signal to the cloud. Once data is in the (4) **Cloud**, there can be further computation for machine learning and AI use cases, or for other applications in plant health management.

3.4 Components of the Agriculture System

To fully understand the capabilities of Edgeless, it is crucial to look at the specific components that make this system. For this, a more detail viewpoint, shown in section Figure 3.4, is required in order to grasp the specific functionalities from the services used during this design process. These components are very niche to a cloud service provider, which means that they may not be translated in other environments or providers. However, the idea is to explain how these components can work to provide optimal computational performance at the edge. As a result, the provider chosen to design and implement this system on was Amazon Web Services (AWS). There were two rationales behind this selection:

1. **Edge technologies.** AWS is not only the leading provider of IT infrastructure with Cloud services such as Amazon EC2 or Amazon RDS, but it is also deepening its technologies in edge computing. As a matter of fact, they are doing some profound

and innovative work in the area of edge computing, since there are a suite of very powerful IoT services that can extend cloud capabilities to the edge.

2. **Production at scale.** The requirements for building a scalable and reliable system were drafted in collaboration with Plense Technologies. They want to integrate this master work in their production pipeline, to eventually handle thousands of sensors deployed in greenhouses and other agricultural fields. Therefore, they must rely on a powerful cloud service provider that can offer these scalable services.

Nonetheless, the following sections are meant to provide more insight into the three core components of Edgeless. The cloud components, i.e. Machine Learning and Data Analytics, also part of the Edgeless ecosystem, are not discussed since they fall out of the scope of this thesis. Therefore, section 3.4.1 explains further the main engine for edge deployment, Greengrass. Then, section 3.4.2 goes more in depth on the nucleus of the system, Lambda functions. Lastly, section 3.4.3 presents data persistency in an object storage service, S3.

3.4.1 Design of Edgeless Engine: Greengrass

AWS IoT Greengrass is an open source Internet of Things (IoT) edge runtime and cloud service that helps build, deploy and manage IoT applications on edge devices (50). It enables devices to process data closer to where it is generated, react autonomously to local events, and communicate securely with other devices on the same local network. However, there were three main design decisions for choosing Greengrass as the engine.

1. **Multiple runtime environment support.** Greengrass enables support for multiple runtime environments, e.g. AWS Lambda functions, Docker containers, Native OS processes or custom runtime. This flexibility can help find the most suitable runtime environment with respects to the requirement (3.2). Furthermore, it helps determine **RQ2** (1.3), which looks at the performance of serverless execution environments at the edge, particularly evaluating the scaling benefits with the cloud.

2. **Edge communication management.** Greengrass also has the capabilities for establishing device to device communication. Message Queuing Telemetry Transport (MQTT), a lightweight messaging protocol designed for small sensors and mobile devices, is the main protocol Greengrass adopts for sending and receiving data between

edge and cloud. This helps make communication among components very manageable, since it is not require making an established connection between a client and a server, but instead publish and subscribe messages to topics, ideal in an IoT setting.

3. **Integration between cloud and edge.** Greengrass is very integrated with the AWS IoT suite as well as other AWS cloud services, making it ideal for building an ecosystem between the cloud and edge. In turn, this would play a bigger role not only in the edge, where computation is done in locally, but also in the cloud, where other applications that can benefit from this preprocessing. For example, a machine learning algorithm can be trained with the processed data from this engine.

3.4.2 Design of Edgeless Function: Lambda

AWS Lambda is a compute service that can run code without provisioning or managing servers (10). Lambda runs code on a high-availability compute infrastructure and performs all the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. However, there were three main design decisions for choosing Lambda as the serverless function.

1. **Scaling in production.** Lambda provides automatic scaling, allowing it to handle varying workloads by adjusting the number of compute resources allocated to a particular functions. This is necessary as per requirement **NFR-2** (3.2.2), which entails processing a high amount of data coming from microphone sensors. As the number of sensors increases, the system should be able to perform at an optimum rate. Therefore, Lambda can guarantee this scaling with parallel executions.

2. **Event-driven executions.** Lambda functions are triggered by events. Events can come from various sources, such as HTTP requests or other internal events. When an event occurs, AWS Lambda executes the corresponding function. Therefore, this presents an ideal setting for IoT applications, as events can be processed when data generated. In the agriculture use case, events would be processed as they are being captured from the microphone sensors, providing efficient resource utilization.

3. **Integration with Greengrass.** Lambda enables seemingless connection between the AWS cloud and the edge services. Functions can be deployed as a component to a core device, bringing compute benefits to the device. Edge functions can be invoked by publishing MQTT messages to a topic. Furthermore, Greengrass provides more flexibility when working with Lambda, since it provides two deployment modes: No

Container is when the function does not use any containerization in the infrastructure. Greengrass Container is when the function runs inside a specialized container.

3.4.3 Design of Edgeless Storage: S3

Amazon Simple Storage, or S3, is an object storage service that stores any amount of data in objects. Objects are usually composed by a file and metadata that describes the file. Furthermore, objects are stored in buckets, which are considered the containers for persisting data (11). The advantage of storing data in S3 is the durability and availability. However, two main design decisions were made for choosing S3 as the storage service.

1. **Strong consistency model.** S3 provides strong read-after-write consistency for create, update and delete requests to objects. After successfully doing one of these operations (create, update or delete), any subsequent read requests immediately receive the latest version of the object. This is beneficial for Edgeless when implementing a big data analytics application, where often it is required to interpret and analyze the signals captured from plants immediate after writing to an object.

2. **Integration with AWS Lambda.** S3 can work very well with Lambda, as it can trigger functions to execute code based on changes that happen in a bucket. These changes are CRUD (create, read, update and delete) operations that are done to an object. Therefore, this synergy between services boost the efficiency of Edgeless ecosystem, since it adopts an event-driven model, where computation happens based on events that are captured with the microphone sensors.

3.5 Design Process and Alternatives

The design process followed during Edgeless development applied similar principles of the AtLarge design vision (51). The process involved repeated ideation and analysis of design alternatives, which assessed implementation of different serverless system. The first step was identifying the problem, underutilization of compute resources at the edge (section 1.2). The second step was gathering requirements (section 3.2), tailored to specific needs from Plense for processing ultrasound data. The last step in this process was bootstrapping a design, which involved coming up with creative solutions to satisfy the requirements.

Alternative solutions for all the components discussed in this section were considered during this design process. First, there were two different type of serverless functions that

could in theory carry the same job as Lambda, processing workloads on event driven invocations. (1) OpenFaaS [1] is an open source portable functions platform that run serverless functions regardless of cloud provider. (2) Kubeless [2] is another open source solution for serverless functions to run in a Kubernetes cluster. The positive side about choosing an open source solution is the freedom to use serverless functions as needed. Choosing AWS Lambda creates some dependency to the platform and to AWS, making it difficult when migrating to another environment or cloud provider. Therefore, open source solutions provide the flexibility to implement a solution that would be more customizable. The downsides of these alternatives can be classified as follows.

1. **Support and Maintenance.** The level of support and maintenance that these solutions may have with respect to Lambda. AWS has millions of clients using AWS Lambda, some in many production systems. Therefore, Lambda is a reliable solution to choose since there will be less trouble in terms of configuration and troubleshooting.

2. **Scalability and Performance.** Even though the principle of scalability is similar across serverless functions, Lambda tends to provide a better performance than other solutions (52). Lambda functions have been designed to provide greater scalability through efficient scheduling of concurrent execution environments with an increase in number of requests.

3. **Edge Integration.** Deploying functions at the edge can be sometimes challenging due to the compatibility with edge devices, e.g. memory limits or operating system. Therefore, Lambda has been chosen due to the seemingless integration with Greengrass engine, making it more intuitive for deploying and managing functions.

4. **Cloud Integration.** The alternatives considered did not provide the same underlying integration with AWS services. Therefore, since Edgeless has been designed and implemented for signal processing at scale, it is important to consider the cloud service provider this analysis would eventually end up.

For the Greengrass engine, several options were considered as alternatives. The adoption of Kubernetes (k8s) [3] is the most notable, since it also has edge deployment mechanism. KubeEdge [4], for instance, is an open-source system for extending native containerized

[1] https://www.openfaas.com/
[2] https://github.com/vmware-archive/kubeless
[3] https://kubernetes.io/
[4] https://kubeedge.io/

Alternative	Component	Advantage	Downside
OpenFaaS	Function	Portability on any platform	Troubleshooting analysis
Kubeless	Function	Customizable environment	Limiting performance
KubeEdge	Orchestrator	Edge k8s functionalities	Cloud orchestration
Continuum	Framework	Benchmarking tools	ARM integration

Table 3.1: Alternatives considered during the design process for Edgeless.

orchestration to manage and host serverless functions at the edge. Additionally, other frameworks were also explored in combination with *k8s* for deploying serverless functions at the edge. Continuum (53) is a framework that integrates cloud, edge and endpoint computing to provide comprehensive tooling for resource management, workload distribution, and performance analysis. K8s and Continuum provide the orchestration and benchmarking layers necessary to run serverless functions, similar to Greengrass. However, the main disadvantage from adopting these options was time. It would have taken longer to configure and set up the necessary resources in the Raspberry Pi to deploy serverless functions. Therefore, due to the familiarity and integration with AWS services, Greengrass was chosen to provide similar capabilities than the previously mentioned technologies.

4

Implementation of Edgeless: Processing Engine at the Edge

This chapter presents the implementation of Edgeless, for processing incoming signals received from microphone sensors attached to tomato plants. Therefore, this section goes into how each element of Edgeless has been developed and deployed. It is composed by four sections: Section **??** dives deeper into the algorithm developed by Plense Technologies, for interpreting and deciphering pulses from plants. Section 4.1 presents the Greengrass engine, where important details are discussed for building and deploying a component into the core device. Section 4.2 explains about how Lambda functions are implemented, touching on the concurrency topic, which is important for its scalability. Section 4.3 touches on how S3 has been implemented to meet the requirements from section 3.2.

4.1 Implementation of Edgeless Engine: Greengrass

In the previous chapter, the main decisions for choosing Greengrass were explained. However, this sections provides a more detail explanation about how Greengrass is implemented. Throughout this passage, the main characteristics of Greengrass are explained, as well as how they were applied during the development of Edgeless. Therefore, to understand the internal mechanism of this engine, it is crucial to discuss the concept of components.

4.1.1 High-Level Overview of Greengrass

Greengrass is the optimal engine component in the design of Edgeless. However, it is still important to highlight five main technical characteristics from the service, as shown in Figure 4.1, just to have a holistic understanding of how it operates internally.

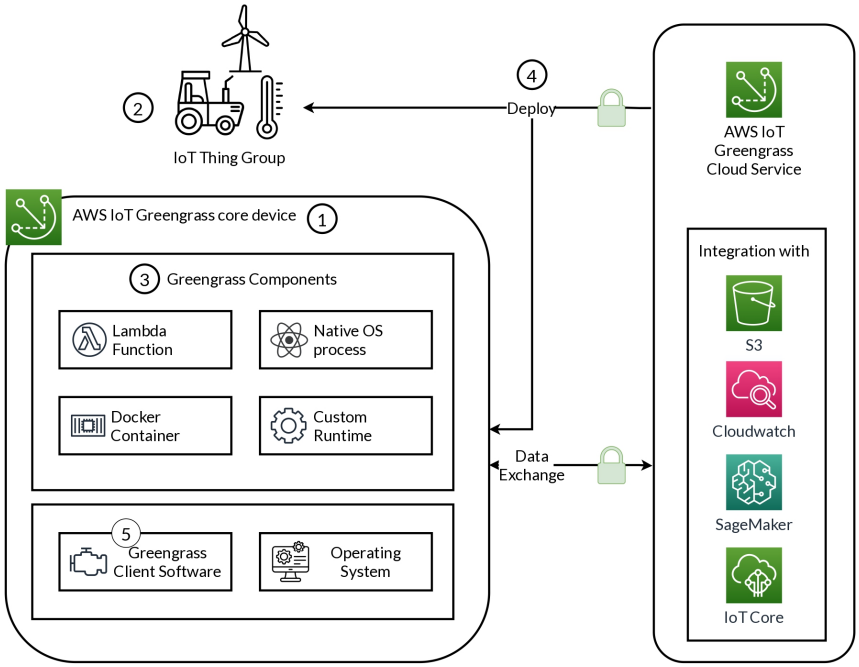

Figure 4.1: AWS IoT Greengrass device is composed by: (1) Core device, (2) Client device, (3) Greengrass Components, (4) Deployments, (5) Greengrass core software.

1. **Core device.** This is essentially the device that runs Greengrass core software. This device plays a central role in extending AWS services to the edge of the network. This is where Greengrass actually performs the computation. Furthermore, there are a variety of platforms that can be supported during the installation process, including Linux (Arm, x86_64) and Windows (x86_64, Windows 10).

2. **Client device.** This is a device that connects to the edge network and communicates with the core device over MQTT. In particular, this can be seen as the microphone sensor and Pico that are deployed closer to the plants. Consequently, the main responsibility of the client device is to push data to the channel state so that the core device is able to process, filter and aggregate it before sending to the cloud.

3. **Components.** This is a software module that is deployed and runs on a Greengrass core device. Every software that is deployed to the core device can be seen as a component. Generally, they have three resources that make them interact with the Greengrass core software: (1) **Recipe:** a JSON or YAML file that describes the software module by defining component details, configurations and parameters. (2) **Artifact:** the source code, binaries or script that define the software that will run on the device. (3) **Dependencies:** the relationship between components that make

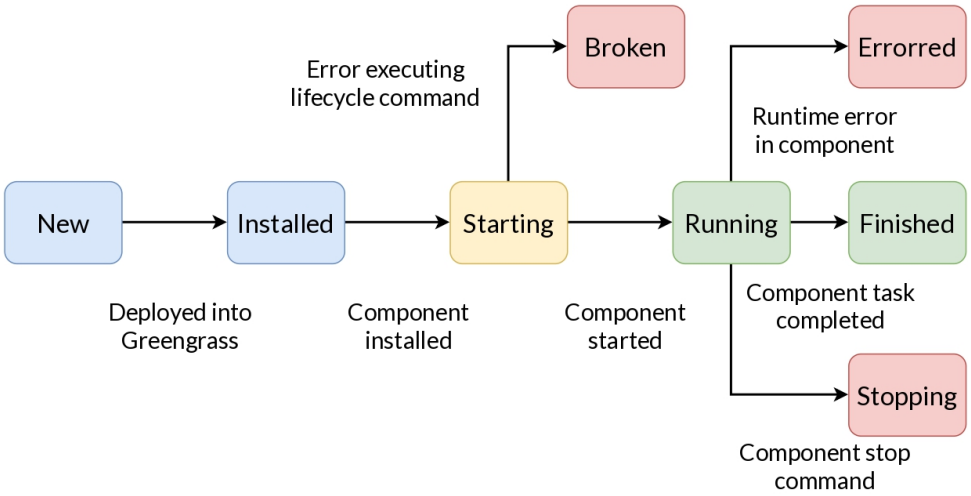

Figure 4.2: Greengrass component lifecycle processes

them work together in the cohesive system. For example, there can be a secure message processing component that is dependent on an encryption component.

4. **Deployments.** This is the process to send and apply the desired component configuration on the core device. Deployments automatically apply any updated component configuration to the target core device, and include any other component that is defined as dependencies. Furthermore, deployments are considered continuous, meaning that any configuration updates made to the component will be reflected in the core device. Lastly, many devices can be associated with a single deployment, making component updates very manageable to roll out to a fleet of devices.

5. **Core software.** This is the set of all AWS IoT Greengrass software that is installed in the core device. It comprises the following: (1) **Nucleus:** this is a required component that provides the minimum functionality of the AWS IoT Greengrass Core software. It manages deployments, orchestration, and lifecycle management of other components. Furthermore, it also facilitates communication between AWS IoT Greengrass components locally on an individual device. (2) **Optional components:** these components that can be deployed to the device as additional features. They are very specific to the AWS services, but can potentially cover areas such as data streaming, local machine inference learning or local command line interface.

4.1.2 Component Lifecycle Management

When components are deployed in the core device, there are multiple phases to evaluate from the moment that it gets installed by Greengrass core software to the moment it finishes running. These steps are crucial to understand the main lifecycle mechanism for deployment, which helps for logging requirement **FR-6**. Figure 4.2 visually shows a picture of the processes that components go through when being deployed at the edge.

1. **New.** This is the initial state when a component is deployed to a Greengrass core device. It indicates that the component has been recognized by the system.

2. **Installed.** In this state, the component has been successfully installed on the device. The necessary files and dependencies are in place, but not started yet.

3. **Starting.** The component is in the process of starting. Initialization operations, such as setting up configurations and establishing necessary connections, are performed.

4. **Broken.** This state indicates that there is a non-recoverable issue with the component. The component requires intervention to fix the underlying issue.

5. **Running.** The component has successfully started, and it is now executing as expected. It is fully operational and interacting with other components or services.

6. **Errored.** An error has occurred during the execution of the component. This state indicates a recoverable error, which might attempt to recover and continue execution.

7. **Finished.** The component has completed its execution successfully. It has performed all its tasks and has now stopped running, waiting for potential redeployment.

8. **Stopped.** This state indicates that the component has been intentionally stopped, either manually by an operator or programmatically through the Greengrass CLI.

The lifecycle of a component is important for understanding their state when deployed in the core device. Therefore, detail implementations about the specific components in Edgeless ought to be discussed in the following subsequent sections, with two different categories to cover. First, the AWS managed components and then after the custom components which run the execution environment for signal processing.

4.1.3 AWS Managed Greengrass Components

These are components that are managed directly by AWS. Some of these are essential for running the Greengrass software. However, some other are add-ons components that can easily be included during deployment. These extra components provide essential functionalities that are key for meeting the requirements 3.2. During Edgeless development, a total of seven AWS managed components were used to meet all the functional requirements.

1. **Greengrass Nucleus.** This is the core runtime component of Greengrass V2. It manages the lifecycle of other components, handles deployments, and facilitates communication between components and the cloud, ensuring operation even if there is network disruption. Furthermore, this component acts as the central control unit, coordinating the execution of all components while also managing their interactions.

2. **Greengrass CLI.** The CLI component permits the interaction with the Greengrass nucleus and other components directly from the command line, providing a way to manage and debug component creation and deployment locally on the Greengrass core device. It enhances productivity as it simplifies management and improves the debugging, making it less complex to develop and deploy components at the edge.

3. **Log Manager.** This component manages the logs generated by the Greengrass nucleus and other components. It can be configured to send these logs to various AWS services, such as CloudWatch Logs, for better persistency and durability. Furthermore, this component enables requirement **FR-6**, since it provides visibility for logging messages about specific executions about signal processing.

4. **Log Debug Console.** This component provides a debug console to view the current status of Greengrass components in real-time. This component is a process running locally in port 1441 and 1442. Furthermore, it provides a local MQTT client for testing communication among components and also a dependency graph for displaying the relationships among components deployed in the core device.

5. **Lambda Manager.** This component is responsible for orchestrating the execution of Lambda Runtimes on a Greengrass device. It handles the deployment, initiation, and termination of Lambda functions, ensuring that they operate efficiently and reliably. Furthermore, it continuously monitors the health of Lambda functions, ensuring that they are working correctly and taking corrective actions if necessary.

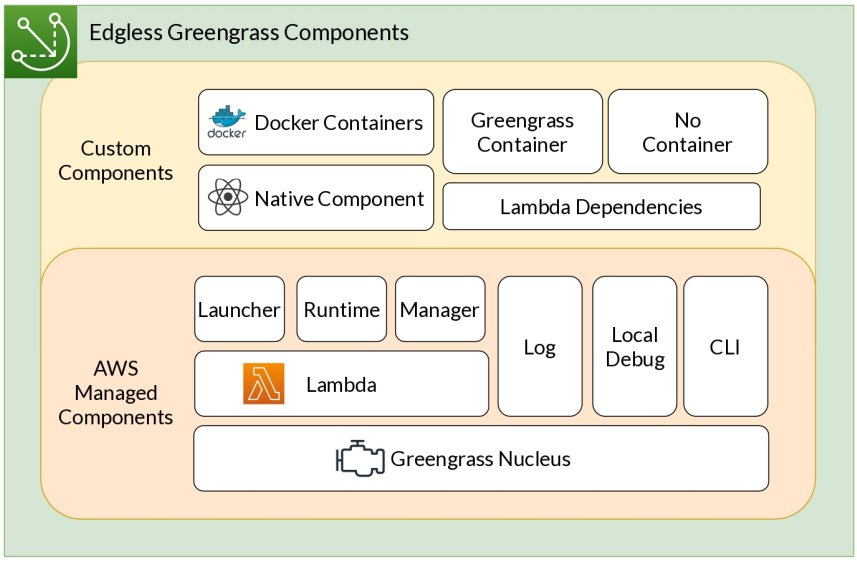

Figure 4.3: Greengrass core device component decomposition.

6. **Lambda Launcher.** This component facilitates the execution of Lambda functions by launching and running them as per the configurations and triggers defined such as MQTT messages, local events, or predefined schedules. It works in conjunction with the *Lambda Manager*, which handles the overall management and scheduling of Lambda functions. This component takes care of the actual execution of code.

7. **Lambda Runtimes.** This component provides the necessary runtime environment to execute Lambda functions on the Greengrass core device. Furthermore, the component helps manage the resources used by Lambda functions, such as CPU and memory. Additionally, this component works closely with the *Lambda Launcher* since it listen to events to create new environments for functions to execute code.

4.1.4 Custom Greengrass Components

Unlike the previous ones, these components are built by developers to be later deployed in the device. Some of these components have dependencies that allow them to perform a particular functionality. During Edgeless development, there were two custom components used, primarily to experiment the different execution environments (native os process and lambda function) for the signal processing application.

Native Component

A native process in Greengrass V2 is a custom executable that runs directly as an OS process on the Greengrass core device. The advantage of adopting this component type is the control and flexibility over the execution environment and behavior of native processes. In other words, since this component is directly interacting with the OS of the core device, it has less overhead. Furthermore, these are generally suitable for performance-critical tasks, custom protocols or when there is a requirement for control over the execution. As mentioned in section 4.1.1, there are two elements that define a component.

1. **Recipes:** these act as the blueprints that define various aspects of the component, such as its identity, configuration, dependencies, and lifecycle. It ensures that components are correctly deployed and, configured within the Greengrass core device.

2. **Artifacts:** these are the essential files where the main application logic resides, i.e. the code to run in the device. These can include executable files, libraries, configuration files, scripts, or any other data that the component needs to operate.

Docker Container Component

Greengrass provides the capabilities for deploying Docker containers as components in the core device. It uses the Docker runtime on the core device to manage containerized applications. This component also uses recipes to set the configuration for the container image and dependencies. Consequently, this recipe will include instructions for the Greengrass nucleus to interact with the Docker daemon. The nucleus uses Docker Engine API to pull container images from a registry, run, stop and manage container lifecycles. Therefore, this component is useful as a benchmark comparison with serverless functions.

Lambda Components

Lambda function is a service that was implemented at the edge using two different mechanisms, one with container and one without containers. In principle, it has the same functionalities as the serverless function in the cloud. However, the difference when deploying at the edge is the flexibility to choose which container mode for the function. Furthermore, Greengrass allows the deployment of *Pinned* and *Non Pinned* functions. (1) **Pinned functions:** these are functions running indefinitely until otherwise manually stopped. (2) **Unpinned functions:** these are functions reacting based on events that can trigger them. This combination makes Lambda functions very appealing for edge deployment.

Greengrass Container. By default, Lambda functions in Greengrass run inside a container. Greengrass containers provide isolation between the function and the host,

which increases security. There are a couple of benefits for opting on this route, important for deployment consideration. First, functions have all Greengrass features available. Second, functions do not have access to the deployed code of other functions, even if they run in the same system group, hence providing secure isolation. Third, since Greengrass core software runs all child processes in the same container as the Lambda function, the child processes stop when the Lambda function stops.

No Container. Lambda functions can also be deployed without container mode. There are certain differences with the former isolation mode, which are important for deployment understanding. First, non-containerized functions have read access to the deployed code of other Lambda functions, making them less secure than previous mode. Second, these type of functions do not use *cgroups*, which are kernel features for limiting and managing resources, e.g. CPU, memory. As a result, they do not have a memory limit, as well as no access to local devices and volume resources.

Lambda Dependency Component

Running the signal processing application inside Lambda requires certain packages necessary at deployment. Therefore, this component is responsible for installing all the necessary libraries and binaries for classifying and reducing the incoming signal from sensors. Dependency components are important since they enable certain functionalities used by Lambda components. In this particular case, Greengrass launches this component first, and then launches the Lambda function component, in that respective order. There are three main libraries that are installed and used for signal processing:

1. **Boto3.** This is the AWS SDK library for Python. It is used for interacting with AWS services directly through an interface. After classifying and reducing the signal, it gets saved in a JSON format into a S3 bucket.

2. **Wave.** This is a built-in library that provides a way to work with audio WAV files. It allows reading and writing WAV files, enabling manipulation and analysis of audio data. It used to process and analyze incoming signals from microphone sensors.

3. **Scipy.** This is an open-source library used in mathematics, science, and engineering. It builds on the capabilities of NumPy and provides additional modules for optimization, integration, interpolation of signal and image processing.

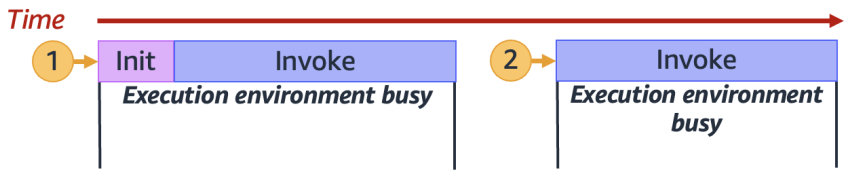

Figure 4.4: Sequential executions with respect to the incoming requests, when a single execution environment handles two requests one after the other. (10).

4.2 Implementation of Edgeless Function: Lambda

Lambda is the optimal function component in the implementation of Edgeless due to its scalability features. Nonetheless, it would be relevant to go into more details on two key points about Lambda scalability: workload executions 4.2.1 and invocation methods 4.2.2. Lambda has many other features, however these are the most relevant for the discussion.

4.2.1 Workload Execution: Sequential and Concurrent

Lambda scalability is very suitable for Edgeless, since functions can handle unpredicted and burst of requests sent by microphone sensors. However, there are two ways of executing workloads in a function, sequential and concurrent. Understanding these two would be crucial for measuring the execution time of the functions, since the Lambda performance and autoscaling features are associated with the kind of workload functions get assigned.

Sequential Executions

Lambda must initialize an execution environment to handle requests, also known as the *Initialize Phase*. Once this environment is ready to use, the function is invoked to handle the incoming requests, also known as *Invocation Phase*. When a function receives its very first request, as shown by the **label 1** in figure 4.4, Lambda creates a new execution environment and runs the code outside the main handler *(Initialize phase)*. After this, Lambda runs the function's main handler code *(Invoke phase)*, which is the one used to process data. During this entire process, this execution environment is busy and cannot process other requests. When Lambda finishes processing the first request, this execution environment can then process additional requests for the same function. For subsequent requests, Lambda doesn't need to re-initialize the environment. As shown by the **label 2** in figure 4.4, Lambda reuses the execution environment to handle subsequent requests.

Concurrent Executions

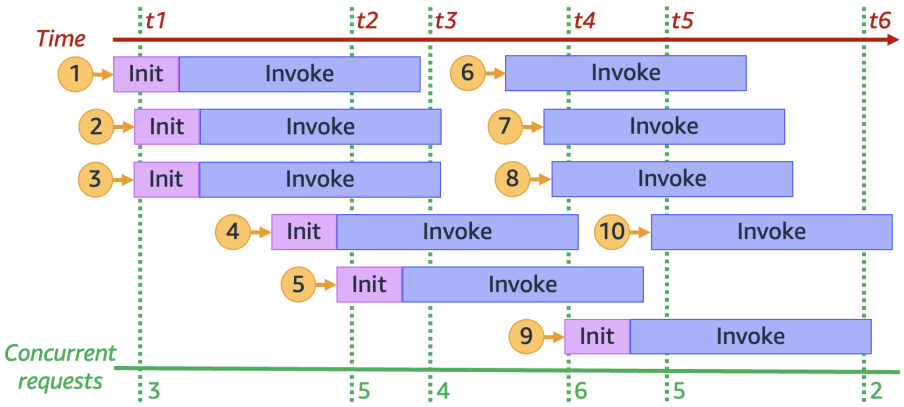

Figure 4.5: Concurrent executions with respect to the incoming requests, when multiple requests are done simultaneously with various execution environments working (10).

Concurrency can first be defined as the number of in-flight requests a Lambda function is handling at the same time. In practice, Lambda may need to provision multiple execution environment in parallel to handle all incoming requests. Therefore, when a function receives concurrent requests at a time, two important decisions are made at runtime:

1. If a pre-initialized execution environment instance is available, Lambda uses it to process the request. Hence, this aids improving the performance of the system, because environments can be reused after shortly finishing the previous execution.

2. Otherwise, Lambda creates a new execution environment instance to process the request. Since Lambda has to build the environment, cold starts tend to happen from the moment the request comes in, to the moment that it gets processed.

As the function receives more concurrent requests, Lambda scales up the number of execution environment instances in response. Figure 4.5 shows what happens when a function receives 10 requests. Vertical lines are drawn at any point in time and count the number of environments that intersect this line. This gives the number of concurrent requests at that point in time. For example, at time t1, there are 3 active environments serving 3 concurrent requests. The maximum number of concurrent requests in this simulation occurs at time t4, when there are 6 active environments serving 6 concurrent requests.

4.2.2 Invocations Methods: Synchronous and Asynchronous

There are two ways of invoking a Lambda function, synchronously and asynchronously. Discussing Lambda invocation methods is essential for understanding some of the key

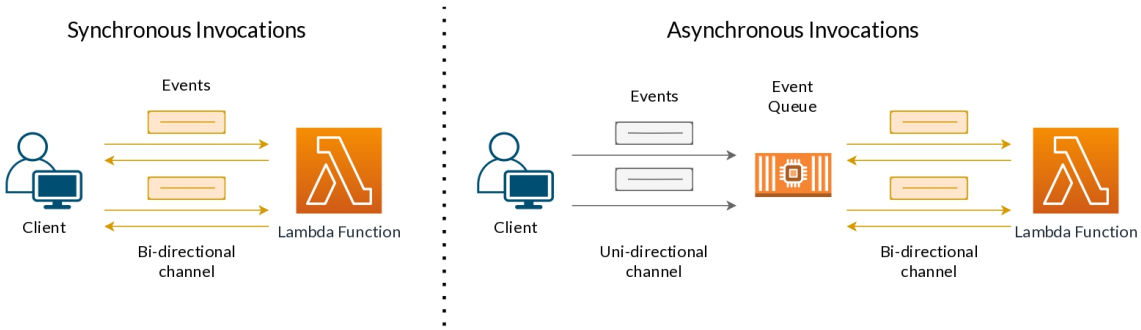

Figure 4.6: Lambda invocation methods: Synchronous, where a dedicated channel is established between the client and function. Asynchronous, where the client does not wait for a response from the function

features of autoscaling. These modes can be applied differently depending on the use case. However, microphone sensors and microcontrollers would ultimately be implemented for sending data asynchronously. Nonetheless, it is important to go over both in more detail.

Synchronous Invocation

When invoking a Lambda function synchronously, clients wait for a response from the function before proceeding to the next operation. When the function completes, Lambda returns a response to the clients over the same dedicated channel. Figure 4.6 shows on a high level process on how this works. In this figure, events are sent by a client, invoking the function to process the request. After doing so, the function sends the response back.

Asynchronous Invocation

When invoking a Lambda function asynchronously, there is no wait time, since a response is not necessarily given immediately. Therefore, once events are handed off to a Lambda function, there is no blocking between the device and the function. For asynchronous invocation, there are a couple of considerations to make when adopting this method.

- **Event Queue.** Lambda places the event in a queue and returns a success response without additional information. As shown in figure 4.6, a separate process reads events from the queue and sends them to a function. Lambda is then responsible for managing this asynchronous event queue to make sure that all events get processed.

- **Retry Mechanism.** Lambda has an internal retry mechanisms in case errors are raise. If the function returns an error, Lambda attempts to run it two more times, with a one-minute wait between the first two attempts, and two minutes between the

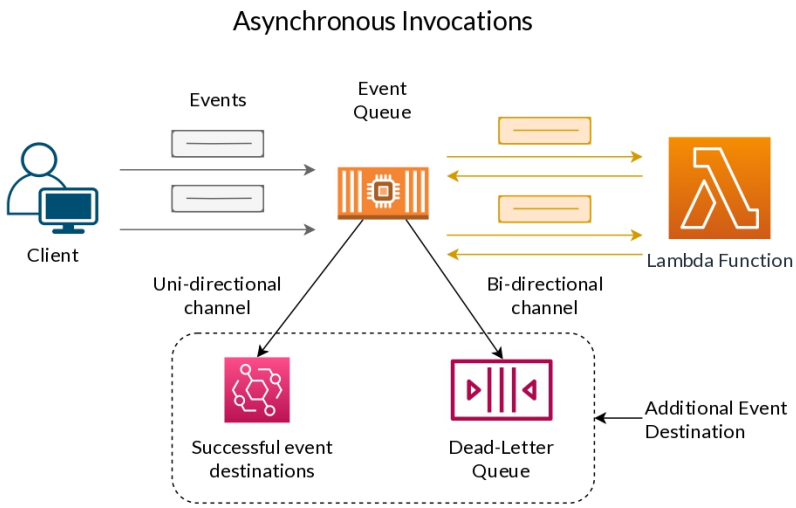

Figure 4.7: Additional destinations where events can go after being processed by a Lambda function.

second and third attempts (10). The retry interval increases exponentially from one second after the first attempt to a maximum of five minutes in subsequent attempts.

- **Expiring Events.** When the queue is very long, new events might age out more quickly, in order to reduce the total processing time. As a consequence, this causes events to expire before being processed by a function. When an event expires or fails all processing attempts, after the retrying mechanism, Lambda discards it. Therefore, alternating destination for failed events will avoid loosing important events.

- **Dead-Letter Queues.** These are often design considerations for distributed asynchronous systems. Similarly, there can be separate services, from Lambda, that can be configured to handle such failed events that do not get processed by a function.

4.3 Implementation of Edgeless Storage: S3

S3 is the optimal storage component in the implementation of Edgeless due to its availability and durability. Nonetheless, it is relevant to discuss two key points about S3 availability: replication 4.3.1 and storage options 4.3.2. There are many features in S3, however these two lay the foundations for understanding the technical storage implications for Edgeless.

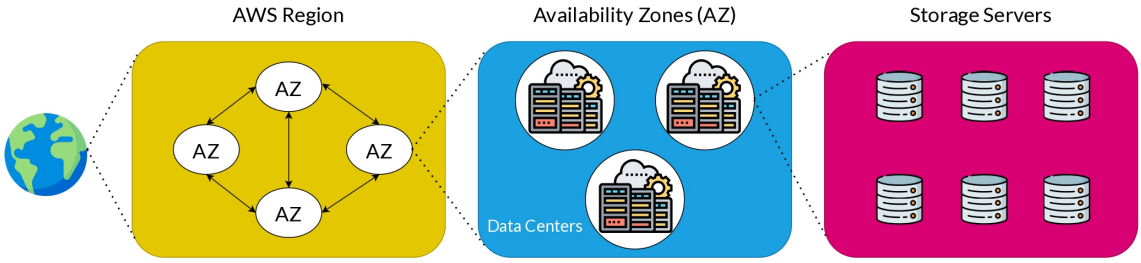

Figure 4.8: S3 data replication process on geographically separated availability zones.

4.3.1 S3 High-Availability: Replication and Versioning

S3 achieves high availability by asynchronously replicating copies of data across multiple servers within AWS data centers. These copies achieve data redundancy, which ensure durability and availability, suitable for **NFR-3** 3.2.2. However, there are two important components for achieving high availability: object replication type, and versioning.

S3 Object Replication Type

S3 offers two different kinds of replications, Same-Region Replication (SRR) and Cross-Region Replication (CRR). Both of them are reliable for storing objects, although they have different specifications that contribute to a variation in data availability and accesibility.

1. **Same-Region Replication.** This mode replicates data across multiple Availability Zones (AZs) within one AWS region. An AZ is a collection of one or more data centers that are physically separated in a metropolitan area. Therefore, when an object is put in a bucket, by default, S3 replicates this object in more than three AZs within the same region (54). Furthermore, S3 maintains redundancy even within a single AZ, by replicating data across multiple disks. Therefore, if there is a failure in one of the disk, objects can still be retrieved. Figure 4.8 illustrates this process.

2. **Cross-Region Replication.** This mode replicates data across different regions. A region is a collection of two or more AZs. At the moment of writing, AWS has launched 32 regions [1] across the globe. Furthermore, leveraging S3 data backups through replication provides an advantage for disaster recovery solutions. Disaster recovery is a measure that can guarantee that data will not be lost (55), even if one entire region fails. Therefore, Edgeless takes into account various levels of replication when storing microphone sensor data, ensuring that no information is loss.

[1] https://aws.amazon.com/about-aws/global-infrastructure/

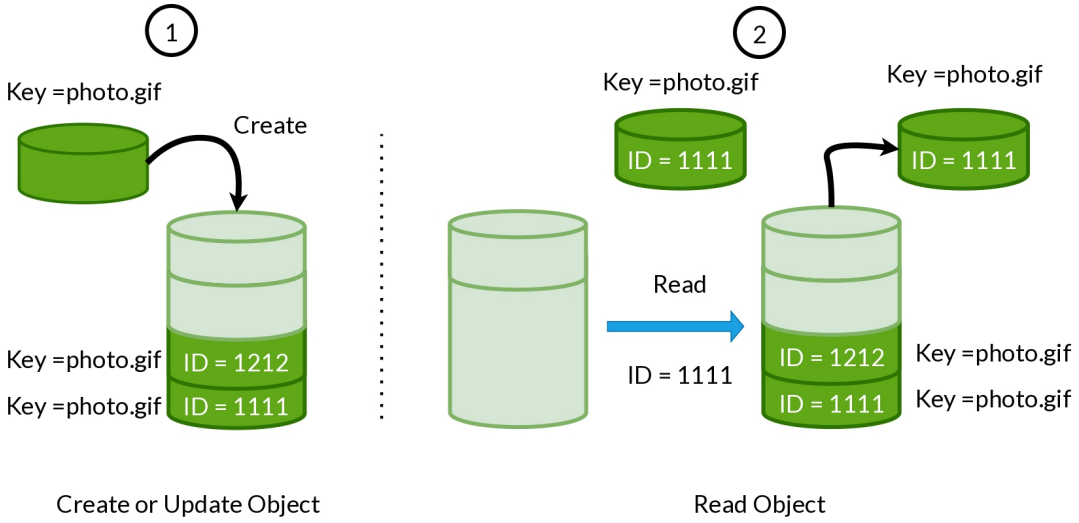

Figure 4.9: S3 versioning on bucket, on (1) create and update object and (2) read object.

S3 Object Versioning

Versioning is another replication feature from S3. It is used to preserve, retrieve and restore every version of an object stored in a bucket. Furthermore, it is a great for recovering objects from accidental deletion or overwrite, allowing to restore the previous version that is available in the bucket. Versioning by assigning IDs to new objects in the following way:

- **Version ID.** Each object has a version ID, which represents a new change state in the object. Initially, this version ID starts as null, but eventually S3 will be generating new version IDs with each corresponding new change that is done in the object.
- **New Versions.** When an object is created or updated inside a bucket (with versioning enabled), the older version of this object is not overwritten. Instead, a new version is created and stacked on top of the older one, like in (1) from figure 4.9.
- **Older Versions.** Additionally, reading the older versions of an object can be done by specifying its version ID when retrieving the object. Just like shown in (2) in figure 4.9, the object ID to retrieve is 1111, which is one of the older versions.

4.3.2 S3 Storage Classes

S3 offers several storage classes, each designed to provide a different availability. Table 4.1 summarizes all the different kind of options S3 provides. Therefore, it is relevant to have some understanding of these classes, as data stored in Edgeless will not always be in the

Service	Storage Class	Durability	Availabilty	AZs
S3	Standard	99.999999999%	99.99%	>=3
	Standard-IA		99.9%	
	One Zone-IA		99.5%	1
Glacier	Instant Retrieval		99.99%	>=3
	Flexible Retrieval			
	Deep Archive			

Table 4.1: Comparison among different S3 storage classes (11)

same storage class. Depending on the longevity of the data captured by the microphone sensors, this data can be moved to a different storage class for cost and logistics efficiency.

1. **S3 Standard.** This class provides high durability, availability, and performance for frequently accessed data. This is the default class when an object is put in a bucket, unless specified otherwise. This class will be used when data first arrives in Edgeless.

2. **S3 Standard-IA** This class is suited for long-lived and infrequently access data. Unlike the former storage class, this one has a lower level of availability. However, just like the former, this class still stores data in geographically separated AZs.

3. **S3 One Zone-IA** This class is also suited for long-lived and infrequent access. However, unlike the previous class, data stored in this class is not as resilient to physical losses against failures. The benefit is the cost-efficiency when storing data.

Additionally, S3 also has another set of classes for archival data. **S3 Glacier** is a low-cost archival storage service designed for long-term retention of data that is accessed infrequently and requires long-term durability. Glacier is suitable for storing data that do not need immediate access. As a result, there are three main classes in this domain.

1. **Glacier Instant Retrieval.** This class is for rarely accessed data that requires milliseconds retrieval. Furthermore, this class offers the same latency and throughput performance as the S3 Standard-IA, but with a higher cost when accessing data.

2. **Glacier Flexible Retrieval.** This class is used when portions of data might need to be retrieved in minutes. Therein, objects have a maximum storage duration of 90 days. Data can be accessed in 1–5 minutes, with expedited retrieval (11).

3. **Glacier Deep Archive.** This class is used for data that does not need to be accessed. Objects have a minimum storage duration of 180 days, and a default retrieval time of up to 12 hours (11). This is the lowest cost for storing data in S3.

5

Evaluation: Setup and Deployment

Edgeless is a prototype implementation for processing microphone sensor data in Precise Agriculture domain. As a result, this chapter presents the experimental setup and deployment to benchmark the throughput performance of the system. This chapter is organized in the following sections: Section 5.1 presents the setup, with information regarding the components used during the experiment. Section 5.2 gives more insights on the type of deployments and the tools used for provisioning the infrastructure in both cloud and edge.

5.1 Experiment Setup

This section explains how the setup is built, detailing a high level overview of the services that were used in order to carry the experiments. Therefore, this section presents how the experiment has been simulated, since the production microphone sensors are still under development. Furthermore, it provides an overview of the cloud services that were used to carry the experiment, as part of **RQ1**. Conversely, details the Greengrass components used during the experiment are discussed in this section, as part of **RQ2**.

5.1.1 Simulation Setup For Capturing Audio Files

The ideal setup would be to have the microphone sensors be sending signals to the microcontrollers, as shown in section 3.3. These microcontrollers, found in the mist (or endpoint) would do some preprocessing before sending the corresponding signal to the next edge layer. However, the production of these microphone sensors are still under development. Therefore, this ideal setup has not been considered for benchmarking the performance of serverless function in both environments. As a result, some readjustments had to be made, which in turn lead to the adoption of simulation as the core of this experiment setup.

Simulation became the most viable option to set up this performance evaluation. Due to experiments that Plense Technology conducted over the course of 2022, there were 1000 files shared as part of the collaboration of this thesis. These files were collected using Petterson M500-384 USB Ultrasound Microphone [1], which allowed Plense to capture signals emitted from plants. These microphones were used for validating their idea, but had no intention of keeping them for the future. Hence, the development of new microphone sensors is still in the process, which limited the ideal setup for sending signals. Therefore, the results from this experiment carried by Plense were used as input for this evaluation setup.

The simulation applied in this experiment is not so different from mimicking the ideal setup. The most important metric of this setup is to capture the time of computation, not the network latency that is generated when transferring data for communication between devices at the edge. In principle, replicating this same experiment in the ideal setup, with the microcontroller and microphone sensor, should not provide such significant result differences. The purpose of this question is to explore findings about serverless performance when deployed at the edge. Hence, the underlying reason is to study whether serverless scalability can be carried to an edge environment, with limiting computational capacity.

5.1.2 Experiment Setup in Cloud Environment

Setting up the environment for evaluating the performance of serverless functions in the cloud required a couple of components. This setup was done on AWS, as it worked best for integrating with the core Edgeless components (3.4). As a result, these components can be divided into three different layers: compute, network and storage.

Compute Layer

This layer correlates with the execution environments for evaluating serverless performance. As part of this setup, two different environments were experimented, serverless functions and docker containers. For serverless environment, there are two lambda functions deployed. Similar, there are two ways of container deployment.

1. **Lambda Functions.** As discussed already in section 3.4.2, this is the main compute engine used in Edgeless. The idea of using cloud Lambda functions in this experiment is to study the scalability feature through concurrency. Therefore, there are two different lambda functions, as depicted in the diagram 5.1. (1) **Sequential:** function that handles requests one after the other one, in a queue style. (2) **Concurrent:** function that can handle requests in parallel, chunks at a time.

[1] https://batsound.com/product/m500-384-usb-ultrasound-microphone/

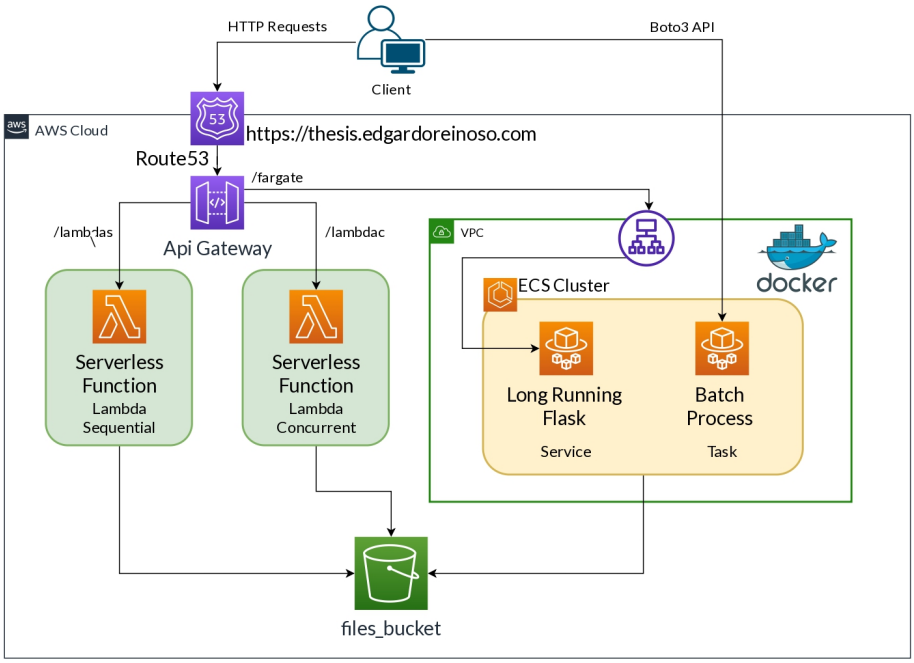

Figure 5.1: Setup for evaluating serverless performance in the cloud

2. **Docker Containers.** The technology used for deploying containers was Elastic Container Service (ECS), where a cluster is created to manage container deployment. Subsequently, the containers run in Fargate [1] mode, which is one of the environment options for deployment in ECS. There are two different kinds of containers run during the experiment. (1) **Batch jobs:** containers are run from a cold start phase. (2) **Flask service:** containers are always run as warm-start in a long-running task.

Table 5.1 presents the predetermined configuration for both compute environments. One common ground is that both environments should be able to run with the same memory allocation capacity, in order to have consistent results. However, it is not possible to configure the amount of CPU in a Lambda Function, since AWS dynamically allocates the amount of cores from the function at runtime, based on the incoming jobs. For docker containers, notice that .25vCPU was allocated as CPU. This means 25% of a CPU core is allocated for running the resource. This amount reflected the average utilization per signal processed, which is not a very CPU intensive operation. Lastly, the Lambda function is set to timeout after 10 seconds, something not applicable in docker containers since they are running a Flask application that is running all the time.

[1] https://aws.amazon.com/fargate/

Compute Service	CPU	Memory	Timeout
Lambda	N/A	512 MiB	10s
Docker	.25vCPU	512 MiB	N/A

Table 5.1: Compute services setup for serverless performance experiment in the cloud.

Network Layer

This layer corresponds to the services required for accommodating the networking infrastructure for the experiments, e.g. firewalls, ingress, APIs. There are four different services that were used during the setup for this cloud experiment.

1. **Route53** [1]**.** This is the scalable and highly available DNS service from AWS. It is used as a way to route traffic to some of the components in the compute layer (Lambda function). A DNS Alias record is created which would bind the API gateway endpoint with the domain name. As a result, a custom subdomain was used for invoking the Lambda functions, in this case *https://thesis.edgardoreinoso.com*.

2. **Api Gateway** [2]**.** This is the API service from AWS. It is used as the gateway mechanism for mapping the Lambda functions to specific HTTP resources. In particular, two resources are created: lambdac for concurrent (/lambdac), and lambdas for sequential (/lambdas). Incoming POST requests (with respective payload) would flow from the DNS service through this gateway to finally arriving at the lambdas.

3. **Elastic Load Balancer.** This is the load balancing service from AWS. It is used during the experiment as an ingress for the service that is deployed in the docker container. Since this service is a flask application, it requires having an internet facing endpoint, which can help serve traffic to the underlying containers. This works directly with the API gateway, since it listens for HTTP post requests.

4. **Virtual Private Cloud (VPC)** [3]**.** This is a logically isolated network within the AWS cloud where resources are provisioned. It is usually composed by subnets, which is a smaller network within the VCP, and security groups, which are firewalls that control the inbound and outbound traffic. For the purposes of the experiment, it is used as the sub-network where containers and load balancer are provisioned.

[1] https://aws.amazon.com/route53/
[2] https://aws.amazon.com/api-gateway/
[3] https://aws.amazon.com/vpc/

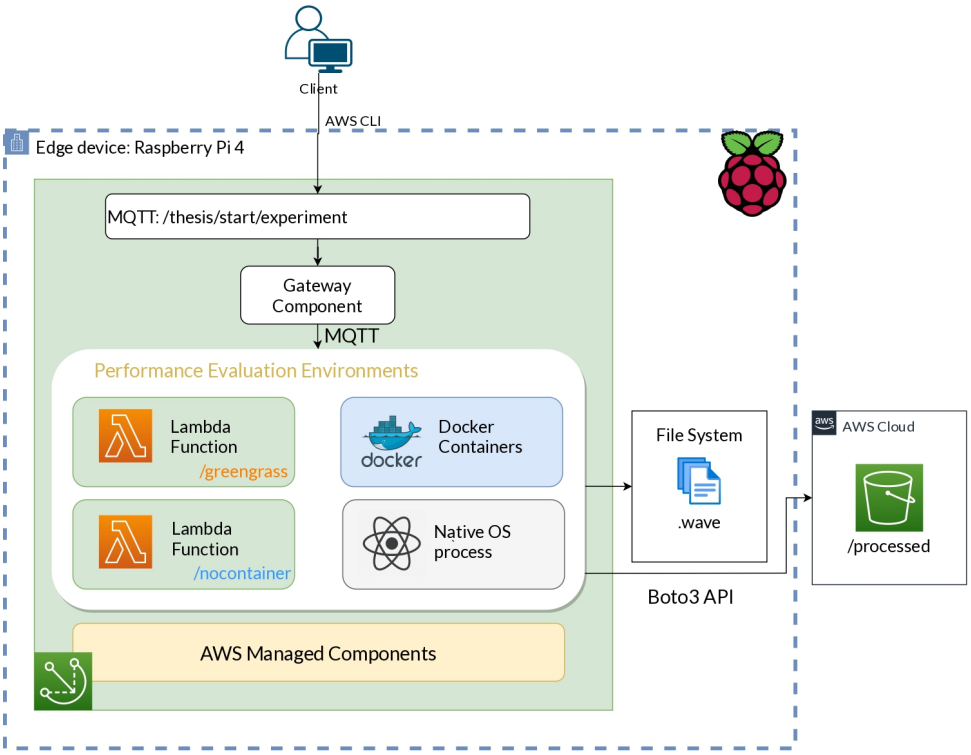

Figure 5.2: Setup for evaluating serverless performance at the edge

Storage Layer

The last layer in this cloud setup is the storage. As discussed in section 3.4.3, the main storage implemented in Edgeless is S3. An S3 bucket is used to find all the audio files. As part of the simulation process 5.1.1, these audio files were placed in a bucket in order to be downloaded when the experiment starts. Therefore, each of the runtime environments, serverless and containers, would download the file to be processed from the bucket by calling the S3 Boto 3 API. This bucket would be divided into two directories: (1) Unsorted directory: it holds are all the files that were collected during the experiment from Plense. (2) Classify directory: it holds the signals that have been processed by the runtime environments, according to the classification algorithm (discussed in section **??**). The process for downloading the files mimics the microphone sensor pushing data to the device.

5.1.3 Experiment Setup in Edge Environment

Setting up the environment for evaluating the performance of serverless functions at the edge required two layers. The compute layer where there are three Greengrass components,

Device	CPU Architecture	Memory	OS
Raspberry Pi 4	1.8 GHz Quad core (ARM v8)	4GB SDRAM	Debian

Table 5.2: Raspberry Pi specifications used during the experiment of serverless performance.

serverless, docker and native, all discussed in detail in section 4.1.4. Furthermore, it also requires a storage layer, where the Raspberry Pi file system and the S3 service are used. Detail information about the Raspberry Pi specifications can be found in table 5.2. For this experiment, a 1.8 GHz Quad core CPU was used, with memory of 4GB SDRAM. The operating system running in the Raspberry Pi is a Debian.

Compute Layer

For this experiment, there are three different execution environments for experimenting the serverless performance. Serverless functions are deployed in two modes, Greengrass container and no container. Docker containers are deployed as components, similar to last experiment. Lastly, Native components are also deployed as a process running in OS.

1. **Lambda Functions.** Two different lambda functions will be deployed as part of the experiment. One function will be running in a **Greengrass container** mode. Another function will be running in a **No container** mode, both explained in detail in section 4.1.4. In addition, both of these function modes were further experimented with pinned and non pinned deployments. Pinned functions are long-running processes. Nonpinned functions are invoked in an event-driven manner.

2. **Docker Containers.** Similar to the previous experiment, docker containers are also carried to the edge. The purpose of having docker containers is to have a performance comparison. This provides some interesting results about how much platform overhead impacts the time for signal processing. Lastly, this component will be running as a long-running process. As a result, it will only be compared with other long-running execution environments for fairness during the experiment.

3. **Native OS Process.** This component is introduced with the purpose of analyzing the performance of long-running, bare metal executions. As this component is directly running as an OS process, it does not require any containerization. Therefore, incorporating this runtime would be beneficial for understanding platform overhead, i.e. Lambda and Docker. Similarly, this component is a long-running process, which can only be compared with other runtime options that have similar behaviors.

Gateway Component

This is a separate component from the compute layer, with relevant importance in the setup for the edge. This component handles the communications for the experiment with the runtime discussed above, i.e. serverless, docker and native. Furthermore, it acts as the endpoint (or the microcontroller) which simulates the data injection to the compute layer, which can be interpreted as the edge. Therefore, this component is responsible for capturing the time taken for a request to be processed by the corresponding runtimes, i.e. throughput operation. First, it communicates with the compute components using AWS IoT Core MQTT. The compute components would subscribe to a topic, for which they receive messages. These messages, sent by the gateway component, contain a reference (name, directory) to the signal, located in the local file system of the Raspberry Pi.

Storage Layer

Lastly, there were two different storage options that were used to set up this experiment. (1) Raspberry Pi Filesystem: this is where the unprocessed audio files will be placed. As discussed previously 5.1.1, a simulation has to be done due to the current state of development of the microphone sensors. Therefore, files were placed in the local filesystem from the Raspberry Pi. The runtime environments would then locate and process these audio files directly from the Raspberry Pi. (2) S3: similar to the previous experiment, a bucket will be used for all the processed audio data. These would pertain to be the signals that have been classified and reduced as part of the algorithms discussed in section **??**.

5.2 Experiment Deployment

The deployment for carrying the experiment is another important component that requires further explanation. The goal behind deploying the setup was efficiency and little manual intervention. As a result, a new methodology was adopted for resource deployment, commonly known as *Infrastructure as Code (IaC)*. In particular, a specific framework was used for configuring AWS resources in this experiment, *Serverless Application Model (SAM)*.

5.2.1 Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is a methodology that uses code and automation to manage and provision IT infrastructure resources such as servers, networks, and databases. Instead of manually configuring these components, IaC allows defining infrastructure requirements through code, usually with JSON or YAML files. Furthermore, this code is

version-controlled, which makes it easily reproducible. As a result, automation of resources leads to more efficient, consistent, agile, and scalable infrastructure management. There are several principles that were considered during the deployment of the experiments.

Automation. It plays a pivotal role in enabling continuous integration and continuous deployment (CI/CD). IaC allows changes to be automatically tested and deployed, ensuring that code changes are seamlessly integrated into the infrastructure. In turns, this facilitates faster and more reliable software releases. As a result, the goal was to create automation when managing resources during this experiment setup.

Version-Controlled. Versioning IaC files is crucial for ensuring stability, collaboration, and reproducibility in this infrastructure setup. Furthermore, by associating a specific version of the IaC code with a particular deployment, that exact infrastructure setup can be replicated at any point in the future. For this case, a github repository is as the source of version control, recording infrastructure changes in the commits.

Idempotence. An operation is considered idempotent when it can be applied multiple times without changing the initial result. In IaC, idempotence ensures that the infrastructure configuration stays reliable and predictable. For instance, if a web server is set up with with an IaC script, no matter how many times this script is run, it will not cause issues if no changes have been made to the initial server configuration.

Scalability. In IaC, scalability ensures that systems can handle varying workloads efficiently. By defining a desirable state of resources, the underlying mechanism for the IaC engine makes sure to meet the corresponding demand. Therefore, resources can be automatically scaled up or down depending on the configurations in the IaC script. Furthermore, replication of additional resources become more manageable.

5.2.2 Serverless Application Model (SAM)

Serverless Application Model (SAM) is an open-source IaC framework, developed by AWS, specifically designed for deploying and managing serverless applications. The reason for choosing this deployment framework was due to the integration capabilities with the AWS resources used for the experiments. Furthermore, SAM provides simplified syntax and faster deployment for serverless applications. However, understanding CloudFormation is essential to grasp how this IaC mechanism works for deploying the services in this setup.

CloudFormation as Deployment Engine

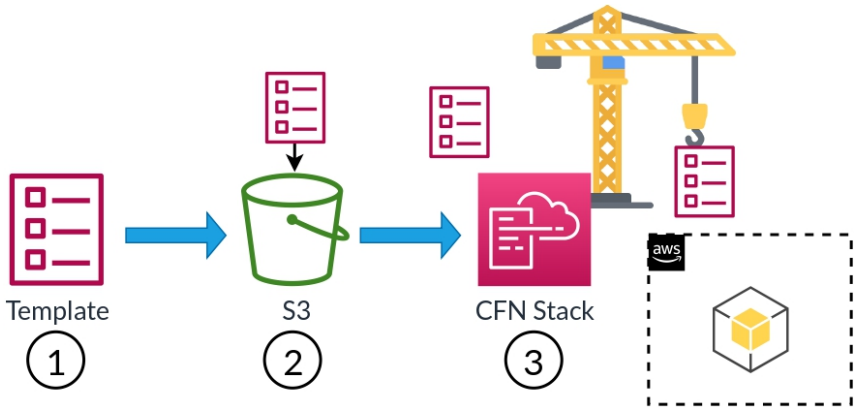

Figure 5.3: Internal process for deploying resources in AWS using CloudFormation

CloudFormation (CFN) is used for provisioning and managing resources using IaC, in AWS. CFN is the underlying engine that actually handles resource deployment in the cloud. Therefore, SAM comes to be an abstraction layer that sits on top of CFN. This layer makes it inherit some of the components from CFN that are used for infrastructure deployment. Therefore, it is beneficial to study the following two concepts on a high level:

1. **Template.** This is a declarative code file, usually formatted in JSON or YAML, that describes the intended state of all the resources to deploy resources in the cloud.

2. **Stack.** This is a collection of AWS resources that can be manage in a single entity. Usually, these resources would be defined in a template. CFN ensures that all stacks are created or deleted appropriately, contributing to the idempotent benefit of IaC.

Figure 5.3 shows a graphical representation of this process. (1) A template would contain all the logical resources that are defined for deployment. (2) The template is uploaded into an S3 bucket, for locating the infrastructure configurations. (3) CFN compiles this template into a stack, which later gets deployed into the cloud.

5.2.3 Experiment Pipeline in Cloud Deployment

A deployment pipeline was an essential component during the execution of the experiment. It allowed resources to be provisioned and destroyed dynamically, which actually made the whole project very cost-efficient. Nonetheless, figure 5.4 shows the directory structure of how the cloud infrastructure was divided, showing two components, code and templates:

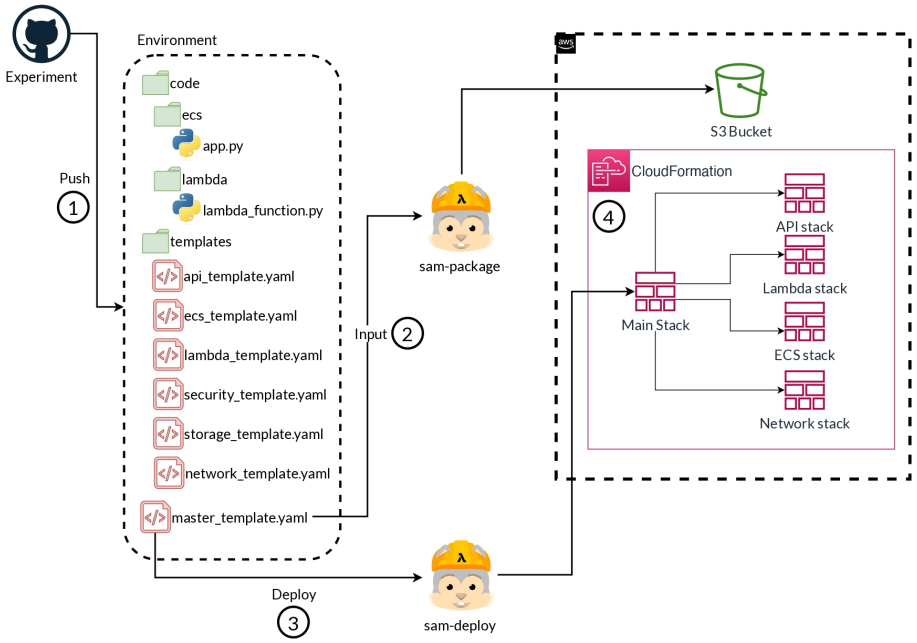

Figure 5.4: Deployment arrangement for cloud experiment setup using SAM

1. **Code directory.** It contains the main business logic that handles processing of the incoming signal. There are two different subdirectories within it, one for the Docker containers, running in Fargate, and one for the serverless Lambda functions.

2. **Template directory.** It holds all the CFN templates (written in YAML) that encapsulate the resources needed for experiment deployment. For better granularity, these templates were split into different files, each attaining their own responsibility.

Figure 5.4 depicts the specific steps in this cloud pipeline process. During this process, Github actions were configured to trigger changes to the infrastructure on every push. (1) **Git push:** whenever there is a git push with some new changes, there is an immediate build which would apply the changes to the infrastructure, using SAM CLI. (2) **SAM build:** any changes, either the code or the templates, will be recorded, built, and later pushed into the S3 bucket. (3) **SAM deploy:** After building, all the resources would be deployed into separate stacks, each of them pertaining to their own responsibilities. (4) **Stack change:** once the changes have been made ready for deployment, CFN provisions the resources by calling the APIs corresponding to the resources defined in the templates.

5.2.4 Experiment Pipeline in Edge Deployment

The components at the Edge have somewhat of a different deployment mechanism than the cloud. Although there are specific elements in CloudFormation for updating and deploying Greengrass components, these were not adopted during the early phases of the experiment. Therefore, two alternative solutions were implemented for updating components in the core device. The code for both of these can be located in the GitHub repository of this thesis.

1. **Flask application.** A web framework was configured for handling all component-related operations (CRUD) in the core device. There were three main methods used, one for the lambda component, one for the non-lambda (gateway, docker, native) component and one for the deployment. This actually helps make component updates much more manageable, due to the API integration, and less manual intervention.

2. **Shell scripts.** There were three scripts (lambda, non-lambda and deployment) used for updating and deploying components, working in conjunction with the web framework. These scripts take the JSON configuration of the compute components as input, which is essentially their recipe for setting up, see section 4.1.4. These scripts call the corresponding component API to deploy new updates in the device.

6

Evaluation: Design and Experiments

Setting up and deploying the environments were the first couple of steps in the evaluation. However, there are still some experiments that need to be design and executed in order to find meaningful results that would provide answers to the research questions. As a result, Section 6.1 provides more information about the type of workloads that are carried in the experiment, as well as the calculation for concurrency. Section 6.2 shows the results from the execution of these experiments, detailing the main findings obtained from these results.

Answering the research questions required executing two different experiments:

- **Throughput Performance in Sequential Workloads.** The system is benchmarked with signal processing requests that are executed one after the other. There are a varying number of requests (100, 300, 500, 700 and 1000) done sequentially by one client. The goal is to measure how fast Lambda functions, and the other runtimes, take to execute these workloads, i.e. requests per seconds.

- **Throughput Performance in Concurrent Workloads.** Similarly, the system is benchmarked with the same requests, but this time parallelly. In other words, there is a set number of requests (100) that is broken into batches (5, 10, 20, 25, 50) which represent the concurrent number of invocations that are done to the system. The goal is to also measure the performance under this type of workload.

6.1 Experiment Design

During the evaluation of Edgeless, two primary experiments (for cloud and edge) were conducted to measure the throughput of different runtime environments. The goal is to explain the behavior of autoscaling benefits from serverless functions, when compared to the

other runtime environments. More specifically, how the effect of concurrent invocations can really strive better performance of Lambda functions in relation to containers, and native OS process. Therefore, this section provides explanation about how throughput is measured, and the type of workloads used, and the design for the edge experiment.

6.1.1 Throughput and Concurrency

The aim of the experiment is to measure the performance from the different runtime in the compute layer for both environments, cloud and edge. Therefore, throughput is measured as requests per seconds (RPS). The number of requests is considered to be the signals that are received from the microphone sensors. As explained in section 5.1.1, the signals would be simulated in the experiment due to the development of the microphone sensors. However, there is an important element about concurrency that needs to be explained, since it provides more understanding on how Lambda function autoscaling works.

Concurrent Execution Estimation

The number of concurrent environment Lambda provisions for autoscaling is the sum of all invocations that can happen in a determined period of time. For example, if a function takes 1 second to run, and 10 invocations are done in parallel, Lambda will create 10 execution environments to process these requests. Therefore, this determines that Lambda can handle 10 requests per second. Therefore, to estimate the concurrent environments from the number of requests per unit of time, the following formula is used:

$$ConcurrentRequests = AvgDurationSecond \times RequestsPerSecond$$

This provides insights into how Lambda can autoscale. For example, if a function takes 5 seconds to run, at 100 requests/second, the number of concurrent requests would be 50. On the contrary, if the function duration is halved to 2.5 seconds, and the number of requests is increased to 300 requests/second, the number of concurrent requests remains the same, 50. As a result, the function duration can really make an impact on how many concurrent execution environments are provisioned, hence improving the throughput measurement.

6.1.2 Experiment Workload Design

For the following experiments, there are two type of workloads used to evaluate the performance for each of the runtime environments. (1) **Sequential:** these types are considered sequential invocations, since signals are processed in a queue. (2) **Concurrent:** these types

Number of Sensors	1	1	1	1	1
Number of Requests	100	300	500	700	1000

Table 6.1: (W1) Fixed number of sensors and several number of requests - Sequential.

Number of Sensors	5	10	20	25	50
Number of Requests	100	100	100	100	100

Table 6.2: (W2) Fixed number of requests and several number of sensors - Concurrent.

are considered concurrent, since multiple threads are span to make parallel invocations. This way, Lambda scales the number of execution environments to meet the demand.

Tables 6.1 and 6.2 explain more about how these workloads would be executed during the experiment. **(W1)** uses a fixed number of sensors, which for the purpose of this experiment would be the gateway component sending data to the runtime environments. On the other hand, the number of requests is the changing factor. This type of experiment is done sequentially, to show the performance of serverless functions in a workflow. **(W2)** uses a fixed number of requests, but the changing factor is the number of sensors. In this experiment, the number of sensors can be simulated as the threads initiated at the start of the experiment. This type of experiment attributes the concurrent invocations, in order to demonstrate Lambda function autoscaling capabilities under a determined stress level.

6.2 Experiment Results

This section will provide the results for the two experiments designed to measure the throughput of serverless functions in cloud and edge. First, the results from **RQ1** are presented, detailing the autoscaling capabilities of cloud Lambda functions in contrast to containers. Once understanding how Lambda scales by provisioning concurrent environments, the question becomes whether these capabilities can be extended to the edge. Therefore, the results from **RQ2** are presented to show similar scaling behaviors in a core edge device, where there are constraints in computational capacity compared to the cloud.

6.2.1 Cloud Experiment Results (RQ1)

As part of the evaluation, there were four fundamental environments (detailed in section 5.1.2) which were treated as independent variables for this experiment. On the one hand,

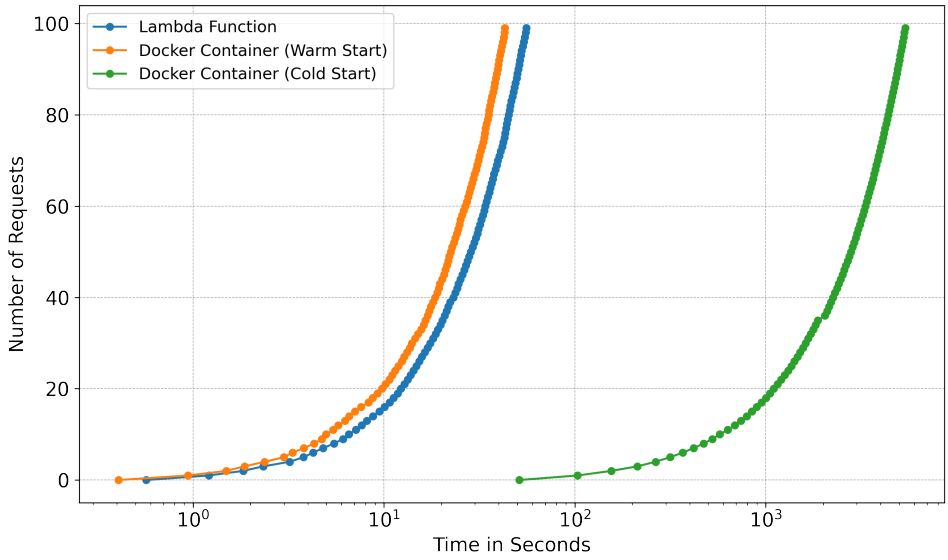

Figure 6.1: Total execution time for **Cloud Sequential Workloads (W1)** with 100 requests from three execution environments: 1. Lambda and Docker (2. cold start and 3. warm start).

Lambda Function	Warm Containers	Cold Containers
55.762	43.056	5378.566

Table 6.3: Total execution time (seconds) for **Cloud Sequential Workloads (W1)** from three execution environment: Lambda, Docker (cold start and warm start).

there are two Lambda functions that handle different workloads, one sequential and another concurrent. On the contrary, there are two different deployments for Docker containers, one as a long-running service and another as a scheduled task, both in the ECS cluster.

Findings W1: Sequential Workloads

Evaluating sequential workloads requires three of the four variables, namely sequential lambda, and the two container deployment modes, long-running service and the scheduled. Figure 6.1 provides an initial result from this execution, with the time in the horizontal axis and the number of requests in the vertical axis. As a result, there are two important interpretations to point out from this graph, relevant to the discussion of throughput.

- **Tasks Docker containers (cold-start).** As noticed in the graph, cold-start containers are the least performing runtime environment mode. This is due to the time taken for ECS to schedule and run the containers. Figure 6.3 shows the container

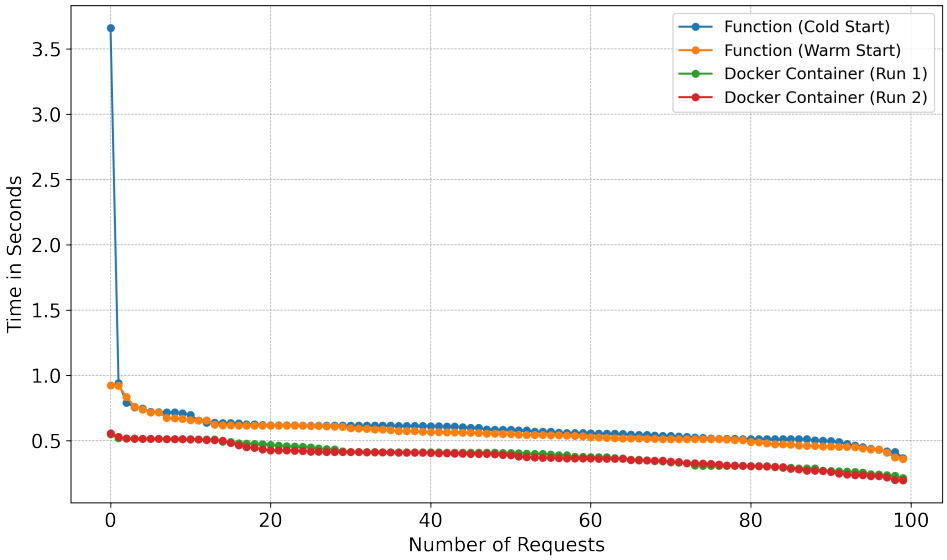

Figure 6.2: Time per execution for each individual request. 100 requests are executed twice by different runtimes, Lambda and Docker Containers. Furthermore, it is visible the dimension of cold start and how much of an impact it may have in the first execution request.

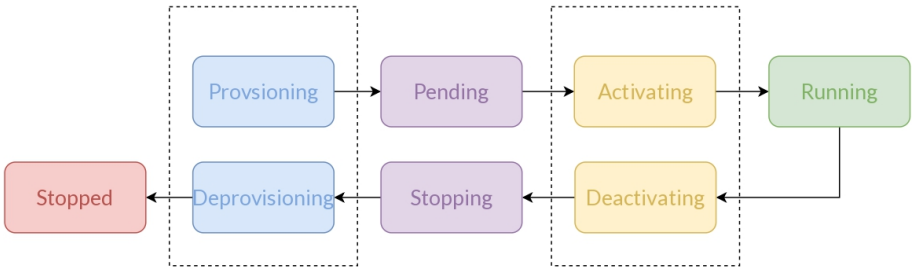

Figure 6.3: Process for scheduling a Docker container in ECS, taking on average between 50 and 60 seconds from the Provisioning phase to the Stopped phase.

lifecycle stages in the ECS task scheduling process. On average, it takes from *50 to 60 seconds* to complete. Therefore, due to lengthy execution process, it is not taken into account as its performance does not provide any useful comparison.

- **Service Docker container (warm-start).** This runtime environment has the best throughput measurement for this subset of the experiment. This is due to the long-running process that is deployed as a Flask web framework (section 5.1.2). Therefore, containers already have a warm start when requests are made. Despite this, it is more beneficial to include due to close performance with the Lambda function.

Figure 6.1 presents the total execution time at the start of the experiment. However, this figure does not really consider the execution of individual functions, limiting the un-

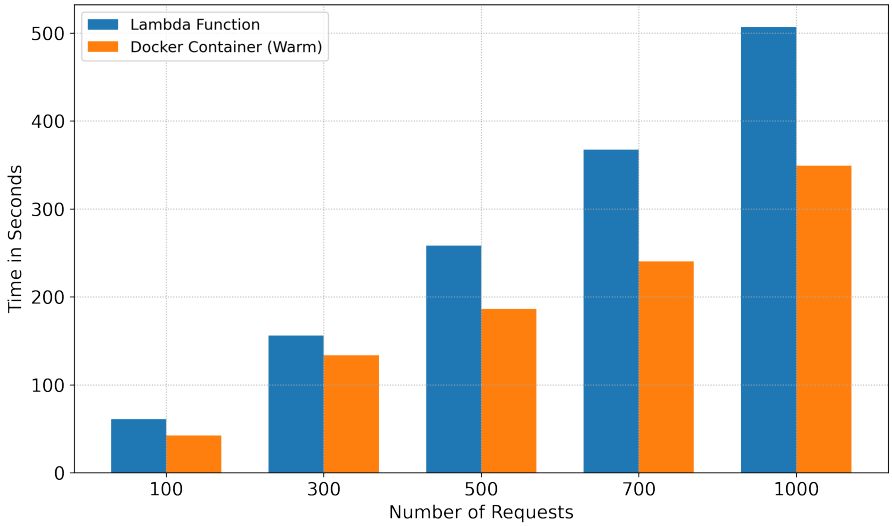

Figure 6.4: Total execution time for **Cloud Sequential Workloads (W1)** with several of requests (100, 300, 500, 700 and 1000) from two runtimes: Lambda and Docker containers (warm).

	100	**300**	**500**	**700**	**1000**
Lambda	61.011	155.978	258.181	367.392	506.915
Docker (warm)	42.280	133.746	186.346	240.381	349.016

Table 6.4: Total execution time (seconds) for **Cloud Sequential Workloads (W1)** with several requests (100, 300, 500, 700 and 1000).

derstanding of the performance impact from cold starts. As a result, figure 6.2 provides a better viewpoint on the time taken by each individual execution. Here, 100 requests are sent sequentially by sensor. Therefore, repeating this experiment twice shows a noticeable difference between the time taken for two runs. During the first run, the first request took about ***3.660 seconds***. However, in the second run, this same request took ***0.608 seconds***. This difference of ***3.052 seconds*** corresponds to the cold start during the run.

Figure 6.4 presents the total execution time based on different number of requests. In this particular case, sequential workloads **(W1)** of Lambda functions are slower than warm Docker containers. Although the difference between cold-start and warm-start may not seem like a fair comparison, Lambda only adds ***1 to 3 seconds*** to provision the execution environment for the first requests. Subsequent requests would reuse the same environment that was initialized. The overhead for scheduling the function is actually what contributes

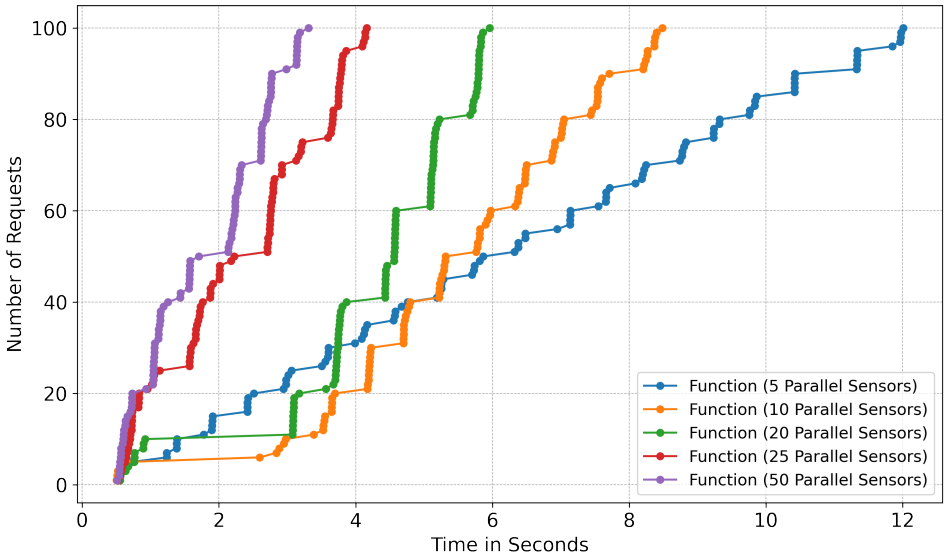

Figure 6.5: Total execution time for **Cloud Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) invoking concurrent Lambda function.

to such a noticeable time difference between Lambda and Containers. Table 6.5 shows the total time difference between Lambda and warm containers. Therefore, cold-starts are really not deterministic when comparing the execution of Lambda and Docker.

Findings W2: Concurrent Workloads

Considering that Lambda functions do not perform well under sequential workloads, it is time to evaluate how concurrent invocations can help visualize the autoscaling benefits from Lambda. Therefore, a separate Lambda function (independent of the sequential) is used for measuring the scalability (number of concurrent execution environments) and throughput. Figure 6.5 shows five runs with different number of sensors and a fixed number of requests. The number of sensors is simulated as concurrent threads that are spun at the start of the experiment. There are two major key takeaways from this particular result:

- **Concurrent threads.** The more concurrent threads are initiated, i.e. more number of sensors, the faster the overall execution time that Lambda takes to respond. As discussed previously, this happens because Lambda automatically provisions multiple execution environments to handle the peak incoming signals from the sensors. Figure 6.6 illustrates this dynamic deployment of execution environments.

- **Line gaps in execution.** Some lines have big gaps in between executions, which have two underlying reasons: (1) *Cold Starts:* this happens because multiple exe-

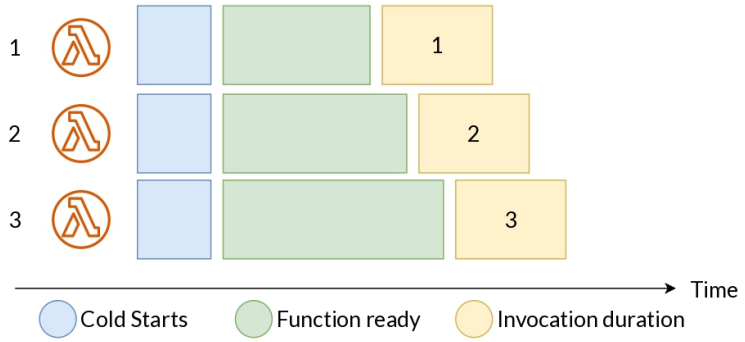

Figure 6.6: Execution environment provisioned by Lambda when invoked concurrently.

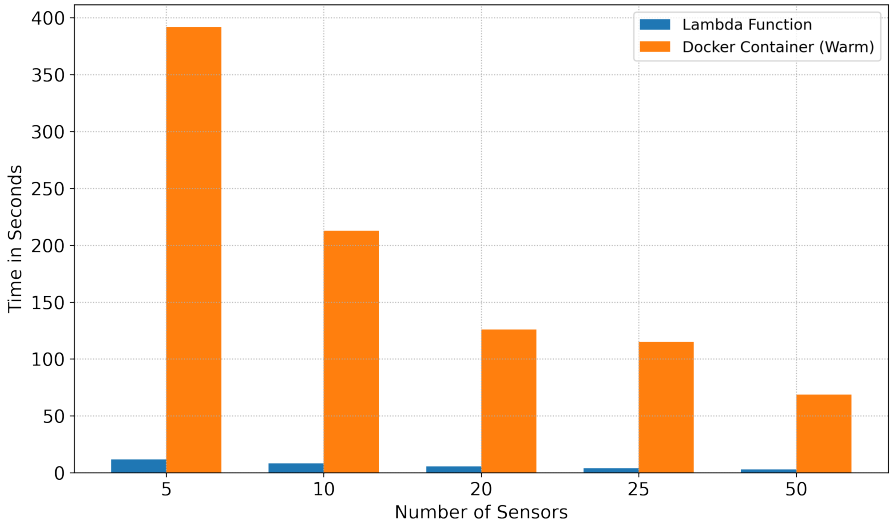

Figure 6.7: Total execution time for **Cloud Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) using two runtime: Lambda and Docker containers (warm).

cution environments are provisioned at a time. (2) *Experiment Conduction:* number of requests is broken down in multiple batches, according to the number of threads. For example, five threads dividing 100 request would result in 20 batches.

Figure 6.7 shows the results when running the experiment with Lambda and Docker containers with concurrent workloads **(W2)**. As seen from the figure, there is a noticeable time difference in the execution of both runtimes. While concurrent invocations using multiple sensors (i.e. threads) have reduced the total execution time in Lambda, it has increased it in containers. This can really show the throughput benefit of Lambda autoscaling effect, since the function is able to process the same amount of requests a lot faster if there are more execution environments. Furthermore, the memory utilization can also

	Sequential	Concurrent (Threads)				
		5	10	20	25	50
Lambda	61.011	11.852	8.263	5.671	4.097	3.137
Docker (warm)	42.280	391.950	212.841	125.979	115.014	68.664

Table 6.5: Total execution time (seconds) for **Cloud Concurrent Workloads (W2)** with 100 requests in parallel with several sensors -threads- (5, 10, 20, 25, 50).

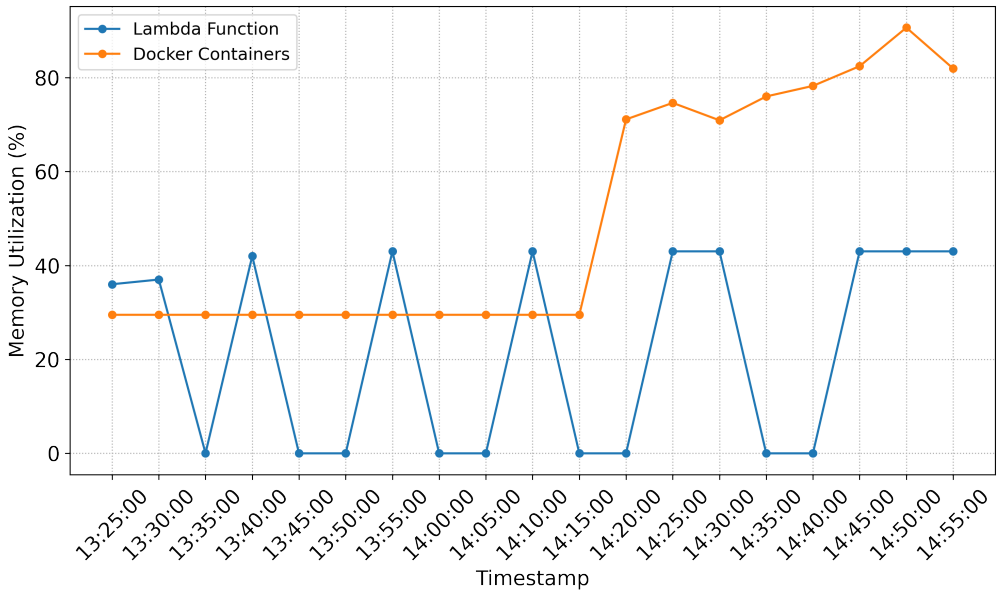

Figure 6.8: Lambda and Docker memory utilization with concurrent workloads.

show (figure 6.8) the difference from both environments. While the concurrent Lambda function is fluctuating due to the event-driven invocation pattern, the Docker container is constantly using memory due to its deployment mode (warm state Flask web framework). In addition, Lambda functions tend to keep a consistent memory utilization even when there are parallel executions. On the contrary, Docker containers tend to use more memory to handle the number of concurrent requests that are done by sensors.

Answer RQ1: What is the performance of Lambda functions in the AWS cloud?

When comparing Lambda functions with Docker containers, using two different type of workloads (sequential and concurrent), some significant results were collected about

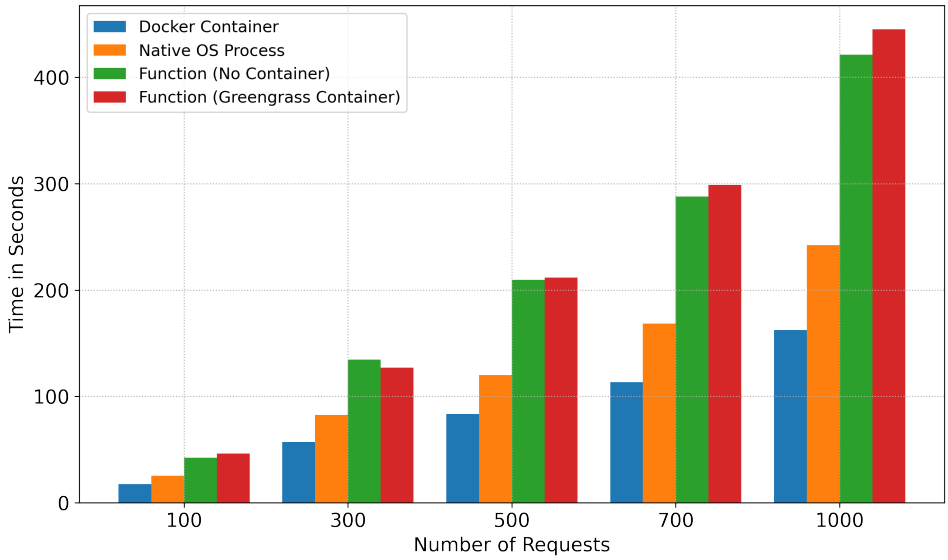

Figure 6.9: Total execution time for for **Edge Sequential Workloads (W1)** with several requests (100, 300, 500, 700 and 1000) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container).

Lambda autoscaling capabilities. In these particular experiments, Lambda function processing time improves with a higher number of sensors (i,e. threads) in the network. The next question would find out whether these scaling benefits apply at the edge, where functions do not run on specialized AWS hardware.

6.2.2 Edge Experiment Results (RQ2)

These experiments required to benchmark four different runtime environments: docker containers, native OS process, lambda (greengrass container), lambda (no container), all discussed in section 5.1.3. The goal is to find interesting results for understanding how Lambda autoscaling works at the edge. Having answered **RQ1**, where Lambda functions perform better with concurrent workloads, the idea of **RQ2** is to find whether Lambda functions can provide the same performance and scaling benefits at an edge device.

Consideration for Non-pinned Lambda Functions

As discussed in the implementation 4.1.4, Lambda functions can be deployed in a Greengrass device through two modes, Pinned and Non-Pinned. During the experiment, the Non-pinned Lambda function configuration was not considered due to two reasons.

1. *Performance Degrading.* When functions start from a cold state, there is more delay

	100	300	500	700	1000
Lambda (No Container)	42.362	83.392	209.649	287.908	421.163
Lambda (GG Container)	46.223	80.888	211.724	298.708	445.286
Docker Container	17.606	32.164	83.466	113.386	162.441
Native OS Process	25.580	48.371	120.156	168.537	242.216

Table 6.6: Total execution time (seconds) for **Edge Sequential Workloads (W1)** with requests (100, 300, 500, 700 and 1000) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container).

when executing the workload, due to the *Init Phase*. Therefore, this negatively affected the throughput for the total execution time when carrying the experiments.

2. *Request Handling.* During the experiments, some issues were encountered with how this type of function handles consecutive requests. The function could not respond very gracefully to sequential calls, most of the time breaking in the process.

Findings W1: Sequential Workloads

Figure 6.9 presents the total execution time based on different number of requests, with each runtime environment. There are a couple of interesting observations from the results of this experiment, which will help understand the throughput performance at the edge.

- **Function Performance.** Just as the results from W1 in Cloud, serverless functions do not provide much throughput benefits when compared to Docker containers or Native OS process. Both of the functions deployed, Greengrass and No Container, have significant performance difference with respect to the other runtimes. However, function with No Container yield a better throughput between both.

- **Docker Containers.** Similar to the experiment in the cloud, containers provide better throughput performance when compared with Lambda functions. Furthermore, they also outperform Native OS Process, an unexpected result considering containers usually add some overhead since code is not executed directly on the hardware. This means in theory they would perform worse, but in practice they were the best.

Findings W2: Concurrent Workloads

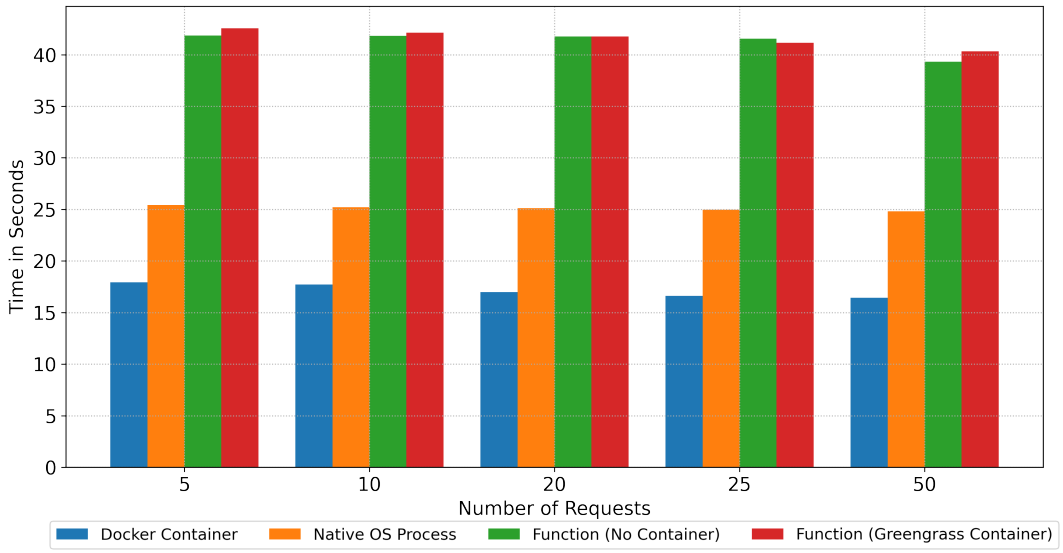

Figure 6.10: Total execution time for **Edge Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container).

	Sequential	Concurrent				
		5	10	20	25	50
Lambda (No Container)	42.362	41.859	41.837	41.753	41.560	39.304
Lambda (GG Container)	46.223	42.569	42.142	41.769	41.155	40.326
Docker Container	17.606	17.915	17.730	16.996	16.627	16.439
Native OS Process	25.580	25.431	25.214	25.105	24.956	24.824

Table 6.7: Total execution time (seconds) for **Edge Concurrent Workloads (W2)** in 100 requests with several sensors -threads- (5, 10, 20, 25, 50) in four runtimes: Native OS process, Docker Container, Lambda (Greengrass and No Container).

Concurrent workloads provided different results at the edge than in the cloud, particularly for serverless functions. While in the cloud, Lambda functions shows a great improvement in throughput performance when invoked in parallel, this is not the case at the edge. Lambda functions do not provide the same autoscaling benefits that were considered in the cloud. Figure 6.10 shows the results of each runtime environments when executed at different concurrent measurements. The most noticeable point to make is the indifference in throughput performance for each of the concurrent threads. In other words, no matter how many threads are initiated in the experiment, there are no apparent differences in the time taken to process requests by any of the runtime environments. There are couple hypotheses, such as multi-threading in Raspberry Pi or the parallel invocations in MQTT, but all required some further exploration that were not considered during this experiment. Table 6.7 shows these results, interpreting that a higher number of threads yields a faster the overall execution time, but only slightly when compared with the cloud experiment.

Answer RQ2: What is the performance of Lambda functions in at the edge?

After thoroughly designing and carrying sequential and concurrent experiments with multiple runtime environments, the results show that Lambda functions do not provide the same scaling performance benefit than in the cloud. As a matter of fact, when considering all the runtimes in both experiment, Lambda functions yield the lowest throughput, i.e. the longest time in processing requests. Therefore, this is an indication that functions do not perform as well in smaller devices with more limiting computational capacity.

7

Limitations

The evaluation of Edgeless has certain cornerstones that should be further discussed. These limitations are considered due to the adoption of AWS services, such as Greengrass and Lambda, but also in the experiment Therefore, it is important to keep these in mind when replicating the setup and the results during the experimentation portion. This section presents the shortcomings of the design, implementation and evaluation of Edgeless.

7.1 Design Limitations: Vendor Lock-In

AWS technologies lay at the core of Edgeless design. Having chosen these suit of services is certainly a soft limitation when analyzing the throughput performance of serverless functions in Edgeless. Greengrass, for example, is a service that is not provided by other cloud services, making it very niche for troubleshooting and debugging. Even though the adoption of AWS has been predominantly rising above other providers, there is still the barrier of flexibly adapting the services used in Edgeless to other environments or platforms.

7.2 Implementation Limitations: Greengrass and Lambda

During the experiment conduction, there were a number of limitations that are important to highlight for the exact reproducibility of the results. They are labelled as hard, since they were technical issues that needed to be left aside, for future work. These issues were mostly related due to combination of edge Lambda Functions when deployed on Greengrass.

7.2.1 Greengrass Component Updates

The internal mechanism to update components in Greengrass is not that efficient. As part of updating new component versions, there are some configuration changes that need to be made before deploying. For example, a component needs a new update to include an MQTT broker, in order to communicate with other components in the network. Making this change is a matter of configuring the permissions in the component to allow the use of MQTT in the code. However, when deploying the component, there were some *Unathorized Errors* encountered when using MQTT, for publishing or subscribing messages to a topic.

7.2.2 Edge Lambda Function Deployment

Despite having some performance benefits, serverless functions at the edge also had certain limitations. These limitations correspond with the Greengrass internal processes when deploying and interacting with the function components. Thus, there are several limiting factors in Lambda functions that are considered during the development of the experiment.

- **Multi-function deployment.** Several issues were encountered when experimenting with multiple functions deployed in the device. Both functions in the experiment had to be deployed separately. Otherwise, one of the functions would have travel subscribing to MQTT messages. As a result, the experiment required each of these to be tested in separate deployments, limiting the results in a real production setup.
- **Sharing underlying components.** As discussed in section 4.1.3, there are several components that Lambda depends on, which are the Manager, Launcher and Runtime. When multiple functions are deployed, they all share these underlying components, which ends up causing a lot of issues, particularly when invoking the function. Therefore, to avoid inconsistent results, functions were deployed separately.

7.3 Experiment Limitations: Multi-thread MQTT Messages.

One of the possible differences in the results of concurrent execution between cloud and edge can be credited due to the MQTT multi-threading messages. During the experiments, some observations were made about the way MQTT carries concurrent invocations. Even though requests were being processed in batches, the responses were being sent sequentially. Therefore, the same protocol to evaluate the throughput performance across the different environments, edge and cloud, would have been beneficial to obtain some interesting results.

8

Related Work

Serverless computing is a technology that has been applied in many areas of academia and industry, since its inception in 2014. Moreover, edge computing is also another field that has covered a lot of ground in scientific literatures, specially since the development of mobile networks and the improvements in microchips and processors. However, it is even more relevant to evaluate literatures that combine both of these concepts, edge and serverless. Furthermore, Precise Agriculture is also a field that needs some study in the literature, specially on how edge or serverless has been combined to create innovative solutions. Therefore, this section focus in the combination of these three fields.

8.1 Edge and Agriculture

These two are possibly the combination of fields which have been explored the longest, comparing serverless, edge and agriculture. Edge computing has made its profound changes in many different industries, including agriculture. This field has evolved tremendously fast due to the rapid advancement in edge technologies.

8.1.1 FarmBeats: An IoT platform for data-driven agriculture (1)

This research paper focuses on precision agriculture, which involves the use of various technologies like sensors, drones, and data analytics to optimize farming practices. It discusses the challenges and solutions related to data collection, processing, and decision-making in the agricultural context. The ultimate goal is to provide insights to farmers on how they can optimize their operations through data-driven approaches. Figure 8.1 provides a visual representation of their architecture.

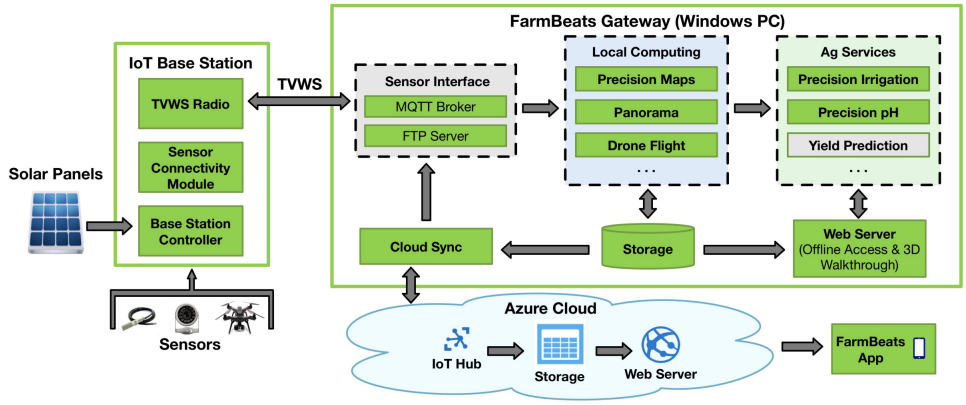

Figure 8.1: FarmBeats architecture for edge processing in drones and other IoT devices (1).

A hybrid technique is used that combines 3D mapping and image stitching methods to create detailed maps of farmland. This process is computationally efficient and can be done at the edge, meaning it doesn't require sending large amounts of data to the cloud for processing. This is particularly useful in scenarios where bandwidth is limited or expensive.

What is the benefit of Edge?

Edge computing in this context allows for real-time data processing at or near the source of data generation (like sensors or drones in the field). This enables quicker decision-making and reduces the need for constant cloud connectivity. For example, the paper discusses the use of Azure IoT Suite for data storage and processing, indicating that some level of edge computing is involved to make the system more efficient.

Trade-off between flying drones at a high altitude to capture a broader view and flying them lower to get more detailed data was discussed. Edge computing can help break this trade-off by allowing for more efficient data processing and decision-making right on the drone or a nearby base station. Therefore, instead of large images sent across the network for further processing in cloud data centers, these are processed locally at the edge.

What is the relation to this thesis?

The design and evaluations that were done in FarmBeats laid the foundations in which Edgeless was inspired on. For starters, the system has a great separation for which data is gathered, processed and analyzed. To this end, in figure 8.1 shows there are three different layers: *IoT Base Station*, *FarmBeats Gateway* and *Azure Cloud*. Similarly, Edgeless is segmented into different layers, which very much align with the classifications of Edge, i.e. mist, edge and fog, as discussed in section 2.2.

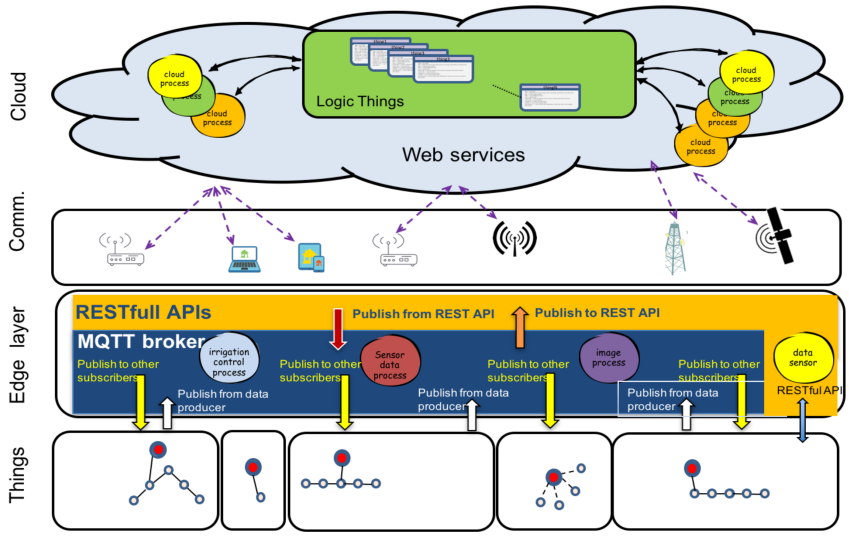

Figure 8.2: Architecture from Ubiquitous Sensor Network in Precision Agriculture (2).

8.1.2 Ubiquitous Sensor Network in Precision Agriculture (2)

This research paper focuses on the application of edge computing in Precision Agriculture. It aims to optimize production efficiency, increase quality, minimize environmental impact, and reduce the use of resources like energy and water. The paper discusses the development and testing of a low-cost sensor/actuator network platform based on the Internet of Things (IoT), integrating machine-to-machine and human-machine-interface protocols.

An experimental greenhouse with hydroponic crop production is used to monitor and control the optimal temperature and soil conditions. This is done by using Ubiquitous Sensor Network (USN) and edge computing on the IoT paradigm. The results show that the integration of technology can encourage the development of Precision Agriculture.

What is the benefit of Edge?

Edge computing is described as a layer between data sources and cloud data centers where machine-to-machine (M2M) protocols, control, data processing, and web services are integrated. It pushes applications, data, and processes away from embedded nodes that control the Ubiquitous Sensor Network (USN) to the cloud. Edge computing provides reliability of response in control processes and is essential for response times and interoperability required for control processes in Precision Agriculture.

Furthermore, edge computing serves as an interface between control processes and cloud services. It enables analytic and knowledge generation at the source of the data. The edge

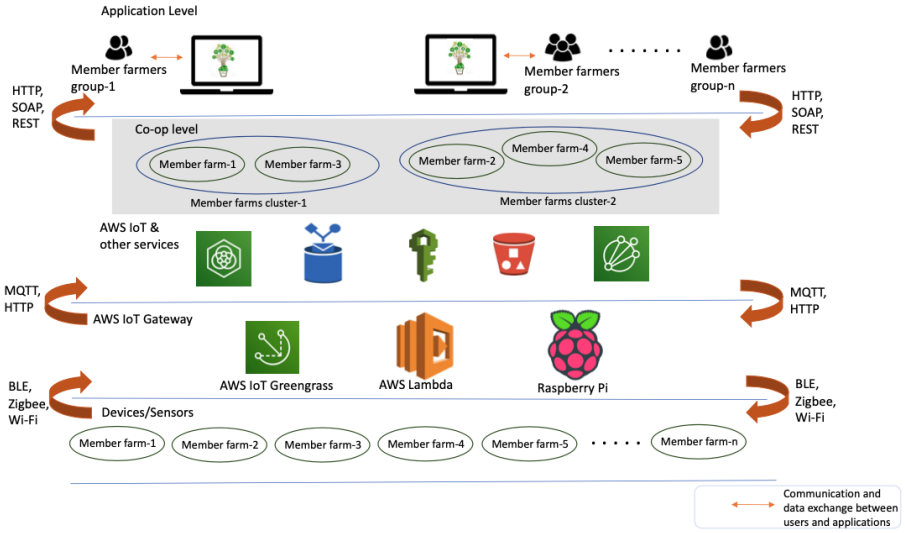

Figure 8.3: Enter Caption

layer is critical for applications and basic control processes, while web services, analytic applications, and other interfaces could be installed in the cloud.

What is the relation to this thesis?

Similar to FarmBeats, previous research work, this paper has influenced the design of Edgeless. There are different layers, as seen in figure 8.2, for which the system gathers, process and analyze data, starting from the lower end devices until reaching the high computing servers in the cloud. Most interestingly, the design of this system adopts MQTT as the protocol for communication between the lower layer and the edge layer. As a result, this paved the way for a similar consideration when deciding protocols adopted in Edgeless.

8.1.3 A Game Theoretic Analysis for Cooperative Smart Farming (3)

This research paper emphasizes that farming, traditionally, has been a community activity, and the advent of modern technologies can enhance the cooperative nature of farming, leading to benefits for all participants. Game theory is employed to model and analyze the strategic behavior of farms in a Cooperative Smart Farming (CSF) environment. This is done to analyze the behavior of this cooperation amongst farmers.

In this game, IoT devices from various farms send their generated data to a centralized cloud. This data then supports all the member farms through Machine Learning (ML) models. The goal of each participant in the CSF game is to maximize their utility, which is a function of the accuracy of the ML models and its associated costs.

What is the benefit of Edge?

Edge computing plays a significant role in the proposed implementation of CSF using AWS cloud and IoT platforms. The proposed implementation utilizes AWS Greengrass, as discussed in 3.4.1, is a service that extends AWS to edge devices, allowing them to act locally on the data they generate. Furthermore, the paper mentions the use of edge gateways, specifically highlighting the Raspberry Pi as an example of a device that can host the AWS Greengrass deployment. Greengrass ensures that IoT devices can respond quickly to local events that are captured from farms of the participants.

What is the relation to this thesis?

The primary reason why this paper was considered as an inspiration for Edgeless was due to its compute component. As shown in figure 8.3, the main service that is used for processing incoming data from farms is Lambda, deployed in the Raspberry Pi using Greengrass. These functions can handle tasks like edge communication and computation, sending notifications, and enforcing access control and privacy policies directly at the edge. This is particularly useful in a CSF environment where real-time data processing and decision-making are crucial. As a result, this paper provided the proper information for validating the assumption on running serverless functions at the edge.

8.2 Edge and Serverless

These two fields have a more recent trajectory when compared to the former, edge and agriculture. Despite, the benefits and positive impacts of serverless computing on scalability and management can be translated to an edge environment. Therefore, recent literatures have studied the works of combining serverless and edge for a variety of use cases.

8.2.1 Serverless management for fog computing framework (4)

The research paper delves deep into the realm of serverless computing and its integration with IoT sensing systems, particularly when combined with fog computing. This approach is especially beneficial for IoT systems, where data is generated in vast quantities and needs to be processed efficiently. Three computational models are introduced in this paper.

1. **IoT and Edge Computing model.** Data is processed right at the source before being sent to a centralized cloud for further analysis or storage, reducing the amount of data transmitted, thereby saving bandwidth and reducing latency.

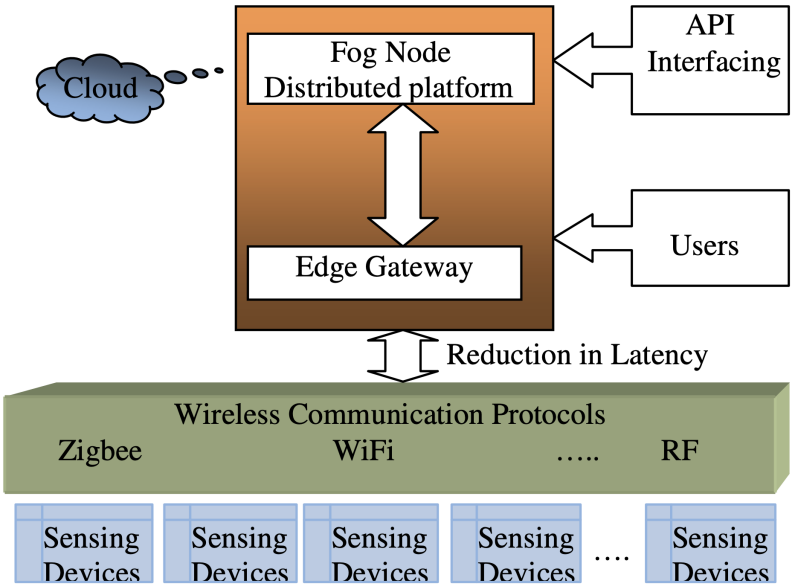

Figure 8.4: Enter Caption (4).

2. **Fog Layer.** Data is first sent to nearby fog nodes before reaching the cloud. This fog node acts as a mini data center, filtering and processing data. By doing this, the system can further reduce latency, making real-time processing more feasible.

3. **Serverless/Distributed Deviceless Model.** User-defined functions run in containers, allowing for a more modular and cost-effective approach compared to traditional Infrastructure as a Service (IaaS) setups.

Experimental environments were set up for validating theories and models. This setup includes edge devices like the Raspberry Pi and Arduino, coupled with various sensors. The data from these sensors flows to the Raspberry Pi, which acts as an edge gateway. Furthermore, they use a cluster of Raspberry Pi devices to simulate a fog environment, all managed using OpenFaaS [1], a popular serverless framework.

What is the relation to this thesis?

The distinction among the different layer and responsibilities has been one of the source of inspiration during the development of Edgeless. Furthermore, this paper employs a similar setup to the one in Edgeless, with a Raspberry Pi and a microcontroller, for collecting and processing data from various sensors, e.g. temperature. In addition, it helped in creating a framework on where components should be located. For example, the serverless function

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

running inside the Raspberry Pi for local computation. As a result, this paper contributed to the composition of the crucial components in Edgeless, as discussed in section 3.4.

8.2.2 Serverless in Machine Learning Applications at the Edge (5)

This paper delves into the utilization of serverless computing for deploying machine learning applications, particularly in edge computing environments. Furthermore, it addresses the growing need for handling vast amounts of data generated by IoT sensors for machine learning applications, such as smart cities, self-driving cars, and augmented reality.

A serverless computing approach is proposed for edge computing to implement machine learning applications. Therefore, the primary case study used for evaluation is image classification with the MNIST dataset, a large database of handwritten digits commonly used for image processing training (56). Furthermore, the testbed setup involves the cloud, a serverless edge layer, and event sources. The serverless edge layer is deployed over Kubernetes, and the authors utilize Knative [1], an open-source serverless platform, for deploying machine learning applications as functions.

The paper evaluates the performance of the serverless approach in terms of response time, running time, and end-to-end delay for machine learning applications. The results indicate that the serverless approach can ensure optimal response and running time, reduce the end-to-end delay of the application, and support distributed machine learning.

What is the relation to this thesis?

This paper is part of related work, in the serverless and edge domain, due to its promising findings for deploying serverless machine learning applications at the edge. It is encouraging to study other literatures that are leveraging serverless functions to compute workloads that were considered heavier at the edge. Although edge computing is not the replacement of cloud, the gap between both paradigms is certainly closing. As a result, the findings in this paper provided some insights into how vast edge computing can become, specially when implementing it serverless functions, for both scalability and efficiency.

8.2.3 A Serverless Real-Time Data Analytics for Edge Computing (6)

The paper delves into the intricacies of integrating serverless computing with real-time data analytics, particularly in the context of edge computing. Serverless computing is especially beneficial in scenarios where data is generated in vast quantities and needs to be processed efficiently and promptly, providing great scalability to the system.

[1] https://knative.dev/docs/

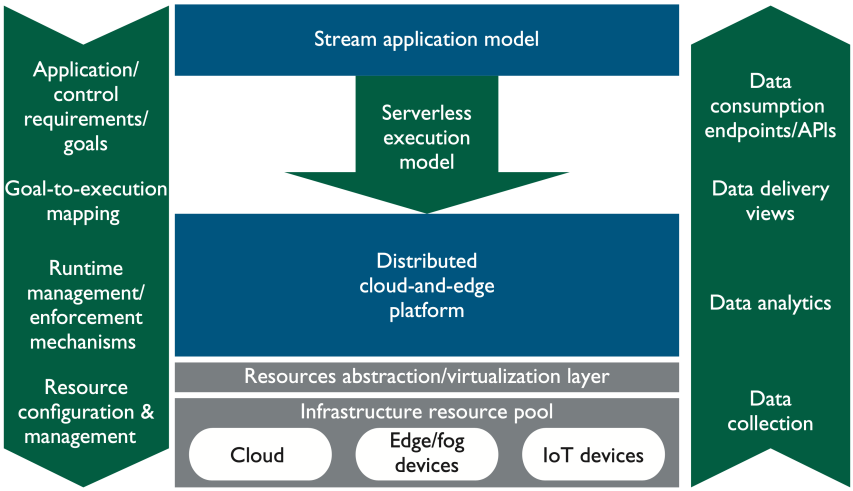

Figure 8.5: High level architecture of cloud and edge data analytics platform (6).

The paper introduces three primary computational models, with a significant focus on the integration of serverless computing with edge and cloud processing. As shown in figure 8.5, the three models are composed by: cloud, edge and IoT devices. (1) The edge focuses on local views, processing data close to its source. (2) The cloud supports global views, combining and analyzing data from various edge devices, regions, or domains. (3) The IoT devices gather data from sensors to be computed at the edge or cloud. This distributed approach ensures that all computing layers are utilized effectively.

To illustrate the need for distributed cloud and edge processing, the paper presents scenarios from IoT mobile healthcare (mHealth) and discuss holistic data analytics. One such scenario involves wearable biosensors in disaster situations, where immediate attention is crucial. These sensors emit data to nearby portable edge devices, like smartphones, which perform basic analytics to determine the overall stability of an injured person.

What is the relation to this thesis?

This paper contributed to the development of Edgeless by creating a framework for separating the different computational models. Cloud, edge and IoT devices are all considered when designing the main components of Edgeless. Furthermore, this paper also provides an interesting serverless stream model for processing data at the edge. This model encapsulates user-defined analytics to process data along the stream. Therefore, adopting has been beneficial for using serverless functions as the compute engine in the core of Edgeless.

9

Conclusion

As technology evolves and data generation becomes more eminent, there need to be more efficient ways to process this data. Today, the proliferation of sensors and other smart devices are seem to be on the rise. All of this actually leads to edge computing, which have been helpful for processing data closer to where it is generated. Agriculture is one of the industries that have heavily adopted sensors for better estimation and practices.

Precision Agriculture is born as a way of measuring and capturing real-time data about crops for more precision in farming. However, there are several limitations in edge computing studied throughout the development of this work. Performance is amongst the most important of these limitations, because it concerns with systems would scale at the edge if there is a high number of sensors concurrently pushing data to an edge device.

9.1 Main Contributions

Given this challenge, the deployment of serverless functions on edge devices was considered as a solution to address performance bottlenecks. As a result, two main research questions came up for exploring performance benefits of serverless computing, in the cloud and edge.

1. *RQ1: How is the performance of serverless functions in the cloud?* The idea was to set a framework for evaluating serverless performance. The cloud was almost the perfect environment due to the longevity and usage that serverless functions.

2. *RQ2: How is the performance of serverless functions at the edge?* Once understanding the performance benefits of serverless in the cloud, the question turns into whether this can be also possible in an edge device, with more associated constraints.

However, some reasonable considerations have to be made when comparing edge and cloud, given their difference in computational capacity. As a result, there were three main contributions which aimed at finding this performance benefit of serverless at the edge.

C1: Design and implementation of Edgeless. A serverless function system was designed and developed for processing workloads in Precise Agriculture. Data was collected using microphone sensors that were deployed in tomato greenhouses. During this process, an extensive collaboration with Plense Technologies was established for gathering requirements and collecting sample data from sensors. The system was divided in three different layers, for end-to-end data pipeline: (1) *Mist:* deployment of sensors and microcontrollers belong to this layer, also called as endpoint. (2) *Edge:* integration of AWS Lambda serverless function with Greengrass in a Raspberry Pi. (2) *Cloud:* further data processing is done in the cloud, with more powerful capabilities to train Machine Learning algorithms to apply correlation for crop health predictions.

C2. Cloud Serverless Evaluation. An evaluation was done to measure throughput (requests per seconds) of serverless functions in the cloud. By setting up, deploying, designing experiments, interesting results were gathered about function throughput performance. The experiment was conducted using simulations, since microphone sensors were not available at the moment of the experiment. **Setup.** Lambda functions and Docker containers were used to compare their performance, and to understand Lambda autoscaling features. **Deployment.** AWS SAM, an Infrastructure as Code framework, was used to deploy all the necessary resources for this experiment. **Results.** Lambda functions thrive when invoked concurrently, due to the provisioning of multiple execution environments, resulting in higher throughput.

C3. Edge Serverless Evaluation. An evaluation was conducted to measure throughput of serverless functions at the edge. Applying a similar methodology as the former contribution, there was a setup, deployment and design of two experiments which aimed at understanding Lambda autoscaling. Since there are some differences in throughput (Lambda and Docker) in the cloud evaluation, due to Lambda provisioning multiple execution environments, the question was whether this was also the case at an edge environment. **Setup.** Greengrass components were used as the main subjects for performance evaluation. **Deployment.** Lambda function, Docker container and Native components were deployed in a Raspberry Pi. **Results.** Lambda functions do

not have scalability features in an edge environment. During the experiments, both Lambdas yielded lower throughput than Docker containers and Native OS Process.

9.2 Future Work

Throughout the development of Edgeless, there were few points that were considered as part of the future work. These points tackle some of the unsolved challenges from edge computing, not covered in this thesis. Besides performance, there are plenty of other challenges that are still open for research and work. Amongst some of these were energy consumption of edge devices, and the network latency in communication between devices.

9.2.1 Energy Consumption

Edge devices are bounded by a certain level of power capacity. Efficient computational models need to be considered for evaluating how much power an execution of signal processing consumes. It may be the case that the battery lifespan of an edge device is shorter than what the incoming data might require. Therefore, this may cause severe a disruption to the overall processing time of the system. Furthermore, this also incurs in some maintenance cost, since the device batteries would have to be changed every so often. There are two immediate directions for conducting this energy consumption research as future work.

1. **Greengrass Energy Consumption.** Understanding the energy consumption from Greengrass is a crucial piece of work in the Edgeless ecosystem. As a result, it becomes really relevant to capture some measurements about CPU power consumption of serverless function components. As noted in section 4.1.4, there are two main ways of edge deployment for Lambda, pinned and non-pinned. Therefore, making proper evaluation and analysis between energy consumption and performance from these two type of deployments can be beneficial for the development of this work.

2. **Computation Offloading.** Moving workloads between computing paradigms can be beneficial for saving energy. As part of the next future work, heuristic evaluations are important because they would provide some baseline for what needs to be offloaded to the cloud, edge or mist. For example, considering the battery level of a device and the energy consumption of processing a signal, can be some fundamental points for offload evaluations. Therefore, edge devices may offload computation if they are below a certain battery level with high incoming workloads for processing.

9.2.2 Network Latency

Depending on the environment where edge devices operate, network latency can have a severe impact when establishing reliable communication among devices. This can be specially seen in Agricultural fields and greenhouses, where network connectivity might be particularly challenging.Therefore, another interesting future work is to experiment with different network protocols to measure and determine latency factor. This way, the implementation of Edgeless can be adapted to use the most reliable protocols for establishing communication. Therefore, there are two different networks layers to further research.

1. **Perception Layer Protocols.** This layer corresponds to the communication established between the mist and the edge. These are lower level protocols that allow connection between microphone sensors and -micro-controllers/processors. Therefore, there are two protocols that can be evaluated to understand latency impact. (1) *LoRA*: low-power, wide-area network (LPWAN) technology for long-range communication between devices with low data rates. (2) *Bluetooth*: short-range wireless communication for exchanging data between devices over short distances.

2. **Application Layer Protocols.** This layer corresponds to the communication established between the edge and cloud. These correspond to higher level protocols that allow connection between the microprocessors and the cloud. As a result, there are also two protocols that can be experimented for measuring the latency impact when transferring data. (1) *HTTP*: follows a traditional requests-response model, like in a client-server architecture. (2) *MQTT*: operates on a publish-subscribe model, allowing devices to publish to topics and other devices to subscribe to these topics.

Appendix A

Artifact Appendix

A.1 Abstract

The source code for Edgeless can be found in this GitHub repository: `https://github.com/edreinoso/Edgeless`. There is a README file in this directory that specifies all the details for deploying and replicating the evaluations during this master thesis. The workloads for signal processing is not taken into account, since there are some proprietary compliance with Plense Technologies. However, this should not matter in practice, since other applications can be deployed with this same setup. Nonetheless, the repository consists of two major components for deployment in cloud and edge. This constitutes on deploying serverless functions and Greengrass software in a microprocessor.

A.2 Checklist

The following checklist contains the main points that are necessary for replicating Edgeless.

- Program: Edgeless: Serverless Computing at the Edge
- Run-time environment: Debian OS
- Edge device: Raspberry Pi 4, 1.8 GHz Quad core (ARM v8), 4GB SDRAM
- Greengrass version specification: v2.12.0
- Metrics: Throughput (requests per seconds)
- Output: logging files for each individual consumer
- Experiments: as displayed in Section 6.1

- How much time is needed to complete experiments (approximately)?: 1-5 minutes, depending on the workload size
- Publicly available?: yes (see GitHub page)

A.3 Description

A.3.1 How to access

Access Edgeless through the following link: `https://github.com/edreinoso/Edgeless`

A.3.2 AWS Managed Component Dependencies

1. *Greengrass Nucleus.* Version 2.12.0
   Core component to run and orchestrate Greengrass main functionalities.
2. *Greengrass CLI.* Version 2.12.0
   Command line interface to interact with the main Greengrass software (nucleus).
3. *Log Manager.* Version 2.3.7
   Upload component logs to Amazon CloudWatch.
4. *Log Debug Console.* Version 2.4.1
   Provides a local dashboard to display information about components deployed.
5. *Lambda Manager.* Version 2.3.2
   Manages work items and inter-process communications in AWS Lambda functions.
6. *Lambda Launcher.* Version 2.0.12
   Starts and stops AWS Lambda function in the core device.
7. *Lambda Runtimes.* Version 2.0.8
   Provides the runtime for the AWS Lambda functions to run and execute.

A.3.3 Custom Managed Components

1. *Lambda Greengrass Container.*
   Lambda function for signal processing in a Greengrass Container.
2. *Lambda No Container.*
   Lambda function for signal processing in No Container.

3. *Docker Container.*
   Docker containers used for signal processing.

4. *Native OS Process.*
   Code is executing natively as a long-running OS process.

Since these components are not managed by AWS, they do not have a specific version. Their version will be changing with the constant modifications done to the component.

A.4 Installation

During the installation Edgeless, there are two major activities to deploy all the functionalities. This includes Greengrass Core Software, and the Component Deployments. The order is important, since the latter builds on top of the former. Component Deployment cannot happen if the Greengrass Core Software is not already installed.

A.4.1 Greengrass Core Software

For installing Greengrass in an edge device, there is a particular step-by-step guide that needs to be followed. Most importantly, it is relevant to consider the prerequisites of the edge device [1] [2] before installing this software. Nonetheless, detail installation of Greengrass core software can be found in the AWS official website [3], with more specifications.

A.4.2 Web Server Setup

Deploying components in Greengrass can be done using CLI or HTTP API endpoint. For Edgeless, Boto3 SDK is used to deploy the main components of the system. First, components can be created and updated in the cloud, and then deployed in a Core edge device. Therefore, Boto3 is installed with Pip, using the following command:

pip install boto3

After installing boto3 library, a flask web application server is configured to interact with the AWS APIs from the Greengrass service to carry operations: (1) Create component versions, (2) Create deployments. During Edgeless development, this flask web server was

[1] `https://docs.aws.amazon.com/greengrass/v2/developerguide/getting-started.html`
[2] `https://docs.aws.amazon.com/greengrass/v2/developerguide/setting-up.html`
[3] `https://docs.aws.amazon.com/greengrass/v2/developerguide/install-greengrass-v2.html`

run locally, although possible to deploy it in the cloud. Nonetheless, after the boto3 library is installed, the web server needs to be started with the following command:

flask run

A.4.3 Component Deployment

After the web server has been started, operations for creating components and deployments can be done using the **localhost:port**. A *deployment.json* file can be found in the component configuration directory from the GitHub repo. This file includes the main components of Edgeless. To execute a deployment, an API call to the flask web server is required, passing the configurations about the deployment. For simplicity, a shell script (found in the Greengrass script directory) is executed to automate this API call process. Therefore, a deployment can be created in the device with the following command:

sh deployment.sh

A.4.4 Lambda Function Update

Lambda functions area crucial for performing the experiments. Therefore, understanding how to change the code in the serverless function will help navigate through how to deploy the function at the edge. In Edgeless, AWS Lambda CLI was used to carry such operations. Updating the version of the function is a very relevant step in the process, for making sure that the edge serverless function can have the most up-to-date version of the code. So, first, zip all the files and dependencies that the function will use, e.g. signal processing.

zip -r function.zip lambda_function.py others.py lib_directory/

After zipping the necessary code for the function to run, then use the following command to update the code for the function:

aws lambda update-function-code –function-name name –zip-file fileb://function.zip

Once the function has been updated in the cloud, a new version of the function has to be deployed. That way, Greengrass can then reference this new version at the edge device:

aws lambda publish-version –function-name name

For ease of automation, this process has been scripted using shell. This script can be located in the Greengrass script component directory from the GitHub repository. To execute it, use the following command:

sh lambda.sh

A.5 Experiment Workflow

As discussed in the experiment design section 6.1, there are two different kinds of experiments that consist of separate workloads, sequential and concurrent. Furthermore, there are also two different ways of running the experiments, depending on the environment in which they are executed, in this case: cloud or edge.

A.5.1 Cloud Experiment Workflow

For the cloud experiment, sequential and concurrent workloads were executed as part of the experiment design. Therefore, the following steps can be carried to run the experiment:

1. Put the signal processing files in a local directory

2. Create a S3 bucket where these signal processing files can be stored.

3. Put the signal processing files in the S3 bucket

4. Run the script for sequential workloads:
   python w1_load_test.py –threads –repetitions –runtime
   The number of repetitions can vary depending on how many files are desired to be processed. This experiment will have a constant number of threads. As for the runtime, this can be lambda or docker.

5. Run the script for concurrent workloads:
   python w2_load_test.py –threads –repetitions –runtime
   The number of threads can change depending on the number of simulated clients. This experiment will have a constant number of repetitions. As the previous one, the runtime corresponds to a lambda function or a docker container.

A.5.2 Edge Experiment Workflow

The edge experiments were conducted somewhat differently than the cloud. The biggest change is the inclusion of MQTT at the start of the experiment. However, in general, the following guidelines provide more information on how to run the experiments at the edge.

1. Put the signal processing files in a local directory
2. Create an S3 bucket where files could be stored
3. Put the signal processing files in the S3 bucket
4. Run the script for starting the experiment:
   sh mqtt.sh
   This script invokes the MQTT topic associated with the core edge device. The Gateway component (discussed in section 5.1.3) wakes up on the invocation call from the MQTT, and this starts doing some job. In more details, this script is able to invoke the other compute service components to carry the actual signal processing.

A.6 Evaluation and expected results

After the experiments have completed, there are several log files that need to be retrieved. These files are referencing the time taken by the runtime to process each operation. This output is calculated with respect to the total execution time from the experiment. A distinction needs to be made between cloud and edge when evaluating these results.

A.6.1 Cloud Evaluation Results

For the cloud results, every time the experiments are run, either sequential and concurrent, there are new files that get created directly in the machine where it is being executed. This is done in an outexecution directory that is inside of test in cloud. This directory is further broken down into several categories, depending on the runtime: lambda (concurrent, sequential) and docker (service, task). File contains the time taken by each operation.

A.6.2 Edge Evaluation Results

For the edge, the results will be written directly in the Raspberry Pi. The Gateway component will be responsible for recording the time taken for each of the compute services (lambda functions, docker container and native code) to execute the request. The logs will be recorded with a timestamp as a file name in a directory for their corresponding runtime.

References

[1] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. {**FarmBeats**}**: an** {**IoT**} **platform for** {**Data-Driven**} **agriculture**. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017. v, ix, 79, 80

[2] Francisco Javier Ferrández-Pastor, Juan Manuel García-Chamizo, Mario Nieto-Hidalgo, Jerónimo Mora-Pascual, and José Mora-Martínez. **Developing ubiquitous sensor network platform using internet of things: Application in precision agriculture**. *Sensors*, **16**(7):1141, 2016. v, ix, 81

[3] Deepti Gupta, Paras Bhatt, and Smriti Bhatt. **A game theoretic analysis for cooperative smart farming**. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2350–2359. IEEE, 2020. v, 82

[4] Suvajit Sarkar, Rajeev Wankar, Satish Narayana Srirama, and Nagender Kumar Suryadevara. **Serverless management of sensing systems for fog computing framework**. *IEEE Sensors Journal*, **20**(3):1564–1572, 2019. v, ix, 83, 84

[5] Ta Phuong Bac, Minh Ngoc Tran, and YoungHan Kim. **Serverless computing approach for deploying machine learning applications in edge layer**. In *2022 International Conference on Information Networking (ICOIN)*, pages 396–401. IEEE, 2022. v, 85

[6] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. **A serverless real-time data analytics**

platform for edge computing. *IEEE Internet Computing*, **21**(4):64–71, 2017. v, ix, 85, 86

[7] Mohammad S Aslanpour, Sukhpal Singh Gill, and Adel N Toosi. **Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research**. *Internet of Things*, **12**:100273, 2020. vii, 13, 14

[8] Erwin Van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. **The SPEC-RG reference architecture for FaaS: From microservices and containers to serverless platforms**. *IEEE Internet Computing*, **23**(6):7–18, 2019. vii, 18, 20

[9] Satadal Dutta, Elias Kaiser, Priscila Malcolm Matamoros, Peter Steeneken, and Gerard Verbiest. **Listening to ultrasound from plants reveals xylem vessel anatomy. Ultrasound characterization of plant vessels.** 2021. vii, 24

[10] **AWS Lambda Official Documentation**. vii, 31, 43, 44, 46

[11] **AWS S3 Official Documentation**. xi, 32, 49, 50

[12] Meticulous Market Research Pvt. Ltd. **Agriculture IoT Market to Grow at a CAGR of 10.82028, Says Meticulous Research**, 2022. 2

[13] Michael Hauben. **History of ARPANET**. *Site de l'Instituto Superior de Engenharia do Porto*, **17**:1–20, 2007. 7

[14] Bart Jacob, Michael Brown, Kentaro Fukui, Nihar Trivedi, et al. **Introduction to grid computing**. *IBM redbooks*, pages 3–6, 2005. 7

[15] Aminul Haque, Saadat M Alhashmi, and Rajendran Parthiban. **A survey of economic models in grid computing**. *Future Generation Computer Systems*, **27**(8):1056–1069, 2011. 7

[16] Susanta Nanda Tzi-cker Chiueh and Stony Brook. **A survey on virtualization technologies**. *Rpe Report*, **142**, 2005. 8

[17] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. **Hypervisor: A survey on concepts and taxonomy**. *International Journal of Innovative Technology and Exploring Engineering*, **2**(3):222–225, 2013. 8

[18] Sajee Mathew and J Varia. **Overview of amazon web services**. *Amazon Whitepapers*, **105**:1–22, 2014. 8

[19] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. **Cloud computing: A study of infrastructure as a service (IAAS)**. *International Journal of engineering and information Technology*, **2**(1):60–63, 2010. 9

[20] Claus Pahl. **Containerization and the paas cloud**. *IEEE Cloud Computing*, **2**(3):24–31, 2015. 9

[21] SK Sowmya, P Deepika, and J Naren. **Layers of cloud–IaaS, PaaS and SaaS: a survey**. *International Journal of Computer Science and Information Technologies*, **5**(3):4477–4480, 2014. 9

[22] Maram Mohammed Falatah and Omar Abdullah Batarfi. **Cloud scalability considerations**. *International Journal of Computer Science and Engineering Survey*, **5**(4):37, 2014. 9

[23] Khalid Rafique, Abdul Wahid Tareen, Muhammad Saeed, Jingzhu Wu, and Shahryar Shafique Qureshi. **Cloud computing economics opportunities and challenges**. In *2011 4th IEEE International Conference on Broadband Network and Multimedia Technology*, pages 401–406. IEEE, 2011. 10

[24] Leah Riungu-Kalliosaari, Simo Mäkinen, Lucy Ellen Lwakatare, Juha Tiihonen, and Tomi Männistö. **DevOps adoption benefits and challenges in practice: A case study**. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 590–597. Springer, 2016. 10

[25] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. **Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices**. *IEEE access*, **5**:3909–3943, 2017. 10

[26] Abedallah Zaid Abualkishik, Ali A Alwan, and Yonis Gulzar. **Disaster recovery in cloud computing systems: An overview**. *International Journal of Advanced Computer Science and Applications*, **11**(9), 2020. 10

[27] Mohamed Firdhous, Osman Ghazali, and Suhaidi Hassan. **Fog computing: Will it be the future of cloud computing?** 2014. 10

[28] Awada Uchechukwu, Keqiu Li, Yanming Shen, et al. **Energy consumption in cloud computing data centers**. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, **3**(3):31–48, 2014. 11

[29] Anca Apostu, Florina Puican, Geanina Ularu, George Suciu, Gyorgy Todoran, et al. **Study on advantages and disadvantages of Cloud Computing–the advantages of Telemetry Applications in the Cloud**. *Recent advances in applied computer science and digital services*, **2103**, 2013. 11

[30] V Krishna Reddy, B Thirumala Rao, LSS Reddy, and P Sai Kiran. **Research issues in cloud computing**. *Global Journal of Computer Science and Technology*, **11**(11):59–64, 2011. 11

[31] Karen Rose, Scott Eldridge, and Lyman Chapin. **The internet of things: An overview**. *The internet society (ISOC)*, **80**:1–50, 2015. 11

[32] John Scourias. **Overview of the global system for mobile communications**. *University of Waterloo*, **4**:7, 1995. 12

[33] Jain Cai and David J Goodman. **General packet radio service in GSM**. *IEEE Communications magazine*, **35**(10):122–131, 1997. 12

[34] Tero Ojanpera and Ramjee Prasad. **An overview of third-generation wireless personal communications: A European perspective**. *IEEE Personal Communications*, **5**(6):59–65, 1998. 12

[35] B Vasavi, Mounika Marepalli, and Leepika Gudur. **Evolution of 4G-research directions towards fourth generation wireless communication**. *International Journal of Computer Science and Information Technologies*, **2**(3):1087–1095, 2011. 12

[36] Xin Li, Mohammed Samaka, H Anthony Chan, Deval Bhamare, Lav Gupta, Chengcheng Guo, and Raj Jain. **Network slicing for 5G: Challenges and opportunities**. *IEEE Internet Computing*, **21**(5):20–27, 2017. 12

[37] Mazliza Othman, Sajjad Ahmad Madani, Samee Ullah Khan, et al. **A survey of mobile cloud computing application models**. *IEEE communications surveys & tutorials*, **16**(1):393–413, 2013. 13

[38] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. **A survey on mobile edge computing: The communication perspective**. *IEEE communications surveys & tutorials*, **19**(4):2322–2358, 2017. 13

[39] Athena Vakali and George Pallis. **Content delivery networks: Status and trends**. *IEEE Internet Computing*, **7**(6):68–74, 2003. 13

[40] Claireice Mathai. **What is mist computing and how does it work**, 2023. 14

[41] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. **Fog computing and its role in the internet of things**. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012. 14

[42] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. **Edge computing: A survey**. *Future Generation Computer Systems*, **97**:219–235, 2019. 15

[43] Huang Zeyu, Xia Geming, Wang Zhaohang, and Yuan Sen. **Survey on edge computing security**. In *2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 96–105. IEEE, 2020. 15

[44] Pawani Porambage, Jude Okwuibe, Madhusanka Liyanage, Mika Ylianttila, and Tarik Taleb. **Survey on multi-access edge computing for internet of things realization**. *IEEE Communications Surveys & Tutorials*, **20**(4):2961–2991, 2018. 15

[45] Linghe Kong, Jinlin Tan, Junqin Huang, Guihai Chen, Shuaitian Wang, Xi Jin, Peng Zeng, Muhammad Khan, and Sajal K Das. **Edge-computing-driven internet of things: A survey**. *ACM Computing Surveys*, **55**(8):1–41, 2022. 15

[46] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. **Challenges and opportunities in edge computing**. In *2016 IEEE international conference on smart cloud (SmartCloud)*, pages 20–26. IEEE, 2016. 16

[47] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. **Edge computing: Vision and challenges**. *IEEE internet of things journal*, **3**(5):637–646, 2016. 16

[48] Yuvraj Sahni, Jiannong Cao, Lei Yang, and Shengwei Wang. **Distributed resource scheduling in edge computing: Problems, solutions, and opportunities**. *Computer Networks*, **219**:109430, 2022. 16

[49] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003. 25

[50] **AWS Greengrass Official Documentation**. 30

[51] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The atLarge vision on the design of distributed systems and ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 32

[52] Garrett McGrath and Paul R Brenner. **Serverless computing: Design, implementation, and performance**. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017. 33

[53] Matthijs Jansen, Auday Al-Dulaimy, Alessandro V Papadopoulos, Animesh Trivedi, and Alexandru Iosup. **The SPEC-RG reference architecture for the compute continuum**. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 469–484. IEEE, 2023. 34

[54] Eno Thereska. **Architecting for high availability on Amazon S3**, 2021. 47

[55] Zhang Jian-Hua and Zhang Nan. **Cloud computing-based data storage and disaster recovery**. In *2011 International Conference on Future Computer Science and Education*, pages 629–632. IEEE, 2011. 47

[56] Wikipedia. **MNIST database — Wikipedia, The Free Encyclopedia**, 2023. 85