INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Serverless computing

JOSÉ MIGUEL OLIVEIRA
Outubro de 2023

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Serverless computing

José Miguel Oliveira

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Prof. Dr. Alexandre Bragança

Evaluation Committee:
President:
To Be Defined

Members:
To Be Defined

Porto, October 14, 2023

Dedicatory

This work is dedicated to all those whose unwavering support and encouragement have been the driving force behind my academic journey, which culminated in this thesis. Firstly, I would like to give a huge appraisal to my parents who have always supported and encouraged me to continue and strive for more, both in my personal life and in my academic one. Then, to the woman who became my wife during this period in which I was writing this document and without whom this moment would be unreachable. Your belief in my abilities, your endless patience, and your enduring love have made this endeavor possible. I am profoundly grateful for the presence of both in my life and for the inspiration you have provided.

I also want to give thanks to the teachers that been part of my academic journey, making me a professional who is ready for the professional world. In this last group, a special word of gratitude to this thesis' supervisor, Alexandre Bragança, which has been very present and helpful throughout this process. Your fast feedback and outstanding suggestions have made this process a lot easier.

Abstract

Serverless computing has emerged as a new mindset when it comes to cloud computing, promising efficient resource utilization, automatic scaling, and cost optimization for a wide range of applications. This thesis explores the adoption, performance, and cost considerations of deploying applications that use intend to use serverless functions, one of the leading Serverless types.

This thesis starts by providing an overview of Serverless computing, including its key advantages and disadvantages and the rising adoption it has gained throughout the recent years. It presents a comprehensive comparison of various Serverless platforms and discusses the unique features offered by each.

After this context phase, this thesis presents a design section composed by a migration guide that allows developers to transition from a traditional application to one that takes advantage of serverless benefits. The guide outlines best practices and step-by-step instructions, facilitating the adoption of Serverless computing in real-world scenarios.

Using the previously created guide, the next section carries out a practical use case: the migration of complex computational logic from a traditional Java application to AWS Lambda functions. Performance evaluations are conducted, considering metrics such as the execution duration and the amount of concurrent executions.

These findings are then evaluated next to the costs associated with deploying and running Java applications in a virtual machine or with a Serverless architecture.

While Serverless computing is quite promising, networking issues often arise in practice, affecting the overall efficiency of Serverless applications. This thesis addresses these challenges, identifying the installation and migration difficulties, how to overcome them, and what are the expected limitations, while proposing potential solutions.

In summary, this thesis offers valuable insights into the adoption, performance, and cost optimization of Serverless computing for Java applications. It provides a roadmap for developers looking to take advantage of the benefits of Serverless computing in their projects.

Keywords: Serverless, Cloud, Function, Infrastructure

Acronyms

API Application Programming Interface.

ARN Amazon Resource Name.

AWS Amazon Web Services.

BaaS Backend as a Service.

CLI Command Line Interface.

CPU Central Processing Unit.

CVE Common Vulnerabilities and Exposures.

DDoS Distributed Denial of Service.

DSR Design Science Research.

DTO Data Transfer Object.

FaaS Function as a Service.

GCP Google Cloud Platform.

GRS Geo-Redundant Storage.

IAM Identity And Management.

IoT Internet of Things.

JVM Java Virtual Machine.

KPI Key Performance Indicator.

LRS Locally-Redundant Storage.

RA-GRS Read Access Geo-Redundant Storage.

RGB Red Green Blue.

SDK Software Development Kit.

VM Virtual Machine.

ZRS Zone-Redundant Storage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis aims to investigate the use of serverless computing as a model for developing and deploying cloud applications. The research is based on an extensive research, which includes a literature review, use cases, and a guide on how to get started with these platforms and with the serverless principles.

1.1 Problem statement

The Serverless computing model is recent, highly referenced, and its adoption seems to be growing strongly. However, this model is still hardly known to potential users, which can lead to wrong implementations or it not being properly adjusted to the problem (Paul Castro 2019). The lack of knowledge and the difficulty in changing a way of working that has prevailed for so many years can be two of the reasons that justify slow transition towards this model and way of working.

Handling large amounts of data in hosted servers is an example of a process that can end up being quite expensive in many fronts. It requires scaling up or down the servers, according to the received load, and doing all kinds of maintaining in the host machines.

With the correct usage of a Serverless approach, this process can become very easy and painless. So, the problem relies on understanding on which types of project it makes sense to apply this recent approach, and which benefits can it bring to the table, while we compare it to a traditional micro service architecture.

1.2 Objectives

The goal of this thesis is to study this model in order to identify use cases that make sense for this computing model compared to other models. Analyze the available platforms, propose a guide for their adoption, design, implementation and deployment of this computing model, with the aim of achieving better efficiency comparing to a traditional hosted architecture.

By the end of this, it is expected to have discovered the benefits and the disadvantages that it brings to the table, and what are the available Serverless frameworks available in the market nowadays, and how they fit to different problems.

The slow migration to this model is also something that is worth investigating. If it were the future and an obvious followup to the current way of doing things, people would be migrating to this approach as fast as possible. What is keeping people from moving? That

is also one of the main reasons for going forward with this thesis, in order to demystify this whole topic.

1.3 Methodology

The methodology used in this thesis is the Design Science Research (DSR). DSR is an iterative process of problem identification, artifact design and development, and evaluation of innovative artifacts to solve complex problems in various fields (Claes Wohlin 2021). DSR is particularly well-suited for this study, as it is a research methodology that emphasizes the development and evaluation of solutions to real-world problems, while allowing continuous adjustments throughout the discovery process (Figure 1.1).

Figure 1.1: Design Science Research approach (Claes Wohlin 2021)

The primary objective of this study is to investigate the effectiveness of serverless computing as a means of addressing the challenges associated with traditional server-based architectures. With this in mind, the DSR methodology was followed, which provides a structured approach to developing and evaluating the serverless solutions under analysis.

1.4 Contribution

This thesis presents valuable contributions towards serverless solutions, with a focus on hybrid Java applications. These contributions consist on the following key elements:

1. Full migration guide: This thesis includes the development of a end-to-end migration guide. This guide offers a detailed road map for seamlessly integrating serverless functions into existing applications. It provides step-by-step instructions, practical insights, and best practices, ensuring that the migration process is both smooth and efficient.

2. Practical implementation: In addition to creating the migration guide, this thesis also demonstrates its applicability on top of a real world use case. By applying the guide to a practical use case, it showcases its effectiveness and struggles with a tangible context.

3. Publicly accessible GitHub repository: To facilitate widespread access and collaboration, all materials and resources used in this thesis will be made available. This includes the code samples used for the proof of concept, as well as any other relevant assets. This comprehensive set of resources will be hosted on a public GitHub repository, fostering a collaborative environment for developers, researchers, and enthusiasts keen on further exploring this theme.

1.5 Document outline

This document follows the following structure. Chapter 2 displays the literature review, which is made of by an introduction to the computing concept and some of the tools and concepts that marked history, such as virtualization and containerization, followed by a serverless computing, where some of its major concepts and platforms are studied. Chapter 3 describes the value analysis that serverless computing represents to the costumer, while discussing the benefits and sacrifices attached to its usage. In Chapter 4, the reader may find the design proposal for a serverless use case, followed by a guide that allows Java developers to incorporate Serverless functions in their applications, step by step. Chapter 5 is where the use case is implemented in an attempt to test out the previously elaborated guide. Throughout the implementation process, some problems were encountered and those are also explained by the end of this chapter. Given the experience of the implementation phase, in chapter 6 we have the evaluation phase, in which the outcome is analysed and compared with the needed metrics and tools. To finalize the content, in chapter 7, there is a conclusion phase in which the whole thesis is analysed in terms of its added values and future work. By the end we have the appendixes, in which we can find things like the repository in which the use case implementation is.

Chapter 2

Literature Review

This chapter goes, initially, through the history of computing, with a special focus on virtualization and containers. Then, travelling a bit to the future, there is a subsection that analyses some Serverless computing topics, as an attempt to better understand its definition and the current serverless options.

2.1 Computing background

Computing has not always been like it is nowadays. Throughout the time, there has been a constant enhancement on the capabilities of computers, both on the software end, and also on the hardware end.

2.1.1 History of computing

Throughout the years, software developers have always needed to launch their application to somewhere. Until the early 2000s, there was only one major option, which is deploying to a local machine. This is also called *on premise*.

Having an application deployed On Premise is like having all your wage on your own wallet or home. It means that a certain company keeps all its data, servers and everything else in its IT environment in-house (Kumar 2022). These people are responsible for running, supporting and maintaining the data all the time.

In the last twenty years, a lot of developments have emerged to tackle this singularity. With the emergence of several Cloud providers, the insecurity and unreliability of On Premise instances has been fading away ever since.

However, regardless of the origin of the computer, there are several ways of running a software inside it. There are several models, architectures and methodologies to optimize and segregate applications inside the same machine. Virtualization and Containerization are some of the trending ways to do so.

Virtualization

Virtualization refers to the creation of a virtual version of something, such as an operating system (OS), a server, a storage device or network resources (Brush 2021).

The concept of virtualization is generally believed to have its origins in the mainframe days (Figure 2.1) in the late 1960s and early 1970s, when IBM invested a lot of time and effort in developing robust time-sharing solutions. Time-sharing refers to the shared usage of

computer resources among a large group of users, aiming to increase the efficiency of both the users and the expensive computer resources they share. This model represented a major breakthrough in computer technology: the cost of providing computing capability dropped considerably and it became possible for organizations, and even individuals, to use a computer without actually owning one (Oracle 2022).

Figure 2.1: One of the first mainframe computers (Brooks 2021)

Similar reasons are driving virtualization for industry standard computing today: the capacity in a single server is so large that it is almost impossible for most workloads to effectively use it. The best way to improve resource utilization, and at the same time simplify data center management, is through virtualization (Oracle 2022).

A key use of virtualization technology lays with server virtualization, which uses a software layer, called a hypervisor, to emulate the underlying hardware (Figure 2.2). This often includes the CPU's memory, input/output (I/O) and network traffic (Brush 2021).

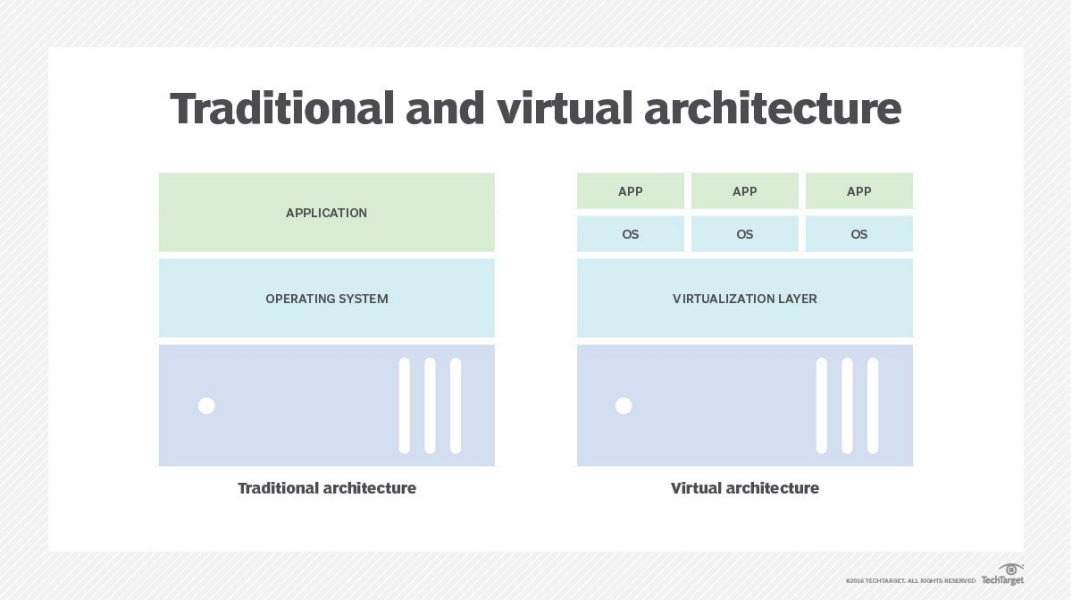

Figure 2.2: Traditional vs Virtual architecture (Brush 2021)

Since most guest operating systems and applications don't need the full use of the underlying hardware, this was a ground breaking approach on computing history.

This allows for greater flexibility, control and isolation by removing the dependency on a given hardware platform. While initially meant for server virtualization, the concept of virtualization has spread to applications, networks, data and desktops(Brush 2021).

Containers

Containers are lightweight software packages that can be run with isolated dependencies on any environment. However, this concept could be easily confused with virtualization.

The key difference between containerization and virtualization is that containers only virtualize software layers above the operating system level, whilst in the virtualization concept it virtualizes an entire machine down to the hardware layers (Figure 2.3).

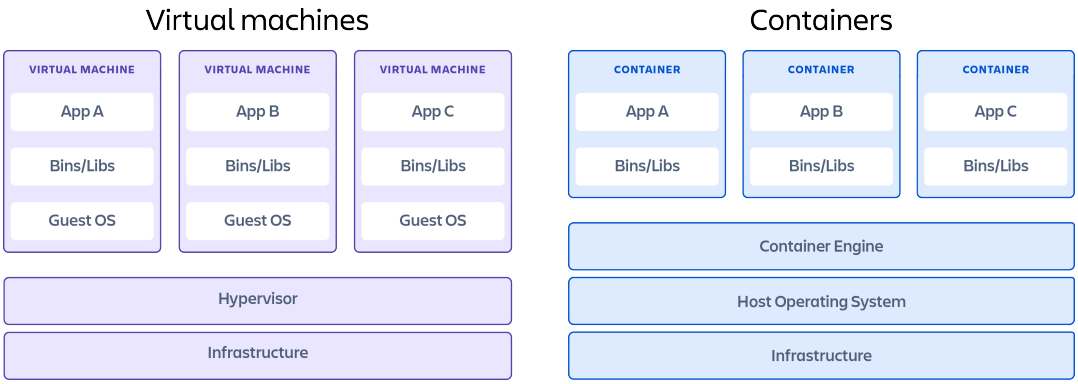

Figure 2.3: Containers vs VMs architecture (Buchanan n.d.)

Throughout the years, the concept of containerization and the approaches towards it kept increasing. History claims that it all started in 1979, with the launching of Unix V7. With

that Linux operating-system system call, called "chroot", people could change the root directory of a process, and its children's, to a new location in the file system which is only visible to a given process. That was the first approach towards containerization, as a host operating system was firstly divided by different users.

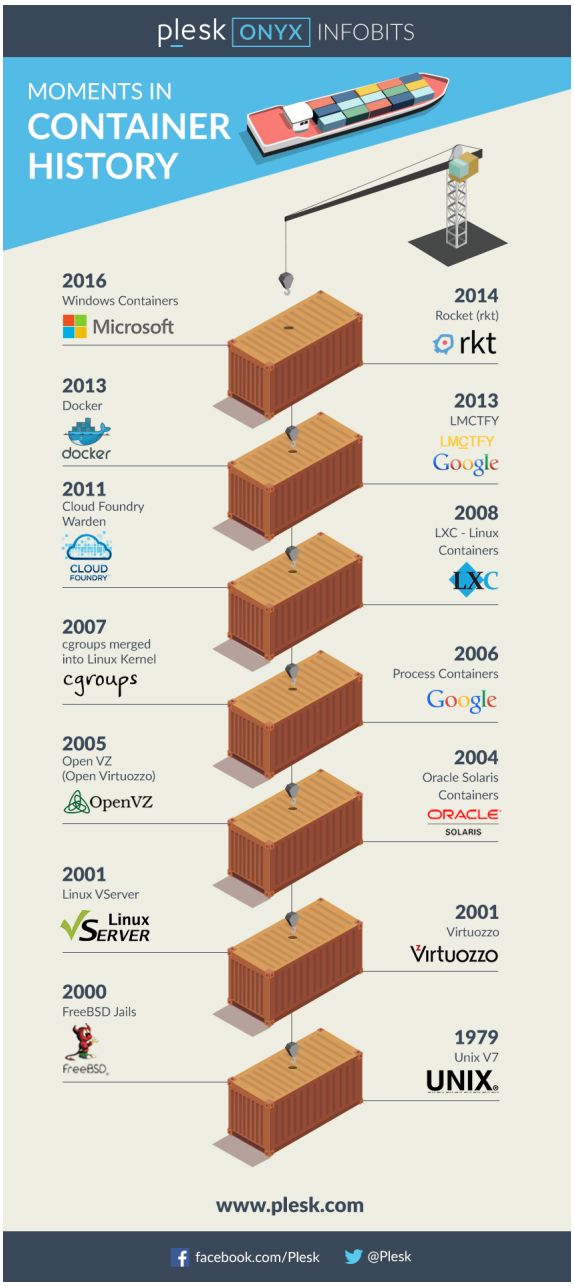

Figure 2.4: Container technologies over the years (Strotmann 2016)

There were several launches over the years (Figure 2.4) regarding containerization frameworks, but it was only in 2013 that Docker was launched, and it still remains the most loved and adopted container nowadays (Figure 2.4).

Docker containers

Docker is one of the most popular and widely used container runtime technologies. It allows building, shipping and running applications fastly and easily on any machine.

Docker brought an easier way to launching microservices and, with it, it introduced a new way of collaborating between teams. This represeted a turning point for DevOps, as it opened way for new technologies and ways of thinking.

With Docker's daemon running on top of the host's operational system, it facilitates the process of launching endless containers (Figure 2.5).

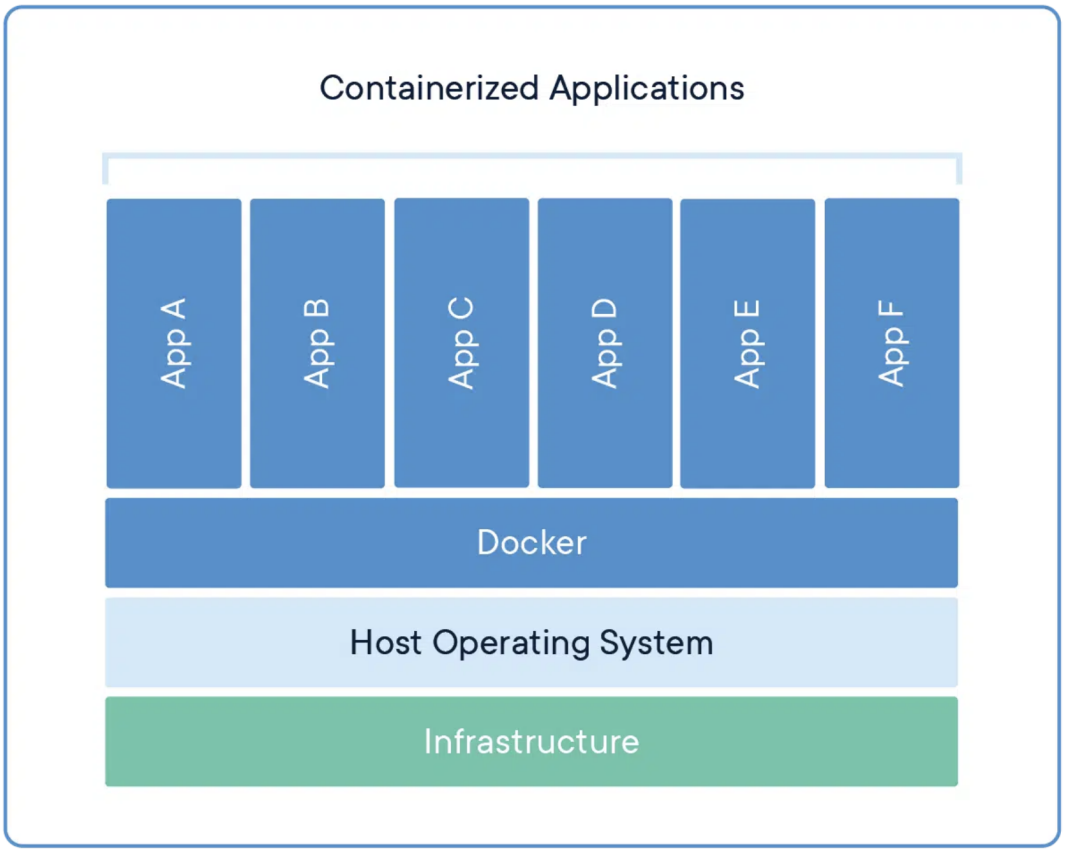

Figure 2.5: Docker architecture (Docker n.d.(a))

With Docker acting as an interface between the host's Operating System and each container, it enables the system administrator to easily manage and launch Docker Images and Docker Containers.

Users can store reusable Docker Images in a private or a public registry, and share them with their teams or even the whole community. Having Docker installed in each one's Desktop, developers can pull the Docker Image and run it on their machine as a Docker Container (Figure 2.6).

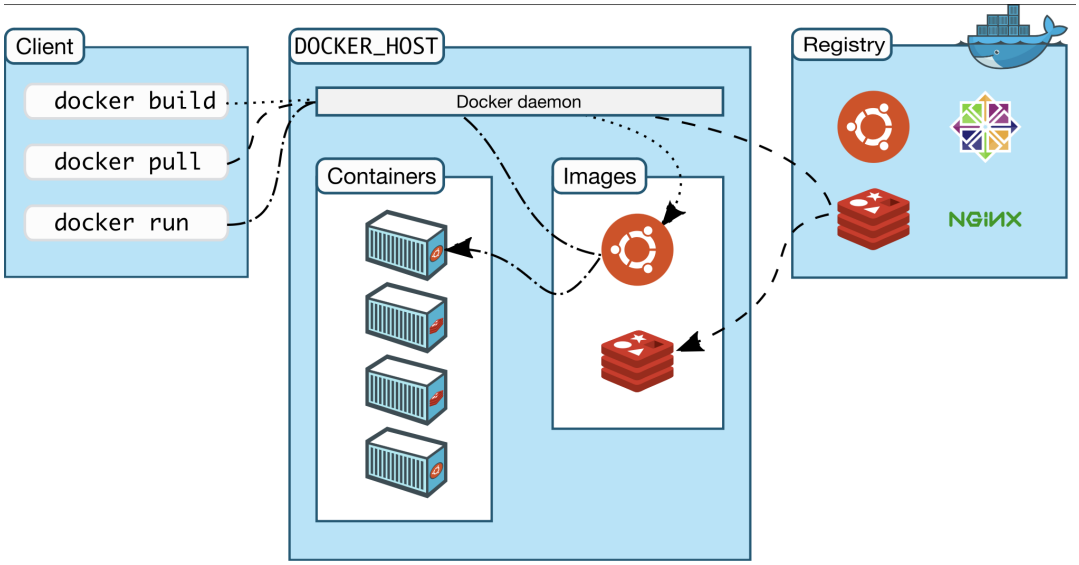

Figure 2.6: Docker flow (Docker n.d.(b))

Docker has its own image registry named DockerHub, which can be found under `https://hub.docker.com`. This enables developers to easily spin up an image of any distribution in a matter of seconds, either in a local machine or in a Serverless solution. For instance, developers can access DockerHub and pull a Linux Alpine image, and run it in their own Windows machine.

Container orchestration

What started as an advantage, fastly became a big burden to bear. The fact that containers were so easy to spin up, allied with the current trend of splitting the applications into several microservices, eventually created another problem. With so many containers hanging around, software engineers started having a bad time managing them.

Container orchestration is the concept associated to the automation, management, scaling and networking of containers (Redhat n.d.). It is what enables enterprises to scale and still be able to manage their large clusters, with huge amounts of containers.

With container orchestration, software engineers are able to automate and manage the following tasks (Redhat n.d.):

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Container availability
- Scaling or removing containers based on balancing workloads across your infrastructure
- Load balancing and traffic routing
- Monitoring container health
- Configuring applications based on the container in which they will run

- Keeping interactions between containers secure

Container orchestration tools

To ease up the burden of orchestrating all the containers, several frameworks have been launched over the years. The most commonly used ones being Kubernetes, OpenShift, Hashicorp Nomad and Docker Swarm.

However, despiste there being several tools being currently used, there is a bigger one amongs them: Kubernetes.

Kubernetes is an open source container orchestration tool developed by Google in 2014. In 2015, it started being maintained by the Cloud Native Computing Fundation. Kubernetes allows you to build applications that span multiple containers, schedule them across a cluster, scale them, and manage their health over time (Redhat n.d.).

With it, Kubernetes brings reliability, security and accessibility to everyone. Its Yaml based configuration makes it readable and easy to install and configure.

It has 4 major components:

- Data plane: provides containers with capacity such as CPU, memory, network, and storage
- Control plane: where all task assignments occur
- Kubelet: installed in each Node, and provides them with capacity such as CPU, memory, network, and storage
- Pod: A group of one or more containers deployed in a Node.

2.2 Serverless Computing

2.2.1 Overview of Serverless computing

Back in 2009, Netflix faced a problem that is very common amongst most IT companies. They wanted to scale, but they had a monolithic architecture which made it hard, and sometimes even impossible, to scale. Back in those days the concept of microservices did not even exist, yet they decided to decompose their monolithic application into several tiny standalone applications. That decision made them an early pioneer in what has become increasingly common today: transitioning from a monolith architecture to a microservices architecture (Harris n.d.). However, microservices need orchestrators to aid in the job of managing them, to ease a job which sometimes may prove to be costly and time consuming.

Not everything was solved with this new approach and, to help solve some of the remaining issues, a new architecture arose.

In this section, the aim is to introduce the concept of Serverless computing, its architecture, its strengths and weaknesses, and also its applicabilities.

Definition

Nowadays there is more than one definition to describe the meaning of Serverless. From the point of view of The Rise of Serverless Computing article, they define Serverless computing as "a platform that hides server usage from developers and runs code on-demand, automatically

scaled and billed only for the time the code is running" (Paul Castro 2019). On the other hand, the CNCF (Cloud Native Computing Foundation) defines Serverless as "the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform, and then executed, scaled, and billed in response to the exact demand needed at the moment" (CNCF n.d.).

All of these definitions touch very important factors of this concept but, independently of the definition adopted, one of the main concepts associated to Serverless is definitely that their users do not need to deal with server administration and hosting. Instead of this, they pass that responsibility to the cloud providers.

There are several services available in the Serverless concept, as people constantly bring in new ideas for Serverless approaches on different domains. Amazon Aurora is one of those examples which differs a bit from the definition that was previously presented while still calling it Serverless. On the one hand, it has powerful auto-scaling capabilities but, on the other hand, it requires minimum memory and CPU allocated to it and hence it does not scale to zero, resulting in ongoing costs.

Despite all this, Serverless architectures are commonly divided into two different categories: Backend as a Service (BaaS) and Function as a Service (FaaS). Both serve different purposes and bring their own unique value.

BaaS

BaaS, also known as MBaaS (Mobile Backend as a Service) is a model in which developers outsource the backend code from specific vendors. These provide pre-written software for common scenarios such as data and file storage, messaging and push notifications, authentication or message bus passing.

Figure 2.7: BaaS (Batschinski n.d.)

With this in mind, developers are able to focus on developing the frontend side of the application, while integrating with the BaaS part of the code via APIs and SDKs.

FaaS

FaaS is a model that allows developers to run self-contained functions in the cloud. It allows small pieces of code, represented as functions, to run for a limited amount of time. It is popularly being used for its real-time processing of data. It hides the underlying infrastructure, as it launches the instances needed to carry out the function successfully with only the resources that it requires, and releasing them at the end (Roberts 2018).

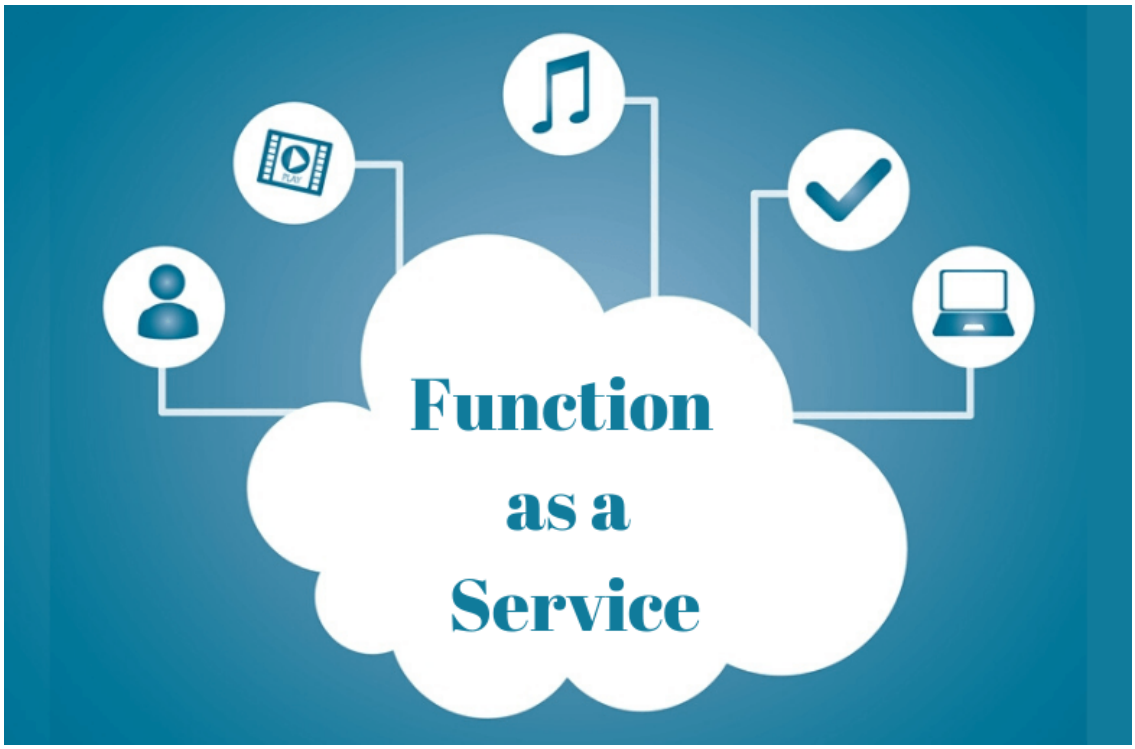

Figure 2.8: BaaS

The functions are stateless, meaning that they do not store any data between executions, and are triggered by events or HTTP request Paul Castro 2019. This allows it to scale easily. It describes a finer-grained deployment model where Serverless applications, consisting of one or more functions, are uploaded to a Cloud platform where they are scaled and billed according to usage.

2.2.2 Comparison of Serverless platforms

One of the major benefits of Serverless computing is that it enables developers to build applications without worrying about infrastructure management. This is possible because Serverless computing frameworks and platforms provide the necessary tools and services to deploy and run Serverless applications. In this chapter, we will discuss some of the popular Serverless computing frameworks and platforms that are publicly available.

Serverless platforms

AWS Lambda

Launched in 2014 by Amazon Web Services, AWS Lambda (Services 2021) is one of the most popular Serverless computing platforms. Lambda executes code only when triggered by an event and automatically scales the computing resources up or down based on the

demand. Due to its pay-as-you-go pricing model, users only pay for the execution time in which their code is running.

It enables developers to write functions in several programming languages, including Node.js, Python, Java, and Go. AWS Lambda supports event-driven computing, and it can be triggered by several AWS services such as S3, DynamoDB, and API Gateway. Additionally, Lambda has a built-in API Gateway that enables developers to create RESTful APIs for their applications.

AWS Lambda also provides some monitoring and logging tools, which can be used to track the performance and usage of the functions.

However, one of AWS Lambda's limitations is that it can only run for a maximum of 15 minutes, which may not be enough for certain applications (Services 2021).

Google Cloud Functions

Google Cloud Platform (GCP) is a Serverless computing platform provided by Google. It supports several programming languages such as Node.js, Python, and Go. Cloud Functions can be triggered by several Google Cloud services, such as Cloud Storage (Google n.d.). Additionally, Cloud Functions can be integrated with GCP services such as Firestore and BigQuery, but also with HTTP requests.

Just like AWS Lambda, Google Cloud Functions also provide a set of monitoring and logging tools, which can be used to track the performance and usage of the functions.

Azure Functions

Microsoft Azure Functions is a Serverless computing platform that enables developers to build and run event-driven applications. Azure Functions supports several programming languages such as Java, JavaScript, and Python. It can be triggered by several Azure services, such as Blob storage and Event Hubs. Additionally, Azure Functions can be integrated with other Azure services such as Azure Cosmos DB and Azure Event Grid.

These can be created and managed through the Azure Portal, a web-based interface that provides a central location for managing Azure resources

OpenFaaS

OpenFaaS (Function as a Service) is an open-source Serverless computing platform. It enables developers to build Serverless functions in any programming language. OpenFaaS has a built-in API Gateway that enables developers to create RESTful APIs for their applications.

The fact that it is an open-source tool, means it can be deployed on any infrastructure, including on-premises or public clouds. It can be used to build complex applications and microservices, as well as to run background tasks and automate workflows.

OpenFaaS also provides a set of monitoring and logging tools, which can be used to track the performance and usage of the functions.

Knative

Knative is an open-source Serverless computing platform that provides a set of building blocks for running Serverless applications on Kubernetes. It supports several programming languages such as Node.js, Java, and Go. Knative has several components such as Knative

Serving, which enables developers to deploy Serverless applications, and Knative Eventing, which provides a way to handle events in a Serverless manner.

Knative is designed to be extensible, meaning it can be customized to fit specific use cases and workloads.

It provides a set of built-in integrations with other cloud services, such as Google Cloud Storage, Amazon S3, and Microsoft Azure Blob Storage.

Knative also provides a set of monitoring and logging tools, which can be used to track the performance and usage of the functions.

Platform differences

Despite addressing the same issue, these platforms have unique features that differentiate them from each other and that might be important in the moment of choosing between one of them.

Vendor lock-in

Vendors are great in simplifying Cloud as they provide a pretty interface as an abstraction layer between the users and the host machines. However, even though most of the serverless tools are supported by a Cloud provider, that is not always the scenario.

AWS Lambda, Google Cloud Functions and Azure Functions are examples of Serverless Cloud vendor platforms that represent the ones that the majority of people use across the world. They provide a Serverless platform supported by the reliability and availability that the Cloud providers can assure.

On the other hand, open-source platforms give users the flexibility to install it on any machine and have full control over it. Both OpenFaaS and Knative are good examples of growing open-source projects. These can be installed on any infrastructure, whether it is on the Cloud or on premise. Knative is built on top of Kubernetes, which makes it easier to integrate it with a Kubernetes Cluster.

Programming languages supported

Even though AWS Lambda, Google Cloud Functions, and Azure Functions support multiple programming languages already, OpenFaaS and Knative are still able to overcome that number by supporting even more programming languages.

That might not be a very important problem for most developers since the more commonly used scripting and coding languages are already supported by the public Cloud vendors. However, it is still a an advantage in favour of the open-source projects.

Integration with cloud services

AWS Lambda, Google Cloud Functions, and Azure Functions provide several built-in integration with other services inside the same Cloud provider as themselves, which is a double edged sword. This eases the integration with other services in the same platform, which may prove to be an enormous advantage. However, on the other hand, OpenFaaS and Knative follow a different approach in which they provide a more general approach to integrating with cloud services, allow for a bigger versatility and foster freedom of choosing any vendor at any moment.

2.2.3 Best practices for Severless development

Serverless computing has gained popularity due to its flexibility and cost-effectiveness, but good development practices are essential for taking advantage of its full potential. This section outlines a set of best practices that developers should consider when building serverless applications.

Scalability

Serverless applications should be designed to allow scaling horizontally and handle different workloads efficiently.

For this, one should be using a microservices architecture. Breaking down complex applications into smaller, independent services, allows for each of them to be served by serverless functions. This enhances maintainability, fosters team autonomy, and allows for granular scaling of services.

Using an event-driven architecture to trigger functions in response to events such as HTTP requests, database changes, or message queue messages, is also very important (Roberts 2018). Having this inserted into the architecture's design, enables the application to autoscale without manual intervention.

Framework selection

Selecting the right runtime environment can be very important. Several factors should be taken into consideration, such as performance, serverless platform support and community contributions.

Using a well established framework will leverage the development process with templates, deployment automation and reusable components.

Single Responsibility Principle

Each function should have a clear, well-defined purpose. Avoid packing unrelated functionalities into a single function, as this can lead to code chaos and less maintainability.

Security

In case the serverless functions are managed by us, then we should consider securing the API with an API gateway. Those can manage incoming API traffic, such as rate limiting and request filtering, to prevent overloading the serverless functions and protect against possible Distributed Denial of Service (DDoS) attacks (Maayan 2023).

Another important security principle that should be applied is the Least Privilege Principle. Identity And Management (IAM) policies allows people to configure only the needed access per each person or group of people. It's a good way of keeping sensitive data and information out of undesired hands.

Infrastructure as Code

The deployment and maintenance of the serverless function should be kept as code, instead of being done through the user interface of the serverless platform. If written as code, the serverless functions be easily upgraded, maintained and reproducible.

Monitoring

The goal of serverless monitoring is to identify and resolve issues that may impact the performance and reliability of serverless applications and to optimize resource utilization for cost efficiency. It involves monitoring various aspects of serverless applications and infrastructure, including function execution, resource utilization and performance metrics (Maayan 2023).

Another important factor that should be monitored is the serverless infrastructure's cost. Graphs can help evaluate the cost-effectiveness to better enable organizations to take the best advantage of the serverless architecture's benefits.

2.2.4 Use cases

One of the advantages of serverless computing is its versatility and ease of use in a large amount of scenarios. Understanding the possible use cases is essential as it enables people to take advantage of its full potential.

According to AWS (AWS 2023), there are three major categories when we talk about serverless use cases:

- Asynchronous data processing
- Synchronous interactions
- Streaming

Asynchronous data processing

As the volume of data grows, coming from increasingly diverse sources, organizations find they need to move quickly to process this data to ensure they make faster, well-informed business decisions. To process data at scale, organizations need to elastically provision resources to manage the information they receive from mobile devices, applications, satellites, marketing and sales, operational data stores, infrastructure, and more AWS 2023.

These kinds of operations can be executed, asynchronously, using cloud functions to compute tasks, queuing services to create queues of data for the functions to consume, databases to store and consume incoming and outgoing data, and others.

Using AWS as a platform for an example scenario, an asynchronous data processing use case would be to have a file upload to a database or storage system, like the AWS S3 Bucket, which would trigger a Lambda Function after receiving the file. The Lambda Function transforms the image and then it can both upload the transformed image to another place, like another S3 Bucket, and even send an email notification for a target email list.

Synchronous interactions

Breaking an application into loosely coupled microservices can make things easier when it comes to scaling up and maintaining a growing application. They commonly communicate through APIs and, by default, the microservices wait for the response to continue the flow, just like a synchronous integration (AWS 2023).

For these operations, cloud functions can be used just like in the asynchronous integration. However, in this scenario, they should be invoked with a different trigger, like an API Gateway which can be invoked with the HTTP protocol.

With this in mind, people can separate the backend of their websites into a serverless function, which can be invoked either via a simple HTTP request, or using the AWS Software Development Kit for the language used.

Streaming

Streaming data allows gathering analytical insights and act upon them, but also presents a unique set of design and architectural challenges. Lambda and Amazon Kinesis are examples of platform services that can process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering (AWS 2023).

A possible streaming use case is one where people build an analytics application. Raw data gets stored in a DynamoDB table and, when items are written, updated, or deleted in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data can provide the relevant details to a Lambda function that generates custom metrics by aggregating the received raw data.

Chapter 3

Value analysis

In this section, the goal is to analyse the value that serverless computing presents to the customer. It is important to study what the customer wants and how this new architecture will impact their life, either it being with a positive or a negative impact. This must always be present, to never loose the focus on the goal to be achieved while keeping the customer satisfied.

The costumer can be both the developer, who builds and maintains the application, or the System Administrator who needs to create and maintain the whole infrastructure.

3.1 Value for the costumer

The primary value that customers receive from a Serverless architecture is increased reliability, scalability and availability of their applications. With the ability to automatically allocate resources on demand, Serverless architecture eliminates the need for manual configurations, which can lead to downtime and worse performance.

Additionally, a Serverless architecture enables customers to focus on their core business functions, rather than managing the underlying infrastructure, which can result in cost savings and improved efficiency.

3.2 Perceived Value

The perceived value of a Serverless architecture is mainly due to the ability to reduce operational costs and improve the for new applications and services. The flexibility and scalability offered by a Serverless architecture can help companies respond quickly to changing business needs and adjusting to the market demands. Additionally, the improved reliability and availability of applications and services can increase trust and credibility in the eyes of customers.

3.3 Benefits and sacrifices

Over the years, people have been using either monolithic or microservice architectures. Both are valid models that present both advantages and disadvantages, depending on the project's scenario.

Since, nowadays, the primary goal is to always leave room for a project to grow and scale, monolithic architectures will be discarded in order to enable the understanding of the major differences between Serverless and Microservices architectures, and what benefits and sacrifices does Serverless bring to the table.

3.3.1 Benefits

Costs

One of the major benefits of moving towards a Serverless model is the cost optimization. People are only charged for the period in which their code is running. This way of working also named "pay-as-you-go". Instead of paying for servers that often are idle, or whose computational capability is been fully utilized, people pay only for the amount of time that their code is running and for the specific resources that they need. To allow proper billing, since the execution time may be short, it is charged in fine-grained time units (like hundreds of milliseconds) and developers do not need to pay for overhead of servers creation or destruction (such as the time that Virtual Machines take while starting up). This cost model is very attractive to workloads that must run occasionally. On the other hand, this represents a big challenge for cloud providers which need to schedule and optimize cloud resources to split their computational capability amongst several users (**Serverless_martinfowler**).

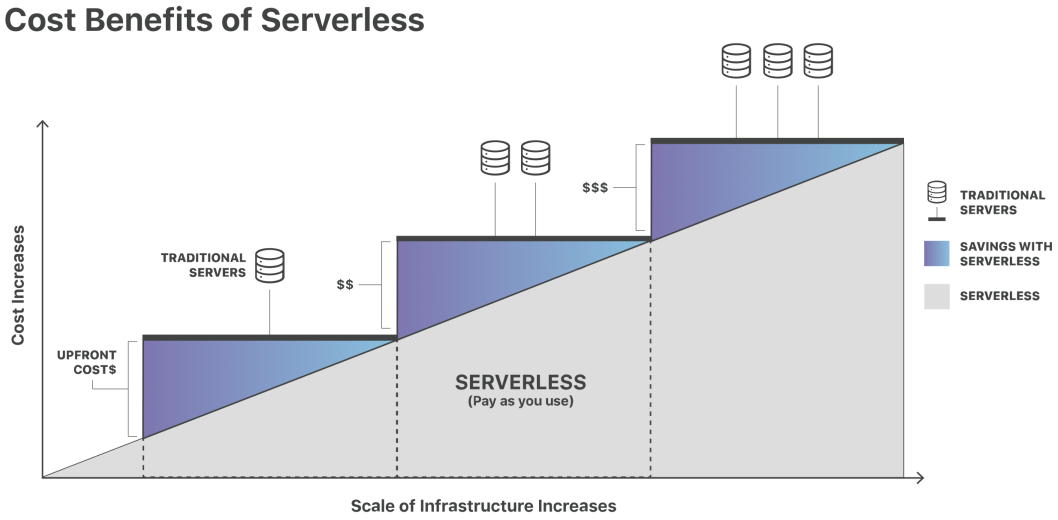

Figure 3.1: Cost benefits of Serverless Computing (Cloudflare n.d.)

Scalability

Since developers do not own and manage the servers that run their code, they stop having several responsibilities. It stops being part of their functions to scale the nodes or to create either automatic or horizontal scaling rules. Cloud providers have the job and responsibility of making sure that they are always available and properly scaled to avoid any throttling.

This is also a huge benefit for system administrators who cease having to apply the latest security updates, as well as a lot of other concerns that used to be part of their routines. Instead, they delegate that function to the cloud provider who is now responsible for making sure that the services are automatically scaled and that they are always available.

Flexibility

Running a function or any backend can be as simple as pressing a button when using a Serverless architecture. Developers can easily deploy and test new features without worrying

about the underlying infrastructure.

Fault tolerance

On a microservice architecture, to avoid having downtime, it is customary to have some sort of redundancy or replication. Most Cloud providers will offer you the following options to strengthen your infrastructure's availability (Azure 2022):

- Locally-Redundant Storage (LRS)
- Zone-Redundant Storage (ZRS)
- Geo-Redundant Storage (GRS)
- Read Access Geo-Redundant Storage (RA-GRS)

These are different options to split the infrastructure in either different local machines, different Cloud zones (e.g. eu-west-1 and eu-west-2) and even in a different geographical location. However, in a Serverless architecture, this is not something we need to opt from or maintain. It distributes the workload across multiple computing resources, which will improve the infrastructure's fault tolerance capability and reduce the risk of downtime.

3.3.2 Sacrifices

Cold start

What can be one of the major advantages of this Serverless architecture, can also be one of biggest disadvantages. Having ephemeral functions is a great benefit because it cuts down costs but, on the other hand, it will take longer than usual to respond. This will only occur on the first request, while the function is still inactive, but it is still something that will happen.

Latency

This second disadvantage is a bit related to the first one. There is the latency referred to in the previous section, but there is also the networking type of latency. Since the machine that will host our service could be anywhere in the world, it is needed to take into account that it could have some impact in the application performance and time of response. The response time of the function will consist on the sum of the time it takes for the request to travel from the client to the Serverless platform, the processing time of the function, and the response time on the way back to the client.

The fact that the service is running on a machine that can have several other resources running side by side to it, might imply that there could be a race for the computing resources of the machine, such as memory and CPU, which can also contribute to having added latency.

Debugging

Not managing the underlying infrastructure can appear to be very good, but there are also some downsides to it. Serverless applications can be composed of multiple loosely-couple functions that run in separated hosts. In case one error occurs in the whole process, it can get pretty challenging to trace and debug the issue as the source of the error can be in any of the world split machines.

Serverless applications are also ephemeral, meaning that they will cease to exists once they finish their job, making them even harder to debug since all the content will disappear unless stored in a permanent place. That can be done using some tools and best practices such as logging, monitoring, distributed tracing and testing frameworks (**Serverless_martinfowler**).

Security

Serverless functions handle sensitive data and could be subject to several security threats. Even though the Cloud provider should secure the machines as much as they can, it is still very important that the developers encode and secure all sensitive information. Even the code, which might not be encoded, can be subject to an attack and respective appropriation as their own.

There are also some compliance requirements, like data protection laws (i.e. RGPD), that these shared machines might not fulfill.

Another security issue is that an application might depend on third party libraries. If a new vulnerability is discovered and the dependency is not updated, that could be a critical entry point for any attacker who is aware its existence. Since most know vulnerabilities can be seen and studied in the Common Vulnerabilities and Exposures (CVE) list, most attackers would know what to look for an easily attack these shared machines.

An attack that can also occur, leading to a huge performance impact, is the DDoS attack. It consists of consuming an API several types, from multiple hosts, to simulate a huge incoming traffic. Even though the Serverless applications are restricted to a maximum of memory and CPU, it can cause resource exhaustion which will severely impact the performance and availability of the application.

Vendor lock in

Once everything is developed on top of a certain Cloud provider, either it being Azure, AWS, GCP, or any other, things start to be too attached. The SDK and API used to interact with the BaaS are different between vendors and, therefore, changing is not an option in a project that is already running in production due to all the risks associated to such a big migration.

Once we get locked to a vendor, we might loose opportunities that other competitors offer due to our lack of flexibility.

Chapter 4

Analysis and design

This chapter describes the analysis of the use case, followed by the design proposal to accomplish the defined goal. After getting to know the use case and the requisites, on the design section it will be presented an adoption guide for this transition to a serverless model.

4.1 Analysis

To showcase the Serverless architecture's usage and benefits, a use case of a real problem was conducted.

4.1.1 Use case

The increasing number and severity of forest fires has become a pressing global concern in the era of climate change and global warming. Rising temperatures, prolonged droughts, and exchanging weather patterns have severely increased the risk of wildfires and, therefore, human lives.

To mitigate the unpredictability of fires, this thesis presents an application designed to analyze satellite images of forested areas and detect potential fires in their early stages.

Using their Red Green Blue (RGB) pixel information, this application compares color values in the image to a predefined "Fire color" reference. A threshold is then applied to determine potential fire hotspots by assessing its proximity to the "Fire color" value. A grayscale output image is then generated with the detected fire hotspots highlighted in red, offering a clear visual representation of potential fire areas.

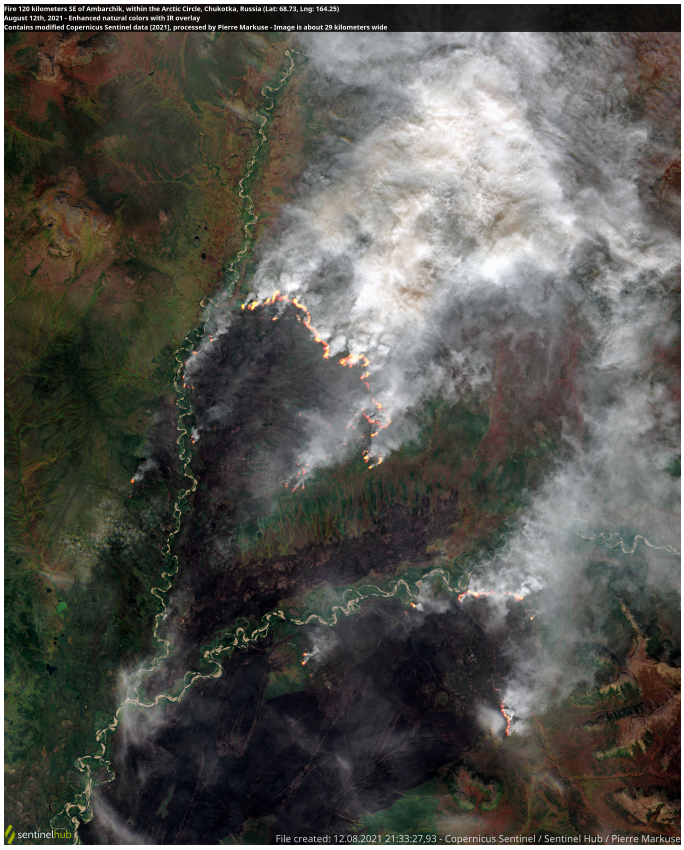

Figure 4.1: Raw fire image

To enhance the processing of these high quality forest images, the initial image should be firstly split into multiple smaller segments that get analysed, pixel by pixel, by an individual thread. Multiple threads concurrently execute this analysis on their assigned image segment, and then they write the resulting image to another file. The transformed segments are then merged into a single image once again, leading to the final and full image.

However, the processing of these images, pixel by pixel, can prove itself to be quite demanding, making it a task that's not suited for the weakest infrastructures.

4.2 Architecture design

The foundation of our serverless forest fire detection system came from a pre-existing Java Backend application. The following decisions and considerations had this premise always present, in order to ensure an easy integration between the serverless function and the existing Java environment.

4.2.1 Serverless platform

By using a Serverless function, we are able to have an infrastructure that is able to adjust itself to the incoming data load without ever compromising the data consumption performance. This would leverage the scalability and cost-effectiveness of serverless computing to process data on demand.

To tackle this computation complexity, the complex logic can be abstracted to a Serverless function. Instead of executing that logic as part of a backend microservice, the needed processing capability can be obtained through a remote function. AWS Lambda is the proposed tool and platform for this remote serverless function.

To select the appropriate platform for hosting our serverless function, compatibility with Java was the main consideration. Given the fact that the application was developed in Java, it was imperative to find a platform that was able to accommodate this programming language. To facilitate the usage and integration with the Java application, a user friendly user interface was also a must. Also, in order to spend as little as possible, a cost-free platform was a nice to have.

The OnPremise solutions, such as the Knative and the OpenWhisk, had the advantage until it was found that AWS granted a cost-free function solution for the first one million requests. Together with a user friendly and reliable interface, AWS turned out to be the best solution. After the free tier, AWS grants the pay-as-you-go model in which the user does not get charged for resources which he is not consuming. Even though it grants lesser customization and control, the existing customization was just what was needed to carry out the current study. AWS Lambda also emerged as a compelling choice, as it offered native support for Java alongside various other programming languages.

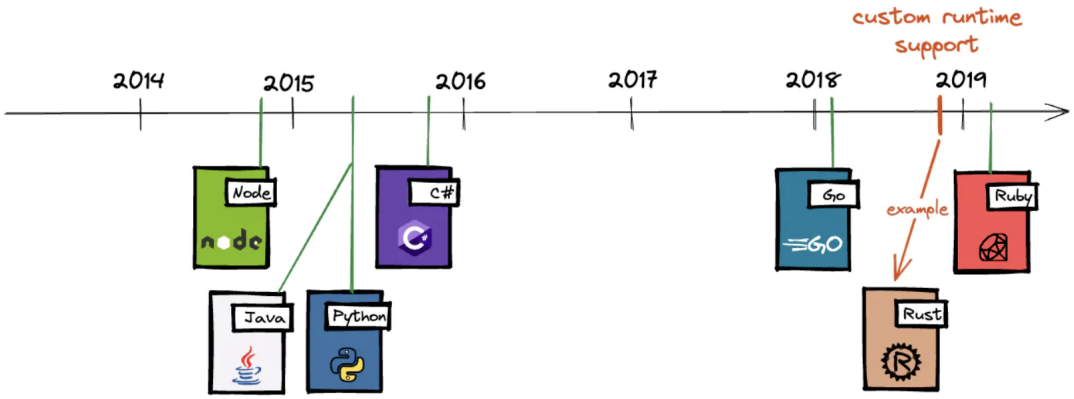

Figure 4.2: Lambda coding languages throughout the years (Schmidt 2023)

AWS Lambda also provides a free tier version for the first 1 million requests, which facilitated in this interaction with the tool. It's a great way to talk developers into trying this platform for initial proofs of concept in order to verify for themselves the real value and advantages that it can bring.

This function can be invoked in several ways. It can be triggered through an HTTP/API Gateway, AWS SDKs, Amazon S3, AWS Console, Amazon SQS, etc. To integrate the backend of our application with the serverless function, since the Backend is written in Java, an AWS SDK can be used. Its result are returned by the SDK function that invokes it.

4.2.2 Components diagram

To better help visualize this architecture, its main components were bundled into the components diagram that can be seen in the Figure 4.3.

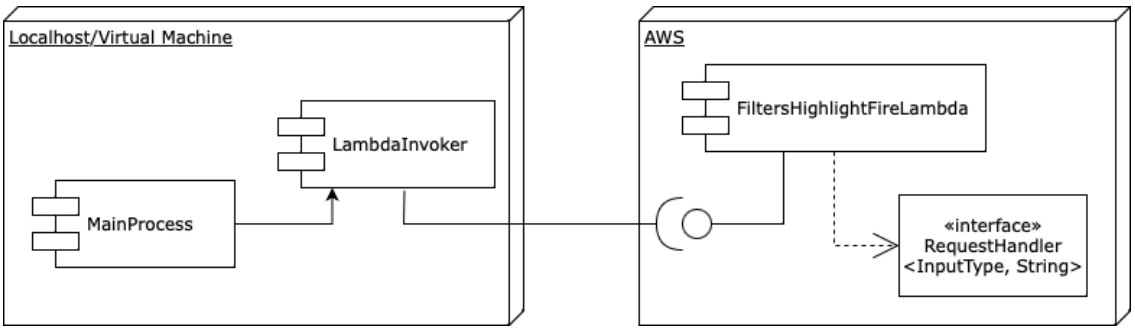

Figure 4.3: Components diagram

Despite the simplicity of this illustration, it demonstrates that the main process/thread can easily generate any amount of traffic and simply invoke the AWS Lambda function, because all it needs to do is invoke the Lambda HTTP API with its LambdaInvoker component, and all the computational complexity will be abstracted.

In order for the AWS Lambda function to work and be able to recognize incoming requests, it must implement the RequestHandler interface and override its methods. Following these instructions, together with a correct implementation of the AWS SDK on the client side, will ensure a smooth communication between both peers.

4.3 Adoption guide

This section provides a step-by-step adoption guide for the process of integrating the AWS Lambda function into a Java application. The goal of this guide is to enable any Java application team to easily start using AWS Lambda functions while walking through the whole transition process and preserving the integrity of the pre-existing codebase.

4.3.1 AWS Lambda creation

In this first step, it is important to create the AWS Lambda function, which will be a core component of the serverless forest fire detection system. Developers should start with the Lambda Function because it defines the input context that it expects to receive, and which the Lambda invoking method will need to provide afterwards.

For the communication between the existing Java Backend application and the newly created AWS Lambda function to work, a series of essential configuration steps need to be taken.

Create a new Java project

The initial step in configuring the AWS Lambda function involves creating a new Java project tailored for the AWS Lambda integration. This project serves as the bridge between our existing code base and the Lambda execution.

There is a Web Editor in the AWS Lambda user interface but, even though AWS Lambda supports several different programming languages, not all of them are allowed in the Web Editor. Java is one of those exceptions, as can be seen in the Figure 4.4.

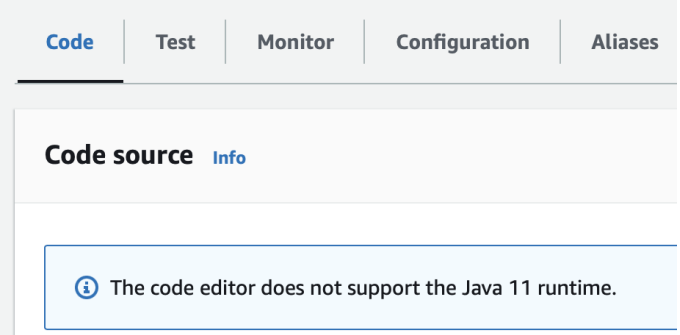

Figure 4.4: Incompatible Java language in AWS Lambda

To overcome that, one must build the Java application on their machine and only then they should upload the resulting Jar file for testing and execution purposes, directly on the AWS Lambda user interface.

Import AWS dependencies

In order to properly communicate with the AWS Lambda function, AWS has released several SDKs. To starting using the AWS Java SDK for Lambda functions (AWS n.d.(a)), developers must firstly import all the needed dependencies for the code to compile successfully.

Published on the public Maven artifact repository "mvnrepository", this SDK can be imported as a dependency in the project's "pom.xml" by fetching the groupid, artifactid and specific version that should be used. This is where the artifact can be found: `https://mvnrepository.com/artifact/com.amazonaws/aws-lambda-java-core`

Create the handler function

In order to allow the Lambda function to process events, a function handler must be implemented according to the framework rules. When the function is invoked, Lambda looks for the handler method and runs it. The function will then run until either the handler returns a response, exits, or times out.

Three important rules must be followed (AWS n.d.(a)):

1- The Java class needs to implement the RequestHandler interface from the com.amazonaws.services.lambda library (AWS n.d.(a)).

```
1 public class LambdaHandler implements RequestHandler<
    InputObject, OutputObject>{
```

2- The interface needs to define two important objects. The first one is the input object, which needs to be provided each time the function gets invoked. The second one is the output object, which gets returned once the function is successfully executed. As an example, the following code sample shows how a class declaration could look like.

```
1 ... implements RequestHandler<InputObject, OutputObject>{
```

3- The handler method needs to Override the parent method from the interface, in order for it to be active.

```
@Override
public OutputObject handleRequest(InputObject event, Context
    context)
{
  // computing logic goes here

  OutputObject resultObject = new OutputObject(result);
  return resultObject;
}
```

Package and upload the function

Since this Java project is managed by the Maven package manager, the packaging process very straightforward and simple.

The developer just needs to take advantage of the pre-existing Maven phase "package" which, when run, creates a JAR file under the "target/" folder of the root directory.

Uploading the JAR file to the AWS Lambda function can be done in one of two ways: through the user interface or via the AWS CLI.

Even though the AWS CLI is more powerful and makes this process automatable, the user interface interface can also be a strong option due to its user friendliness and easy access for less experienced Cloud users.

(Optional) Create a function trigger or destination

Now that the application is deployed and tested, AWS Lambda needs to be made available so that the client application is able to reach it. There are several ways to do it. Each with its own advantages and disadvantages.

Lots of resources are available to receive incoming requests to this Lambda Function, such as an HTTP API Gateway, which makes this function available through a URL, which then can be used by the client application. The Lambda function can also be invoked in many other ways, such as through an S3 Bucket, which is a storage location that will notify the function for any newly added files.

However, besides all these ways of invoking the function, there is another one which does not require an AWS resource behind it. That is with the AWS SDK for Java applications, which allows the users to create a client that is able to invoke the function by providing only three attributes:

- The function Amazon Resource Name (ARN), which is an identifier composed by a unique identifier such as *arn:aws:lambda:us-east-2:037523038573:function:highlight_fire*
- The AWS region, which can be a value such as *us-east-2*, for the US East (Ohio) location

- The API Credentials, which are composed of an Access Key and a Secret Key

4.3.2 Implement the AWS SDK onto the existing Java application

Now that the AWS Lambda is ready to run the computational logic the developers want to abstract, and is listening for any incoming requests, the client application also needs to undergo some adjustments. That is where the AWS SDK steps in.

Afterwards, and in order to isolate this AWS Lambda behaviour from the rest of the application into a centralized place, it is a good pattern to create a Java class for that purpose.

Add the needed maven dependencies

The first step is usually to prepare the environment to start working. In order to communicate successfully with the Lambda function, we are going to use a Java library named *aws-java-sdk-lambda* which can be retrieved at the following Maven repository: `https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-lambda`

In order to start using a dependency in a Java project that is managed by Maven as the package manager, we just need to add the following lines to to the pom.xml file, under the "dependencies" element, while specifying one of the available versions in the URL above.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-lambda</artifactId>
    <version>1.12.547</version>
</dependency>
```

Instantiate the client

In order to start using the AWS SDK and some of its functions, we need to create an instance of the AWS Lambda client.

This is the stage in which the developer starts defining the connection attributes which are needed for the following requests: the AWS region, the API credentials and the function's ARN.

Create and use the defined DTO for the input context

Once the client is initialized, the payload must be built in order to match the previously defined input context, on the Lambda function's end.

This payload data needs to be a Json String. In order to build it, either the String can be built with a String.format() command, or the input object can be serialized into a Json string automatically, using the right several methods such as a public library like JsonSerializer, or by adjusting the toString() method of this Data Transfer Object (DTO).

Replace the computational logic with the Lambda function invoking

In this final step, the developer just needs to head to those places that are meant to be moved to the Lambda function, and replace that code by an instantiation of the invoker class.

This method may provoke several types of errors, such as connectivity errors, parsing errors, syntax errors, "entity too large" errors, and others. Therefore, it is important to surround this Lambda invoking logic with a try/catch mechanism in order to prevent unexpected and unhandled incoming errors.

Chapter 5

Implementation

Given the designed adoption guide from the previous chapter, the goal of this chapter is to test it, by following all the steps on top of a real use case scenario. This use case, also presented in the previous chapter, consists on a Java application that wants to start using an AWS Lambda function in order to start taking advantage of its benefits.

While following the predefined and predicted steps in the guide, some challenges were encountered. Despite the setbacks, the implementation was successful, and that is what is about to be shown.

5.1 The use case in depth

As previously introduced, the use case consists on an application that attempts to, intelligently, identify possible fire spots by analysing an image and comparing each pixel's color value to a predefined "fire color" value. The pixels closer to the "fire color" value will turn red, while all the others will turn grey.

The application processes an image as a matrix of Color objects, representing each image pixel, and then generates a new image based on the analysis result. In order to better compute this analysis which, depending on the image size, could take some time and resources to process, the application splits the image into several tinier images, which get assigned to a thread for the threshold processing. The Figure 5.1 illustrates this exact behaviour

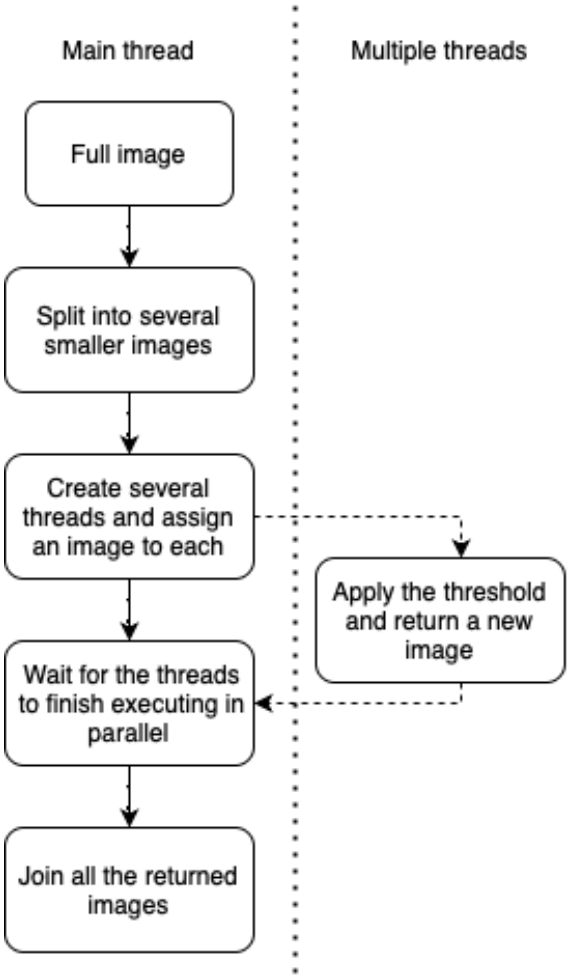

Figure 5.1: Application flowchart

Once all the threads have finished their execution, the main thread continues its processing into the final step: joining all the images into a single one, ready to be written into the filesystem.

5.1.1 Application startup and inputs

When starting the application, the user must provide a couple of configurations, that will be crucial for the upcoming execution. Those are:

- **Number of threads**: This will define the amount of small images that will be created. If too large, the cost of splitting them may not justify this parallelism and/or remote execution, in the case of using a Function As A Service
- **Threshold**: The threshold to be used in the mathematical formula in which the fire spot gets identified
- **Source image**: The image to be analysed for possible fire outages
- **Target location**: The place where the newly generated image will be written to

5.1.2 Calculation formulas

In order to better generate the resulting image, a few mathematical formulas were used in this implementation process.

Firstly, in order to generate the grayscale pixel of those places where fire was not detected, the red, green and blue values where replaced by their average.

$$\text{avg}=\frac{(\text{r} + \text{g} + \text{b})}{3}$$

The result of this equation will be a similar pixel to the previous, in order not to disrupt the whole image structure, but it will get a lot darker for those pixels, in a grayscale format.

However, for the fire spot calculation, the logic underneath is a little more complex. To know if a pixel is a possible fire spot, there are several high precision studies that are quite complex and out of scope for this thesis' scope.

In order to simplify this, some ground rules will be followed in order to determine the fire pixels:

- The red value of the RGB pixel is higher than the average values of the red, green and blue values of the pixel multiplied by a threshold

$$\text{r} = \frac{(\text{r} + \text{g} + \text{b})}{3} * \text{threshold}$$

- The green value is bellow some value vG
- The blue value is bellow some value vB

If the three rules defined above are true, then the pixel is translated into a red pixel in the output image, meaning that it will keep the following RGB properties: Red = 255, Green = 0 and Blue = 0.

As an example, given a threshold of 1.35, vG = 100 and vB = 200, then a pixel with red value pR, green value pG and blue value pB is considered a red pixel if:

$$\text{pR} = \frac{(pR + pG + pB)}{3} * \ 1.35 \ \textit{and} \ \text{pG} < 100 \ \textit{and} \ \text{pB} < 200$$

Therefore, a pixel with RGB values of (250, 100, 150) is converted into a red pixel as it follows the three pre-defined rules.

5.1.3 Execution mode

This application was developed with the main purpose of showing the advantages of multi-threading for application that were highly demanding in terms of computation resources.

For small images and low definition images, meaning few pixels, the single threaded approach was often the best approach. However, when it comes to high definition satellite image, with thousands of pixels, it can get pretty demanding for the Java Virtual Machine (JVM) but also for the host machine itself.

In the multi-threaded approach, the main application splits the main image into several tinier images and assigns a single thread per each created image, therefore splitting the amount of assigned pixels per thread. In this case, instead of handling all the threads in the host machine, the goal is to query the Lambda function to execute that code leaving the thread with the single responsibility of querying the Lambda function and waiting for the response, which ends up being a lot lighter.

5.2 Migration process

Now that the application is developed and up-and-running, it is time to start testing the previously created guide for a successful partial migration to the serverless function.

As mentioned in the previous subsection, the computational logic will be migrated to the Lambda function, leaving the Java threads with the responsibility of invoking it and handling the response while parsing the incoming and outgoing data, if needed.

Next, the implementation technical steps will be presented according to the proposed guide steps, in the same order, so that it can be validated.

5.2.1 Create a new Java project

To create the Java application that will run on the AWS Lambda, the first step is to generate a Maven project. It can be done using the Maven Command Line Interface (CLI) named "mvn" and its "archetype" phase.

```
mvn archetype:generate
  -DgroupId=<group_id>
  -DartifactId=<artifact_id>
  -DarchetypeGroupId=<arch_group_id>
  -DarchetypeArtifactId=<arch_artifact_id>
```

Listing 5.1: Generate maven project

This command above will make sure that a template Java project gets created with all the Maven required files and with a recommended structure.

5.2.2 Import AWS dependencies

Then, it is essential to import the necessary dependencies for the fire detection application to be able to connect to the Lambda function. Since we are using a Maven project, we only need to update the project's "pom.xml" where the most important Maven project-specific settings lay.

As described in the guide, the needed dependency is an artifact with the artifactId "aws-lambda-java-core". Below is a partial code sample of how the "pom.xml" should look like once the new dependency is added.

```
<dependencies>
    <!-- AWS SDK for Java Lambda -->
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-core</artifactId>
```

```
6           <version>1.2.2</version> <!-- Use the latest version
    -->
7       </dependency>
```

Listing 5.2: AWS SDK pom.xml dependency

5.2.3 Define the input DTO

Knowing the data that the function will need to execute the image processing, the next step consists on creating a object that gathers all the data that is needed in a single place.

Therefore, and in order to better prepare for the following step, the "InputType" DTO was created.

This object contains two important attributes:

- String: imageStr
- float: threshold

Due to the fact that the AWS Lambda framework does not support transferring images or even color arrays, the image had to be converted into an encoded string for transferring purposes.

Together with the threshold, this Data Transfer Object contains all the needed data for the function to execute all the needed logic.

5.2.4 Serialization of the images

The AWS Lambda function comes with a great constraint when coupled with the AWS SDK for Java application. It can only receive primitive object types as parameter since the input object receives a Json structure, therefore it does not allow the application to provide the image, which is a matrix (an array of arrays) of Color typed objects.

To counter this, both the invoking application and the Lambda function itself must be able to serialize and deserialize the image to a String representation of itself, making it possible to the passed from one end to the other.

Looking at the following code sample, an image is converted to a base64 String before being passed onto the Json structure that is passed onto the Lambda function.

```
1  try {
2      // Create a BufferedImage from the Color[][] matrix
3      int width = image.length;
4      int height = image[0].length;
5      BufferedImage bufferedImage = new BufferedImage(width,
       height, BufferedImage.TYPE_INT_RGB);
6
7      for (int x = 0; x < width; x++) {
8          for (int y = 0; y < height; y++) {
9              bufferedImage.setRGB(x, y, image[x][y].getRGB());
10         }
11     }
12
```

```
13      // Convert BufferedImage to byte array
14      ByteArrayOutputStream byteArrayOutputStream = new
    ByteArrayOutputStream();
15      ImageIO.write(bufferedImage, "png", byteArrayOutputStream
    );
16      byte[] imageBytes = byteArrayOutputStream.toByteArray();
17
18      return Base64.getEncoder().encodeToString(imageBytes);
19  } catch (IOException e) {
20      e.printStackTrace();
21      return null;
22  }
```

In order to accomplish this conversion, the Base64 library is used to encode a byte typed array into a base64 string. However, given that the image is initially in the format of a Color[][] matrix, the image needs to be transformed into the desired object type.

The first step is to clone the image onto a BufferedImage and, afterwards, convert it to a byte array so that the used library can do its job and get the image into the desired type.

This utility method, as well as its opposite (which can be consulted in the Appendix A), need to exist both in the Java application and in the Lambda function Jar.

5.2.5 Create the handler function

In this step, the developers need to start developing the function itself. Going into the *"src/main/java/com/example"* folder, which may vary depending on the given package name, a new Java file needs to get created. The package "com.example" is the default one, so that should be resulting one out of this template project.

According to the guide, there were three rules to follow according to the framework:

- 1. Implementing the AWS Lambda framework interface
- 2. Defining the interface objects
- 3. Overriding the handleRequest method

In the first requirement, it was defined that the RequestHandler library needed to be imported and its interface imported, as can be seen in the following code sample.

```
1  import com.amazonaws.services.lambda.runtime.RequestHandler;
2
3  public class FiltersHighlightFireLambda implements
       RequestHandler<InputType, String> {
```

The second requirement was also accomplished in the previous code sample, as we are defining that the InputType object (a custom DTO) is the input object that the Lambda function will receive, and we are also defining the that Lambda function will return an object of type String.

Instead of a String as the output object, it is also possible to define another DTO for a more complex return data structure.

Then, fulfilling the third requirement, the last part consists on overriding the interface method named "handleRequest".

```
1     @Override
2     public String handleRequest(InputType input, Context
    context) {
3         // Lambda invoking logic
4
5         return "function result";
6     }
```

By doing this, we are ensuring that this method gets executed instead of the default implementation that the interface might have, on some higher class in the Java hierarchy, making this the entrypoint when it comes to executing a specific Lambda function.

5.2.6 Package and upload the function

Now that the Java function is fully developed and ready, the next step is to package it and upload it to the AWS platform.

Using the Maven CLI, the JAR artifact is a simple command away from being built.

```
1 mvn clean package
```

The command in the code sample above compiles the Maven project into a JAR file, which gets created in the "target/" folder by default.

Developers will be able to find an option in AWS Lambda to upload the JAR file, in a quite straightforward way, as it can be seen in the Figure 5.2.

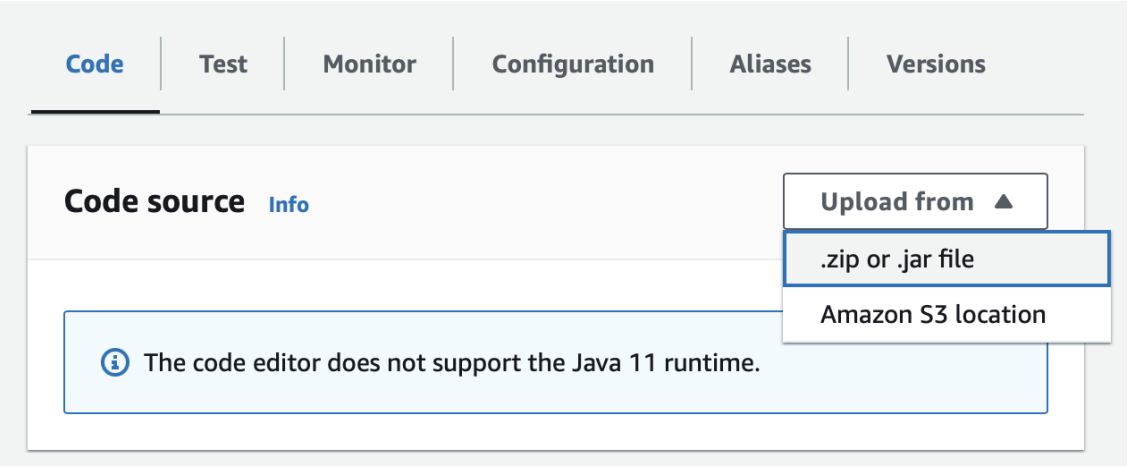

Figure 5.2: Upload Lambda function Jar file

After uploaded, developers can head to the "Test" tab and perform some different tests with different parameters. By changing the keys and values of the Json parameter structure, developers can validate the functionality of their recently uploaded application.

5.2.7 Add the needed maven dependencies to the Java application

Now that the AWS Lambda is ready and tested, it is time to start adjusting the local Java application to start communicating with the Lambda function.

In a similar fashion to the AWS Lambda Java application, the local application also needs some external dependencies to properly function.

On the "pom.xml" file, the following dependency must be added for the developers to be able to start using some objects that facilitate this integration.

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-lambda</artifactId>
        <version>1.12.547</version>
    </dependency>
```

5.2.8 Instantiate a client

It is good practice to isolate components in different classes, or even modules. In this case, to avoid conflicts and confusing the already existing code of a possible large monolith or of a broad microservices software, a new class will be created.

The first thing to configure in this new class is the AWS Credentials to access the AWS Lambda, which can be inserted into a BasicAWSCredentials object, from the *com.amazonaws.auth* package.

```
BasicAWSCredentials awsCreds = new BasicAWSCredentials("
   AKIAQRPEMEFXJNDRCVWZ", "lzYP6We2r3m+SdfbFMtJ//
   WWkK979VRYKjfLM52P");
```

Then, having initialized the credentials object, it can be injected into the AWSLambda-ClientBuilder from the *com.amazonaws.services.lambda* package. Through this object, we are able to query the Lambda function with custom payloads.

```
AWSLambda lambdaClient;
lambdaClient = AWSLambdaClientBuilder
    .standard()
    .withRegion(Regions.US_EAST_2)
    .withCredentials(new AWSStaticCredentialsProvider(
   awsCreds))
    .build();
```

5.2.9 Create and use the defined DTO for the input context

Knowing the InputType object that was defined on the AWS Lambda side of things, developers need to follow that same structure when invoking the Lambda function using the AWSLambda object.

Regardless of way of building it, the important thing to do here is to convert this DTO into a Json string structure. In the end, whether through a conversion library or through any other way, the important thing is that the input string ends up in a way similar to the following code sample.

```
String inputPayload = String.format("{\"imageStr\": \"%s\",
   \"threshold\": \"%.2f\"}", imageStr, threshold);
```

After this, all is set to start invoking the AWS Lambda function. The request is done using the InvokeRequest and InvokeResponse objects of the *com.amazonaws.services.lambda.model* package.

```
String functionName = "arn:aws:lambda:us-east-2:037523038524:
   function:highlight_fire";

InvokeRequest invokeRequest = new InvokeRequest()
    .withFunctionName(functionName)
    .withPayload(inputPayload);

InvokeResult invokeResult = lambdaClient.invoke(invokeRequest
   );
```

5.2.10 Replace the computational logic with the Lambda function invoking

Running in parallel, the threads compute all the image processing logic in a Java class. The place which used to contain all the computing logic, now hold the Lambda invoking method only.

Previous method:

```
    // Highlight Fires.
    public Color[][] HighLightFireFilter(String outputFile,
   float threshold) throws IOException {
        Color[][] tmp = Utils.copyImage(image);
        for (int i = 0; i < tmp.length; i++) {
            for (int j = 0; j < tmp[i].length; j++) {
                // fetches values of each pixel
                Color pixel = tmp[i][j];
                int r = pixel.getRed();
                int g = pixel.getGreen();
                int b = pixel.getBlue();
                // takes average of color values
                int avg = (r + g + b) / 3;
```

```
13                  if (r > avg * threshold && g < 100 && b <
    200)
14                      // outputs red pixel
15                      tmp[i][j] = new Color(255, 0, 0);
16                  else
17                      tmp[i][j] = new Color(avg, avg, avg);
18
19              }
20          }
21          return tmp;
22      }
```

New method:

```
1       @Override
2       public Color[][] call() throws Exception {
3           FiltersHighlightFire filter = new
    FiltersHighlightFire(images.get(0));
4
5           Color[][] result = filter.HighLightFireFilter(
    outputFilePath, threshold);
6           Utils.writeImage(result, outputFilePath);
7
8           return result;
9       }
```

Given that the whole Lambda invoking logic logic is isolated in the FiltersHighlightFire class, the processor only needs to use that method and wait for the response.

Chapter 6

Evaluation

The adoption of serverless function in distributed architectures allows better performance and scalability, while often granting a cost decrease.

In this chapter, it is presented a way to evaluate the results of the use case in order to assess the effectiveness of serverless computing in addressing a real-world problem. Then, there is also a section in which these tools are used against the use case previously presented in order to take some conclusions based on the gathered data.

6.1 Performance assessing metrics and tools

This section provides an overview of the evaluation process used for assessing the benefits of adding a serverless integration in the context of a Java application. The section begins by outlining the methodologies, metrics and tools employed for performance measurement. Additionally, this chapter introduces AWS CloudWatch metrics on the AWS Lambda side as a key component of the evaluation process.

Methodologies

In order to come up with valuable data to evaluate this solution as best as possible, some methodologies were put in place.

Firstly, in order to extract the duration metrics, the Java application was slightly changed to record the timestamps previous and after the function invocation. With simple maths, this method returns the duration time to the standard output, in milliseconds.

Another important thing to take into consideration is the data size. In order to have a relevant amount of data, a decision was taken to have at least 5 executions. This allows a better data analysis through maximums, minimums, mediums and standard deviations per scenario.

Using CloudWatch, in AWS, these and several other metrics are available, supported by graphs, to assess the performance and stability of our serverless function.

Metrics

The use case was designed to compare the performance of a serverless architecture with a traditional hosted microservice architecture, with a focus on processing large volumes of data quickly and efficiently. Therefore, the goal is measure some important metrics and Key Performance Indicators (KPIs) that bring some value to these approaches.

The proposed Serverless function aims to solve the problem of processing large volumes of data quickly and efficiently, without the need for always-on infrastructure. To evaluate the effectiveness of this approach against traditional methodologies, several metrics can be considered.

Firstly, the scalability of the Serverless architecture can be compared to the scalability of a traditional hosted architecture. In a traditional architecture, the infrastructure needs to be provisioned by excess, so that it is able to handle the maximum expected load. This means that, in periods of lower demand, the infrastructure is not fully used, leading to higher costs. In contrast, the Serverless architecture can scale dynamically, processing only the required data and, therefore, reducing costs. To evaluate this property, a load test can be run against the virtual machine and the same test against the Serverless function. Looking at the behaviour of each platform, this scalability property will come up as the virtual machine will run out of memory, or space, or some other resource that represents the maximum amount of load that it can take. Unlike that, the serverless function will simply scale horizontally in order to handle the current amount of incoming requests.

Another metric that can be used to evaluate the effectiveness of the Serverless architecture is the performance of data processing, more precisely the execution duration. Traditional architectures may have higher processing performance in some cases due to the dedicated resources, but they are less flexible in adapting to varying workloads. In contrast, the Serverless architecture can process data on demand, without the need to maintain always-on infrastructure. Therefore, the Serverless architecture can offer good performance and flexibility. On the other hand, it is also important to consider the networking problems and possible delays. The communication between the host machine and the serverless location can sometimes turn out to be more costly than the execution itself.

Finally, the cost of the Serverless architecture can be compared to the cost of a traditional architecture. Traditional architectures require a higher investment in hardware and infrastructure maintenance, whereas Serverless computing operates on a pay-as-you-go model, which can lead to lower costs. Additionally, the Serverless architecture can offer cost savings due to the efficient use of resources, avoiding the under usage of infrastructure.

Tools

Since the metrics required are quite straightforward to obtain, there is no need for any extra sophisticated tool. Through the Java application itself it is possible to print the timestamps prior and after the function's execution, leading to the execution time. It is important to bear in mind that the invocation lag is considered part of the execution itself as well.

To obtain more precise data on the function's execution, some metrics, logs and traces can be obtained with AWS CloudWatch. CloudWatch allows the users to view some metrics such as number of invocations, the execution duration, the throttle, number of concurrent execution, and others. Since this is only a backend application, traces will not show any relevant data and, therefore, are not used. However, while testing and troubleshooting both during the function's development and the live execution, the logs are very relevant and helpful.

This data obtained by AWS CloudWatch will provide us with the most important KPIs which are needed to reach some conclusions.

6.2 Performance results and analysis

This section presents the performance data obtained during the evaluation process. Visual representations, including graphs and charts, are utilized to better display the performance metrics obtained both before and after the integration of the serverless function. The analysis section interprets observed improvements and addresses any challenges or anomalies encountered during the evaluation.

6.2.1 Pre-Integration Metrics

The baseline performance metrics of the Java application before the introduction of serverless functions are showcased in this subsection. Visual representations of these metrics establish a performance benchmark that will be important when comparing to the metrics obtained after the Lambda function being integrated.

Given a different number of threads but the same base image and threshold used on the analysis, these were the results obtained after 5 executions, in milliseconds:

nThreads	1	2	3	4	5
10	2657	2535	2470	2504	2565
15	2618	2466	2665	2583	2564

Table 6.1: Total execution time before the serverless function integration

Looking at the values in the table 6.1, it is possible to identify that the results are very lookalike. Whether the image gets split and assigned to 10 or 15 threads, the results fall within the same values.

In the 10 threads' sample, the results show a medium of 2546.2ms per execution whilst looking at the 15 threads' sample there is a similar value of 2579.2ms.

Even though these values seem irrelevant at this point, they are going to be very relevant once the serverless function integration comes, since each thread will imply a new invocation to this serverless platform.

6.2.2 Post-Integration Metrics

Performance metrics subsequent to the integration of the serverless function are presented in this section. With the aid of some graphics and other measurements, this analysis section interprets the significance of these improvements and their impact on the overall performance of the application.

Following the same strategy, the same executions were run using the version of the application in which the Lambda function was invoked. Once again, two series of 5 executions were performed: one with 10 threads and another with 15 threads, obtaining the output values in milliseconds.

The following table displays the overall duration of the whole execution, with metrics obtained on the application side.

The results in the table 6.2 give us already lots of insights.

Looking at the 10 threaded execution, it is easy to notice a pattern with the 14000 to 16000ms per execution, apart from two exceptions. The 29233ms execution was the first

nThreads	1	2	3	4	5
10	29233	14598	81260	15459	15021
15	64802	60943	62973	61326	65566

Table 6.2: Total execution time after the serverless function integration

one of this sample, which proves one of the previously presented disadvantages of using Serverless, which is the cold start of the machines. The 81260ms execution is not easily explainable, so it will be address in a later phase.

Looking at the 15 threaded execution, there is clearly a pattern between the execution time with an average value of 63122ms and a standard deviation of 2444ms. The reason for such an high value needs to be further investigated with other data as support.

A bit shocked with these results and in order to better understand where most of the time was being spent to justify the performance decrease when comparing to the previous solution, the thread invocation was isolated from the rest of the duration, leading to the Table 6.3.

Invocations	Exec1	Exec2
1	8928	7206
2	8979	7924
3	9074	8157
4	9212	8166
5	10287	8240
6	10499	8256
7	10836	8427
8	10935	8889
9	11161	8917
10	11187	8959
11	11233	9145
12	11316	10314
13	12120	54591
14	50126	60496
15	60001	62576

Table 6.3: Post-integration Lambda invoking duration

The results on the table 6.3 prompt a series of execution values ranging from 7206ms to the 62576ms. There is clearly a standard of apparently normal values between the 7206ms and the 12120ms marks, with an average of 9534ms. However, outside that range, there are still a couple of values which go beyond the 50000ms mark.

Given that the local execution had an average duration of 2546ms, the reason for these execution values was still unclear.

To further analyze these results, the next go-to place was the AWS Lambda function on the AWS Console User Interface. Through AWS CloudWatch, it is possible to take advantage of some very useful metrics.

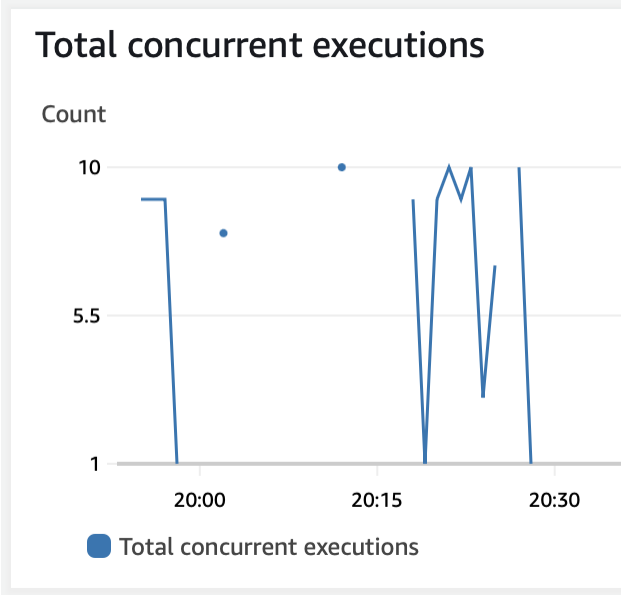

Figure 6.1: Concurrent Lambda function executions

Analysing the figure 6.1 it can seen that the number of concurrent exceptions ranges from 0 to 10, but never surpassing it. The reason for this is the fact that the AWS Lambda free version which is being used is capped 10 concurrent executions. This will make the AWS SDK keep on pooling the AWS Lambda function for free runners, leading to some thread executions of over 50000ms.

However, this still does not fully explain the reason for the executions to be taking 2546ms locally and spending 9534ms on a remote call to the Lambda function. This is where the next graph steps in.

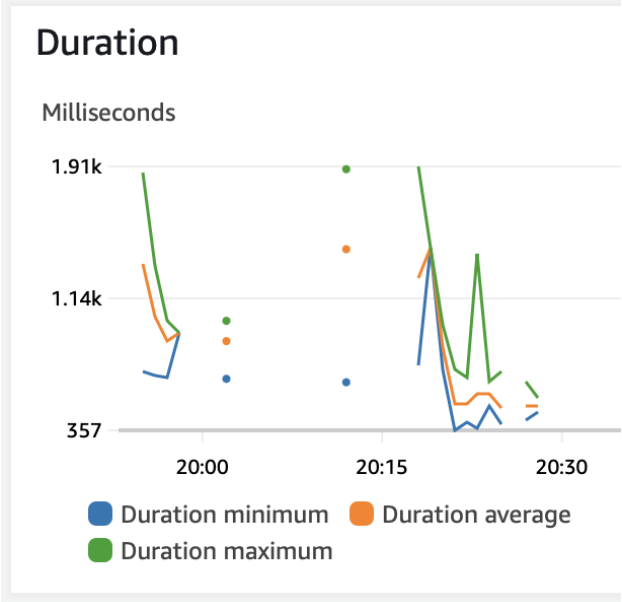

Figure 6.2: Lambda function execution duration

The figure 6.2 is displaying several executions with duration ranging from 357ms to 1910ms. This 1910ms is lower than the local execution time but also a lot lower than the average of the Lambda executions, which was at 9534ms.

The following code sample shows that the metrics were extracted right before and after the Lambda invoke request, which isolates this request from the rest of the computational logic of the application.

```
1 long startTime = System.currentTimeMillis();
2 InvokeResult invokeResult = lambdaClient.invoke(invokeRequest
    );
3 long endTime = System.currentTimeMillis();
4 long elapsedTime = endTime-startTime;
5 System.out.println("Thread elapsed time (Milliseconds): " +
    elapsedTime);
```

Given all this, and adding it to the fact that the function only takes up to 1910ms to execute, it means that there is an average of 7624ms that gets lost due to networking difficulties and other problems which are often out of control. This 500% increase is a huge setback that should be taken into consideration for not so complex scenarios.

6.2.3 Results discussion

The pre-integration metrics established a solid performance benchmark for the Java application, consistently reflecting the application's behavior across various thread counts.

After the integration of the serverless function, the performance landscape got more complex. Execution times demonstrated distinct patterns, with some occasional spikes attributed to several reasons.

The first execution always lasted longer than the encountered pattern, leading to the already expected fact that Lambda functions have a cold start time, in which the machines are still booting or recovering from sleep mode.

A closer examination on the AWS CloudWatch metrics and graphics helped discover two more setbacks that justify the out of range values that were encountered. One was the fact that AWS Lambda only accepts ten concurrent executions at the same time. Another was the fact that the function only lasted, at most, 1910ms to execute, leaving us with network latencies and external factors to blame for the 500% performance degradation.

Given all this, for this specific use case in which the local execution duration was already relatively low, a Lambda Function might not be a good approach unless a highly reliable and efficient environment is considered, just like an EC2 instance with several nines of availability (which is an AWS Virtual Machine) and share the same VPC (Virtual Private Cloud) to enable an higher performance.

Beyond performance, it is imperative to weigh the cost implications of adopting serverless computing. Serverless functions offer a cost-efficient model, billing users only for the actual usage rather than continuous server uptime. This cost-effectiveness becomes particularly pronounced when compared to the traditional paradigm of perpetually running instances, like an EC2 instance, assuming the worse case scenario to ensure availability.

This pay-as-you-go model translates into substantial cost savings, especially for applications characterized by variable workloads. Depending on the capability expected out of the Lambda function, its costs can be as low as $0.20 per 1 million requests and $0.0000166667 for every GB-second. If, on the other hand, the idea is to go for an always up-and-running EC2 (Elastic Compute Cloud) instance, the prices can quickly escalate. Taking the "t4g.nano" instance type as an example, the weakest machine type, the prices still go for $0.0042/hour (AWS n.d.(b)).

Therefore, looking at these two values, a conclusion can be reached that if the serverless function is always running, and under a lot of requests, then its prices will get more expensive than the EC2 instance. If that function will have a more sporadically usage, then the serverless function is definitely something to consider.

Chapter 7

Conclusion

In this concluding chapter, the key outcomes and future directions of the thesis are summarized. This chapter is divided into three sections: "Objectives Achieved", "Limitations" and "Future Work".

7.1 Objectives achieved

Throughout this thesis' journey, several key objectives have been successfully achieved. Namely:

- Serverless Adoption: An in-depth exploration of Serverless computing, including its advantages and the practical experience obtained with its adoption. The focus was on adjusting a Java application to start using Serverless functions, providing valuable data into the benefits and challenges of this approach.
- Performance Evaluation: A critical part of this thesis involved assessing the performance of the Java application both before and after integrating the AWS Lambda function. With some important metrics like the execution duration and the amount of executions in parallel, it was possible to assess where the performance degradation was coming from. These findings emphasize the potential of Serverless computing for scalable, event-driven workloads, if applied to the right scenario.
- Cost Optimization: An examination of cost optimization strategies within Serverless computing revealed the cost-effectiveness of this model, particularly when coupled with best practices and a right scenario. The scenario where a function is always running is maybe the only scenario in which the AWS Lambda function is not the best fit. Strategies for cost optimization, like the pay-as-you-go model, make sure the owners do not pay for resources that they do not need, or are actively consuming.
- Networking Challenges: One of the major contributions of this thesis was the networking challenged identified when implementing the use case. The identified delays during the Lambda function invocations help raise awareness for the complications that might occur when using Serverless function.

7.2 Limitations

Apart from the known disadvantages and limitations, throughout the implementation of the use case there were some setbacks that represent a learning opportunity, and something that should be exposed.

One of the limitations is the fact that the application deals with images that cannot be passed directly to the Lambda function as a parameter. This issue required some agility to overcome, obliging the conversion of image into a String representation in order to allow its movement. All of this serialization process could and should be avoided, as it seriously affects the performance of the execution. However, since the computational logic requires working with the image Java object, converting it to a base64 representation was the solution chosen.

Another limitation that was encountered and imposed a setback was the maximum size that AWS Lambda accepted for its Json payload parameter. Converting the base image to a base64 representation turned it into a oversized string, which ended up resulting in a 417 error, "RequestEntityTooLargeException". To overcome this limitation, instead of using a worse definition image, the image was split into a greater number of smaller parts, in order to reduce its size and, therefore, reduce the size of its base64 representation.

7.3 Future work

While this thesis has explored various aspects of Serverless computing, it is evident that the networking challenges associated with Serverless platforms remain a critical area of concern. The need to minimize the invocation lag and optimize data transfer between functions is a pressing issue which requires further research. Future work in this domain can significantly contribute to enhancing the efficiency and reliability of Serverless applications.

Future research can dive into ways of reducing latency during the invocation of Serverless functions. A way to reduce the latency could be to place both components (an EC2 instance and the AWS Lambda function) in the same VPC, therefore reducing the networking complexity. A more complex approach would involve exploring optimized network topologies, data caching strategies, or intelligent routing algorithms to minimize the delays associated with function execution.

In the rapidly evolving landscape of cloud computing, Serverless has emerged as a dominant paradigm. This thesis dove into key aspects of Serverless adoption, performance, cost optimization, and networking challenges within the context of deploying Java applications on AWS Lambda.

The findings revealed that Serverless adoption is mainly driven by the will to have its announced advantages, like its scalability and cost saving features. However, in order to take advantage of the full potential of Serverless functions, it requires a conscious decision making process, considering both performance implications and cost-efficiency for each scenario.

The performance evaluations highlighted the need to align applications with Serverless principles, particularly in addressing cold starts, parameter handling, coding languages and latency problems. The networking issues faced were a huge setback that reflects a real scenario that can happen as developers migrate to a Serverless function.

Cost analysis made it visible that sometimes it can be more cost-efficient to use an EC2 instance, but most of the time it is more cost-efficient to have a Serverless function for less used logic. Depending on the application that is being used, it might make sense to go for a long lived instance, but it all relates to the amount of time and resources that this function takes to execute. A good resource management can prove itself to be very importance when it comes optimizing costs.

In order better analyse all these metrics, a large scale use case should be carried out. This would result in more accurate and trustworthy outcomes, helping in tasks such as testing the scalability of the serverless function, against the lack of it in the local machine (or in an EC2 instance). This would not only help test the scalability capability of each platform, but also provide more data to analyse the networking difficulties.

In summary, this thesis showcased the Serverless landscape, focusing on the available platforms and a migration guide. Applying this migration guide to a practical use case and evaluating it shed some light on its benefits and challenges. Hopefully this thesis will guide developers towards considering and easily trying out Serverless computing options to consciously take the best decision and start taking advantage of its benefits.

Bibliography

AWS (2023). *Identifying serverless use cases*. Last accessed 18 September 2023. url: https://docs.aws.amazon.com/serverless/latest/devguide/serverless-usecases.html.

– (n.d.[a]). *AWS Lambda function handler in Java*. Last accessed 21 September 2023. url: https://docs.aws.amazon.com/lambda/latest/dg/java-handler.html.

– (n.d.[b]). *AWS Lambda Pricing*. Last accessed 21 September 2023. url: https://aws.amazon.com/lambda/pricing/?trk=bd068269-b7dc-4d56-81c4-3c5f58721bef&sc_channel=ps&s_kwcid=AL!4422!3!651612449180!p!!g!!lambda%20pricing&ef_id=CjwKCAjwg4SpBhAKEiwAdyLwvIFoMLlR6lm67SX71XpEedzXJZyq1LMrxv2kQe9_5Va6hJKxnqWEMBoCUEwQAvD_BwE:G:s&s_kwcid=AL!4422!3!651612449180!p!!g!!lambda%20pricing!19836375964!149982286551.

Azure (2022). *Azure Storage redundancy*. Last accessed 16 September 2023. url: https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy.

Batschinski, George (n.d.). *What is BaaS - Backend-as-a-Service?* Last accessed 11 February 2023. url: https://blog.back4app.com/backend-as-a-service-baas/.

Brooks, Rodney (2021). *A Quadrillion Mainframes on Your Lap*. Last accessed 8 December 2022. url: https://spectrum.ieee.org/ibm-mainframe.

Brush, Kate (2021). *virtualization*. Last accessed 8 December 2022. url: https://www.techtarget.com/searchitoperations/definition/virtualization.

Buchanan, Ian (n.d.). *Containers vs. virtual machines*. Last accessed 11 December 2022. url: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms.

Claes Wohlin, Per Runeson (2021). "Guiding the selection of research methodology in industry–academia collaboration in software engineering". In: Last accessed 16 September 2023. url: https://www.sciencedirect.com/science/article/pii/S0950584921001361.

Cloudflare (n.d.). *What is serverless computing?* Last accessed 9 February 2023. url: https://www.cloudflare.com/en-gb/learning/serverless/what-is-serverless/.

CNCF (n.d.). *CNCF Serverless Whitepaper v1.0*. Last accessed 24 January 2023. url: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview.

Docker (n.d.[a]). *Docker architecture*. Last accessed 11 December 2022. url: https://www.docker.com/resources/what-container/.

– (n.d.[b]). *Docker overview*. Last accessed 11 December 2022. url: https://docs.docker.com/get-started/overview/.

Google (n.d.). *Cloud Functions*. Last accessed 16 February 2023. url: https://cloud.google.com/functions.

Harris, Chandler (n.d.). *Microservices vs. monolithic architecture?* Last accessed 24 January 2023. url: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

Kumar, Priyaj (2022). *A Comprehensive Guide For On-premise vs Cloud Computing Tutorial*. Last accessed 8 December 2022. url: https://www.edureka.co/blog/on-premise-vs-cloud-computing/.

Maayan, Gilad David (2023). *Running Serverless in Production: 7 Best Practices for DevOps*. Last accessed 18 September 2023. url: https://devops.com/running-serverless-in-production-7-best-practices-for-devops/.

Oracle (2022). *Brief History of Virtualization*. Last accessed 8 December 2022. url: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html.

Paul Castro, Vatche Ishakian (2019). *The rise of serverless computing*. Last accessed 9 February 2023. url: https://www.researchgate.net/publication/337429660_The_rise_of_serverless_computing.

Redhat (n.d.). *What is container orchestration?* Last accessed 12 December 2022. url: https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

Roberts, Mike (2018). *Serverless Architectures*. Last accessed 16 September 2023. url: https://martinfowler.com/articles/serverless.html.

Schmidt, Tobias (2023). *Supported Languages at AWS Lambda*. Last accessed 21 September 2023. url: https://blog.awsfundamentals.com/supported-languages-at-aws-lambda.

Services, Amazon Web (2021). *AWS Lambda*. Last accessed 16 February 2023. url: https://aws.amazon.com/lambda/.

Strotmann, Joerg (2016). *Infographic: A Brief History of Containerization*. Last accessed 11 December 2022. url: https://www.plesk.com/blog/business-industry/infographic-brief-history-linux-containerization/.

Appendix A

Repository URL

You can access the repository with the code used for this project at the following URL:
`https://github.com/14ZOli/lambda_highlight_fire`