Master's Degree in Informatics Engineering
Thesis
Final Report

Elastic Microservices Platform

Designing a Platform for Implementing Microservices-based Elastic Systems for Deployment in Cloud Environments

Fábio de Carvalho Ribeiro
fdcr@student.dei.uc.pt

Supervisor:
Prof. Filipe João Boavida Mendonça Machado de Araújo

Co-Supervisors:
Prof. Rui Pedro Pinto de Carvalho e Paiva
Prof. António Jorge Silva Cardoso

September 3, 2018

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

The decision to use the cloud is appealing because it is usually associated with lowered costs and simplified deployment and management. A Platform as a Service (PaaS) provides such services by allowing users to develop, run and manage their applications without the need to build and maintain their own infrastructure.

Ensuring that the user's applications are able to automatically and elastically scale, requires some additional configuration. The existing platforms that provide such services are proprietary and rely on user-made rules to achieve their elastic and scaling capabilities. They do not perform an automatic analysis that provides a global vision over the applications to the user.

Our platform aims to provide automatic and elastic scaling of deployed applications. In the future, with tracing and a scheduling algorithm, we will achieve an automatic analysis that provides a global vision over the applications to the users. To experiment our approach, an open source platform for implementing microservices-based systems for deployment in cloud environments was designed and implemented. This platform achieves great scaling capabilities and allows users to deploy and manage their applications in a simple way.

Keywords

Microservices, Cloud, Scalability, Elasticity, Tracing.

Resumo

A decisão de utilizar a cloud é apelativa porque está habitualmente associada a custos reduzidos e a uma simplificação de instalação e manutenção. Uma Plataforma como Serviço (PaaS) fornece tais serviços permitindo os utilizadores desenvolverem, correrem e gerirem as suas aplicações sem a necessidade de construir e manter a sua própria infraestrutura.

Certificar que as aplicações dos utilizadores permitem escalar elasticamente e automaticamente, requer alguma configuração adicional. As plataformas existentes que fornecem tais serviços são proprietárias e baseiam-se em regras feitas pelos utilizadores para alcançarem as suas capacidades elásticas e escaláveis. Elas não realizam uma análise automatica que fornece uma visão global sobre os microserviços ao utilizador.

A nossa plataforma visa fornecer uma escalabilidade elástica e automatica às aplicações instaladas. No futuro, com tracing e um algoritmo de decisão, iremos alcançar uma análise automática que irá fornecer uma visão global sobre as aplicações para os utilizadores. Para testar a nossa abordagem, uma plataforma open source para implementação de sistemas baseados em microserviços para instalação em ambientes de cloud foi projetada e implementada. Esta plataforma alcança elevadas capacidades de escalabilidade e permite aos utilizador fazerem a instalação e gestão das suas aplicações de uma maneira simples.

Palavras-Chave

Microserviços, Cloud, Escalabilidade, Elasticidade, Tracing.

Acknowledgements

This thesis would not have been possible without the help and contributions of a special group of people.

I would like to first thank the supervisor of this thesis, Professor Filipe Araújo. He was of the most importance in the work developed. His knowledge, help and guidance throughout the entire journey were necessary for the success of this thesis. Eng. Jaime Correia, was also crucial for the success of this thesis. He was always willing to help, offering his vast knowledge and ideas to improve the work performed. I would like to thank him for all his help and contributions to this thesis. Prof. Rui Paiva and Prof. Jorge Cardoso were also important for the development of this work and deserve proper mention. Their help during the meetings performed to evaluate the state of the project was appreciated. I am also thankful to my colleagues Fábio Pina, Bruno Lopes and Artur Pedroso for their contributions during the meetings.

I would like to express my profound gratitude to my parents. I would not me able to successfully complete this thesis without their support and advices during my academic course. They gave me the chance to have a higher education and for that I am truly thankful for them. To all my friends and family, thank you for all the support that was provided to me during this journey.

Finally, I would like thank my girlfriend, Ingrid Oliveira, for always believing and encouraging me, even in the hardest times.

This work was carried out under the project PTDC/EEI-ESS/1189/2014 — Data Science for Non-Programmers, supported by COMPETE 2020, Portugal 2020-POCI, UE-FEDER and FCT.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Results 2
1.4 Work Plan 3
1.5 Collaborators 6
1.6 Document Scope 6

2 Background **9**
2.1 Concepts 9
2.1.1 Microservices 9
2.1.2 Scalability 11
2.1.3 Elasticity 12
2.2 Technologies 12
2.2.1 Docker 12
2.2.2 Kubernetes 14
2.2.3 Amazon EC2 15
2.2.4 AWS Elastic Beanstalk 16
2.2.5 Amazon Elastic Container Service 16

3 Architecture Description **19**
3.1 Requirements 19
3.1.1 Functional Requirements 20
3.1.2 Quality Attributes 21
3.2 Proposed Architecture 24
3.2.1 Context Diagram 24
3.2.2 Containers Diagram 25
3.2.3 Components Diagram 27
3.2.4 Chosen technologies 28

4 Implementation **31**
4.1 Microservices Application 31
4.1.1 Original Project 31
4.1.2 Architecture 32
4.1.3 Users Microservice 33
4.1.4 Songs Microservice 33
4.1.5 Playlists Microservice 33
4.1.6 Main App Gateway 34
4.1.7 Running everything on containers 34
4.2 EMP CLI 35
4.3 EMP Server 37

4.3.1 EMP Server Module 38
4.3.2 Cluster Manager Module 39
4.3.3 Kubernetes Controller Module 39
4.4 Scheduler 39
4.5 Container and Cluster Manager 40
4.5.1 Kubernetes in Bare Metal 41
4.5.2 Kubernetes in GKE 42
4.6 Microservices Application Instrumentation 45
4.7 EMP Detailed Overview 47
4.8 EMP Service Requirements Specification 48

5 Experiments **51**

6 Conclusion **53**

List of Figures

1.1 Gantt Chart for the First Semester 4
1.2 Kanban board for the second semester 4
1.3 Gantt chart for the second semester 5

2.1 Monoliths and Microservices[34] 10
2.2 Monoliths and Microservices Database Organization[34] 10
2.3 Docker Container Diagram[4] 12
2.4 Docker's Architecture[4] 13
2.5 Auto Scaling Group Illustration[27] 15
2.6 Elastic Beanstalk Workflow[28] 16
2.7 Amazon ECS Basic Components[25] 17

3.1 EMP Context Diagram (C1) 24
3.2 EMP Containers Diagram (C2) 25
3.3 EMP Components Diagram (C3) 27
3.4 Kubernetes Pod Startup Latency[39] 29
3.5 Kubernetes API Call Latencies - 5000 Node Cluster[39] 30

4.1 Microservices Application Architecture 32
4.2 EMP CLI Overview 35
4.3 EMP Control API Overview 37
4.4 EMP Server Files 38
4.5 EMP Kubernetes Overview 44
4.6 EMP custom decorator usage example 46
4.7 EMP Detailed Overview 47

List of Tables

3.1 Utility Tree 22

Acronyms

CLI Command Line Interface. 2, 16, 35, 38, 44, 48, 51–54

DEI Department of Informatics Engineering. 5, 40–42

EMP Elastic Microservices Platform. 1–3, 5, 6, 19–26, 28, 31, 35, 37–46, 48, 51–54

gcloud Google Cloud Shell. 43

GKE Google Kubernetes Engine. x, 5, 6, 42–44, 52

PaaS Platform as a Service. i

SLA's Service Level Agreements. 15

UUID Universally Unique Identifier. 39

Chapter 1

Introduction

This document presents the *Master Thesis* in *Informatics Engineering*, of the student *Fábio de Carvalho Ribeiro* during the school year of 2017/2018, taking place in the *Department of Informatics Engineering of the University of Coimbra*

1.1 Motivation

The existing platforms that allow users to deploy and scale their applications in the cloud, such as *Amazon's EC2* or *Beanstalk*, are proprietary and rely on user made rules to achieve their elastic and scaling capabilities. They do not perform an automatic analysis that provides a global vision over the applications to the users. Since there is no automatic analysis, users must specify rules for their applications to scale according to those. Each application is treated independently and this can affect their scaling capabilities. A simple example to illustrate this is as it follows: Imagine there is a service A and B running, in which A depends on B. If service B starts to have some problems it will impact service A's performance. Current market solutions are not able to automatically detect that service A is getting slower due to service B. In this case, allocating the proper resources to service B would increase the performance of both services but such conclusions are not possible without a global vision over the applications.

1.2 Objectives

To solve the issues mentioned in section 1.1, and to fill the market's gap, an open-source platform for implementing microservices-based systems for deployment in cloud environments is going to be designed and implemented. This platform will offer a global vision over the entire application, providing an automatic elastic scaling over the several microservices that compose that application through enough tracing and a decision algorithm. Tracing is a sophisticated use of logging that can monitor information regarding a program execution. The platform will take into account workload metrics, such as throughput, latency or availability and provide automatic elastic scaling without the need for specific user made rules. After an application is deployed, the user no longer needs to worry about its scaling needs because the platform will take care of that automatically.

A global vision over the applications that are running inside the Elastic Microservices Platform (EMP) will be provided to the users and management decisions will be done

taking that into account. Tracing capabilities are necessary to have that global vision over the applications. This traces will be used by a decision algorithm that will be responsible for making resource allocation decisions. This decision algorithm is important because it is responsible for the system's elasticity by scaling according to accurate traces that model the system's performance. This algorithm will have the applications tracing information as input, analyzing it and decide the need to scale up or down an application based on its information.

The main goal of this thesis is to design and implement an entire open source platform that is capable of achieving great scaling capabilities without user made rules. This work will be used by Eng. Jaime Correia for his doctoral program, so the entire system implemented must be as functional as possible. The only component that will not be implemented is the *Scheduler Algorithm* which will be Eng. Jaime Correia doing it. The platform implemented should achieve great scaling capabilities and allow users to deploy and manage their application in a simple way. After the user deploys an application, the platform will make sure it stays running and allow end users to consume them. The *Scheduler* component that Eng. Jaime Correia will develop will be responsible to automatically analyze the applications tracing information and perform a decision on whether it is necessary to scale up or down a specific application. This means that the EMP must be prepared to receive such commands although the *Scheduler* component will be implemented later. In the end, the EMP must be fully functional and allow a simple integration with the future development of the *Scheduler* component.

It is necessary to specify a set of requirements that users must satisfy in order to achieve elastic scalability automatically. It is also necessary to abstract the infrastructure and resources and have well defined interfaces to achieve a high level of modularity regarding the tracing component and the decision algorithm. This allows for the possibility to swap components if necessary.

After the implementation, it is necessary to test, optimize and validate the system to achieve a better performance and efficiency. To do so, a testing system has to be implemented to perform quality tests that can be later used for its validation.

1.3 Results

The work performed in this thesis satisfies the objectives that were proposed. In the end, an open-source platform for implementing microservices-based systems for deployment in cloud environments was designed and implemented.

The EMP has a Command Line Interface (CLI) for users to deploy and manage their applications. This CLI communicates directly with the EMP server component which is responsible for all the logic to operate a *Kubernetes* cluster and to store the platforms state. This *Kubernetes* cluster is where all the users applications will be running and is responsible to both manage them and to manage the infrastructure resources. The users applications that are instrumented, send their traces over *Kafka*, that is running inside *Kubernetes*. There is a *Zipkin* server also running inside the *Kubernetes* cluster that is responsible to collect those traces from *Kafka* and to present them to users in its UI.

This platform achieves great scalability and was designed and implemented to be highly modular. In case a user is using our open source platform and wants to change *Kubernetes* as a *Container and Cluster Manager* for something like *Mesos*, he can do it in a simple way without the need to change the entire system. This also allows Eng. Jaime Correia

to easily integrate his *Scheduler* component to the EMP once it is implemented.

Many simple tests that assure the correct platform behavior and ensure that it is ready for Eng. Jaime Correia to use for his work were performed. It is safe to say that the objectives proposed for this thesis were met and the work performed was a success. Although the *Scheduler* component that provides an automatic analysis over the applications and is responsible for an elastic scalability is not yet implemented, all the implementation necessary for its integration and correct operation is complete.

1.4 Work Plan

In this section, the work performed in the first and second semesters will be presented.

Since this is an investigation project, it was necessary to perform exploratory work and there is not a specific development methodology implemented. Instead, meetings every two weeks were done to discuss and analyze the work performed. The meetings were attended by myself, professor Filipe Araújo and Eng. Jaime Correia. I would also participate in another meeting in the same week with professors Filipe Araújo, Rui Paiva and Jorge Cardoso, my colleagues Fábio Pina, Bruno Lopes and Artur Pedroso and also doctoral student Eng. Jaime Correia. All these meetings were helpful because it was possible to share ideas and solutions together. New deadlines were always proposed in order to progress in the work developed. In the end, although there was no specific software development methodology implemented, these meetings were more than enough to guide this thesis and to assign the work that needed to be done every two weeks, allowing for a productive and high quality work done.

In Figure 1.1 the Gantt chart illustrating the work schedule performed in the first semester is presented. It started in the middle of September when the project was presented and some core topics were discussed. After the project contextualization, it was now time to start researching some core concepts that would be discussed and used in the work performed. While doing the background search, it was necessary to implement a microservices system that would be used for testing the platform once it was built in the second semester. This microservices system development was very time consuming because the need to adapt a monolithic system into a microservices one presented a lot of problems and several bugs from the original project were fixed. Detailed information about the microservices system development is presented in section 4.1.

Once the microservices system was finished, I could now focus on background research of concepts and technologies that were going to be used. At the same time, requirements gathering was being done.

After the functional requirements and quality attributes elicitation, the first architecture diagrams were designed. This architecture work took a lot of time to complete because it was also necessary to be constantly looking into which technologies could possibly satisfy the system requirements.

At the end of December, once the architecture was defined and detailed, it was time to start writing the intermediate report that had to be delivered in the end of January.

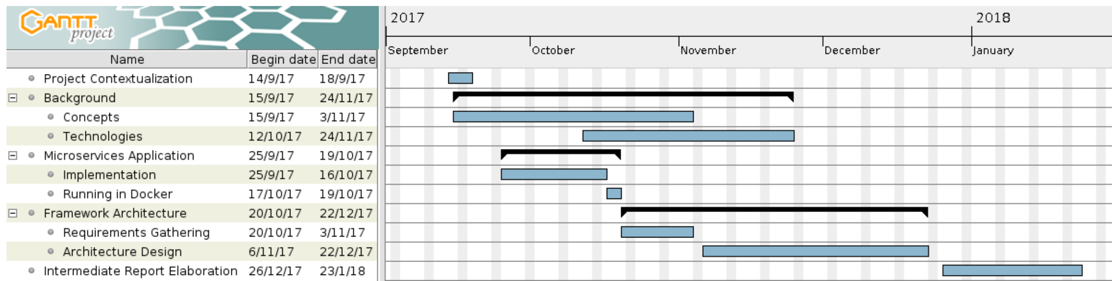

Figure 1.1: Gantt Chart for the First Semester

In the second semester, the meetings were performed every two weeks with Prof. Filipe Araújo, Prof. Rui Paiva, Prof. Jorge Cardoso, my colleagues Fábio Pina, Bruno Lopes and Artur Pedroso and also doctoral student Eng. Jaime Correia. For this semester, a detailed planning was necessary. A kanban board with user stories and simple tasks was created as it is possible to see in figure 1.2. If a given task was dependent on another or if it took more than one days to complete, a tag was assigned to it. This kaban board was useful because I could now have an overall picture of the entire project and what was left to do. By dragging a card to the "Doing" pile, I could focus on a task at a time. Throughout the semester, some task needed to be discarded and others needed to be implemented again in a different way. With a kanban board was easier to keep track of the tasks completed, the tasks that were discarded and those that were still not implemented.

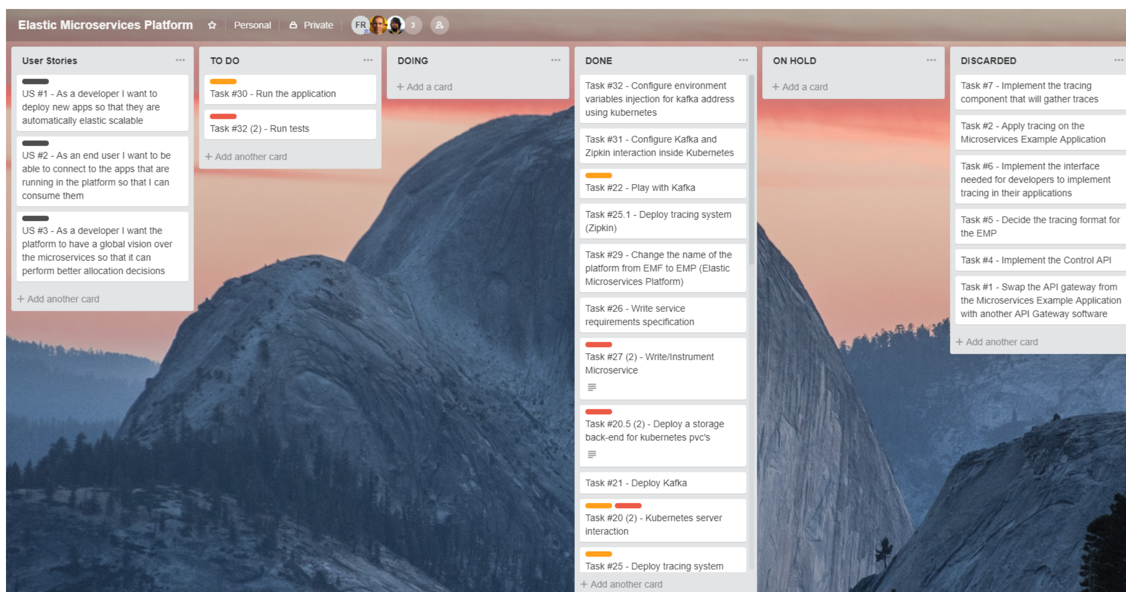

Figure 1.2: Kanban board for the second semester

In figure 1.3 a detailed gantt chart that was used to complement the kanban board is presented. This gantt chart and kanban board were created in the beginning of the second semester and were updated whenever it was necessary.

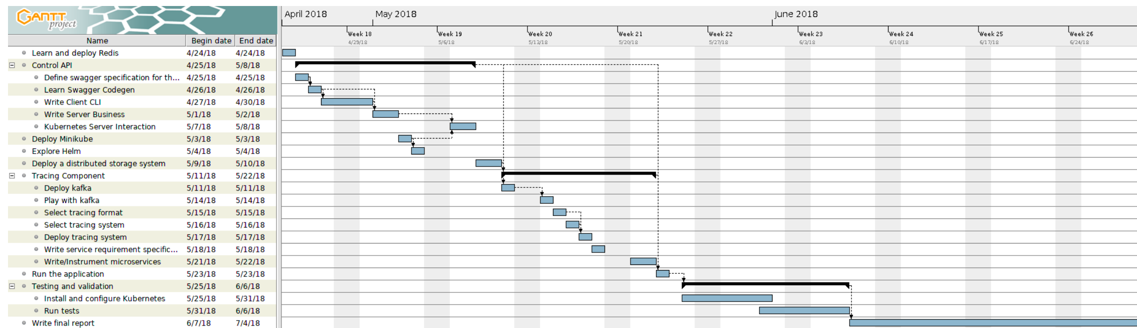

Figure 1.3: Gantt chart for the second semester

Implementing the *Control API* and a *Client CLI* was the first big step for this project. The next step was to deploy minikube which is a local installation for kubernetes to test the *Client CLI* and the *Control API*. After minikube was working properly, I started to explore *Helm* that is helpful for deploying kafka and a tracing system easier. This is when I saw minikube limitations and decided to use a real kubernetes cluster.

To meet the requirements for a real kubernetes cluster, a request for the needed resources was made to Department of Informatics Engineering (DEI). When the resources were available, a fresh installation of kubernetes in bare metal was made which was very time consuming. Kubernetes installation on bare metal is not very well documented and there is a need to configure so many things for it to work. In the end, this bare metal installation in the DEI cluster was aborted. After all the configurations, for the kubernetes cluster to allow its application to be reached from outside of its own network, specific configuration was necessary. This configuration required the DEI helpdesk to make a range of IP's available for my cluster to automatically assign using a custom load balancer which proved to be dificult to achieve because of the configurations necessary and the availability of DEI helpdesk. The solution found was to use Google Kubernetes Engine (GKE).

Installing and configuring kubernetes proved to be very time consuming and caused the delay of this thesis. With GKE, it was necessary to learn their *CLI* and configure a kubernetes cluster from scratch.

The deployment of kafka and the tracing system (zipkin) was also time consuming because they needed special requirements to work in a kubernetes environment. A custom zipkin container was implemented for it to work in my kubernetes cluster.

The microservices songs application that implemented, was now instrumented using a python library called py_zipkin. The way that it was instrumented follows the *OpenTracing* standard. For this library to work the way I wanted, I needed to implement a custom decorator that I could use to trace all my requests from the microservices songs application. This instrumentation took some time because the library that was used had a bug that I reported and was fixed by its development team and also because I had to develop a custom decorator for it to work the way I need it to. After this instrumentation, local tests were performed to test the flow of the traces. I configured and used kafka and zipkin on my local machine and did some tests to see if the traces would be sent from the microservices songs application to kafka and see if zipkin would be able to collect those traces from kafka and show them in its UI.

For ease of deployment, a script to deploy and configure kubernetes on GKE was made. This script is able to create, deploy and configure a kubernetes cluster for the EMP, which

will also deploy and configure kafka and zipkin inside kubernetes. After the entire cluster is working on GKE, a configuration on the *Control API* was required for it to communicate directly with the cluster. After all this, a simple test deploying the microservices songs application in the EMP was made and worked.

To be able to show the EMP working, a simple algorithm was implemented that is responsible to decide if there is a need to launch or stop an instance of a specific application.

To check if the EMP is working properly and performs well, some tests were made and its results collected and analyzed.

In the end, the final report was then written to better detail and explain the entire work performed during this master thesis.

1.5 Collaborators

The main persons involved in this project, which contributed in a valuable way, will be mentioned in this section.

Every two weeks a meeting was held to see the evolution of this project and to discuss future work. All the members that attended the meetings sharing their ideas and opinions that helped this project were: Professors Filipe Araújo, Rui Paiva and Jorge Cardoso, Doctoral student Eng. Jaime Correia and Master's students Bruno Lopes, Fábio Pina and Artur Pedroso.

Professor Filipe Araújo, the supervisor of this thesis, contributed with his knowledge, help and guidance throughout the entire project duration.

Eng. Jaime Correia, currently attending a doctoral program, always contributed with his knowledge and ideas that helped improve the work performed. He helped solving some problems that I encountered and also helped planning this project tasks. After this thesis is completed, he will be responsible to develop a scheduling algorithm. This algorithm will receive traces from the platform, analyze them automatically and issue control commands regarding the need to shut down or launch new application instances.

Fábio Pina, currently attending a Master's degree in Informatics Engineering, also contributed for this project. He was responsible to update the microservices system, that I originally developed, from Python 2.7 to Python 3.6. He also added new features and improved the overall quality and structure of the entire application. This microservices application is used for testing purposes to validate the platform.

1.6 Document Scope

The present document is organized as follows:

Chapter 2 contains most of the researched topics. It starts by explaining some core concepts needed to understand the work performed in this thesis and it also presents several technologies that were researched that could possibly be used for the EMP.

In chapter 3 the architecture of the EMP is presented. It starts with the functional requirements and quality attributes, then the different architecture diagrams and its explanation are covered and finally the reasons behind some technologies choices are presented.

Chapter 4 contains all the implementation details, showing all the difficulties and challenges encountered and how they were handled. It starts by explaining the Microservices Application, section 4.1, that is used for testing purposes and all the changes it suffered. The Control API that is responsible for the interaction between the user and the container and cluster manager, is presented in section **??**. The container and cluster manager is then presented in section 4.5, where the approach to its configuration and deployment steps are analyzed and discussed in detail. Finally, in section **??**, the tracing component responsible for collecting traces from the applications running inside the container and cluster manager is presented.

In chapter 6, the conclusions for this thesis are presented. Some tests were performed and will be presented in this section in detail.

Chapter 2

Background

In this chapter, the research that was done covering the main topics considered for this work is going to be presented.

The first section, 2.1, contains all the major concepts required to better understand the work performed. In the second section, 2.2, the most important technologies that could be used in this work are presented, giving an overview regarding their architecture and how they work.

2.1 Concepts

There are some core concepts that are necessary to explain in some detail, in order to understand the work performed in this thesis. These concepts are going to be presented in this section.

2.1.1 Microservices

The microservices architecture pattern is becoming more popular. The reason behind it, is that microservices offer a lot of advantages over monolithic applications, and therefore, people are starting to adopt more the microservices approach when building a new system.

Unlike monolithic style, microservice architectural style "is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API"[34].

Monolithic applications are usually simple to develop, test and deploy. In the beginnings of a project, they are also simple to scale by running multiple instances behind a load balancer[38]. However, as it starts to grow, the scaling of a monolithic application is going to bring a lot of problems. The application will become complex and hard to understand since its code base will become huge. Developers will not be able to understand the entire code, and therefore implementing new features or fixing bugs will be tremendously time consuming. The time needed to deploy the application will increase and any change made will cause the application to be rebuilt and re-deployed making the process even more time consuming. Adopting new frameworks or languages is also very time consuming and hard especially if the monolithic application is large. It would be necessary to change the entire application, making it harder to start using a newer and better technology.[38]

Microservices can solve many of the problems mentioned above and will now be analyzed in detail.

Each service, in a microservices architecture, will have a specific functionality or features, making it easier to scale by replicating only the needed services, unlike a monolithic application, as it is possible to see in Figure 2.1.

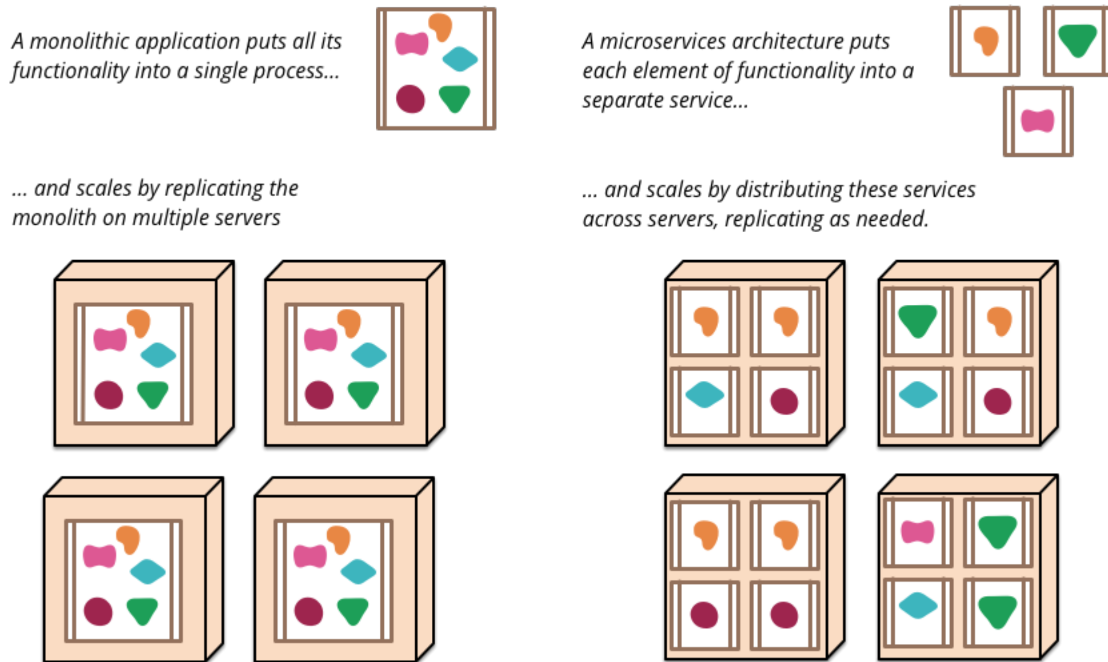

Figure 2.1: Monoliths and Microservices[34]

Figure 2.2, shows a visual representation of a decentralized data management of microservices against a single database monolith application.

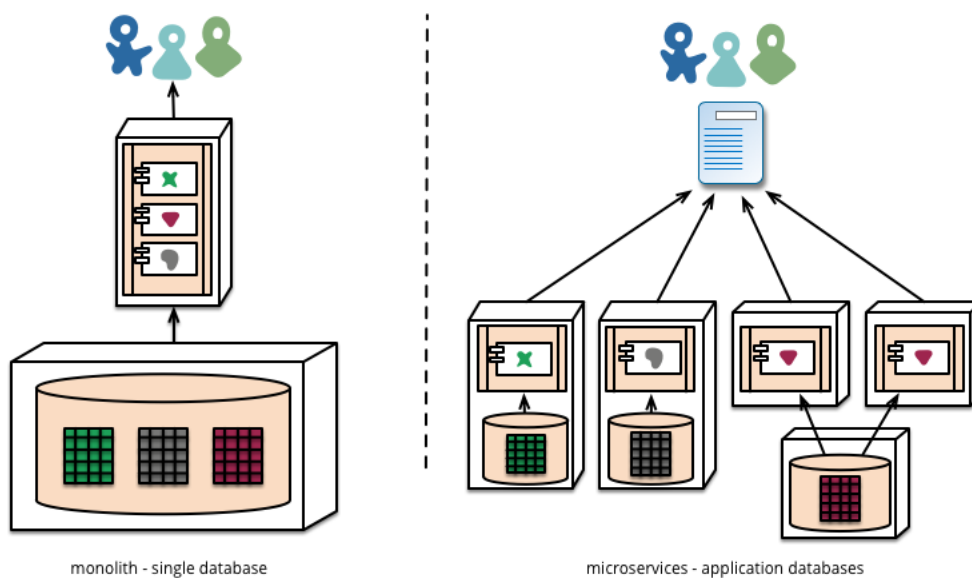

Figure 2.2: Monoliths and Microservices Database Organization[34]

Services are exposed using API's, "the two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging"[34]. They need to be as decoupled as possible, and because of that, a database schema for each service type is necessary. This allows for each service to use a database technology that better suits their needs, achieving a better performance.

Since the microservices architecture pattern relies on a set of services instead of a single monolithic application, the complexity of each component is less, making them easier to understand, deploy and develop individually. Developers do not need to understand the entire application, instead they just need to focus on the service they are working on, being able to implement new features and fix bugs easier and faster than they would in a monolithic application. Different services can be implemented using different languages or frameworks. Since they have an API specified for communication, the technologies used do not matter. This provides more options when building a new service regarding the technologies or frameworks to use, allowing developers to choose the ones they feel is the best for that situation. It is to note that in a microservices architecture, changes that could cause errors after deployment are easily managed than they are on monolithic applications, since it is possible to isolate the cause (specific service) and then rollback the changes done and fix the problem.[38]

Microservices architecture certainly is good but it also has drawbacks. Since microservices usually use partitioned databases, tasks that need to update several databases are hard to do. In a monolithic application that would not be a problem since it would be done in a single transaction, but in microservices, an eventual consistency approach has to be implemented[34]. In a microservices application, since it is a distributed system, things like slow requests or an unavailable service must be dealt with, increasing complexity. In case there are services that present dependencies to others, testings and changes that are necessary to perform might be harder, since it will involve all of them. Deploying will also be a lot more complex, comparing to a monolithic application, because in a microservices architecture there will be more components that needs to be deployed, scaled and monitored. In order to scale, microservices deployment should be as automatic as possible.[38]

"The golden rule: can you make a change to a service and deploy it by itself without changing anything else?"[36, p. 3] If the answer is yes then the microservices architecture is on the right path.

2.1.2 Scalability

Scalability is the "capability of a system, network or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth"[33]. There are two different types of scalability, Scale Vertically/Scale Up and Scale Horizontally/Scale Down.

The first one, is when there is an upgrade into a more powerful machine. For example, adding more resources like CPU power to a single node in the system. The second one is when a new node is added to the system. In a cloud environment, this could mean to just add a new instance of an application running in a virtual machine and then redistribute the load across all the nodes.[21][33]

2.1.3 Elasticity

"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"[37]. In a cloud infrastructure, it involves creating containers or virtual machines to match the current demand in real-time. While scalability is the systems capability of handling increasing amounts of work, using more resources, elasticity takes into account the time factor, by matching resources according to the demands in a specific time [33][37].

Elasticity in a systems brings a lot of advantages regarding the resources used. It is popular in the could since users will only pay for what they use on an elastic system. If suddenly there is a spike in workload, an elastic and scalable system should be able to provide resources to match the current demands. Once they are not needed anymore, because the workload decreased, the system should be able to detect that and stop using unnecessary resources providing a more efficient use of resources overall[33].

2.2 Technologies

The main technologies that were researched for this work are going to be presented in this section. They will be analyzed in detail to provide a good overview of their capabilities.

2.2.1 Docker

"*Docker* is an open platform for developing, shipping, and running applications"[4]. It is based on Linux containers and is open-source. Although containers are not a new thing, *Docker* became popular due to several advantages:

Docker is easy to use. By packaging an application in a container, it offers the ability to developers to build and run their application faster. It is possible to do so in the developers own laptop for example, in a private or public cloud or even on bare metal. A container is a loosely isolated environment and it is possible to run many of them simultaneously in a single host, as it can be seen in Figure 2.3.[32]

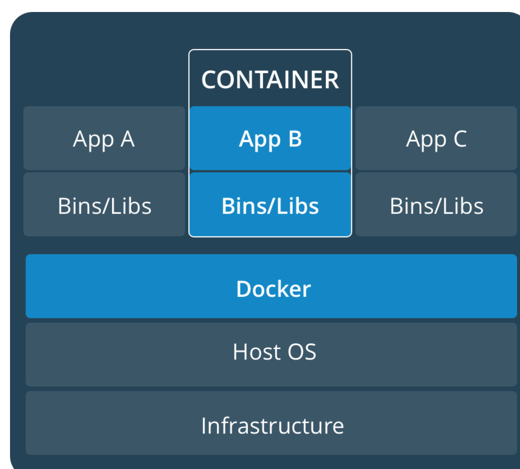

Figure 2.3: Docker Container Diagram[4]

Docker containers are lightweight and fast. While *Docker* containers can be built and run in seconds, *Virtual Machines* take a lot more time since they need to boot the entire operating system.[32]

Docker Hub, is a place to store public images created by the community and is also possible to retrieve them. It is really simple to just pull an image available and use it with slightly or no modifications at all.[32]

Modularity and scalability potential is also crucial. With *Docker*, it is simple to break down an application into individual containers and connect those containers together. This makes applications easier to scale and achieve a high level of modularity since it is possible to just launch more containers of a specific application component according to the current needs.[32]

In Figure 2.4, *Docker's* architecture is presented.

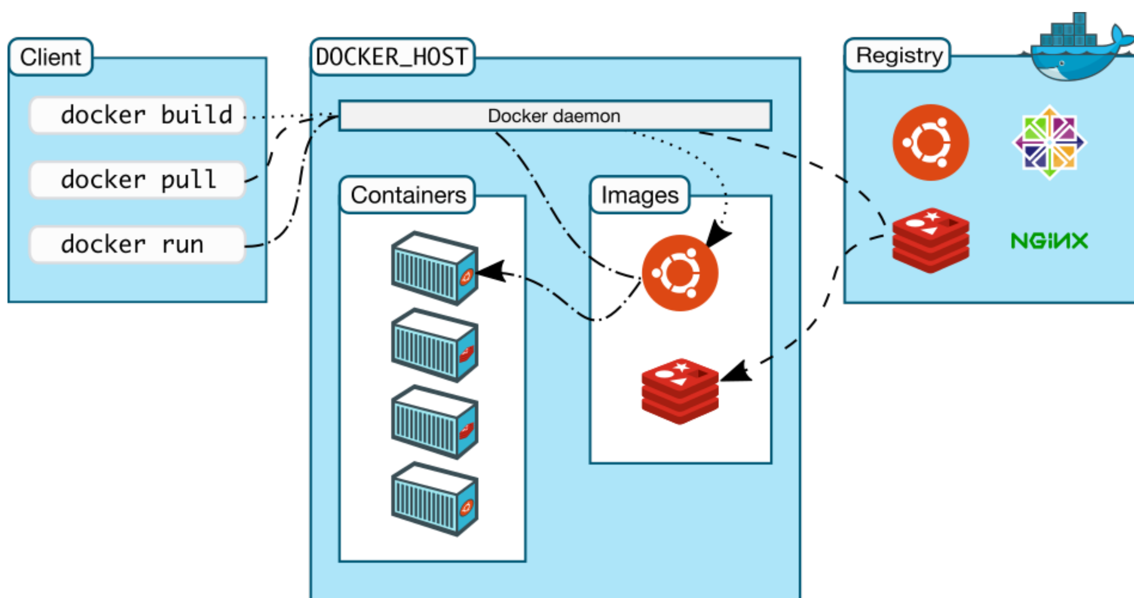

Figure 2.4: Docker's Architecture[4]

Docker uses a client-server architecture. *Docker daemon* component is responsible for building, running and managing *Docker* containers. The *Client* communicates with *Docker daemon* using REST API. It is to note that *Docker Client* and *Docker Daemon* can run on different host machines. *Docker Registry* is where docker stores its images. By default, *Docker* is configured to look for images in *Docker Hub* but it can be configured to use a private registry.[32]

Regarding *Docker objects*, an *Image* "is a read-only template with instructions for creating a Docker container"[4]. It is possible to create images or use those already published in a registry by others. A *Docker file* is needed in order to create and run an image. Each instruction in a *Docker file* creates a layer in the image and when a change is made, only the layers that are affected are rebuilt. This is one of the reasons *Docker* is so fast and lightweight when compared to virtual machines for example. "A container is a runnable instance of an image"[4]. The *container* has everything the applications needs to run. It has the operating system, application code, system tools, system libraries and more. It is possible to create a new image based on a containers current state. "*Services* allow to scale containers across multiple Docker daemons, which all work together as a swarm with

multiple managers and workers"[4]. It is possible to define a desired state in which that will be maintained as much as possible.[4][32]

2.2.2 Kubernetes

"*Kubernetes* is an open-source platform designed to automate deploying, scaling, and operating application containers"[29]. *Kubernetes* is useful for microservices applications since it groups containers that compose an application into logical units for easier management. Using containers brings a lot of advantages when developing an application but once the size of the entire application increases, a framework for managing all these containers is necessary. *Kubernetes* is able to schedule and run application containers either on physical or virtual machines. [29][35]

Regarding its architecture, *Kubernetes* needs at least 1 *Master Node* and can have multiple *Nodes*. *Master Nodes* make all the global decisions about the cluster (scheduling for example), they also detect and respond to cluster events and are responsible for exposing the API. It is to note that these *Master Nodes* can run on any *Node* of the cluster.[35]

Each *Node* runs at least one container, like *Docker* for example with a node agent that communicates with the *Master Node*. Each *Node* will also have and run components responsible for logging, monitoring and service discovery. It is also possible to add optional add-ons. These *Nodes* can be either virtual machines or bare metal servers.[35]

A *Pod*, is *Kubernetes* core management unit. It can have one or more containers running inside it. Usually it has only one container but if they are tightly coupled for some reason, they can both run inside the same *Pod*, sharing resources like mounted volumes for example. If it is necessary to scale an application component, it can be done by adding or removing *Pod's* according to the current needs. It is to note that when a *Pod* fails, they are never brought back, instead *Kubernetes* will take care of the problem by creating a new *Pod*.[35]

"Replica sets deliver the required scale and availability by maintaining a pre-defined set of pods at all times"[35]. Services are used to expose *Pods* to internal or external consumers.

Kubernetes has a huge scaling capability. Not only can it scale *Pods* whenever is necessary, these scalling capabilities can take full advantage if stateless *Pods* are used.[35]

Kubernetes is able to provide availability regarding the infrastructure and at the application level. Each *Kubernetes* cluster component can be configured to achieve high availability. It supports several distributed file system to ensure that data is persisted even when something unexpected happens. It is also possible to configure a minimum number of *Pods* in which *Kubernetes* will try to maintain those *Pods* running. In case any of them crashes for some reason, *Kubernetes* has a built-in health checks that are able to detect that and then a new *Pod* is launched to reach the configured state.[29][35]

When comparing *Kubernetes* to *Docker Swarm*, *Kubernetes* has more advantages than drawbacks. *Kubernetes* has auto scaling based on CPU utilization, which on *Docker Swarm* would need to be done manually. *Docker Swarm* has limited functionality and fault tolerance when compared to *Kubernetes*. While *Docker Swarm* needs to use third party logging and monitoring tools, *Kubernetes* has those built in. Overall *Kubernetes* is a more complete platform offering more features and the ability to tweak and customize a lot more things than *Docker Swarm* does. Also, *Kubernetes* is a more mature solution and more popular than *Docker Swarm*. However, *Kubernetes* has also drawbacks. It is

much harder to install and configure than *Docker Swarm*, has a steeper learning curve, and it is incompatible with *Docker CLI* and *Docker Compose*[16] tools.

2.2.3 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a webservice that provides virtual machines in the cloud. Such machines offer storage and compute power for users to use them as they wish. Amazon provides different EC2 instance types that differ in CPU, memory and storage to better suit each users needs. They charge users only for the resources that were used. This is very appealing for users because they do not have to build and maintain their own infrastructure when they can use Amazon's services to develop, run and maintain their applications. To access and manage Amazon EC2, they provide a Management Console or the user can use their AWS Command Line Interface (CLI).[26][1]

Amazon EC2 also provides users with the ability to choose from different Operating Systems, different storage options that better satisfy each users needs, control over the security of each instance and much more. It is to note that they also provide the ability to migrate an existing application into EC2 and their Service Level Agreements (SLA's) guarantee at least 99.95% uptime.[1]

Amazon EC2 also allows users to declare conditions for their applications to automatically scale with AWS Auto Scaling. These collections of EC2 instances are called Auto Scaling groups. User-made rules are necessary to declare the minimum and maximum number of instances each Auto Scaling group has, so they would not go below or above that specified number. It is also possible to declare a desired capacity and the AWS Auto Scaling will ensure that number of instances are running.

"For example, the following Auto Scaling group has a minimum size of 1 instance, a desired capacity of 2 instances, and a maximum size of 4 instances. The scaling policies that you define adjust the number of instances, within your minimum and maximum number of instances, based on the criteria that you specify."[27]

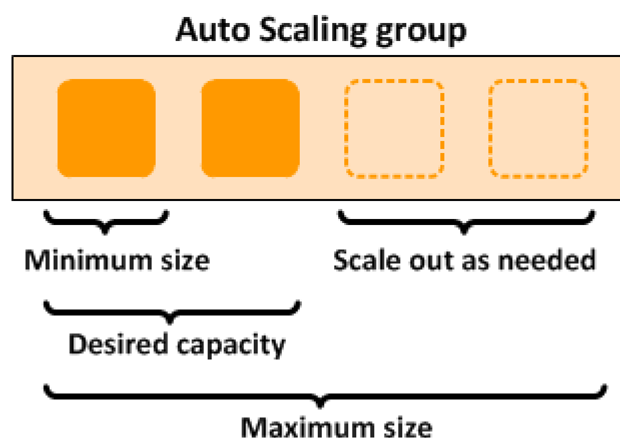

Figure 2.5: Auto Scaling Group Illustration[27]

2.2.4 AWS Elastic Beanstalk

AWS Elastic Beanstalk allows users to develop and scale their applications in a simple way. Users will not have to manage their applications resources because Elastic Beanstalk does that automatically. Users will only need to perform some prerequisites for their applications to work with Elastic Beanstalk. It supports several languages and configurations for each language. Elastic Beanstalk will use AWS resources like Amazon EC2 instances to run the users applications.[28]

Elastic Beanstalk is able to automatically perform health monitoring on applications, scale, load balancing and provide storage. Although it can do all of this automatically, users have control over their applications resources and can access and change them if they so desire.[28]

The first step to use AWS Beanstalk is to create and application and then upload it. Elastic Beanstalk will then launch and environment, creating and configuring the AWS needed resources. Users are able to manage their environment once it is launched successfully. In figure 2.6, the Elastic Beanstalk workflow is presented.[28]

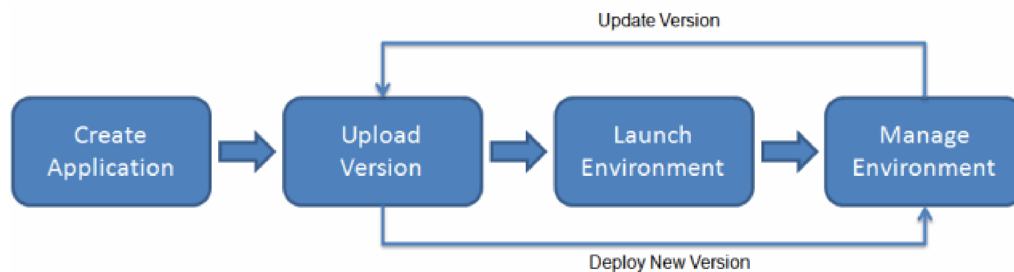

Figure 2.6: Elastic Beanstalk Workflow[28]

It is possible to access all kinds of information about the application that was deployed through their API, AWS Command Line Interface (CLI) or Management Console. AWS Elastic Beanstalk is a simple service that allows users to deploy their applications without worrying about managing and configuring the necessary infrastructure resources.[28]

It is to note that with AWS Elastic Beanstalk, the user can only set hard memory limits in container definitions. This means that the user either sets more memory than what they need or try to fit all the containers in one instance. AWS Elastic Beanstalk provides a more primitive scheduler but in return users get the ease of use. Users are not "able to independently schedule a replicated set of queue workers on the cluster"[5] because all cluster instances must run the same set of containers.[5]

2.2.5 Amazon Elastic Container Service

"Amazon Elastic Container Service (Amazon ECS) is a highly scalable, high-performance container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS"[2]. It is a cluster of EC2 machines that allows users to run their containerized applications inside those virtual machine instances.[25]

We can compare it to Kubernetes or Docker Swarm for example. Amazon ECS allows users to deploy their containerized applications without the need to install, configure and

manage a container orchestration software. They handle the scaling of the cluster and the scheduling of the containers inside the virtual machines. To manage the applications and to see detailed information about them, simple API calls or the AWS Management Console is used.[2][25]

In figure 2.7 Amazon ECS basic components are presented.

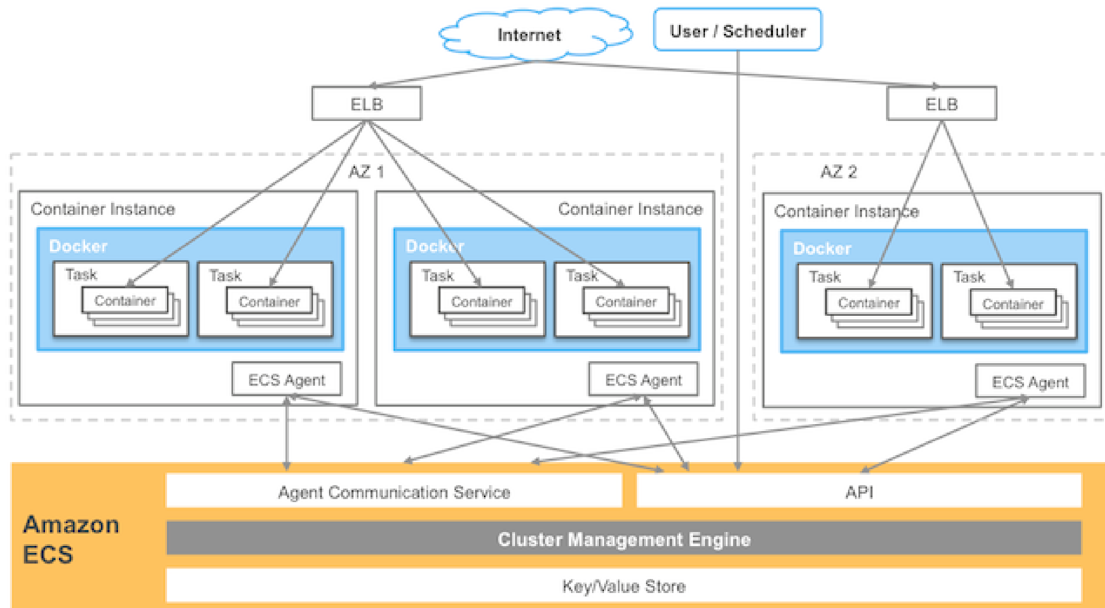

Figure 2.7: Amazon ECS Basic Components[25]

The cluster manager is responsible for managing the platforms state and to coordinate the cluster operations. This cluster manager is the core component of Amazon ECS. There are several schedulers that are decoupled from the cluster manager to allow users to build their own if they so desire. With Amazon ECS Container Agent that is running inside every machine, Amazon ECS is able to manage the EC2 instances that are running inside the cluster.[25]

To manage and coordinate the entire cluster, it is necessary to store data that keeps the platforms state updated. This data is very useful to know what are the available resources or occupied, to see how many instances are running and what containers they have and much more. Storing the platforms state is necessary to be able to manage and coordinate the cluster. In case of Amazon ECS, they use a key/value store. This key/value store is robust, reliable and scalable because it is distributed, achieving a higher availability and durability. Since the key/value store is distributed, it is now necessary to handle concurrency and ensure the data is consistent. This increases complexity of development and Amazon ECS achieves concurrency control by "using one of Amazon's core distributed systems primitives: a Paxos-based transactional journal based data store that keeps a record of every change made to a data entry"[25]. Amazon ECS is able to achieve optimistic concurrency when storing the cluster state information. "This architecture affords Amazon ECS high availability, low latency, and high throughput because the data store is never pessimistically locked"[25]. The cluster manager allows users to access the key/value store, that contains cluster state information, through the API. Users are able to use a set of commands to retrieve the desired information in a structured manner.[25]

Chapter 3

Architecture Description

This chapter contains all the information regarding the architecture of the Elastic Microservices Platform (EMP) that is going to be developed and is the main focus of this thesis.

In section 3.1 the requirements for the platform are going to be presented such as functional requirements in section 3.1.1 and also the quality attributes in section 3.1.2. The proposed architecture and all its design details are going to be discussed and analyzed in section 3.2, including the technologies that are going to be used to develop the EMP and the reasons behind it.

Before defining the functional requirements and the quality attributes, it was necessary to have a solid idea of what it was needed to implement to achieve the objectives proposed. Several concepts and technologies were researched to have a better understanding and ideas on how to develop our own platform.

To develop the EMP, several components needed to be implemented and connected together. A system to manage both deployed applications and the infrastructure resources was necessary. Such system must be able to achieve high scalability and provide features to manage the users applications. It is also necessary that this system has a way to provide access to the applications that are running inside it. To control this system, a core component must be implemented that handles all the logic regarding the management operations of the users applications. The control system is also responsible to receive requests from the developers and from a scheduler algorithm component to manage the deployed application. For the developers to be able to use the EMP and execute the operations necessary to satisfy their needs, some kind of User Interface must be implemented. This UI would communicate directly with the control system to manage the applications. The scheduler algorithm would be responsible to analyze the deployed applications and perform a decision based on that analysis. Such decision would be sent to the control system that would then execute the necessary operations into the applications management system.

3.1 Requirements

The requirements are very important when designing an architecture. They can have a great impact in the final architecture and because of that they are going to be presented before the proposed architecture for the EMP.

The functional requirements are listed in subsection 3.1.1 and the quality attributes are presented in subsection 3.1.2.

3.1.1 Functional Requirements

After the contextualization of the problem and the definition of the objectives that are necessary to achieve, the functional requirements became more clear to identify and are presented below:

1) Account Management

This task's objective is to provide users the ability to create and login into their accounts in order to use the EMP.

- REQ-1: Allow users to create and account in the EMP. (High priority)
- REQ-2: Allow users to login and use their EMP accounts. (High priority)

Each account will be linked to their deployed applications so it's easier to check which application belong to which user and perform the corresponding operations.

1) Deploy Application

This task's objective is to allow the users of the EMP (Developers) to deploy their applications in a way that makes them elastic and scalable.

- REQ-3: Allow users to deploy new applications in the EMP. (High priority)
- REQ-4: Guide and assist users during their application deployment to comply with EMP requirements. (High priority)
- REQ-5: Allow users to declare the quality metrics their application must meet. (High priority)
- REQ-6: Allow users to declare the resources each container of their application must have (CPU and memory). (Low priority)

It is important to note that when deploying applications, the platform will guide the developers to comply with the requirements needed to achieve elasticity and scalability.

2) Consume Application

This task's objective is to make the applications deployed inside the EMP available to the outside world for the end users to consume.

- REQ-7: Allow users to connect to the applications that are running in the platform. (High priority)

It is necessary to provide a gateway API for the end users to connect to the applications desired.

3) Manage Application

This task's objective is to allow users (Developers) to manage their applications once they are deployed inside the EMP.

- REQ-8: Support runtime changes such as update quality metrics to allow developers for a better control over their product and in this way, provide a better quality service. (High priority)
- REQ-9: Allow users to see detailed information about their applications. (High priority)
- REQ-10: Option to stop an application that is running. (High priority)
- REQ-11: Option to start an application that is stopped in the platform. (High priority)
- REQ-12: Option to completely remove an application that was deployed in the EMP system. (High priority)
- REQ-13: Allow users to list all their deployed applications and see their general details. (High priority)
- REQ-14: Allow users to see their applications tracing information. (Medium priority)

It is to note that it the user decides to shut down their entire application or specific instances that the databases of those corresponding instances should remain intact.

4) Scheduler

This task's objective is to provide elasticity to the system with the help of traces and a decision algorithm responsible for launching or shutting down instances of applications depending on the traces provided.

- REQ-15: Automatic analysis of tracing information regarding the workload of the applications that are running in the EMP. (Low priority)
- REQ-16: Automatic decision regarding the need to launch or to shut down instances of an application by analyzing its traces. (Low priority)

Since the scheduler is not the main focus of this thesis, because it will be Eng. Jaime Correia developing it in the future, their tasks priority is low.

3.1.2 Quality Attributes

When designing an application, quality attributes must be taken into account seriously because they will often impact the architecture, some more than others. That is why the quality attributes that were defined for the EMP are represented in a Utility tree in the table 3.1, ordered by their priority.

Table 3.1: Utility Tree

Quality Attributes	**Attribute Refinement**	**ASR**
Scalability	Able to support a large quantity of applications	The platform needs to be able to grow into a very large size (At least 4000 nodes) and the scaling between the system capacity (number of applications running) and the number of nodes needs to be linear. (H, H)
Elasticity	Platform able to perform well with increasing load	When the platform load increases, it needs to respond accordingly by launching a new instance of the application and distribute the load in a way that the impact in performance is very low. It is necessary to increase and decrease the resources used in order to achieve an elastic system and also to be an efficient one. (H, H)
Maintainability	Modular system	The platform needs to be designed and implemented in a way that some components can be replaced making it modular. (H, H)
Performance	Fast to deploy new instances	The platform needs to be fast at launching new instances of applications, making them available in less than 15 seconds. (M, H)
	Low API call latency	The platforms API calls need to be executed in less than 3 seconds to achieve a good performance (M, H)
Availability	System up and running	The platform needs to be available as much as possible providing a functional system to deploy and to consume the applications above 99.99 % of the time. (H, H)
Reliability	System working properly	The applications that are running in the platform need to behave as expected. In case any of them crashes for some reason, the system should be able to recover by launching a new instance of the application making it operational once again (M, H)

The quality attributes presented in the table 3.1 are the ones that were considered the most important for the success of the EMP system and are now going to be analyzed in greater detail. It is to note that the quality attributes present a notation system in which (L = Low, M = Medium, H = High). The first letter in this notation that is used represents the impact that quality attribute has in the architecture. The second letter represents the importance and value that this quality attribute represents for the business.

The most important quality attribute for the EMP system is scalability. It is really important that the system is able to grow into a very large size (above 4000 nodes) and the scaling between the number of applications running and the number of nodes needs to be linear. To achieve such scaling capabilities, the Container and Cluster Manager needs to be implemented with a technology that supports such demands and presents a good

performance. This allows users to be sure that if their applications starts to grow, the EMP system will be able to provide enough resources and stability for it to continue to run normally and to withstand a large amount of workload. This quality attribute is also connected to the ability of the system to be able to distribute the workload accordingly by load balancing it, providing a good performance of the users applications despite their scaling.

The second most important quality attribute is elasticity. This attribute is crucial for the EMP system because it allows users to only pay for what they spend. The EMP system will take care of analyzing the workload of the users applications and deciding if it is necessary to launch a new instance if the load is growing or to shutdown instances in case the load lowers and it is no longer necessary to have those instances up and running. This essentially means that the resources used by the users applications will only be as much as necessary according to the demand in a current time frame. Users will not have to worry about allocating resources in real time because the system does that for them. They do not have to buy their own infrastructures because it would be more expensive than using the EMP system. If a user had its own infrastructure, in times where the workload was low, they would not be using their infrastructures to the full potential so they would be wasting resources. In the EMP system that does not happen because it automatically detects the need to launch or shut down instances according to the workload of the users applications in real time.

Maintainability comes in third position of the most important quality attributes. It is necessary that the system is modular allowing to replace a component with another. For example, if it was necessary to replace the scheduling algorithm component with a different one, that needs to be possible and without having impact in the other architecture components. By planning and designing a good architecture it is possible to achieve a modular system regarding some components, like that scheduling algorithm component for example.

Performance comes next in the most important quality attributes because the system needs to have a nice performance in terms of the time that takes to launch new instances when needed, otherwise the users would not want to use this platform. It is really important that the system takes at most 15 seconds to launch a new instance of a users application that is running in the EMP system because it needs to respond accordingly to the increase or decrease in workload of the application. It is necessary to find a solution that allows such performance despite the fact that it also needs to allow for great scalability and elasticity potential. It is also very important that the EMP API calls have a low latency, of at maximum 3 seconds, to be responsive and considered to have a good performance.

Availability comes in fifth place in the most important quality attributes ranking. The EMP system needs to be available for at least 99.99% of the time making it a highly available system. To achieve such availability, we can replicate the EMP server and the Container and Cluster Manager as much as necessary. Users will want to deploy their applications in a system which they can get some guaranties about its availability. It has to be available as much as possible so that the users applications deployed in the EMP are also as available as possible. It is to note that the users can also specify the availability of their applications when they are deploying them or change that value in runtime and the EMP system must be able to meet those requirements.

Finally, the last quality attribute of the most important ones is reliability. To achieve a better service quality, the platform needs to be able to recover from application crashes. When an application that is running properly crashes, the platform must be able to detect

that automatically and launch a new instance of that application, providing in that way a better quality service not only for the end users but also for the developers because they will also appreciate that the platform does this for them.

3.2 Proposed Architecture

The architecture of the EMP was designed following the Simon Brown's C4 Model[30][31]. This approach in designing an architecture consists in drawing diagrams at different levels of abstraction (C1 - Context, C2 - Containers, C3 - Components and C4 - Classes). The focus will be in the first three diagrams. The context diagram is presented in section 3.2.1, the containers diagram is presented in section 3.2.2 and the components diagram is in section 3.2.3.

3.2.1 Context Diagram

The context diagram is useful as a starting point when designing an architecture. It is a way to look at the big picture and realize how to the main system will interact with its users and other systems.

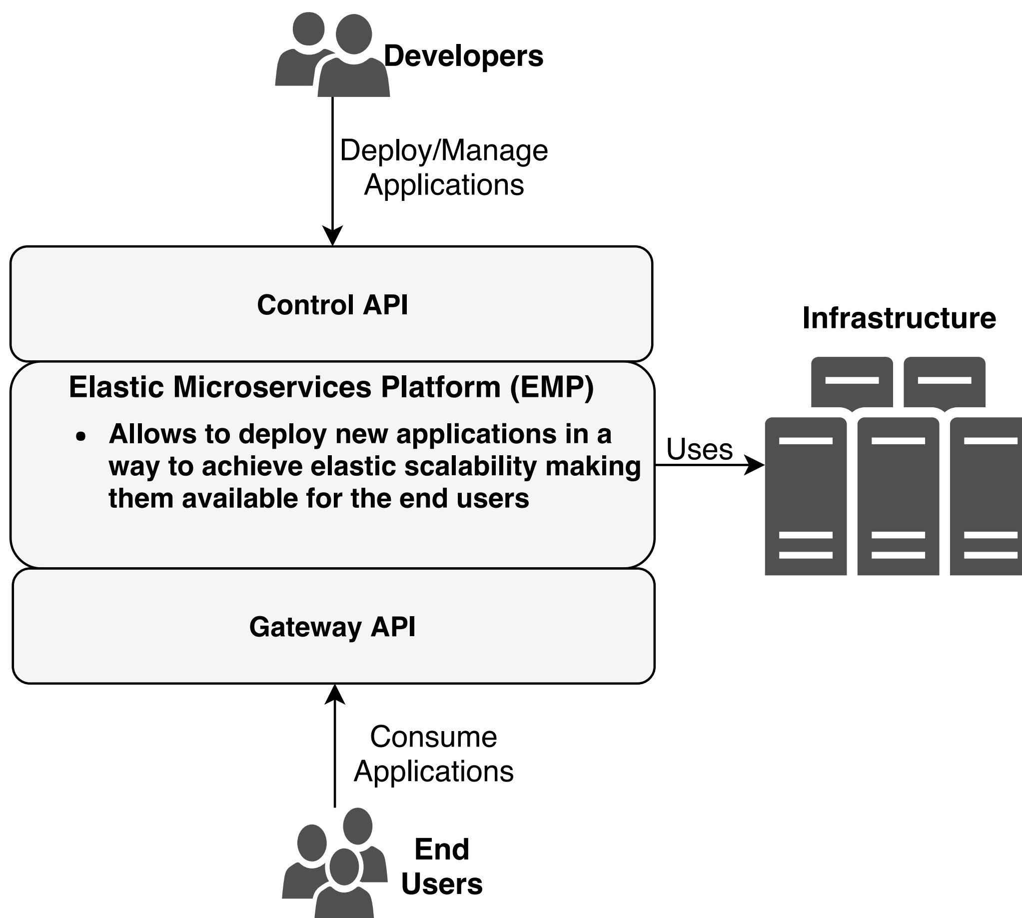

Figure 3.1: EMP Context Diagram (C1)

In Figure 3.1, it is possible to observe the platform's context diagram (C1). The

EMP system will have two type of users, developers who will deploy their microservices applications into the platform, and consumers/end users who will consume the applications that are running in the platform. Both types of users are represented in the diagram, and they need different access points to the EMP. The developers will perform their actions by using a *Control API* which is responsible to control and interact with the EMP. The end users will connect to the applications running inside the platform via a *Gateway API*. There will also be an *Infrastructure* in which the EMP will be installed on and also manages it.

3.2.2 Containers Diagram

After the context diagram was done, it was possible to start thinking about high-level technologies and how the containers will communicate with each other. An extensive research on possible technologies that could be used was made, taking into account that it was necessary to satisfy all the requirements presented in 3.1. It was now possible to decide which technologies are going to be used in the EMP.

It is to note that the decisions regarding why certain technologies were chosen over others were presented and analyzed in section 3.2.4.

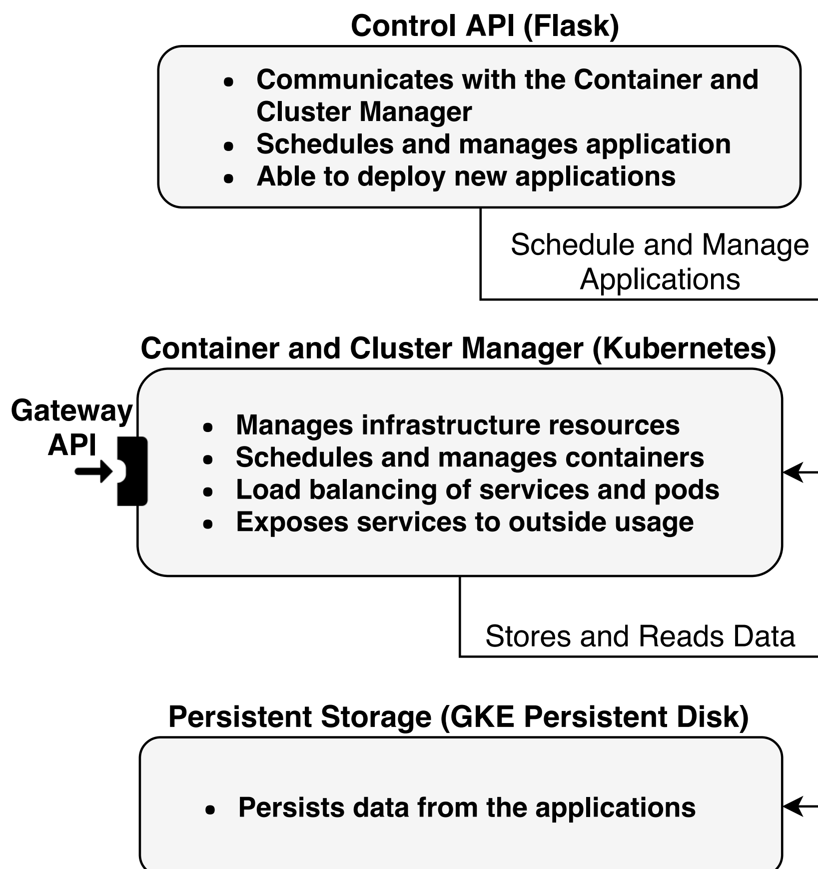

Figure 3.2: EMP Containers Diagram (C2)

In Figure 3.2, the platforms containers diagram is presented (C2), with the technologies used in each container. There are three major containers:

- Control API
  - The *Control API* is responsible to communicate with the *Container and Cluster Manager*, scheduling and managing applications, and allows users (developers) to deploy applications in the platform. This *Control API* can be a web server developed in *python*, using *Flask*[6] framework and an *OpenAPI Specification* (*Swagger*)[15]. It would be possible to scale this as needed, depending on the amount of users (developers) that would be using the EMP.
- Container and Cluster Manager
  - The *Container and Cluster Manager* will be responsible to manage the entire infrastructure, to load balance the services that are running in the platform, scheduling and managing the containers. It is also responsible to expose the services running to the outside world, providing a *Gateway API* for the end users to access it and to be able to consume the applications. The technology chosen for the *Container and Cluster Manager* was *Kubernetes*[29]. It uses the *Persistent Storage* to store and read information of the applications that are running in containers. *Kubernetes* is an excellent choice for this *Container and Cluster Manager* and was explained in better detail in section **??**.
- Persistent Storage
  - The *Persistent Storage* will be responsible to persist data from the application that are running inside *Kubernetes*. This persistent storage is going to be used if the users want their applications data to be persisted. The technology chosen for the *Persistent Storage* is *Google Compute Engine (GCE) Persistent Disk*[9].

To satisfy the maintainability quality attribute and the functional requirements of the tasks *Deploy Application and Manage Application*, the EMP needs to have a *Control API* that sits between the users (developers) and *Kubernetes*. The fact that the *Persistent Storage* is also exterior to *Kubernetes*, makes the system more modular because it is possible to swap these components. Elasticy is achieved because the *Control API* will be responsible to schedule and manage the applications that are running inside *Kubernetes*, launching or shutting down instances according to the current workloads. The *Gateway API* is represented because an access point to the applications that are running inside *Kubernetes* for the end users to consume is needed (REQ-7).

By choosing *Kubernetes* to be the *Container and Cluster Manager*, the quality attributes of scalability, performance, availability and reliability are satisfied. All these details of *Kubernetes* were discussed and analyzed in detail in section **??**.

3.2.3 Components Diagram

After the context and containers diagrams done, it is time to do the components one. The components diagram is a zoom in of the containers diagram, showing how each container is divided into components, what each component is and how do they interact. This is the last diagram that is going to be analyzed since the class diagram (C4) will not be covered.

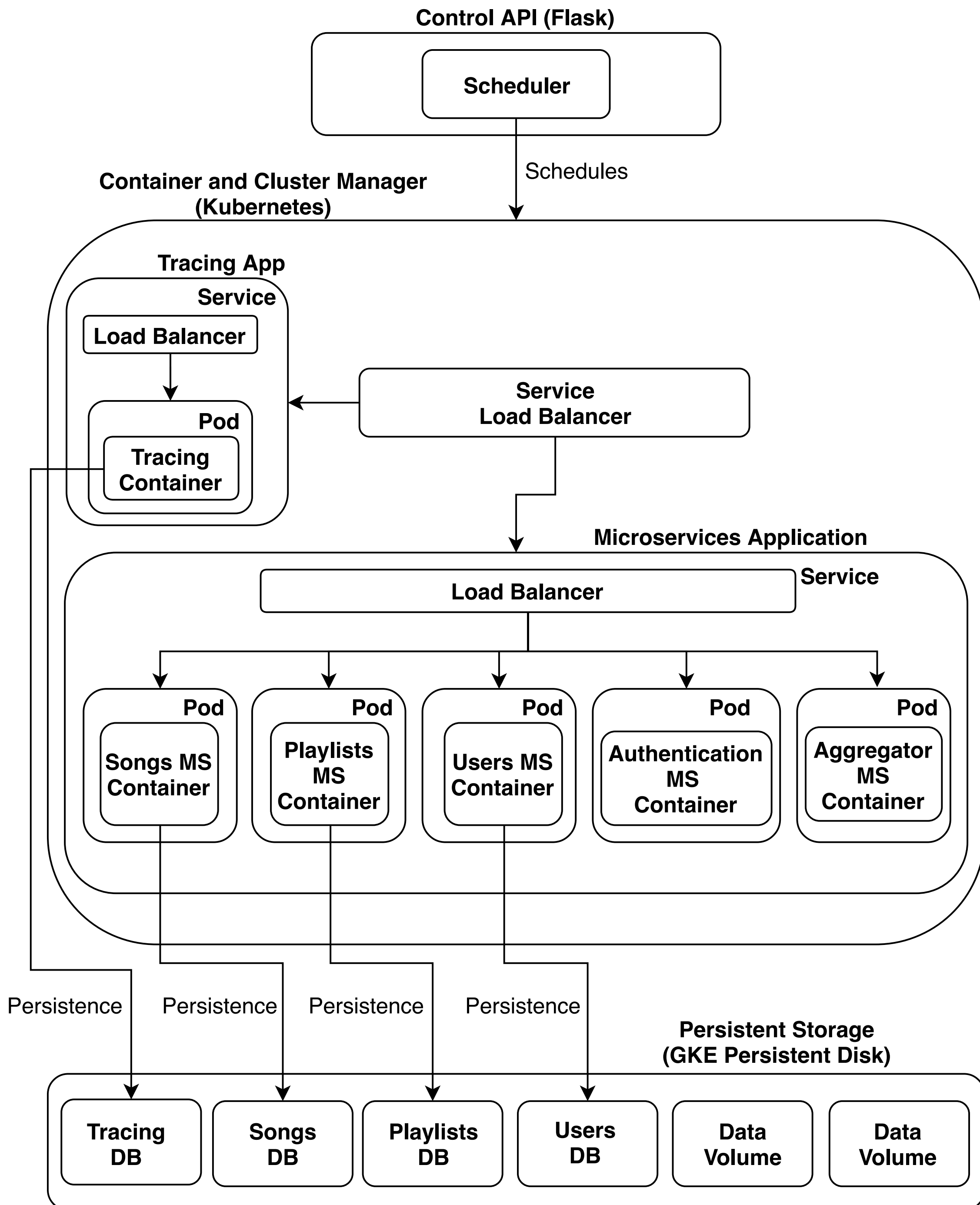

Figure 3.3: EMP Components Diagram (C3)

In Figure 3.3, the components diagram (C3) is presented. This diagram shows a lot

more detail than the other two and allows to see all the internal components.

- Control API
  - The *Control API*, as described in the previous diagram, will be implemented in *python* and *Flask* framework will be used. It has a *Scheduler* component that will be responsible to schedule and manage the applications instances by launching or shutting down applications based on the traces received from the *Tracing App*. As mentioned earlier, the *Control API* also allows users (developers) to deploy their microservices applications into the platform and manage them.
- Container and Cluster Manager
  - The *Container and Cluster Manager*, in this case will be *Kubernetes*, has a lot of important components. *Kubernetes* is able to load balance workload among each *Service*, using for example *Nginx*[14], using that component called *Service Load Balancer*. Another important component is the *Tracing App*. This component will be responsible to collect traces from all the applications inside *Kubernetes* and feed them to the *Scheduler* that is inside the *Control API* (REQ-14 and REQ-15 are satisfied). The *Tracing App* will also store all the traces in a persistent database inside the *Persistent Storage*. *Kubernetes* will have many services, for example the Microservices Application, running and inside each service there will be a *Load Balancer* to make sure the workload is distributed properly. Each *Service* can have many *Pods* in which the containers run inside them. This allows a better scaling because it is possible to launch new *Pods* to match the current demands for the application.
- Persistent Storage
  - The *Persistent Storage* is a distributed data storage that has several *Data Volumes* to store the users applications data. When a *Pod* dies, the data is lost, that is why a *Persistent Storage* was necessary in order to persist the applications data even when a *Pod*, *container* or *Service* goes down.

For the EMP system to be more modular, the *Control API* will be running in a different machine from *Kubernetes* allowing it to be replaced easily if needed.

3.2.4 Chosen technologies

Extensive research for technologies that were able to satisfy the functional requirements and the quality attributes the platform needed to achieve, was performed. The chosen technologies are going to be presented in this section.

It was necessary to decide which programming language and frameworks to use to develop the *Control API*. The language that was decided to use is *python* due to its simplicity in coding and its easy with the right tools. I also had experience with this language making it an obvious choice for me.

For the web server development, *Flask*[6] or *Django* could be chosen. *Flask* is a really powerful and easy to use web framework and since I already had experience using it, I chose it over *Django*. *OpenAPI Specification (Swagger)*[15] is going to be used to describe the API.

For the *Persistent Storage*, Google Compute Engine (GCE) Persistent Disk were chosen.

Both *Kubernetes*[29] and *Docker (Docker Swarm)*[24] were good choices for the Container and Cluster Manager. The reasons behind choosing *Kubernetes* as a *Container and Cluster Manager* were not only influenced by the aspects detailed in section 2.2.2, but also its ability to satisfy most of the quality attributes mentioned in section 3.1.2. *Kubernetes* is able scale up to 5000 nodes[39], satisfying in this way the most important quality attribute for the system. In terms of performance, *Kubernetes (v1.6)* is able to satisfy both quality attribute refinement. It is fast to deploy instances, since "99% of pods and their containers (with pre-pulled images) start within 5s"[39] as it is possible to see in Figure 3.4. *Kubernetes* API is very responsive because "99% of all API calls return in less than 1s"[39] as it is possible to see in Figure 3.5. It is to note that this information and graphics that were extracted from a website[39], really shows the performance and scalability potential *Kubernetes* has. The reliability quality attribute is also satisfied with *Kubernetes* since it has an automatic mechanism that detects if any Pod crashed (health checks), and makes sure to launch a new instance returning the system into a consistent state. Regarding the availability quality attribute, *Kubernetes* is also able to satisfy it, by having more *Master Nodes* running, providing an availability as high as intended. Since *Kubernetes* will be running on several machines, each *Master Node* can run on a different machine providing a higher availability. It is also necessary to replicate components such has storage and the API Server.

Since *Kubernetes* has advantages over *Docker Swarm* and is able to satisfy the quality attributes that are related to the *Container and Cluster Manager*, it makes perfect sense to choose it over *Docker Swarm*.

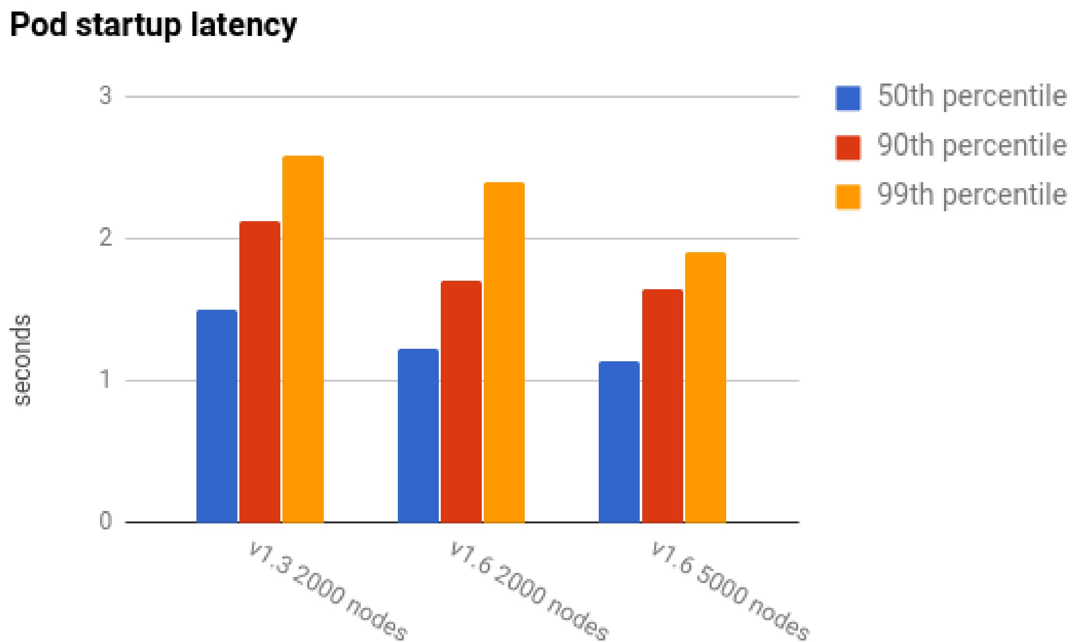

Figure 3.4: Kubernetes Pod Startup Latency[39]

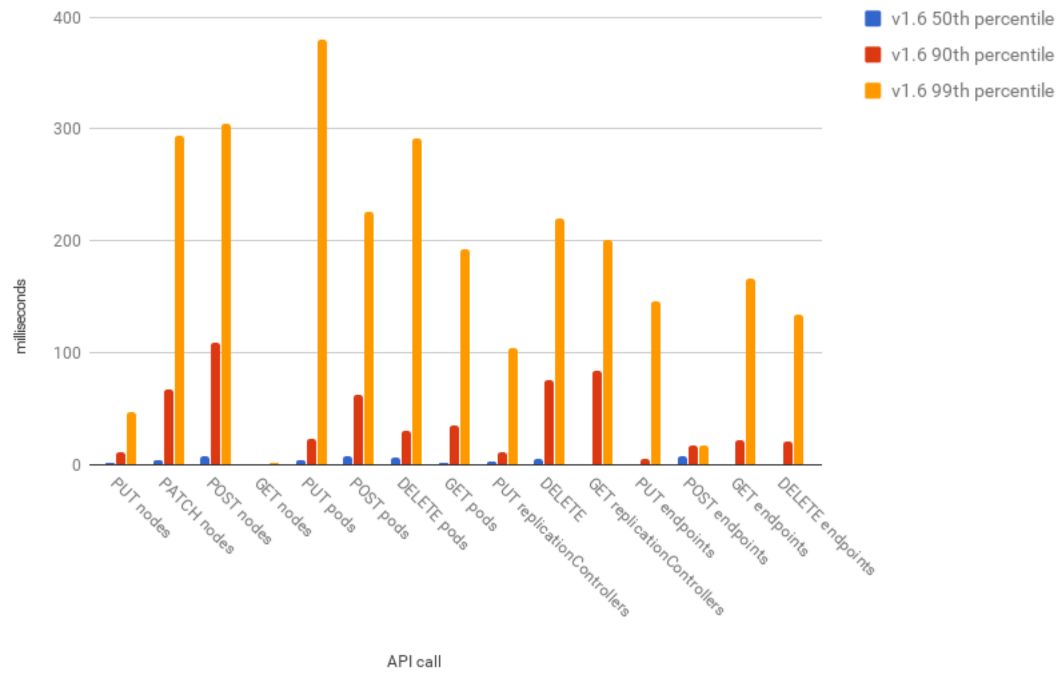

Figure 3.5: Kubernetes API Call Latencies - 5000 Node Cluster[39]

Chapter 4

Implementation

In this chapter a detailed description of the implemented components in the final platform of this thesis is going to be presented.

In section 4.1 the microservices application developed, that is going to be used to perform tests in the final platform, is going to be described in detail regarding its architecture and how I implemented it.

4.1 Microservices Application

In order to perform better tests in the EMP that is going to be developed, a microservices system was implemented to serve as a real example of an application that a user could deploy in this platform. This microservices system was based on a monolithic project that I developed in the course of Service Engineering, alongside with the student Fábio Figueiredo Pina, in the school year of 2016/2017. Much work needed to be done to turn this monolithic into microservices and some new features were added.

4.1.1 Original Project

The monolithic project was implemented in python 2.7 using the *Flask*[6] framework, *Swagger* for the REST API specification, *SQLAlchemy*[22] as an Object Relational Mapper and React[19] for its user interface. The goal was to develop a web application to manage several users and their music playlists in which they could add or remove songs available in the platform.

This application had three distinct layers:

- CRUD
  - A single database to store information about the users, songs and playlists.
- Business
  - Responsible to interact with the database and provides a REST interface to the outside. This layer had also an Open API specification for the client and server interaction called Swagger.
- Presentation

– This is the presentation layer for web browsers developed in React and communicates with the Business layer.

4.1.2 Architecture

The microservices application that was based in the monolithic project described above is composed by 3 small microservices (User, Songs and Playlists) and a Main_App that has the user interface developed in React and acts as a gateway for the requests to the microservices. Its architecture is shown in Figure 4.1 to better demonstrate how all the components communicate. It is to note that the microservices never communicate with each other directly. However there are certain operations that require information that is present in a different microservice. In that case, the microservice that needs that information will send a request for the gateway to request that data to the other microservice. One example of such operation is when it is necessary to show all the information about the songs that are present in a playlist. A request for the Playlists Microservice is made to get all its information about which songs it has, then that response arrives in the Main App which then makes another request to the Songs Microservice containing all the songs ID's that are present in the playlist to get all their songs information from the database. This ensures that isolation needed for the microservices, which is crucial to better scalability, is not possible with a monolithic application.

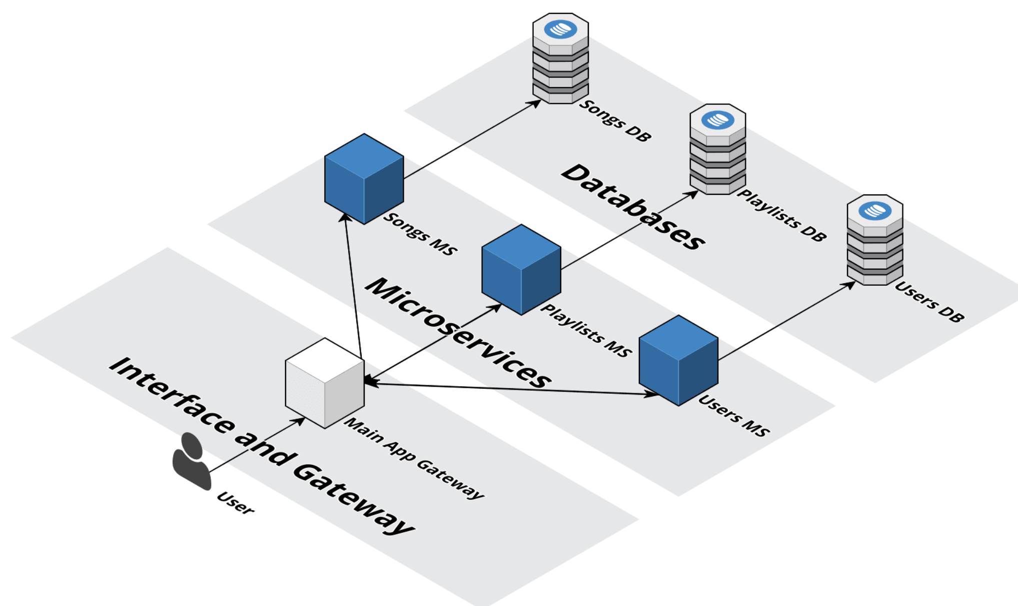

Figure 4.1: Microservices Application Architecture

It took a lot of work to break down the monolithic example into a microservices one, and a lot of problems were encountered. Some of the problems were due to lack of experience in using some tools or frameworks to develop the final microservices example such as *Flask-JWT*[7], *PyJWT*[17] and *Docker*[4] in general. It was also necessary to create three databases, one for the Users, one for the Songs and one for the Playlists. This took some time because in the original project, there was a single database containing all the tables that had relations to each other, making it harder to isolate each microservices information.

4.1.3 Users Microservice

This microservice is responsible to handle all the requests that are related to a given user. For the authentication part of the microservices application, json web tokens were used. Since *Flask-Login*[8] presented a lot of problems and faults in the original project, *Flask-JWT* was used to solve all of that. By sending the encoded authorization token in the headers request is possible to easily check if a user is logged in or not, if the token has already expired or if the user has permissions to make such request by decoding and validating the token.

This microservice features are:

- Create, Update and Delete a user
- Get a specific user information
- Check if a user exists

4.1.4 Songs Microservice

This microservice is responsible to handle all requests that are related to songs. Unlike the Users Microservice, it was not possible to check the user authentication using *Flask-JWT* because it was necessary to implement 3 default functions for that to work and those functions needed to read the user object from the database, which was only possible in the Users Microservice. Since all the microservices are isolated and do not communicate to each other and they also do not have access to each others databases, instead of using *Flask-JWT*, *PyJWT* python library was used instead. With *PyJWT* it was possible to decode the token that was sent in the heads of the request without the need to read the user object from the database. The only drawback from this solution is that in the Main App gateway, when a request was made for either the Songs or Playlists microservice, the word "JWT" needed to be appended at the start of the token for it to work properly when decoding in those microservices that implemented the *PyJWT* instead of the *Flask-JWT*.

This microservice features are:

- Create, Update and Delete a song
- Get a specific song
- Get songs that satisfy a given search criteria
- Get all the songs of a user

4.1.5 Playlists Microservice

This microservice is responsible to handle all the requests that are related to playlists. This microservice presents the same problem that was stated in 4.1.4 regarding the authentication verification problem and the solution implemented was also the same.

This microservice features are:

- Create, Update and Delete a playlist

- Add/Remove a song to a playlist
- Get a specific playlist
- Get all the songs from a playlist
- Get all the playlists of a user

4.1.6 Main App Gateway

This component is responsible to redirect all the requests to the correct microservice, in the end it acts as a gateway. A gateway connects two different components and is responsible to manage and redirect the traffic between the two. It is also in *Main App Gateway* that the user interface that was developed in React is present. This is where the core of the applications logic is and when it is necessary to use information from more than one microservice to fulfill the users request, this component handles all that is necessary to achieve that. Python "requests" library was used to make the HTTP requests necessary to the microservices. The interface was developed using Bootstrap and React. It is to note that the user will need connect to this Main App Gateway in order to see the interface and perform the actions desired. It is also possible to access the microservices without the user interface implemented since they were developed in a way that makes that possible.

4.1.7 Running everything on containers

After the application was successfully divided into microservices and was running properly, it was time to make everything run on containers. It was necessary to research the options available and *Docker* was chosen for its popularity and several features. To begin with, a "Dockerfile" for each microservice was written, specifying which port is available, which dependencies is docker going to install in the container with "requirements.txt" file. This file was generated for each microservice with all its dependencies and also the environment variables.

Once each microservice was built it was time to run the container and the app. After that was successful, the next step was to publish those images into the docker repository for further use.

After all the images were published into docker repositories, it was time to find a way to make everything run in a fast and simple way. The solution for that was *Docker Compose*[16]. To actually put everything running properly in *Docker Compose* was a bit challenging because it was necessary to research and learn how to do it. The environment variable mentioned earlier are important for this step because it allows the application to run properly when docker launches a container for example of the database and links it in runtime with the other containers that are running the microservices. When a microservice needs to access the database, it will know the container's IP address and successfully connect to it because an environment variable will be used by docker when the database container launches. It is to note that the database is an official image of *MariaDB*[12] and is also running on a docker container.

Docker Compose makes it simple once it is all well defined because with the help of one command in the terminal console, the application is up and running smoothly.

4.2 EMP CLI

The implementation performed during the second semester, started with this EMP CLI. This CLI is responsible for the interaction between the developers and the EMP Server. It is a command line interface that allows users to execute tasks, guiding them in a way that it compatible with the EMP system.

A simple illustration on how the CLI fits among the other components can be seen in figure 4.2

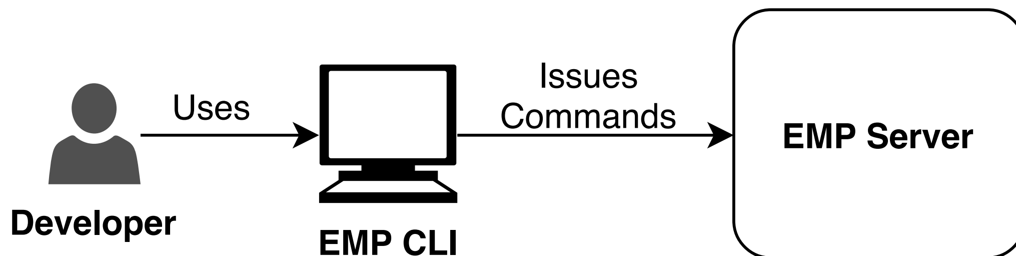

Figure 4.2: EMP CLI Overview

I needed to find a tool that allowed me to build this CLI, and so I searched on the internet for something that would meet my requirements. I found a Python package called *Click*[3] that allowed me to build a command line interface. There were more tools similar to *Click* but this one looked simpler and had a better documentation which impacted my decision to choose it.

The first step I took to develop my CLI was to define all the options that the CLI would have, what parameters they should receive, and what kind of response each command would return. After all the options were defined I started by doing a simple implementation that would just execute the command line interface commands and print a sample response.

At this point, I needed to have a simple implementation of the EMP Server to be able to test my CLI while finishing its implementation. To do so, I described the entire REST API for my application using the Swagger Online Editor[23] and following the OpenAPI Specification[15]. The resulting *yaml* file contains all the REST endpoints defined as well as their input and output response types. This is useful because we can validate a request or a response based on its type before it is executed. After the REST API was described, I used Swagger Online Editor to generate the client and server stubs for python language with Flask framework. Once that step was completed, I was now able to integrate the generated client code with my CLI and test it with the simple generated server. This allowed to me to finish the entire CLI while testing its connection to the server.

The commands that are available in the EMP CLI are:

- emp create_account
  - Creates a new user account based on the username and password provided. This command requires a "username" and a "password". The functional requirement REQ-1 is satisfied.

- emp deploy FILE
  - Deploys an application in the platform. This command requires a path to a file. This input file must be in json format and contain the following fields:
    * name - Name of the application.
    * docker_image - Docker image for the application to be deployed.
    * stateless - If the application is stateless set it to "true". Otherwise set it to "false".
    * port - Port number desired for the application to run.
    * envs - Array of environments variables that must contain the following elements: "name" (environment variable name) and "value" (value for that environment variable)
    * quality_metrics - Contains an array of the following elements: "metric" (metric name) and "values" (valued for that metric).

    The functional requirements REQ-3, REQ-4 and REQ-5 are satisfied.
- emp info ID
  - Returns all information about a specific application in the platform. This command requires an "id" of a specific application as an argument. The functional requirement REQ-9 is satisfied.
- emp list
  - Returns all information about all applications of the current user in the platform. The functional requirement REQ-13 is satisfied.
- emp login
  - Authenticates a user by validating its username and password. This command requires a "username" and a "password". The functional requirement REQ-2 is satisfied.
- emp remove ID
  - Removes an application from the platform. This command requires an "id" of a specific application as an argument. The functional requirement REQ-12 is satisfied.
- emp start ID
  - Starts an application that is stopped in the platform. This command requires an "id" of a specific application as an argument. The functional requirement REQ-11 is satisfied.
- emp stop ID
  - Stops an application that is running in the platform. This command requires an "id" of a specific application as an argument. The functional requirement REQ-10 is satisfied.
- emp tracing ID
  - Returns a link containing traces of a specific application. This command requires an "id" of a specific application as an argument. The functional requirement REQ-14 is satisfied.

- emp update_metrics ID METRIC VALUES
  - Updates the application quality metrics. This command requires an “id” of a specific application as an argument, the “name” of the quality metric to update and the “values” for that metric in the form of a string. The functional requirement REQ-8 is satisfied.

4.3 EMP Server

The *Control API* is responsible for all the logic necessary to operate the entire EMP system. We can divide it into two main components:

- EMP Server
- Scheduler

The EMP Server can be seen as the core component of the EMP. It is responsible to interact, control and maintain *Kubernetes*. It is to note that before implementing this component, careful planning was made to ensure that the platform could meet the requirements that were specified, namely the modularity one.

In figure 4.3, and overview of the main interactions of the EMP Server is presented.

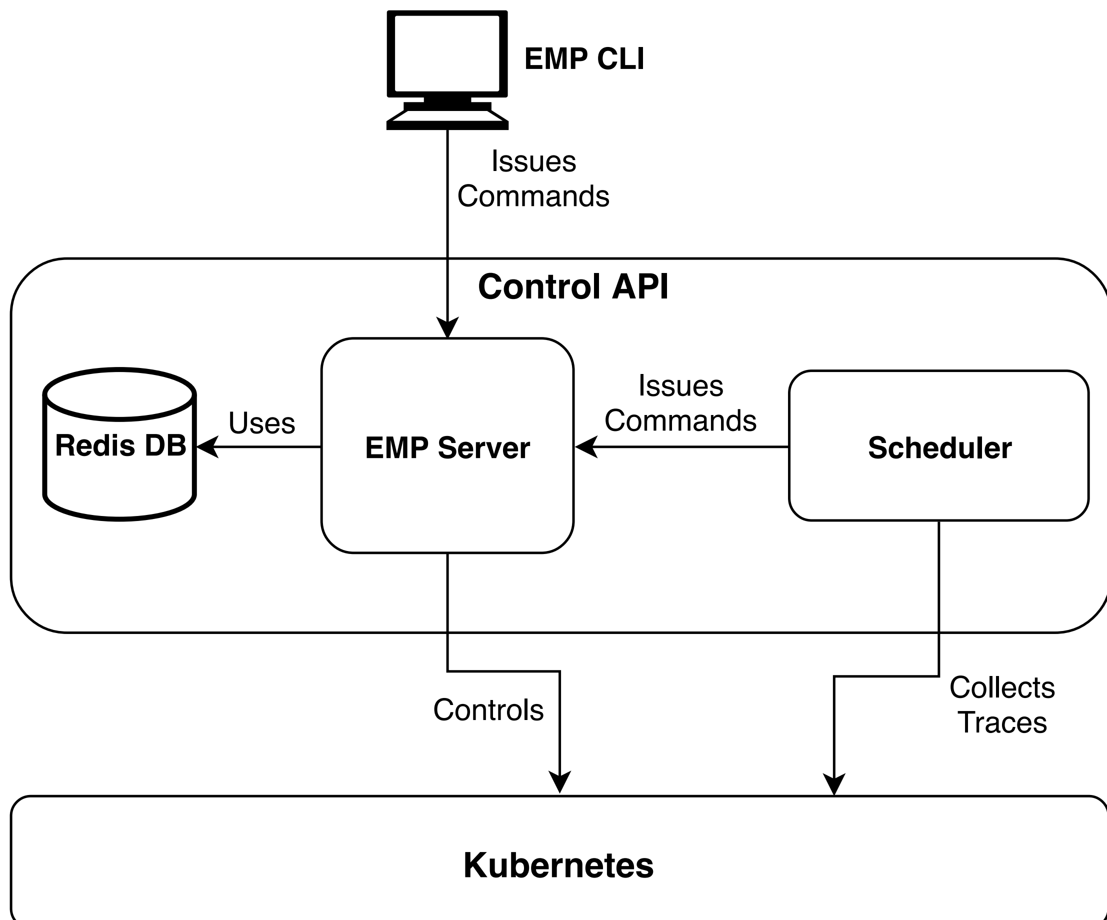

Figure 4.3: EMP Control API Overview

When starting to develop the EMP Server, the first step was to generate the python flask server from the REST API specification that I implemented. The EMP Server has three main code files, called modules that were structured in such way to achieve a higher level of modularity. Those three modules are:

- emp_server
- cluster_manager
- kubernetes_controller

In figure 4.4, an overview of the EMP server modules is presented.

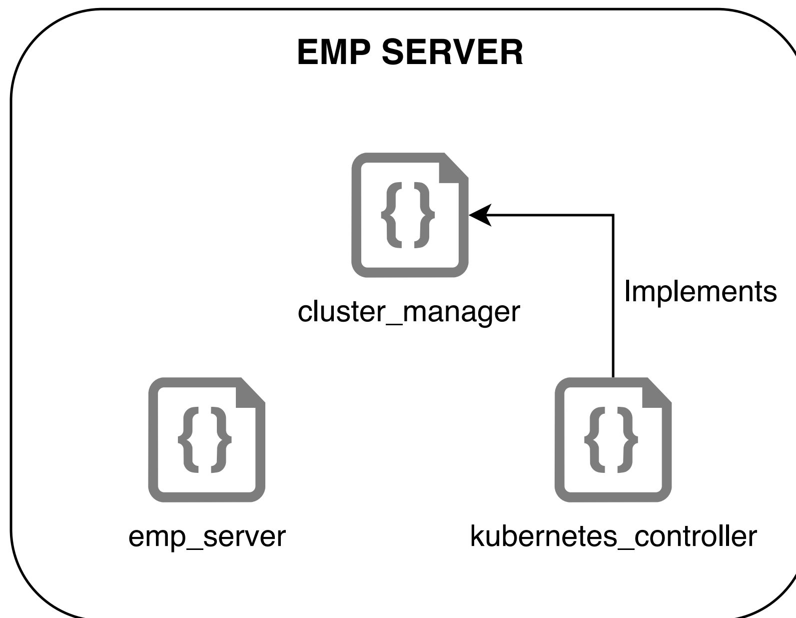

Figure 4.4: EMP Server Files

4.3.1 EMP Server Module

The *emp_server* module is where all the requests coming from the EMP CLI or the *Scheduler* component, will be processed and executed. As we can see in figure 4.3, the EMP Server component uses a *Redis*[20] database. It is in this file that all the operations involving using *Redis* were implemented. This database is where all the information regarding the platform is stored in order to check and validate the platform's state and requests made. *Redis* is a very fast database that works as a key-value, like an hash map, and is able to store more complex data structures. It is open source and was chosen due to its simplicity and performance. The information stored on *Redis* is structured in this way:

- A hashmap for the users information, storing details about their username and password.
- A hashmap taht keeps each user's applications information regarding their general details and state in the platform.

It is to note that each application has an Universally Unique Identifier (UUID) that was generated and assigned to identify and keep track of each application. Since this *Redis* database is always updated, when the user requests any information, there is no need to execute a command directly to the *Kubernetes* cluster overloading it. Instead, the information will be retrieved from the *Redis* database, unless it is really necessary to access the *Kubernetes* cluster for some reason.

4.3.2 Cluster Manager Module

The *cluster_manager* module is almost like a python interface. It is a class that has methods which will throw a *NotImplementedError* in the descendent classes if that method is not implemented. This is a useful way to define all the methods a given class must implement in order to work with the EMP system. For example, in my case I developed the *KubernetesController* class that implements all the methods in the *ClusterManager* and handles all the *Kubernetes* interactions. If for some reason it is necessary to replace *Kubernetes* as a *Container and Cluster Manager* for something like *Mesos*, it is possible and simple to do. All that is needed to implement is the specific class for *Mesos* that follows the *cluster_manager* "interface" and it works. This is an important aspect since the platform will be open source and users will be able to use it and replace components if they desire, in a simple way.

4.3.3 Kubernetes Controller Module

The *kubernetes_controller* module, as it was said above, implements all the methods specified in the *cluster_manager* for the platform to successfully work with *Kubernetes* as its *Container and Cluster Manager*. It is to note that this *kubernetes_controller* module was implemented after *Kubernetes* was installed and configured in order to test and better implement the necessary operations. To implement the *kubernetes_controller* module, kubernetes-client[11] for python was used. It was difficult to determine which *Kubernetes* API endpoints I needed to use because the documentation is not clear on what each of them does.

To better control and scale a given application, I needed to deploy an application as a *Kubernetes* deployment and not a stand alone application. That way I can easily control the number of instances a given *Kubernetes* deployment must have which is very useful for this system. Upon creating a deployment of a given application, I will also need to create a *Kubernetes* service, if the user so desires, to expose that application to the outside world. Without a service, an application or deployment that is running inside *Kubernetes* cannot be accessed outside the *Kubernetes* network. It took some time to have these features working because the documentation lacks a detailed explanation on each command.

4.4 Scheduler

The Scheduler, is a small component that belongs to the *Control API*. This is where the automatic decision on scaling elastically the applications that are running on *Kubernetes* happens. It collects the application traces from *Kafka*, that is deployed inside *Kubernetes* and automatically analyzes them. After such analysis, a decision about the need to scale up or down a specific application is made and a command is issued to the EMP Server.

It is important to note that this small component will be implemented by Eng. Jaime Correia. This is the only component that is not yet implemented. The platform is prepared to receive the *Scheduler* commands from outside, using REST, making it fully functional and ready to use. The impact caused by this component not being implemented yet, is the platform not being able to automatically analyze and scale the applications based on their tracing information. However, the platform is able to execute all the commands mentioned in 4.2. Although it does not scale automatically yet, that specific command is already implemented and can be executed manually. The same commands that the *Scheduler* might send the EMP Server to execute can be sent manually, making the entire platform completed with exception of the *Scheduler* that will be Eng. Jaime Correia implementing it.

4.5 Container and Cluster Manager

Since I chose *Kubernetes* to be the *Container and Cluster Manager*, I needed to install and configure it. This proved to be the hardest and the most time consuming task of them all. There was a lot of struggle to achieve the resources necessary from helpdesk of the Department of Informatics Engineering (DEI) to have a *Kubernetes* cluster. Since they took some time to make the resources available, I started by experimenting *Kubernetes* on my local machine using *minikube*.

Minikube is a simple and local installation of *Kubernetes* with just one node, that is really useful for testing and learning purposes. While I was using *minikube*, I also used *Helm*[10] to simplify the deployment of applications that were necessary for the EMP. *Helm* is a package manager for *Kubernetes* and instead of having to configure and deploy by myself an entire application such as *Kafka*, *Helm* does it all automatically. The goal was to have the deployed application's traces sent over to *Kafka* that was running inside *Kubernetes* and have the tracing system collect them and present them in their UI. That is why I looked into *Helm*, because I would need to deploy *Kafka* and a tracing system inside my *Kubernetes* cluster, and *Helm* does it easily. This is where the problems began. *Kubernetes* is really hard to debug and it takes a huge amount of time to read the documentation necessary to find a solution for a given problem. I tried to deploy *Kafka* and tracing systems *Jaeger* and *Zipkin* using *Helm* but I was not successful. I later found out the problem was due to not having enough resources on my personal computer. I had to postpone the *Kubernetes* installation because DEI's helpdesk was taking too long to make the resources necessary available. In the mean time I was working on other parts of the project. I read both *Zipkin* and *Jaeger* documentations and saw their available clients for instrumentations and decided to go with *Zipkin*. *Jaeger* is not really compatible with python 3 and since the microservices application was rebuilt using python 3, I would have some problems when instrumenting it to support tracing. That made my choice clear and simple to just use *Zipkin* and a community instrumentation for python that I found called *py_zipkin*[18].

Eventually they provided some resources for a cluster and I could finally start trying to install and configure *Kubernetes* on bare metal. Unfortunately installing and configuring *Kubernetes* on bare metal proved to be very hard and time consuming, resulting in a decision to use *Google Cloud* to deploy the *Kubernetes* cluster.

Kubernetes documentation focus more on deploying it on the cloud such as *Amazon* or *Google* cloud instead of installing it on a custom cluster. All that happened during the *Kubernetes* installation on bare metal is presented in section 4.5.1 and all the details

regarding the *Kubernetes* installation on *Google Cloud* is presented in section 4.5.2.

4.5.1 Kubernetes in Bare Metal

Once DEI's helpdesk provided access and the resources necessary for creating a custom cluster, I started to install and configure *Kubernetes*. It took a lot of hard work to understand how to install *Kubernetes*.

At first, the idea was for me and Fábio Pina to install *Kubernetes* on DEI's cluster together, since he would also need and benefit from it. We tried to install it by creating three machines with *Fedora* as their operating system. One of the machines was the *Master Node* and the other two were the *Workers*. We followed a tutorial that we found online and used the *Kubernetes* documentation to help. We were unable to install *Kubernetes*. After that, while I was working on the EMP server implementation, Fábio Pina tried to install *Kubernetes* alone during a week and was also unsuccessful. He then decided that for his thesis, instead of using *Kubernetes* he would use *Docker Swarm*.

I tried to postpone the *Kubernetes* installation as much as I could because it was delaying my thesis but I reached a point where I really had to install and configure it in order to move forward with my work. This time I followed several different tutorials while using *Kubernetes* documentation and was finally able to install it on three machines. Those machines had *Ubuntu* operating system installed, and were part of my *Kubernetes* cluster. One of the machines was the master node and the other two were the workers. To test my *Kubernetes* installation I tried to deploy a simple application and see if that would work on my custom cluster. At this point, everything seemed fine but a lot more problems were coming into my way.

After the *Kubernetes* cluster was up and running, I installed *Helm* so I could deploy both *Kafka* and *Zipkin*. I was hoping that installing *Helm* and using it to deploy both *Kafka* and *Zipkin* would be simple but instead it was really hard and very time consuming. After *Helm* was successfully installed into my custom *Kubernetes* cluster, I followed the instructions to deploy *Kafka* and *Zipkin* but they both did not work. It is really hard to debug why specific a *Pod* or *Service* is not running inside *Kubernetes* because their *Helm* charts had a lot of configurations. In this case, the problem was that the resources were not enough, so more nodes were added to the *Kubernetes* cluster and their *RAM* and *CPU* cores increased. After that problem was taken care, both *Kafka* and *Zipkin* were not working properly. Since this was a bare metal installation of Kubernetes, it was necessary to declare a *Storage Class* so *Kubernetes* knows where it can store the applications persistent data. In *Google Cloud*, this is done automatically, since they have a default *Storage Class* that uses *Google's Persistent Disks* to store the data. To solve this problem I had to learn how to create a *Kubernetes Storage Class* by reading their documentation and understanding how it works alongside *Kubernetes Persistent Volumes* and *Persistent Volume Claims*. I thought I could use each *Kubernetes* node own storage to persist my data but instead, I choose to build a *NFS* server. A *NFS* server is a distributed file system, so each *Kubernetes* node could write and read from that *NFS* server, so the data would be available for all the nodes.

Implementing a *NFS* server and configure it so each *Kubernetes* node is able to read and write from it was my next step. Once the *NFS* server was working, it was time to test if the *Kubernetes* nodes were able to use it. To do so, I connected via *SSH* to a worker node on *Kubernetes* and tried to create a file in that *NFS* server and access it from a different node. It still did not work because I had to make some adjustments to the

Kubernetes Storage Class to use the *NFS* server. I could now create a file and access it from a different*Kubernetes* node, meaning the *NFS* server was working. I tried to deploy *Kafka* and *Zipkin* but they still did not work. I saw in the documentation that I would needed to manually create a *Persistent Volume*. With this, *Kafka* application was now working properly but *Zipkin* server still was not. I later found out that for that specific *Helm* chart of *Zipkin*, I had to manually define its *Persistent Volume Claim*. A *Persistent Volume* is a piece of storage in the cluster while a *Persistent Volume Claim* is a request that is made for that storage. The *Helm* chart for *Zipkin* was not able to automatically create a *Persistent Volume Claim* unlike *Kafka*, so I had to do it manually. At this moment I finally had both *Kafka* and *Zipkin* server working as expected.

It was now time to start implementing and testing the EMP Server interactions with *Kubernetes* as it was mentioned in 4.3. To access the *Kubernetes* cluster, a *CLI* called *kubectl* is required. This *CLI* must be configured to have permissions to access a given cluster. So in order to access the *Kubernetes* cluster from the EMP Server using the python API, I needed to provide it with a configurations file. After successfully accessing the *Kubernetes* cluster using the EMP Server, I could finally start testing and implementing its *kubernetes_controller* module. This implementation was hard since I had to learn everything by myself using *Kubernetes* documentation and when that documentation was missing, I had to do it by trial and error. One important aspect about deploying applications inside the *Kubernetes* cluster, is that the applications will end up in the *default* namespace by default. Instead of having all the applications inside the same namespace, for each user I create a unique namespace based on their username. This way, each user will have their applications running inside their own namespace, achieving a better platform organization.

When everything seemed to be going well, I found the biggest problem regarding this *Kubernetes* installation on bare metal. To be able to access an application outside the *Kubernetes* network, it is necessary to expose an application using a *Kubernetes* service. When I tried to deploy an application early on and see if it worked on the *Kubernetes* installation, I actually accessed it while inside a node, so I was accessing it inside its own network. To solve this issue, I tried many different solutions. It was really hard to find something useful on their documentation regarding my problem. Their documentation is really good when it comes to deploying *Kubernetes* on the cloud such as *Amazon's* or *Google's*, but when it comes to bare metal *Kubernetes* installation, they do not provide a detailed guidance. I read their documentation, searched on the internet for solutions and tried many of those with no success, until I joined their *Slack* chat room. There I was told that since I was using *Kubernetes* on bare metal and not the cloud, I had to install a network load balancer like *MetalLB*[13], and have a static IP range being routed to it using BGP. I needed to ask DEI's helpdesk to give me a static IP range and that would be something that either they would refuse or take too much time to fulfill my request because they were currently having a lot of technical problems. For that reason and because I would also need to configure this whole network and my knowledge on the subject is limited, I decided that the best solution was to install and configure *Kubernetes* on *Google Cloud*. The installation and configuration needed to deploy *Kubernetes* on Google Kubernetes Engine (GKE) is presented in section 4.5.2.

4.5.2 Kubernetes in GKE

Although deploying *Kubernetes* in bare metal was not possible, it allowed me to learn a lot more about and how it works in its low level. I gained a lot of knowledge that I would not be able to if I just deployed *Kubernetes* on GKE. This knowledge and experience that

I got from deploying *Kubernetes* in bare metal, was really useful when deploying it on GKE. I also encountered some problems while using GKE but in the end it was a really good decision to use it for my *Kubernetes* cluster deployment.

Unlike the *Kubernetes* bare metal deployment, in GKE I was able to use their UI for some of the simple tasks which really accelerated the installation process. I knew that I would need to delete my *Kubernetes* cluster on GKE several times and deploy it again, so to make this process faster, I wrote a script. Instead of using the GKE UI, I had to learn how to use their CLI called Google Cloud Shell (gcloud). With gcloud I was able to deploy and configure my *Kubernetes* cluster by executing terminal commands. After learning and getting used to gcloud, I wrote the script that would deploy my *Kubernetes* cluster on GKE automatically.

Once I had the *Kubernetes* cluster deployed and configured on GKE, I installed *Helm* and tried to deploy both *Kafka* and *Zipkin* charts. Both applications did not work but since I already had some experience using *Kubernetes*, I was able to detect that the problem was due to the storage provisioning. This time I had to learn how to use the *Default Storage Class* and how to create a *Volume* on *Google Cloud*. After reading the documentation, I was able to create a *Volume* and activate *Kubernetes* automatic storage provisioning. *Kafka* was now working but *Zipkin* still did not. I even tried to manually create a *Persistent Volume Claim* to assign the existing *Persistent Volume* to *Zipkin* but that did not work either.

Zipkin's Helm chart was very complex and had a lot of configurations in its yaml file that I did not understand. I tried to use different approaches to deploy it, tried different configurations and charts but was still unsuccessful. I also tried to change *Zipkin's* deployment file and customize it, using different ways to change the existing configuration.

This was taking too much time, and I needed to make a decision. Instead of using an existing chart to deploy *Zipkin* inside *Kubernetes*, I created my own *Zipkin Kubernetes* deployment file. I started by creating a docker container with *Zipkin* server in it and test it locally on my machine. After that step was complete and after creating my custom *Zipkin* docker image, I began to write its *Kubernetes* deployment file. The *Zipkin* charts that I tried to deploy did not work because there was a database pod that was causing an error and was not able to be up and running. Since I was building my custom *Zipkin* deployment file, I chose to deploy it without its own database because I was really getting so delayed by all the problems that I encountered. Finally I was able to successfully deploy *Zipkin* on my *Kubernetes* cluster and was ready to move on to the next step.

It was now time to start instrumenting the microservices application so it would serve as an example on how the EMP would handle a deployed application that had now tracing capabilities. This would reflect a real user deploying his instrumented application in order for it to scale and be analyzed by the EMP automatically. All the instrumentation process and all the changes that were made before that, are presented in detail in section 4.6.

After the microservices application was instrumented, it was time to test it on the EMP. When deploying this application, a set of environment variables need to be injected, so this applications knows the address of the *Kafka* application to send its traces. To be able to access the application from outside the *Kubernetes* network, it is necessary to create a service. *Google Cloud* assigns an external IP for each service that is running inside the *Kubernetes* cluster. With a service, end users are now able to consume the deployed application via its IP address.

Since the application is instrumented, each request and operation generates a trace,

which is sent to *Kafka*. The *Zipkin* server will consume those traces from *Kafka* and present them in a user friendly interface. For this entire flow to work, additional work regarding the *Zipkin* and *Kafka* configuration was required. In *Zipkin's* UI, it is possible to see the order of the requests, what microservices or functions were used and how much time each request took to complete. This information can also be consulted by the developers since there a option for that in the EMP CLI, satisfying the functional requirement REQ-14.

It was necessary to reconfigure the EMP server access permissions to the *Kubernetes* cluster because I was now using GKE. Once that was taken care of, I ran some simple tests to check if everything was working as expected, by deploying applications using the EMP CLI and see if they would be up and running inside *Kubernetes*. At this point, small improvements in every component were made to ensure a more mature and complete work.

In figure 4.5, a *Kubernetes* deployment on GKE overview is presented.

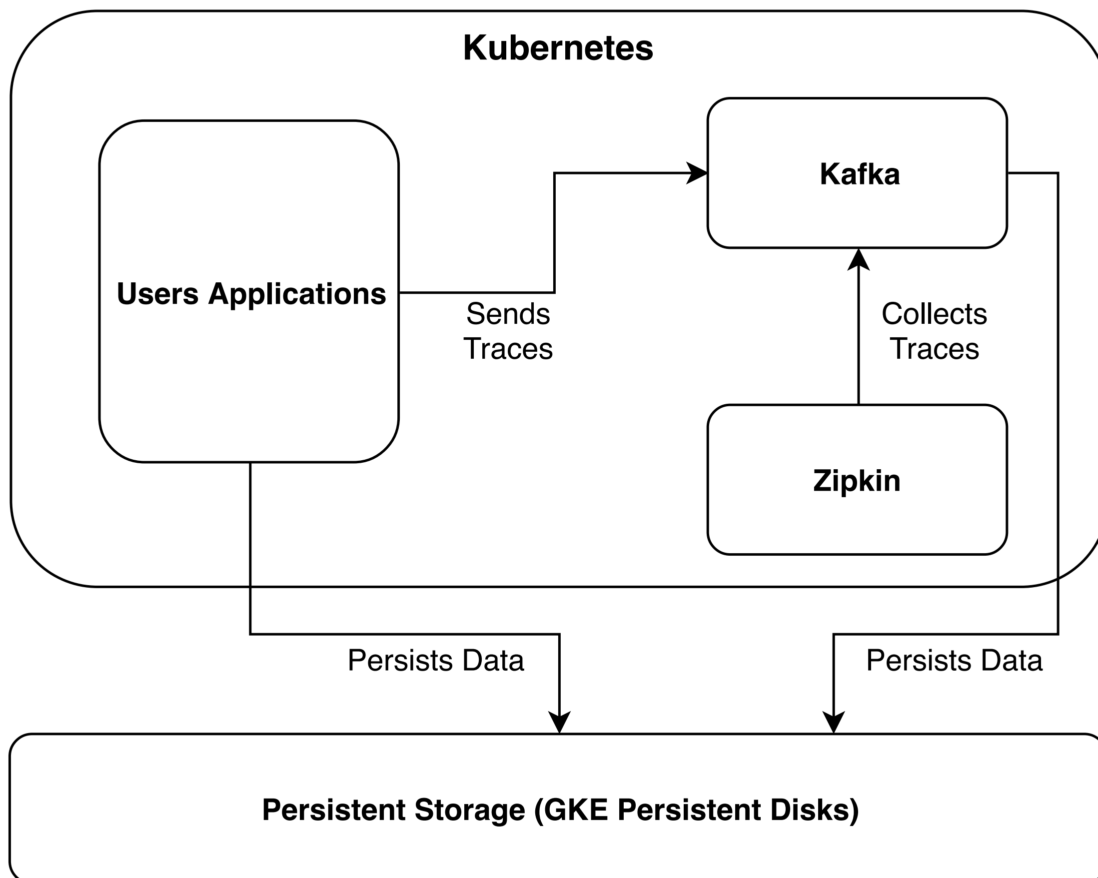

Figure 4.5: EMP Kubernetes Overview

As it was mentioned above, inside the*Kubernetes* cluster I deployed *Kafka* and *Zipkin* as part of the Infrastructure operations. The instrumented users applications will send traces to *Kafka* that will store them in the *GKE Persistent Disks* and *Zipkin* server will collect them and present that information in its UI that is available to the developers. If the users applications need persistent storage, they will use *GKE Persistent Disks*.

4.6 Microservices Application Instrumentation

To be able to fully test the EMP system, I needed to instrument an application that I could use to deploy inside *Kubernetes* and that it would send traces over to *Kafka*.

Fábio Pina also needed a microservices application for his work, so he took the one that I implemented and described in section 4.1 and improved it. He upgraded the application from python 2 to python 3. Since the application had its authentication inside the *User's* microservice, he created a new microservice just to handle the authentication operations. He also created a new microservice called *Aggregator*, that would just make multiple request to the others microservices so the application flow is more complex for testing purposes. Finally he also improved the overall application structure.

Instead of using the application that I originally developed, I used Fábio Pina's version that was upgraded from mine. Since I chose to use *Zipkin*, I needed to find a *Zipkin* python library to instrument my application. The library that I chose to use is called *py_zipkin*[18]. This library was chosen because it appeared simple to use and already had an example on how to use *Kafka* as the transport layer. The goal was to be able to trace the applications information and send them over *Kafka*.

When I started the microservices application instrumentation, I had a *Zipkin* server running on my local machine for testing purposes. I did not start immediately using *Kafka* because that would add another complex component for the development. Instead I used *HTTP* as a transport layer and sent the traces directly to *Zipkin* server to see if the application requests were being traced successfully. This allowed me to quickly test if things were working or not and improved the development speed overall. It is to note that after an update, this library had a bug which I reported on their *Github* page and was later fixed.

A trace has one or more spans and each span has information regarding each requests such as what was the time the request started and ended, the microservice name, the function that was executed and more information if necessary. After I implemented the instrumentation on one of the microservices, I ran some tests and thought that everything was working as expected. Once I instrumented another microservice and ran some tests regarding requests made from one microservice to another, I saw that when that happens, the tracing information is not properly shown in *Zipkin's* UI. If for example a microservice A does a request to microservice B, there needs to be one trace that has at least two spans. One span describing the A request and another span that is descendant of the first, describing the request of B. Instead, I would get two different traces with one span each, which means that the library was not doing the operations correctly.

This py_zipkin library is not an official library, it is a community implementation and it barely has any documentation. I contacted directly one of the contributors and explained my problem and ways of thinking. He told me that in most cases their library is used to trace website pages, in which the trace begins at the *index.html* and the pages that follow will be descendants of that span. Since I was not able to use py_zipkin library in the way it was implemented to solve my problem, I decided to implement a custom decorator myself. Based on py_zipkin, I implemented a custom decorator that it would take those conditions into account. This implementation was challenging since I never did a decorator by myself and because I was using a library that had almost no documentation to read and understand. To be able to implement the custom decorator, I had to make some questions to that py_zipkin contributor so I could understand how I could manipulate their implementation and adapt to my own. After some hard work, I was able to implement

a custom decorator that would automatically detect if a trace has began and if so, it would generate a span that is descendant of that trace instead of creating an entirely new one. If there was no trace created before that request, a new trace would be started from that point forward. It is to note that in the event of a trace has already started and a request is made to another microservice, there is a need to generate some *http headers* and pass some information regarding the parent span for the tracing information to remain correct.

My custom decorator was hard to implement but very simple to use. An example on how to use it is presented in figure 4.6.

```
@emp_zipkin_decorator(service_name='songs_ms', span_name='songs_controller.delete_song', port=5001)
@requires_auth
def delete_song(id):
    """ Deletes an active song given an id"""
```

Figure 4.6: EMP custom decorator usage example

The decorator just needs to be declared above the python function, and some parameters are passed to it. The *service_name* is the name of the microservice that identifies it. The *span_name* is to identify what was the function that was executed and in this case I use the name of the file and the function name. The *port* is the port number on which that microservice is running. It is to note that optional parameters can also be passed to the *emp_custom_decorator* and also some annotations. The *emp_custom_decorator* uses the *py_zipkin* decorator as its core but with some changes that allows for a more complex usage. The instrumentation on this microservices application follows the *OpenTracing* standard. This standard dictates some rules regarding the naming of the annotations that will be present in the trace. It is to note that all applications needs to be instrumented following the *OpenTracing* standard in order for the automatic analysis and scaling of the EMP system to work.

4.7 EMP Detailed Overview

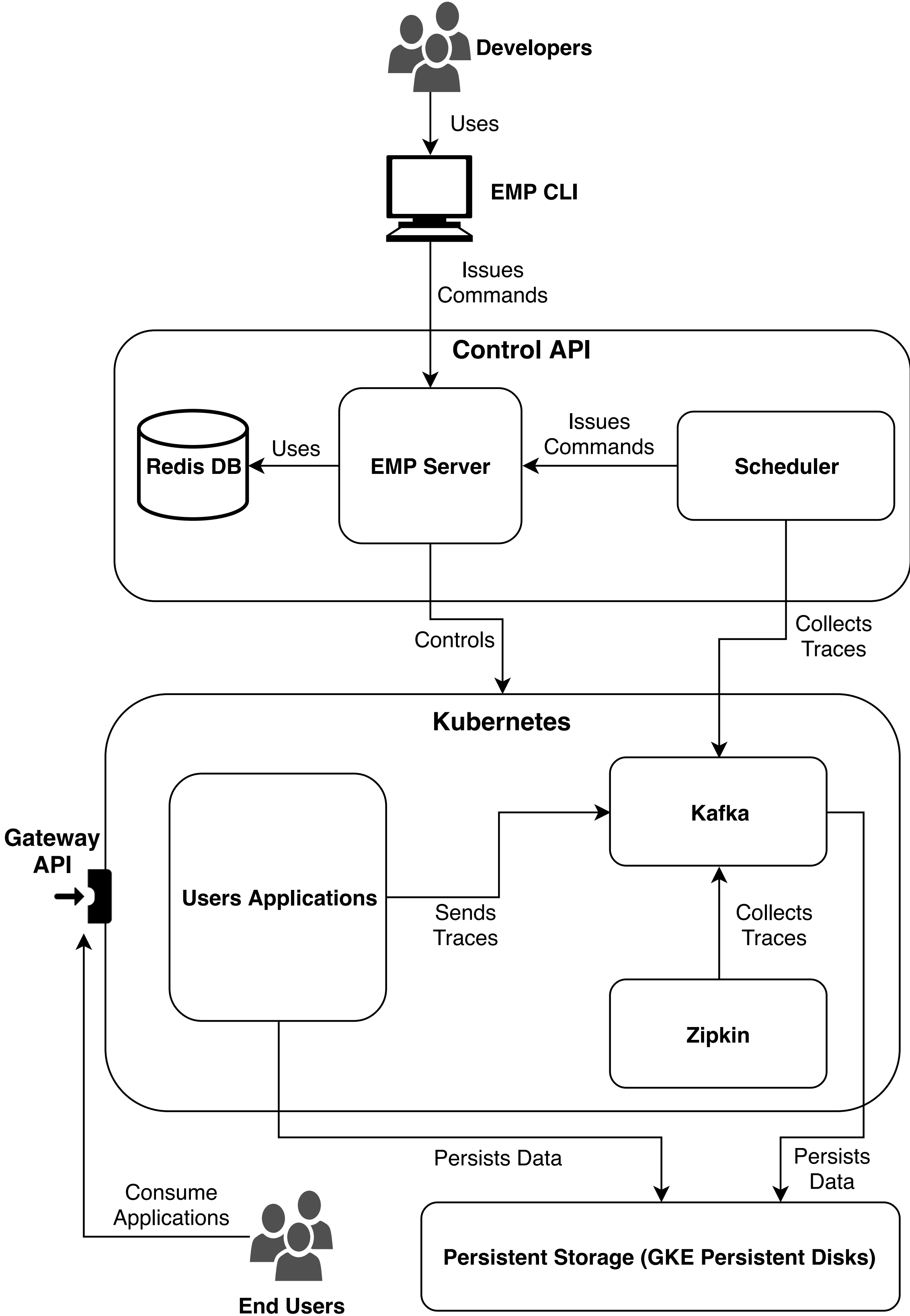

Figure 4.7: EMP Detailed Overview

In figure 4.7, a detailed overview over the entire EMP is presented. The developers will use the EMP CLI to interact with the platform, executing the tasks they desire by communicating with the EMP Server.

The *Control API* handles all the logic part that makes the EMP system work and it is mainly compose by two distinct components. The *Scheduler* will collect traces directly from *Kafka*, that is running inside *Kubernetes*, and will automatically analyze and perform a decision based on them. This decision will then be passed to the EMP Server that will execute the necessary operations on *Kubernetes* to scale up or down a given application and register that change on *Redis* database. The *Redis* database, keeps the information necessary for the EMP to work properly, keeping track of the users applications that are deployed in the platform.

The *Kubernetes* cluster handles the management of all the deployed applications, ensures they stay running by performing health checks and allows end users to consume those applications by its gateway, satisfying the functional requirement REQ-7. Inside the *Kubernetes* cluster, the instrumented user applications will send their traces to *Kafka*, which will store them in the *Persistent Storage*, and then *Zipkin* will consume them to present them in its UI. If the users applications need to store data, they can use the *Persistent Storage* to do so.

In the end, the EMP only misses the *Scheduler* component implementation. Although at its current state the EMP does not automatically analyzes traces and scales the users applications, which is the *Scheduler's* job, everything else is implemented and working as expected. As it was already mentioned, Eng. Jaime Correia will be responsible to implement this *Scheduler* component in the future since it will be very complex and was not the main focus of this thesis.

4.8 EMP Service Requirements Specification

Applications deployed in the EMP need a set of requirements to be elastically and automatically scalable.

Every application a user wants to deploy needs to be instrumented, which means that it has to have a tracing implementation. The requirements that the applications and their tracing implementation must meet are:

- The tracing implementation **must** follow *OpenTracing* standard.
- Every function that is directly connected to a REST request **must** be instrumented. The rest of the application functions instrumentation is optional since the **EMP** is able to elastically and automatically scale the applications without that supplementary information.
- Each span's service name **must** match this criteria: *username/application_name*. For example, if the username of the user is *fcribeiro* and the application that will be deployed is called *songs_ms*, then the span's service name must be *fcribeiro/songs_ms*.
- The application **must** be prepared to receive a environment variable called *KAFKAADDRESS*. This environment variable will contain the *Kafka Address* to which the application must send its spans. The user has the responsibility to ensure the spans will be sent over *Kafka* using the address given from the *KAFKAADDRESS* environment variable.

- The application **must** be containerized.

Chapter 5

Experiments

In this chapter, a detailed description of the tests performed on the EMP is presented. Testing the EMP is required to find and correct some errors that could have gone unnoticed. While testing, it is possible to notice something that could be improved, making the overall platform more robust and polished.

It is to note that during the implementation of every component of the EMP, informal tests were done for validation. After the EMP implementation, I started testing the EMP CLI and the EMP server interactions. The initial tests were made to see how the EMP CLI would handle all the commands that were executed. I tested every command available in the EMP CLI to see if the requests were made correctly to the EMP server and if that requests was carrying the correct information. I also tried to send invalid requests such as trying to deploy an application without specifying its name to see what was the outcome. Since I wrote a *REST API* specification with the object models necessary and generated the python client and server using swagger, both EMP CLI and the EMP server would not allow invalid requests or responses. They already had a parameter check to prevent those invalid requests or responses from executing. The next step was to test the EMP server. Using the EMP CLI, I tested how the EMP server would behave. I started by analyzing if the information that was necessary to store in the *Redis* database was correct. Depending on the command executed, the stored information in the *Redis* database would suffer some changes. To validate the proper execution of such commands, several informal tests were performed and the information stored in the *Redis* database was analyzed. After the EMP CLI and its interactions with the EMP server successfully passed all the tests, I began to test the *Kubernetes* cluster and its interactions with the EMP server.

The *Kubernetes* cluster is a very important component for the EMP and because of that, careful testing was done to validate it and to ensure its correct behavior. I started by deploying an application that was instrumented, into the EMP. Once the application was up and running inside *Kubernetes*, I could now start testing all the available operations for the users to manage their applications. From deploying and application and stopping it, to starting it again and removing it completely from the platform, all operations were working as expected. Informal tests were also made regarding the ambient variable injection that was necessary to perform when an application is deployed inside *Kubernetes*. Users may specify ambient variables they wish their application has and that was also tested. Several tests were made to ensure a deployed application is running inside the *Kubernetes* cluster by making requests to it. This also validates the correct behavior of the operation that exposes the application to the network outside of *Kubernetes*, assigning an IP to it. To see if the effect each operation had in the *Kubernetes* cluster, I used *kubectl* which is a

CLI to access and manage the *Kubernetes* cluster. Inside *Kubernetes*, *Kafka* and *Zipkin* are running to provide the necessary flow the traces from the application need. To test the platforms tracing capabilities, after the instrumented application was deployed, some requests were made to that application in order to generate traces. After the traces were generated from the requests to the application, I accessed *Zipkin's* UI to see if it was able to collect them from *Kafka*. I could see the traces in *Zipkin's* UI and this means that I was able to successfully inject the *KAFKA Address* necessary for the application to send its traces to *Kafka* and that *Zipkin* was able to collect them from it. In order to test the future *Scheduler* component interaction with *Kafka*, I implemented a standalone application that would just collect the traces that were present in *Kafka*. The test was successful as I was able to validate that both *Zipkin* and the future *Scheduler* implementation would be able to collect traces from *Kafka* at the same time. I also tested scaling up and down a deployed application and this operation worked as expected. Finally, I tested the script I wrote to deploy *Kubernetes* in GKE and it passed the tests.

The *Scheduler* component, as it was already mentioned, will be responsible to automatically analyze the applications tracing information and to issue commands to the EMP server to scale a specific application. The EMP will achieve an automatic and elastic scaling capabilities once this *Scheduler* component is implemented. Since this component will be implemented in the future by Eng. Jaime Correia, to simplify its necessary integration with the EMP and to able to test how the *Scheduler* would impact the EMP, a *REST* endpoint was created. This endpoint receives a *REST* request that will carry information regarding a specific application and how many instances of that application must be running. For example, if application A has two instances running in the EMP and the *Scheduler* component decides that it needs five instances to be able to manage its current load, a *REST* request is made to that EMP server endpoint with that information. The EMP server will then retrieve the necessary information from the *Redis* database, and execute the proper commands to the *Kubernetes* cluster to make five available replicas of application A running. Although the *Scheduler* component is not yet implement, since the *REST* endpoint and all the logic operation are, it is possible to simulate and test the *Scheduler* commands. To test this, I deployed an application in the EMP and started to make requests for that *REST* endpoint to scale up the application. I also sent requests to shutdown some instances of that application and all the tests were successful. The *Scheduler* component is not yet implemented but all the logic and operations regarding the EMP are working as expected and ready for its integration.

Finally, the entire EMP system was tested as a whole. I performed several tests that would simulate a real user. I started by deploying an application, consulting its information and trying to access it from the external assigned IP. I tested all the commands once again and I also simulated several *Scheduler* component requests to its specific *REST* endpoint to scale up or down a specific application. Once all the tests passed, it is safe to assume that the EMP is validated and working as expected.

Chapter 6

Conclusion

In this document, the design and implementation of an open source platform for implementing microservices-based systems for deployment in cloud environments was presented. The final architecture of the EMP is very similar to the one that was proposed and presented in section 3.2. This shows that careful planning was done regarding the EMP proposed architecture. This entire platform was conceived and implemented to achieve high modularity. It is possible to swap the components if necessary, according to the users desire. This also makes the integration of the *Scheduler* component that Eng. Jaime Correia will have to do very simple. This platform supports deployment of applications with tracing capabilities and handles all the tracing flow necessary for a future automatic analysis.

With the tests that were made, it is possible to validate that all the commands that are available in the EMP CLI work as expected. It is also possible to prove that once the *Scheduler* component is implemented, the platform will be able to perform the operations required to ensure elastic scalability automatically.

This work shows that it is possible to implement an open source platform that achieves great scaling capabilities. This platform allows users to deploy and manage their applications in a simple way, without the need to manage their own infrastructure. They do not have to manage the resources for their applications, load balancing or scaling. The EMP also provides users with the ability to see detailed information about their applications and a set of useful commands to manage and deploy new ones.

Although it is not the main goal, users will also be able use the EMP for testing purposes. They can deploy and make changes to the EMP system to test a cloud platform implementation since it will be open source. With this open source platform and the way it was designed and implemented, it is easier to replace some components with other ones if the users so desire. This provides users with a platform for testing purposes in cloud environments that can be modified according to their needs.

In the future, a wide variety of improvements on this work can be accomplished. The main component to be implemented in the future is the *Sheduler*. As it was already stated, this work will be used by Eng. Jaime Correia and he will be the one implementing such *Scheduler* component that is responsible for the automatic and elastic scaling capabilities. Another improvement that can be done is to deploy a *Zipkin* server with its own storage component, which the current deployment does not have.

Some new interesting features can be added, providing a richer and better user experience. Some of those new features could be:

- Add auto complete features to the EMP CLI for ease of use. This feature could really improve the user experience making the CLI feel more polished and smoother.
- Fulfill functional requirement REQ-6 that allows users to declare the resources each container of their application must have (CPU and memory). This would provide users with a greater control over their application resources allocation.
- Select and decide new interesting information or statistics regarding the users applications to present them.
- Allow the user to set maximum and minimum limits regarding the number of instances that can be running at the same time of a specific application.
- A dashboard could be implemented to show important information and statistics about the applications to the users. When a user deploys an application, the EMP will scale it whenever it is necessary, so instead of consulting its details and statistics using the CLI, a dashboard would be an interesting choice. This would certainly be appealing for users to be able to view important information and statistics regarding their deployed applications in the EMP in a dashboard.

In the end, an open source platform for implementing microservices-based systems for deployment in cloud environments was designed, implemented, tested and validated. In its current state, the EMP is able to achieve great scaling capabilities, provide users with several management options, present important information regarding their applications and it is simple to use. The objectives that were proposed were met and this platform is ready to be used by Eng. Jaime Correia for his future work. Once he develops the *Scheduler* component and integrates it with the EMP, an automatic analysis over the applications and an elastic scalability will be achieved.

References

[1] Amazon elastic compute cloud features explained.
https://searchaws.techtarget.com/feature/Amazon-Elastic-Compute-Cloud-features-explained. Accessed: 07/02/2018.

[2] Amazon elastic container service.
https://aws.amazon.com/ecs/?nc1=h_ls. Accessed: 09/02/2018.

[3] Click documentation.
http://click.pocoo.org/5/. Accessed: 15/02/2018.

[4] Docker overview.
https://docs.docker.com/engine/docker-overview/. Accessed: 07/10/2017.

[5] Elastic beanstalk vs. ecs vs. kubernetes.
https://fortyft.com/posts/elastic-beanstalk-vs-ecs-vs-kubernetes/. Accessed: 09/02/2018.

[6] Flask documentation.
http://flask.pocoo.org/. Accessed: 02/10/2017.

[7] Flask-jwt documentation.
https://pythonhosted.org/Flask-JWT/. Accessed: 02/10/2017.

[8] Flask-login documentation.
https://flask-login.readthedocs.io/en/latest/. Accessed: 02/10/2017.

[9] Google compute engine persistent disk documentation.
https://cloud.google.com/compute/docs/disks/. Accessed: 14/08/2018.

[10] Helm documentation.
https://docs.helm.sh/. Accessed: 20/02/2018.

[11] Kubernetes python client documentation.
https://github.com/kubernetes-client/python/tree/master/kubernetes. Accessed: 10/04/2018.

[12] Mariadb documentation.
https://mariadb.com/kb/en/library/documentation/. Accessed: 02/10/2017.

[13] Metallb documentation.
https://metallb.universe.tf/. Accessed: 05/04/2018.

[14] Nginx web page.
https://www.nginx.com/. Accessed: 13/10/2017.

[15] Openapi specification.
https://swagger.io/specification/. Accessed: 02/10/2017.

[16] Overview of docker compose.
https://docs.docker.com/compose/overview/. Accessed: 07/10/2017.

[17] Pyjwt documentation.
https://pypi.python.org/pypi/PyJWT/1.4.0. Accessed: 02/10/2017.

[18] Py_zipkin github page.
https://github.com/Yelp/py_zipkin. Accessed: 20/04/2018.

[19] React documentation.
https://reactjs.org/docs/getting-started.html. Accessed: 02/10/2017.

[20] Redis documentation.
https://redis.io/documentation. Accessed: 12/03/2018.

[21] Software design - scalability (scale up—out).
https://gerardnico.com/wiki/code/design/scalability. Accessed: 03/10/2017.

[22] Sqlalchemy documentation.
http://docs.sqlalchemy.org/en/latest/. Accessed: 02/10/2017.

[23] Swagger editor.
https://editor.swagger.io/. Accessed: 02/10/2017.

[24] Swarm mode overview.
https://docs.docker.com/engine/swarm/. Accessed: 07/10/2017.

[25] Under the hood of amazon ec2 container service.
https://www.allthingsdistributed.com/2015/07/under-the-hood-of-the-amazon-ec2-container-service.html. Accessed: 07/02/2018.

[26] What is amazon ec2?
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed: 07/02/2018.

[27] What is amazon ec2 auto scaling?
https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html. Accessed: 07/02/2018.

[28] What is aws elastic beanstalk?
https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html. Accessed: 07/02/2018.

[29] What is kubernetes.
https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed: 13/10/2017.

[30] Simon Brown. C4 model poster.
http://www.codingthearchitecture.com/2014/08/24/c4_model_poster.html, August 2014. Accessed: 27/10/2017.

[31] Simon Brown. *Software Architecture for Developers - Volume 2, Visualise, document and explore your software architecture*. Ebook, 2017.

[32] Preethi Kasireddy. A beginner-friendly introduction to containers, vms and docker.
https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b, March 2016. Accessed: 07/10/2017.

[33] Esther Levine. What's the difference between elasticity and scalability in cloud computing.
https://www.stratoscale.com/blog/cloud/difference-between-elasticity-and-scalability-in-cloud-computing/. Accessed: 03/10/2017.

[34] James Lewis Martin Fowler. Microservices.
https://martinfowler.com/articles/microservices.html, March 2014. Accessed: 27/09/2017.

[35] Janakiram MSV. Kubernetes: An overview.
https://thenewstack.io/kubernetes-an-overview/, November 2016. Accessed: 15/10/2017.

[36] Sam Newman. *Building Microservices*. 2015.

[37] Ralf Reussner Nikolas Roman Herbst, Samuel Kounev. Elasticity in cloud computing: What it is, and what it is not.
https://sdqweb.ipd.kit.edu/publications/pdfs/HeKoRe2013-ICAC-Elasticity.pdf. Accessed: 03/10/2017.

[38] Chris Richardson. Introduction to microservices.
https://www.nginx.com/blog/introduction-to-microservices/, May 2015. Accessed: 27/09/2017.

[39] Wojciech Tyczynski. Scalability updates in kubernetes 1.6: 5,000 node and 150,000 pod clusters.
http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html, March 2017. Accessed:13/10/2017.