UNIVERSIDADE Đ
COIMBRA

Diogo Valente Martins

Scaling of Applications in Containers

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Filipe Araújo and Pedro Neves from Altice Labs, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Diogo Valente Martins

Scaling of Applications in Containers

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Filipe Araújo and Pedro Neves from Altice Labs, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

Acknowledgements

I would like to thank the supervisors, Pedro Neves and Paulo Sousa, for their support, advice, and encouragement during this internship. Likewise, I also would like to thank Professor Filipe Araújo for the support, advice and feedback given during the internship. I also would like to thank the Altice Labs and ACM team for being very supportive and communicative by providing feedback and overall professionalism. As a final note, I want to thank my family, friends and girlfriend for supporting me during less than happy times.

Abstract

With advances in the Internet and telecommunications, the amount of data that needs to be processed is increasing, making it difficult to estimate the hardware requirements needed to process them over time. Additionally, in a traditional software installation, the required hardware is always estimated for the worst case, which means that a large part of the time is underused. One of the ways to reduce costs is to install these applications in the cloud allocating the minimum of required resources or to share the hardware with other platforms that have processing requirements at different times. For this to be possible, it is necessary to have a mechanism to predict the resources needed at a given time and that the software be prepared to be activated or deactivated using automatic mechanisms, that is, that the software is prepared to be rescaled. The Active Campaign Manager (ACM) platform receives millions of events per day in real time from various sources (eg call logs, prepaid topups, geolocation events, etc) and triggers actions (proposal of offers, activation/ deactivation of offers, etc) according to pre-defined rules. This platform is currently being prepared to support these mechanisms in order to be able to be instantiated with lower infrastructure costs. At the moment the platform is instantiated using containers, however, it does not yet support instantiation in Kubernetes. This curricular internship aims to make a first deployment of the platform in Kubernetes, identify which platform components can be instantiated multiple times, identify which metrics are important for the decision to increase/decrease the number of instances of each component, implement and test a concept proof.

Keywords

Kubernetes. Containers. Scaling. Costs.

Resumo

Com os avanços na Internet e nas telecomunicações, a quantidade de dados que precisam de ser processados está a aumentar, tornando difícil estimar os requisitos de *hardware* necessários para os processar ao longo do tempo. Além disso, numa instalação tradicional de software, o hardware necessário é sempre estimado para o pior caso, o que significa que uma grande parte do tempo é subutilizada. Uma das formas de reduzir os custos é instalar estas aplicações na *cloud*, atribuindo o mínimo de recursos necessários ou partilhando o hardware com outras plataformas que tenham necessidades de processamento em momentos diferentes. Para que isto seja possível, é necessário que exista um mecanismo de previsão dos recursos necessários num determinado momento e que o *software* esteja preparado para ser ativado ou desativado através de mecanismos automáticos, ou seja, que o software esteja preparado para ser redimensionado. A plataforma ACM recebe milhões de eventos por dia em tempo real de várias fontes (ex.: registos de chamadas, carregamentos pré-pagos, eventos de geolocalização, etc.) e desencadeia acções (proposta de ofertas, ativação/desativação de ofertas, etc.) de acordo com regras pré-definidas. Esta plataforma está atualmente a ser preparada para suportar estes mecanismos, de modo a poder ser instanciada com custos de infraestrutura mais baixos. De momento a plataforma é instanciada utilizando containers, no entanto, ainda não suporta a instanciação em Kubernetes. Este estágio curricular tem como objetivo fazer um primeiro *deployment* da plataforma em Kubernetes, identificar que componentes da plataforma podem ser instanciados múltiplas vezes, identificar que métricas são importantes para a decisão de aumentar/diminuir o número de instâncias de cada componente, implementar e testar uma prova de conceito.

Palavras-Chave

Kubernetes. Containers. Scaling. Custos.

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives . . . 2

2 Planning **3**
- 2.1 First Semester . . . 3
- 2.2 Second Semester . . . 4
- 2.3 Risk analysis . . . 4
  - 2.3.1 Threshold of Success . . . 4
  - 2.3.2 Risk Analysis . . . 5

3 State of Art **7**
- 3.1 ACM Platform . . . 7
- 3.2 Concepts . . . 8
  - 3.2.1 Containers . . . 8
  - 3.2.2 Docker . . . 9
  - 3.2.3 Kubernetes . . . 10
  - 3.2.4 Kubernetes Controllers . . . 13
- 3.3 Kubernetes Autoscaling . . . 13
  - 3.3.1 Horizontal Pod Autoscaling . . . 13
  - 3.3.2 Vertical Pod Autoscaling . . . 14
  - 3.3.3 Cluster Autoscaling . . . 15
- 3.4 Kafka . . . 16
- 3.5 Prometheus . . . 16
- 3.6 KEDA . . . 17
- 3.7 Final Remarks . . . 18

4 Architectural Drivers **21**
- 4.1 Current System . . . 21
  - 4.1.1 Overall Architecture . . . 21
- 4.2 System Functionality . . . 22
- 4.3 Quality Attributes . . . 25
- 4.4 Restrictions . . . 26
  - 4.4.1 Business Restrictions . . . 26
  - 4.4.2 Technical Restrictions . . . 26

5 Architecture **27**
- 5.1 C4 Model . . . 27

5.1.1 System Context Diagram 27
5.1.2 Container Diagram 28
5.1.3 Components Diagram 29
5.1.4 Code 31
5.2 Analysis 31

6 Implementation **33**
6.1 Approach 33
6.1.1 Integrated Development Environment 33
6.1.2 Deployment 33
6.2 Development 34
6.2.1 Zookeeper 34
6.2.2 Kafka 35
6.2.3 Kakfa Exporter 36
6.2.4 Prometheus 37
6.2.5 Grafana 38
6.2.6 Event Stream Processor 39
6.2.7 Horizontal Pod Autoscaler 41

7 Testing **43**
7.1 Manual functional test 43
7.1.1 Manual functional test results 46
7.2 Quality attribute verification 46
7.3 Scalability Tests 47

8 Cloud Costs **49**
8.1 Server-based Solution 49
8.2 Serverless Solution 49
8.3 Cost analysis 50
8.3.1 On-Premises System 50
8.3.2 Serverless system 51
8.3.3 Virtual Machines 53
8.3.4 Messaging Costs 53
8.4 Results 53

9 Conclusion **55**
9.1 Planned vs Real schedule 55
9.2 Difficulties 56
9.2.1 Deployment of the applications in Kubernetes 56
9.2.2 Scaling of the Components 56
9.3 Study of cloud costs 57
9.4 Future Work 57
9.4.1 Documentation 57
9.4.2 Deployment Scaling of the Engine 57
9.5 Final Thoughts 58

Appendix A Requirements not implemented **65**

Acronyms

ACM Active Campaign Manager

API Application Programming Interface

AWS Amazon Web Services

CA Cluster Autoscaler

CISUC Centre for Informatics and Systems of the University of Coimbra

CNCF Cloud Native Computing Foundation

GiB GibiBytes

GiBs GibiBytes second

HPA Horizontal Pod Autoscaler

KEDA Kubernetes-based Event Driven Autoscaler

MiB MibiBytes

MiBs MibiBytes second

ms miliseconds

POD Plain Old Datastructure

USD United States Dollar

VPA Vertical Pod Autoscaler

List of Figures

2.1 Project Schedule for the first semester 3
2.2 Project Schedule for the second semester 4

3.1 ACM Platform 8
3.2 Virtual Machines VS Containers [6] 9
3.3 Docker Architecture [8] 10
3.4 Kubernetes Cluster[12] 12
3.5 Vertical Pod Autoscaler (VPA) architecture [16] 15
3.6 Prometheus architecture [20] 17
3.7 Kubernetes-based Event Driven Autoscaler (KEDA) architecture [21] 18

4.1 Diagram of the current Active Campaign Manager (ACM) platform 22

5.1 Software Architecture: Context Level 28
5.2 Software Architecture: Context Level Legend 28
5.3 Software Architecture: Container Diagram 29
5.4 Software Architecture: Container Diagram Legend 29
5.5 Software Architecture: Components Diagram 30
5.6 Software Architecture: Components Diagram Legend 30

6.1 ConfigMap Prometheus 38
6.2 ConfigMap Grafana 39
6.3 Volumes of Event Stream Processor 41
6.4 Horizontal Pod Autoscaler (HPA) Event Stream Processor 42

7.1 Lag by consumer group for the different scenarios 48

9.1 Original Schedule of the second semester 55
9.2 Real Schedule of the second semester 55

List of Tables

4.1 REQ1 - Deploy the Zookeeper 23
4.2 REQ2 - Deploy Kafka 23
4.3 REQ3 - Deploy Kafka Exporter 23
4.4 REQ4 - Deploy Prometheus 24
4.5 REQ5 - Deploy Grafana 24
4.6 REQ6 - Deploy Event Stream Processor 24
4.7 REQ7 - HPA for scaling down 25
4.8 REQ8 - HPA for scaling up 25

7.1 Results for the test cases of REQ2 43
7.2 Results for the test cases of REQ3 44
7.3 Results for the tests cases of REQ4 45
7.4 Results for the tests cases of REQ5 45
7.5 Results for the tests cases of REQ6 46

8.1 Server running the service. 49
8.2 Metrics regarding system load. 50
8.3 Resources necessary per core. 50
8.4 Data used to compute the Lambda execution costs. 51
8.5 Data used to compute the Step Functions execution costs. Amazon Web Services (AWS) charges a minimum of 100 ms per workflow. . . 52

A.1 REQ9 - Scaling up the Kafka Topics Partitions 65
A.2 REQ10 - Scaling down the Kafka Topic partitions 65

Chapter 1

Introduction

This report is a component of the curricular internship in Software Engineering specialization of the Master in Informatics Engineering from the University of Coimbra which was carried out in conjunction with Altice Labs in the curricular year of 2022/2023.

1.1 Context

Altice Labs proposed that the curricular internship is integrated into the goals of the Big Data & Data Monetization department, specifically, in the line of activity dedicated to clouding applications. This proposal is also in line with the objectives of the group LCT of the Centre for Informatics and Systems of the University of Coimbra (CISUC) and aims to take advantage of the existing synergies between these two institutions.

The company's proposed goal is for the student to implement an automatic way to deploy and scale applications in containers by using Kubernetes technology (used technology for container orchestration). This will allow application components to be deployed and, according to the load requested at each moment, automatically increase or decrease the number of active containers in real time.

1.2 Motivation

Nowadays, the growth of technologies and different platforms is enormous, and developing applications that can fit in multiple environments is not easy. Containers can be used as a base technology for deploying large scale applications.

In a container environment, installing an application and all the dependencies necessary to run the application is possible. In addition, containers make running, stopping, releasing, and upgrading new versions effortlessly. Currently, the ACM platform is running in virtual machines, which, compared with containers, introduce more overhead and are heavier [1]. In addition, containers can

resolve problems such as performance and portability [1] also deploy large-scale big data applications that are required to manage many components. The ACM platform is on-premises. By running the applications in containers in the future, it is possible to change from on-premises to a cloud environment.

The containers are not enough for this project despite being an important actor. With the evolution of the internet and telecommunications, the amount of data that needs to be processed is enormous. With the increase of data to be processed, the need for computing resources and costs also increases. So it is necessary to use a container orchestrator, such as Kubernetes. This framework has auto-scaling mechanisms that use metrics to compute the resources needed and activate the scale actions.

For example, the ACM platform has all instances instantiated and receives millions of events per second, which is a waste of resources. However, with the scaling mechanisms present in Kubernetes, it is possible to optimize the computed resources being consumed and reduce the costs associated with this operating environment.

The motivation for this project is to have a portable application that can be deployed in different environments without changing the source code and also automate the application's scaling actions according to the application's load in real time.

1.3 Objectives

The main scope of this internship is to improve the ACM platform already built by Altice Labs. The initial goal is to deploy the application in Kubernetes, and this is the first big step of the implementation. After the application runs in Kubernetes, some tests should be performed to test if the behavior is as before so it is possible to move to the following objectives.

First, the system has components with a fixed number of instances. The next goal is implementing auto-scaling mechanisms on these components, which will be possible through auto-scaling Kubernetes mechanisms.

Before the implementation phase, it is necessary to research the required technologies, for instance, Kubernetes, and it is necessary to define the requirements. Following this is the responsibility of the intern to develop and design the new architecture.

To resume, the intern's objectives are to study the technologies needed to implement, define and design the architecture that allows the system to have autoscaling mechanisms. After the most theoretical part has been done, the intern should implement the solution designed previously that will allow the current system to increase or decrease instances according to the load in real-time.

Chapter 2

Planning

2.1 First Semester

This chapter will describe the internship schedule and the risks that may occur during the internship. The internship is divided into two semesters with entirely different proposes. The Gantt Project [2] was used to help with the visualization of the time and tasks.

The internship supervisor performed the plan for the first semester. It started with the presentation of the Active Campaign Manager (ACM) platform. Following the presentation, research was conducted on the state-of-the-art and required technologies. During this research, a proof of concept was implemented. After the research, the system requirements and architecture were defined. Finally, the last month was dedicated to writing the intermediate report. This schedule is represented in Figure 2.1 as a Gantt chart.

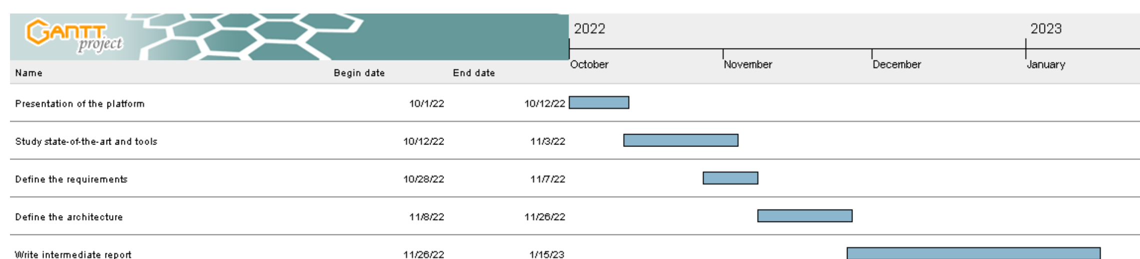

Figure 2.1: Project Schedule for the first semester

The life cycle adopted was an agile methodology. Every week a meeting was held with the project team, supervisors from Altice Labs, and the Professor from the university. In each session, it was discussed what had been done in the past week and what should be done in the next week. This is similar to an agile methodology called scrum [3], where the work is planned in iterations called sprints.

2.2 Second Semester

Following the intermediate defense, the plan is to set up everything necessary and start the implementation. However, first, deploying the applications in a Kubernetes Cluster is necessary, then scaling the applications should be automated. Then, after the implementation, the application must be tested to verify if it complies with the requirements. Finally, the last month is dedicated to writing the final report.

The Gantt chart in Figure 2.2 represents the proposed schedule.

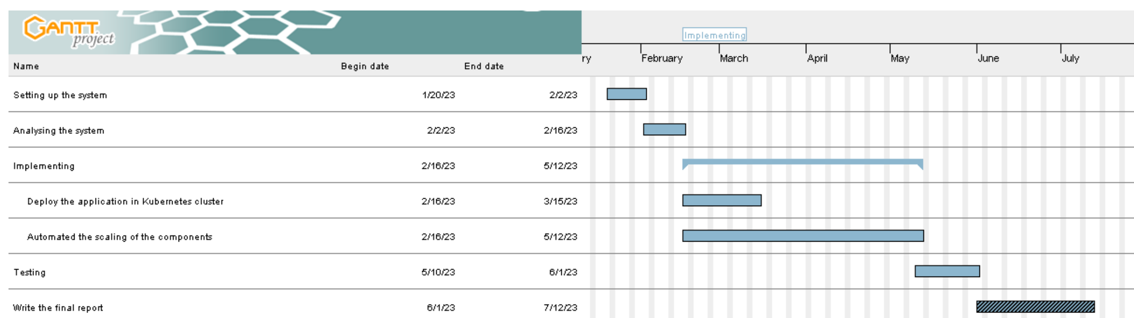

Figure 2.2: Project Schedule for the second semester

2.3 Risk analysis

This section describes risks that could compromise the success of this curricular internship. Additionally, the threshold of success the curricular internship must meet is defined.

2.3.1 Threshold of Success

For the success of the curricular internship, the following points have to be fulfilled:

- The development of the system should not exceed the time of the curricular internship.
- The system must be able to process an event as in the original application.
- The system must activate the scaling mechanisms if the workload exceeds the defined limit.
- The system functionalities specified in this document should be developed, tested, and ready to deploy.

2.3.2 Risk Analysis

The potential risks found in the first semester were:

- **Delays in access to data.** Due to the necessity of having metrics of external systems for implementing the scaling, it is expected that the Altice ALbs will provide these metrics. Therefore, the implementation will be delayed if these metrics are not available. One way to mitigate this is implementing using only metrics from the applications and then upgrading the system if other metrics are available.

- **Change in requirements.** The implementation is an entirely new technology for this platform in Altice Labs. Therefore, as we will have meetings and the project state is discussed, there may be changes in the requirements. These changes will undoubtedly create delays in the schedule. One solution to mitigate this problem is to present prototypes more frequently to get more feedback.

Chapter 3

State of Art

This chapter will explain the study performed on the relevant technologies and concepts. It starts with 3.1, which presents the current Active Campaign Manager (ACM) platform that needs improvement. After the application has been explained, section 3.2 will introduce the basic concept of Docker and Container, followed by Kubernetes, which are necessary for improving the platform. In the final section, we study how Kubernetes can implement auto-scaling, and the technologies used to fit our use case better will be presented.

3.1 ACM Platform

The ACM platform is a software developed by Altice Labs, that allows to monitor, run, and optimize multichannel marketing campaigns. With this, companies can design and release real-time promotions that impact their customers and contribute to increasing business revenue.

This platform is an on-premise software that receives a massive number of events per second from different sources. For example, it can receive events synchronously or asynchronously, and the type of source events received can be HTTP requests via an interface Rest, through files, databases, or other types of sources.

Most events are asynchronous and written in a Kafka topic, and these are called raw events that Event Processors will consume. After the events have been processed, they will be written again on a Kafka topic, and this time they will be called ready events. After these, the engines will consume these ready events and write them in another Kafka topic so that the actions are executed as, for example, sending an SMS with a campaign promotion.

The system has instantiated a fixed number of topics, Events processors, and engines. The problem is that the allocated instances are using the resources, always counting on the worst case scenario, which is at Christmas time. During the year, only 30% to 40% of the resources are being used. In the future, the application should be able to scale some of these instances up and down according to the real-time load so the free resources can be used to run internal Altice Labs appli-

cations, for instance.

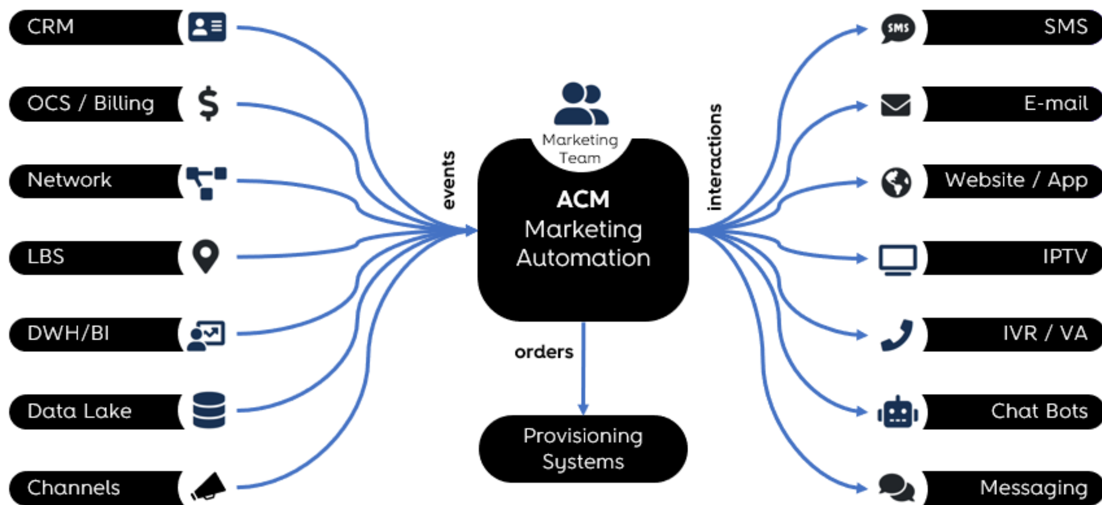

Figure 3.1: ACM Platform

Figure 3.1 is a high-level representation of the ACM platform and how it works. Visualizing all types of events received that are processed internally by the platform is possible in Figure 3.1. After being processed the events, they can generate multiple interactions with the final customer, as shown in Figure 3.1.

3.2 Concepts

This section describes concepts related to the curricular internship, for instance, what a container is, and this is mandatory to know to develop a solution for this problem of autoscaling applications in containers using Kubernetes as an orchestrator.

3.2.1 Containers

There is an evolution in how the applications were deployed, which can be divided into three phases. In the beginning, the applications were deployed on physical servers. However, the deployed on physical servers have many problems with resource allocation, and at the same time, it is expensive. So the second phase to solve this was to deploy applications in virtual machines.

In the physical server, the only way to have two applications running without having problems with the resources was to have different servers for each application, which was very expensive. The second phase resolved this problem, allowing multiple virtual machines to run on the same server.

The third is the phase of containers which are lightweight packages of an application that include all components required to run in any environment. Containers

virtualize the operating system this way, allowing them to run anywhere[4] while the virtual machines also virtualize the hardware[5]

There are several benefits related to containers. Firstly they consume fewer resources than an operating system based on hardware virtualization[4] and also provide application isolation because containers virtualize CPU, memory, storage, and network resources at the operating system level and workload portability, that is, the ability to run anywhere and make it easy to develop and deploy.

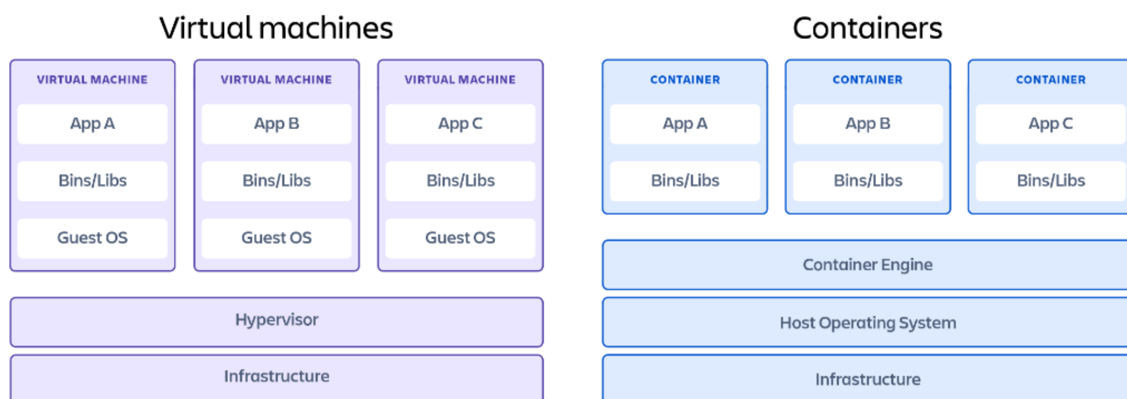

Figure 3.2: Virtual Machines VS Containers [6]

3.2.2 Docker

The most well-liked and user-friendly platform for managing and operating containers is Docker. The fact that the great majority of containerized tools are packaged as Docker images serves as evidence for this[7]. Docker enables an easy way to deploy, build, run and update containers, which allows the applications to run in any environment.

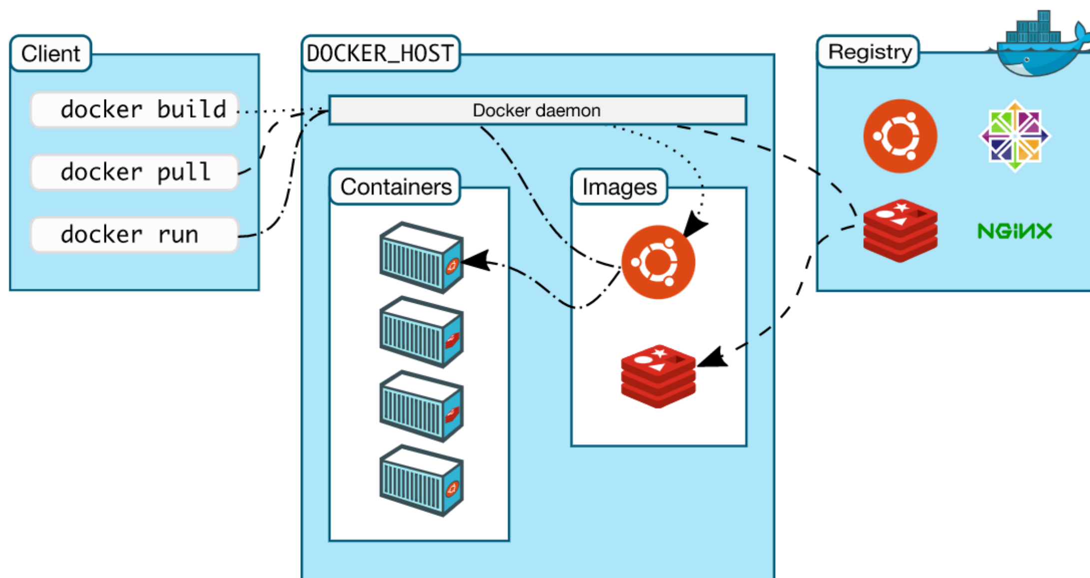

Figure 3.3: Docker Architecture [8]

Figure 3.3 represents Docker's architecture and its components. The **docker daemon** manages docker objects like images, containers, volumes and can communicate with other daemons. The **client** part in the architecture allows a user to interact with Docker through commands using the Docker Application Programming Interface (API). This API is the one used by daemons to manage the requests. Finally, the images are kept in the **registry**. Anyone can utilize Docker Hub[9], a public registry, and by default, Docker is set up to search there for images. Even running your private register is an option.

The **images** are templates with the source code, libraries, and dependencies necessary to deploy the applications. These **images** are used to create containers.

3.2.3 Kubernetes

As referred to in the previous section, deploying and running the applications in containers have advantages. However, in a production environment, it is necessary to ensure that the applications do not have downtime. Kubernetes is an open-source platform for automating containerized applications' deployment, scaling, and management. Google originally developed it, and it is now maintained by Cloud Native Computing Foundation (CNCF).

Kubernetes provides a unified API and tools for managing containerized applications across multiple hosts and environments. It abstracts away the underlying infrastructure and provides a declarative way of defining the desired state of the application, which Kubernetes then reconciles with the actual state to ensure that the application is always running as expected.

Kubernetes is also a key component of the cloud-native ecosystem, which is focused on building and deploying applications designed to be scalable, resilient, and easily managed in dynamic, cloud-based environments.x

For example, these are features presented in the Kubernetes documentation[10]:

- **Service discover**: Kubernetes give each container an IP address and a DNS name, so exposing a container to the exterior is possible.
- **Load balacing**: Kubernetes can distribute the load of a container and so stabilize the deployments.
- **Storage orchestration**: It is possible to use Kubernetes to automatically mount any storage system, including local drives, public cloud services, and more.
- **Automated rollouts and rollbacks**: Using Kubernetes, it is possible to specify the desired state for your deployed containers, and it will gradually convert the current state to the desired state. For instance, Kubernetes can be automated to generate new containers for deployment, remove existing containers, and adopt all the resources from the old containers to the new ones.
- **Automatic bin packing**: The user has to determine how much CPU and memory is necessary for each container, and Kubernetes will automatically fit the containers to make the best use of the resources.
- **Self-healing**: Kubernetes can kill, restart and replace if there are problems with the containers and do not allow the use of containers that are not ready.
- **Secret and configuration management**: It is possible to store sensitive information as passwords without being revealed.

After this explication of the features of Kubernetes, it is important to understand the architecture of a Kubernetes cluster and each component of the architecture to comprehend how Kubernetes accomplish the features previously described.

Kubernetes operates with a cluster of distributed nodes that communicate with one another to function as a single unit. This service allows for the deployment, management, and scale of applications in containers efficiently.

A Kubernetes cluster is composed of a master node and one or more worker nodes.[11] The master node is responsible for making the decisions on the cluster, such as scheduling, scaling up and down, and launching new updates of an application. The communication between the master and the worker is managed by a process named kubelet.

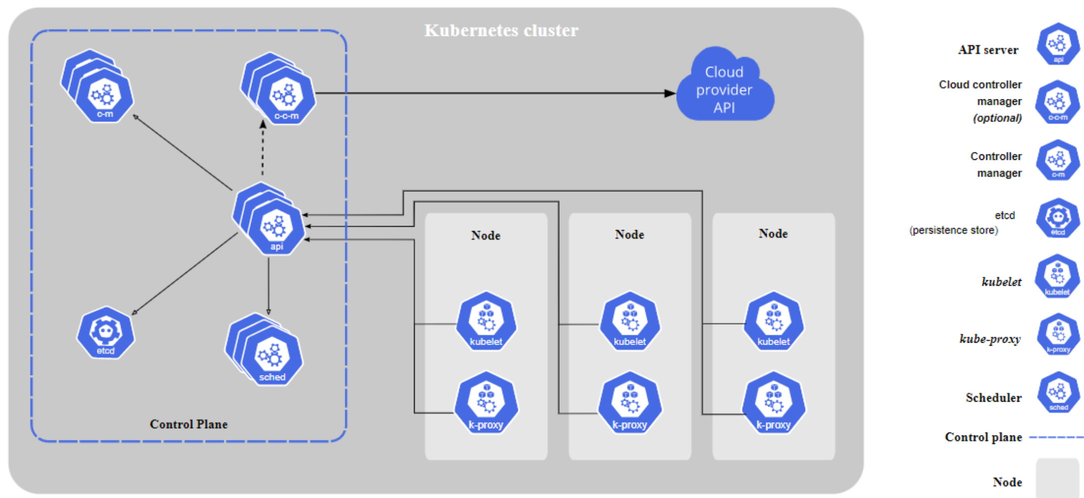

Figure 3.4: Kubernetes Cluster[12]

Starting by enumerating and explaining the master node components:

- **kube-apiserver** is the foundational management component that can communicate with all other components, and any change to the cluster state must go through it.
- **etcd** it is the backup storage for the cluster's configuration data.
- **kube-scheduler** Watches for newly created Pods with no assigned node and selects a node for them to run. Individual and collective resources requirements like hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines are all considered for the scheduling.
- **kube-controller-manager** is the process that ensures that the cluster runs in the correct state. For instance, if a pod fails kube-controller-manager must ensure that a new replica is created.

The several worker nodes that are part of a cluster and controlled by the master node have the following components:

- **kubelet** is a process that runs inside the node and operates the pods through different mechanisms that are provided and ensures that the pods are kept alive and healthy.
- **Kube-proxy** is a network proxy that runs inside the cluster and supports network rules, allowing external and internal communication via a unique IP assigned to each pod of the cluster.
- **Container runtime** is the software responsible for running the containers. Kubernetes supports container run times as Docker, CRI-O and many others. The most common is Docker[13], which allows the automatic deployment of applications in containers.

- **Container Resource Monitoring** is a mechanism that provides metrics data from the local host or the containers about resource usage. This data can be used for monitoring proposes by exporting the data to external tools such as Prometheus[14].

3.2.4 Kubernetes Controllers

In the context of Kubernetes, Plain Old Datastructure (POD) is the smallest deployable unit that the Kubernetes ecosystem can manage. A POD in Kubernetes is a logical host for one or more containers, providing an environment for running those containers. It is designed to be ephemeral, meaning they can be created, destroyed, and replaced at any time. Each POD in Kubernetes has a unique IP address, and the containers within the POD share the same network namespace, allowing them to communicate using localhost.

PODs in Kubernetes can also be managed by Kubernetes controllers, such as Deployments or ReplicaSets, which provide additional features for managing the lifecycle of the containers running within the POD. In summary, a POD in Kubernetes is a fundamental unit of deployment that encapsulates one or more containers and provides an environment for running those containers, which can be managed and controlled by Kubernetes.

3.3 Kubernetes Autoscaling

Autoscaling is an application's ability to scale up and down when the load of the application increases without human intervention. A feature of Kubernetes is that it allows applications in containers to autoscale and run without failures and human intervention. This open-source platform provides three types of scalers: Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA).

3.3.1 Horizontal Pod Autoscaling

HPA is a feature of Kubernetes that, when it is necessary to increase the resources due to an increased load of an application, automatically will raise the number of pods and thus increase the computational power without the need to stop the application. Moreover, once they are ready, the newly launched pods can share the load with those already functioning. The HPA is implemented by Kube-control-manager, which by a default period of 15 seconds, collects metrics and compares these metrics with the thresholds defined in the configurations.

The following equation determines the appropriate number of pods. Once the metric's value reaches the threshold, the HPA will automatically increase the number of pods according to the equation.

$$numPods = [currentPods * \frac{currentMetricValue}{desiredMetricValue}] \quad (3.1)$$

In the equation 3.1 presented before, **CurrentPods** is the pods that are running at the moment, **NumPods** is the number of pods desired after scaling, **currentMetricValue** is the collected and **desiredMetricValue** is the target threshold defined in the configuration file.

3.3.2 Vertical Pod Autoscaling

VPA is another feature of Kubernetes for autoscaling. However, instead of raising the number of pods like HPA, the VPA scales the CPU and memory resources allocated to a Pod based on its actual usage.

To use VPA, first, we need to enable the feature in our Kubernetes cluster and then annotate the deployment with the desired minimum and maximum resource limits. The VPA controller will then monitor the resource usage of the deployment and adjust the allocated resources as needed to maintain a balanced utilization.

The VPA consists of three main components [15]. The first one is the **Recommender** responsible for monitoring the metrics by requesting the Metrics API. The VPA is also responsible for providing the CPU and memory recommendations to each container. These recommendations have a lower and upper bound, along with target values.

The second component is the **Updater**. This checks if the pod has the correct resources set. If this is not true, the component will shut down the pod so the controllers can create them again with the new resources request set.

The third and final component is the **AdmissionController** responsible for setting the correct resource requests on new Pods.

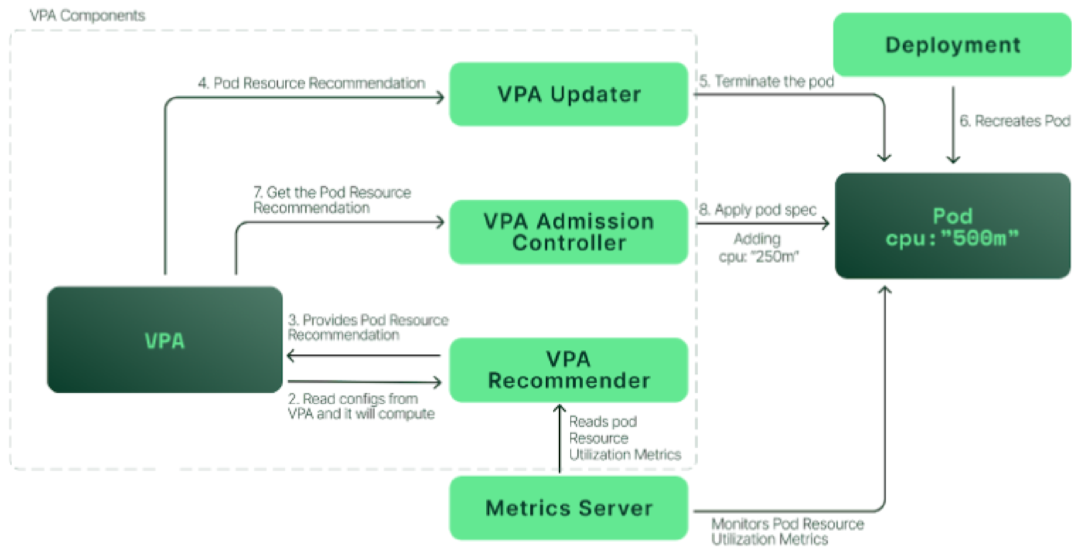

Figure 3.5: VPA architecture [16]

In Figure 3.5, it is possible to visualize how the three main components described previously interact with each other. Also, it is possible to confirm that in a VPA architecture, the **Recommender**, **Updater**, and **AdmissionController** are the main three components.

3.3.3 Cluster Autoscaling

The CA differs from the two types of auto scalers described previously, while the other two have implications at a pod level. This only removes and add nodes in a cluster based on resource request from the pods[17]. The CA checks every ten seconds by default if any pod is pending and assumes that assigning them to a node is impossible due to insufficient cluster capacity.

The functioning of CA can be resumed in four steps. First, the autoscaler will check for pending pods. As mentioned before, these checks occur every ten seconds by default and can be configurable. The second step occurs if there are pending nodes. If this condition is verified, the cluster needs more resources, CA will launch new nodes. After the new nodes are launched, they must be registered in the control panel to make it available to the Kubernetes scheduler for assigning new pods. Finally, the Kubernetes scheduler allocates the pending pods to the new node.

However CA have some limitations that can impact our project. For instance, CA does not make scaling decisions using CPU or memory usage. Another problem is the time that CA takes to scale up the cluster. This delay can mean the application performance may be worst while waiting for the extended cluster capacity.

3.4 Kafka

Apache Kafka [18] is an open-source distributed streaming platform developed by the Apache Software Foundation. It is designed to handle high-volume, real-time data streams and provide a scalable, fault-tolerant, and reliable infrastructure for stream processing.

At its core, Kafka is a distributed publish-subscribe messaging system that allows producers to publish messages to topics and consumers to subscribe to those topics and process the messages in real-time. The messages are organized into topics, divided into partitions, and replicated across a cluster of servers called brokers.

The following definitions are key concepts in Kafka:

- **Producers:** Applications that publish messages to Kafka topics.
- **Consumers:** Applications that subscribe to Kafka topics and process the messages.
- **Topics:** Categories or streams of messages in Kafka. Producers publish messages to specific topics, and consumers subscribe to topics to receive and process the messages.
- **Partitions:** Topics can be divided into multiple partitions, allowing messages to be distributed and processed in parallel across multiple consumers.
- **Brokers:** The servers that form the Kafka cluster. They store and replicate the message data, handle the publishing and consuming of messages, and ensure fault tolerance and high availability.
- **Consumer Groups:** Consumers can be organized into groups, where each group processes messages from one or more partitions of a topic. This allows for horizontal scalability and load balancing across consumers.

3.5 Prometheus

Monitoring and alerting open-source system toolkit called Prometheus was created at SoundCloud. Numerous businesses and organizations have used Prometheus since its launch in 2012, and the project has a thriving developer and user community. It is now an independent open-source project maintained by no particular business.

Prometheus gathers and saves its measurements as time series data, which means that the information about the metrics is kept along with the timestamp at which it was captured and optional key-value pairs known as labels.

Prometheus provides a multi-dimensional data model with time series data identified by metric name and key/value pair, similar to how Kubernetes organizes

infrastructure metadata. In order to provide flexible searches and real-time alerting, Prometheus uses a pull mechanism over HTTP to gather the real-time metrics in a time series database. Other essential features are, for instance, multiple modes of graphing and dashboarding support, no reliance on distributed storage, and the targets discovered via service discovery or static configuration [19].

Before explaining the Prometheus components and what they are used for, it is essential to understand two concepts that are related to Prometheus which are:

- **Metrics** are numeric measurements that play an essential role in understanding what is happening in the applications. For example, if we have an application with a high number of requests, the application can become slow. With this metric, it is possible to spot a reason to increase the number of instances to share the load.
- **Time series** mean that the metrics are recorded over the time.

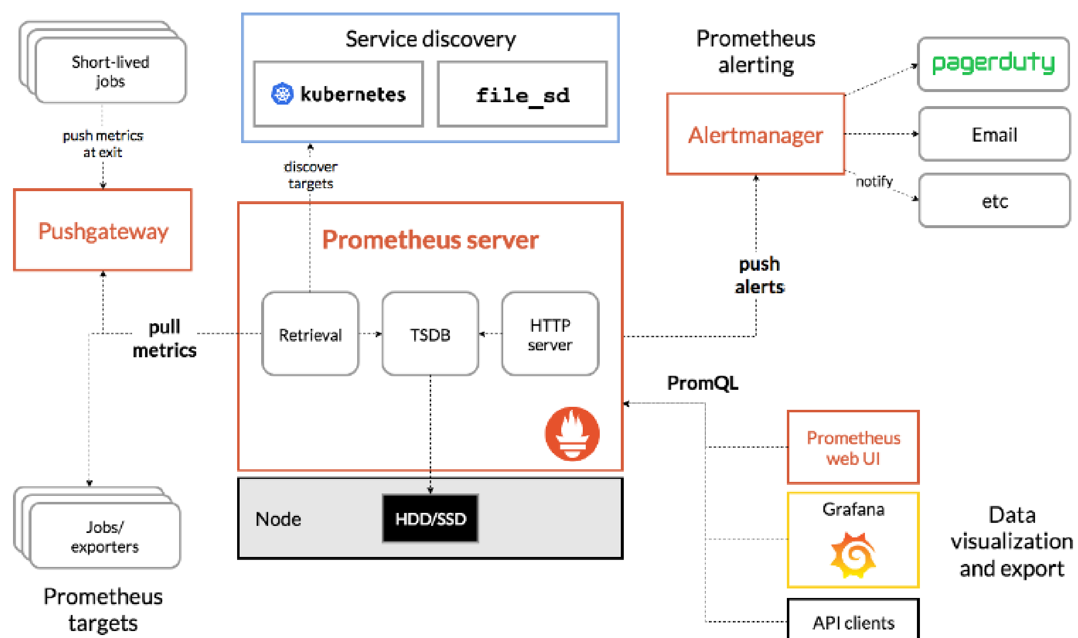

Figure 3.6: Prometheus architecture [20]

Figure 3.6 illustrates Prometheus architecture and its components which are: the **Prometheus server** scrapes metrics from jobs and stores time series data; **client libraries** for instrumenting application code; **Alertmanager** to handle the alerts; **Exporters** for sharing current metrics from external systems; and various support tools for example to data visualization [19].

3.6 Kubernetes-based Event Driven Autoscaler (KEDA)

KEDA [21] is an open-source framework designed to provide event-driven autoscaling capabilities for applications running on Kubernetes. It enables auto-

matic scaling of workloads based on the number of events or messages received from various event sources.

KEDA extends the functionality of Kubernetes by integrating with different event sources such as message queues, streaming platforms and other cloud-native event systems. It acts as an event-driven scaler that monitors the event sources and dynamically adjusts the number of instances of an application or workload to handle incoming events efficiently.

The architecture below shows how KEDA works in conjunction with the Kubernetes Horizontal Pod Autoscaler, external event sources, and Kubernetes etcd data store:

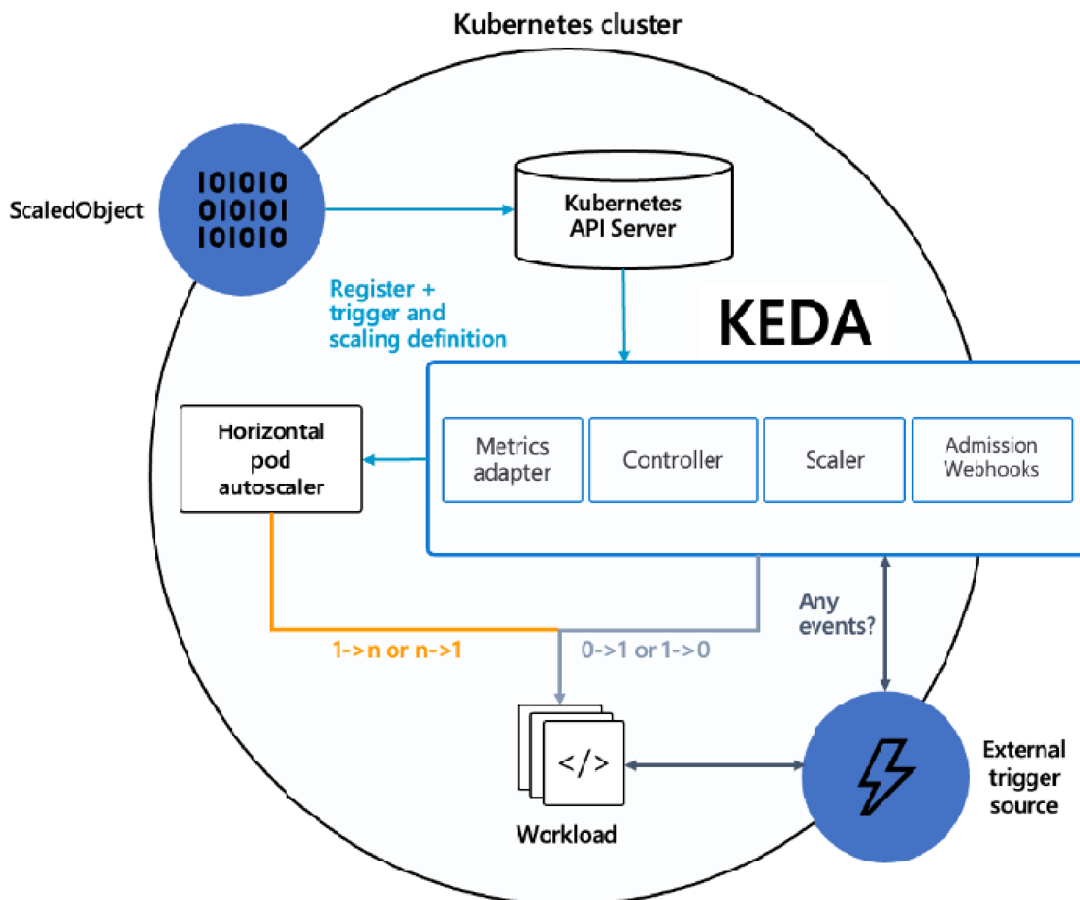

Figure 3.7: KEDA architecture [21]

3.7 Final Remarks

As mentioned in the previous section, there are three types of autoscaling that Kubernetes provides.

The VPA could be a problem when scaling because it causes a restart of the pod, and while the pod is restarting to increase the resource allocation, the service

could be temporarily down. In order to avoid this problem in reference, [22] it is proposed a non-disruptive solution where a checkpoint image is created to resolve the problem of restarting the pods and having a delay in the service.

It is also possible to combine these auto scalers. For instance, the authors in the reference [23] propose combining two auto scalers HPA and CA for where all the nodes are busy. The problem is that CA is only available in some cloud providers, which is very limited. Another combination is Libra [24], which combines VPA and HPA but, in this case, must be used with an operator because assigning the appropriate scaling levels to the service is not trivial to human operators.

KEDA allows for scaling based on the number of messages in a queue, for example, messages in a Kafka topic, instead of using traditional resource-based metrics like CPU and memory usage. This allows for more efficient use of resources by only scaling up when there are messages to process and scaling down when no messages are in the queue [21].

KEDA can be used with other Kubernetes scaling technologies like HPA and Cluster Autoscaler to create a hybrid scaling strategy that provides both resource-based and event-driven scaling, depending on the specific needs of your application[21].

In the reference [11], the author uses Prometheus custom metrics for HPA operations, where Prometheus scrap metrics are stored in the form of time series data for later on be exposed to the Prometheus adapter.

As described in the previous subsection, there are many ways to do scaling. However, as the Altice Labs have already instantiated a Prometheus server, choosing this technology for the custom metrics is essential. At the moment, the platform has several instances already instantiated. Therefore, the HPA is the wisest choice. One of the components for implementing autoscaling is a Kafka Consumer. It could be possible that KEDA is a good option for these because it is also compatible with the Prometheus Custom Metric.

It was done a proof of concept with KEDA where a server apache was overloaded with unlimited requests. The behavior of KEDA was very positive as expected.

The HPA is a good choice because, currently, the system has a fixed number of instances. This number of instances could be the maximum number of replicas defined in the configuration. Furthermore, KEDA could be an option, and it is possible to use it with HPA.

For the metrics Prometheus, it is the most common, and Altice Labs have a server already instantiated.

Chapter 4

Architectural Drivers

This chapter contains information regarding the current state of the Active Campaign Manager (ACM) platform. In addition, the requirements that have an impact on the architecture, such as functional requirements, technical and business restrictions, and quality attributes, are presented in this chapter as well.

4.1 Current System

As referred earlier, in this internship, the goal is to scale some crucial components automatically in an already built system by Altice Labs. Therefore, this section will describe the current system and it is components.

4.1.1 Overall Architecture

The system receives an enormous amount of events per second through a files repository, database, and web services, as represented in Figure 4.1. The system's goal is to process these events through the different components and create a campaign for the final customers.

For this internship, two main components must be changed to scale automatically. Currently, they have a set of instances that are always instantiated.

- Kafka
- Event Stream Processors

As it is possible to see in Figure 4.1, there are two ways of processing internally the events: assyncronhsly and syncronshly. More than 90% of the events are assyncronhsly and written on the raw-events Kafka Topic. Then the Event Stream Processor consumes the raw-events topic events to enrich them and writes the event to another topic called ready-events. These ready-events topic events are consumed by the Engine to be processed and generate campaigns for the final client.

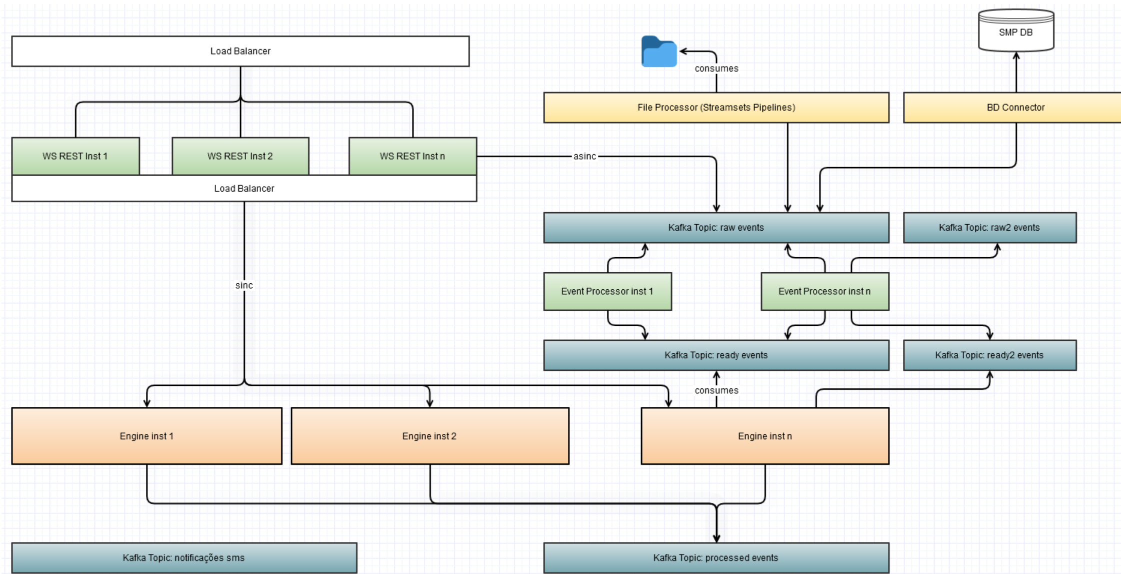

Figure 4.1: Diagram of the current ACM platform

4.2 System Functionality

In this section, we can find the requirements needed for this project. They serve to understand better what will be implemented in the system, the conditions necessary, and alternative scenarios. The requirements impact two components mentioned in the previous section 4.1.1.

- Kafka
- Event Stream Processors

Currently, the application is running in containers to make it possible to scale automatically according to load is necessary to use an orchestrator. For the orchestrator, Altice Labs has a Kubernetes Cluster available. To be able to perform scaling, it is necessary to collect metrics from the applications, so it was necessary to introduce new components that were not in the system, which are:

- Kafka Exporter
- Prometheus
- Horizontal Pod Autoscaler (HPA)

In the requirements defined below, there are some that the alternative scenario was left blank. This was on purpose, as there is no alternative scenario that allows normal operation.

Table 4.1: REQ1 - Deploy the Zookeeper

REQ1 - Deploy Zookeeper in the Kubernetes Cluster	
Level	Sea
Actor	Zookeeper
Objective	The Zookeeper must be deployed in Kubernetes Cluster
Preconditions	The Zookeeper is running in containers
Postconditions	The Zookeeper has to work has before in the cluster
Main Scenario	1 - The replicas are up and running 2 - There are no error logs 3 - The Kafka successfully connects to Zookeeper
Alternative Scenario	

Table 4.2: REQ2 - Deploy Kafka

REQ2 - Deploy the Kafka in the Kubernetes Cluster and Configure the topics	
Level	Sea
Actor	Kafka
Objective	Consuming events from different sources
Preconditions	Zookeeper must be running on the cluster for Kafka to connect
Postconditions	List of available topics and event consumption
Main Scenario	1 - The replicas are up and running 2 - Topic creation 3 - Open a producer for the topic created and write a message 4 - Open a consumer for the topic and visualize the message
Alternative Scenario	

Table 4.3: REQ3 - Deploy Kafka Exporter

REQ3 - Deploy Kafka Exporter in the Kubernetes Cluster	
Level	Sea
Actor	Kafka Exporter
Objective	Scrap metrics from the Kafka Brokers
Preconditions	The Kafka Brokers must be up and running
Postconditions	It must be possible to view the metrics through the endpoint
Main Scenario	1 - The replica is running and up 2 - Access to the endpoint 3 - Display of the metrics corresponding to the brokers
Alternative Scenario	

Table 4.4: REQ4 - Deploy Prometheus

REQ4 - Deploy Prometheus in the Kubernetes Cluster	
Level	Sea
Actor	Prometheus
Objective	Scrap metrics from the applications to the HPA
Preconditions	The Kakfa Exporter is running
Postconditions	The HPA calculates the desired replicas
Main Scenario	1 - Prometheus scrapes metrics from the Kafka Exporter 2 - Query custom metrics collected by Prometheus 3 - Register metrics in the customer metrics API 4 - Horizontal Pod Autoscaler query metrics from the API
Alternative Scenario	1a. Prometheus can not scrape metrics from the applications 1a.1 Use resources metrics to get CPU and Memory Usage

Table 4.5: REQ5 - Deploy Grafana

REQ5 - Deploy Grafana in the Kubernetes Cluster	
Level	Sea
Actor	Grafana
Objective	Graphic display of the metrics
Preconditions	The Prometheus is running
Postconditions	It must be possible to view the metrics
Main Scenario	1 - The replica is running and up 2 - Access to the endpoint and login 3 - Add the Kafka Exporter dashboard 4 - visualization of the metrics in graphs for purposes of monitoring
Alternative Scenario	3a. It is not possible to add Kafka Exporter dashboard 3a.1 Create your own dashboard

Table 4.6: REQ6 - Deploy Event Stream Processor

REQ6 - Deploy Event Stream Processor in the Kubernetes Cluster	
Level	Sea
Actor	Event Stream Processor
Objective	Enrich the consumed events of the raw-events topic
Preconditions	Events in the Raw-events topic to be consumed
Postconditions	Enriched events are written in the ready-events topic
Main Scenario	1 - The replicas are up and running 2 - Consumes a valid event from the raw-events topic 3 - Enriches the event consumed 4 - Write the valid event in the topic ready-events
Alternative Scenario	2a. Consumes an invalid event from the raw-events topic 3a.1 The event is not enriched and is discarded

Table 4.7: REQ7 - HPA for scaling down

REQ7 - Scaling down the Event Stream Processor	
Level	Sea
Actor	Kubernetes
Objective	Scaling down the number of instances of the Event Stream Processor
Preconditions	The lifetime of the replicas is more significant than that defined The metrics' values are below their thresholds The number of replicas is greater than the minimum defined
Postconditions	The replicas are removed
Main Scenario	1 - The control manager communicates the number of desired replicas to the Kube API server 2 - Kube API server communicates with Kubelet 3 - kubelet is in charge to remove the replicas
Alternative Scenario	3a. Kubelet can not remove the pod 3a.1 Notify the developer to solve the issue

Table 4.8: REQ8 - HPA for scaling up

REQ8 - Scaling up the Event Stream Processor	
Level	Sea
Actor	Kubernetes
Objective	Scaling up the number of instances of the engines
Preconditions	• The number of replicas is lower than the maximum number of replicas defined
Postconditions	• The new replicas are created and start running
Main Scenario	1 - The control manager communicates the number of desired replicas to the Kube API server 2 - Kube API server communicate with Kubelet 3 - kubelet is in charge to deploy new instances
Alternative Scenario	3a. Kubelet can not deploy the pod on three consecutive tries 3a.1 Launched a different pod

4.3 Quality Attributes

In this section, the quality attributes of the system, also known as non-functional requirements, will be described. Each non-functional requirement will have a brief description and a rating(H for high, M for medium, and L for Low) of how much it will impact the architecture.

- **Portability (H)**
  - Currently, the system is deployed on-premises, but in the event of deploying on the cloud, the system should function without modifying the source code.
- **Elasticity (H)**
  - In the occurrence of a workload fluctuation, the system should be able to automatically add or remove instances without affecting the regular operation of the system.

- **Availability (H)**
  - If a system failure occurs, that impedes its correct behavior, the system should replace or restart the container as soon as possible.

4.4 Restrictions

Restrictions can be divided into technical and business. Usually, they are constraints to the architecture and have limited flexibility. Some examples are internship decisions, deadlines, and projects that already exist.

4.4.1 Business Restrictions

Business restrictions do not have any influence on the architecture though they can have an impact on it.

The business restrictions identified are:

- **Development time**
  - Definition: The development time of this project should not exceed the duration of the internship.
  - Flexibility: The duration of the internship can be extended.
  - Alternatives: None

4.4.2 Technical Restrictions

Technical restrictions are the ones that have more influence on the architecture when compared with business restrictions. These restrictions include technologies, protocols, and program languages. The technical restrictions identified are:

- **The system must work when deployed on cloud and on-premises infrastructure**
  - Definition: Unknown;
  - Flexibility: None;
  - Alternatives: None;
- **The system must run on a Kubernetes cluster**
  - Definition: Unknown;
  - Flexibility: None;
  - Alternatives: None;

Chapter 5

Architecture

This chapter describes the system architecture, which was developed based on the requirements presented in the previous chapter. The architecture was done through multiple views using the C4 model [25].

5.1 C4 Model

The C4 Model is an architectural model of software developed by Simon Brown. This model splits the software into four levels and goes from general to more detailed. The four levels are context, containers, components, and code. The following section describes the four levels of our system.

5.1.1 System Context Diagram

Context Diagram is the higher level of the diagram, which illustrates how the system interacts with external systems and users.

In this case, as it is possible to verify in Figure 5.1, the system interacts with two users, one configures the campaigns, and the final user o receives the campaigns.

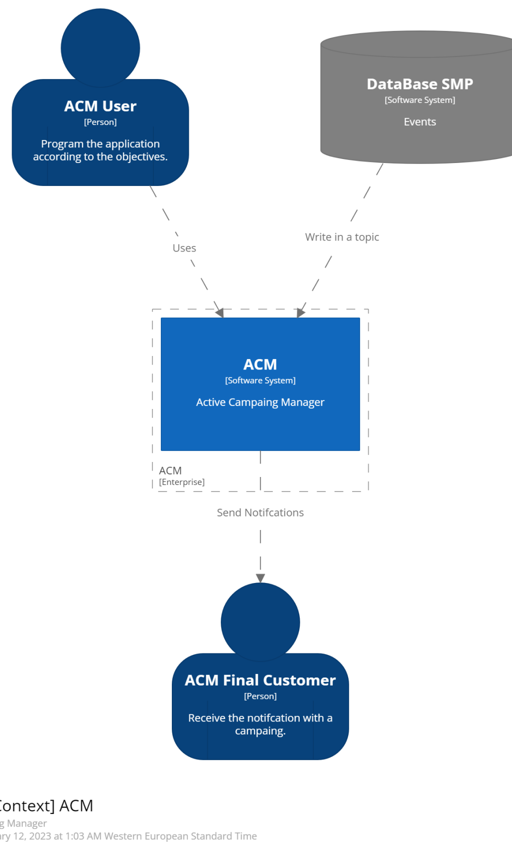

Figure 5.1: Software Architecture: Context Level

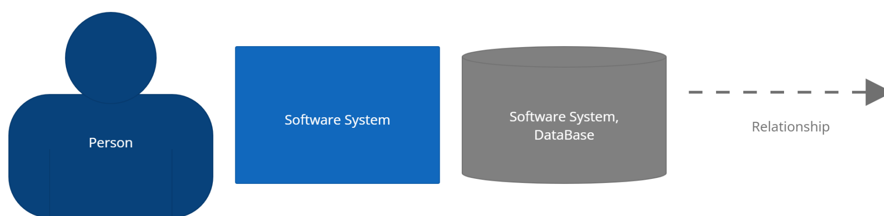

Figure 5.2: Software Architecture: Context Level Legend

5.1.2 Container Diagram

The following diagram contains more detail than the previous one. It represents the system in the form of containers. This diagram is represented in Figure 5.3, followed by a legend for a better understanding.

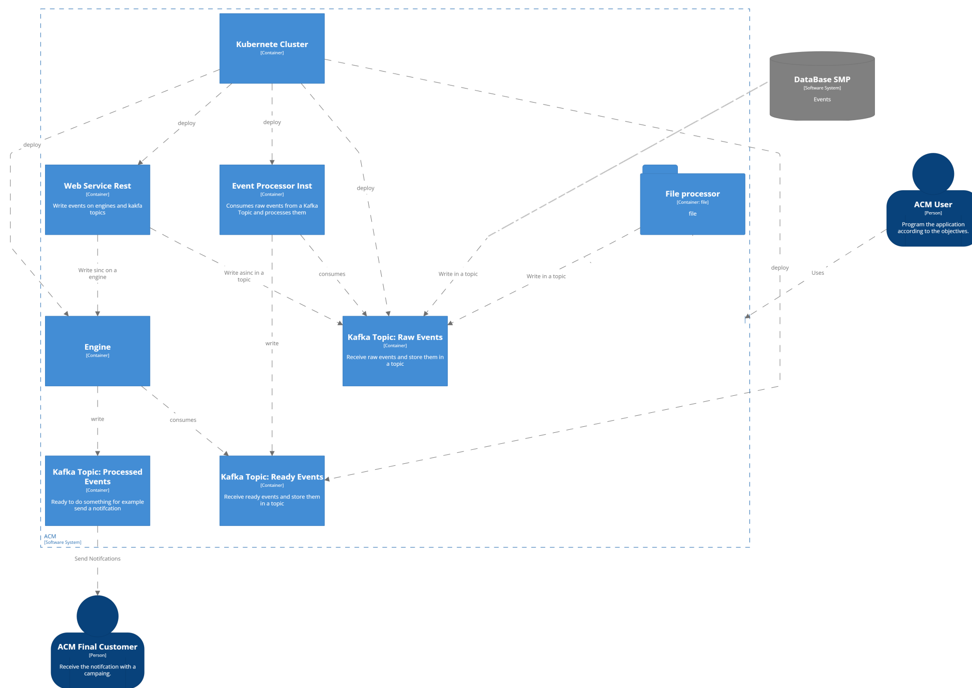

Figure 5.3: Software Architecture: Container Diagram

Figure 5.3 illustrates that the system sends events to a Kafka topic in three forms. Also, it is possible to send events synchronously directly to an engine despite being a minor. All the others are assincronusly and sent to a Kafka topic. The Database, the file processor, and the Kafka Topic of processed events are out of the scope of this internship.

As seen in Figure 5.3, all the other elements are being deployed by Kubernetes, and this is the internship's main focus. The main goal is that the system can scale up and down, which means adding or removing instances according to the workload of the application.

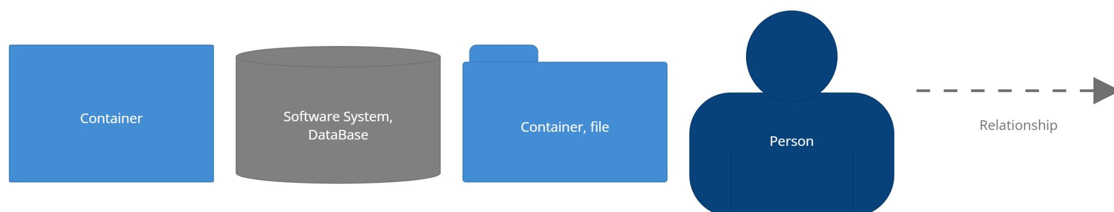

Figure 5.4: Software Architecture: Container Diagram Legend

5.1.3 Components Diagram

The components diagram goes into detail about each container to understand the functioning of each container and its interactions.

Only one instance of each container is represented except the Kubernetes Container for an easier understanding of the architecture. Otherwise, the information will be redundant because two instances of the same applications have the same connections.

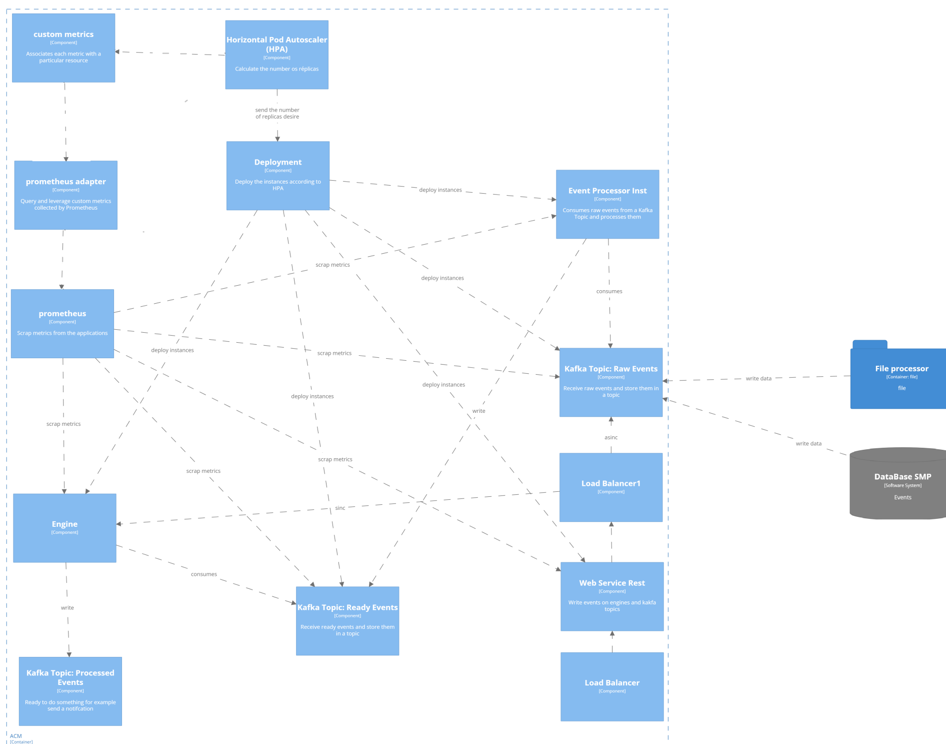

Figure 5.5: Software Architecture: Components Diagram

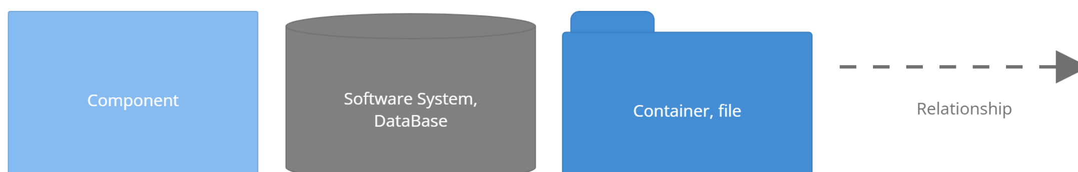

Figure 5.6: Software Architecture: Components Diagram Legend

In the Kubernetes container, there will be five components. First, the Prometheus component is responsible for scrapping metrics from the different applications. Figure 5.5 illustrates that this component will get metrics from the Kafka topics, events processors, web service rest, and engines. With these metrics in the Prometheus, the component Prometheus Adapter will query the metrics for custom metrics and register them on the Custom metrics component. Then the Hor-

izontal Pod Autoscaler (HPA) will collect these custom metrics from the Custom metrics API server and calculate the number of replicas desired. Subsequently, the Deploy component will instantiate new replicas or remove them according to the HPA.

5.1.4 Code

According to the documentation of the C4 Model [25], the code level is optional because it is necessary to have a high level of detail in the project's initial phase. For that reason was not described in this stage.

5.2 Analysis

This section will analyze the architecture to verify if it complies with the Architectural drivers in section 4.

- **Requirements** (section 4.2)
  - **REQ1, REQ2, REQ6** Currently, the applications are running on virtual machines. The goal is to deploy the application into Kubernetes containers.
  - **REQ3 and REQ4** are a requirements available in the following components: *Prometheus*, which scrapes metrics from the Kafka Exporter, *Prometheus adapter*, and custom metrics.
  - **REQ7** and**REQ8**, these requirements are presented in the *Kubernetes Cluster*. This container is responsible for the scale of the instances up and down. HPA calculates the number of instances desired based on metrics and components of the cluster, such as *Kubelet*t, *Control manager*, and *Kube API server* is responsible for the scaling.
- **Restriction** (section 4.4)
  - In this project the concept of Kubernetes will be present. It is possible to verify that through the Container Diagram in Figure 5.3.
  - Kubernetes can be deployed on-premises but also can be on the cloud. So it is possible to deploy the application on the two infrastructures.
- **Quality Attributes**
  - **Portability (H):** As mentioned before, Kubernetes can be deployed on-premises and cloud, fulfilling this quality attribute
  - **Elasticity (H):** This quality attribute is one of the main goals of the project. Therefore, the HPA should calculate the number of replicas desired, and Kubernetes automatically deploy and remove instances.

- **Availability (H):** Kubernetes can guarantee availability because one of the Kubernetes features is Self-healing. This allows for killing, restarting, and replacing the containers if any problems occur.

Chapter 6

Implementation

This chapter describes the work developed during the second semester, which consisted of deploying several components of the ACM platform and other parts needed to make the scale of instances possible. It starts by explaining the tools and practices used in the development and describing how the system is deployed. Subsequently, the requirements and the architecture defined were implemented and will be explained.

6.1 Approach

This section is dedicated to the tools and practices related to the development of the system during the internship. It describes the Integrated Development Environment (IDE) used in the implementation and why it was chosen. Other tools were also used to monitor and retrieve logs from the instances running on the cluster.

6.1.1 Integrated Development Environment

An IDE is a piece of software where it is possible to edit source code and may also include tools to assist in developing applications, such as a compiler and debugger. During the internship, the IDE used was Visual Studio Code by Microsoft [26]. It is a very lightweight application that can be used with third-party extensions.

6.1.2 Deployment

This system is deployed on a single Kubernetes cluster from the Altice Labs. In this section, the current deployment will be explained.

Altice Labs provided one Kubernetes cluster to use in this internship. The permissions allowed inside the cluster were restricted to a namespace named "ACM"

because other developments are being accomplished on the cluster.

Altice Labs also provided one machine to use in this internship. This machine has 4 CPU cores, 8GB of RAM, and 100GB of storage. Currently, this machine contains all the scripts, code developed, and configuration files created throughout this internship.

Regarding the address, the current one is dm-om.ptin.corppt.com. To access the machine, the protocol ssh was used. On top of that, FortiClient VPN was also used to access Altice Labs' internal network, and there was also a Firewall Authentication. Furthermore, some configuration files must be on the machine to access the Kubernetes Cluster. Since these files needed to be on the machine, FileZilla[27] was used to transfer these files, and the connection was made through port 22(SSH access).

To deploy the components, Kubectl and Kustomize[28] were used. These two methods allow various components to be built, deployed, updated and stopped quickly and easily. Also, Kubectl allows checking logs of each component.

6.2 Development

In this section, the development in the second semester will be explained. The topic will explain how different parts were deployed in Kubernetes and the new parts needed for autoscaling.

6.2.1 Zookeeper

One of the essential components of this acm prototype is Kafka. Since the version deployed is 2.8.1, it was mandatory to deploy a Zookeeper only from version 3.0.0 to the latest is not necessary to deploy Zookeeper.

Zookeeper is a distributed coordination service commonly used for maintaining configuration information, naming, and providing synchronization across distributed systems. For deploying Zookeeper in Kubernetes, the following steps were followed.

A Kubernetes manifest was created to define the Zookeeper deployment. This manifest includes details such as the container image, resource requirements, and other necessary configurations, for instance, the ports for communication with other services. Also very important is to specify the number of replicas.

The following parameters are the most important in the configuration, and they were used to configure the StatefulSet.

- **image:** bitnami/Zookeeper:latest
- **ports:** 2181, 2888, 3888

- **replicas:** 3
- **servers:**
  crs-zookeeper-0.zookeeper-cluster.acm.svc.cluster.local:2888:3888,
  crs-zookeeper-1.zookeeper-cluster.acm.svc.cluster.local:2888:3888,
  crs-zookeeper-2.zookeeper-cluster.acm.svc.cluster.local:2888:3888

Using a StatefulSet was necessary to ensure stable network identities and persistent storage for each ZooKeeper instance. A StatefulSet provides unique hostnames and stable network identities to each ZooKeeper pod. It also enables each pod to have its persistent storage volume.

After the deployment, it is necessary to expose the Zookeeper within the cluster. The service provides a stable network endpoint for client applications to connect to the Zookeeper ensemble. In this case, the connected application was Kafka. The following parameters are the ones used to configure the service

- **ports name:** client, follower, leader
- **ports:** 2181, 2888, 3888

The commands used to deploy both, StatefulSet and service are the following:

- kubectl –kubeconfig kube.config apply -f statefulset.yaml -n acm
- kubectl –kubeconfig kube.config apply -f service.yaml -n acm

6.2.2 Kafka

Apache Kafka is a distributed streaming platform that allows for the high-throughput, fault-tolerant storage and processing of streams of records. For deploying Kafka, the following steps were followed. A Kubernetes manifest was created to define the Kafka deployment. This manifest includes details such as the container image, resource requirements, and number of replicas, among other configurations.

- **hostname:** Kafka
- **image:** rdocker.ptin.corppt.com/exmirrors/kafka:2.8.1
- **resources:** memory 1GB/ cpu 1
- **port:** 9092
- **replicas:** 3

replicas: 3 As with Zookeeper, it is necessary to ensure stable network identities and persistent storage to guarantee that a StatefulSet was used.

Also, it is necessary to create environment variables. In this variable's environment, it will be assigned the Zookeeper to which the Kafka should be connected to run correctly. For example:

- **name:** KAFKA-CFG-ZOOKEEPER-CONNECT
- **value:**
crs-zookeeper-0.zookeeper-cluster.acm.svc.cluster.local:2181,
crs-zookeeper-1.zookeeper-cluster.acm.svc.cluster.local:2181,
crs-zookeeper-2.zookeeper-cluster.acm.svc.cluster.local:2181

Kubernetes Service object was also created to expose the Kafka brokers. The service provides an endpoint for a client application to connect to the Kafka. For example, the Kafka Exporter in the next section will connect to this endpoint to scrape metrics from Kafka. The service has the following parameters:

- **name:** Kafka-cluster
- **port:** 9092

After the previous steps had been carried out, some tests were executed to verify if the Kafka was working correctly. They are explained in the next chapter.

As explained before, this Kafka broker needs to have two topics. The topics are "raw events" and "ready events."

To create topics in the Kafka broker, there are several methods available. However, the easiest way (opening the console and manually creating the topics) may not be the most reliable. In case of an application failure, the topics may be lost. To execute this method, use the following command: **kafka-topics.sh –bootstrap-server kafka-ready:9094 –topic ready-events –create –partitions 5 –replication-factor 3**

For this application, the topics can not be lost. Therefore, the problem was solved by creating a ConfigMap.

Once the ConfigMap is defined, it can be used as an environment variable, or it is possible to mount it as a volume. This allows the Kafka brokers to read the topic configuration from the ConfigMap and create the respective topics during the startup.

Using a ConfigMap to define the persistent topics makes it easy to manage and update the topic configuration if necessary without modifying the Kafka deployment.

6.2.3 Kakfa Exporter

Kafka Exporter is a software tool designed to collect metrics from Apache Kafka and export them to a monitoring system for analysis and visualization. It acts as a bridge between Kafka and the monitoring system, facilitating the monitoring and observability of Kafka clusters.

- **Metrics Collection:** Kafka Exporter connects to Kafka clusters and collects various metrics related to the Kafka ecosystem. It retrieves metrics from Kafka brokers, topics, partitions, producers, consumers, and other components of the Kafka infrastructure. These metrics provide insights into the performance, behavior, and resource utilization of the Kafka cluster.
- **Metric Exposition:** Kafka Exporter exposes the collected metrics in a standardized format that can be consumed by monitoring systems. It typically uses the Prometheus exposition format, allowing integration with widespread monitoring tools such as Prometheus and Grafana. In addition, the exporter provides an HTTP endpoint where the monitoring system can scrape and collect the metrics.

The Kafka Exporter is an open-source project [29], and it is possible to obtain the source code on GitHub [30]. Also, this repository contains the instructions to build the image.

A Kubernetes manifest was created to define the Kafka exporter deployment. The essential configurations on the manifest are the image and the arguments to initiate the image.

- **hostname:** osm-kafka-exporter
- **image:** danielqsj/kafka-exporter:latest
- **port:**
  - **name:** metrics
  - **containerPort:** 9308
- **args:**
  - –kafka.server=kafka-cluster:9092
  - –web.listen-address=:9308

Kubernetes Service object was also created to expose the Kafka exporter metrics. The service has the following parameters.

- **port**
  - **name:** metrics
  - **containerPort:** 9308

6.2.4 Prometheus

Prometheus is used to scrape metrics periodically from Kafka. Because Prometheus makes HTTP requests and Kafka does not provide endpoints, the metrics are scraped from Kafka Exporter.

Three manifests were created for Prometheus, the Deployment, a ConfigMap, and an Ingress. The ConfigMap contains the configurations necessary to scrape the applications. This configuration also contains three specific applications Prometheus itself, the Kafka Exporter, and an application called Express that was used for testing.

```
scrape_configs:
  - job_name: "prometheus"
    static_configs:
    - targets: ["localhost:9090"]
  - job_name: "kafka"
    static_configs:
    - targets: ["osm-kafka-exporter:9308"]
  - job_name: "express"
    static_configs:
    - targets: ["express:8081"]
```

Figure 6.1: ConfigMap Prometheus

The Deployment manifest contains the image of the container, the port, the mount to use the configurations that are present in the ConfigMap, and a Service because in order to use an Ingress in Kubernetes, it is necessary to have a Service defined for the backend application or service to expose. The Ingress resource acts as a traffic router and requires a Service to direct the incoming requests.

The Ingress is an Application Programming Interface (API) object that manages external access to services within a cluster. It acts as a traffic controller, routing incoming requests to the appropriate services based on user-defined rules. In other words, it allows to expose Kubernetes services to the outside world. The following endpoint *https://acm-prometheus.t-k8s.ptin.corppt.com/* does access to the service.

6.2.5 Grafana

Grafana is used for visualization and monitoring the Kafka Brokers' obtained data via Prometheus.

As in the previous section, the same three manifests were also required. The ConfigMap manifest is used to keep the configurations of the data sources. In this case, the only data source is Prometheus, and the configuration is represented below.

The deployment is very similar to the Prometheus but with different values. The

```
{
    "apiVersion": 1,
    "datasources": [
        {
            "access":"proxy",
             "editable": true,
             "name": "prometheus",
             "orgId": 1,
             "type": "prometheus",
             "url": "http://prometheus-srv:9090",
             "version": 1
        }
    ]
}
```

Figure 6.2: ConfigMap Grafana

only significant difference is in the service because it is necessary to have an annotation with the following values.

- **prometheus.io/scrape:** ’true’
- **prometheus.io/port:** ’3000’

The Ingress is also very similar, and the difference is the number of ports and the endpoint to access the service, which is http://acm-grafana.t-k8s.ptin.corppt.com/.

After deploying and verifying that the application is running is necessary to create or import a template to visualize the data. Since there was already a template with all the necessary information, the dashboard Kafka Exporter Overview [31] was imported into the Grafana application.

6.2.6 Event Stream Processor

The Event Stream Processor is an application that consumes the events from the Raw Events topic and then enriches these events. After this enrichment process, the enriched event is written on the Ready Events topic to be consumed by the Engine.

Three manifests were created to deploy the Event Stream Processor. This image of the application requires four volumes to work correctly. These files were copied into a ConfigMap.

- 0-acm-event-builder-config.xml
- domain.xml
- oms-config.properties
- oms-daemon-config.properties
- oms-evstream-config.json

The 0-acm-event-builder-config.xml file contains the rules for the enrichment each event that is read from the source topic and makes the event ready. The oms-evstream-config.json is where the bootstraps servers, source topics, and target topics, among other parameters, are defined for the proper functioning of the application.

The remaining files define the OMS daemon service, which collects and forwards data from various sources to the OMS service for analysis and monitoring. A daemon image was also deployed for the OMS daemon service to connect to the Event Stream Processor. Finally, after these two manifests are created and deployed, the system is ready to deploy the Event Stream Processor.

A Statefullset was used instead of deployment to deploy the Event Stream Processor because of the previously referred properties. This configuration has the following parameters:

- **replicas:** 1
- **port:** 9095
- **image:** rdocker.ptin.corppt.com/acm/acm-event-stream-processor:6.6.1-202303141433

There is also a configuration for all the volumes described above, as it is possible to visualize in the image below.

```yaml
        volumeMounts:
          - name: config-volume
            mountPath: /opt/ptin/acm/jboss/domain/configuration/domain.xml
            subPath: domain.xml
          - name: config-volume
            mountPath: /opt/ptin/oms/conf/oms-daemon-config.properties
            subPath: oms-daemon-config.properties
          - name: config-volume
            mountPath: /opt/ptin/acm//jboss/servers/event_stream_1/conf/oms-config.properties
            subPath: oms-config.properties
          - name: config-volume
            mountPath: /opt/ptin/acm//jboss/servers/event_stream_1/conf/oms/oms-evstream-config.json
            subPath: oms-evstream-config.json
          - name: config-volume
            mountPath: /opt/ptin/acm/conf/event-builder/0-acm-event-builder-config.xml
            subPath: 0-acm-event-builder-config.xml
      volumes:
        - name: config-volume
          configMap:
            name: event-stream-config-map
            items:
              - key: domain.xml
                path: domain.xml
              - key: oms-daemon-config.properties
                path: oms-daemon-config.properties
              - key: oms-config.properties
                path: oms-config.properties
              - key: oms-evstream-config.json
                path: oms-evstream-config.json
              - key: 0-acm-event-builder-config.xml
                path: 0-acm-event-builder-config.xml
```

Figure 6.3: Volumes of Event Stream Processor

Service was also created for the application.

6.2.7 Horizontal Pod Autoscaler

Keda is used to orchestrate the application, which automatically adjusts the number of replica pods based on the custom metrics from Kafka.

In order to utilize Keda, the installation of Keda onto the cluster environment is required. The installation has been carried out in conjunction with the supervisor due to the high level of permissions required.

After the installation, is it possible to create a manifest a manifest where the Kind parameter is ScaledObject. This manifest contains the scaled target, the number of minimum and maximum replicas, the scraped interval, cooldown period, among other configurations in the image below.

```yaml
kind: ScaledObject
metadata:
  name: event-stream-processor-scaledobject
  namespace: acm
spec:
  scaleTargetRef:
    apiVersion: apps/v1  # Optional. Default: apps/v1 # mandatory if it is a satefulset
    kind: StatefulSet   # Optional. Default: Deployment # mandatory if it is a satefulset
    name: acm-event-stream-processor
  pollingInterval: 30
  cooldownPeriod:  300
  minReplicaCount: 1
  maxReplicaCount: 3
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: 10.42.0.14:9092,10.42.1.35:9092,10.42.2.37
      topic: raw-events
      consumerGroup: event-stream-prc
      lagThreshold: "100"
      offsetResetPolicy: latest
```

Figure 6.4: Horizontal Pod Autoscaler (HPA) Event Stream Processor

Chapter 7

Testing

In order to verify compliance with the architectural drivers described in Chapter 4, manual tests were performed on the system. As different components were deployed is hard to define a test structure because each deployed application has its specificities. The system compliance with the quality attributes is described as well in this chapter.

7.1 Manual functional test

This section focuses on tests for the requirements implemented, some of which require scripts and interaction between the different applications.

REQ2 - Deploy Kafka

For this test, a topic was created in the broker. Then a consumer and a producer should be created and be able to communicate.

- **Test case 1:** List all the topics. It is expected to appear the topics previously created.
- **Test case 2:** Open a consumer and a producer, then send a message through the producer. The message sent by the producer to the consumer is expected to appear in the producer console.

Results

The following table represents the results for the REQ2. It includes the outcome of each test, which can be "Pass" or "Fail," along with a comment.

Test case #	Result	Comment
1	Pass	The topic appears in the list of topics
2	Pass	The message sent to the producer was read in the consumer

Table 7.1: Results for the test cases of REQ2

REQ3 - Deploy Kafa Exporter

For this test, a specific configuration was done in the Kafka broker. Then some actions were executed to verify if the metrics were according to the configuration and the actions performed.

For example, three Kafka broker instances were deployed. Then a topic was created with five partitions. A producer was opened in the brokers, and three messages were sent without a consumer to simulate lag.

- **Test case 1:** Visualize all metrics through the endpoint. It is expected to access all metrics via http://10.112.76.36:30257/metrics.
- **Test case 2:** The kafka-brokers metric value is as defined. It is expected that the value to be three.
- **Test case 3:** The kafka-topic-partitions metrics value is as defined. It is expected that the value to be five.
- **Test case 4:** The kafka-consumergroup-lag is as defined. It is expected that the value to be three.

Results

The following table represents the results for the REQ3. It includes the outcome of each test, which can be "Pass" or "Fail," along with a comment.

Test case #	Result	Comment
1	Pass	The metrics were visualized in the browser via the endpoint
2	Pass	The metric value was three
3	Pass	The metric value was five
4	Pass	The metric values was three

Table 7.2: Results for the test cases of REQ3

REQ4 - Deploy Prometheus

For this test, it was necessary to consider whether the deployment was correctly working. After this and guarantee that the REQ3 has passed all the tests, we should test if the metrics of the Kafka Exporter are in the Prometheus.

- **Test case 1:** Access to the endpoint http://acm-prometheus.t-k8s.ptin.corppt.com/. The interface is expected to appear, and after executing the query, http-requests-total should return the number of HTTP requests done.
- **Test case 2:** Access to the endpoint http://acm-prometheus.t-k8s.ptin.corppt.com/targets. It is expected to appear two targets which are Prometheus and Kafka, and their state should be UP.

- **Test case 3:** Using the exact configuration of the Kafka broker and querying the same metrics as in the tests for the REQ3. It is expected that the values returned are the same.

Results

The following table represents the results for the REQ4. It includes the outcome of each test, which can be "Pass" or "Fail," along with a comment.

Test case #	Result	Comment
1	Pass	The endpoint was accessed and the query was executed with success
2	Pass	The endpoint was accessed and the targets were up
3	Pass	The values of the querying and metrics are the same

Table 7.3: Results for the tests cases of REQ4

REQ5 - Deploy Grafana

The REQ4 must be fully functional for this test, and the metrics collected to the Prometheus from the Kafka Exporter should be displayed in graphics over time.

- **Test case 1:**Access to the endpoint http://acm-grafana.t-k8s.ptin.corppt.com/. It is expected to be directed to a login page, and we should be able to navigate to the dashboard tab.
- **Test case 2:**Generate lag in a consumer group. It is expected to visualize a growth from zero to the value of messages that were not consumed.
- **Test case 3:** Consume the messages that generate lag. It is expected to visualize a decrease in the lag from the number of messages not consumed to zero.

Results

The following table represents the results for the REQ5. It includes the outcome of each test, which can be "Pass" or "Fail," along with a comment.

Test case #	Result	Comment
1	Pass	The endpoint was accessed, the login was successful and it was possible to navigate on the dashboard
2	Pass	The value was greater than zero
3	Pass	The values were zero after the messages had been consumed

Table 7.4: Results for the tests cases of REQ5

REQ6 - Event Stream Processor

For this test, it was necessary to configure the Kafka brokers to have two topics: raw-events and ready-events. It is also necessary to have an event in JSON format to send to the topic.

- **Test case 1:** Write on topic raw-events a valid event. The Event Stream Processor is expected to enrich this event and write on topic ready-event.
- **Test case 2:** Write on topic ready-events an invalid event. The Event Stream Processor is expected to ignore this event and not appear on the topic ready-event.

Results

The following table represents the results for the REQ6. It includes the outcome of each test, which can be "Pass" or "Fail," along with a comment.

Test case #	Result	Comment
1	Pass	The enriched event appears on the topic read-events
2	Pass	The invalid event does not appear on the topic ready-events

Table 7.5: Results for the tests cases of REQ6

REQ7 and REQ8 - Horizontal Pod Autoscaler

For this test, first, installing KEDA in the Kubernetes cluster was necessary after the installation through commands in the console, and by creating a manifest, the kind is ScaledObject, it was possible to verify that KEDA was installed and working.

The tests on this requirement are explained in the next section related to scalability.

7.1.1 Manual functional test results

All the manual tests pass, indicating that no expected behavior has been discovered. Although the behavior is as expected does not imply that it is bug-free. It just implies that the present tests are unable to find any bugs.

Currently, tests cover some functional requirements(REQ2 to REQ7). The following section will cover the quality attributes previously defined.

7.2 Quality attribute verification

This section analyzes the system's compliance with the quality attributes described in Section 4.3.

- **Portability:** The quality attribute is complied with using Kubernetes and Docker containers. Kubernetes is built around containerization, explicitly using the Docker container runtime. All dependencies and runtime requirements are encapsulated, making them more portable across different environments.

- **Availability:** The system currently complies with the quality attribute since Kubernetes provides a robust framework for managing and orchestrating containerized applications, which can enhance system availability. For example, replication and scaling help mitigate the impact of a failure in a replica, and the self-healing mechanisms that allow Kubernetes to restart pods if they fail or become unresponsive.

- **Elasticity:** The system complies with the quality attribute, and these two things were done. First, deploying the application in Kubernetes allows pod scaling and replicating and using a Horizontal Pod Autoscaler that launches and removes pods according to workload fluctuation.

7.3 Scalability Tests

The core of this internship is the scaling of an application when there is a significant increase in events written in the Kafka topic. Consequently, the tests considered several variables: the number of events, replicas, and the scaling mechanism active and deactivated.

In order to maintain the consistency of the results, the events were sent via script to be continually sent at equal time intervals. Furthermore, each test was performed ten times to prevent possible outliers.

First, it is necessary to ensure that the scaling is working correctly. A scenario was designed to send 75 000 events. The events were sent three times in the following order: the scaling deactivated, the second with the scaling activated, and then with two active replicas of the previous scaling.

- **Scenario 1:** 75 000 Events, no scaling, and one replica.

- **Scenario 2:** 75 000 Events, scaling, and one replica.

- **Scenario 3:** 75 000 Events, scaling, and two replicas from the scaling in scenario 2.

Results

Through the console, it was possible to visualize that new replicas were created, but this is not enough to ensure that the scaling works correctly despite being a good sign. After verifying that the new replicas are being created, it is necessary to analyze the consumption of events, and for that, Prometheus was used.

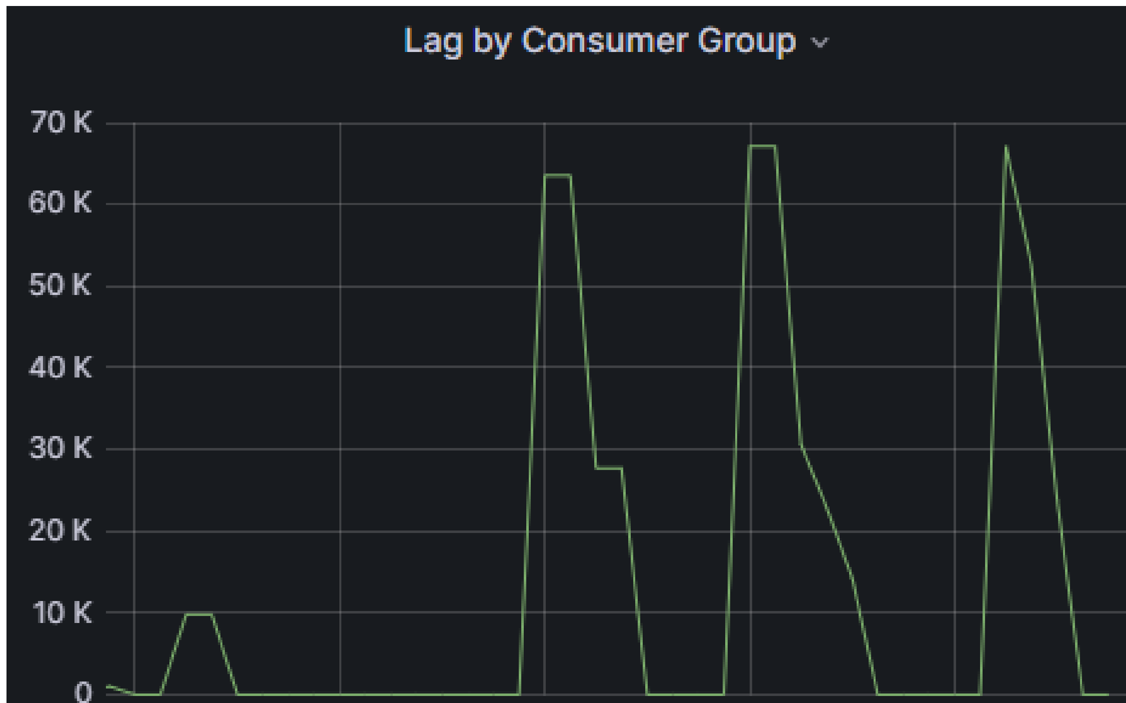

Figure 7.1: Lag by consumer group for the different scenarios

As we can see from the image, when the events are sent, the lag peaks in the graphic are close to 75 000. The first peak corresponds to scenario one, and as it is possible to visualize around 30 000, there is an interval in which this value is constant, which means that the Event Stream Processor is full and can not consume more events. The second peak corresponds to scenario two, and it is possible to visualize that the constant value like the one on the previous peak does not exist because the scaling mechanism is triggered and both replicas are consuming events. Finally, the third peak corresponds to scenario three, as expected. Since the events are sent, they are immediately consumed by the two replicas, thus having a steeper line of descent.

Chapter 8

Cloud Costs

The aim of this internship is the scaling of applications in order to save resources and also to be able to run the application in different environments, such as the cloud, using Kubernetes technology. A study was carried out that led to the writing of an article in conjunction with Professor Filipe Araújo about cloud costs with serverless environments. For this study, accurate data from an actual client was used.

8.1 Server-based Solution

The system running on-premises stands on three 36-core servers. These use the amount of memory and have a CPU utilization average, as in Table 8.1, where the core utilization figure results from the number of cores and CPU utilization in percentage. We thus get a total of 199 GibiBytes (GiB) of memory and 68.4 cores.

Server Number	Memory (GB)	Cores	CPU (%)	Core utilization
#1	76	36	75%	27
#2	66	36	55%	19.8
#3	57	36	60%	21.6
Average	66.33	36.00	63%	22.8
Total	199	108		68.4

Table 8.1: Server running the service.

8.2 Serverless Solution

Computations for the serverless involve more variables than the server-based solution. The variables collected are presented in Table 8.2. The Amazon Web Services (AWS) price calculator for Lambda functions requires the number of requests per unit of time, the duration of each request, allocated memory, and ephemeral storage allocation. To determine the allocated memory, we divide 76

GB by the average number of events that run simultaneously. To estimate the number of events running simultaneously, we start with a 78.41 ms average running time per event. We would have $24 * 3600/0,07841 = 1,101,900.3$ slots for events if they were completely aligned. Ideally, we would need 154.3 processors running the $1,101,900.3$ events back to back to fit the $170,000,000$ events.

Metric	Value
Requests / day	1,70E+08
Average request time (s)	0,07841
Average requests processed / core / day	1101900,268
Average message size (KBytes)	5,00E+00

Table 8.2: Metrics regarding system load.

Table 8.3 represents the different values computed for memory required per core based on the Server-based and Serverless solutions.

Item	Fewer cores	More cores
Necessary cores	68.40	154.28
Memory/core (GB)	2.91	1.29

Table 8.3: Resources necessary per core.

8.3 Cost analysis

In the cost analysis, we consider the case of a system running on-premises and a system running on a serverless approach.

8.3.1 On-Premises System

For the infrastructure values Altice Labs has provided information that allow us to compare with cloud solutions.

Price of the Infrastructure for five years (United States Dollar (USD)): 160,000

Note:

As of the second year, a 30% surcharge is applied to the cost of the infrastructure for the manufacturer's warranty.

Cost of Warrant (USD): $160,000 \times 0.30 \times 4 = 192,000.00$

Total cost for five years (USD): $160,000 + 192,000 = 352,000.00$

Total monthly cost (USD): $352,000 \div 60 = 5,866.66$

There are other expenses that we have not been able to obtain, such as the cost of maintenance, energy costs and space to have the hardware. However, the information is considerable, and conclusions can be drawn.

8.3.2 Serverless system

For the Serverless, two different approaches were considered. Aws Lambda and Aws Step Functions. Lambda functions [32] execute a function handler in response to some event. According to Amazon Web Services (AWS) [33], Step Functions is a "serverless orchestration service" based on a workflow of event-driven steps. These steps can interact with virtually any AWS or external service, for example, via a Lambda function, as in other workflow-based languages.

AWS Lambda

We resort to the figures of our infrastructure to provide all the input parameters of the AWS pricing calculator for Lambda functions [34]. We summarize their values in Table 8.4.

Table 8.4: Data used to compute the Lambda execution costs.

Item	Value
Architecture	Arm
Number of requests	$170,000,000$
Time unit	Day
Duration of each request (in ms)	78
Amount of memory allocated	1.29 & 2.91
Memory unit	GiB
Amount of ephemeral storage allocated	512
Ephemeral storage unit	MiB

The cost calculations of AWS Lambda go as follows for the 1.29 GiB case:

- Number of requests per month: $170,000,000\frac{730}{24} = 5,170,833,333.33$
- Total compute time (seconds): $5,170,833,333.33 \times 0.078 = 403,325,000.00$
- Total compute (GibiBytes second (GiBs)): $1.29 \times 403,325,000.00 = 520,289,250.00$
- Total compute with free tier discount GiBs: $520,289,250.00 - 400,000 = 519,889,250.00$
- Number of requests per month with free tier discount: $5,170,833,333.33 - 1,000,000 = 5,169,833,333.33$
- Monthly request charges (USD): $5,169,833,333.33 \times 0.0000002 = 1,033.97$
- Monthly compute charges (USD): $519,889,250 \times 0.0000133334 = 6,931.89$

- Total monthly costs (USD): $6,931.8913 + 1,033.97 = 7,965.86$

Monthly compute charges depends on the GiBs used. The more one uses, the least expensive it becomes within a limited number of tiers. In the case of our two extreme values for memory allocation, 1.29 and 2.91 GibiBytes (GiB), we are in the lowest tier, and the price is the same, 0.0000133334 USD per GiBs. Hence, the cost for 2.91 GiB in the same item is $6,931.89 \times \frac{1,173,675,750.00-400,000}{520,289,250.00-400,000} = 15643.75$, with the total cost being $15643.75 + 1,033.97 = 16,677.72$ USD per month.

AWS Step Functions

In Table 8.5, we show the input parameters for the pricing calculator for the Step Functions case. An important detail is that even though our requests take an average of 78 miliseconds (ms), AWS charges at least 100 ms. This means that the average charged duration must be above 100 ms, and the real costs will necessarily be higher. Also, the memory is rounded up by 64 MibiBytes (MiB), to 1344 and 3008 MiB. Based on these numbers, the cost calculations of AWS Step Functions go as follows for the 1.29 GiB (1344 MiB) case:

Table 8.5: Data used to compute the Step Functions execution costs. AWS charges a minimum of 100 ms per workflow.

Item	Value
Number of requests	$170,000,000$
Time unit	Day
Duration of each request (in ms)	100
Memory consumed by each workflow	1321 & 2980
Memory unit	MiB

- Number of requests per month: $170,000,000\frac{730}{24} = 5,170,833,333.33$
- Number of seconds of execution per month: $517,083,333.33$
- Total compute (MibiBytes second (MiBs)): $517,083,333.33 \times 1,344 = 694,959,999,995.52$
- Total compute (GiBs): $694,959,999,995.52/1024 = 678,671,875.00$
- Monthly request charges (USD): $5,170,833,333.33 \times 0.000001 = 5,170.83$
- Tiered prices for $3,600,000$ GiBs (USD): $3,600,000x0.0000166700 = 60.01$
- Tiered prices for $14,400,000$ GiBs (USD): $14,400,000 \times 0.0000083300 = 119.95$
- Tiered prices for $660,671,875$ GiBs (USD): $660,671,875 \times 0.0000045600 = 3,012.66$
- Note:
  $3,600,000 + 14,400,000 + 660,671,875 = 678,671,875$

- Monthly compute charges (USD): $60.01 + 119.95 + 3,012.66 = 3,192.63$
- Total monthly costs (USD): $5,170.83 + 3,192.63 = 8,363.46$

Note that the charges for computation go down as the amount of computation in GiBs increases according to the three tiers. For the 2.91 GiB case (round up to 3008 MiB), computation is similar except for the last tier where the calculation is $1,500,932,291.66 \times 0.0000045600 = 6,844.25$ USD instead of $3,012.66$. Hence, the final cost goes up by $3,831.59$ from $8,363.46$ to $12,195.05$

8.3.3 Virtual Machines

We need to estimate the costs of running hardware on the cloud. If we restrict our search to AWS, we must first select among the different categories of virtual machines currently available: "general purpose", "compute optimized", "memory optimized", "accelerated computing", and "storage optimized". We discard the latter two because the machines running on-premises do not currently need any special hardware accelerators or especially fast disk access. Among the others, the "compute optimized" machines have a balance between vCPUs and memory that is closer to our physical machines.

Using the calculator[34] with the parameters as close as possible to the physical machine, we arrive at a value of $7,378.56$ USD per month.

8.3.4 Messaging Costs

A strong candidate to replace Kafka in AWS for the messaging requirements of our architecture is Kinesis [35]. We use the numbers in the previous tables, like the number of messages per day and the size of messages. Using the AWS calculator [34], we arrive at the value of 310.10 USD.

As there are three instances, it is necessary to multiply by three, giving a monthly total of 930.30 USD.

8.4 Results

Below are summarised the results we concluded from the study done in the previous section.

- Total monthly costs of on-premises System (USD): $5,866.66$
- Total monthly costs of AWS Lambda (USD): $16,677.72 + 930.30 = 17,608.02$
- Total monthly costs of AWS Step Functions (USD): $12,195.05 + 930.30 = 13,125.35$

- Total monthly costs of Virtual Machines (USD): $7,378.56 + 930.30 = 8,308.86$

We conclude with the data that we were able to obtain that a system of this size and with these characteristics is less expensive in a solution on-premises than in the cloud. The results could be different, including energy and infrastructure maintenance costs, but it should also be noted that the prices obtained for the cloud do not include the database, which can also be a determining factor and expensive in the cloud. However, the data that has been used is relevant and gives reliability to the results.

Chapter 9

Conclusion

In this section, we can find an analysis of this internship, more concretely, how it diverged from the original plan, what was developed, what could be done better, and what can be developed in the future. Some personal thoughts on this internship will also be present.

9.1 Planned vs Real schedule

In contrast to the original timetable shown in Figure 9.1, the planned schedule diverged from the actual schedule, as seen in Figure 9.2.

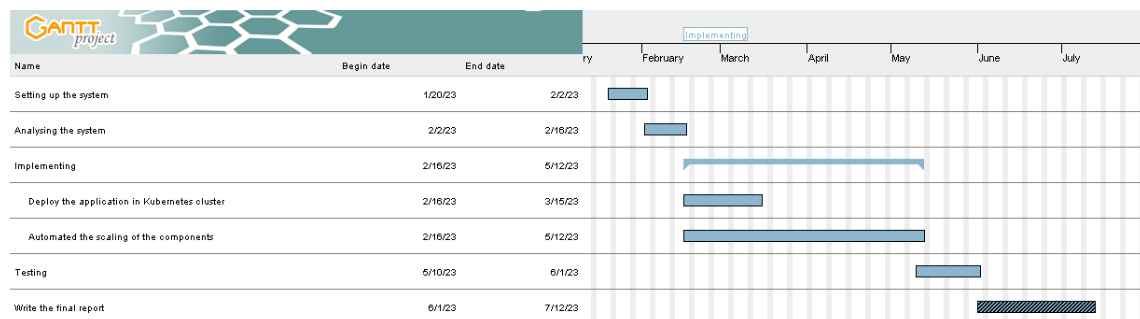

Figure 9.1: Original Schedule of the second semester

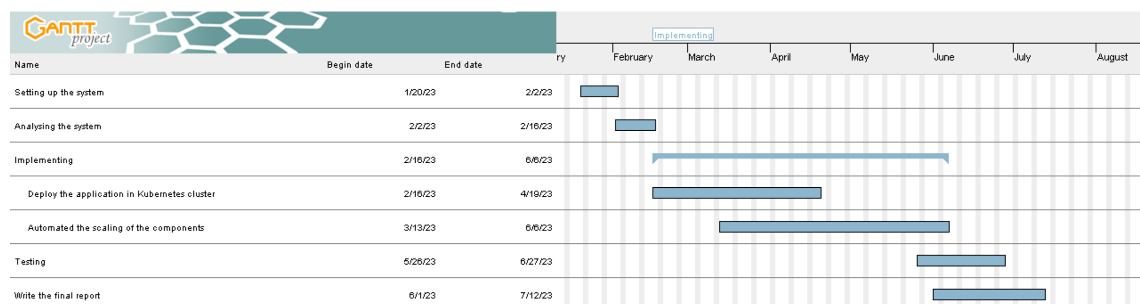

Figure 9.2: Real Schedule of the second semester

The schedule for the second semester generally followed the same order as the planned one. Firstly the preparation of the system to start the implementation went within the expected timeframe. Altice Labs already had an active cluster and the namespace created.

Secondly, the deployment of the applications in Kubernetes took longer than expected. The time set was too optimistic due to the complexity of the system combined with the inexperience of the intern. When creating a new timetable in the future, the intern should consider technologies that are new or about which he or she has little expertise. If this is the case, he should devote extra time to the anticipated issues.

Thirdly, due to the delay in deploying the applications consequently, it also delayed scaling. Furthermore, having to install Horizontal Pod Autoscaler (HPA) on the cluster was a challenge because the intern did not have the necessary permissions on the cluster to do so, causing a delay in implementation.

Fourthly some tests on the isolated applications were done to verify the correct operation while implementing scaling.

Finally, the tests on the scaling part were done simultaneously with the writing of the report.

9.2 Difficulties

This section outlines some of the challenges that slowed the project's growth and could have been avoided if the intern had more expertise.

9.2.1 Deployment of the applications in Kubernetes

Definitely, the part where the intern encountered the most difficulties. These difficulties originated due to the complexity of the system. As the different applications had to communicate with each other to work properly, there was a difficulty, for example, in connecting Kafka to Zookeeper and connecting Kafka Exporter to Kafka. The Event Stream Processor did not have much documentation about the errors, and it's an application with many peculiarities. This caused a delay in the deployment and, due to the lack of documentation, particular attention from the supervisor.

9.2.2 Scaling of the Components

Another part of this internship that took more time than expected was the scaling. The Kubernetes-based Event Driven Autoscaler (KEDA) installation took longer than expected due to permissions problems on the cluster. This problem was beyond the intern's control. We first tried to install it in such a way that permissions were not a problem because, in a real environment, the client does not

always assign administrator permissions for the entire cluster. After seeing that it was not possible and as it was not a real environment was installed with admin permissions for academic purposes.

9.3 Study of cloud costs

This work was not initially planned, but after a review with Professor Filipe Araújo, it is a work that makes sense in the context of this thesis, and thanks to Altice Labs, who provided us with data from a real environment, it was possible to perform a study with much reliability about costs in Cloud and on-premises.

For the paper, we are trying to gather even more information to draw accurate conclusions, yet it was not possible during the writing of the thesis, but it will be future work.

9.4 Future Work

In this section, the work that might be done in the future will be described. Even though this internship is finished, the concept is interesting and new experiences could be made with the system.

9.4.1 Documentation

No documentation was made while this technology was being developed. Since the intern would not have to worry about documentation, this decision was made to hasten the development. It was ok because the intern was the only one who had interaction with the system directly and learned about it as the internship went on. However, comprehensive documentation is a requirement if another person decides to work with the system because it reduces the learning curve.

9.4.2 Deployment Scaling of the Engine

Doing the Engine deployment and scaling is crucial in the future, which was not possible to do in the duration of this internship due to the complexity of the system and the fact that there were no metrics to collect. Other requirements were not implemented, which are represented in the index, because they did not make sense and affected the functioning of the system. For example, the topic of Kafka is a configuration that is made a priori that, when changed at runtime, could cause a loss of events, and all consumers had to stop consuming and make a rebalancing which could lead to a loss of performance.

9.5 Final Thoughts

To conclude, I think this internship was done successfully. The functionalities were implemented, and the system was left in a usable state. However, work can still be done in this system, as stated in section 9.3.

In technical terms, this internship allowed me to acquire knowledge in the area of Kubernetes, which is a technology that is currently widely used and requested in the real world, it can be very useful in the future. Moreover, it also helped to deepen my knowledge of Kafka, some of which I already had some kind of knowledge of due to classes, but some of them were completely new to me and might be useful in the future. In addition to all this and although I knew what Prometheus and Grafana were, I had no previous experience with it, and this internship helped me gain basic knowledge of it.

Professionally, it was a new but valuable experience, working on a project that, despite working alone, I spent the time of the internship as part of a team. This proved to be interesting, as it made me reevaluate my work ethic. Nevertheless, I know that this work experience will be valuable going forward.

References

[1] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 207–214. IEEE, 2017.

[2] Ganttproject - free project management application. `https://www.ganttproject.biz/`. (Accessed on 01/11/2023).

[3] What is scrum? | scrum.org. `https://www.scrum.org/resources/what-is-scrum`. (Accessed on 01/11/2023).

[4] Mohamed Mohamed, Robert Engel, Amit Warke, Shay Berman, and Heiko Ludwig. Extensible persistence as a service for containers. *Future Generation Computer Systems*, 97:10–20, 2019.

[5] Containers vs virtual machines | atlassian. `https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms`. (Accessed on 01/10/2023).

[6] Containers vs virtual machines | atlassian. `https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms`. (Accessed on 01/13/2023).

[7] Abdulrahman Azab. Enabling docker containers for high-performance and many-task computing. In *2017 ieee international conference on cloud engineering (ic2e)*, pages 279–285. IEEE, 2017.

[8] Docker overview | docker documentation. `https://docs.docker.com/get-started/overview/`. (Accessed on 01/13/2023).

[9] What is docker hub? | docker. `https://www.docker.com/products/docker-hub/`. (Accessed on 01/10/2023).

[10] Overview | kubernetes. `https://kubernetes.io/docs/concepts/overview/`. (Accessed on 01/10/2023).

[11] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16):4621, 2020.

[12] Kubernetes components | kubernetes. `https://kubernetes.io/docs/concepts/overview/components/`. (Accessed on 12/05/2022).

[13] Docker: Accelerated, containerized application development. `https://www.docker.com/`. (Accessed on 12/07/2022).

[14] Prometheus - monitoring system & time series database. `https://prometheus.io/`. (Accessed on 12/07/2022).

[15] Thomas Wang, Simone Ferlin, and Marco Chiesa. Predicting cpu usage for proactive autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 31–38, 2021.

[16] The guide to kubernetes vpa by example. `https://www.kubecost.com/kubernetes-autoscaling/kubernetes-vpa/`. (Accessed on 01/13/2023).

[17] Mulugeta Ayalew Tamiru, Johan Tordsson, Erik Elmroth, and Guillaume Pierre. An experimental evaluation of the kubernetes cluster autoscaler in the cloud. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 17–24. IEEE, 2020.

[18] Apache kafka. `https://kafka.apache.org/`. (Accessed on 07/07/2023).

[19] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 257–262. IEEE, 2019.

[20] Prometheus metrics: Types, capabilities, and best practices. `https://www.containiq.com/post/prometheus-metrics`. (Accessed on 01/13/2023).

[21] Keda | kubernetes event-driven autoscaling. `https://keda.sh/`. (Accessed on 01/11/2023).

[22] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[23] Brandon Thurgood and Ruth G Lennon. Cloud computing with kubernetes cluster elastic scaling. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, pages 1–7, 2019.

[24] David Balla, Csaba Simon, and Markosz Maliosz. Adaptive scaling of kubernetes pods. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2020.

[25] The c4 model for visualising software architecture. `https://c4model.com/`. (Accessed on 01/05/2023).

[26] Visual studio code - code editing. redefined. `https://code.visualstudio.com/`. (Accessed on 05/22/2023).

[27] Filezilla pro - ftp and cloud storage tool for windows, mac and linux. https://filezillapro.com/. (Accessed on 06/01/2023).

[28] Kustomize - kubernetes native configuration management. https://kustomize.io/. (Accessed on 06/01/2023).

[29] Github - danielqsj/kafka_exporter: Kafka exporter for prometheus. https://github.com/danielqsj/kafka_exporter. (Accessed on 06/05/2023).

[30] Github: Let's build from here · github. https://github.com/. (Accessed on 06/05/2023).

[31] Kafka exporter overview | grafana labs. https://grafana.com/grafana/dashboards/7589-kafka-exporter-overview/. (Accessed on 06/14/2023).

[32] What is aws lambda? - aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. (Accessed on 07/04/2023).

[33] What is aws step functions? - aws step functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. (Accessed on 07/04/2023).

[34] Aws pricing calculator. https://calculator.aws/#/. (Accessed on 07/04/2023).

[35] Amazon kinesis documentation. https://docs.aws.amazon.com/kinesis/index.html. (Accessed on 07/08/2023).

Appendices

Appendix A

Requirements not implemented

Table A.1: REQ9 - Scaling up the Kafka Topics Partitions

REQ9 - Scaling up the Kafka Topics Partitions	
Level	Sea
Actor	Kubernetes
Objective	Scaling up the number of instances of the Kafka topics partitions
Preconditions	• The number of replicas is lower than the maximum number of replicas defined
Postconditions	• The new replicas are created and start running
Main Scenario	1 - The control manager communicates the number of desired replicas to the Kube API server 2 - Kube API server communicate with Kubelet 3 - kubelet is in charge to deploy new instances
Alternative Scenario	3a. Kubelet can not deploy the pod on three consecutive tries 3a.1 launched a different pod

Table A.2: REQ10 - Scaling down the Kafka Topic partitions

REQ9 - Scaling down the Kafka Topic partitions	
Level	Sea
Actor	Kubernetes
Objective	Scaling down the number of instances of the Kafka Topic partitions
Preconditions	• The lifetime of the replicas is more significant than that defined • The metrics' values are below their thresholds • The number of replicas is greater than the minimum defined
Postconditions	• The replicas are removed
Main Scenario	1 - The control manager communicates the number of desired replicas to the Kube API server 2 - Kube API server communicates with Kubelet 3 - kubelet is in charge to remove the replicas
Alternative Scenario	3a. Kubelet can not remove the pod 3a.1 Notify the developer to solve the issue