Instituto Politécnico de Tomar

Escola Superior de Tecnologia de Tomar

João Pedro Bernardino Canoso

Orchestration of Music Emotion Recognition Services – Automating Deployment, Scaling and Management

Master’s degree Project Report

Supervised by:

Professor Renato Panda, Instituto Politécnico de Tomar

Report presented to the Polytechnic Institute of Tomar
to fulfill the mandatory requirements
to obtain the Master's degree in
Informatics Engineering - Internet of Things

“Give a man a container and you keep him busy for a day

Teach a man Kubernetes and you keep him busy for a lifetime”

Kelsey Hightower

ABSTRACT

Every day, thousands of new songs are created and distributed over the internet. These ever-increasing databases introduced the need for automatic search and organization methods, that allow users to better filter and browse such collections. However, fundamental research in the MER field is very academic, with the typical work presenting results in the form classification metrics – how good the approach worked in the tested datasets and providing access to the data and methods.

In order to overcome this problem, we built and deployed a platform to orchestrate a distributed, resilient, and scalable, music emotion recognition (MER) application using Kubernetes that can be easily expanded in the future. The solution developed is based on a proof of concept that explored the usage of containers and microservices in MER but had some gaps. We reengineered and expanded it, proposing a properly orchestrated, container-based solution, and adopting a DevOps development culture with continuous integration (CI) and continuous delivery (CD) that in an automated way, makes it easy for the different teams to focus on developing new blocks separately.

At the application level, instead of analyzing the audio signal recurring to only three audio features, the system now combines a large number of audio and lyric (text) features, explores different parts of audio (vocals, accompaniment) in segments (e.g., 30-second segments instead of the full song) and uses properly trained machine learning (ML) classifiers, a contribution by Tiago António. At the orchestration level, it uses Kubernetes with Calico as the networking plugin, providing networking for the containers and pods and Rook with Ceph for the persistent block and file storage. To allow external traffic into the cluster, will use HAproxy as an external ingress controller on an external node, with BIRD providing BGP peering with Calico, allowing the communication between the pods and the external node. ArgoCD was selected as the continuous delivery tool, constantly syncing with a git repository, and thus maintaining the state of the cluster manifests up to date, which allows totally abstracting developers from the infrastructure. A monitoring stack combining Prometheus, Alertmanager and Grafana allows the constant monitoring of running

applications and cluster status, collecting metrics that can help to understand the state of operations. The administration of the cluster can be carried out in a simplified way using Portainer. The continuous implementation pipelines run on GitHub Actions, integrating software and security tests and automatically build new versions of the containers based on tag releases and publish them on DockerHub. This implementation is fully cloud native and backed only by open source software.

Keywords: music emotion recognition, containers, orchestration, Kubernetes, DevOps, GitOps, CI/CD, continuous integration, continuous deployment, cloud native.

RESUMO

Todos os dias, milhares de novas músicas são criadas e distribuídas através da internet. Estas bases de dados musicais, em constante crescimento, introduziram a necessidade de métodos automatizados de pesquisa e organização, que permitam aos utilizadores melhor filtrar e pesquisar tais coleções. Contudo, a investigação na área do reconhecimento de emoção em música (MER) ainda é essencialmente experimental, sendo os resultados apresentados na sua maioria métricas de classificação em determinado conjunto de dados, fornecendo um conjunto de métodos.

Com o objetivo de mitigar este problema e tornar o tópico demonstrável para a população em geral, foi desenvolvida e implantada uma plataforma para orquestrar uma aplicação de MER distribuída, resiliente e escalável, que possa ser facilmente ampliada no futuro. A solução desenvolvida partiu de uma prova de conceito anterior que explorou a utilização de *containers* e microsserviços em MER, mas possuía alguns problemas. Esta foi redesenhada e ampliada, propondo uma solução de orquestração robusta, baseada em *containers* e adotando uma cultura de desenvolvimento DevOps com técnicas de integração contínua (CI), implantação contínua (CD) que, de forma automatizada, permite a diferentes equipas concentrarem-se no desenvolvimento dos microsserviços separadamente.

A nível da aplicação, em vez de se analisar o sinal áudio recorrendo apenas a três características, o sistema combina agora um grande número de características do áudio e líricas (texto), explora diferentes partes do áudio (*acappella,* acompanhamento) em segmentos (ex., segmentos de 30 segundos em vez da música completa) e utiliza diversos classificadores de *machine learning* (ML), contribuição do meu colega Tiago António. A nível de orquestração, é utilizado Kubernetes com Calico como plugin de *networking*, permitindo a ligação entre *pods* e nós, e Rook com Ceph para o armazenamento persistente de ficheiros. Para permitir a entrada de tráfego externo no *cluster*, é utilizado HAproxy como *ingress controller* num nó externo, com BIRD a fornecer BGP Peering com o Calico, permitindo assim a comunicação entre *pods* e o nó externo. Para ferramenta de implantação contínua foi escolhido o ArgoCD, estando constantemente a sincronizar com um repositório Git, mantendo assim o estado dos manifestos do cluster atualizados, o que permite abstrair

totalmente os desenvolvedores, da infraestrutura. O conjunto de ferramentas de monitorização engloba Prometheus, Alertmanager e Grafana que permitem a monitorização constante de aplicações em execução e do estado do cluster, recolhendo constantemente métricas que podem ajudar a compreender o estado das operações. A administração do *cluster* pode ser realizada de uma forma simplificada utilizando o Portainer. As pipelines de implementação contínua são executadas com recurso ao GitHub Actions, integrando testes de software e de segurança, contruindo e atualizando de forma automática novas versões dos *containers* no DockerHub, com base no lançamento de *tags*. Esta implementação é totalmente *cloud native* e suportada apenas por projetos de software de código aberto.

Palavras-chave: reconhecimento de emoção em música, containers, orquestração, Kubernetes, DevOps, GitOps, CI/CD, integração contínua, implantação contínua.

AGRADECIMENTOS

Este projeto foi o culminar de incansáveis horas dedicadas a pesquisa, investigação, testes, mas sobretudo muita perseverança, sendo sem dúvida, um dos trabalhos mais desafiantes já desenvolvidos por mim. O mérito não é apenas meu, mas de todos os que me acompanharam de perto nesta longa jornada.

Quero começar por agradecer aos meus pais, José e Rosa, irmãos, Duarte e Diogo e avós Georgete, José e Leontina todo o conforto, amor e suporte dados de forma incansável durante todos estes anos.

Um agradecimento muito especial ao Professor Renato Panda pelo acompanhamento, motivação, conhecimentos transmitidos e por sempre se encontrar disponível para ajudar e apoiar em todas as fases do projeto, sem ele, nada disto teria sido possível.

Agradeço ainda aos professores e colegas do Mestrado de Engenharia Informática do Instituto Politécnico de Tomar, em especial, ao colega Tiago António, que trabalhou lado a lado comigo nesta aventura, sempre disposto a aprender e ajudar sem qualquer hesitação.

Quero ainda agradecer a todos os meus colegas da equipa ISCP, por estarem sempre disponíveis para ajudar e esclarecer quaisquer dúvidas.

Este trabalho é dedicado a todos os meus amigos, mas em especial ao Batata, Bó, Diogo, Henrique, Miguel A., Miguel O., Pedro, Quim, Rui, Tatiana e Vanessa. Moldaram a pessoa que sou hoje e sem vocês não teria conseguido chegar tão longe.

Por fim agradeço ao Centro de Informática e Sistemas da Universidade de Coimbra (CISUC), pela disponibilização de infraestrutura computacional, e ao Centro de Investigação em Cidades Inteligentes (Ci2) do Instituto Politécnico de Tomar, financiado pela Fundação para a Ciência e a Tecnologia (UIDP/05567/2020).

A todos, o meu muito obrigado.

CONTENTS

ABSTRACT ... iii

RESUMO ... vii

AGRADECIMENTOS ... xi

CONTENTS ... xiii

LIST OF FIGURES ... xix

LIST OF TABLES ... xxiii

GLOSSARY ... xxiv

Chapter 1 Introduction ... 1

1.1. Problem and Motivation ... 2

1.2. Objectives ... 3

1.3. Initial Proof of Concept ... 3

1.4. MERmaid – A Robust and Scalable MER System ... 5

1.5. Document Outline ... 7

Chapter 2 Background and Concepts ... 9

2.1. Introduction to Music Emotion Recognition ... 10

2.2. Introduction to DevOps ... 12

2.2.1. GitOps ... 13

2.2.2. Continuous Integration and Continuous Delivery ... 13

2.2.3. DevSecOps ... 14

2.3. Virtualization Concepts ... 15

2.4. Cloud Concepts ... 16

2.5. Microservices ... 17

Chapter 3 State of the Art ... 19

3.1. Music Streaming Platforms ... 20

3.1.1. Deezer ... 20

3.1.2. SoundCloud ... 21

3.1.3. Spotify ... 22

3.1.4. YouTube Music ... 23

3.1.5. Final Considerations About Streaming Platforms ... 23

3.2. Container Orchestrators ... 24

3.2.1. Apache Mesos ... 25

3.2.2. Docker Compose ... 26

3.2.3. Docker Swarm ... 27

3.2.4. Kubernetes ... 28

3.2.5. Final Considerations About Container Orchestrators ... 31

3.3. Container Orchestration Distributions ... 32

3.3.1. K0s ... 32

3.3.2. K3s ... 32

3.3.3. Kubernetes (vanilla) ... 33

3.3.4. Microk8s ... 34

3.3.5. OKD ... 34

3.3.6. Orchestrion Solutions in the Cloud ... 35

3.3.7. Final Considerations About Containers Orchestration Distributions ... 35

3.4. Deployment Tools ... 36

3.4.1. Kubeadm ... 36

3.4.2. Kops ... 37

3.4.3. Kubespray ... 37

3.4.4. Final Considerations about Deployment Tools ... 37

Chapter 4 Technology Analysis ... 39

4.1. Container Runtime ... 40

4.1.1. Container Runtime Interface (CRI) ... 40

4.1.2. Containerd ... 41

4.1.3. CRI-O ... 41

4.1.4. Docker ... 41

4.1.5. Final Considerations About Container Runtimes ... 42

4.2. Networking ... 42

4.2.1. Networking Concepts ... 43

4.2.2. Container Networking Interface (CNI) ... 44

4.2.3. Calico ... 45

4.2.4. Flannel 46

4.2.5. Weave 46

4.2.6. Final Considerations About the Networking Plugins 46

4.2.7. Ingress 47

4.2.8. Bare Metal Considerations 48

4.2.8.1. MetalLB 49

4.2.8.2. NodePort Service 50

4.2.8.3. NodePort with external LoadBalancer 50

4.2.8.4. Host Network 51

4.2.8.5. Bare Metal Conclusions 52

4.2.9. Ingress Controller 52

4.2.9.1. NGINX Ingress Controller 54

4.2.9.2. HAproxy Ingress Controller 54

4.2.9.3. Final Considerations about Ingress Controllers 56

4.2.10. Domain Name System (DNS) 56

4.2.11. SSL Termination 57

4.3. Container Storage 59

4.3.1. Container Storage Interface (CSI) 59

4.3.2. Longhorn 60

4.3.3. Rook + Ceph 60

4.3.4. Network File System (NFS) 61
4.3.5. Final Considerations About Storage 61
4.4. Kubernetes GitOps 62
4.4.1. ArgoCD 62
4.4.2. Flux v2 62
4.4.3. Werf 63
4.4.4. Final Considerations About Kubernetes GitOps 63
4.5. GitOps Platforms 63
4.5.1. GitHub Actions 64
4.5.2. TravisCI 64
4.5.3. DevOps as a Service 65
4.5.4. Final Considerations About GitOps Platforms 65
Chapter 5 Implementation 66
5.1. Deployment Models 67
5.1.1. Clusters Tiers 69
5.2. Architecture 70
5.2.1. etcd 72
5.3. Rook + Ceph 73
5.4. Networking 74
5.4.1. External Ingress 76

5.4.2. Ingress ... 78

5.5. Portainer ... 80

5.6. Helm ... 81

5.7. ArgoCD ... 81

5.8. Observability ... 82

5.8.1. Grafana ... 84

5.8.2. Prometheus ... 84

5.8.3. Alertmanager ... 85

5.9. MER Application Development ... 85

5.9.1. GitHub Actions ... 88

5.9.2. Container Image Registry ... 90

Chapter 6 Conclusion and Future Work ... 91

6.1. Conclusion ... 91

6.2. Future Work ... 92

References ... 95

Appendix 1 Kubernetes Setup ... 101

LIST OF FIGURES

Figure 1 - General architecture of the proof of concept, adapted from (R. M. António, 2019) 4

Figure 2 - General architecture of the final solution (T. M. António, 2021) 6

Figure 3 - Typical supervised machine learning strategy applied in MER studies (R. E. S. Panda, 2019) 11

Figure 4 - DevOps cycle together with popular tools (Rodolfo Gobbi, 2019) 13

Figure 5 - Virtualization vs. containers 16

Figure 6 - Monolithic vs microservices architectures (Piotr Karwatka, 2020) 18

Figure 7 – Apache Mesos architecture (Platform9, 2017) 25

Figure 8 – Simple example of a *docker-compose.yml* file orchestrating two containers (web and redis) 26

Figure 9 – Docker Swarm architecture (Docker Swarm, 2021) 27

Figure 10 – Kubernetes architecture (Platform9, 2017) 29

Figure 11 – Kubernetes master (left) and node/worker (right) taxonomies (Vamsi Chemitiganti, 2019) 30

Figure 12 - K3s architecture (Hussein Galal, 2021) 33

Figure 13 - Interaction between kubelet and containerd using the CRI plug-in (Rosso et al., 2021) 41

Figure 14 - Interaction between kubelet and CRI-O using the CRI API (Rosso et al., 2021) 41

Figure 15 - Interaction between kubelet and the Docker Engine using dockershim (Rosso et al., 2021) 42

Figure 16 - Cluster nodes with pods CIDR 43

Figure 17 - Calico BGP peering (Rosso et al., 2021) 45

Figure 18 - The ingress resource 48

Figure 19 – Usage of a cloud LoadBalancer in a cloud environment 48

Figure 20 – MetalLB IP assignment using layer 2 (NGINX, 2021a) 49

Figure 21 – Using NodePort to access application (NGINX, 2021a) 50

Figure 22 - NodePort with external LoadBalancer (NGINX, 2021a) 51

Figure 23 – User access using host network (NGINX, 2021a) 52

Figure 24 - How the ingress controller exposes applications (NGINX, 2021b) 53

Figure 25 - Interaction between the Kubernetes cluster and the external ingress controller 55

Figure 26 - SSL termination 58

Figure 27 - SSL termination with SSL connection to the backend 58

Figure 28 - SSL passthrough 58

Figure 29 - The 4 hosts forming the cluster – 3 Kubernetes nodes and 1 edge node, all running on Xen Orchestra 71

Figure 30 - Cluster architecture 71

Figure 31 - Kubernetes highly available topology (Kubernetes, 2021) 72

Figure 32 - Rook Ceph pods allocation 74

Figure 33 - Network diagram 75

Figure 34 - Calico configuration YAML to enable BGP and set pod CIDR....................... 76

Figure 35 - Calico configuration YAML to set the AS and enable BGP peering 76

Figure 36 - BIRD configuration file 77

Figure 37 - Example of a HTTP Ingress YAML.. 78

Figure 38 - Lets Encrypt production ClusterIssuer YAML.. 79

Figure 39 - Example of an HTTPS Ingress YAML 79

Figure 40 - Sequence diagram of the cert-manager (Rosso et al., 2021) 80

Figure 41 - ArgoCD workflow (Kostis Kapelonis, 2020) 82

Figure 42 - Architecture of Prometheus and some of its ecosystem components (Prometheus, 2021) 83

Figure 43 - Sequence diagram of the build action.. 89

Figure 44 - Sequence diagram of the publish action 89

Figure 45 - Software development lifecycle, adapted from (Ando, 2020).......................... 90

LIST OF TABLES

Table 1 – Comparison of music streaming services, adapted from (Wikipedia, 2021) 24

Table 2 - Summary of basic capabilities, adapted from (Zakhar Snezhkin, 2021) 36

Table 3 - Comparing deployment tools, adapted from (Densify, 2021).............................. 38

Table 4 - Summary of the benchmark results (Ducastel, 2020) 47

GLOSSARY

AKS	*Azure Kubernetes Service*
Annotation	A key-value pair that is used to attach arbitrary non-identifying metadata to objects
API	*Application Programming Interface*
AS	*Autonomous System*
AWS	*Amazon Web Services*
BGP	*Border Gateway Protocol*
CA	*Certificate Authority*
CaaS	*Containers as a Service*
CD	*Continuous Delivery*
CI	*Continuous Integration*
CIDR	*Classless Inter-Domain Routing* – is a notation for describing blocks of IP addresses and is used heavily in various networking configurations.
CLI	*Command Line Interface*
Cluster	A set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
ClusterIP	Is an internal fixed IP that can be created in front of a pod or replica.

CNCF	*Cloud Native Computing Foundation* – builds sustainable ecosystems and fosters a community around projects that orchestrate containers as part of a microservices architecture
CNI	*Container Network Interface* – is a type of Network plugin that adheres to the app/CNI specification
Container	A lightweight and portable executable image that contains software and all of its dependencies
Controller	Controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.
CORS	*Cross-Origin Resource Sharing*
COS	*Container Orchestration Systems*
CRD	*Custom Resource Definition* – Custom code that defines a resource to add to your Kubernetes API server without building a complete custom server
CRI	*Container Runtime Interface* – API for container runtimes to integrate with *kubelet* on a node.
CRS	*Certificate Signing Request*
CSI	*Container Storage Interface* – defines a standard interface to expose storage systems to containers.
DaemonSet	Ensures a copy of a Pod is running across a set of nodes in a cluster.
DNS	*Domain Name System*
EKS	*Amazon Elastic Kubernetes Service*
GCE	*Google Compute Engine*

GKE	*Google Kubernetes Engine*
gRPC	*Google Remote Procedure Call*
GUI	*Graphical User Interface*
HA	*High Availability*
HiFi	*High Fidelity*
HTTP(S)	*Hypertext Transfer Protocol (Secure)*
IKS	*IBM Cloud Kubernetes Service*
Image	Stored instance of a Container that holds a set of software needed to run an application
Ingress	An API object that manages external access to the services in a cluster, typically HTTP
IoT	*Internet of Things*
IPAM	*IP Address Management*
IT	*Information Technology*
KaaS	*Kubernetes as a Service*
MER	*Music Emotion Recognition*
MGR	Ceph Manager
MIR	*Music Information Retrieval*
ML	*Machine Learning*

MON	Ceph Monitor
Namespace	Abstraction used by Kubernetes to support isolation of groups and resources within a single cluster
NFS	*Network File System*
NIC	*Network Interface Controller*
Node	A worker machine in Kubernetes
OS	*Operating System*
OSD	*Object Storage Daemon*
OSS	*Open Source Software*
PaaS	*Platform as a Service*
Pod	Set of running containers in the cluster
PV	*Persistent Volume* – An API object that represents a piece of storage in the cluster. Available as a general, pluggable resource that persists beyond the lifecycle of any individual Pod
PVC	*Persistent Volume Claim* – Claims storage resources defined in a PersistentVolume so that it can be mounted as a volume in a container
QA	*Quality Assurance*
ReplicaSet	Maintain a set of replica Pods running at any given time.
REST	*Representational State Transfer*
SDN	*Software Defined Network*

Service	Abstract way to expose an application running on a ser of *pods*
SLA	*Service Level Agreement*
SLO	*Service Level Objective*
SSL	*Secure Sockets Layer*
StatefulSet	Manages the deployment and scaling of a set of Pods and provides guarantees about the ordering and uniqueness of these Pods.
Storage Class	Provides a way for administrators to describe different available storage types
TLS	*Transport Layer Security*
TOR	*The Onion Router*
UI	*User Interface*
URL	*Uniform Resource Locator*
VM	*Virtual Machine*
Volume	A directory containing data, accessible to the containers in a Pod
VPC	*Virtual Private Cloud*
VPN	*Virtual Private Network*
VXLAN	*Virtual Extensible LAN*
YAML	*YAML Ain't Markup Language*

Chapter 1
Introduction

Music has been present in the daily life of our species since the beginning, used for the most diverse purposes, from entertainment to religion or war. This happens because music serves us as a universal language to communicate emotions, used across cultures and epochs (Cooke, 1959) (Pannese et al., 2016).

Nowadays music is always following us, be it in advertisement campaigns, ambient music in shopping malls, in our cars and TVs, diverse entertainment and so on. The way we access music has always changed, with players such as Spotify providing millions of songs anywhere using streaming services, e.g., Spotify catalog contained 70+ million songs in 2020, with 40 thousand new songs added daily (R. Panda et al., 2021).

With such massive music databases readily available to the user, the traditional search methods to browser for new music have become limited. Typically, users can search by artist or title, or discover new songs thanks to recommendations which are mostly based on other users' listen history and handcrafted playlists (R. Panda et al., 2021). For this reason, a new research field called Music Emotion Recognition (MER) appeared, where researchers aim to capture emotional information directly from the audio or lyrics signals. This is still an open problem, with researchers exploring different approaches interconnecting machine learning (ML), psychology, music theory and other fields.

In this chapter, we present the problem, motivation, and a condensed summary of this work, as well an outline of the dissertation. This chapter is organized as described in the following paragraphs.

Section 1.1. Problem and Motivation

This first section is used to introduce the main problem and motivation tackled in this specific work, aiming to produce a scalable, robust demonstration of a research field which is typically academic, hard to grasp by the general public.

Section 1.2. Objectives

Following the motivation, we state the objectives of the work and briefly introduce the adopted technologies.

Section 1.3. Initial Proof of Concept

Next, we describe the starting point of this work, an existing proof of concept produced that studied the feasibility of microservices for MER.

Section 1.4. MERmaid – A Robust and Scalable MER System

The fourth section introduces MERmaid, the system proposed in the scope of this work.

Section 1.5. Document Outline

Finally, we conclude the chapter by briefly detailing the structure of the remaining document.

1.1. Problem and Motivation

Every day, thousands of new songs are created and distributed over the internet. These ever-increasing databases introduced the need for automatic search and organization methods, that allow users to better filter and browse such collections. However, fundamental research in the MER field is very academic, with the typical work presenting results in the form classification metrics – how good the approach worked in the tested datasets and providing access to ML models or methods. As a result, it is hard for people outside of the field to experiment with the ideas and get a better understanding of the possibilities of such approaches. There is a lack of more applied research in the field, having an intuitive and robust MER prototype, that is able to illustrate the functionality to the general public, raising awareness to the field, and helping improve the dissemination of novel advances.

1.2. Objectives

Our main objective is to build and deploy a platform to orchestrate a distributed, resilient, and scalable, MER application that can be easily expanded in the future. Starting from a previous proof of concept that explored the usage of containers and microservices in MER, we aim to reengineer and expand it, proposing a properly orchestrated, container-based solution, with continuous integration (CI) and continuous delivery (CD) in an automated way, that makes it easy for different teams to focus on developing new blocks separately. To this end, we use state-of-the-art software engineering approaches, and adopt a DevOps development culture, which allows totally abstracting developers from the infrastructure.

1.3. Initial Proof of Concept

As previously mentioned, this project started from a proof of concept, developed by the master's student Ricardo António and the bachelor students Tiago António and Tiago Areias, and its main goal was to study the feasibility of a microservices approach to classify emotion in music, having YouTube as audio source. The project was based on 3 main microservices, to which the Frontend and the API would be added later (illustrated in Figure 1):

- Video Extractor – Microservice that takes a URL from YouTube, provided by the user, and downloads the video, automatically converts it to a music type file (.wav), saving it in a folder, so that later other microservices can access it.
- Feature Extraction – Uses digital signal processing to extract 3 audio features (characteristics that describe the audio) from the music file.
- Music Classifier – Takes the extracted features of a song and performs the classification (predicts the emotion) based on them, to this end using a previously trained ML model.

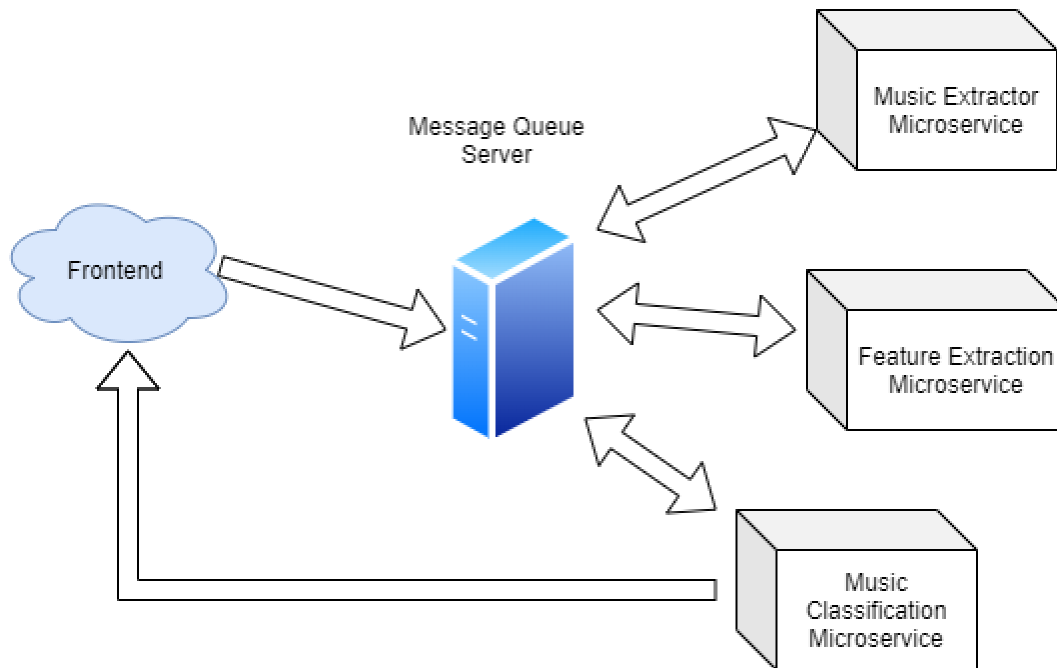

Figure 1 - General architecture of the proof of concept, adapted from (R. M. António, 2019)

The workflow of the project was developed in a simplistic but fully functional way: the user starts by entering a YouTube address and the system takes care of downloading the video, extracting its audio, classifying the sound, and returning the classification to the user, showing it on the web dashboard. The development of the proof of concept proved that it is entirely possible to use microservices to perform emotion classification in music in the idealized context, with a good degree of confidence. Still, in terms of implementation and development several problems were found that should be fixed in a newer version.

Although the development of the project was based on robust and widely used technologies such as Docker and docker-compose, its execution did not take the best path, and there is room for improvement. Some examples are:

- The usage of static IP addresses[1] in a microservices environment is not recommended at all, and in this scenario, container names should be used instead. The use of names

[1]https://github.com/mer-team/DockerMER/blob/9425eb3bed4775b217be39d69e81ee7dbced98ca/MusicClassification/musicClassificator.py#L6

allows docker to make the IP assignment dynamic, so that the destruction and later removal of a container does not cause any downtime in the system.

- The project was using a mono-repo architecture instead of poly-repo[2]. When dealing with a microservices architecture it is normal that systems become more and more complex, which makes the adoption of poly-repo more natural. With the use of poly-repo each microservice, application and library have its own repository, which makes it possible to integrate CI/CD pipelines in an easier way.
- There are no tests performed on the developed software, so it is not possible to test the microservices before building them, assuring their correct operation.
- No CI/CD automation methods has been used or implemented to build and deliver containers. Its use would allow the automatic build, distribution, and versioning of microservices, allowing the choice of versions and thus facilitating the transition between versions.
- No continuous implementation tools were used, which means that every time changes were made to the code or containers, they had to be done manually in the code describing the infrastructure, leading to possible errors and waste time.
- The technologies used were suitable for development, but the project is not fully ready for production. For example, docker-compose can only be used on a single (one) host, which is not ideal in a real-life scenario. The current project goal is to be able to run on multiple nodes (e.g., 3 or more).

1.4. MERmaid – A Robust and Scalable MER System

Based on the lessons learned from the initial proof of concept, we proposed a MER system that is robust, scalable, resilient, much more capable, and easier to improve. The system follows a similar microservices architecture but suffered a complete reorganization and expansion of each module, as illustrated in Figure 2. Instead of analyzing the audio signal recurring to only three audio features, the system now combines a large number of

[2] Mono-repo stores all the code base in a single repository while poly-repo splits each microservice in its own repository.

audio and lyric (text) features, explores different parts of audio (vocals, accompaniment) in segments (e.g., 30-second segments instead of the full song) and uses properly trained ML classifiers, a contribution by Tiago António (T. M. António, 2021).

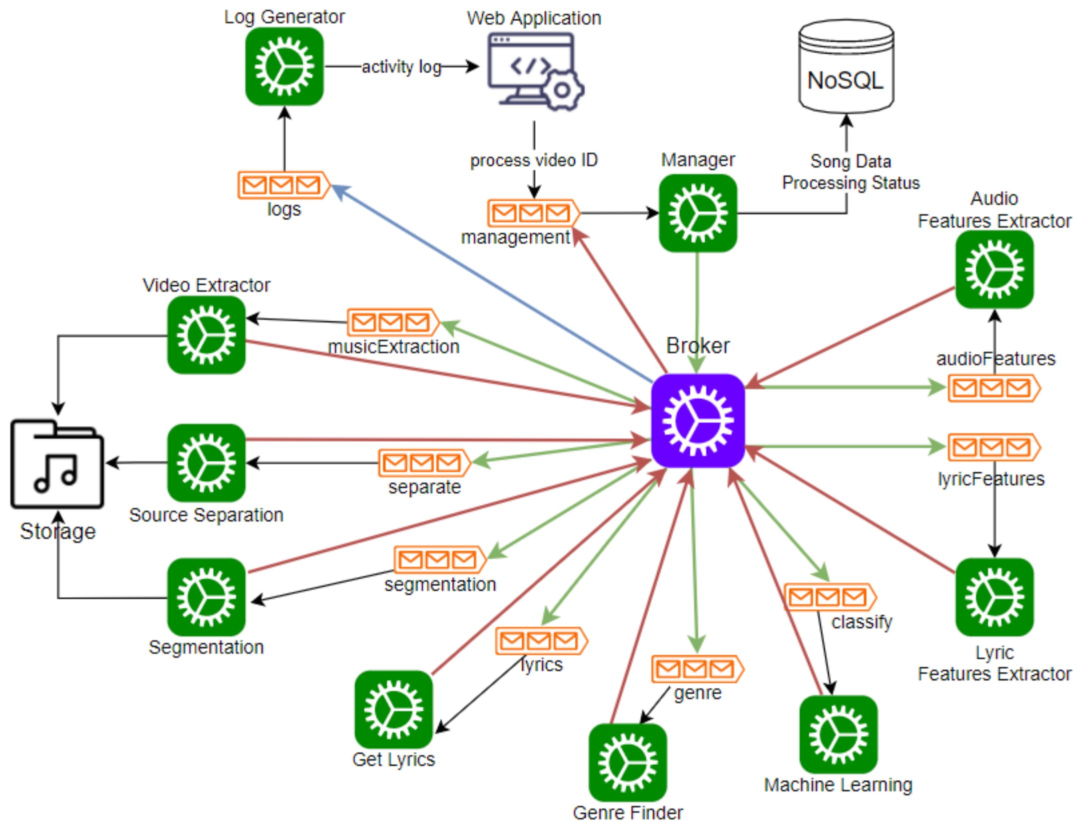

Figure 2 - General architecture of the final solution (T. M. António, 2021)

The focus of this work was the orchestration of this complex set of services, in order for it to be resilient. With this in mind, the novel solution runs on Kubernetes (k8s), the de facto orchestration platform nowadays. It is using Calico as network plugin, providing networking for containers and pods, Rook with Ceph for block and file storage, HAproxy as external Ingress, ArgoCD as the continuous delivery tool and a monitoring stack combining Prometheus, Alertmanager and Grafana. The cluster is managed using Prometheus which have a web dashboard and serves as a *kubectl* proxy. Several metrics related to the cluster and applications status are collected, helping to understand the state of the operations. Container images are built according to the addition of tags to commits on GitHub. These tags (e.g., v1.0.1) are assigned to the code of a specific microservice when one major or

stable version of the software is reached, abstracting developers from the infrastructure. In conclusion, it is now a developer-oriented system, aiming to help overcome the operations problems, and making the workflow faster and more practical. This reengineered structure, based on Kubernetes and microservices, makes the project highly scalable, easier to collaborate in, and fault tolerant.

1.5. Document Outline

This document is organized as follows.

Chapter 1 Introduction

The first chapter introduces the problem, motivation, and the objectives of this work. It also describes in a condensed format the starting point as well as the proposed solution.

Chapter 2 Background and Concepts

The second chapter helps the reader grasping foundational concepts regarding Music Emotion Recognition, DevOps, virtualization, cloud and microservices architecture.

Chapter 3 State of the Art

The third chapter scrutinizes what currently being done in the industry (and how) with respect to the music streaming area, container orchestrators and respective distributions, as well as the deployment tools.

Chapter 4 Technology Analysis

Based on the conclusions drawn from chapter three, where the Kubernetes was identified as the ideal solution to our problem, the fourth chapter presents an analysis of each of the technologies required for its proper implementation, followed by our selection given the constraints and requirements of the project. This includes the container runtimes, networking objects, container storage, Kubernetes GitOps tools and respective GitOps platforms.

Chapter 5 Implementation

The fifth chapter documents the implementation of the entire solution in a more technical way, covering the deployment models, architecture choices, storage, networking, administration, continuous deployment, observability, and the application development.

Chapter 6 Conclusion and Future Work

The sixth and last chapter presents the conclusions drawn from this work, as well as a list of aspects that can be improved and implemented in the future.

Chapter 2
Background and Concepts

This chapter allows the reader to have a better understanding of the essential concepts needed to fully understand this project. Next, a brief description of each section is provided to help the reader understand the chapter organization.

Section 2.1. Introduction to Music Emotion Recognition

Since the aim of the proposed system is to demonstrate MER concepts, we start this chapter by giving a brief introduction to the research topic that supports it.

Section 2.2 Introduction to DevOps

Next, we explain the concept of DevOps, its lifecycle and automation, that supported the entire development of the project.

Section 2.3 Virtualization Concepts

After understanding the DevOps concept, we introduce virtualization, since it is the base of today’s infrastructure, required to host any project.

Section 2.4. Cloud Concepts

Thereafter, we describe cloud computing, an essential part of modern distributed solutions, including the one followed in this work.

Section 2.5. Microservices

Going up a step, from infrastructure to software development, we describe the microservices architecture, as well as the merits of adopting one to our specific use case, as opposed to monolithic approaches.

2.1. Introduction to Music Emotion Recognition

In our newly reengineering MER system, the emotion recognition work was carried out by my colleague Tiago António as part of his MSc degree at the same institution (Polytechnic Institute of Tomar). Among others, he was responsible for the entire classification logic, studying the use of scientific validated approaches (R. Panda et al., 2020b) with industry-ready, instead of academic, audio features (R. Panda et al., 2020a). His work originated our internal microservices, and such topics are fully covered in his work (T. M. António, 2021).

In this section we present a very brief explanation of the typical machine learning approach in MER, which has three distinct parts, as illustrated in Figure 3:

- Collection of ground truth data;
- Feature extraction;
- Classification (training and testing).

The process starts with a dataset, created by collecting of a set of songs and respective labels (annotations) that best describe the emotional content of each song. The collection of labels is a complex process, usually done by a group of volunteers, being susceptible to errors that can compromise the quality of the data. Next, the audio files are processed by computer algorithms in order to extract some features that characterize them (e.g., beats per minute, the duration, or even abstract energy metrics). The extracted characteristics must be carefully selected, because using too many may result in an excess of information or introduce noise (e.g., irrelevant information to the problem), which will increase the complexity of the classifier. In the next phase, known has training, a subset of songs is used, feeding their features and respective labels to ML algorithms that will try to recognize patterns in the data (e.g., high values in feature x and y are associated with label z). At the end, we obtain our trained model, and to evaluate it, we must test it. The subset of songs from the dataset that was not previously used for training is now used for testing. To this end, the trained model assigns an emotional classification to each of them (i.e., predict labels), which are then compared to the real labels, previously identified in the dataset.

Metrics such as accuracy are extracted from this comparison, making it possible to evaluate the performance of different classifiers in order to select the best one.

In our new MER prototype, the training phase occurs offline, using both audio (R. Panda et al., 2020b) and lyrics (Malheiro et al., 2018) datasets. The obtained models are then included in the microservices and classification phase (i.e., emotion prediction) occurs in a real environment, when a user requests the classification of a song. To this end, the audio and lyrics are downloaded, their characteristics are extracted, filtered, and then provided to the trained models in order to obtain an emotional classification (T. M. António, 2021).

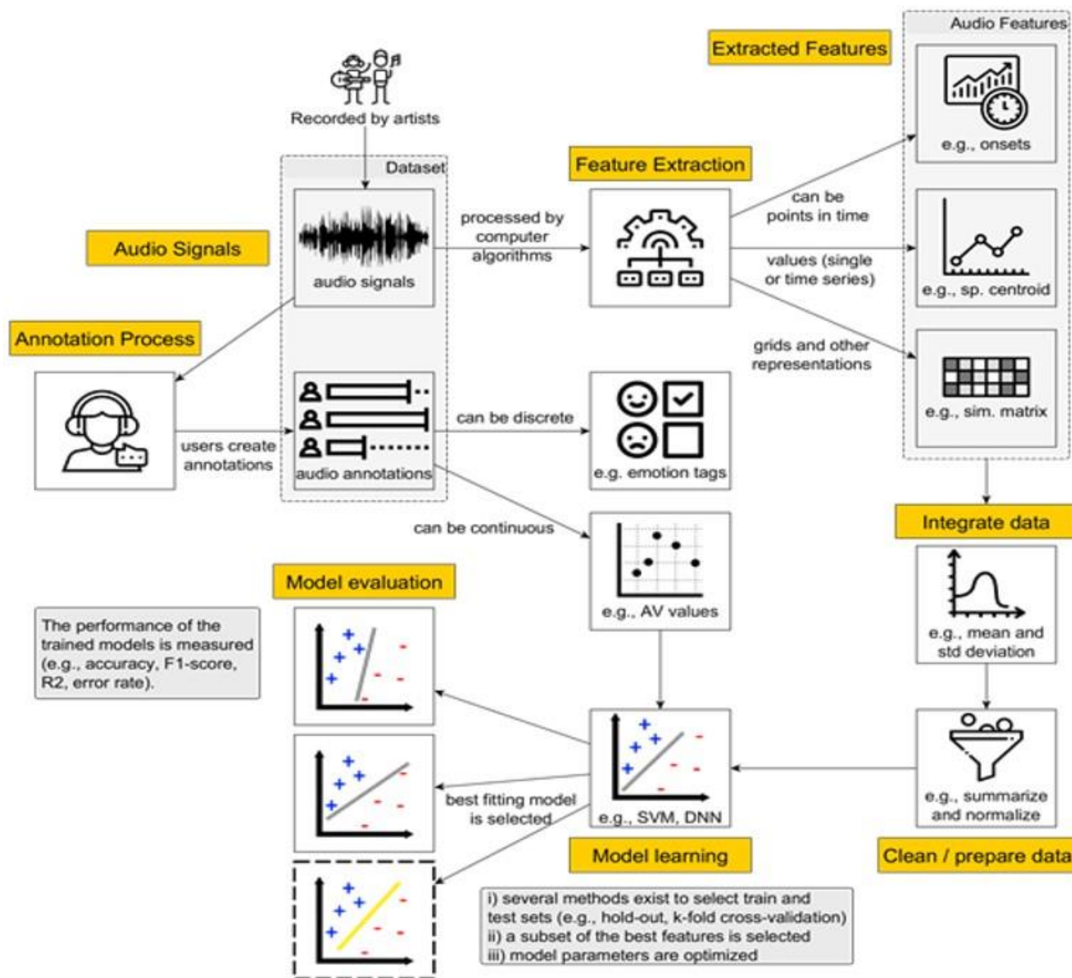

Figure 3 - Typical supervised machine learning strategy applied in MER studies (R. E. S. Panda, 2019)

2.2. Introduction to DevOps

DevOps, a blend of the term's "development" and "operations", is a combination of cultural philosophies, practices and even tools that aim to improve the speed at which a company is able to deliver applications and services. It includes changes to how individuals and teams think about their work, supports intentional processes that accelerate the rate by which businesses realize value, and measures to assess the effect of social and technical change. DevOps is about finding ways to adapt and innovate social structure, culture, and technology together in order to work more effectively (Davis & Daniels, 2016).

DevOps sets new standards for how software is built. Previously, developers, operations, and security teams typically worked in silos. Developers wrote code, Quality Assurance (QA) teams tested it, and Information Technology (IT) operations teams deployed it to production and managed the infrastructure. Security teams checked code for vulnerabilities only after the deployment. If an issue was detected, the entire process started over again, which made software development slow and frustrating for everyone involved. The introduction of a DevOps culture improves productivity by reducing manual tasks and as a result gaining consistency, reliability, and efficiency (GitHub, 2021b).

In practical terms, DevOps can be viewed as breaking the barriers between development and operations teams, by having both collaborating on the application lifecycle in whole. This can be seen as a continuous cycle, has illustrated in Figure 4, where several tools and practices are followed, automating processes, such as integrating code, testing, deploying, or infrastructure initialization.

The introduction of this set of practices in our project made our development more agile, allowing for faster development and reducing collisions within the team. Several terms and practices that under this umbrella (e.g., GitOps or continuous delivery) are explained in the following paragraphs.

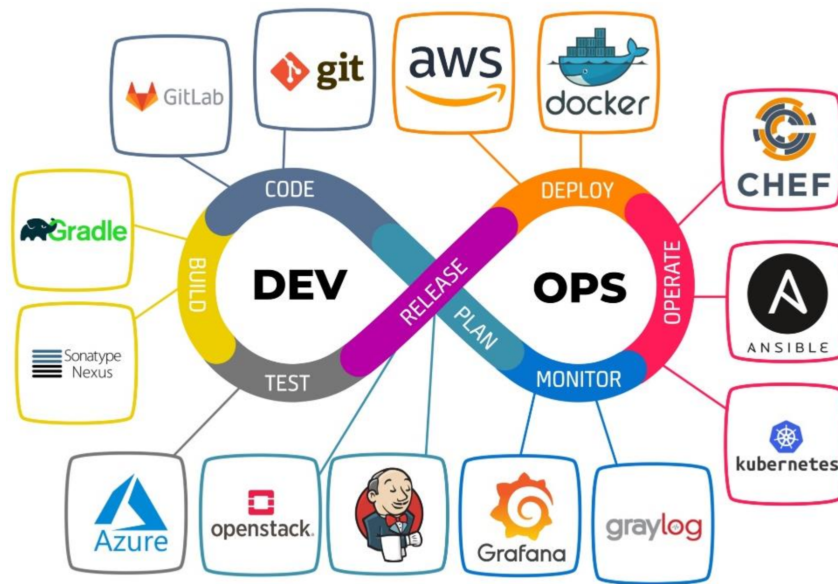

Figure 4 - DevOps cycle together with popular tools (Rodolfo Gobbi, 2019)

2.2.1. GitOps

GitOps was a term coined by Alexis Richardson to describe the idea of making operations automatic for the whole system, based on a model of the system which was living outside the system – Git in this scenario, hence the name "Git" and "Ops". It has 4 principles: 1) the entire system is described declaratively, 2) the canonical desired system state is versioned in Git, 3) must have a system to approve changes that can be automatically applied to the system and 4) must have agents to ensure correctness and alert on divergence (Weaveworks, 2021b). Our whole project relies on GitOps, because our container images are automatically tested and built from the Git repository and ArgoCD deploys and watches for changes also from Git.

2.2.2. Continuous Integration and Continuous Delivery

Continuous integration and continuous delivery (CI/CD) are a natural evolution of the DevOps transition, combining every step of the software release process into one integrated workflow. CI/CD comprises of continuous integration and continuous delivery or continuous deployment, forming a pipeline that is basically a series of automated workflows

that allows DevOps teams cut down on manual tasks. Continuous integration is responsible for automatically build, test, and integrate code changes within a shared repository, continuous delivery automatically delivers code changes to production environments for approval while continuous deployment automatically deploys code changes to the production environments. When we talk about CI/CD, the “CD” is usually *continuous delivery*. The difference between them is that, in continuous delivery automation pauses when developers push code changes to production, usually requiring one user to manually sign off before the final release, adding more delays to the process. Continuous deployment automates the entire release process, causing code changes to be deployed as soon as they pass all the required tests. The main advantage of CI/CD is the speed, because ongoing feedback allows developers to commit smaller changes more often, versus waiting for one release. Other advantages include stability and reliability because the continuous testing ensures that codebases remain stable and release-ready at any time, consequently fostering the business growth. This happens since without manual tasks, organizations can focus resources on other fronts such as innovation, customer satisfaction, or even paying down technical debt (GitHub, 2021a). CI/CD techniques were integrated into the GitHub Actions pipelines, allowing us to automatically perform tests, security audits, build container images and update them on DockerHub[3].

2.2.3. DevSecOps

DevSecOps integrates automated security testing into every part of the DevOps culture, tooling, and processes. Instead of being restricted to the end of the software development life cycle, DevSecOps security starts at the source code. Using automated security tools, developers can find and address security vulnerabilities without having to wait for security teams to address them after deployment. This allows development, operations and security teams to find and remediate security issues faster (GitHub, 2021b). In our project, this type of methodology was used in the integration of security analysis tools directly into the build

[3] https://hub.docker.com/u/merteam

pipelines. This ensures that any released image is free from known vulnerabilities. In case vulnerabilities are detected, the build pipeline fails, and information related to the error is returned to the developer to fix it.

2.3. Virtualization Concepts

Containers are technologies that allow the packaging and isolation of applications with their entire runtime environment. They are a set of one or more processes that are isolated from the rest of the system. All the files necessary to run them are provided from a distinct image, which allows to move contained application between environments (e.g., development, production, etc.) while retaining full functionality (RedHat, 2018b).

To implement containers, operating system (OS) support for control groups (cgroups) and namespaces is required. Control groups allow the OS to impose limits on the amount of resources a process can use (e.g., memory, CPU), while namespaces control what is accessible to which process (e.g., processes, network interfaces, etc.). Docker used these primitives, available under Linux since 2008, to make containers accessible to the masses, creating an abstraction that enabled developers to build and run containers in a user-friendly way. Instead of having to know the low level concepts needed to deploy container technology, all they (developers) had to do was type *docker run* in their terminal (Rosso et al., 2021).

Using containers improved many stages of the software development lifecycle, allowing developers to codify the applications environment without struggling with application dependencies. They also impacted testing, by providing totally reproducible environments for testing applications.

There is no virtualization when running containers. As can be seen in Figure 5, the main difference between virtualization and containers is that virtualization allows running multiple operating systems simultaneously on a single hardware system, while containers share the same operating system kernel and isolate the application processes from the rest of

the system. Therefore, virtualization uses a hypervisor to emulate hardware and each VM requires a full copy of the operating system, which is not as lightweight as using containers.

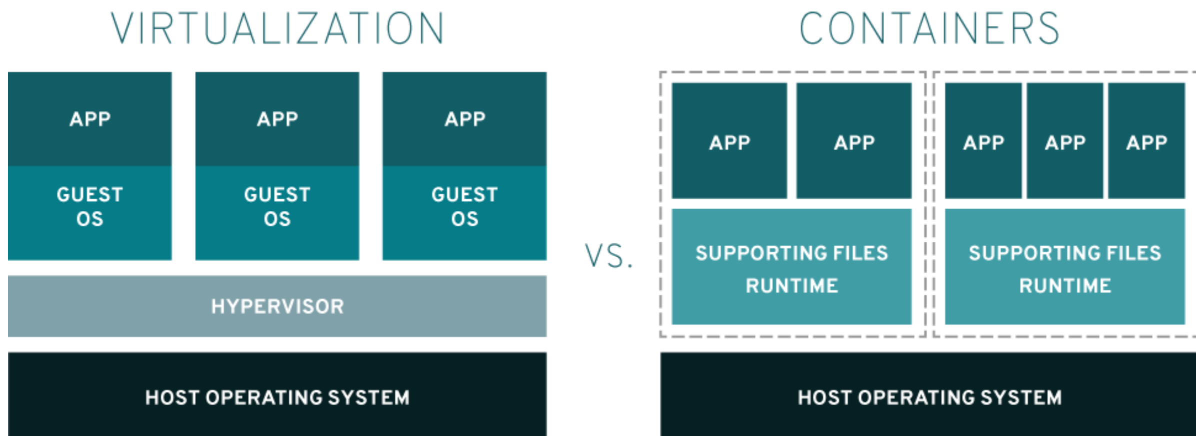

Figure 5 - Virtualization vs. containers

2.4. Cloud Concepts

Cloud computing is a type of shared internet-based computing where users can request and rent shared computing resources, offered by various cloud providers, according to their demands and needs. The main added value of using the cloud is that it spares organizations from the overhead of having to purchase, install, and maintain their own hardware and consequent infrastructure, leading to lower IT costs, improved agility and time-to-value, and increased scalability (Davis & Daniels, 2016). There are several types of cloud that can be categorized by location, ownership and more.

Public Clouds are cloud environments created from resources not owned by the end user, and that can be redistributed by multiple tenants. Usually these are automatically provisioned and allocated among multiple clients through an interface. Today's public clouds are a mix of environments that leads to higher security and performance, lower cost, and a wider availability of infrastructure, services, and applications (RedHat, 2018a). Some well-known public cloud providers are Amazon (AWS), Google (GCP) or Microsoft (Azure).

Private Clouds are cloud environments solely dedicated to the end user, usually within the user's firewall. Traditionally they run on-premises but now organizations are building private clouds on vendor-owned data centers located off-premises. This type of solution is often used when the customer cannot move to the public cloud due to security policies, budgets, compliance requirements, or regulations (RedHat, 2018a). Traditional private cloud solutions are based owning datacenters and using software such as RedHat OpenStack, Xen Orchestra, or VMware Cloud Director. Nowadays some public cloud providers also offer such, as is the case of Amazon Virtual Private Cloud service.

Hybrid Clouds are IT architectures with some degree of workload portability, orchestration, and management across two or more environments. They are based on two or more private or public cloud environments. This type of cloud used to be the result of literally connecting a private cloud environment to a public cloud, but today they are mostly built focusing on the portability of the applications that run in the environments (RedHat, 2018a).

Multiclouds are an approach made up of more than one cloud service, from more than one cloud vendor that can be public or private, referring to the presence of more than one cloud deployment of the same type (public or private), sourced from different vendors. This type of architecture brings greater flexibility, proximity to the clients and protection from outages (RedHat, 2018a).

During this project development we used a semi-public cloud running on-premises provided by the Centro de Informática e Sistemas da Universidade de Coimbra (CISUC)

2.5. Microservices

Microservices are defined as "independently releasable services that are modeled around a business domain". Services encapsulates functionalities and make them accessible to other services via the network, allowing the construction of more complex systems from these smaller building blocks. Microservices are an architecture choice that is focused on giving many options for solving problems that might arise. From the outside, a single microservice is treated as a black box. It hosts business functionality on one or more

endpoints (e.g., a queue or a REST API), over whatever protocols are most appropriate. Consumers access this functionality via these endpoints. Internal implementation details (such as the technology) are entirely hidden from the outside world (Newman, 2021).

As can be seen on Figure 6, instead of a traditional, monolithic, approach to application development, where everything is built into a single piece, microservices are all separated and work together to accomplish the same tasks. Monolithic systems are used when all functionality in a system must be deployed together, deploying all the code as a single process. It is still possible to have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process. Monolithic has some downsides: with larger applications, it becomes harder to quickly address new problems and add new features. It is also harder to scale specific components of a monolithic application. A microservice-based helps solve these issues and boost development and response (RedHat, 2018c), however it also has its downsides, the main one being the possible increase in complexity (e.g., communication, testing, debugging) that any distributed system has.

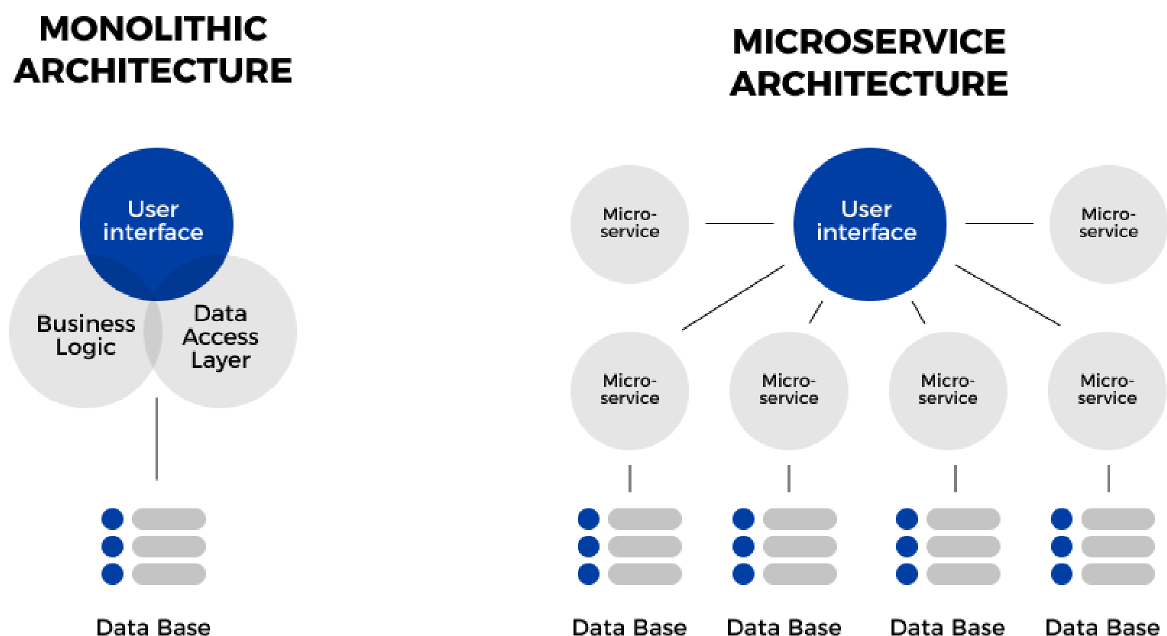

Figure 6 - Monolithic vs microservices architectures (Piotr Karwatka, 2020)

Chapter 3
State of the Art

The proposal of a proper solution, as in any other problem, requires a good understanding of what has been done in the field so far, as well as a good understanding of the available technologies that can be applied in our case. This chapter critically covers the related state of the art, that served us to plan our solution.

Section 3.1. Music Streaming Platforms

Starting by the existing music streaming platforms, we study their infrastructure implementation and usage of emotion recognition, when this kind of information is available. The goal is to understand what is currently being done, and how is it being done in the industry.

Section 3.2. Container Orchestrators

Since our goal is the creation of a distributed, resilient, modern application using microservices, we studied which tools allow us to host and orchestrate the cluster. In this section, we discuss the pros and cons of each and the reasons that led to the selection of Kubernetes.

Section 3.3. Container Orchestration Distribution

Next, we review the major container orchestration distribution options, in order to choose the appropriate one for our project.

Section 3.4. Deployment Tools

Finally, we compare the tools available to deploy a Kubernetes cluster, which led us to select the *kubeadm* command line interface (CLI).

3.1. Music Streaming Platforms

The streaming music market in Europe is valued at over $6 billion in 2019, and is projected to reach almost $10 billion by 2027, proving that there is still plenty of room for growth (Statista Research Department, 2021). The pandemic has influenced these numbers in a positive way, with a 7.4% increase in music sales, largely driven by streaming platforms (Lusa, 2021). Proving to be a very profitable market, it still does not draw full income from its raw material, and the area of Music Information Retrieval (MIR) can help in this direction with, for example, the retrieval of information of the music listened by a user and consecutive music suggestion based on its feeling. Next, we review the technological solutions adopted by the major streaming services.

3.1.1. Deezer

Deezer[4] is an audio streaming service launched in 2007 which has more than 73 million songs and more than 16 million users. It can be accessed through its web version, or mobile application available for iOS or android. The platform is available to the users through a free plan but with advertising, as well as two premium paid plans: “Premium”, that provides ads-free and offline streaming, and “HiFi”, that provides all the advantages of last one plus high-fidelity sound. All the plans provide the usual basic functionality of creating an account, search and listening to music through artists, playlists, albums, profiles, and music genres. It also provides music suggestions based on a user’s tastes, built with the music they follow and listen.

Deezer has contributed with work to the MER area, with artificial intelligence research based on audio and lyrics using deep neutral networks, such as multimodal mood prediction using audio signals and lyrics of tracks (Delbouys et al., 2018). Still, we could not find evidence of the use of such research on the Deezer platform.

[4] https://www.deezer.com/

At the operations level, it is possible to understand that Deezer uses Kubernetes (Denis Germain, 2021) and an adjacent set of technologies like Grafana, Elasticsearch, InfluxDB, Prometheus and Kafka, (Lamia Caliati, 2018) but there is no documentation available to understand to what extent.

3.1.2. SoundCloud

SoundCloud[5] is an online music distribution platform based in Berlin, used mainly by music professionals. This platform allows musicians to collaborate, share, promote and distribute their music, bringing them much closer to their listeners. The initial goal of the site was to allow industry professionals to exchange ideas about compositions they were working on, allowing easy collaboration and communication before a release, being now also used by listeners. The website allows users to upload music to the platform and then make it available for worldwide streaming. It is available in web version and mobile app for iOS and Android and counts with a free “Basic” plan, that allows upload of music up to 3 hours, and “Pro Unlimited”, that allows unlimited uploads time and allows to monetize the uploaded material. There are also two new “Go” and “Go+” plans, to compete with platforms as Spotify, that feature ad-free listening, saving tracks, offline listening, and high-quality audio for the plus plan. As it allows users to upload music, all songs are analyzed for copyrighted content using a content identification system[6], but there is no reference to the usage of any music emotion recognition system in the platform.

At the operations level, it may have started as a monolithic application developed in Ruby on Rails, using Capistrano, and Chef to describe the infrastructure, but quickly ran into several limitations such as slow scaling, instability of the deployment scheme and the time-consuming deployment of new applications. Initially, the company created its own containers managing system, “bazooka”, that was a Platform as a Service (PaaS) to solve the problems of quickly rolling out and rolling back application releases. When comparing their

[5] https://soundcloud.com/
[6] https://blog.soundcloud.com/2011/01/05/q-and-a-content-identification-system/

solution with Kubernetes, the team chose to use k8s because it had simple and understandable basic objects (e.g., containers, pods, services, among others), powerful network capabilities, a system of labels for grouping resources, and a large community behind it (Flant Blog / Sudo Null IT News, 2017). SoundCloud's use of Kubernetes dates back to 2016[7], making them one of the early users in a truly large-scale production project.

3.1.3. Spotify

Spotify[8] is the world's most popular and widely used streaming service. Like its main competitors, Spotify offers features such as creating and sharing playlists, creating an account, managing favorites, searching songs and playlists, but it stands out from the competition by being more popular and thus ensuring a large user base. There are playlists on this platform that are dynamically created and customized based on the user's tastes and based on basic moods such as "Happy" or "Sad" that are entered using metadata manually.

At the development level in the area of sentiment identification, there is research work being done with the goal of understanding user's musical tastes based on their play history (Spotify, 2016) and identifying the user's mood based on their voice and background noise (Spotify, 2018). It also provides some audio features, including arousal and valence values using their free API (R. Panda et al., 2021).

At the operations level, nowadays Spotify's implementation uses Kubernetes in the Google cloud, a service known as Google Kubernetes Engine (GKE), confirmed by the engineering team (Reddit, 2021). The company was an early adopter of microservices and Docker, having containerized microservices running in multiple Virtual Machines (VMs) and using a homegrown container orchestration system called Helios[9]. This homegrown solution fell in favor of Kubernetes, because it was much more efficient to adopt a technology that was already supported by a bigger community. Kubernetes was much more

---

[7]https://developers.soundcloud.com/blog/how-soundcloud-uses-haproxy-with-kubernetes-for-user-facing-traffic
[8] https://www.spotify.com/
[9] https://github.com/spotify/helios

feature rich than Helios and allowed Spotify to benefit from the increased velocity, reduced cost, while also aligning with the rest of the industry on best practices and tools. This migration resulted in several improvements, among which, it reduced the time to create new services from hours to minutes or even seconds, and decreased the CPU utilization, on average, two to threefold, giving the development teams more time to deliver new features for Spotify, instead of wasting time with manual capacity provisioning (Kubernetes & Spotify, 2021).

3.1.4. YouTube Music

YouTube Music[10] is one of the most recent music streaming services, having been launched in November 2015 by YouTube, with a web version and an app for iOS and Android. Built on the immense music video catalogues already available in the YouTube platform, it has the traditional functions of creating an account, creating, and sharing playlists, searching for songs, playlists, and albums, but there are also automatically generated playlists based on moods and moments (e.g., Happy, Relaxing, Training). It was not possible to find any information regarding the service orchestration. Still, being owned by Google – a major cloud provider and creator of Kubernetes, there is a very high chance of it running on such solutions.

Our MER system uses music from the YouTube platform as source of audio data, searching and extracting songs from it. This decision was taken during the initial proof of concept since it is possible to access the data without an account and there are libraries to access to it.

3.1.5. Final Considerations About Streaming Platforms

All platforms are quite similar in the functionalities offered, differing only in details such as providing lyrics, high quality audio or video. It is interesting to note that some platforms started to explore sentiment in music in a very basic way, providing playlists based

[10] https://music.youtube.com/

on sentiment, but mostly based on metadata and social information, not yet using the full potential of the data they have in hands. It is expected that in the future they will also apply techniques such as automatic extraction of sentiment, to improve music recommendation, user taste modeling, cataloguing newly added songs or generating playlists. A brief comparison of these services is presented in Table 1.

Service	*Free Access*	*# of tracks (millions)*	*# Users (millions)*	*# Paying Users (millions)*	*Release Date*
Deezer	Yes	73	16	7	2007
SoundCloud	Yes	+250	76	N.A.	August 2007
Spotify	Yes	+70	356	158	October 2008
YouTube Music	Yes	60	N.A.	30	November 2015

Table 1 – Comparison of music streaming services, adapted from (Wikipedia, 2021)

It is important to emphasize that we only considered streaming services where it was possible to obtain information about their operations infrastructure and in the three cases considered (excluding YouTube Music), all have transitioned to Kubernetes to take advantage of its velocity, reduced cost and functionalities.

3.2. Container Orchestrators

Since this is a microservices based project, the choice of the container orchestration tool has a very important weight in the development of the project. This tool will be responsible for orchestrating all containers, their number of replicas and deployments, orchestrating storage, networking with all its components (i.e., service discovery and load balancing), exposing services to the internet and managing self-healing, which is the functionality responsible for restarting failed containers and destroying unresponsive ones, launching new instances of them, not announcing them to clients until they are ready.

In the following paragraphs the main options regarding container orchestrators are analyzed in order to understand which stands out as the best choice to our problem.

3.2.1. Apache Mesos

Apache Mesos[11] is an open-source container orchestrator from Apache Foundation, built using the same principles of the Linux kernel, just at different levels of abstraction. This kernel runs on every node, providing APIs for resource management between datacenters and cloud environments, abstracting hardware from hosts, and allowing distributed and fault-tolerant systems to be easily created. Mesos is not recommended for small architectures (smaller than 20 nodes), being used for Big Data and analytics, with tools like Hadoop, Kafka, Spark, Elasticsearch, and others (Vexxhost, 2017). Mesos comes with several frameworks that consist of a scheduler and an executor, taking advantage of its resource sharing capabilities provided by its master / slave architecture. Figure 7 describes the Mesos architecture, with the master managing resources in the cluster, the slaves running agents, which report resources to the master, and the Docker executer executing tasks from the Marathon scheduler (Platform9, 2017).

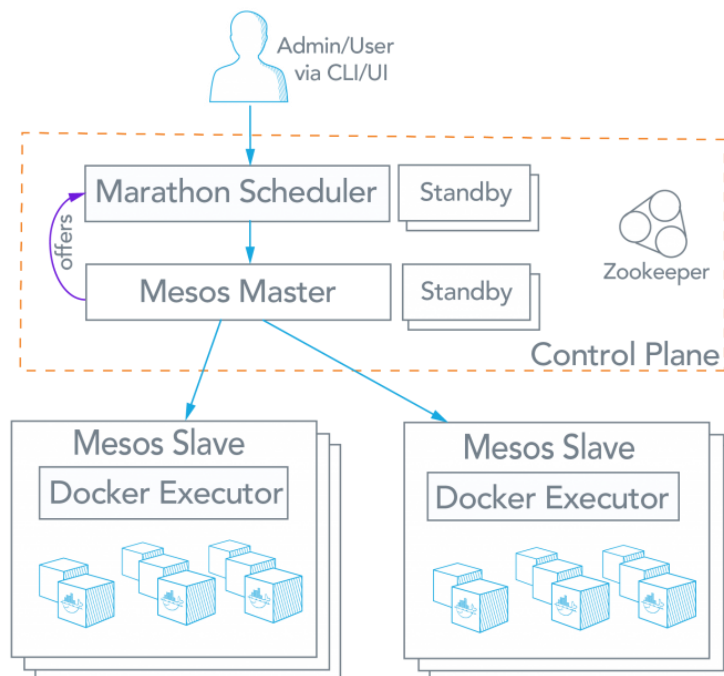

Figure 7 – Apache Mesos architecture (Platform9, 2017)

[11] http://mesos.apache.org/

3.2.2. Docker Compose

Docker Compose[12] is a tool for defining and running multiple Docker container-based applications, making it possible to run multiple services in isolated environments on a single host. YAML files, as illustrated in Figure 8, are used to configure the services, defining specifications such as service names, ports, containers, volumes, and networking. In the end, a single *docker compose up* command makes the entire application available, while Docker Compose takes care of its orchestration.

```yaml
version: "3.9" # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Figure 8 – Simple example of a *docker-compose.yml* file orchestrating two containers (web and redis)

This tool is mostly used in development environments since it can only orchestrate containers on a single host. Still, it is a valid production solution for those specific cases, and there are applications that use it as a deployment method, such as AzuraCast[13] which is a web radio management suite.

[12] https://docs.docker.com/compose/
[13] https://azuracast.com

3.2.3. Docker Swarm

Docker Swarm[14] consists of a mechanism to tie multiple hosts running Docker in swarm mode. To run this mode a user just needs to install the Docker Engine, define a node that will be the manager and join all other nodes (workers) to the swarm. In docker swarm, the running containers are known as services and it is possible to perform deployments in an imperative and declarative way using YAML, just like in Kubernetes, changing the syntax of the files / commands. The main advantages are its simple installation and configuration, and the gentle learning curve for inexperienced users in container orchestration, making it a good choice for simple projects. On the other hand, there are major limitations in terms of functionality, for example the integration of continuous delivery becomes much more complex and resource provisioning is limited. The Docker Swarm architecture is divided in master and worker nodes. The master nodes are the managers, maintaining the cluster state, scheduling services, and serving the swarm HTTP API endpoints, while the worker nodes are responsible for the execution of containers, as represented in Figure 9. The deployment of applications with multiple containers is done with Docker Compose.

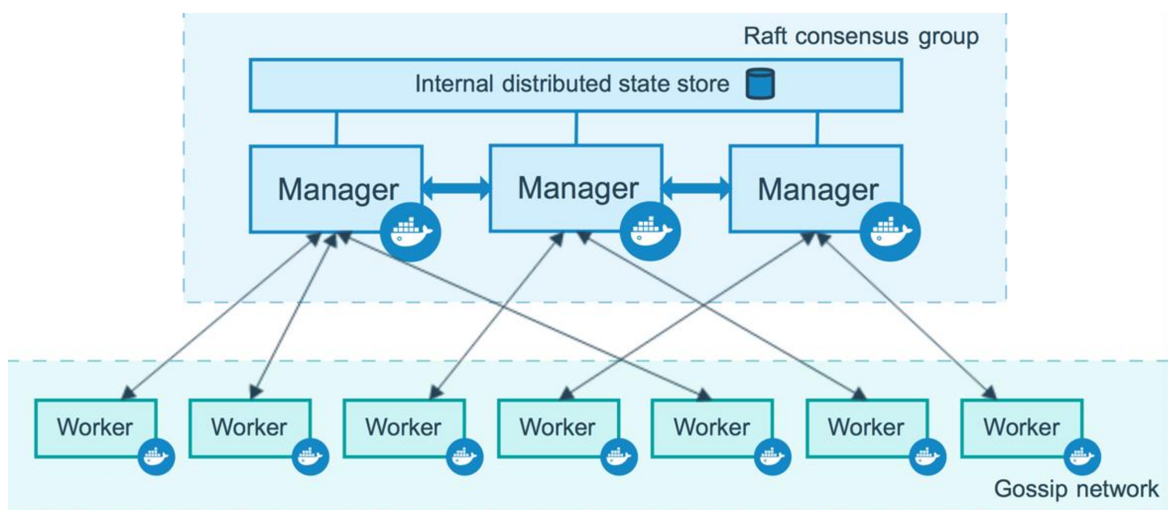

Figure 9 – Docker Swarm architecture (Docker Swarm, 2021)

[14] https://docs.docker.com/engine/swarm/

3.2.4. Kubernetes

Kubernetes[15] is an open-source, production-grade, system for automating deployment, scaling and management of containerized applications. Also called “k8s”, Kubernetes (Greek for helmsman) was originally designed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Google has been working with containers for over 10 years, being one of the market leaders and an example to follow. During their initial research, 3 systems were created to manage containers according to the available technologies. The first internal project was called Borg and had the objective of managing long-running services and batch jobs that were previously performed by different systems. A derivation of Borg called Omega was later released to improve its software engineering. Kubernetes was released later, with the goal of making it easier to manage and deploy complex distributed systems, taking advantage of the benefits that containers provide (Burns et al., 2016). Kubernetes is a very flexible and extensible platform thanks to its extensible API and the possibility of installing addons.

The most distinctive functionalities of the Kubernetes container orchestration are:

- Automated rollouts and rollbacks – allow progressively rolling out changes, while monitoring application health;
- Storage orchestration – which automatically handles the storage requests;
- Self-healing – restarts containers when they fail, replaces, and reschedules containers when nodes die, kills containers that do not respond, and does not advertise them to clients until they are ready to serve;
- Horizontal scaling – allows the application to grow, either manually, using a graphical user interface (UI), or automatically based on CPU usage.

Kubernetes is thus an advanced orchestration tool, allowing the integration of a fully cloud-native, scalable, distributed and future-proof stack that aims to reduce the burden of

---

[15] https://kubernetes.io/

orchestrating underlying compute, network, and storage infrastructure. Moreover, it enables application operators and developers to focus entirely on container-centric workflows for self-service operation, building customized workflows and higher-level automation to deploy and manage applications composed of multiple containers (Vamsi Chemitiganti, 2019).

As can be seen on Figure 10, Kubernetes follows a master / worker architecture, where the master is also called control plane. All architectures are composed of at least one master, a distributed storage system (etcd, which is explained in detail under section 5.2.1) and one or more workers.

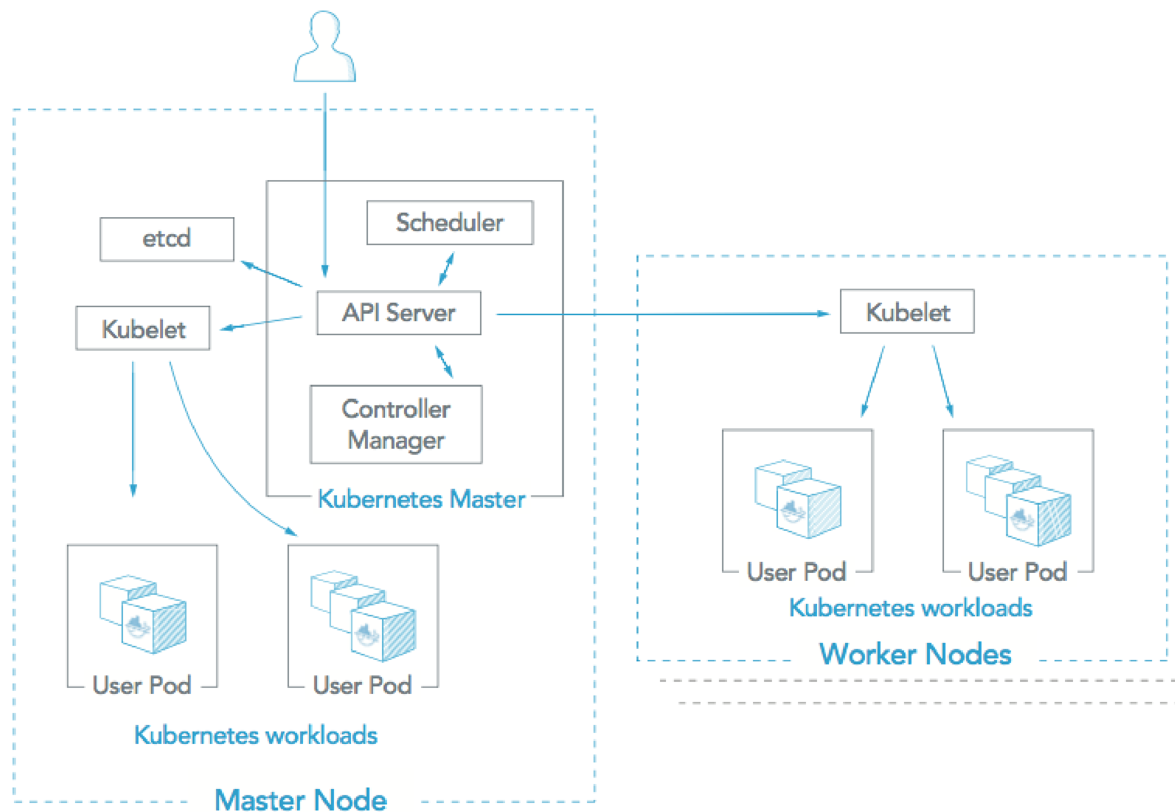

Figure 10 – Kubernetes architecture (Platform9, 2017)

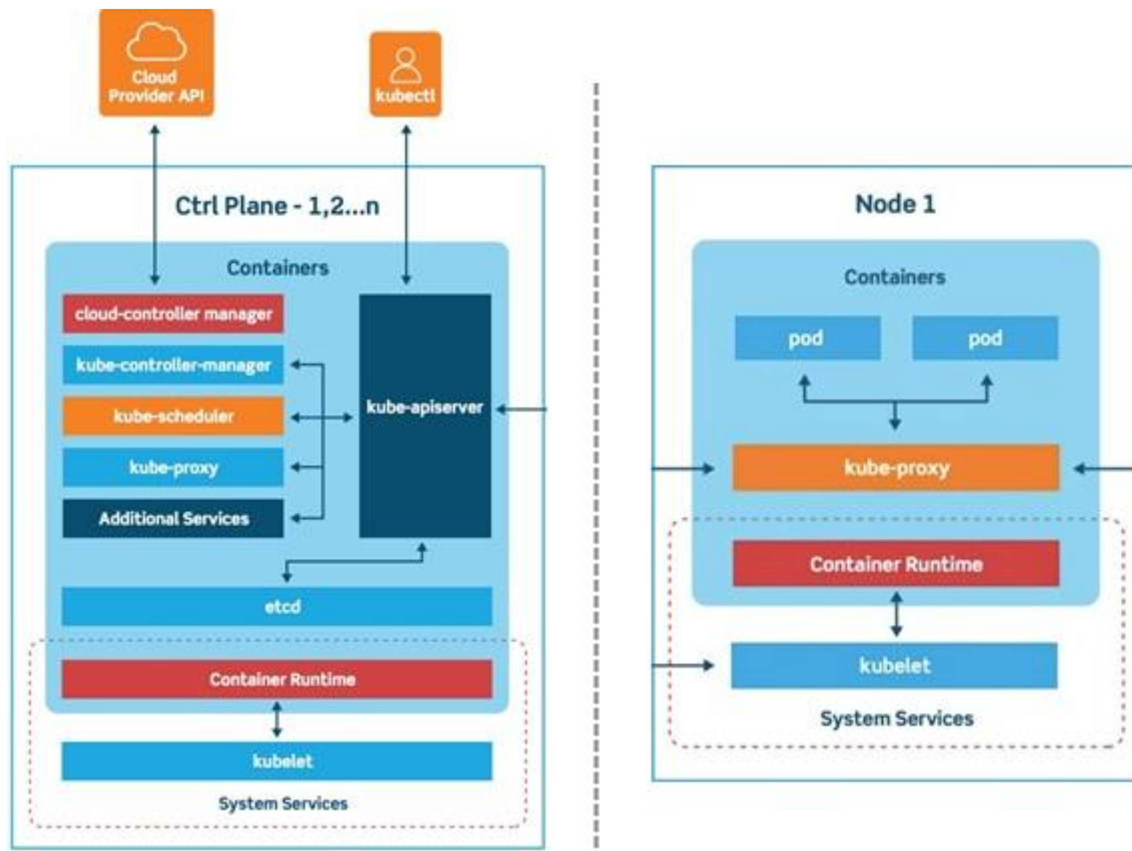

Figure 11 – Kubernetes master (left) and node/worker (right) taxonomies (Vamsi Chemitiganti, 2019)

The control plane (Figure 11, left) is responsible for maintaining a record of all objects, continuously managing states, responding to changes in the cluster, and making the actual state of system objects match the desired state. These are the major components that can run on a single master node, or replicated across multiples master nodes for high availability:

- *kube-apiserver* – component of the control plane that exposes the Kubernetes API
- *etcd* – backend store for all cluster data
- *kube-scheduler* – component of the control plane that watches for newly created pods and selects a node for them to run on.
- *kube-controller-manager* – component of the control plane that runs controller processes, watching the shared state of the cluster through the *kube-apiserver,* and making changes, moving the current state towards the desired state.

- *Cloud-controller-manager* – component of the control plane that embeds cloud-specific control logic.

The cluster worker nodes are responsible for running the workloads using containers and are managed by the master nodes. Each worker node runs a *kubelet* agent, *kube-proxy* and container runtime as shown in Figure 11 (right):

- *kubelet* – agent that runs on each node of the cluster, assuring that containers are running in a pod.
- *kube-proxy* – network proxy that runs on every node, implements part of the service concept.
- *Container runtime* – software that is responsible for running containers. This topic is addressed in Chapter 4.1.

In summary, Kubernetes is the standard for container orchestration in serious production environments nowadays, available in any major cloud provider as well as on-premises. It provides better scalability, resilience, and high availability mechanisms when compared with the alternatives. Some of its drawbacks are the added complexity and required resources, which might make it unsuitable for small projects or single node scenarios.

3.2.5. Final Considerations About Container Orchestrators

To choose our orchestrator, we had a few requirements in mind. Our application required a container orchestrator that was resilient, easy to scale, reliable and tested. Since the beginning, docker-compose was put aside because it is not production ready, and Apache Mesos because our goal is not a cluster with a big number of nodes. There is a limitation in terms of available computational resources so we opted for a solution that can be molded and configured according to our goals – Kubernetes. K8s has all the features needed, and it is a reference in the market, so its use makes perfect sense. Given our resources, Docker Swarm could also be used. However, it provides less features that were central to us (e.g., scalability, continuous delivery) and would also make it harder to make general solution that can be easily ported to other cloud providers if needed.

3.3. Container Orchestration Distributions

As explained in the last section, we have chosen Kubernetes as our platform orchestrator. However, there are many Kubernetes distributions to choose from, each one with their own pros, cons, objectives and focused on certain implementation features. Choosing the distribution is as important as choosing the orchestrator, so in this section we compare all the major distributions in order to choose thoughtfully the best one to use in our project.

3.3.1. K0s

K0s[16] is an extremely minimalist CNCF certified Kubernetes distribution, designed to be lightweight at its core. It is distributed as a single binary without requiring any dependencies other than the operating system, working on any infrastructure: public & private clouds, on-premises, edge, and hybrid. The motivation behind the project is to provide a robust and versatile base for running Kubernetes where friction is reduced to zero, allowing anyone with no special expertise in Kubernetes to easily get started. Is a very comparable distribution to k3s but it is also much more recent and so it is not advisable to use it in production.

3.3.2. K3s

K3s[17] is a Kubernetes distribution announced[18] by Rancher Labs in 2019, with the goal of enabling the creation and execution of Kubernetes clusters in environments with very limited resources. This distribution is used in the edge and IoT environments and its footprint is very reduced because all original addons and some functionalities like tags "legacy", "alpha" and "non-default" have been removed. It is a CNCF certified distro, production ready and is distributed as a single binary with 54 Megabytes (MB). As can be seen in Figure

[16] https://k0sproject.io/
[17] https://k3s.io/
[18] https://hub.packtpub.com/rancher-labs-announces-k3s-a-lightweight-distribution-of-kubernetes-to-manage-clusters-in-edge-computing-environments/

12, k3s runs as an agent or as a server. The k3s server is responsible for the control plane (Kubernetes API, controller, and scheduler), uses SQLite as default backend storage and provides reverse tunnel proxy. The k3s agent is responsible for the workloads, running the *kubelet* and *kube-proxy*, using Flannel for networking, *containerd* as container runtime and internal load balancing (Hussein Galal, 2021). K3s can be deployed in various ways and in a wide range of scenarios.

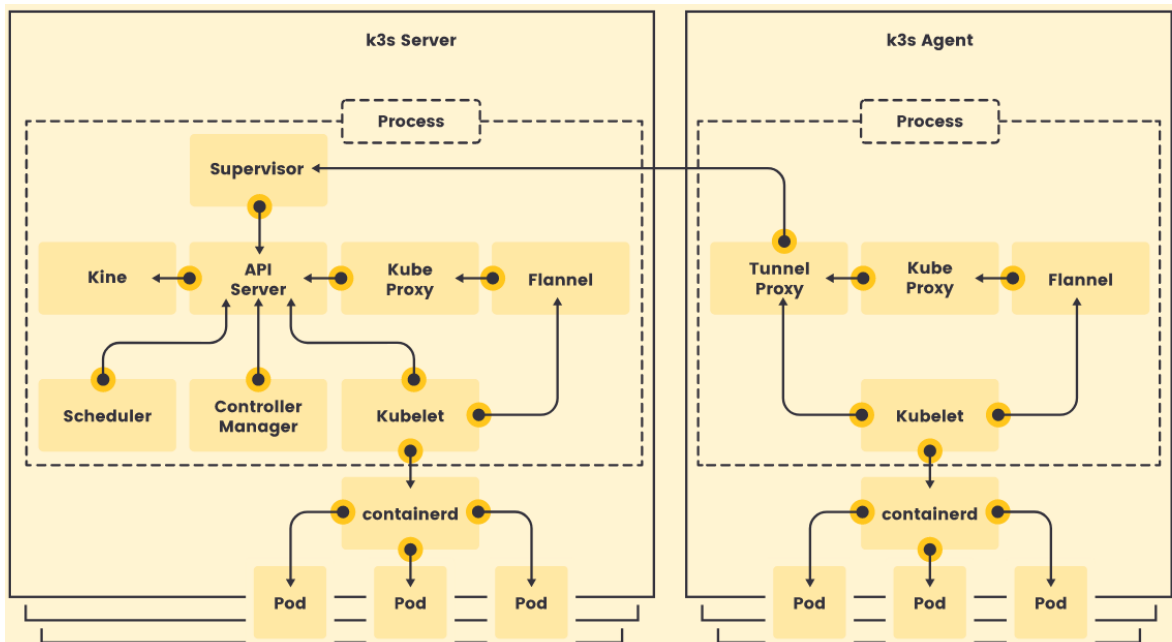

Figure 12 - K3s architecture (Hussein Galal, 2021)

3.3.3. Kubernetes (vanilla)

Kubernetes vanilla is the default k8s project that already has been covered in the Section 3.2.4 Kubernetes. It is provided as-a-service by major cloud providers, and can be installed on-premises, normally under Linux hosts, but also available to other operating systems.

3.3.4. Microk8s

Microk8s[19] is a production-grade Kubernetes distribution developed and distributed by Canonical that aims to simplify the usage of Kubernetes on public and private clouds. It is a lightweight upstream Kubernetes, with no APIs added or removed and defaults to the most used Kubernetes options (e.g., Calico for networking) with tracing, metrics, service mesh, registry, and others, without requiring any additional configuration. With support for the major operating systems, it supports both Intel and ARM architectures and so can be used anywhere, from workstations to the edge and Internet-of-Things (IoT). When multiple nodes are put together, microk8s will become automatically high available and it can provide a zero-ops experience because it does not need a user to maintain the remote nodes, with microk8s applying security updates automatically by default, and making it easy to upgrade to new versions using a single command.

3.3.5. OKD

OKD[20] is a community-maintained Kubernetes distribution optimized for continuous application development and multi-tenant deployments. OKD natively features multiple fully open-source tools that help development and operations teams rapidly develop, deploy, and scale applications to increase application lifecycle (OKD, 2021).

The main reasons we did not choose OKD as an orchestration solution were the limited supply of operating systems that allow its installation, the high minimum requirements for the cluster architecture, and the fact that, since it is a ready-to-use solution, it comes with built-in tools, many of them geared towards the operations teams, that we did not intend to use and that would therefore unnecessarily consume resources, reducing our flexibility, such as the container registry and Jenkins.

[19] https://microk8s.io/
[20] https://www.okd.io/

3.3.6. Orchestrion Solutions in the Cloud

With the digital transition and the transfer of services to the cloud, Kubernetes services have begun to emerge as CaaS (Container-as-a-Service), removing the complex tasks of installing and maintaining clusters from the equation. These types of services are available from major cloud providers such as Google under the name GKE (Google Kubernetes Engine), Amazon under the name EKS (Amazon Elastic Kubernetes Service), Azure under the name AKS (Azure Kubernetes Service) and IBM under the name IKS (IBM Cloud Kubernetes Service). Since they are cloud services, they have payment plans depending on usage and being self-managed solutions, there are always extra costs besides the hosting, which is often not included. This type of solution can also become limiting in cases of specific implementations.

3.3.7. Final Considerations About Containers Orchestration Distributions

There is a wide variety of container orchestration distributions these days. We are looking for a distribution that is fully production-ready and therefore well tested, so that if there are problems, they are relatively easy to find and fix. This first requirement eliminates k0s, since it is production ready only after November 2020. As our project can rely on a supply of infrastructure resources, the use of cloud solutions is out of the equation, but there are limitations in terms of available resources[21], so we had to opt for a more modest solution that we can mold and configure according to our goals. In terms of performance comparison, microk8s shows a higher resource usage, while Kubernetes (vanilla) and k3s are side by side, with only small differences on applying deployments and draining workers, where k8s wins (Böhm & Wirtz, 2021). Considering k3s and Kubernetes (vanilla), we choose the later because it has all the features we need and is a reference in the market. Furthermore, k3s does not count with the full Kubernetes API, which could limit the choices in terms of addons

[21] Initially 8 vCPUs, 8 GB RAM and 150 GB storage, later increased to 18, 18 and 300 respectively.

for our project in the future. Table 2 summarizes this comparison and further supports our choice.

	Orchestration Distribution			
Feature	**K0s**	**MicroK8s**	**k3s**	**k8s (vanilla)**
Managing nodes Creation / deletion	No	No	Yes	Yes (kubeadm)
Node management system	No	No	Docker	Any
Container runtime	containerd	containerd	CRI-O	Any
Default CNI	Calico	Calico	Flannel	Any
Addons	No	Yes	No	No
Vanilla Kubernetes	Yes	Yes	No	Yes

Table 2 - Summary of basic capabilities, adapted from (Zakhar Snezhkin, 2021)

3.4. Deployment Tools

In order to facilitate the process of making a Kubernetes cluster available, there are several tools that allow not only the deployment of a cluster on multiple machines, but also the deployment of whole infrastructures, each one with its own objectives. The tools that we are going to review allow the easy and fast bootstrapping of clusters.

3.4.1. Kubeadm

Kubeadm[22] is a tool that can be used to build viable Kubernetes clusters providing best practices. It allows users to setup a minimum viable, secure, cluster in a user-friendly way. This is one of the simplest tools and was designed to simplify the bootstrapping and installation of clusters. Since it has no infrastructure dependencies, Kubeadm is the best choice for bare-metal installations. At the infrastructure level, the servers should already be provisioned, then Kubeadm will run on each node, staying in the middle of the stack, creating it and then talking to the Kubernetes API.

[22] https://github.com/kubernetes/kubeadm

3.4.2. Kops

Kops[23] stands for Kubernetes operations and allow the users to create, destroy, upgrade, and maintain production-grade, highly available Kubernetes clusters while also provision the necessary cloud infrastructure. Amazon Web Services (AWS) is officially supported, with DigitalOcean, Google Compute Engine (GCE) and Openstack in beta support. Its main features are the automation of provisioning of highly available clusters, as well as the ability to generate Terraform and zero-config addons.

3.4.3. Kubespray

Kubespray[24] is a community project designed to deploy Kubernetes clusters in the cloud or on premises. Can be deployed on the major cloud providers like AWS, GCE, Azure, or bare metal, with the option to create high available clusters. It was originally based on Ansible playbooks but now has started using Kubeadm internally for cluster creation in order to consume its life cycle management domain knowledge.

3.4.4. Final Considerations about Deployment Tools

The list of tools that allow a user to deploy a Kubernetes cluster is not very extensive, but each tool is focused on its own goals which makes the tool choice easier. When comparing the two tools that can implement infrastructure: Kubespray and Kops, the use of Kubespray is more appealing due to its multi-cloud and bare metal support. When comparing Kubespray with Kubeadm and considering our scenario, the usage of Kubeadm makes more sense because it is simpler, thus reducing the tool complexity, and because our deployment does not require the infrastructure creation present on Kubespray. The complete comparison between the different deployment tools can be seen in Table 3.

[23] https://kops.sigs.k8s.io/
[24] https://kubespray.io/

Tool	*Infrastructure Creation*	*Production Ready Cluster*	*Lightweight*	*Manage Cluster Lifecycle*
Kubeadm	No	Yes	Yes	Yes
Kops	Yes	Yes	No	Yes
Kubespray	Yes	Yes	No	Yes

Table 3 - Comparing deployment tools, adapted from (Densify, 2021)

Chapter 4
Technology Analysis

In the previous chapter, we studied the solutions adopted by music streaming platforms, as well as different container orchestration technologies from a more theoretical perspective. Based on this, Kubernetes was selected as the orchestration solution for our system. The adoption of Kubernetes continues to increase, with its use in production rising from 59% in 2020 to 65% in 2021 (VMware Inc., 2021). However, its installation, configuration and maintenance are not simple, requiring specialized knowledge, causing many companies to look for production-ready solutions without major configurations, such as CaaS. In this chapter, we delve into the more practical questions the arise from our previous choice, i.e., Kubernetes (vanilla) on premises, analyzing the technologies that are part of a Kubernetes solution, and justifying the ones we adopt to solve our specific needs.

Section 4.1. Container Runtime

With that in mind, in this first section we explore the available container runtimes, responsible for managing the container's execution, in order to select one (CRI-O).

Section 4.2. Networking

Next, we analyze the networking options, an essential part of Kubernetes, which will control not only how the traffic flows across nodes, into containers and pods, but also the access from and to the outside world.

Section 4.3. Container Storage

Containers are ephemeral in its nature, made to be short-lived, restarting in a different node as needed. However, several parts of our system require persistent storage, which must survive container destructions and be accessible across nodes. To address this, different container storage solutions were tested, with the results described in this section.

Section 4.4. Kubernetes GitOps

To automate the deployment of applications, there must exist a cluster-wide tool in the target environments that pulls the changes from a remote repository and applies them in the cluster. In this chapter we will study the existing tools and explain why we chose ArgoCD.

Section 4.5. GitOps Platforms

For that continuous delivery tool to work, we based our project on GitHub Actions, that not only allowed us to keep a record of changes of the project, but also describe our entire deployment in a declaratively form. With its unique features, we built workflows to automatically execute software and security tests, build docker images of the application microservices and update them with the new versions of the software.

4.1. Container Runtime

Kubernetes requires that a container runtime be installed on each node of the cluster to be able to run pods. Container runtimes are software that creates and manages containers on a node. On Linux systems, containers use kernel primitives like control groups (cgroups) and namespaces to launch isolated processes from images. Kubernetes has a tool (kubelet) that interacts with the container runtime to launch containers (Rosso et al., 2021). Since we are using Kubernetes vanilla, there are essentially 3 different types of container runtimes to choose from. These do not come installed by default and therefore one needs to be selected.

4.1.1. Container Runtime Interface (CRI)

CRI is the bridge between kubelet and the container runtime and was introduced in version 1.5 to foster the growth of container runtimes. Prior to this, adding support for a new container runtime required releasing a new version of Kubernetes and consequently required a big knowledge of its code base. CRI allows developers of container runtimes to abstract

away from the implementation details in Kubernetes, namely *kubelet*, allowing easier integrations, specifying the interface that the container runtime must implement to be compatible with Kubernetes.

4.1.2. Containerd

Containerd is the most common runtime container, being used in several managed Kubernetes offerings such as AKS, EKS, GKE, among others. This runtime container implements the CRI through a native plugin that comes active by default, exposing the Google Remote Procedure Call (gRPC) API on a Unix socket.

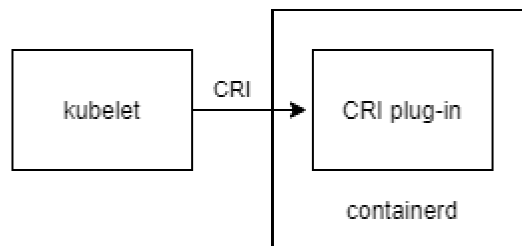

Figure 13 - Interaction between kubelet and containerd using the CRI plug-in (Rosso et al., 2021)

4.1.3. CRI-O

CRI-O is a container runtime specifically designed for Kubernetes, being an implementation of CRI, and thus it has no use beyond Kubernetes. This runtime is used by RedHat in OpenShift and like containerd, exposes the CRI on a Linux socket.

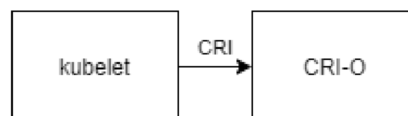

Figure 14 - Interaction between kubelet and CRI-O using the CRI API (Rosso et al., 2021)

4.1.4. Docker

Docker engine is available as a container runtime using CRI shim, called dockershim. This component is present in kubelet and is essentially a gRPC server that implements CRI

services. This extra component is necessary because Docker Engine does not implement CRI directly.

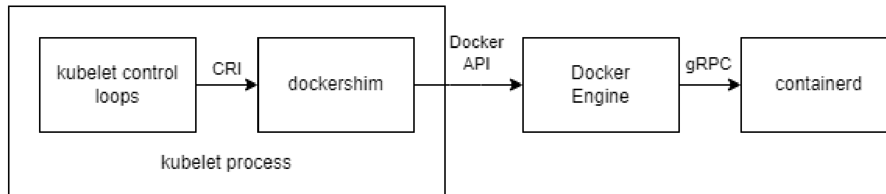

Figure 15 - Interaction between kubelet and the Docker Engine using dockershim (Rosso et al., 2021)

4.1.5. Final Considerations About Container Runtimes

Container runtimes are a fundamental component of any Kubernetes implementation as it is impossible to run containers without installing them. In addition to these solutions, there are Kata Containers that allow a much better level of isolation between containers but using virtual machines to achieve it, so this solution was not considered. From the abovementioned three, we chose to install CRIO-O because it is used in the OpenShift project, is specially designed for Kubernetes use, being therefore a simpler solution, and there are no downsides when compared to the other options.

4.2. Networking

Fundamentally speaking, networking is one of the core components of Kubernetes since it is what allows nodes, services, pods, and the internet to communicate. Software defined networks (SDNs) have been widely used to solve problems like propagation of known routes, routing of packets, and uniquely addressing hosts on cloud ecosystems like Amazon VPCs (Virtual Private Cloud). As a result, they are also being used on Kubernetes to solve the same kind of issues but with pods instead of hosts. This section will explore the concept of networking and the challenges faced in Kubernetes

To make it possible for the pods to communicate, they must be uniquely addressable, and so, each one receives an IP address that may be internal to the cluster or externally routable. Since pods are ephemeral, they can be destroyed, restarted, or rescheduled, to

match the desired state of the cluster or to recover in case of system failure. This requires the IP address distribution and allocation to be quickly and efficient. The IP Address Management (IPAM) is responsible for this management and is implemented based on the chosen Container Network Interface (CNI) plugin. During our initial cluster provisioning, the pod network's Classless Inter-Domain Routing (CIDR) has been defined on the *kubeadm* as a flag: *sudo kubeadm init --pod-network-cidr=10.148.0.0/16*. Kubernetes will allocate a subnet to each node, using by default a /24 that can be automatically adapted in case of need.

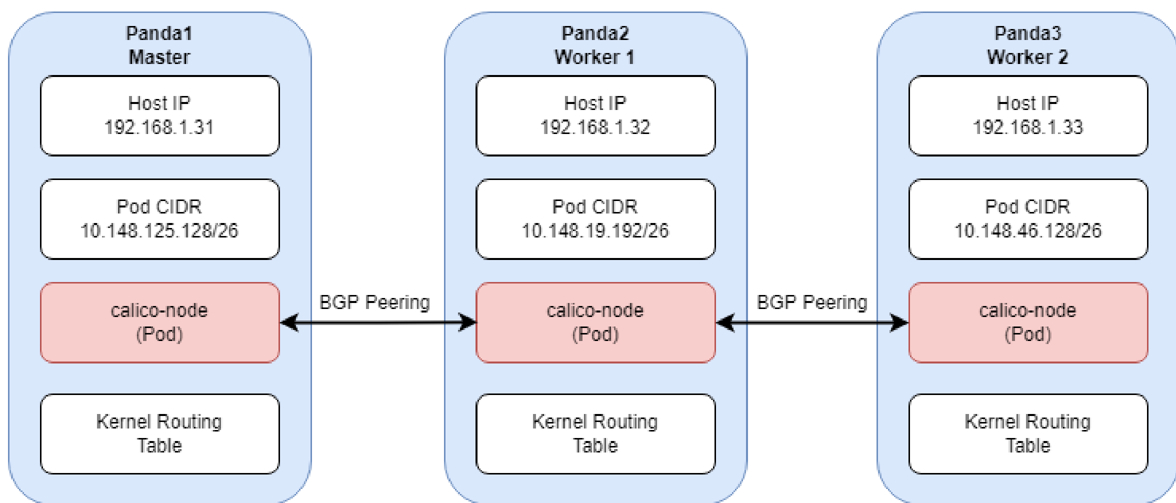

Figure 16 - Cluster nodes with pods CIDR

4.2.1. Networking Concepts

To understand the paradox that we had to face in making services available outside of the cluster, there are some basic concepts that the reader must be aware of. First it is important to remember that each pod has its own IP address. However, the set of pods running in one moment in time could be different from the set of pods running a moment later and when a pod needs to communicate with another, it needs to know its IP address. *Services* in Kubernetes allow the location of other pods by defining a logical set of pods and

a policy by which to access them[25]. They offer load balancing across multiple pods in the layers 3 and 4 of the OSI model. There are different types of services that essentially change the way traffic is routed:

- *ClusterIP* – Is an internal fixed IP that can be created in front of a pod or replica. The ClusterIP provides a load-balanced IP address that then forwards the traffic to the pods. The ClusterIP IP address is not routable outside of the cluster.
- *NodePort* – NodePorts are built on top of ClusterIP by exposing the ClusterIP service outside of the cluster, but only on high ports (30000 - 32767). When a specific port is not defined, Kubernetes automatically chooses a free port.
- *LoadBalancer* – Exposes the service externally using a cloud provider's load balancer implementation. Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

4.2.2. Container Networking Interface (CNI)

The Container Networking Interface (CNI) is a project of the CNCF providing a specification and set of libraries for writing plugins that configure network interfaces on Linux containers. The sole purpose of the CNI is to ensure connectivity of containers and to remove allocated resources when they are deleted. To choose the right CNI plugin there are a set of parameters that we must take into account, such as the number of pods in each node, IPv6 usage, network policies usage, encapsulation usage, among others. CNI has a wide range of options[26], and we will only cover the most robust and relevant implementations.

---

[25] https://kubernetes.io/docs/concepts/services-networking/service/
[26]https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model

4.2.3. Calico

Calico[27] is an open-source networking and security solution for containers, virtual machines, and workloads on native hosts. Calico supports a multitude of platforms including Kubernetes, OpenShift, Docker EE, OpenStack, and bare metal services. There is also a paid version with support called Tigera[28]. The main advantages of calico are its high performance, flexibility and reliability, reasons that make it one of the most popular network interface controllers (NICs).

In technical terms, calico operates at layer 3, using Border Gateway Protocol (BGP) to propagate routes between nodes, offering integration with datacenter fabric. Using BGP facilitates packet forwarding, since there is no additional encapsulation required, and it is simpler and more optimized than VXLAN. On each node runs a calico-node agent that uses BIRD for BGP peering, and Felix that puts known routes into the kernel routing tables, as illustrated in Figure 17. Calico also allows the usage of network policies.

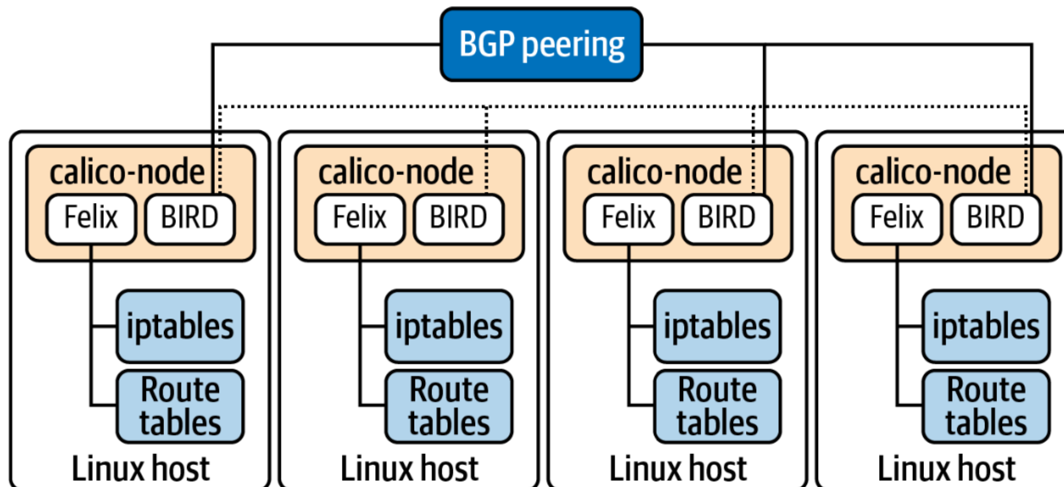

Figure 17 - Calico BGP peering (Rosso et al., 2021)

[27] https://docs.projectcalico.org/about/about-calico
[28] https://www.tigera.io/

4.2.4. Flannel

Flannel[29] was one of the first implementations to use CNI, being one of the oldest and most mature. Flannel implements networking at layer 3 of the OSI model, creating an internal network including all nodes, a topology known as fabric, connecting all nodes to each other in the cluster. Virtual Extensible LAN (VXLAN) is used as default backend, but it is possible to use UDP. The main disadvantage of the flannel is its lack of features like network policies and firewall (Platform9, 2021).

4.2.5. Weave

Weave[30] is a networking plugin developed by Weaveworks[31], totally compatible with CNI, which counts also with a paid version. To enable traffic exchange, weave creates a mesh network between all nodes in the cluster, using a shortest path algorithm, i.e., fast datapath (Weaveworks, 2021a), to route it. The network traffic is constantly being analyzed for route optimization. This type of solution can generate implementation problems for clusters with a very large topology.

4.2.6. Final Considerations About the Networking Plugins

As one of the methods of determining which CNI is best suited for our implementation, in addition to studying existing projects, we investigated which solutions are used by cloud providers. One of the great values of using cloud services is that when a Kubernetes cluster is provisioned, many elements work without any extra configuration, among them networking. This research was not very revealing because each cloud provider supports

[29] https://github.com/flannel-io/flannel
[30] https://github.com/weaveworks/weave
[31] https://www.weave.works/

networking by implementing its version of a CNI (e.g., Amazon CNI[32]). We then considered the analysis by (Ducastel, 2020), summarized in Table 4, which shows that Calico, Canal and Flannel are among the fastest CNI. We chose to use Calico because it is a mature plugin that is quite old in the community and has better performance, without cutting down on features such as network policies or encryption that may be useful in the future.

CNI Benchmark August 2020 infraBuilder	Config	Performances (bandwidth)				Resources consumption (cpu/ram)					Security features			
	MTU	Pod to Pod		Pod to Service		Idle	Pod to Pod		Pod to Service		Network Policies		Encryption	
	setting	TCP	UDP	TCP	UDP	none	TCP	UDP	TCP	UDP	in	out	activation	Performance
Antrea	auto	Very fast	Very fast	Very fast	Slow	Low	Low	Low	Low	Low	yes	yes	at deploy time	Slow
Calico	manual	Very fast	Very fast	Very fast	Fast	Low	Very low	Very low	Very low	Very low	yes	yes	anytime	Very fast
Canal	manual	Very fast	Very fast	Very fast	Very fast	Low	Very low	Very low	Very low	Very low	yes	yes	no	n/a
Cilium	auto	Fast	Very fast	Very fast	Very fast	High	High	High	High	High	yes	yes	at deploy time	Slow
Flannel	auto	Very fast	Very fast	Very fast	Very fast	Very low	Very low	Very low	Very low	Very low	no	no	no	n/a
Kube-OVN	auto	Fast	Very slow	Fast	Very slow	High	High	High	High	High	yes	yes	no	n/a
Kube-router	none	Slow	Very slow	Slow	Very slow	Low	Very low	Low	Very low	Low	yes	yes	no	n/a
Weave Net	manual	Very fast	Very fast	Very fast	Fast	Very low	Low	Low	Low	Low	yes	yes	at deploy time	Slow

Table 4 - Summary of the benchmark results (Ducastel, 2020)

At a later stage, this choice proved to be the right one for our problem, since the addition of the external ingress controller required the usage of BGP, which Calico supports by default.

4.2.7. Ingress

Ingress is an API object that is responsible for managing external access to the services in a cluster (Figure 18). Ingress is needed because, by default, pods are not accessible from external networks, but only from other pods within the same cluster. It may provide load balancing, SSL termination and name-based virtual hosting. The traffic routing is controlled using rules defined in the ingress resource, allowing services to have externally reachable URLs, exposing HTTP and HTTPS routes from outside the cluster to services within the cluster. Ingress solves some service limitations, as the limited routing capabilities and the cost, when it is required to implement multiple load balancers in a cloud environment.

[32] https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html

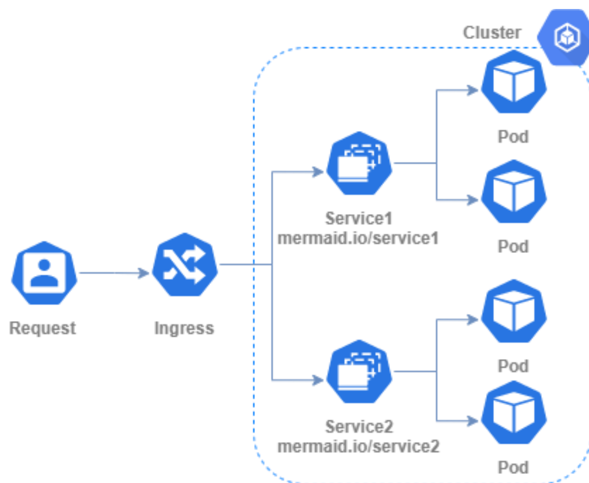

Figure 18 - The ingress resource

4.2.8. Bare Metal Considerations

In traditional cloud environments the network *LoadBalancers* components are available on request, but in our scenario this component does not exists, so when we try to create a *LoadBalancer* service in Kubernetes, it remains in the "Pending" state. This kind of problem is not common in the Kubernetes world, since its development is mostly aimed towards cloud deployment. This became a major obstacle in the course of the project, since practically all ingress controllers rely on *LoadBalancer* objects (from cloud providers, as in Figure 19) to request an external interface and consequently an IP address to receive external traffic.

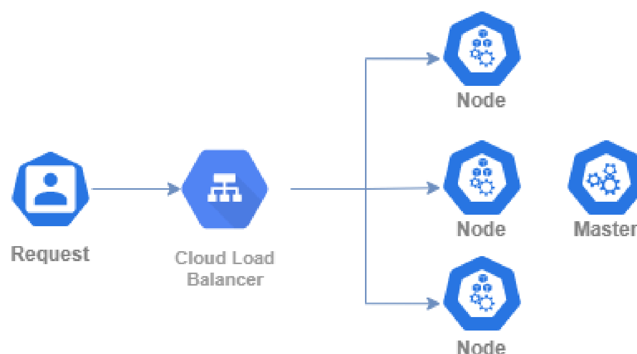

Figure 19 – Usage of a cloud LoadBalancer in a cloud environment

Despite this limitation, there are some methods that allow external traffic to enter the cluster.

4.2.8.1. MetalLB

MetalLB[33] is an open-source load-balancer implementation for bare-metal Kubernetes that uses standard routing protocols, providing support for LoadBalancer services. Basically, when a LoadBalancer is requested MetalLB allocates an IP address for it based on a pool provided by the user. It runs on the cluster and can operate in layer 2 mode or BGP mode.

In layer 2 mode, illustrated in Figure 20, one machine in the cluster is responsible for the service and uses address discovery protocols, like ARP for IPv4 and NDP for IPv6, to make those IPs available, assigning multiple IP addresses to a single machine.

In BGP mode, all machines establish BGP peering sessions with nearby routers, providing routing instructions to the service IPs. BGP allows true load balancing across multiple nodes and fine-grained traffic control.

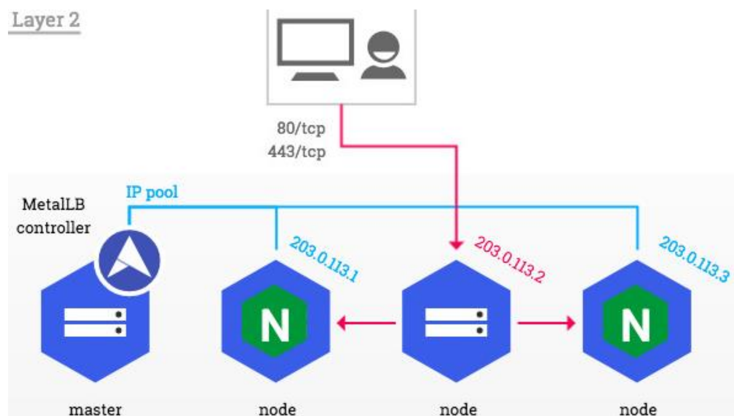

Figure 20 – MetalLB IP assignment using layer 2 (NGINX, 2021a)

33 https://metallb.universe.tf/

This is an excellent solution but unfortunately it was not possible to apply this method on our implementation since we did not have an external elastic IP pool to setup the MetalLB pool.

4.2.8.2. NodePort Service

Using a NodePort service, the user will expose the application using an unprivileged port (range: 30000 - 32767) on every node of the cluster (Figure 21). This means that every time a user wants to access our application, he had to manually define the service port on the request, which would be a major drawback and reason enough to put this solution aside.

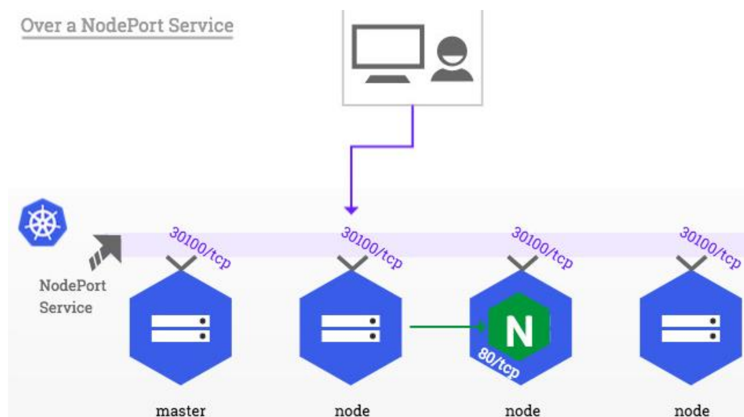

Figure 21 – Using NodePort to access application (NGINX, 2021a)

4.2.8.3. NodePort with external LoadBalancer

Together with the aforesaid NodePort solution, it is possible to implement an LoadBalancer that is external to the cluster, facing the internet and pointing to each Kubernetes node, as shown in Figure 22. This method is similar to the one provided by the cloud providers, requiring an edge network component. This way, external clients do not access the nodes directly, which is also suitable for environments where none of the cluster nodes has public IP addresses. This solution solves the major issue haunting NodePort, that is having to manually define the service port on the request, but it also brings its own challenges, like the need for a manual configuration of the LoadBalancer that is external to the cluster, increasing the project complexity.

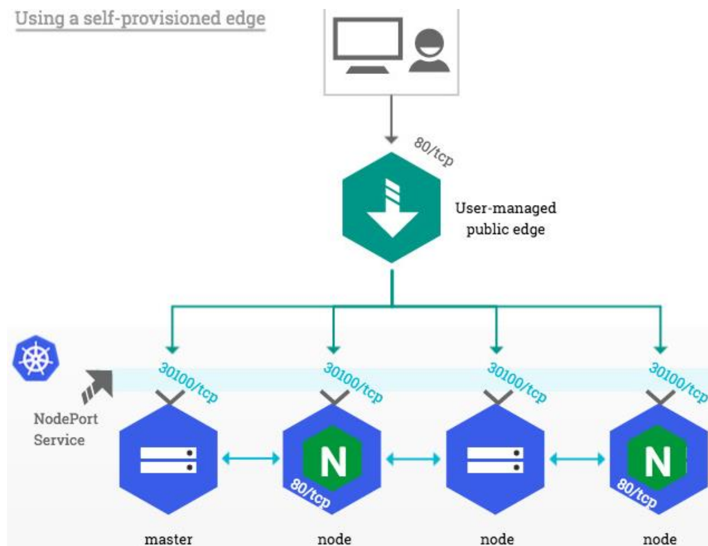

Figure 22 - NodePort with external LoadBalancer (NGINX, 2021a)

4.2.8.4. Host Network

It is possible to configure the ingress controller to use the host network. This means that when the ingress controller uses the ports 80 and 443 (usually only these ports are used), those ports are bind directly to the Kubernetes nodes interfaces, without extra network translation imposed by NodePort (Figure 23). This solution is very limitative since it is only possible to run a single ingress controller pod on each cluster node, because is not possible to bind the same port multiple times. Another limitation is that if pods are not run as DaemonSet[34], they will only be available on the node they are running on.

[34] A DaemonSet schedules one pod for each cluster node.

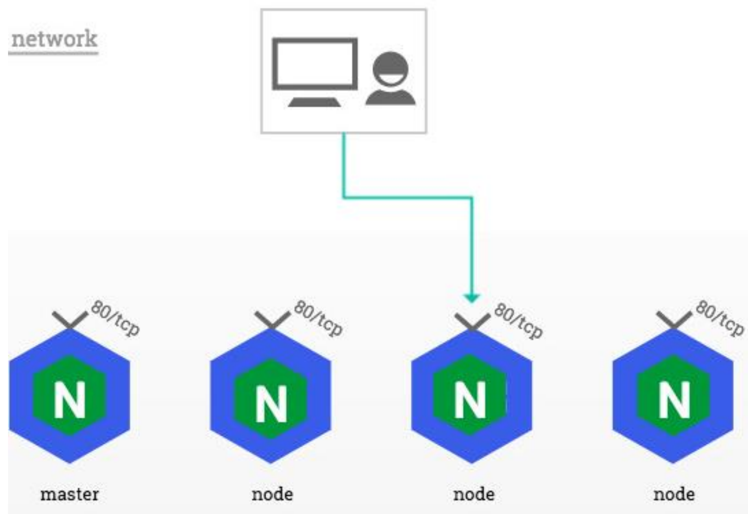

Figure 23 – User access using host network (NGINX, 2021a)

4.2.8.5. Bare Metal Conclusions

Initially we implemented the MetalLB solution but quickly realize that it would not be possible in our scenario because our external IP is not a floating one and it would require all the nodes to be exposed to the outside. It would be possible to use MetalLB to assign local IP addresses and have one node acting as LoadBalancer but this would add complexity to the solution. Our temporary solution was using NodePort with an external load balancer, as described earlier (see section 4.2.8.3). Over time this solution has proven to be difficult to manage because it would be necessary to manually update the load balancer with the published services routes. In the next section we look at two existing ingress controllers and how we solved our problem in a much more elegant and professional way.

4.2.9. Ingress Controller

In order to be able to use ingress, an ingress controller must be deployed on the cluster, and Kubernetes does not come with one by default. The difference between ingress and the

ingress controller is that the ingress is a simple API object that defines the routing rules, and the ingress controller is the component responsible for its implementation.

The ingress controller is a specialized load balancer that runs in the cluster, managing the layers 4 and 7 traffic entering Kubernetes clusters, and potentially the traffic exiting them. Its method of operation is shown in Figure 24, where the Ingress controller is marked as green "NGINX". It connects to the Kubernetes API and watches for different resources like Ingress, Services, Endpoints and others. When these resources change the controller receives a watch notification and configures the data plane to act according. There are more than 20 different ingress controllers[35] to choose from and it is possible to deploy any number of ingress controllers within a cluster.

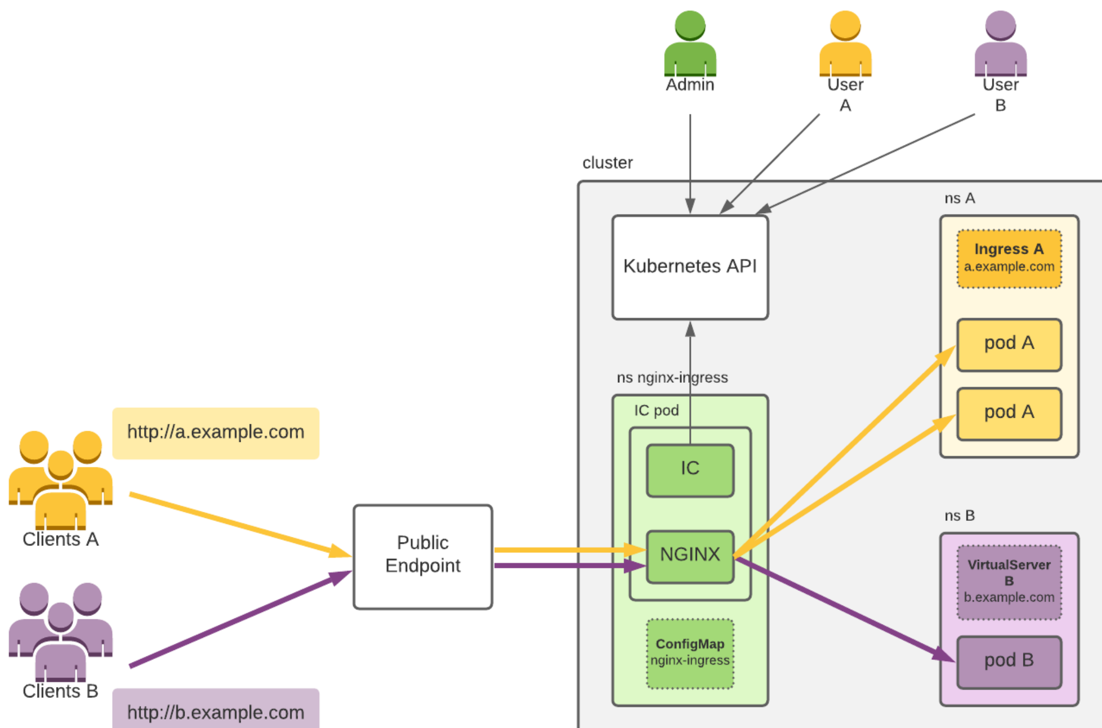

Figure 24 - How the ingress controller exposes applications (NGINX, 2021b)

To select the right ingress controller, we must consider the needs of certain features, like the support for Cross-Origin Resource Sharing (CORS), rate-limit, TCP/UDP endpoints,

[35] https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

and others. Next, we explain the difficulties experienced during our tests, mostly motivated by the bare metal implementation, and how we overcame them by choosing the right ingress controller.

4.2.9.1. NGINX Ingress Controller

The NGINX Ingress Controller[36], known as *ingress-nginx*, is an ingress controller for Kubernetes that uses NGINX as reverse proxy and load balancer, developed by the Kubernetes community. It is important to not confuse this project with the NGINX Ingress Controller[37] provided by the F5 Company, since they are different. This ingress controller is built around the Kubernetes ingress resource and uses a ConfigMap to store its configuration. It assembles a configuration file (nginx.conf) which implies (not always) the reload of the NGINX to apply the changes, that means there is a potential loss of service involved. This ingress controller checks the state of ingresses, services, endpoints, secrets, and ConfigMaps to generate a point in time configuration file that reflects the state of the cluster. It then uses the synchronization loop pattern to check if the desired state in the controller is updated or a change is required[38].

4.2.9.2. HAproxy Ingress Controller

The HAproxy Ingress Controller[39] is an open-source ingress controller implementation launched in 2019, based on HAproxy, with more than 10 million downloads and overseen by engineers at HAProxy Technologies. HAproxy is a well-known and established fast and reliable TCP and HTTP reverse proxy and load balancer. It is important not to confuse this project with the community driven implementation HAproxy Ingress Controller[40] since they are different. It offers richer features[41] like zero downtime reloads (by using hitless reloads,

[36] https://kubernetes.github.io/ingress-nginx/
[37] https://docs.nginx.com/nginx-ingress-controller/
[38] https://kubernetes.github.io/ingress-nginx/how-it-works/
[39] https://github.com/haproxytech/kubernetes-ingress
[40] https://github.com/jcmoraisjr/haproxy-ingress
[41] https://thenewstack.io/how-haproxy-streamlines-kubernetes-ingress-control/

changes that do require a reload cause no downtime), observability, advanced authentication, rate limiting, IP whitelisting, the ability to add request and response headers, connection queuing, and more recently, the ability to run the controller external to a Kubernetes cluster.

This last functionality – run the controller external to the cluster, caught our attention and ended up being used as our ingress solution. Traditionally, and like in every other ingress controller, the HAproxy would run as a pod inside the cluster, having access to its network and consequently allowing the routing and load balancing of traffic. Still, clients outside of the cluster cannot reach it unless it is exposed as a *service*. As already discussed, since our setup is on-premises, NodePort or Host Network are typically used, and then a load balancer is configured in front of the cluster. Both solutions require a load balancer in front of the ingress controller, which means that there are two layers of proxies through which traffic must travel to reach the pods, as shown previously in Figure 22.

Since version 1.5 of HAproxy Kubernetes Ingress Controller it is now possible to run the ingress controller outside of the cluster, which removes the need for an additional load balancer in front of it. HAproxy gets direct access to the physical network and becomes routable to external clients and pods using Calico as the Kubernetes network plugin and BIRD as router to enable BGP Peering (Figure 25).

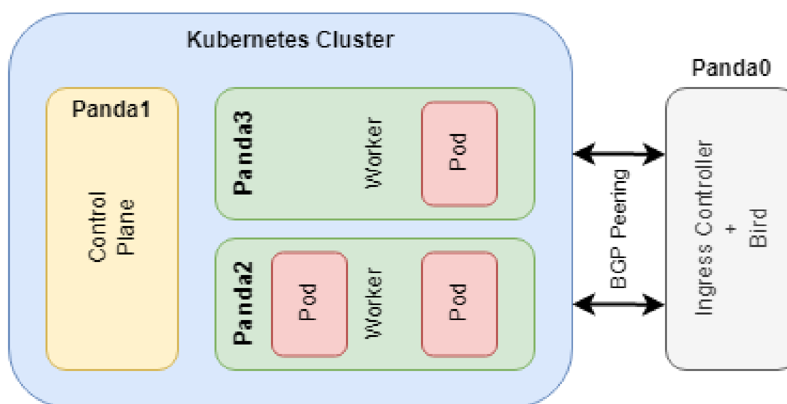

Figure 25 - Interaction between the Kubernetes cluster and the external ingress controller

4.2.9.3. Final Considerations about Ingress Controllers

Initially we plan to implement the MetalLB solution but quickly realize that it would not be possible because the external IP provided was not a floating one, the need to expose all the nodes and the added complexity solution. As an initial setup, we implemented a NodePort with NGINX as an external LoadBalancer, but over time this has proven to be more difficult to manage, because the external edge node would always need a manual configuration. We then proceed to test the HAproxy external ingress controller, which worked very well and brought benefits such as a reduction in the number of hops.

Saying that HAproxy saved our project is not an overstatement. Its special external ingress controller feature allowed us to create an external node that will handle all the external traffic and route it to the required pods and services inside the cluster. But it is not all good, one possible drawback that we may face is the creation and use of service mesh addons like Istio, since they are very dependent on the ingress controller and cluster-wide speaking, we do not have one. Another limitation is that having a single edge node means a single point of failure, something that could be mitigated with two edge nodes.

4.2.10. Domain Name System (DNS)

All our applications share the same Ingress between them and thus share a single-entry point to the cluster. One of the ways to select the correct destination of a request is by the target hostname (the *Host* header in the case of HTTP), making DNS indispensable in the implementation. In our use case, we will be using a wildcard DNS record to assign a domain name to the environment and using subdomains to access different applications, implementing a “subdomain-based routing”. This kind of setup is based on a creation of a wildcard DNS record (e.g., *.example.com) that resolves to the external IP address of our ingress controller. This kind of implementation brings several benefits, like allowing the usage of any path, including the root path (/) that can prevent problems with assets load (we had this problem implementing Portainer) and makes the DNS implementation relatively straightforward since there is no need for interaction between Kubernetes and the DNS provider.

On cloud scenarios it is possible to use tools like external-dns[42] that is a controller to automate the creation of domain names, but our DNS provider was not compatible with it.

4.2.11. SSL Termination

Ingress controllers uses certificates and private keys to serve applications over TLS and one of the biggest problems of these Kubernetes implementations is certificate management. By using cert-manager[43] it was possible to add TLS certificates to the ingress routes, simplifying the process of obtaining, renewing, and using those certificates. Cert-manager is a controller that runs in the cluster and using a set of Custom Resource Definitions (CRDs), automates the certificate management in cloud native environments, providing X.509 certificates from a variety of sources, whereby we will be using Let's Encrypt[44] as our Certificate Authority (CA).

Let's Encrypt is a nonprofit Certificate Authority that provides TLS certificates, fully automatically and for free, to more than 260 million websites, allowing the use of HTTPS on the websites.

One of the best features of cert-monitor is its integration with the ingress API, enabling the automatic generation of certificates for Ingress resources. As described in section 4.2.9.3, we used the Ingress controller HAproxy, which allows SSL termination and since it also works as load balancer, by enabling SSL termination, we are adding secure communication to all services at once. Typically, SSL termination allows the performing of all encryption and decryption of the traffic at the edge of the network, stripping away the encryption and passing the clear messages to the pods (SSL offloading). This is the method that we used, as illustrated in Figure 26, because our nodes communicate securely in an isolated network and therefore do not need to implement a new TLS connection to the backend pod. The main benefits of this implementation are the easier management of certificates, better control over

[42] https://github.com/kubernetes-sigs/external-dns
[43] https://cert-manager.io/
[44] https://letsencrypt.org/

the exposed routes to the internet and reduced overhead from the task of processing encrypted messages, freeing up CPU (Nick Ramirez, 2019).

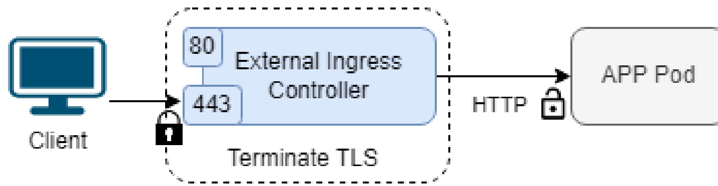

Figure 26 - SSL termination

For environments where it is not possible to communicate securely, it is recommended to implement in-cluster pod traffic encryption using the CNI (Calico in our scenario) or follow the architecture shown in the Figure 27, that will implement a second secure connection to the backend pods. This solution is not recommended as it adds overhead and consequently CPU usage to the pods.

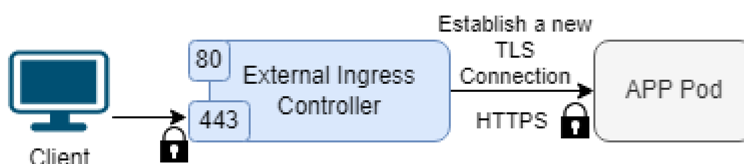

Figure 27 - SSL termination with SSL connection to the backend

It is also possible for the external ingress controller to implement SSL passthrough, passing the data through the external ingress controller without decrypting it as illustrated in Figure 28. This implementation would make sense in scenarios were the deployments already have TLS certificates (e.g., ArgoCD), but we did not use it because it would make the certificate management more complex.

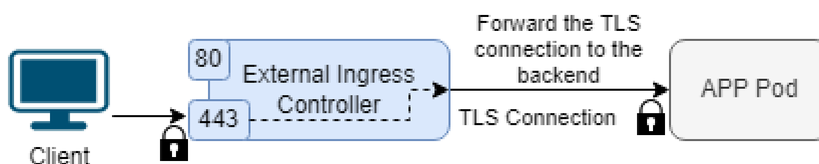

Figure 28 - SSL passthrough

4.3. Container Storage

Kubernetes is a container-based orchestrator and in an ideal scenario all pods are stateless, which means that no data nor state is stored on the pod, but usually these need to be persistent or to save data persistently. This raises a problem when persistent storage is needed between services. To solve this problem, Kubernetes storage is attached to pods via Persistent Volumes (PV) and Persistent Volume Claims (PVCs) to provide increased resilience and availability to face events like application crashing or a workload being rescheduled on a different node. Storage allows the customization of some requirements like access modes, volume expansion, volume provisioning and kind storage. The access modes available are *ReadWriteOnce*, used when a single Pod can read and write to the volume, *ReadOnlyMany,* used when multiple Pods can read the volume and *ReadWriteMany,* used when multiple Pods can read and write to the volume. The volume expansion allows the request of additional storage from the orchestrator when the volume is filling up, also expanding its filesystem. To use these features, it is necessary to use tools that can communicate with storage, providing a bridge to connect containers and physical.

4.3.1. Container Storage Interface (CSI)

The Container Storage Interface (CSI) has been developed as a standard to expose block storage and file storage in Container Orchestration Systems (COS) such as Kubernetes. CSI implements a controller plugin and a node plugin. The controller service is responsible for managing the creation and deletion of volumes, while the node service is responsible for preparing volumes to be consumed by pods on the node. With the adoption of this plugin, the hosting layer becomes fully extensible, allowing the use of storage systems such as Google Persistent Disk[45], Amazon S3[46] and Ceph[47].

[45] https://cloud.google.com/persistent-disk
[46] https://aws.amazon.com/pt/s3/
[47] https://ceph.io/

4.3.2. Longhorn

Longhorn[48] is a free and open source distributed storage system based on block storage that implements distributed storage using containers and microservices. It works by creating a dedicated storage controller for each volume, synchronously replicating them across multiple replicas stored on multiple nodes. Once installed, longhorn allows the use of persistent volumes in the cluster. Initially developed by Rancher Labs, it is now an incubating project of CNCF as of 12/11/2021, which means that it is still in early stage of development with its userbase being made essentially of early adopters, known as "Visionaries". The main features of longhorn are incremental snapshots using block storage, distributed storage with no single point of failure, and S3-compliant backups.

4.3.3. Rook + Ceph

According to the official documentation, Rook[49] is an open source, cloud-native storage orchestrator that provides the platform, framework, and support for broad range of storage solutions for native integration into cloud-native environments. Rook is ready to handle multiple high-end storage providers, including Ceph, Cassandra, and NFS, the latter with Alpha support.

Cassandra is a high performance, fast and scalable NoSQL database. Network File Storage (NFS) allows remote hosts to mount file systems over the network and perform operations as if they were mounted locally. During Rook's development, several databases have been losing support and in the latest versions solutions such as *CockroachDB*, *EdgeFS* and *YugabyteDB* cannot be use (available in version 1.5).

Ceph is a highly scalable distributed storage solution for block storage, object storage, and shared filesystems with several years of production.

[48] https://longhorn.io/
[49] https://rook.io/

4.3.4. Network File System (NFS)

The solution of using off-cluster storage was also considered. In Kubernetes this is one of the preferred solutions when you want to let the nodes just do the processing of the workloads, thus physically splitting the storage (Data Plane) and processing solution. The main advantages of NFS are the simultaneous write support by multiple nodes, its robustness and very good support (Skender, 2020).

4.3.5. Final Considerations About Storage

The lack of features and implementation options put the choice of longhorn aside. Due to hardware availability limitations, our cluster cannot have the 3 nodes needed to create the distributed storage quorum, and longhorn does not allow the implementation with only one node. Furthermore, the use of Rook with Ceph allows the creation of block storage and object storage, if needed, without major modifications to the implementation, which is not possible on longhorn.

The use of NFS or other external storage was considered, but quickly discarded due to the extra resource usage in its implementation, as another VM would be required to implement the software. The use of cloud storage would also be an option, but it requires hiring external services and therefore also discarded.

To conclude this topic, we chose to use Rook + Ceph, currently without the recommended redundancy of the 3 nodes, due to hardware limitations available for the deployment, thus using 1 monitor and 1 manager, as an implementation without high availability. This type of implementations is not recommended, for obvious reasons: if the only node that has storage available fails, all services in the cluster are left without access to it. Since this is an academic work, and the cluster itself does not have high availability, and high availability itself has a cost in terms of resource consumption, it does not make sense to provide high availability only on storage.

4.4. Kubernetes GitOps

The GitOps tools used in Kubernetes are responsible for keeping the cluster configuration up to date with a single source of truth in the Git repository. This tool will be deployed on the cluster itself and has special permissions to be able to perform configuration changes.

4.4.1. ArgoCD

ArgoCD[50] is a continuous delivery tool for Kubernetes used by companies such as edX, IBM, Electronic Arts, PayPal, Volvo, and others. This is a declarative tool and follows the GitOps pattern, using the repositories to define the state of the cluster. This is important for version control of applications, configurations, and development environments, allowing application deployment and lifecycle management to be automated, verifiable, and easy to understand. Its working method is quite simple: ArgoCD is implemented as a controller that is constantly checking the running services and their current states against the states defined in the Git repository. When the states are different it is considered that there is an "OutOfSync", and an update is performed. This makes it possible to automate the deployment of certain applications to the specified environments. ArgoCD can also apply updates to branches, tags, or select a certain version of the manifests in a commit and automatically apply the changes.

4.4.2. Flux v2

Flux[51] is a tool that keeps Kubernetes clusters synchronized with a configuration source (e.g., a Git repository) and automatically performs updates whenever there are new code versions. The version 2 (Flux v2) has been rebuilt from the ground up to use the Kubernetes

---

[50] https://argoproj.github.io/argo-cd/
[51] https://fluxcd.io/

API extensions, allowing an easy integration with Kubernetes core components. Flux is designed to be configured using a CLI client and does not have an administration dashboard.

4.4.3. Werf

Werf[52] is an open-source command line tool, developed in Go, designed to increase the speed of software development and delivery. It is not a complete CI/CD solution but allows the creation of pipelines that can be embedded into already created CI/CD systems. Its use has been disregarded because it is a recent tool without much use in the market, and it does not have a driver for Kubernetes, being geared towards CLI usage.

4.4.4. Final Considerations About Kubernetes GitOps

The area of GitOps in Kubernetes is recent, and there are few tools that can update the cluster state with the git source. We chose to use ArgoCD because it has a dashboard, which makes it easy to integrate and configure, and there is no loss of functionality when compared to Flux.

4.5. GitOps Platforms

GitOps platforms play two fundamental roles in the development of the project. On one hand, they are responsible for keeping a record of changes to the project source code (i.e., the application microservices), on the other, with the use of GitOps, it is possible to describe the entire deployment declaratively, and thus keep a record of the cluster deployment, maintaining a single source of truth. Allied to this, with CI it is possible to test and build software or Docker images automatically when a pull request is approved. By using these practices, it is possible to increase productivity, make life easier for the development and

[52] https://werf.io/

operations teams, increase confidence in the developed software, ensuring greater stability, reliability, and security.

4.5.1. GitHub Actions

GitHub Actions[53] are part of GitHub and, according to GitHub, "allow you to automate, customize and execute workflows directly in the repository, streamlining the software development cycle". The actions are event-driven, which means that commands are executed as certain events occur. One of the simplest examples of why this kind of architecture makes so much sense in this development environment is to trigger software testing every time a developer creates a pull request in each repository.

Users can discover, create, and share GitHub actions to do any job, including CI and CD, combining them to create a complete workflow.

Actions are available at the GitHub marketplace, where you can search by name, category, and cost, and are open source and therefore verifiable.

4.5.2. TravisCI

TravisCI[54] is a CI/CD tool that allows you to test and build software projects hosted on GitHub, Bitbucket, GitLab, and Assembla. As a form of support for open-source projects, this was the first tool to allow free testing and building of software on open source software (OSS) projects, an offering that ended in late 2020 with the increase in abuse of cryptocurrency mining and node creation for the The Onion Router (TOR) network (Travis, 2020).

[53] https://github.com/features/actions
[54] https://www.travis-ci.com/

4.5.3. DevOps as a Service

Paid cloud services such as Azure DevOps[55] have been developed by cloud providers to keep customers within their ecosystems. The main advantage of this type of service is its easy installation, configuration, and integration with version control of software in use. Its use was not considered because of the vendor lock-in in the software development, and the very limited number of free "credits".

4.5.4. Final Considerations About GitOps Platforms

There is a wide range of tools capable of performing CI/CD, and they all have almost the same features, but each implementation is always done in different ways. Therefore, it is important to consider the pros and cons of each tool, and GitHub Actions, due to its marketplace, has a greater versatility, allowing complete customization of the pipelines.

We also considered using Jenkins, but because it does not have a free cloud component it would require installation, configuration, maintenance, and consequent consumption of our limited resources.

[55] https://azure.microsoft.com/en-us/services/devops/

Chapter 5
Implementation

This chapter documents the technical aspects of our proposed solution, deployed into a real-life system. It follows the decisions made over the last two chapters, first with the selection of Kubernetes as the orchestration mechanism, and secondly, with the adoption of several technologies to support our cluster, namely the runtime engine, the networking mechanisms to route traffic into the cluster and the persistent storage mechanisms. Here, we document the actual system, its architecture, how each part was deployed, is configured, and can be managed.

Section 5.1. Deployment Models

The proper implementation of a Kubernetes cluster deployment starts with the configuration of an entire hardware support infrastructure that corresponds to our needs. Such setup can be made available in several ways. In this chapter we explain the existent deployment models in order to take a better architectural decision to our specific use case.

Section 5.2. Architecture

Having defined our cluster deployment model, the next step was to design the cluster architecture considering the available resources, minimum platform requirements and possible limitations at the networking level.

Section 5.3 Rook + Ceph

With the architecture defined and consequently the disk space assigned, we perform the deployment of Rook with Ceph, which is detailed in this section.

Section 5.4. Networking

As explained in section 4.2, our deployment had several constrains at the networking level. In this section we detail our final implementation, which allowed us to overcome such constrains and route external traffic to the cluster.

Section 5.5 Portainer

Portainer was used to solve a range of problems brought by our type of deployment. In this section we explain what Portainer is and what problems it did solve.

Section 5.6. Helm

In order to install some technological stacks, we used Helm. This section explains what Helm is and which are its main benefits, when compared to traditional installation methods.

Section 5.7 ArgoCD

ArgoCD will allow us to manage the continuous delivery within the cluster, enabling the automatic deployment and update of applications. This section covers its installation, operation method and configuration.

Section 5.8. Observability

Observability is the concept of being able to understand what is happening in the cluster, at system and application level, at any given time. In this section we cover the adopted solution, which features Prometheus, Alertmanager and Grafana.

Section 5.9 MER Application Development

Finally, we conclude with the changes made to the MER application development workflow, implementing a DevOps working method, that aims for a faster and more agile software development process.

5.1. Deployment Models

The first step to successfully implement a Kubernetes cluster deployment is the existence of an entire hardware support infrastructure that corresponds to our needs. With the digital revolution, it is possible for this infrastructure to be made available in several ways, shapes, and sizes such as Infrastructure as a Service (IaaS) (e.g., AWS), where the

user must do its own platform management, managed Kubernetes as a Service (KaaS) (e.g., EKS, AKS, etc.), or even as bare metal.

Cloud providers offer managed Kubernetes services, which allow the quick and easy provisioning of the cluster without any technical knowledge, saving on engineering effort. This type of implementation is very reliable, but it is important to remember that there is always an additional monetary cost associated with it. Moreover, only certain versions of the orchestrator can be used. Usually, alongside the application, many other platform services are executed, like metrics collection and CI tools, that can be made available as external services on the same cloud provider, with the disadvantage of vendor lock-in. The use of this type of solutions makes sense when there is not enough technical knowledge, or on small teams.

With a limited budget, it may make more sense to do your own Kubernetes management, with the rolling of your own version. The use of this solution comes with several benefits such as no vendor lock-in, the possibility of moving to any cloud, either public or private, the possibility to fully customize the solution with any version of the platform and plugins, and the price should always be more affordable than paying for a managed solution (excluding human resources). However, this requires qualified personnel who knows how to manage and implement the platform, with technical knowledge to fully customize the implementation.

One of the pertinent questions that can arise is the use of virtual machines versus bare metal to host the cluster. Looking at the problem in a simplistic way, it may seem strange to run containers inside virtual machines, firstly because they are two solutions to the same problem and secondly because of the overhead. The truth is that this is a common practice in the market and Kubernetes can be used on both bare metal and virtual machines. The use of virtual machines in our scenario makes more sense, due to the benefits brought by them, such as the ability to make better use of existing hardware (allowing to “slice” each machine into a fully isolated computer), easily create backups / snapshots and clone machines, change resources without disrupting the server and allowing the configuration of advanced networking setups.

In our solution we rolled our own Kubernetes mainly due to budget constraints and because we were offered a set of resources that could be used for this project. This made it possible to study and understand in a deeper way how the whole orchestration system and its deployment works, as well as customize the whole solution according to our requirements, taking into account our problem.

5.1.1. Clusters Tiers

The organization of clusters into tiers or environments is a good practice used all over the world. Usually, each environment is associated with specific service level objectives (SLOs) and service level agreements (SLAs), as well as a specific purpose for the cluster that in our case is explained ahead (Rosso et al., 2021). The correct usage of this methodology will help to ensure that the deployed applications and all its ecosystems work as expected when applied to critical production clusters.

Testing – Usually these are ephemeral clusters that have a defined time-to-live, such that they are automatically destroyed after some time. Their purpose is to test particular components or platform features under development but can also be used by developers when a local cluster does not have sufficient requirements for testing. There is no SLO or SLA for these clusters.

Development – Development clusters are generally permanent clusters not associated with any time-to-live. These are usually equipped with all the features of a production environment with the aim of running the first round of integration tests, compatibility testing and application development. The availability of these clusters will often be near production-level because outages would impact developer productivity.

Staging – Staging clusters are permanent clusters that share the same versions as production clusters. They are used for final integration testing and approval before production.

Production – Production are the real clusters used to serve the applications to the clients. Only approved, production ready, and stable releases of the software are allowed to run on

these clusters, together with fully tested and approved stable releases of the platform. Normally detailed and well-defined SLOs and SLAs are used and tracked.

Due to lack of resources our environment can only count with one production cluster, while our application development was performed on a local cluster using minikube[56]. Prior to deployment on production, all Kubernetes configurations were tested on a cluster like the production one.

5.2. Architecture

In order to deploy a Kubernetes cluster, there are a number of minimum requirements that must be met. The operating system should be a Linux host, with at least 2 GB of RAM per machine and 2 CPUs or more, with network connectivity between hosts within the same cluster and disabled swap.

For this project the Centre for Informatics and Systems of the University of Coimbra (CISUC) has provided us 18 CPUs, 18GiB of RAM and 300 GiB of storage under a Xen Orchestra managed system. The availability of these resources under a virtualization platform allowed us to manage them in a way that was most convenient to us.

As can be seen in Figure 29, we divided our resources into 4 separate nodes, one of which is not part of the Kubernetes cluster and is used only as ingress.

[56] https://minikube.sigs.k8s.io/docs/start/

Figure 29 - The 4 hosts forming the cluster – 3 Kubernetes nodes and 1 edge node, all running on Xen Orchestra

In terms of cluster capacity, the total computing power can be calculated by adding up the CPU and RAM. In our case 16 GiB RAM, 16 CPU and 250GiB disk, since the ingress node cannot run any workloads, so it does not count to the sum. The architecture of our cluster is composed of 1 master node with 4 CPU and 4 GiB RAM, and 2 worker nodes with 6 CPU and 6 GiB each (Figure 30), which were chosen taking into consideration several factors. With the resources we had, it would also be possible to create a cluster with a larger number of nodes (e.g., 4 worker nodes with 3 CPU and 3 GiB RAM). This decision is entirely dependent on the applications to be deployed on the cluster. In our case, a larger number of nodes would create a higher system overhead, a more complex cluster with consequently difficulted management and more pressure on *etcd*. More importantly, the pods could not consume as many resources, which in our implementation would be a problem, because we have a microservice (Spleeter) that has high RAM requirements.

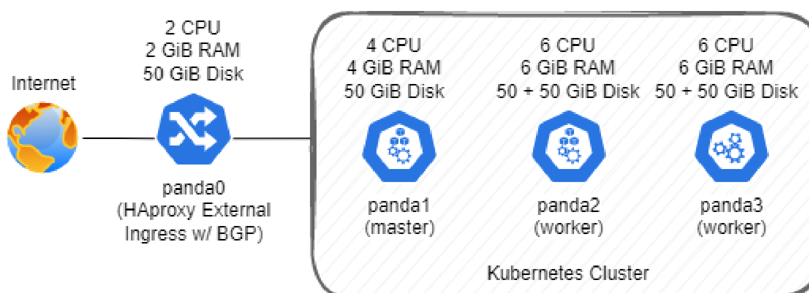

Figure 30 - Cluster architecture

As one can see from the previous picture, our cluster has two single points of failure: 1) if for any reason, the ingress node becomes unavailable, the cluster would be working properly but not available to the external clients; and 2) if the master node becomes unavailable, the cluster would not work properly. This could be avoided if we implemented a high available cluster with at least 3 master nodes, a stacked *etcd* cluster and 2 or more worker nodes, as shown in Figure 31. Since the Ingress does not implement quorum, two ingress nodes would be enough, with the appropriate DNS changes. As already explained, this topology was not applied due to lack of resources.

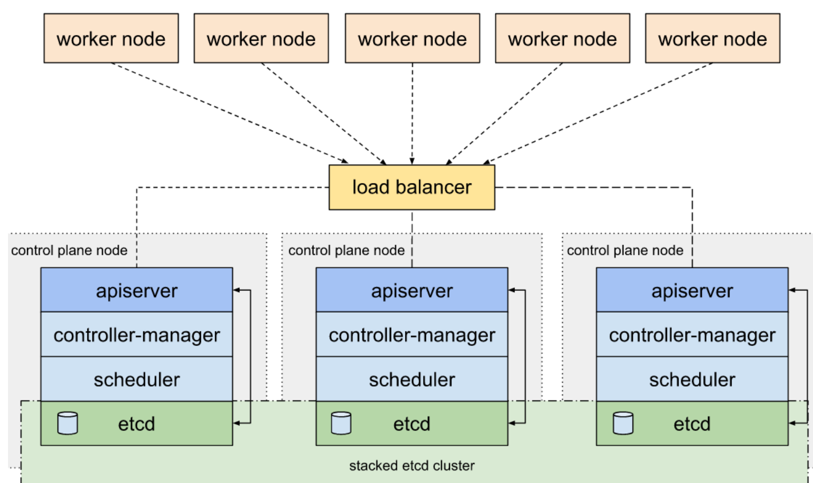

Figure 31 - Kubernetes highly available topology (Kubernetes, 2021)

5.2.1. etcd

Etcd is a consistent distributed database that supports the Kubernetes cluster. It provides highly available key-value store for all cluster data. Keeping etcd clusters stable is critical to the stability of Kubernetes clusters. Therefore, etcd clusters should run on dedicated machines or isolated environments, which is impossible in our scenario. Since we use kubeadm for the cluster provisioning, etcd defaults to run as a single member, in a static pod managed by the kubelet on the control plane node (master). This is not recommended because it is not a high availability (HA) setup, but since our cluster has only one master, it becomes irrelevant having HA on the etcd.

5.3. Rook + Ceph

As described on section 4.3, we use Rook with Ceph to allow the persistent file storage. The setup is straight forward, as we based all our implementation on the official documentation[57]. We started by adding a secondary 50 GiB empty drive to each worker node on the Xen Orchestra that is used to create the filesystem and save the files. We then proceed with the installation, including the *crds.yaml* and *common.yaml* files. These files create the CRDs and common resources that are necessary to start the operator and the Rook cluster. Next, we installed the *operator.yaml* file that deploys the Rook operator, and our modified version of the file *cluster-test.yaml* that defines the settings for the rook-ceph cluster with 1 monitor (MON) and 1 manager (MGR), which are common settings for a small test cluster. Limits and requests were added to the original version of the file, to prevent the overconsume of resources given that there are minimum requirements that must be met: the manager should have 2 GB RAM and 2 CPUs, bearing in mind that memory will grow the more MGR modules are enabled. The monitor requests 4 CPUs and 2.5GB RAM minimum, bearing in mind that these values can be higher for each added disk (Gardner, 2020).

It was possible to restrict Ceph to a portion of the nodes, but because our nodes are similar in specs it was not necessary. After the deployment it was possible to observe that each worker node had an Object Storage Daemon (OSD) pod and the second node (panda2) had the manager and monitor pods allocated, as illustrated in Figure 32.

A properly fault tolerant Ceph cluster should have at least three monitor nodes, preferably spread across fault-tolerant zones so that it is possible to achieve quorum and have HA. However, this is a stack that consumes too many resources, and, besides that, our cluster is not implementing fully HA, so it does not make sense to apply HA on the storage.

[57] https://rook.github.io/docs/rook/v1.7/

```
>kubectl get pods -n rook-ceph -o wide
NAME                                            READY   STATUS      NODE
csi-cephfsplugin-2lhb9                          3/3     Running     panda2
csi-cephfsplugin-provisioner-689686b44-9jgqd    6/6     Running     panda2
csi-cephfsplugin-provisioner-689686b44-kmd95    6/6     Running     panda3
csi-cephfsplugin-tkxjp                          3/3     Running     panda3
csi-rbdplugin-bjs6k                             3/3     Running     panda2
csi-rbdplugin-ftqv2                             3/3     Running     panda3
csi-rbdplugin-provisioner-5775fb866b-cqnz8      6/6     Running     panda2
csi-rbdplugin-provisioner-5775fb866b-vnpvd      6/6     Running     panda3
rook-ceph-mgr-a-595845cc44-5fv4f                1/1     Running     panda2
rook-ceph-mon-a-d5fc7669b-k5w2n                 1/1     Running     panda2
rook-ceph-operator-7bdb744878-wv5zp             1/1     Running     panda2
rook-ceph-osd-0-5f9ff4f74b-2m9mw                1/1     Running     panda3
rook-ceph-osd-1-844c9ff957-pd662                1/1     Running     panda2
rook-ceph-osd-prepare-panda2--1-59sb5           0/1     Completed   panda2
```

Figure 32 - Rook Ceph pods allocation

5.4. Networking

Two separate networks were used to connect the cluster nodes to each other and to the internet, in order to ensure maximum safety. The network 192.168.1.0/24 was used as the private network of the cluster, with the address 192.168.1.30 assigned to the ingress, 192.168.1.31 to the master, and 192.168.1.32 and 192.168.1.33 to the first and second workers respectively. The second network was not under our management and is used for administration. Others VMs (from other projects) have access to it, so it was not considered safe for the cluster traffic. During cluster provisioning, the 192.168.1.0/24 network was defined (on the kubeadm) to be used the node-to-node communication, ensuring that the networks are used for different purposes. As for the ingress node, the CISUC team bind an

external IP address to it, providing access from outside the cluster. The following diagram (Figure 33) represents the network topology with both networks, the 4 nodes and the internet.

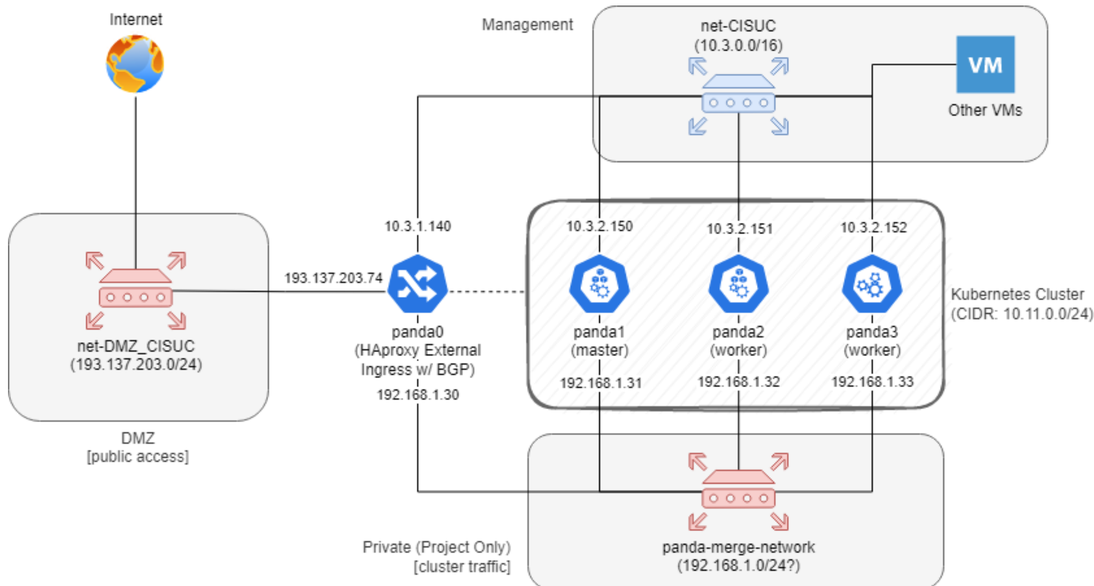

Figure 33 - Network diagram

Calico, the CNI, was the first component to be installed on the cluster. The installation is based on the official docs[58] and the first step was to install the Tigera operator, that then creates the calico pods on each node (Figure 34). The second step was to deploy the calico configuration YAML that enables the BGP and assigns the IP address range for the pod network, which is the same address space specified on the *kubeadm* provisioning. The last step was to deploy a *BGPConfiguration* and a *BGPPeer* objects using the calicoctl CLI client. These objects, shown in Figure 35, enable the full mesh network mode, and assign an Autonomous System (AS) number (65000) to Calico. The second object sets the *peerIp* as the BIRD router (the same as the external ingress controller) and sets its AS number.

[58] https://projectcalico.docs.tigera.io/getting-started/kubernetes/quickstart

```yaml
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    bgp: Enabled
    ipPools:
    - blockSize: 26
      cidr: 10.11.0.0/16
      encapsulation: IPIP
      natOutgoing: Enabled
      nodeSelector: all()
```

Figure 34 - Calico configuration YAML to enable BGP and set pod CIDR

```yaml
apiVersion: projectcalico.org/v3
kind: BGPConfiguration
metadata:
  name: default
spec:
  logSeverityScreen: Info
  nodeToNodeMeshEnabled: true
  asNumber: 65000
---
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: my-global-peer
spec:
  peerIP: 192.168.1.30
  asNumber: 65000
```

Figure 35 - Calico configuration YAML to set the AS and enable BGP peering

This is how Calico shares its pod-level network routes with BIRD so that the ingress controller can use them.

5.4.1. External Ingress

As described earlier, the external ingress uses HAproxy with BGP peering provided by BIRD. This section details how this external node is configured, and how it is able to automatically update the ingress routes without external configuration. This setup started as

a regular HAproxy installation, to which we added the HAproxy external ingress controller binary[59] and a respective service file, in order to enable the automatic start. Then, the BIRD router was installed and configured according to Figure 36. In this file we defined the *router id* as the ingress IP address and created a protocol *bgp section* for each master and worker nodes, with the *filter* being set for the pods CIDR.

```
router id 192.168.1.30;
log syslog all;
protocol bgp { # controlplane
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.31 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}
protocol bgp { # worker1
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.32 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}
protocol bgp { # worker2
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.33 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}
```

Figure 36 - BIRD configuration file

The last step was to copy the *kubeconfig* file from the master to the ingress node, on the same directory. This will allow the external ingress controller binary to access the master Kubernetes API and watch for changes on the ingress objects, updating immediately and automatically the HAproxy routes without loss of service.

[59] https://github.com/haproxytech/kubernetes-ingress/releases

5.4.2. Ingress

For the ingress implementation a set of objects were created, as exemplified in Figure 37. In these files, a number of hosts can be defined that must point to a *service* with a *port number*. These objects are highly customizable, allowing the usage of *annotations* that can enable features such as basic authentication, rate limits, CORS, headers manipulation and others.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: portainer-ingress
  namespace: portainer
spec:
  rules:
  - host: portainer.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: portainer
            port:
              number: 9000
        path: /
        pathType: Prefix
```

Figure 37 - Example of a HTTP Ingress YAML

As previously mentioned, the usage of the cert-manager made the deployment and serving of TLS certificates much easier. To install it we followed the official docs[60] and installed the provided YAML. Next, we added a *ClusterIssuer* object, as shown in Figure 38, that represents a CA that signs the certificates. In this scenario we will be using the Let's Encrypt because it allows us to sign our certificates for free.

[60] https://cert-manager.io/docs/installation/

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-production
spec:
  acme:
    email: myemail@company.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: mermaid-issuer-account-key
    solvers:
    - http01:
        ingress: {}
```

Figure 38 - Lets Encrypt production ClusterIssuer YAML

Due to the integration of the cert-manager with the Ingress API, it is very easy to enable TLS on the hosts. To this end, it is only necessary to add an annotation specifying which cluster-issuer to use to sign the certificate, define the *tls* host where the protected route will be exposed, and define the *secretName* secret that will save the generated certificate, as can be seen on Figure 39.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
  name: portainer-ingress
  namespace: portainer
spec:
  rules:
  - host: portainer.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: portainer
            port:
              number: 9000
        path: /
        pathType: Prefix
  tls:
  - hosts:
    - portainer.mermaid.dei.uc.pt
    secretName: portainer-cert
```

Figure 39 - Example of an HTTPS Ingress YAML

Under the hood, cert-manager generates the private keys, creates the certificates signing request (CSR) and submits the CSR to the CA. When the issuer sends back the certificate, it gets saved in a secret and the ingress controller can expose the route with it, as can be seen in Figure 40.

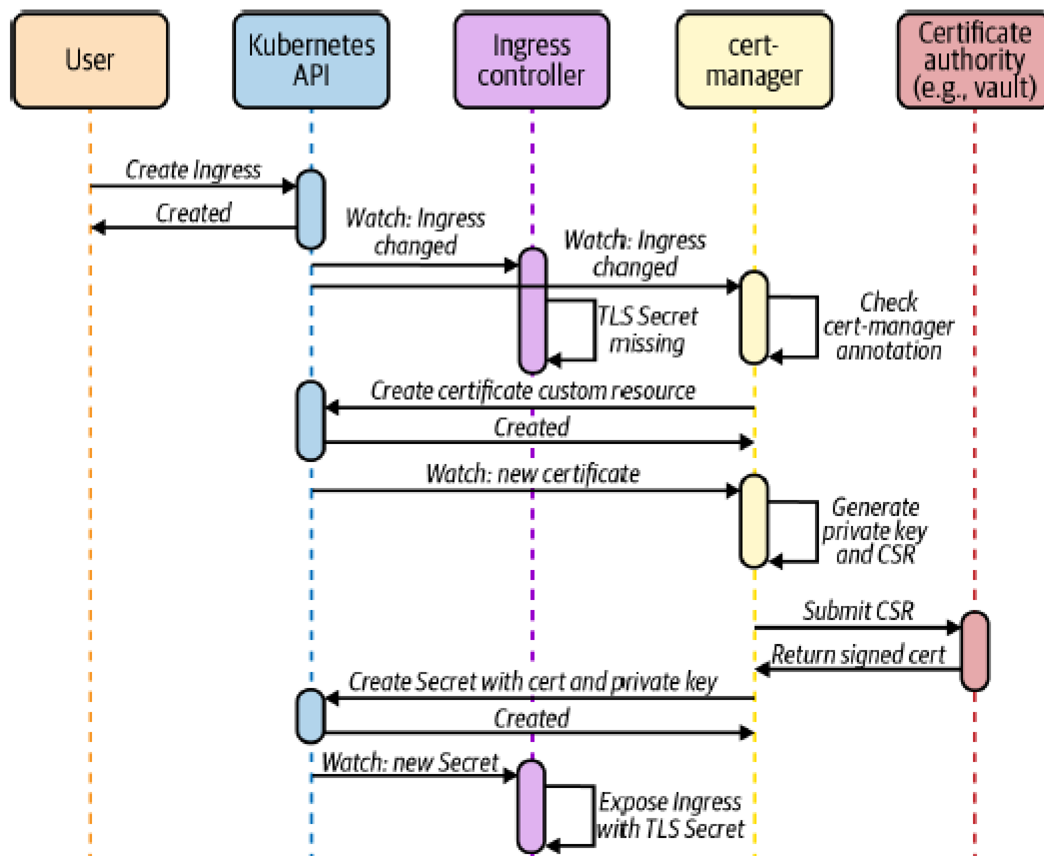

Figure 40 - Sequence diagram of the cert-manager (Rosso et al., 2021)

5.5. Portainer

Portainer[61] is a powerful enterprise grade container service delivery platform. It can be seen as a Kubernetes graphical user interface (GUI), easing the management of the entire

[61] https://www.portainer.io/

cluster. Portainer was used mostly to compensate for the lack of a direct access to the cluster admin functionality using *kubectl*. Our Kubernetes cluster is behind a private network, protecting the direct access to the Kubernetes API, being only accessible with a virtual private network (VPN), that not all the team had access to. Portainer contains a virtual terminal served by a powerful dashboard that allows to manage all the Kubernetes components, like namespaces, deployments, pods, services, installation of Helm charts and others. After it was exposed via HTTPS, it was possible to download a *kubeconfig* file that used Portainer as a proxy to the Kubernetes API, allowing to run kubectl commands without exposing the cluster API directly. Portainer was installed using Helm charts, which are explained in the next subchapter. After the installation it was only necessary to add the ingress so that it was possible to access Portainer from outside of the cluster.

5.6. Helm

Helm[62] is a tool for managing charts that can be seen as a packet manager for Kubernetes. Charts are a set of default values with sensible settings that can be easily modified, containing the description of the package and one or more templates, which contains Kubernetes manifest files. This allows powerful customization via templating and deploying without in-depth knowledge. The main advantages of using Helm are the easiness to find and share software packaged as Helm charts to run on Kubernetes, and the ability to create reproducible builds and easily manage Kubernetes manifests files and its releases, thus allowing easier updates. Helm can be installed as a binary that can run anywhere, connecting to the Kubernetes API to apply the charts that can be stored on disk, or fetched from remote repositories.

5.7. ArgoCD

As explained on section 4.4.1, ArgoCD is responsible for the deployment of solutions on the Kubernetes cluster, following a GitOps paradigm. The installation was carried out

[62] https://helm.sh/

using Helm charts, and because the ArgoCD ingress is exposed with a SSL certificate by default, it was necessary to perform a config on the chart to allow the insecure exposing of the route. The password can be obtained by accessing a secret with the *kubectl* command and the project creation was performed in the web dashboard. In our use case, ArgoCD will continuously monitor a Git repository with the project Kubernetes manifests, listening for commit events. When a commit happens, a "synchronization" process is started, that is responsible for bringing the cluster configuration to the same state as described in Git. When this process ends, the application is updated with the last version, as illustrated in Figure 41.

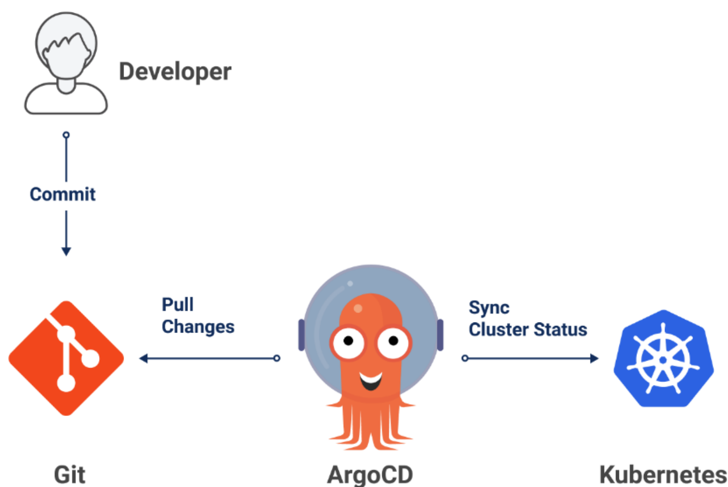

Figure 41 - ArgoCD workflow (Kostis Kapelonis, 2020)

5.8. Observability

Observability allows the teams understand what is happening to the cluster at the system and application level. Typically, this area encompasses logs, metrics, and tracing. Logging is responsible for forwarding log data from workloads to a target backend system, from which is possible to aggregate and analyze them in a consumable and easy way. Metrics are numeric measurements that represents some state at a point in time, allowing its aggregation or scraping for analysis. Tracing allows the understanding of the interactions between the various services.

To collect and aggregate data, generate alerts, and visualize the collected data in the cluster, we use the kube-prometheus-stack[63]. This stack is a collection of Kubernetes manifests that implements Grafana with dashboards, Prometheus with rules and Alertmanager. It is widely used to provide end-to-end Kubernetes cluster monitoring, so it is pre-configured to collect metrics from all Kubernetes components and ships with a default set of dashboards and alerting rules. The stack is composed of the Prometheus operator, Prometheus and Alertmanager with or without high availability, *node-exporter*, *kube-state-metrics* and Grafana. The *node-exporter* is responsible for exporting metrics of the hardware and operating system (OS), and the *kube-state-metrics* is a service that listens to the Kubernetes API and generates metrics about the state of the objects. It was installed using Helm to ease future updates, since it is composed of many components. In the original chart we defined the external ingress with the respective TLS, defined passwords and added some additional scrap configs for Ceph and HAproxy.

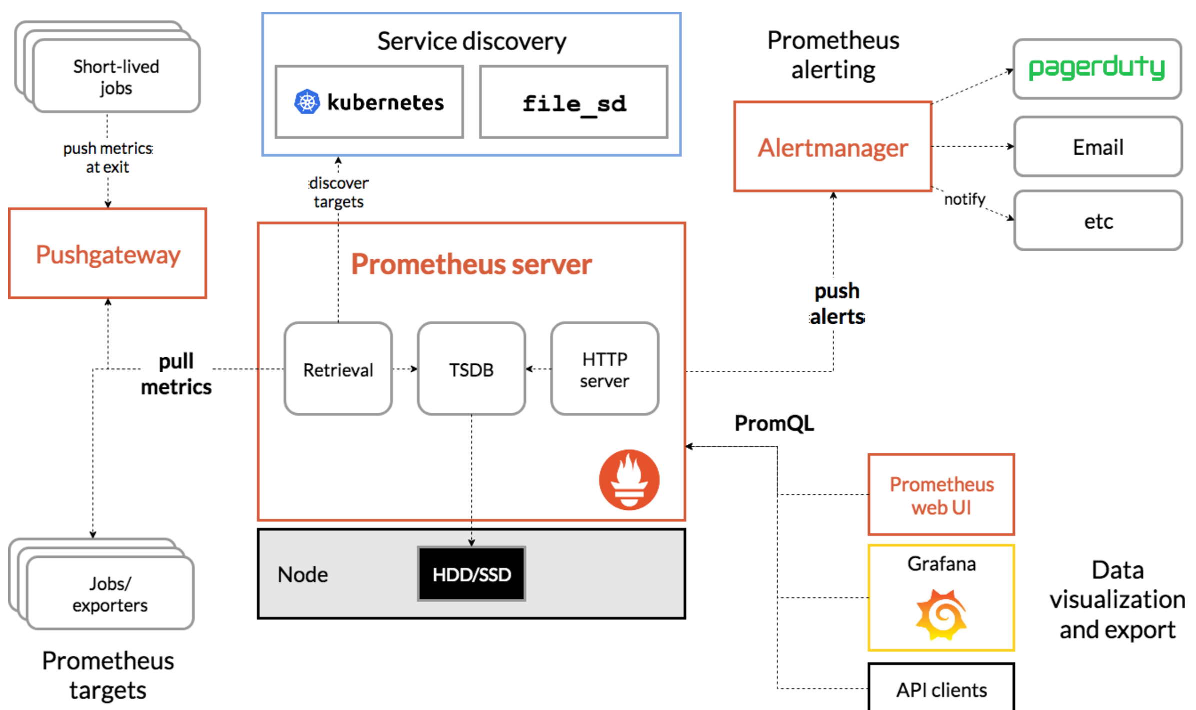

Figure 42 - Architecture of Prometheus and some of its ecosystem components (Prometheus, 2021)

[63] https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack

As can been seen in Figure 42, Prometheus scrapes metrics from various sources, either directly or via an intermediary push gateway. It then stores the scraped data locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts via email, Pagerduty[64] and others. Grafana or other API consumers can be used to visualize the collected data (Prometheus, 2021).

5.8.1. Grafana

In our context, Grafana[65] is an operational dashboard that allows operators to understand what is happening with the Kubernetes cluster and its applications, enabling the creation of dashboards more easily. It builds charts and dashboards from the data stored in Prometheus, being the default solution for viewing such metrics. The marketplace allows the installation of extensions such as custom dashboards developed by the community, the collection and export of data using plugins from many vendors, as Jira or Google Sheets, and their visualization. Our implementation of Grafana has dashboards that show information about the Ceph status, HAproxy, Prometheus Stats and information related to the cluster that can be filtered by node, namespace and much more. After the correct installation of Grafana, we had to manually add community dashboards to allow the Haproxy, K8s cluster, node-exporter and Ceph metrics.

5.8.2. Prometheus

Prometheus[66] is a system monitoring and alerting toolkit that has become the prevalent solution for Kubernetes monitoring, helping manage the various components of its metrics system in our cluster. Prometheus uses several custom resources such as Alertmanager deployments, scrape configurations that inform Prometheus of the targets to scrape metrics from, and rules for recording metrics and alerting on them.

[64] https://www.pagerduty.com/
[65] https://grafana.com/
[66] https://prometheus.io/

5.8.3. Alertmanager

Alertmanager[67] is responsible for handling the alerts sent by the applications (e.g., Prometheus server). It will process the alerts, deduplicating, grouping, and routing them to the correct receiver integration such as email, Slack, and others, generating events in the specified cases. Prometheus has alerting rules that fire off alerts in response to measurable conditions. Those alerts are then sent to Alertmanager, where they are grouped, deduplicated and forwarded to the notification systems, such as email, Slack, or PagerDuty. We did not implement any alert on Alertmanager, but it is something that may be necessary in the future.

5.9. MER Application Development

To be able to understand the changes we have made to the project (i.e., a scalable and distributed music emotion recognition system), it is first necessary to understand what we are trying to achieve. As already explained in the project's objectives, we want to achieve a full cloud native solution, that allow us to develop software in a faster and agile way. This frees the development teams from worrying about services orchestration, abstracting them from the infrastructure, making them more productive and efficient, which triggers an increase in the quality of the solution.

Cloud Native is the concept of building and running applications, taking advantage of the distributed computing offered by the cloud delivery models. Cloud native applications are designed and built to exploit the scale, elasticity, resiliency, and flexibility provided by the cloud.

The first step to achieve this, was to divide what was a mono-repo[68], to a poly-repo environment on Git. This allows for more agility in the future, making it possible for different teams to work simultaneously on different microservices without any conflicts. It also enables the development of different pipelines per repo, to build, test and publish each

[67] https://prometheus.io/docs/alerting/latest/alertmanager/
[68] The initial proof of concept was used a single git repository, containing the entire code for all the services.

microservice, as explained in section 5.9.1. After separating the codebases, during the initial development, we hit a problem caused by the microservices trying to connect to the RabbitMQ server while not yet ready, and thus the connection failed. Docker best practices advocate that container images should only execute a process at a time, so we developed a simple script that checked if a connection to the RabbitMQ server was available, and if the result was positive, would launch the main microservice process.

As noted above, GitHub Actions allow the usage of Actions from the community when they are published in the marketplace. We needed an action to create a RabbitMQ server to run the CI tests, but at the time, none allowed to enable and specify the management port from the RabbitMQ. So, we developed and published an action[69] to create an RabbitMQ server, fully customizable, that allowed setting the username, password, port, management port and image version. This was an added value contribution because it came to be used by several people in the community.

In line with the development of the various microservices, individual software tests that would be integrated into the CI/CD pipelines latter were also created. One of the advantages of using these pipelines is that we are allowed to use different programming languages on the main development of the microservice and on the tests, making the tests more uniform among themselves.

Each repository contains the microservice source code, a Dockerfile that will generate a Docker image and the test suite. Next, we will briefly explain each microservice.

- *vidExtractor*[70] – NodeJS microservice responsible for checking and downloading the YouTube audio file and save it as a file on a shared PV. It is also responsible for creating a NoSQL document with the video metadata like title, artist, and others, sending them in the end to *LyricsExtractor* and *GenreFinder*. The video ID gets sent to *SourceSeparation.*

---

[69] https://github.com/marketplace/actions/rabbitmq-action-with-mng
[70] https://github.com/mer-team/vidExtractor

- *musicClass*[71] – Python microservice responsible for the music emotion classification. It saves the predicted emotion classification in the database and when the processing of all the excerpts ends, sends them to the API, so they can be shown to the users.
- *LyricsProcessor*[72] – Extracts the song lyrics, saving them to the database and sending at the end to the classifier.
- *API*[73] – NodeJS microservice responsible for the backend API. It will receive from the user and send to *vidExtractor* the video URL.
- *featExtractor*[74] – Python microservice responsible for extracting features from the provided music files. At the end sends the features to the microservice *musicClass* so they can be processed.
- *wave2image*[75] – NodeJS microservice responsible for generating and saving waveform images based on the downloaded audio files on a PV.
- *SourceSeparation*[76] - Takes the original audio and separates its sources (e.g., vocals) depending on the model in use. Then sends the video ID to *Segmentation*.
- *frontend*[77] - Frontend microservice based on React. This repo has two Dockerfiles, one for development and another for production that has nginx to serve the generated assets. The DockerHub is updated with booth images.
- *Segmentation*[78] - Segments an audio file into smaller files with 30 seconds length and 15 seconds of overlapping. These files are saved with one audio channel only and 22500 Hz of frequency. At the end it updates the database with the number of excerpts and send the ID to *featExtractor*.
- *LyricsExtractor*[79] - Tries to get the lyric of a song and save it in a text file. At the end, the file name is sent to *LyricsProcessor*, to extract its features.

---

[71] https://github.com/mer-team/musicClass
[72] https://github.com/mer-team/LyricsProcessor
[73] https://github.com/mer-team/API
[74] https://github.com/mer-team/featExtractor
[75] https://github.com/mer-team/wave2image
[76] https://github.com/mer-team/SourceSeparation
[77] https://github.com/mer-team/frontend
[78] https://github.com/mer-team/Segmentation
[79] https://github.com/mer-team/LyricsExtractor

- *GenreFinder*[80] – Get the five most suitable music genres for a song and updates them in the database.
- *Manager*[81] – Microservice to manage the flow of MER execution, managing all the others microservices by determining the sequence of instructions in the system. Created because, previously, microservices called each other, generating an unnecessary degree of dependency between them.

5.9.1. GitHub Actions

GitHub actions gives developers the ability to automate their workflows across issues, pull requests, and more with native CI/CD functionalities. They bring automation directly into the software development lifecycle via event-driven triggers that can be something like a pull request or commit. These automations are handled via workflows which are *yaml* files placed on the *.github/workflows* directory. In that folder we created 2 different actions:

- *build.yaml* - runs on every code push or pull request and builds the microservice docker image, makes a container security scan, installs dependencies, and run the software tests, as illustrated in Figure 43.
- *publish.yml* - runs on every tag creation, builds the docker image and pushes the updated docker image to the corresponding DockerHub repository, as illustrated in Figure 44.

[80] https://github.com/mer-team/GenreFinder
[81] https://github.com/mer-team/Manager

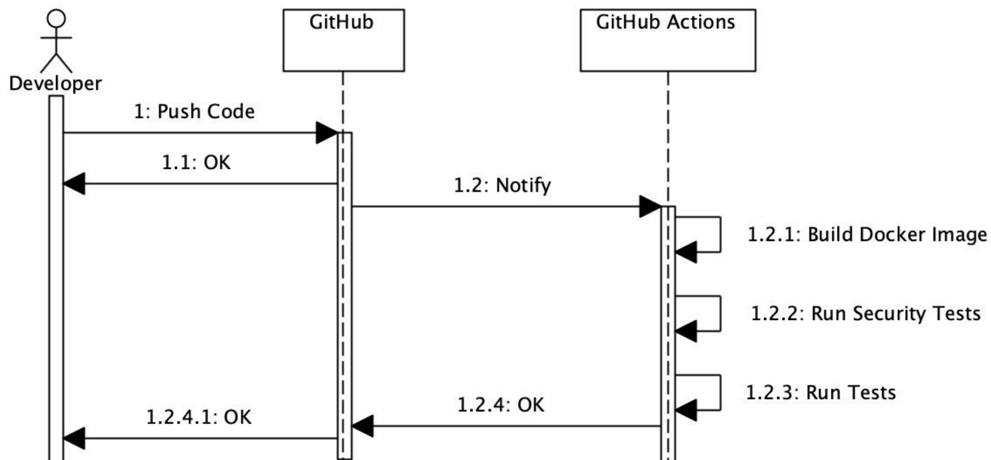

Figure 43 - Sequence diagram of the build action

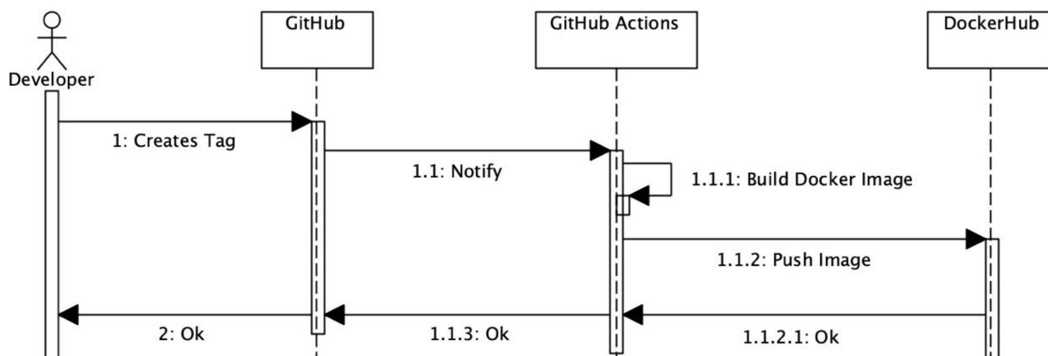

Figure 44 - Sequence diagram of the publish action

At the end, the software development lifecycle should look like the Figure 45. The developer makes changes on the microservices source code through commits and the CI tool, in our scenario, the GitHub Actions, builds and updates the new images when the tests pass. The operations team updates the Kubernetes manifests with the new image's version and ArgoCD will detect such changes and sync with the repository, applying the changes on the cluster.

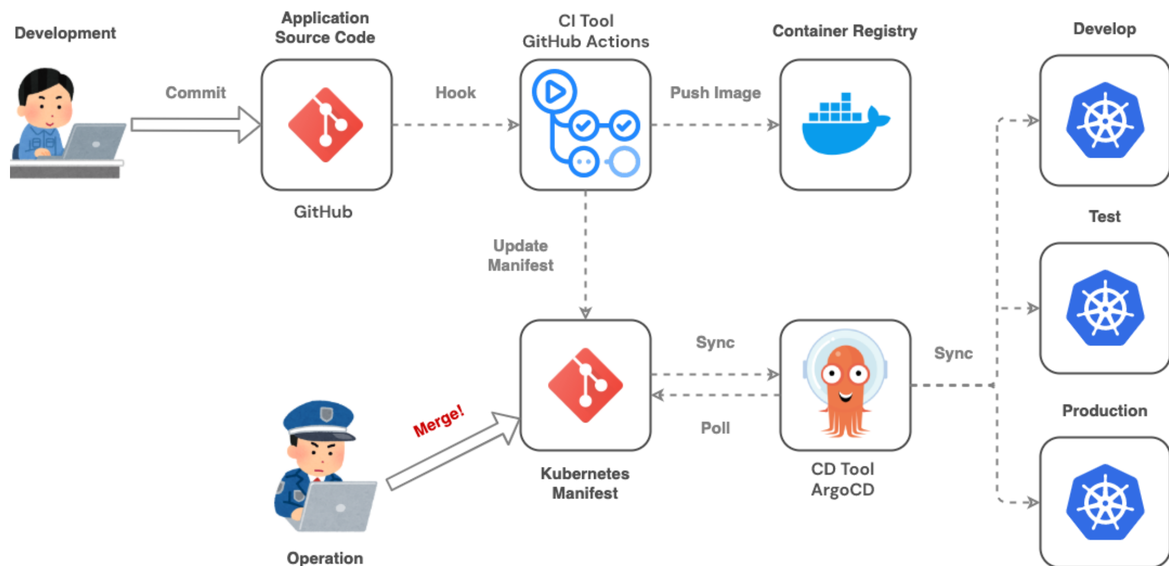

Figure 45 - Software development lifecycle, adapted from (Ando, 2020)

5.9.2. Container Image Registry

Container image registry is a stateless, highly scalable application that stores and distributes container images. It is responsible for storing and allowing the download of container images, having some sort of authentication system implemented that controls the image repository that each user is allowed to push, edit, or delete. During the project development was used DockerHub as a container registry[82] because it has an unlimited storage space, unlike the GitHub Packages[83] that has a limit of 2GB on the Pro plan. There are also solutions that allow the implementation of container registers inside the Kubernetes cluster like Harbor[84] or Trow[85]. These self-hosted solutions were set aside because they would consume resources in the cluster, whereas we could get the same advantages totally for free by using DockerHub.

[82] https://hub.docker.com/
[83] https://github.com/features/packages
[84] https://goharbor.io/
[85] https://trow.io/

Chapter 6
Conclusion and
Future Work

In this final chapter we present a general overview of the work developed, highlighting the main achievements, but aware that our contributions are a step forward in a project that still has many more to go. Thus, we conclude by addressing some points left open, leaving tips on how to develop them.

6.1. Conclusion

The area of MER has been a topic of constant development, given its pertinence and financial interest. This work adds value in this area, by focusing more on the applied research part, laying the foundations which we hope can serve as a basis for other teams and works in the future.

The main objective of this MSc work was to orchestrate a distributed, resilient, and scalable infrastructure to host the MERmaid application. We started from a previous proof of concept, which was almost exclusively the MER web application logic using three microservices, in which we fixed the identified problems and split it in multiple repositories. Accompanying the development of the new version of the application, pipelines were created to integrate the software tests and apply a whole CI/CD logic. This enables developers to work on microservices creating new features, and when the code base gets stable, a release is created that automatically generates updated Docker images. The continuous implementation platform will then take care of update the Kubernetes manifest on the cluster. To support all this, a three-node Kubernetes cluster was planned and provisioned, exposed by a fourth edge node running HAProxy + BIRD (BGP peering). The cluster contains the requirements to host the current or future MER applications, supporting object and file storage using Ceph with Rook, monitoring and alerting capabilities with Prometheus, Grafana and Alertmanager, Portainer for managing the cluster and teams, and ArgoCD to achieve continuous deployment.

With the thesis completion, we can say with certainty that it is possible to implement an on-premises orchestration platform based on Kubernetes, that is distributed, resilient and fully scalable, but more importantly, future-proof. Current, but tested, technologies were used, that have proven themselves for use in both development and production environments. As an additional indicator of the success of this project, we aim to publish a scientific article with the results achieved as soon as the application itself is fully demonstrable.

6.2. Future Work

The developed work prepared a whole base structure for the continuity of the project. The essential aspects for its operation have been addressed, however, some features have not been tackled nor implemented and may be developed in the future. One of these features is the backup system. We chose Ceph with the future in mind, and because of its s3 support, it should be easy to create an automated backup system with Velero[86] to a s3 bucket. Another feature that would be interesting to use is Kubeflow[87], that is a ML toolkit for Kubernetes. Kubeflow would allow us to deploy simple, portable, and scalable ML workflows, but due to infrastructure limitations (i.e., lack of GPUs), its usage was out of the scope of this work. It would be interesting to have several clusters in order to have multiple development environments (e.g., a production cluster and a development cluster). At the observability level we did not implement collection and analysis of the tracing metrics. This could be achieved using Istio[88] with Jaeger[89], Zipkin[90] or Kiali[91], giving us the information needed to keep the microservices healthy and predictable.

In terms of supporting new features of the MERmaid application, and forgetting the orchestration part, it would be interesting to improve the Frontend and API. It would also be

[86] https://velero.io/
[87] https://www.kubeflow.org/
[88] https://istio.io/
[89] https://jaegertracing.io/
[90] https://zipkin.io/
[91] https://kiali.io/

interesting to support various classification models (for example dimensional models with arousal and valence using regression) and even allow users to provide extra models. This would increase collaboration and considering that running already trained models is lightweight, it could motivate the use and dissemination of the application.

References

Ando. (2020, December 24). *Introduction to GitOps*. https://tech.drecom.co.jp/ac2020-beginning-gitops/

António, R. M. (2019). *Microsserviços para Reconhecimento de Emoção em Música*. http://hdl.handle.net/10400.26/31444

António, T. M. (2021). *MER: Estudo e restruturação de um sistema de reconhecimento emocional em música áudio usando o YouTube*.

Böhm, S., & Wirtz, G. (2021). Profiling lightweight container platforms: MicroK8s and K3s in comparison to kubernetes. *CEUR Workshop Proceedings*, *2839*(February), 65–73.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Lessons learned from three container-management systems over a decade*. https://queue.acm.org/detail.cfm?id=2898444

Cooke, D. (1959). *The Language of Music*. Oxford University Press. https://philpapers.org/rec/COOTLO-5

Davis, J., & Daniels, K. (2016). Effective DevOps. In *O'Reilly Media, Inc.* https://www.oreilly.com/library/view/effective-devops/9781491926291/

Delbouys, R., Hennequin, R., Piccoli, F., Royo-Letelier, J., & Moussallam, M. (2018). *Music Mood Detection Based On Audio And Lyrics With Deep Neural Net*. https://arxiv.org/pdf/1809.07276.pdf

Denis Germain. (2021, June 15). *Back from KubeCon & CloudNativeCon*. Deezer I/O. https://deezer.io/back-from-kubecon-cloudnativecon-europe-2021-key-learnings-and-takeaways-9553c1d7a0d0

Densify. (2021). *Comparing deployment tools | Densify*. https://www.densify.com/kubernetes-tools/kubeadm

Docker Swarm. (2021). *How nodes work | Docker Documentation*. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

Ducastel, A. (2020). *CNI benchmark*. https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-august-2020-6e1b757b9e49

Flant Blog / Sudo Null IT News. (2017, October 11). *Kubernetes success stories - SoundCloud*. https://sudonull.com/post/66192-Kubernetes-success-stories-in-production-Part-4-SoundCloud-authors-Prometheus-Flant-Blog

Gardner, B. (2020). *SUSE Best Practices Rook Best Practices for Running Ceph on Kubernetes SUSE Enterprise Storage, Ceph, Rook, Kubernetes, Container-as-a-Service Platform*. https://rook.io/docs/

GitHub. (2021a). *Continuous Integration and Continuous Delivery (CI/CD)*. https://resources.github.com/ci-cd/

GitHub. (2021b). *What is DevOps*. https://resources.github.com/devops/

Hussein Galal. (2021). *Introduction to K3s*. SUSE & Rancher Community. https://community.suse.com/posts/introduction-to-k3s

Kostis Kapelonis. (2020, December 17). *Solving configuration drift using GitOps with Argo CD*. https://www.cncf.io/blog/2020/12/17/solving-configuration-drift-using-gitops-with-argo-cd/

Kubernetes. (2021). *Highly Available topology*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/

Kubernetes & Spotify. (2021). *Spotify Case Study | Kubernetes*. https://kubernetes.io/case-studies/spotify/

Lamia Caliati. (2018, April 5). *FOSDEM 2018*. Deezer I/O. https://deezer.io/fosdem-2018-d2edf86cf06a

Lusa. (2021). *Receitas Musica*. https://www.lusa.pt/article/cOYAr1BlgpbevJ_4pPI1RzMSZM5iuSI1/receitas-de-venda-de-música-tiveram-aumento-global-de-7-4-em-ano-de-pandemia

Malheiro, R., Panda, R., Gomes, P., & Paiva, R. P. (2018). Emotionally-relevant features for classification and regression of music lyrics. *IEEE Transactions on Affective Computing*, *9*(2), 240–254. https://doi.org/10.1109/TAFFC.2016.2598569

Newman, S. (2021). *Building Microservices, 2nd Edition*. O'Reilly Media, Inc. https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018/

NGINX. (2021a). *Bare-metal considerations*. https://kubernetes.github.io/ingress-nginx/deploy/baremetal/

NGINX. (2021b). *How Ingress Controller Works*. https://docs.nginx.com/nginx-ingress-controller/intro/how-nginx-ingress-controller-works/

Nick Ramirez. (2019, June 15). *HAProxy SSL Termination*. https://www.haproxy.com/blog/haproxy-ssl-termination/

OKD. (2021). *OKD*. https://www.okd.io/

Panda, R. E. S. (2019). *Emotion-based Analysis and Classification of Audio Music* (Issue January). https://estudogeral.sib.uc.pt/handle/10316/87618

Panda, R., Malheiro, R. M., & Paiva, R. P. (2020a). Audio Features for Music Emotion Recognition: a Survey. *IEEE Transactions on Affective Computing*, *3045*(c), 1–1. https://doi.org/10.1109/taffc.2020.3032373

Panda, R., Malheiro, R., & Paiva, R. P. (2020b). Novel Audio Features for Music Emotion Recognition. *IEEE Transactions on Affective Computing*, *11*(4), 614–626. https://doi.org/10.1109/TAFFC.2018.2820691

Panda, R., Redinho, H., Gonçalves, C., Malheiro, R., & Paiva, R. P. (2021). *How Does the Spotify API Compare to the Music Emotion Recognition State-of-the-Art?*

https://doi.org/10.5281/zenodo.5045099

Pannese, A., Rappaz, M. A., & Grandjean, D. (2016). Metaphor and music emotion: Ancient views and future directions. *Consciousness and Cognition*, *44*, 61–71. https://doi.org/10.1016/J.CONCOG.2016.06.015

Piotr Karwatka. (2020, January 14). *Monolithic architecture vs microservices*. https://www.divante.com/blog/monolithic-architecture-vs-microservices

Platform9. (2017, July 11). *Kubernetes vs Mesos + Marathon | Platform9*. https://platform9.com/blog/kubernetes-vs-mesos-marathon/

Platform9. (2021). *Using Calico*. https://platform9.com/blog/the-ultimate-guide-to-using-calico-flannel-weave-and-cilium/

Prometheus. (2021). *Prometheus Overview* . https://prometheus.io/docs/introduction/overview/

Reddit. (2021). *Rethinking Kubernetes*. https://www.reddit.com/r/kubernetes/comments/lwb31v/were_the_engineers_rethinking_kubernetes_at/

RedHat. (2018a). *Understanding cloud computing*. https://www.redhat.com/en/topics/cloud

RedHat. (2018b, January 31). *What's a Linux container?* https://www.redhat.com/en/topics/containers/whats-a-linux-container

RedHat. (2018c, March 9). *Understanding microservices*. https://www.redhat.com/en/topics/microservices

Rodolfo Gobbi. (2019, June 27). *Qual o nível de maturidade DevOps da sua empresa*. https://blog.4linux.com.br/qual-o-nivel-de-maturidade-devops-da-sua-empresa/

Rosso, J., Lander, R., Brand, A., & Harris, J. (2021). *Production Kubernetes*. https://tanzu.vmware.com/content/ebooks/production-kubernetes

Skender, M. (2020). *Kubernetes on bare-metal*. https://faun.pub/kubernetes-on-bare-metal-roll-your-own-9ef99312df09

Spotify. (2016). *Spotify patent recognising and indexing context signals in order to generate contextual playlists and control playback*. https://patents.justia.com/patent/11003710

Spotify. (2018). *Spotify patent - Identification of taste attributes from an audio signal*. https://patents.justia.com/patent/10891948

Statista Research Department. (2021). *Music streaming market share* . Statista. https://www.statista.com/statistics/653926/music-streaming-service-subscriber-share/

Travis. (2020). *Travis new princing model*. https://blog.travis-ci.com/2020-11-02-travis-ci-new-billing

Vamsi Chemitiganti. (2019, May 28). *Kubernetes Concepts and Architecture*. Platform 9. https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/

Vexxhost. (2017). *kubernetes mesos comparasion*. https://vexxhost.com/blog/kubernetes-mesos-comparison-containerization

VMware Inc. (2021). *The State of Kubernetes*. https://tanzu.s3.us-east-2.amazonaws.com/campaigns/pdfs/VMware_StateOfK8s_2018.pdf

Weaveworks. (2021a). *Fast datapath*. https://www.weave.works/docs/net/latest/concepts/fastdp-how-it-works/

Weaveworks. (2021b, July 13). *The History of GitOps*. https://www.weave.works/blog/the-history-of-gitops

Wikipedia. (2021). *Comparison of music streaming services*. https://en.wikipedia.org/wiki/Comparison_of_music_streaming_services

Zakhar Snezhkin. (2021). *Small local Kubernetes Comparison*. https://blog.flant.com/small-

local-kubernetes-comparison/

Appendix 1

Kubernetes Setup

Kubernetes Setup

EXECUTAR EM TODOS:

```
# Update the system
sudo apt update && sudo apt dist-upgrade -y
```

EXECUTAR NO INGRESS:

```
# Instalar HAProxy
sudo add-apt-repository -y ppa:vbernat/haproxy-2.4
#sudo apt update
sudo apt install -y haproxy
sudo systemctl stop haproxy
sudo systemctl disable haproxy

# permitir bind das portas 80 e 443
sudo setcap cap_net_bind_service=+ep /usr/sbin/haproxy

# Instalar o ingress controller
wget https://github.com/haproxytech/kubernetes-ingress/releases/download/v1.7.0/haproxy-ingress-controller_1.7.0_Linux_x86_64.tar.gz 1> /de
mkdir ingress-controller
tar -xzvf haproxy-ingress-controller_1.7.0_Linux_x86_64.tar.gz -C ./ingress-controller
sudo cp ./ingress-controller/haproxy-ingress-controller /usr/local/bin/

# criar ficheiro service
cat <<EOF | sudo tee /lib/systemd/system/haproxy-ingress.service
[Unit]
Description="HAProxy Kubernetes Ingress Controller"
Documentation=https://www.haproxy.com/
Requires=network-online.target
After=network-online.target

[Service]
Type=simple
User=root
Group=root
ExecStartPre=/bin/mkdir -p /tmp/haproxy-ingress/etc/
ExecStartPre=/usr/bin/wget https://raw.githubusercontent.com/haproxytech/kubernetes-ingress/master/fs/usr/local/etc/haproxy/haproxy.cfg -P 
ExecStart=/usr/local/bin/haproxy-ingress-controller --external --configmap=default/haproxy-kubernetes-ingress --program=/usr/sbin/haproxy -
ExecReload=/bin/kill --signal HUP $MAINPID
KillMode=process
KillSignal=SIGTERM
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl enable haproxy-ingress
sudo systemctl start haproxy-ingress
```

EXECUTAR NAS MAQUINAS K (kmaster, kworker01 e kworker02):

```
sudo swapoff -a
sudo nano /etc/fstab (comentar linha que tem o /swap.img)

######################## Networking requesites
# https://kubernetes.io/docs/setup/production-environment/container-runtimes/
# Create the .conf file to load the modules at bootup
cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
```

```
EOF

sudo modprobe overlay
sudo modprobe br_netfilter
sudo modprobe rbd

# Set up required sysctl params, these persist across reboots.
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

sudo sysctl --system

######################### Setup CRIO-O
OS=xUbuntu_20.04
VERSION=1.21
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
deb https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/ /
EOF
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.list
deb http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:/cri-o:/$VERSION/$OS/ /
EOF

curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/Release.key | sudo apt-key --keyring /etc/apt/tr
curl -L https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-o:$VERSION/$OS/Release.key | sudo apt-key --keyring

# Install kubernetes certs and depedencies
sudo apt-get install -y apt-transport-https ca-certificates curl -y
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc
sudo apt-get update

# Install CRIO-O
sudo apt-get install cri-o cri-o-runc -y

sudo systemctl daemon-reload
sudo systemctl enable crio --now

# Enable and start iscsid
sudo systemctl enable iscsid --now
sudo systemctl start iscsid

# Install and hold kubernetes
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

EXECUTAR NO MASTER:

```
# Cluster setup (pod-network-cidr é um bloco que não esteja em utilização)
# Vai ser gerado um comando do género 'kubeadm join (...)' que é preciso guardar
# para aplicar no worker
sudo kubeadm init --pod-network-cidr=10.11.0.0/16

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

# Instalar operador do calico
kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml

# Substituir o cidr utilizado aqui (não modifiquei o blockSize)
cat <<EOF | sudo tee ./calico-installation.yaml
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  # Configures Calico networking.
  calicoNetwork:
    bgp: Enabled

    # Note: The ipPools section cannot be modified post-install.
    ipPools:
```

```
    - blockSize: 26
      cidr: 10.11.0.0/16
      encapsulation: IPIP
      natOutgoing: Enabled
      nodeSelector: all()
EOF

# Aplicar o ficheiro criado no ultimo passo
kubectl apply -f calico-installation.yaml

# Instalar o calicoctl
wget https://github.com/projectcalico/calicoctl/releases/download/v3.20.2/calicoctl 1> /dev/null 2> /dev/null
sudo chmod +x calicoctl
sudo mv calicoctl /usr/local/bin/
sudo mkdir /etc/calico

# Configuração do calicoctl
cat <<EOF | sudo tee /etc/calico/calicoctl.cfg
apiVersion: projectcalico.org/v3
kind: CalicoAPIConfig
metadata:
spec:
  datastoreType: "kubernetes"
  kubeconfig: "$HOME/.kube/config"
EOF

# Configurar o calico BGP
# MUDAR O peerIP PARA O IP DO LoadBalancer
cat <<EOF | tee ./calico-bgp.yaml
apiVersion: projectcalico.org/v3
kind: BGPConfiguration
metadata:
  name: default
spec:
  logSeverityScreen: Info
  nodeToNodeMeshEnabled: true
  asNumber: 65000

---
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: my-global-peer
spec:
  peerIP: 192.168.1.30
  asNumber: 65000
EOF

# Configurar o BGP peering
calicoctl apply -f calico-bgp.yaml

# Criar ConfigMap para o ingress controller
kubectl create configmap haproxy-kubernetes-ingress
```

EXECUTAR NOS WORKERS:

```
# Join the cluster using the command provided by the kubeadm
sudo kubeadm join <ip>:6443 --token <token> --discovery-token-ca-cert-hash <hash>
```

EXECUTAR NO INGRESS:

```
sudo mkdir -p /root/.kube
# FAZER NESTE PASSO:
# Copiar ficheiro /etc/kubernetes/admin.conf do MASTER para /root/.kube/config do LoadBalancer
sudo chown -R root:root /root/.kube

# Instalar o Bird
sudo add-apt-repository -y ppa:cz.nic-labs/bird
sudo apt update
```

```
sudo apt install bird

# Ficheiro de config do bird - MUDAR ROUTER ID, IP's, CIDR
# Deixei o gist aqui apenas por descargo de conscicencia, o comando deve criar o ficheiro
# https://gist.github.com/jpcanoso/ad7948256f6fc026cf61b33e55e427e4

cat <<EOF | sudo tee /etc/bird/bird.conf
router id 192.168.1.30;

log syslog all;

# controlplane
protocol bgp {
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.31 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}

# worker1
protocol bgp {
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.32 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}

# worker2
protocol bgp {
    local 192.168.1.30 as 65000;
    neighbor 192.168.1.33 as 65000;
    direct;
  import filter {
    if ( net ~ [ 10.11.0.0/16{26,26} ] ) then accept;
  };
  export none;
}

# The Kernel protocol is not a real routing protocol. Instead of communicating
# with other routers in the network, it performs synchronization of BIRD's
# routing tables with the OS kernel.
protocol kernel {
  scan time 60;
  export all;   # Actually insert routes into the kernel routing table
}

# The Device protocol is not a real routing protocol. It doesn't generate any
# routes and it only serves as a module for getting information about network
# interfaces from the kernel.
protocol device {
  scan time 60;
}
EOF

sudo systemctl enable bird
sudo systemctl restart bird
```

EXECUTAR NO MASTER:

```
# Fix Cluster Error Status
kubectl get componentstatuses
# If componenentes are Unhealthy:
sudo vi /etc/kubernetes/manifests/kube-scheduler.yaml
sudo vi /etc/kubernetes/manifests/kube-controller-manager.yaml
# Remove the '--port=0' line on the spec->containers->command
sudo systemctl restart kubelet.service
```

```
# Setup rook ceph on kubernetes
git clone --single-branch --branch v1.7.5 https://github.com/rook/rook.git
cd rook/cluster/examples/kubernetes/ceph
kubectl create -f crds.yaml -f common.yaml -f operator.yaml

# Descarregar esta versão do cluster-test.yaml:
# https://gist.github.com/jpcanoso/8a1ddfc2683cdd3fd8cb6d92062a8296
# E depois aplicar:

kubectl create -f cluster-test.yaml

# Get status (esperar por PHASE = READY) (vai demorar)
kubectl get cephcluster -A

# Create pod ceph tools (not needed)
# kubectl apply -f toolbox.yaml
# kubectl -n rook-ceph exec -it deploy/rook-ceph-tools -- bash
# $ ceph -s
# $ ceph osd status
# $ exit
# kubectl delete -n rook-ceph deploy/rook-ceph-tools

# Create StorageClass
kubectl create -f csi/rbd/storageclass-test.yaml
```

```
# Ativar metrics
# Descarregar yaml
# https://gist.github.com/jpcanoso/1de1b54b0d081e1596f05ced2b6259ab
# E aplicar

kubectl apply -f metrics.yaml
```

```
# INSTALAR o cert-manager
# (É seguro correr estes passos mesmo sem o dns configurado) :)
kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.5.4/cert-manager.yaml

# Continuar para o proximo passo apenas quando os pods estiverem ready (kubectl get pods -n cert-manager)
# Criar e aplicar o ClusterIssuer de staging
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    email: myemail@company.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret used to store the account's private key.
      name: example-issuer-account-key
    # Add a ACME HTTP01 challenge solver
    solvers:
    - http01:
        ingress: {}

# Criar e aplicar o ClusterIssuer de production
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-production
spec:
  acme:
    email: myemail@company.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret used to store the account's private key.
      name: example-issuer-account-key
    # Add a ACME HTTP01 challenge solver
    solvers:
    - http01:
        ingress: {}
```

```
# INSTALAR O Portainer
# Install HELM -> Run on MASTER
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
sudo chmod +x get_helm.sh
./get_helm.sh

# Patch storage class
kubectl patch storageclass rook-ceph-block -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'

helm repo add portainer https://portainer.github.io/k8s/
helm repo update
helm install --create-namespace -n portainer portainer portainer/portainer --set service.type=ClusterIP

# Create ingress
cat <<EOF | tee ./portainer-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: portainer-ingress
  namespace: portainer
spec:
  rules:
  - host: portainer.casa.canoso.pt
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: portainer
            port:
              number: 9000
EOF
kubectl apply -f portainer-ingress.yaml
```

```
# MONITORING STACK
kubectl create ns monitoring

# Adicionar repo no helm
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

# adicionar secret username, pw
kubectl create secret generic -n monitoring haproxy-credentials --from-literal=admin=$(openssl passwd -1 mermaid)

# Save the file as values.yaml
https://gist.github.com/jpcanoso/dcf99847bf4055864728a2b18350cf75

# Edit the file:
# lines 229 -> alertmanager domain
# line 650 -> grafana dashboard password
# lines 675 -> grafana domain
# lines 1920 -> prometheus domain

helm install prometheus -n monitoring -f values.yaml prometheus-community/kube-prometheus-stack
```

```
Importar Dashboards no grafana:
Aceder ao grafana, efetuar login
Na barra lateral ir ao + (Create) -> Import
Importar os IDs:
12693
11328
1860
2842
e seleccionar o Prometheus como source
```

```
# Ativar o Monitoring no CEPH
cd rook/cluster/examples/kubernetes/ceph/monitoring
kubectl create -f service-monitor.yaml
kubectl create -f prometheus.yaml
kubectl create -f prometheus-service.yaml
```

```
# ARGOCD
kubectl create ns argocd
helm repo add argo https://argoproj.github.io/argo-helm
helm repo update

# Download raw as argovalues.yaml
# https://gist.github.com/jpcanoso/0711c7bd2dd3db428c39609efca2b473

helm install argocd -n argocd -f argovalues.yaml argo/argo-cd

# Create ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
  name: argocd-ingress
  namespace: argocd
spec:
  rules:
  - host: argocd.mermaid.dei.uc.pt
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: argocd-server
            port:
              number: 443
  tls:
  - hosts:
    - argocd.mermaid.dei.uc.pt
    secretName: argocd-cert

# Get dashboard password
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
```

FIM do Setup

Ingress

```
# Prometheus Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
    haproxy.org/auth-realm: Authentication Required
    haproxy.org/auth-secret: monitoring/haproxy-credentials
    haproxy.org/auth-type: basic-auth
    haproxy.org/ssl-redirect: "false"
  name: prometheus-kube-prometheus-prometheus
  namespace: monitoring
spec:
  ingressClassName: haproxy
  rules:
  - host: prometheus.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: prometheus-kube-prometheus-prometheus
            port:
              number: 9090
        path: /
```

```
        pathType: Prefix
  tls:
  - hosts:
    - prometheus.mermaid.dei.uc.pt
    secretName: prometheus-cert

# Grafana Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
  name: prometheus-grafana
  namespace: monitoring
spec:
  ingressClassName: haproxy
  rules:
  - host: grafana.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: prometheus-grafana
            port:
              number: 80
        path: /
        pathType: Prefix
  tls:
  - hosts:
    - grafana.mermaid.dei.uc.pt
    secretName: grafana-cert

# Alertmanager Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
    haproxy.org/auth-realm: Authentication Required
    haproxy.org/auth-secret: monitoring/haproxy-credentials
    haproxy.org/auth-type: basic-auth
    haproxy.org/ssl-redirect: "false"
  name: prometheus-kube-prometheus-alertmanager
  namespace: monitoring
spec:
  ingressClassName: haproxy
  rules:
  - host: alertmanager.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: prometheus-kube-prometheus-alertmanager
            port:
              number: 9093
        path: /
        pathType: ImplementationSpecific
  tls:
  - hosts:
    - alertmanager.mermaid.dei.uc.pt
    secretName: alertmanager-cert

# Prometheus Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
  name: portainer-ingress
  namespace: portainer
spec:
  rules:
  - host: panda.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: portainer
            port:
              number: 9000
        path: /
```

```yaml
        pathType: Prefix
  - host: portainer.mermaid.dei.uc.pt
    http:
      paths:
      - backend:
          service:
            name: portainer
            port:
              number: 9000
        path: /
        pathType: Prefix
  tls:
  - hosts:
    - portainer.mermaid.dei.uc.pt
    secretName: portainer-cert

# Ceph Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ceph-ingress
  namespace: rook-ceph
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-production
spec:
  rules:
  - host: ceph.mermaid.dei.uc.pt
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: rook-ceph-mgr-dashboard
            port:
              number: 7000
  tls:
  - hosts:
    - ceph.mermaid.dei.uc.pt
    secretName: ceph-cert
```