UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecos BCN

Analysis And Practice Of Micro-Services Deployment Technologies based on Containers

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Jesús López Pocino

In partial fulfilment

of the requirements for the degree in

Bachelor's degree in Telecommunications Technologies and Services ENGINEERING

Advisor:Jose Luiz MuñozTapia

Barcelona, January 2021

Abstract

This project's target is the investigation and practice of micro-services deployment technologies based on containers from the point of view of a growing organization that never had before work with containers. Representing the journey from the old way of developing apps and conceiving architectures to the containerized one of the current days.

In the beginning, we will deal with small-sized projects, but as they grow, and their requirements change, the analysis and evaluation of the container technologies available would be essential to choose the best solution. The thesis will focus on Docker and Kubernetes plus the various ways to implement them and the vast market options to complement them.

To do that, we have arranged the thesis in two significant parts: The theoretical basis is where we are going to examine the components and complements behind these technologies analyzing the state or the art.

And the practical cases are where we will produce some setups to represent different size projects and specifications. And then analyze the product obtained and its results.

Resum

L'objectiu d'aquest projecte és la investigació i pràctica del desplegament de microtecnologies basades en recipients des del punt de vista d'una creixent organització que mai havia treballat amb contenidors. Representant el viatge des de l'antiga forma de desenvolupar i concepció d'arquitectures fins a la via actual usant contenidors.

Al principi, tractarem projectes de mida petita, però a la vegada que creixen, i els seus requisits canvien, l'anàlisi i avaluació de les tecnologies de contenidors disponibles serà clau per triar la millor solució. La tesi se centrarà en Docker i Kubernetes, a més de les diverses opcions per implementar-los i els complements disponibles en el mercat.

Per fer-ho, hem organitzat la tesi en dues parts clarament remarcades:

La base teòrica, on examinarem els components i els complements darrere aquestes tecnologies analitzant l'estat en el qual es troba la tecnologia.

I els casos pràctics, on reproduirem algunes configuracions per representar diferents mides de projectes i especificacions . Per després analitzar el producte obtingut i els seus resultats.

Resumen

El objetivo de este proyecto es la investigación y práctica del despliegue de microtecnologías basadas en contenedores desde el punto de vista de una organización en crecimiento que nunca antes había trabajado con contenedores. Representando así el viaje des de la antigua forma de desarrollo y concepción de arquitecturas hasta la manera actual usando contenedores.

En un principio, nos ocuparemos de proyectos de pequeño tamaño, pero a medida que crecen, y cambian sus requisitos, el análisis y la evaluación de las tecnologías de contenedores disponibles serán clave para elegir la mejor solución. La tesis se centra en Docker y Kubernetes, además de las diversas formas de implementarlos y también las amplias opciones del mercado para complementarlos.

Para ello, hemos organizado la tesis en dos partes marcadas:

La base teórica, donde vamos a examinar los componentes y complementos detrás de estas tecnologías analizando el estado actual del arte.

Y los casos prácticos, donde produciremos algunas configuraciones para representar diferentes tamaños de proyectos y especificaciones. Para después analizar el producto obtenido y sus resultados.

Acknowledgements

I want to take this opportunity to express my gratitude to my supervisor Jose Luis Muñoz, who offered me this project and made it possible with his advice during the realization.

I knew José from having had him as a professor at the ETSETB, and I cannot be happier with the decision I made to get involved with him to realize the final degree thesis.

Thanks to Rafa Genés Duran, a PhD student from ISG, he provided us with the remote machine and technical aid when needed.

Finally a big thanks to Lluís Baró Cayetano for being such a good co-worker and helping me when necessary.

Revision history and approval record

Revision	Date	Purpose
0	01/12/2020	Document creation
1	03/12/2020	First Draft of the Contents Organization
2	03/12/2020	Introduction
3	07/12/2020	Docker Documentation
4	21/12/2020	Kubernetes Documentation
5	30/12/2020	Completion of empty chapters
6	01/12/2020	SetUp, result and conclusions
7	07/01/2021	Document completion
8	10/01/2021	Last changes

Document Distribution List

Name	e-mail
Jesús López Pocino	
Jose Luis Muñoz Tapia	

Written by:		Reviewed and approved by:	
Date	08/01/2021	Date	Last Change Date
Name	Jesús López Pocino	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Table Of Contents

Abstract.....2

Resum.....3

Resumen.....4

Acknowledgements.....5

Revision history and approval record.....6

Document Distribution List.....6

Table Of Contents.....7

List Of Figures.....9

List Of Tables.....11

Glossary.....12

1. Introduction.....13

1.1. Statement of purpose.....13

1.2. Requirements and specifications.....14

1.3. Methods and procedures.....14

1.4. Work Plan.....16

1.5. Incidences.....21

1.6. Deviations.....22

2. State of the art:.....23

2.1. Containers.....23

2.2. Docker.....26

2.3. Basic Container Orchestrators.....30

2.4. Container Orchestrators: Docker Swarm vs Kubernetes.....32

2.5. Container Orchestrators:Kubernetes.....34

2.6. Kubernetes Complements.....44

3. Project Development:.....46

3.1. Communication....46
3.2. Software....46
3.3. Documents Deliveries....47
4. Setups....48
4.1. Docker & Docker-Compose....48
4.2. Kubernetes Resources....54
4.3. Minikube....57
4.4. Microk8s....61
4.5. Kubeadm....64
5. Results....74
5.1. Docker & Docker-Compose....74
5.2. Minikube....76
5.3. Microk8s....77
5.4. Kubeadm....79
6. Budget....80
7. Conclusions....81
8. Future Development....83
References....84

List Of Figures

Figure 1: Work Breakdown Structure.....16

Figure 2: Monolithic vs Micro.....23

Figure 3: Containers vs VM.....24

Figure 4: Syntax Example of Dockerfile.....26

Figure 5: docker-compose file example.....30

Figure 6: Docker Swarm vs Kubernetes.....32

Figure 7: Kubernetes Worker Node.....35

Figure 8: Kubernetes Master Node.....36

Figure 9: LENS Menu Toolbar.....45

Figure 10: Web App.....50

Figure 11: User Data Stored.....50

Figure 12: Our Web App Dockerfile.....51

Figure 13: DockerHub repository.....51

Figure 14: Docker-Compose File.....53

Figure 15: ConfigMap & Secret YAML.....54

Figure 16: PersistentVolume & PersistentVolumeClaim YAML.....55

Figure 17: MongoDB YAML.....55

Figure 18: App Web Kubernetes Components Deployed.....58

Figure 19: Minikube Cluster Info in LENS.....59

Figure 20: App Web Kubernetes Services.....60

Figure 21: Number of Web-App Replicas.....63

Figure 22: Grafana Login.....72

Figure 23: Grafana Worker Node CPU Usage.....72

Figure 24: Grafana CPU Usage Graphic Colors code.....72

Figure 25: Cluster Wide Resource Chart.....73

Figure 26: Node resources status.....73

Figure 27: Docker Scenario.....74

Figure 28: Docker Desktop Windows.....75

Figure 29: Minikube Cluster with K8s Components.....76

Figure 30: MicroK8s Cluster.....77

Figure 31: Kubeadm Kubernetes Cluster.....79

Figure 32: Equipment Summary.....80

Figure 33: Summary of Personal Salaries.....80

Figure 34: Hosting Summary.....80

List Of Tables

Table 1: Work Package 1.....17

Table 2: Work Package 2.....17

Table 3: Work Package 3.....18

Table 4: Work Package 4.....19

Glossary

FS: File System.

CRI: Container Runtime Interface

YAML: Yet Another Markup Language or Ain't Markup Language.

Manifest: Also know as YAML file. Contains K8s description.

K8s: Kubernetes, 8 are the letters between K and s.

LXC: Linux Container

LXD: Linux Container Hypervisor

CLI: Command Line Interface

IDE: Integrated development environment

CNI: Container Network Interface

1. Introduction

In recent years the way applications and services projects are developed changed drastically, with Micro-service Architecture and new methodologies, like Agile, established themselves as the standard. These news components also came with special needs like scalability, continuous integration continuous deployment, wide compatibility, and to fulfil them a wave of container-based technologies have been rising.

Our project will focus on the study and implementation of Docker and Kubernetes, two of the most used container technologies. These container technologies allow companies to increase their performance and use their resources more efficiently. That's why in this document, and after the theoretical foundation, we will evaluate the use of container technologies at different setups.

Also, as a complement to these technologies, we will look at some of the most useful tools the community build, enforcing the open-source spirit of Docker and Kubernetes.

1.1. Statement of purpose

The way projects are developed changed drastically when using containers technologies, and this project wants to expose all the advantages of using them.

The main objective is to analyse and study container technologies and understand how they impact and improve project development and maintenance.

Let's have a quick view on how project development changed:

- Before Containers:

  Development team produced artefacts with instructions and configurations on how to install them at the production environment

  Later the operations team had to set up the environment to install the artefacts and deploy them.

  This process of installation caused problems and time was lost installing everything needed.

- After Containers:

  Development and Operations team can work together to define the Container or the Image, and deploy it at the environment desired.

This work uses the container's portability and wide compatibility characteristic to deploy artefacts ready to run at the desired environment.

Cutting time lost installing/configuring/running binaries. We will see further in the 'SetUps' section, how using container solutions, allows the user to install complete functional applications with few commands.

Apart from affecting the developing process and posterior operations, these technologies come up with a deployment challenge. There are also various ways to use it and deploying it. A part of multiple architectures, tools, service providers.

And we want to have a close view of that all those types of deployment can offer.

The last and more general purpose of that study is to have sufficient knowledge to decide within the vast possibilities which type of solution is better in each of the setups we ran. Allowing us to have a real vision of the requirements and blockers a real developer team can found.

1.2. Requirements and specifications

Docker and Kubernetes have wide compatibility with the more used operating systems, Windows, Ubuntu and macOS. And therefore, the experiments and studies carried out are compatible with any of these OS, reinforcing the characteristic of wide compatibility.

In our case, for the project base, we used Linux like machines and LXC containers to simulate the devices that take part in the different setups.

The project base requirements are:

- One Linux distribution, preferably Ubuntu 20.04.
- Updated version of library LXC(Linux Containers).
- Docker Requirements [1]
  - Docker-Compose and Docker-Swarm depend on Docker
- Kubernetes requirements [2]

1.3. Methods and procedures

This project is based on the DevOps (Development and Operations) development method and uses Docker and Kubernetes container management tools as the core part of the thesis.

Both technologies are open-source, and they have a large community of users and developers who create software in plug-in or tool form for use in conjunction

with Docker or Kubernetes. And so we will use some of that software developed by the community as well.

1.3.1 Software

List of the main used software:

- **Docker:** The simplest and most widely used container construction and management tool.
  - Docker-Compose: It is part of the Docker environment. Is a tool for defining and running multiple containers at the same machine.
  - Docker-Swarm: It is part of the Docker environment. Is a simple tool for managing clusters of containers across multiple hosts.
- **Kubernetes:** Complex tool for Container Cluster Management. It has a wide range of exclusive functionalities and is the most used.
  - Minikube: Tool to easily locally run a cluster with one node using Kubernetes.
  - Microk8s: Simplest and lightweight Kubernetes deployment.
  - Kubeadm: It is part of the Kubernetes environment. Fast paths for creating Kubernetes clusters with the total of Kubernetes on it.
- **LENS:** Kubernetes IDE used for managing the Cluster
- **VSCode:** Used for writing the YAML files. Files used for starting Kubernetes resources and components.
- **Terminal:** For managing cluster and configure SetUps.

1.3.2 Documentation

The deliverable documentation carried out during the thesis is based on the templates provided by the university. All the text edition has done with LibreOffice.

Once done, the documentation has submitted to the project advisor for verification and evaluation.

1.3.3 Communication

In matters of communication, due to the pandemic and for commodity, all had been done in remote, using call applications such as Google Meet or Jitsi Meet.

Doing the stand-up meetings, we used a remote desktop hosted at a university machine that allowed us to share screen, view and edit in the same workspace and share docs and code.

1.4. Work Plan

Although this project consists of two differentiated parts, Theory and Practice, the work plan was conceived so that the theory and practice of each section were covered linearly. Thus, the different points have been covered one after the other to focus the resources and be able to export the more excellent knowledge of one to another since they share the same principles.

A quick summary of the main work packages:

- Docker: Study and Analysis of Docker
- Kubernetes: Study and Analysis of Kubernetes
- Deployments: Tools and predefined Kubernetes implementation
- Complements: Study and Analysis of the community's complements to improve the developer experience with the mentioned technologies.

1.4.1 Work Breakdown Structure

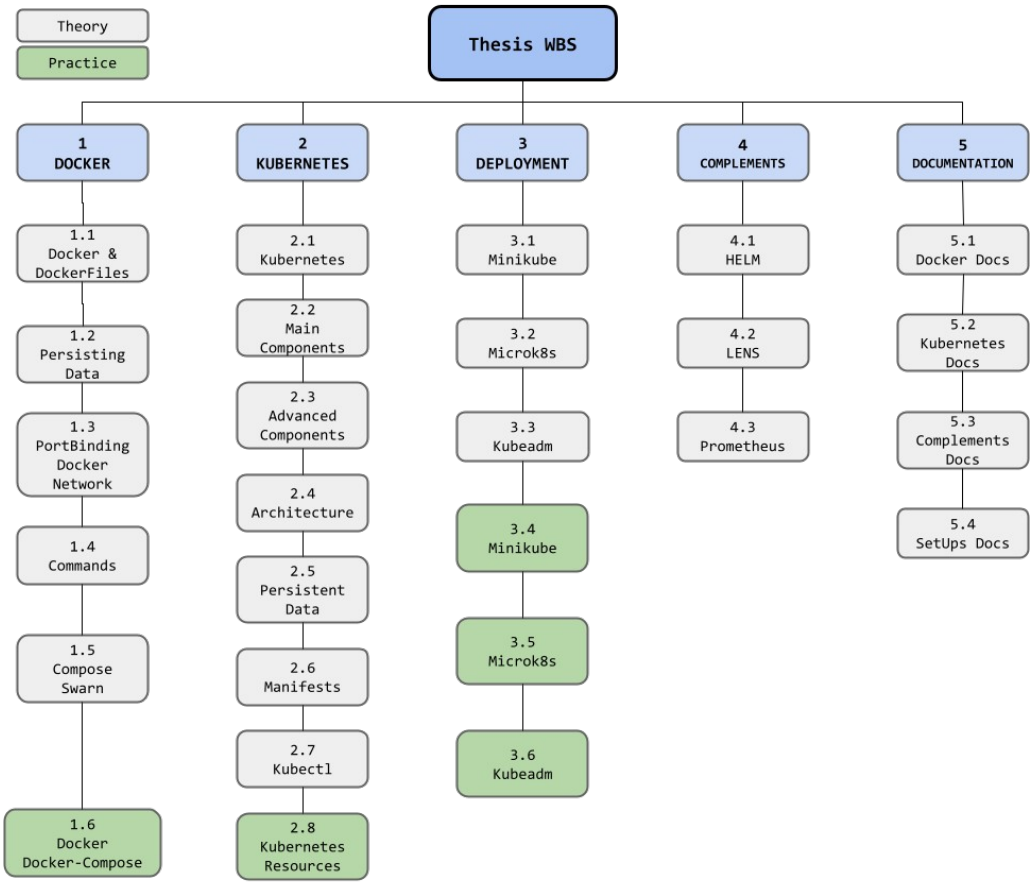


Figure 1: Work Breakdown Structure

Table 1: Work Package 1

Project: Docker	**WP ref:** 1
Major constituent: Software	
Short description: In this package, we will study and analyze what Docker, and his related tools, have to offer to the DevOps process. Primary information source YouTube and the slide the supervisor gave access.	**Start date:** 20/08/2020 **End date:** 11/09/2020
Internal task T1: Understand what the main components of Docker are and how to use Dockerfiles.	
Internal task T2: Acquire the knowledge of how to provide Docker with persisting data.	
Internal task T3: Learn how the Container Networking works and how the ports are assigned.	
Internal task T4: Acquire sufficient knowledge to list all the most used Docker commands.	
Internal task T5: Get a surface knowledge level of docker swarm. As it may be not crucial as Kubernetes but still have a role in the containers orchestrators.	
Internal task T6: Design and implement a setup to use all the knowledge acquired during the realization of this work package.	

Table 2: Work Package 2

Project: Kubernetes	**WP ref:** 2
Major constituent: Software	
Short description: In this package, we will study and analyze what Kubernetes and his related tools have to offer to the DevOps process while realizing academic slides.	**Start date:** 14/09/2020 **End date:** 22/10/2020

Internal task T1: Acquire the overall knowledge of Kubernetes tool.	
Internal task T2/T3: Get deep into Kubernetes components as they are the main reason it is so powerful and have so many features.	
Internal task T4: A quick analysis of the Kubernetes Node Structure.	
Internal task T5: Study of how to enable persistent data at Kubernetes.	
Internal task T6: Study the syntax and redaction of Manifests. They act like Dockerfiles do for Docker.	
Internal task T7: Acquire acknowledgement of the Kubernetes CLI tool.	
Internal task T8: Design of the Kubernetes resources used at our future SetUps that involve k8s.	

Table 3: Work Package 3

Project: Deployment	**WP ref:** 3
Major constituent: Software	
Short description: Study and Analysis of the various ways and tools for implementing Kubernetes.	**Start date:** 09/11/2020 **End date:** 26/11/2020
Internal task T1: Acquire the necessary knowledge for running Minikube tool.	
Internal task T2: Acquire the necessary knowledge for running Microk8s tool.	
Internal task T3: Acquire the necessary knowledge for running Kubeadm tool.	
Internal task T4: Design and implementation of the SetUp that involve Minikube.	

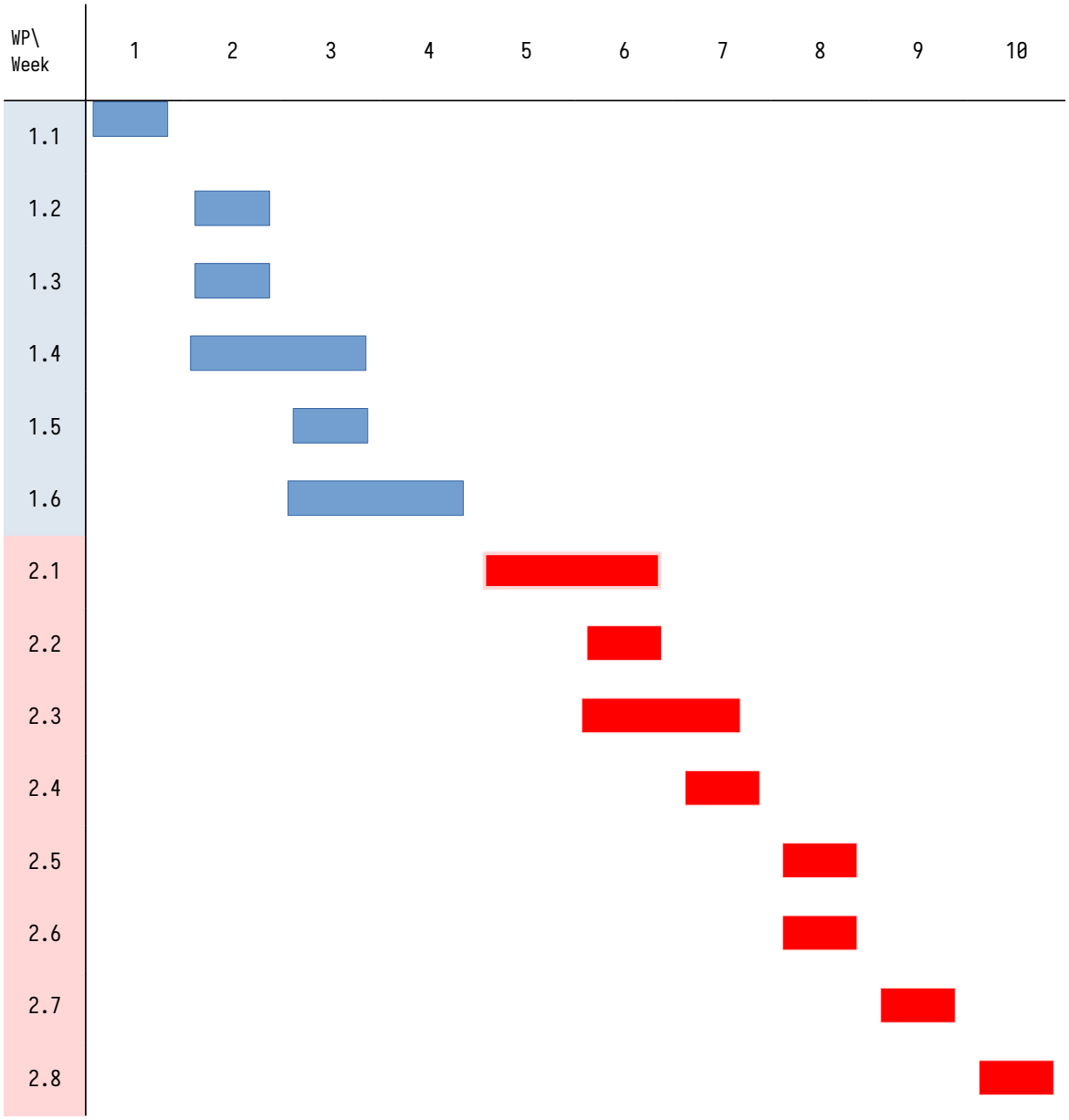
Internal task T5: Design and implementation of the SetUp that involve Microk8s.	
Internal task T6: Design and implementation of the SetUp that involve Kubeadm.	

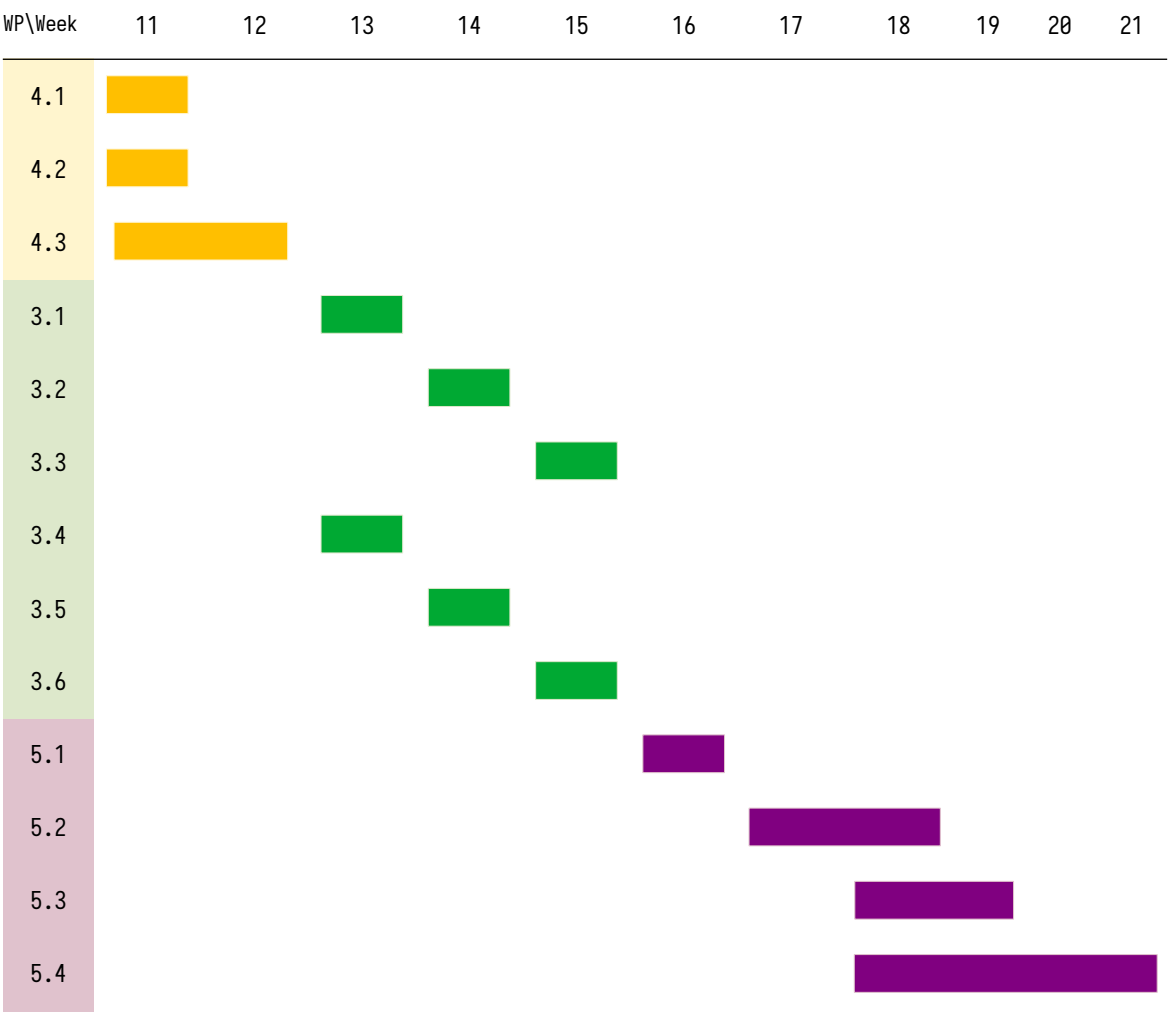
Table 4: Work Package 4

Project: Complements	**WP ref:** 4
Major constituent: Software	
Short description: This is considered a lightweight package and it's conceived to get in touch with most of the used community's designed complements for Docker and Kubernetes.	**Start date:** 26/10/2020 **End date:** 06/11/2020
Internal task T1: Quick review and study of the Kubernetes package manager HELM.	
Internal task T2: Quick review and study of the Kubernetes IDE Lens.	
Internal task T3: Quick review and study of the Kubernetes monitoring tool Prometheus.	

Project: Documentation	**WP ref:** 5
Major constituent: Documentation deliverables	
Short description: This WP is dedicated to the redaction of the thesis document and the reviews.	**Start date:** 04/12/2020 **End date:** 10/01/2021

1.4.2 Gantt Diagram





1.5. Incidences

During the development of the project, we had a couple of remarkable technical incidences.

The first was that the advisor's development environment, which was a host machine within the school network, broke down. A power outage caused this due to maintenance of the campus electrical network. As the host was a virtual machine nested, once the power re-established he couldn't boot correctly and so we had to put it down, losing some of the work developed there. As a solution to that, we configured our desktop computer with the tools needed to implement and develop the project. We were lucky there, as my computer had enough resources to handle the requirements of the setups we ran. And from there originated the other incidence.

Due to my inexperience working with LXC and the other technologies, I did make a wrong configuration and trying to fix it, I broke my entire OS and had to reinstall it again. Once again losing part of the work done.

1.6. Deviations

At first, this project was thought to get deep in Kubernetes and some of his resources and components. To get more knowledge possible on Kubernetes and how it is related to Cloud Networking.

As the project progressed, it took us longer than expected with Docker, since it was deeper than we had originally supposed. And we still had to get started with Kubernetes.

Seeing that we realized that with our lack of starting knowledge on the matter, we might have been ambitious thinking that we could understand Kubernetes quicker.

So, we decided to change the vertical approach of the beginner to one more wide, which included Docker, Kubernetes plus some complements and tools.

2. State of the art:

In this chapter, we will break down all the components that are part of our thesis focused on improving the development and deployment of services and applications, taking advantage of new technologies.

Due to the need for an in-depth study and our thesis's theoretical focus, this chapter is somewhat dense. However, in the end, it will make us aware of a large number of options in the current market and the importance of conducting a good study to opt for the solution that best suits our needs.

2.1. Containers

Around 2008, the introduction of C-groups(control groups) to the Linux Kernel and the conception of the LXC(Linux Containers) technology with them, caused the rise of virtualized OS(operating systems) and processes[3].

Quickly people realized the potential of virtualization applied to the new tendency of Micro-services Architecture. This new architecture was quickly surpassing the old Monolithic architecture, which was unable to fulfil the needs of the new way of developing and deployment services and applications. The primary needs are:

- Continuous Integration: Simple change integration and version update. Allowing for a constant integration and development of the application.

- Scalability: Easily scalable components. We can increase the number of containers without compromising the integrity of the ones already deployed.

- New Technologies Friendly: Able to apply new technologies without doing a high load of work at the already deployed architecture.

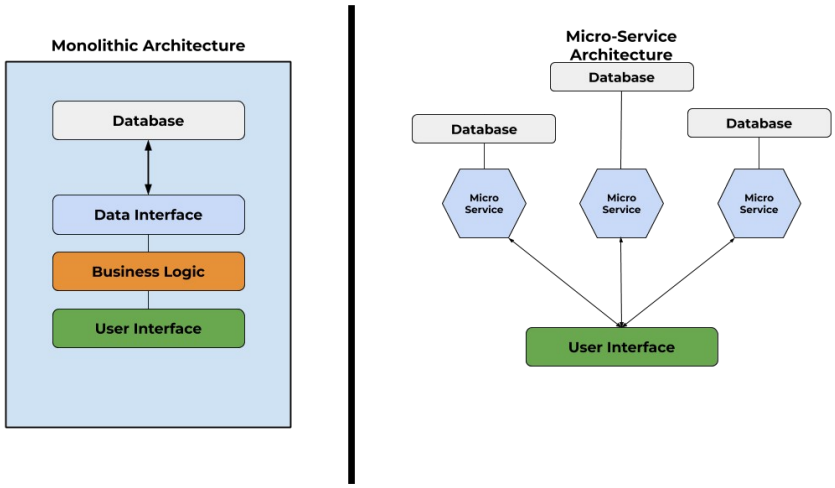


Figure 2: Monolithic vs Micro

With those needs in mind, and with the new technologies rising, the community found two solutions: Containers and Virtual Machines. Containers are an artefact with an entire runtime software package, containing an application plus all its dependencies, libraries, other binaries, and configuration files needed to make it work quickly. This makes containers wide compatible with all kind of equipment environment. The package created can be called Image, which is the Container description before being executed at container runtime.

An image is a bunch of binaries on a file system that stores all necessary to run and mount the containers. So basically, they are the foundation on what the Container is created at runtime.

Unlike Virtual Machines(VM's), they do not bundle full operating system, they are just an abstraction at the app layer, virtualizing the OS. This characteristic makes containers efficient, lightweight and assures that the software contained will always run the same regardless of where it is deployed.

Most of them are Linux containers. In this thesis, we do not use VM's, so there is no point in explaining them, but what is worth to know about them is how they compare with Containers.

Containers

- OS level virtualization
- Tens of MB
- Resources shared between containers
- Process Isolation

Virtual Machines

- Hardware level virtualization
- Tens of GB
- Resources assignation for each
- System Isolation

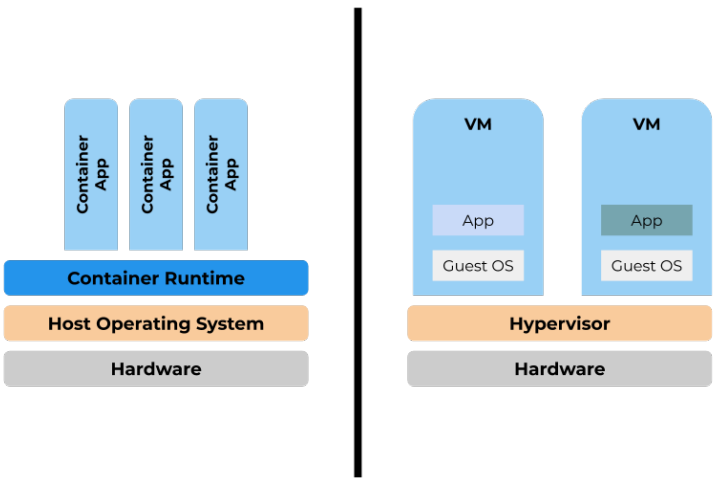


Figure 3: Containers vs VM

In the end, Containers amply meets the need of Micro-service Architecture, having wide compatibility and cutting the time lost installing and running binaries and software packages, plus being less resource demandant.

2.2. Docker

In this chapter, we are going to study and analyze all the principal components that Docker have.

Docker is a container virtualization runtime open-source tool made for simplifying development, deployment and execution of the software packages called containers. Specifically, Linux based containers. Apart from creating containers, Docker also manages them.

However, even though Containers can virtualize the entire operating system, Docker has specialized in running single processes inside their containers. Making them more lightweight, portable and scalable.

2.2.1 DockerFiles

As we saw before, a container needs an image to run, and Docker Containers are no exception. The image blueprint at Docker is called Dockerfile. Which written with a specific syntax, can be build to an Image. These Images can be found at repositories, private or public. The most used is DockerHub[4].

Docker is the main repository to find all kind of application images like GitLab is for sharing code. These companies and users can upload to a repository their images for everyone use it. For example, we can found the NginX Server Image or MySQL Image provided by their companies. Like GIT sites, there are public and private repositories. So companies can develop their images to use them in their projects. This is the option we will use further at the chapter of setups.

To have an image created available at DockerHub, an Image has to be built, tagged and pushed to the repository.

2.2.1.1 Syntax

This is set to study the basic syntax of a dockerfile, based on 'INSTRUCTIONS arguments'.

Let have a look into a Dockerfile that runs a NodeJS app:

```
FROM node:13-alpine

ENV MY_VAR=1234

RUN mkdir -p /home/somefolder

COPY ./code /home/somefolder

WORKDIR /home/somefolder

RUN npm install

CMD ["node", "server.js"
```

Figure 4: Syntax Example of Dockerfile

The first line always has to be 'FROM image:version'. This field declares the origin image from what the image based on the Dockerfile will be created.

- If none version is specified, Docker automatically will look for 'latest'.
- If none repository is referenced, Docker automatically will look at DockerHub.

In this case, we are building on a NodeJS image, version 13-alpine, which has already installed NodeJS.

ENV: Where we can configure environmental variables to use within the container. Useful to help at the configuration of the container. Some images specify if they need any ENV variable to be defined in order to run the container.

RUN: Used by executing any command inside the container. As Docker uses Linux containers, these commands are Linux commands.

COPY: A copy command executed on the Host allowing to transfer files from Host to the Container environment. Useful when having app-code that need to be inside the container in order to start it.

WORKDIR: Sets the working directory for any instructions that follow it in the Dockerfile. If it does not exist, it will be created even if it is not used in any subsequent Dockerfile instruction.

CMD: Command, execute commands at container entry-point. The main difference with **RUN**, is that CMD is unique, we can only have one.

Instructions CMD and RUN, have two forms available:

- Shell form: RUN <command>
- Execution form: RUN ["executable", "param1", "param2"]

For more information on Dockerfiles visit docs.docker.com[5].

2.2.2 Persisting Data

Every time a container is reset or removed the data it contains is lost forever, as the data is stored at a virtual files system.
To solve this, Docker uses Volumes, which means plug-in the file system of the container to a physical file system at the host machine or even to one at the cloud. Meaning that if a data change occurs in either file system, it gets replicated to the other.
If we create a new container bound to an existing volume with old data from a removed container, the new will have the data.

The available volume types are:

1. **Host Volume:** You decide where on the host FS the reference is made.
2. **Anonymous Volume:** Let docker automatically create the volume by only specifying the container directory to mind. Docker will create it at */var/lib/docker/volumes*
3. **Named Volume:** Same as Anonymous but with the name specified.

2.2.3 Docker Networks

By default, Docker creates his Isolated Docker Network where the containers are running in. Inside the Isolated Docker Network, the containers can talk to each other using just the container name instead of URL.

With Docker CLI, we can manage those networks:

- **List:** List the available networks.

  `$ docker network ls`

- **Create:** Create a docker network.

  `$ docker network create my-network`

- **Connect:** Connect a container to an existing network.

  `$ docker network connect my-network myContainer`

2.2.4 Main Commands

After understanding what a Container is and knowing how to create them with Docker, let us see how to use and manage them with the Docker CLI, and its main commands:

- **Images:** List images that had been already pulled.
- **PS:** List all running containers with useful information.
- **PULL:** Pull an image from a repository.

  `$ docker pull [OPTIONS] image:tag`

  `$ docker pull ubuntu: 20.04`

  - tag(version) is optional, by default Docker pulls the latest.
  - With the flag –all-tags,-a we can pull all the available tags

- **START:** Start one or more stopped containers.

  ```
  $ docker start [OPTIONS] containerName or containerID
  ```

  ```
  $ docker start ubuntu:20.04
  ```

  - --attach,-a: Attach STDOUT/STDERR and forward signals.
  - --interactive,-i: Attach container's STDIN.

- **RUN:** Run first pull the image and creates a writeable container layer over the specified image and then starts it using the specified command at the [OPTIONS] command argument.

  ```
  $ docker run [OPTIONS] image:tag
  ```

  ```
  $ docker run --name myredis -ip 192.168.0.45 redis:alpine
  ```

  - -p hostPort:containerPort: Port binding between host and the container
  - --net: To connect the container to an existing network.
  - -v hostDirectory:virtualDirectory: Bind the volume into a directory
  - -e : Stands for environmental variable. Some images need ones to start the container properly.

- **STOP:** Stop the container. Useful when we want to do a full reset a container not working as indeed.

  ```
  $ docker stop [OPTIONS] containerName or containerID
  ```

  ```
  $ docker stop myredis
  ```

- **LOGS:** This command allows us to retrieve logs present at the time of execution. Helping to debug errors when containers are failing to start.

  ```
  $ docker logs [OPTIONS] containerID or containerName
  ```

  ```
  $ docker logs --timestamps myredis
  ```

We will use these commands later in the SetUps chapter. For more information on Docker CLI check docs.docker.com[6].

2.3. Basic Container Orchestrators

All we have seen so far with Docker is the creation and running of containers in a linear manner. However, what if and application needs more than one container running at the time to be able to run? That is why we have Docker-Compose.

2.3.1 Docker Compose

Docker Compose has been created by Docker Inc and is highly integrated with the use of Docker Containers. DCompose allows us to start several containers simultaneously on our machine using a docker-compose file and manage them easily with commands from the terminal.

Docker-Compose file, like Dockerfiles, has its own syntax which helps us defining and running complex applications, like this:

```
version: "3"

services:
  mongodb:(1)
    image: mongo:latest (2)
    Ports: (3)
      - 27017:27017
    Environment: (4)
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=1234
    Volumes: (5)
      - mongo-data:/data/db
  mongo-express:
    image: mongo-express
    ports:
      - 8080:8081
    depends_on: (6)
      - mongodb
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=1234
      - ME_CONFIG_MONGODB_SERVER=mongodb
Volumes: (5.1)
  mongo-data:
    driver: local
```

Figure 5: docker-compose file example

- Version is always there, and define the docker-compose version we are gonna use.
- Services is also always there. It is the list of the containers we are gonna run with docker-compose.
- For each defined service there are a number of fields to configure them. Let's see the most used:

1. The name of the container, in our case ‘mongodb’. The final name of the container, considering the file is inside ‘myfolder’ will be: myfolder_mongodb_1

2. Image from which the container will be created. Same way as for Dockerfiles.

3. Ports configured. ‘hostPort:containerPort’.

4. Environmental variables. They have the same use as the ones at Dockerfiles.

5. Volume definition for persisting data

   5.1. If a volume has been defined for any of the services, it needs to be defined at the end of the docker-compose file. Here we can add characteristics, like in this case indicate the storage type.

6. It’s possible that some containers need to have another container running, so with this instruction, the service will not be build until the container which it depends is build.

To enable the communication between containers, DCompose does something similar to Docker, creates a shared network for the containers defined at the docker-compose file.

Worthy to say that as Docker-Compose is based on Docker, all the Containers, Networks, and other Docker components are manageable with the Dockers commands we saw previously.

2.4. Container Orchestrators: Docker Swarm vs Kubernetes

As we exposed at the start of the thesis, the trend from Monolith architecture to micro-services increases container technologies usage. Ending with applications and services based on more that one container.

With what we have seen so far with Docker and Docker-Compose, performing these apps' deployment and maintenance begins to be somewhat tedious. That is why we have Cluster Managing Tools. Docker is a tool for configuring, building and distributing containers while Cluster Managing tools take care after the container has been deployed, scheduling, scaling and managing its deployment.

Before getting deep into Kubernetes, as one of the principal technology to study, we will take a quick look at Docker Swarm, and compare them.

Docker Swarm is the native clustering engine for and by Docker. All the components that run with Docker run equally in Swarm as they use the same command line. Knowing this, let see him versus Kubernetes:

Docker Swarm	Kubernetes
• Easy Installation	• Complex Installation
• Easy to use	• Hard to use
• Manual Scaling	• Auto-Scaling
• Uses Docker CLI	• Native support for monitoring
• Lightweight	• Wide integration with Cloud Providers
• Suitable for small environments	• External CLI
	• Bigger Community = more features

Figure 6: Docker Swarm vs Kubernetes

From the comparative, we can establish that Swarm is better at smaller and simple environments and lower resources, as it is easily installed. Nevertheless, if the number of nodes is significant, and we need more features, Kubernetes is the option.

Even though Kubernetes only has the upper hand at more significant sized environments, why is the most used? Big community plus more, customizable features and a powerful CLI called Kubectl are the answer.

The big community behind Kubernetes make it a more viable option, as a more significant community means more people developing add-ons, extras and tools. Just take, for example, LENS IDE or microk8s deployment tool. A more significant community means that more people could answer our doubts or resolve our problems while working or developing with Kubernetes. All of this makes the technology advance faster than its competitors, achieving more useful features and versions.

Also, as we mentioned, the Kubectl CLI has a substantial upper hand versus Swarm. Swarm commands are related to Docker and can get somewhat larger and unintuitive. Meanwhile, Kubectl has its style, with a more intuitive syntax, more features and the possibility of being integrated with tools and complement the community developed for Kubernetes.

2.5. Container Orchestrators:Kubernetes

After knowing that Kubernetes is more suitable for wider environments and is the most used by the community, let us see WHAT Kubernetes is and why it is so special.
Kubernetes, also known as K8s because the eight letters between K and S, is an open-source platform framework for managing containerized workloads and services as clusters. Initially developed by Google, supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface)[7].

The main difference with Docker-Compose is that helps to manage containerized applications with several containers on a cluster with different environments(physical machines, VM, cloud).

Main features:

- **HA(High Availability)**: No downtime. Application is always accessible by the user.
- **Scalability:** High performance when scaling up(Vertically) and scaling out(Horizontal).
- **Disaster Recovery**: Possibility to backup and restore to previous states. The infrastructure has the mechanism to deal with errors and avoid loses. Furthermore, keep running on the last stable version.

After looking at the main features, let us see what makes Kubernetes unique and puts it ahead of its rivals.

2.5.1 Architecture

Kubernetes architecture operates with two types of nodes, MASTER and WORKER.

These nodes have inside them some processes that realize most of the manage and maintenance workload, guaranteeing the cluster's right behavior.

2.5.1.1 Worker Node Processes

Worker nodes contain three main processes that must be installed in every worker node. The Master uses these processes to manage and motorize them plus schedule container deployments in them.

1. **Container Runtime(CR):** Docker, Containerd, CRI-O. This needs to be installed as the nodes containing pods, which have containers running inside.

2. **Kubelet(Scheduler):** Proper of K8s. Is the process that schedules and manages those pods responsible for running/creating/starting a pod with a container inside. That’s why interacts with the container runtime and the machine(node) to get the configuration and assign the resources to the pods.

3. **KubeProxy:** Proper of k8s. Responsible for forwarding requests from services to pods. Has intelligent forwarding logic inside that makes sure that the communication works even with low overhead. (This logic, for example, priors pods on the same node where the request was created). Avoiding charging the network to much.

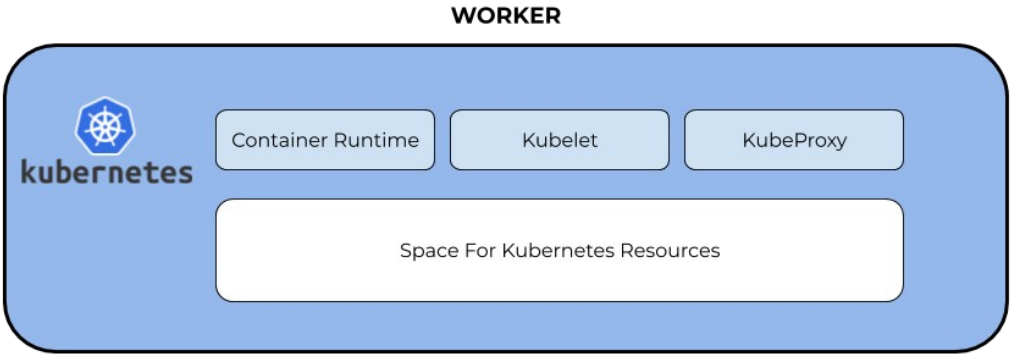


Figure 7: Kubernetes Worker Node

2.5.1.2 Master Node Processes

Mostly contains management processes. Four processes must run on every master node.

1. **API Server:** Cluster gateway. Which the client interacts with to manage the cluster.
    - Gatekeeper for authentication: Only authorized users are let into the cluster.
    - Validates request and forwards them to the other processes.
    - Only entry-point to the cluster, through UI, API or CLI.
2. **Scheduler:** Logic intelligent to decide on which worker node apply the changes.
    - List the resources of the change we want to create, and then evaluates the preferable node to receive the new pod.
    - Decides on which node a new pod will be created
    - New Pod REQ → API SERVER → Scheduler: Decides where → Kubelet(N-W)
3. **Controller Manager:** Detect cluster state changes(pod dying), and tries to recover as soon as possible from the effect of the change on the cluster state. To do that call the Scheduler that does its cycle again.

4. **etcd(key-value store):** Cluster brain. Every change in the cluster gets stored into etcd. Stores data that is used by Scheduler and the ControlManager in order to do their job. (resource available, state changes, pods created, cluster components). Does not contain application data.

In case we are asking if there can be more than one Master node in a big cluster, the answer is yes.

How? Summarizing, each of the masters will run their master processes, and they will share the data extracted of the status of the cluster. And then to manage the access to the cluster, the API server would act a LoadBalancer distributing the workload among the master nodes.

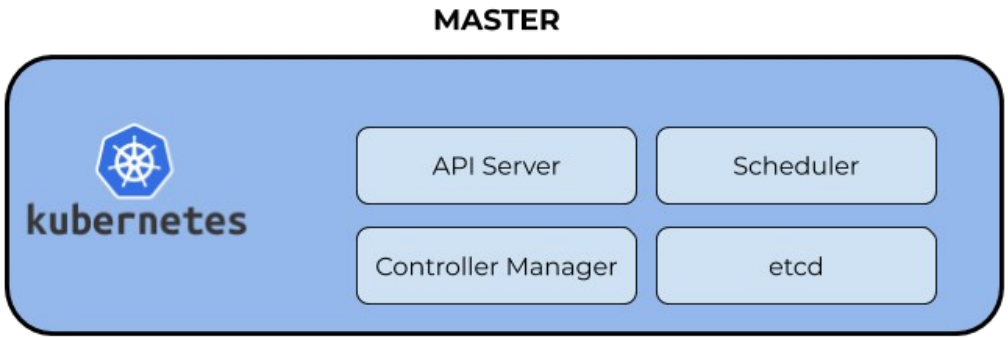


Figure 8: Kubernetes Master Node

2.5.2 Main Components

In this section, we will describe and explain the main components, also known as resources, that take part in a Kubernetes cluster. Just a reminder that Kubernetes is also called K8s because of the eight letters between K and S.

2.5.2.1 Pod

A Pod is the smallest unit of Kubernetes resource. Kubernetes creates an abstraction layer on the Container to deal/manage with the Container without interacting with the Container runtime(f.e. Docker) and only interacting with the K8s layer.

- Is meant to run one container(1 pod, 1 Container).
- Each Pod gets an IP address, which only accessible from inside the cluster.
- They are ephemeral. Can 'die' quickly, whenever an application crash happens or server out of resources, for example. However, on the other hand, they are replaced with the same facility.

2.5.2.2 Node

A node identifies a machine, server, Virtual Machine, Linux Container or anything Kubernetes is operating on and is part of the cluster.

Each node has its own identifier and can have as many Pods.

2.5.2.3 Service

Service is a resource for enabling communication between the Pod is serving and the cluster or even the cluster's exterior. Main characteristics:

- Provides a permanent IP address attached to each Pod.
- Life-cycle independent of the pod.
- They can be internal, communicating with the cluster. Alternatively, external, enabling communication from outside the cluster.
- Acts as a LoadBalancer, between pods available.

2.5.2.4 Config Map

It is like a '.properties' file, containing fields and values useful for the configuration of other Kubernetes resources. Allowing to apply changes at the running applications without rebuilding the images, for example in case some bind URL changes. The Pod will connect to it and extract the information.

One important thing is that it must be created before the K8s resource that uses it. From now on, we will see that some resources rely on others to be running already when they are deployed to the cluster.

2.5.2.5 Secret

Similar to ConfigMap but instead used to store private or confidential data. For example, authentication credentials o TLS certificates. This data is stored encoded with base64.

Important to know that storing sensitive data in a Secret does not automatically make it secure. There are built-in mechanisms, like encryption, for basic security, which are not enabled by default. As ConfigMap, it must be created before the K8s resource that uses it.

2.5.2.6 Deployment

Blueprint for Pods where we can specify how many replicas we want of a pod. The deploy will make sure to have as much as specified pods up and running, providing the cluster with the High Availability(HA).

If a Pod dies or is removed, Deployment will make another of the same specification.

Even when a Deployment is already deployed, we can scale up/down the number of replicas we are running.

It is important to know that it is heavily recommended not to use Deployment with Pods containing applications with state data or capacity to write into databases. This recommendation is for trying to avoid data inconsistencies at the cluster.

2.5.3 Advanced Components

2.5.3.1 DaemonSet

This resource is quite similar to Deployment, but instead of having some replicas, it ensures that all nodes within a configuration run a Pod copy. Even when a node is added after the daemon set is deployed, it adds a pod copy.

The other difference with the Deployment is that with Deployment, more than one Pod can be started at one Node, but with DaemonSet, we are sure that only one Pod will be at the Node.

2.5.3.2 StatefulSet

This resource is the solution to the data inconsistency of using Deployment with applications with state date.

It is explicitly designed to deploy applications such as databases or any app that needs to store data to keep track of a state. These applications must be created with StatefulSet, which has two principal characteristics:

- Responsible for replicating pods and putting them into operation.
- Ensures that reads and writes are synchronized, preventing data inconsistency and memory leaks.

The main difference with Deployments are that the PODS managed are not identical, they have an identity. So the creation and removal of them cannot be random, as it is vital to know which POD of the StatefulSet we are accessing each operation. This identity is kept if a POD is replaced by another.

StatefulSet achieves its characteristics, making one of the PODs the master and the rest their slaves. Master can read and write data, and the slaves only read.

Also, each POD has its own storage(volume), causing them to continuously synchronize their data to be up to date with the changes. The sync is done by the master every time he makes changes.

2.5.3.3 NameSpace

This k8s resource allows organizing the cluster components dividing the cluster into spaces, like creating virtual clusters inside the main K8s cluster.

This feature is quite useful in having multiple developing teams, as it is possible to configure isolated environments. Also, some resources are only visible from inside the NameSpace to mess with resources deployed by others.

By default, Kubernetes creates the following ones:

- **System:** Not meant for user to use it.
  - Contains system processes
  - Master Processes
  - Kubectl processes
- **Node-Lease:** Holds info about the nodes
  - Heartbeats of nodes
  - Availability of nodes
- **Public:** Contains public accessible data, even without authentication.
- **Default:** NS where the component created will deploy if other NS has not been defined.

2.5.3.4 Ingress

Ingress, also know as IngressRule, are used to create like an entry to an IP table, but for the K8s resource known as IngressController.

With the IngressRule the IngressController knows to which Service redirect the request for a host if the host is in any IngressRule. This gives to the Services and external access without having to expose themselves.

2.5.3.5 Ingress Controller

Evaluates and processes IngressRules, connecting requests to the services specified at the IngressRule.

As it forwards traffic into and out of the cluster, we can say that it is like an entry-point to the cluster. Supporting HTTP and HTTPS for more secure connections. However, even though the request received is HTTP, all the traffic inside the cluster is encrypted.

It is one of the few components that does not have a base version defined by the Kubernetes creator's team. Instead, Kubernetes creators support third-

Party Implementations. The most used implementations are the one done by Nginx, GoogleCloudPlatform of AmazonWebServices.

In our 'Setups' section, we will use the one from Nginx.

2.5.3.6 Virtual Networks

In Kubernetes, networking inside the cluster is provided by Container Network Interface (CNI). It consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, which are used in Kubernetes.

In our project, we are going to have a quick view of Flannel and Calico. Flannel is one of the simplest CNIs that can be installed in the cluster and is based on the CNI-plugin from the Cloud Native Computing Foundation which allows configuring network interfaces in Linux containers. It runs a small, single binary agent called flanneld on each host (it is deployed as DaemonSet). It is responsible for allocating a subnet lease to each host out of a larger, preconfigured address space.

The flanneld creates some route rules in the kernel's route table to forward traffic. Furthermore, that allows the intercommunication between nodes and resources.

On the other hand, Calico is more complex, adding more features relating to security. The security features are flexible, and they are encapsulation and native routing using BGP.

The main component created by is a Daemon Set in charge of creating 'calico-node' pods in each cluster node. Inside 'calico-nodes' we can found two resources, Bird and Felix, which are used to forward and receive networking information between the nodes.

To get more information visit the official sites of Flannel[8] and Calico[9].

2.5.4 Persistent Data

If some pod/app generates data and dies, all the data is gone. To avoid that, there is a component, called Volumes, like in Docker.

This type of storage is like an external hard drive plugged to the K8s cluster providing the K8s resources with a file system to store their data, as K8s does not manage data.

Some storage-type examples are the same machine where Kubernetes is running and cloud storage.

The storage, or volume, is independent on the pod life-cycle, as it is something external plugged into the cluster available for all nodes and resources.

Persistent data have two principal resources:

- Persistent Volume Component: resource used for defining storage that will be available for the cluster.
- Persistent Volume Claim Component: Used for defining a criteria that will look for a PersistentVolume that suits that. Is used by the resources that need a PersistentVolume to have one bind to them.

2.5.5 Kubectl

To control everything and interact with the cluster K8S has a command-line tool highly used that is called Kubectl. It is used for accessing the API Server at the Master nodes and give the user the possibility to interact with the cluster and manage it. Interacts with any type of cluster setup.

2.5.5.1 Main Commands

Kubectl commands and documentation have been used to do this section[10].

Create:

Commonly used for crating any type of k8s component. The most basic resource to do is a deployment from a manifest(YAML file).

```
kubectl create [component] [OPTIONS]
```

```
kubectl create deployment NAME --image=image -- [OPTIONS]
kubectl create deployment -f mymanifest.yaml
```

Edit:

Edit a resource from the default editor. Allows you to change the configuration of a resource directly from the CLI.

```
kubectl edit [resource Name] [OPTIONS]
```

```
kubectl edit my-deploy ( to edit a deployment called 'my-deploy'
```

Delete:

Delete resources of the cluster.

```
kubectl delete [resourceName] [OPTIONS]
```

```
kubectl delete my-deploy --force
```

GET:

Most important command in order to acknowledge the status of different k8s components of the cluster. Prints a table of the most important information about the specified resource/component.

```
kubectl get [resourceOptions] [OPTIONS]
```

```
kubectl get nodes (get information related to the nodes of the cluster, like their status),
```

LOGS:

Useful when working with apps to be able to debug any problems when deploying.

```
kubectl logs [resource] [OPTIONS]
```

```
kubectl logs podName
```

DESCRIBE:

Also useful when trying to debug pods or components and check their status.

```
Kubectl describe [COMPONENT] [NAME] [OPTIONS]
```

```
kubectl describe pod my-pod
```

EXEC:

Executes the command specified in a container with the possibility of adding arguments. Also pretty useful when debugging.

```
kubectl exec [OPTIONS] [podName/Container] [COMMAND] [args]
```

2.5.6 Manifests YAML

Syntax and contents of K8s YAML configurations files, also called manifests. Main tool for creating and configure components at K8S Clusters.

Possible to have more of one component description in one YAML, separating them by '---'.

2.5.6.1 Composition

Manifests have three main parts done by the user, and one added by k8s:

1. **apiVersion/Kind:** Declaring what we want to create.
2. **Metadata:** Fields like name, labels.
3. Specification(spec): Attributes of this field are specific to the kind of component we are creating.
4. Status: Compares the desired state and the actual.(Self healing feature).
   - K8S update that state continuously. And compares the actual vs desired.
   - The information comes from etcd.

2.5.6.2 Format

Manifests are written with YAML(Yet Another Markup Language). The main characteristics are:

- Human Friendly: Data serialization for all programming languages, wide syntax usage.
- Strict Indentation: Important to use some YAML validators to avoid errors.

- Localization dependent: better to store them with the app code. To be part of the hall app.

2.5.6.3 Labels & Connectors(selectors)

One of the critical part of the YAMLS spec part that allows interconnection between components are labels and selectors.

- Labels: Any key-value pair used to identify the component.
- Connectors: Looks for the label specified and connects to it the component defined at the YAMLS where the connector remains.

2.5.7 Kubernetes Deployments

So far, we have defined the components of Kubernetes, structure, how it works, among other things. Furthermore, now we will see how to deploy a fully functional Kubernetes cluster using different tools available.

The first way we will see will be Kubeadm, followed by minikube and microk8s.

2.5.7.1 KubeAdm

Kubeadm is a tool, directly developed by Kubernetes team, for providing fast paths to create Kubernetes clusters easily thorough a CLI.

The installation could be considered a basic Kubernetes, as it comes with no addons or preconfigured elements. Also, it uses the Kubernetes architecture of master-slave and initializes all the master processes at the master node.

Apart from that, kubeadm does not provide nodes or machines for the cluster. These must be previously available to deploy Kubernetes on them. To manage the cluster generated, we will still need Kubectl, as Kubeadm does not provide any tool to access the Kubernetes API Server.

Later, in the configuration chapter, we will see the commands that allow us to obtain the cluster with kubeadm and configure it.

The information related to kubeadm was taken from the official website[11].

2.5.7.2 Minikube

Minikube is a tool, that provides a single-node Kubernetes cluster, where Master and Node functions are done simultaneously. This Node will use the container runtime previously installed, for example Docker.

This architecture is oriented to testing environments, enabling to test a new k8s component before putting it at production environment.

2.5.7.3 Microk8s

MicroK8s is a simple tool to implement a simple, lightweight and focused production-grade upstream for creating Kubernetes clusters with more than

one node. Unlike Kubeadm, Microk8s does not initialize master nodes or master processes. Instead, it runs them as a snap service. That is a double edge sword, as they are conceived to simplify the installation and improve the user experience. However, in the end, at environments like the one we are using, LXC containers as nodes, this feature results in errors, and when it works it does, the cluster formation has a considerable delay.

One of the pros of having a tool like that is his over twenty add-ons, like HELM, MetalLoadBalancer, Prometheus. These complements are pretty handy to have in a cluster, helping to reduce the time expend installing and configuring those elements from out-of-box.

2.6. Kubernetes Complements

2.6.1 HELM

Is a package manager for K8S, like apt is for Ubuntu machines. It is a convenient way for packaging YAML files collections, as a group of Kubernetes resources that form an application, and distributing them in public and private communities[12].

Those packages or bundles of YAMLs files are called Helm Charts. We can create our own Charts. Push them etc.Most of the complex applications have a HELM CHART.

This opens up for template creation, which gives the possibility of creating a blueprint for common YAML files fields with the option to replace the values mostly repeated by placeholders.

The values.yaml acts as a .properties file containing the values to use at the YAML files.

2.6.2 LENS

LENS is a clear example of how open-source policy can contribute to our product. It is an open-source integrated development environment (IDE) for Kubernetes. Is a standalone application available on Windows, Linux and macOS[13].
It is considered one of the most potent IDEs thanks to the great utility provided by its functionalities. Among which there are:

- Intuitive Interface that allows us to assure our cluster configuration quickly. Having access to all the important resources and components of the cluster.
- Real time debugging: Possibility to access real time status and logs steams
- Implemented support for Prometheus, the complement will see at the next point. This implementation allows LENS to have real-time statistics about the cluster. For example, CPU and Memory usage at Nodes.

LENS also has an integration with HELM where we can manage and install more easily HELM charts.

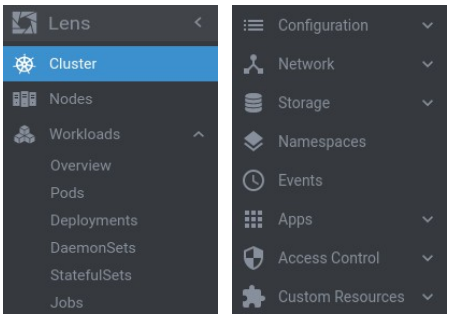


Figure 9: LENS Menu Toolbar

2.6.3 Prometheus

In the containerized environment, many components take part in the cluster's proper functioning, and it is challenging to keep track of everything that happens in it, such as tracking errors, response latency, and usage of resources.
That's why if an error occurs, being able to quickly identify the origin of it is critical in order to re-establish as soon as possible the service. Prometheus uses the cluster data available to, constantly monitor all the service, alert when errors happen and identify problems. Furthermore, that is what Prometheus is, a monitoring tool for these environments.
Prometheus has two principal components, the Server and the Exporter. The Server has three main components that do the actual monitoring work.

- Data Retrieval Worker: In charge of pulling the metric data that Exporter extracts from the applications and services.
- Time Series Database: Stores the metrics at a database.
- HTTP Server: Accepts queries for that stored data

The Exporter is installed at the service or application we want to monitor and export the Prometheus server metrics.

3. Project Development:

This section includes all relevant methods used during the development of this project. It also explains the research methods and the software used, which is the central resource used as the technology is oriented to the management of clusters containing containerized applications hosted at servers.

The development of this project has been carried out in two phases: The first consisted of essential learning about container technologies to have a precise vision of their possibilities and their main functionalities. With that in mind, we began to define the scope, taking into account the time available and the type of thesis we wanted to do.

The result was that we opted for a project that analyzed a wide range of characteristics of these technologies, instead of getting deep into a specific point, which is also entirely possible due to the great depth of the themes that compose them.

In the second phase, with the scope defined, we start with development. The development consisted of the analysis and practical application of the Docker and Kubernetes container technologies and some of its complements. Generating theoretical documentation of the key points of the technologies as well as the creation and testing of a practical scenario.

3.1. Communication

Due to the causes of the pandemic and as a preventive measure, remote communication became the primary way of communicating during the thesis.

We started in mid-July, almost two months before the start of the new academic year. A fact that allowed us to have more time to carry out telematic meetings with our supervisor. Moreover, this was reflected in the considerable progress of the project.

One tool that greatly influenced the progress was the VNC Tiger remote desktop viewing tool, connected to one machine our supervisor enabled for us at one host of the university network.

With this machine, we could share a lot more than words. Inside her, we had available documentation related to Docker and Kubernetes, and also we used it as a repository to put our work for revision.

3.2. Software

Previous applications and software implemented by companies and users have been used to carry out this project. These users and companies have shared their product selflessly as part of the Open-Source culture.

Having free access to it, the community has provided add-ons, tools and expansions to improve the experience of using these products.

The main software used is Docker and Kubernetes. Both are widely explained in the chapter '2.State of the Art'. But summarizing, they are a container management tool for containerized applications, with the difference that Kubernetes is more cluster oriented and for bigger environments.

We also used Microk8s and Minikube, which are simple ways of creating Kubernetes clusters with useful add-ons. They are free to use and came from the community due to the Open-Source policy adopted by Google, who is the owner of Kubernetes.

Last but not least, we also used LXC containers and its manager LXD. Allowing us to create and add nodes to our clusters.

3.3. Documents Deliveries

The thesis report's preparation has been carried out linearly as the analyses and experiments of each point of interest were completed.

After the investigation phase and the scenario testing were completed, the report's corresponding part was written with LibreOffice. LibreOffice is also free software, consequently intensifying the project idea of promoting the use of open-source software. A first version was sent to the advisor for validation.

In parallel, apart from the main document, we have done slides about Kubernetes for academic purpose in LaTeX language and shared at the working machine set up by the advisor.

4. Setups

This section will define and deploy five scenarios using the different technologies studied at this project. Once the deployment is applied, the experiment results will be analyzed to extract the advantages of having used the appropriate technology for each scenario.

Each scenario consists of three points: Objective, Implementation and Results.

- **Prerequisites:** Scenario requisites in order to be able to implement the setup.
- **Objective:** Scenario definitions with requirement. Plus the final setup defined to meet the requirements.
- **Implementation:** Tools, methodologies and technologies used for reaching the setup.

Each scenario and its setup are conceived to achieve our purpose of choosing the best way to develop application and services using containers technologies.

From now on this section, the foundation is that we are a software development small company and we have heard about containers technologies. So as the demand uprises, we began to contemplate the possibility of developing using container technologies.

Therefore after researching and training, we decided to apply it in the following projects. Our product consists of a MongoDB database, Mongo-Express a web user interface for the database, and a Node.js web application.

4.1. Docker & Docker-Compose

In this scenario, we will carry out the transformation of the software towards a deployment with containers.

First of all, we will design the image from which our web app's container will be created. We will perform deployment and configuration with Docker, which will later be simplified with Docker-Compose.

4.1.1 Prerequisites

To reproduce this scenario, it is necessary to have a machine with Docker installed in it, following the instruction available at Docker official site.[1].

There is no need for installing Docker-Compose as Docker already installs it by default.

4.1.2 Objective

Simplify the deployment of the app pre-designed taking advantage of Docker and Docker-Compose.

In the end, we will have our three containers, MongoDb, Mongo-Express and Node.js running at Docker runtime. First with Docker and later with Docker-Compose.

4.1.3 Implementation

First we need to get the MongoDB and Mongo-Express containers running:

1. Create Docker Network so the containers can connect to each other.

```
$ docker network create myapp-network
```

2. Once the network is created we can pull the image and start the containers at the same time with the '*run*' command.

```
$ docker run -p 27017:27017 --name mongoDB \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=1234 \
-d --net myapp-network  mongo
```

```
$ docker run -p 8080:8081 --name mongo-e \
-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_ADMINPASSWORD=1234 \
-e ME_CONFIG_MONGODB_SERVER=mongoDB -d \
--net myapp-network  mongo-express
```

- -p hostPort:containerPort: For binding the container port to the host.
- --net <>: Connect container to the Docker Network previously created so containers can communicate using containers names.
- -e : Environmental variables need by the app in order to start running.

With that we can see the containers running with *'docker ps'* and access mongo-express at *'localhost:8080'*.

```
$ docker ps
CONTAINER ID   IMAGE          CREATED              STATUS              PORTS                     NAMES
e6806f988830   mongo-express  About a minute ago   Up About a minute   0.0.0.0:8080->8081/tcp    mongo-e
bb993176bc2c   mongo          6 minutes ago        Up 6 minutes        0.0.0.0:27017->27017/tcp  mongoDB
```

Now is turn for our web application. It is a simple Node.js app with a main page that allows the user to edit the name, email and interests and this credentials get stored at the mongoDB.

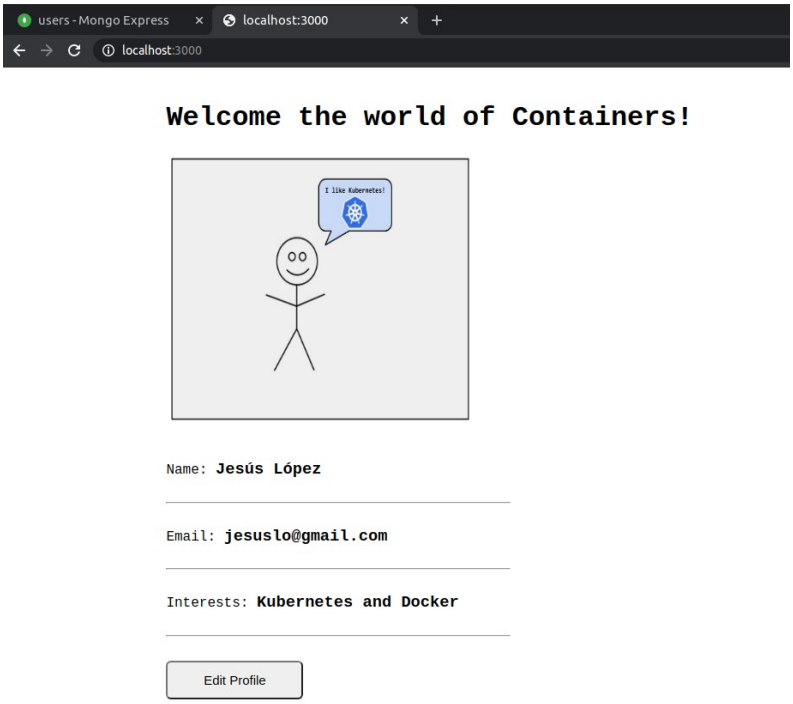


Figure 10: Web App

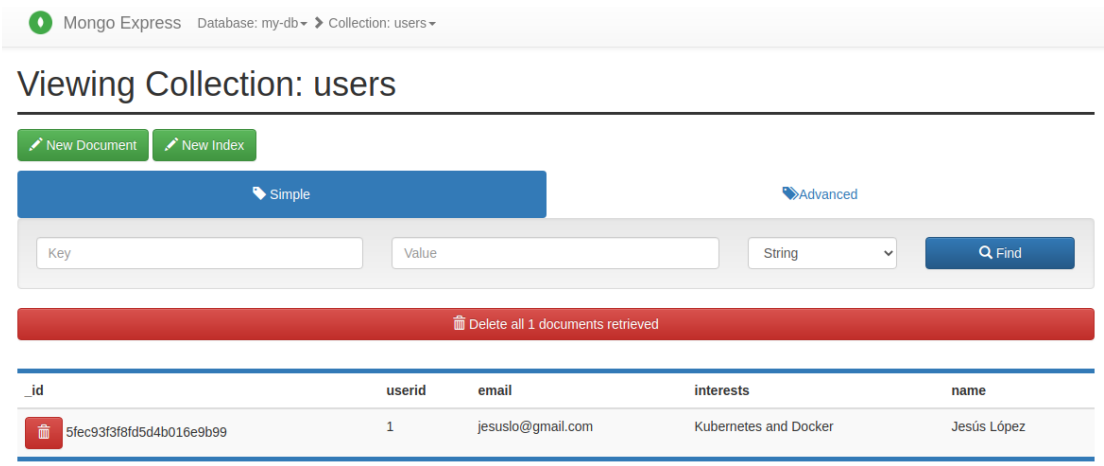


Figure 11: User Data Stored

Before running our app as a container, we need to write the Dockerfile, build the image and push it to our DockerHub.

```
FROM node:13-alpine

ENV MONGO_DB_USERNAME=admin \
    MONGO_DB_PWD=1234

RUN mkdir -p /home/app

COPY ./app /home/app

WORKDIR /home/app

RUN npm install

CMD ["node", "server.js"]
```

Figure 12: Our Web App Dockerfile

Once we got the image we need to build the image, tag it and push it:

```
$ docker build -t web-app:0.0.1 <pathToDockerfile>
$ docker tag 1d53d4bcd906 magicby/web-app:0.0.1
```

```
$ docker image ls

REPOSITORY          TAG             IMAGE ID       CREATED          SIZE

web-app             0.0.1           1d53d4bcd906   7 seconds ago    125MB

magicby/web-app     0.0.1           1d53d4bcd906   3 minutes ago    125MB
```

We can see the two images, the one we build locally and the one we just tagged ready to be pushed.

```
$ docker push magicby/web-app:0.0.1
```

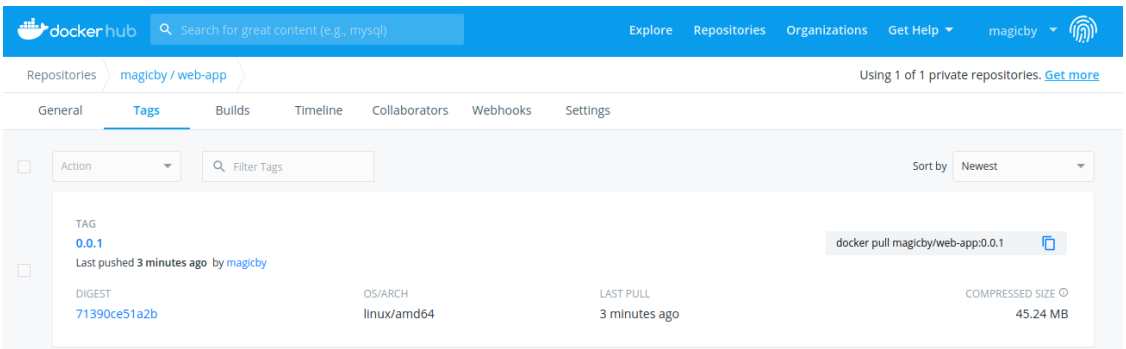


Figure 13: DockerHub repository

Now that the image is at DockerHub, we can execute the command *run* with it, as by default Docker looks for images at DockerHub.

```
$ docker run -p 3000:3000 --name app-web \
--net myapp-network \
-d magicby/web-app:0.0.1
```

- It is key to remark that as the app container is in the same DockerNetwork as the others, we can connect to the database using the container name instead of a URL host like

  MongoDB URL without being in the same network:
  mongodb://admin:1234@localhost:27017

  MongoDB URL being in the same network, where *mongoDB* is the container containing mongoDB:
  mongodb://admin:1234@mongoDB

Doing *'docker ps'* we can see the three containers running and the relative apps working:

```
$ docker ps
CONTAINER ID   IMAGE                  STATUS          PORTS                    NAMES
a69f48ca763d   magicby/web-app:0.0.1  Up 6 minutes    0.0.0.0:3000->3000/tcp   app-web
e6806f988830   mongo-express          Up 16 minutes   0.0.0.0:8080->8081/tcp   mongo-e
bb993176bc2c   mongo                  Up 16 minutes   0.0.0.0:27017->27017/tcp mongoDB
```

So far we have seen that deploying with containers cut off the time lost downloading, configuring and running the software package at new environments. The commands to do it are tedious, have many arguments, and can only manage the containers one by one.

To solve this, we will use Docker-Compose. As state before, it allows the user to start multiple containers at once using its own version of Dockerfile called docker-compose file. Let us see it:

```
version: "3"
services:
  my-app:
    image: magicby/web-app-compose:latest
    ports:
      - 3000:3000
  mongodb:
    image: mongo
    ports:
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=1234
    volumes:
      - mongo-data:/data/db
  mongo-express:
    image: mongo-express
    ports:
      - 8080:8081
    depends_on:
      - mongodb
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=1234
      - ME_CONFIG_MONGODB_SERVER=mongodb
volumes:
  mongo-data:
    driver: local
```

Figure 14: Docker-Compose File

Looking into the file, we can see how what with Docker are command arguments, as '-p' for ports or '-e' for environment variables, with compose are fields at the file. Thus simplifying the deployment and managing task.

Also, there is no need to specify which network they connect, as by default, Compose creates one for all the services created with the file.

Now that we have the file, using the docker-compose CLI, we can use it:

```
$ docker-compose up --build -d
```

**needs to be executed within the folder containing the docker-compose file*

Once again if we repeat *'docker ps'* we should see the three containers running with the same of the service established at docker-compose file with the name of the folder as a prefix.

To stop the services started by the compose file:

```
$ docker-compose stop
```

4.2. Kubernetes Resources

As the next sections all going to use Kubernetes, this sections is not about a setup. Instead, it is the definition of the Kubernetes resources and components to be used in all upcoming scenarios.

These components will be present in all since they are those that allow us to configure our application.

However, each scenario may need some other resource that will be defined in the corresponding section.

4.2.1 ConfigMap & Secret

As stated in the theoretical section, ConfigMap and Secret are useful k8s components to store useful information and variables used in other parts of the cluster.

In our case, we are going to store the database URL in the ConfigMap and the database credentials in the Secret, as secret codifies its content.

Figure 15: ConfigMap & Secret YAML

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongodb-configmap
data:
  database_server: mongodb-service
```

```
apiVersion: v1
kind: Secret
metadata:
    name: mongodb-secret
type: Opaque
data:
    mongo-root-username: dXNlcm5hbWU=
    mongo-root-password: MTIzNA==
    mongo-url: bW9u . . .Z29E3
```

At the ConfigMap, we can see the variable where other components can find the database server. As in Docker, we can use the terminology of the technology we are using. In this case, the value is the Kubernetes Service name, which enables communication with the database Pod.

Meanwhile, at the Secret, we can found the credentials of the database Base 64 encoded.

4.2.2 Volumes

From the study realized, we know that for providing to an application persisting data, we need to supply it with a Volume and a PersistentVolume and PersistentVolumeClaim.

Figure 16: PersistentVolume & PersistentVolumeClaim YAML

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: app-pv
  labels:
    type: local
spec:
  storageClassName: generic
  capacity:
    storage: 500Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/var/lib/mongo"
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-pvc
spec:
  storageClassName: generic
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

4.2.3 Deployment and Services

Our application environment has three main applications to deploy: MongoDB, mongo-express, and our web-app. We will use the Deployment resource to deploy them at the cluster. The only app with more than one replica will be web-app.

Also, we take advantage of the fact that Kubernetes allows defining more that one resource in the YAML files, and we are going to define the service for each deployment at the same file.

We are only showing the MongoDB Deployment and Service YAML file because of the document's length and similarity with the other two.

Figure 17: MongoDB YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb-deployment
  labels:
    app: mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
```

```
        app: mongodb
    spec:
      containers:
        - name: mongodb
          volumeMounts:
            - mountPath: /var/lib/mongo
              name: mongodb-data
          image: mongo
          ports:
            - containerPort: 27017
          env:
            - name: MONGO_INITDB_ROOT_USERNAME
              valueFrom:
                secretKeyRef:
                  name: mongodb-secret
                  key: mongo-root-username
            - name: MONGO_INITDB_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mongodb-secret
                  key: mongo-root-password
      volumes:
        - name: mongodb-data
          persistentVolumeClaim:
            claimName: app-pvc
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: mongodb
  name: mongodb-service
spec:
  selector:
    app: mongodb
  ports:
    - port: 27017
      targetPort: 27017
```

From the file, we can see the use of the PersistentVolumeClaim and the Secret. Also, at the Service definition, we can see the name we assign to it,*'mongodb-service'*,is stored previously at the ConfigMap. Furthermore, it is the service with which the mongo-express and the web-app component will seek to connect.

4.2.4 Ingress Rules

Because we do not want to expose our Services, we need to deploy some Ingress Rules and bind them to the right Services. Later in each scenario, an Ingress Controller will behave like a proxy server forwarding the requests for services.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
    - host: myapp.ex.com
      http:
        paths:
          - backend:
              serviceName: myapp-service
              servicePort: 8080
    - host: mongo-express.com
      http:
        paths:
          - backend:
              serviceName: mongo-express-service
              servicePort: 8081
```

If we observe, each host's entry is bound to the Service name and port defined at the YAML file of mongo-express and web-app.

4.3. Minikube

The experience with Docker was terrific, and now our deployment environment is getting bigger, and the clients interested in our product also have bigger networks. So we want to do the next step, Kubernetes. However, we have seen that it is not as easy as Docker to get used to it, and we want to do some testing before saying we are ready to work with Kubernetes.

And that is where Minikube enters the game. With his one single-node cluster, it sets up the perfect environment for testing and developing k8s solutions before passing into the deployment and production status.

4.3.1 Prerequisites

As Minikube only needs one node or machine with Docker installed, the requisites are the same as the previous scenario. So literally we can reuse the machine from the previous scenario.[14]

4.3.2 Objective

Transform the Docker deployment solution previously encountered to one based on Kubernetes.

At the end having a single-node Kubernetes cluster with all the resources and application working.

4.3.3 Implementation

First of all, we need to install Kubectl for managing the cluster from the terminal[15].

```
$ apt install kubectl
```

After that we install Minikube and start the cluster as shown in the Minikube documentation [7].

```
$ minikube start
```

Minikube installs Kubernetes and the necessary complements for starting the cluster, and configures Kubectl for interact with the cluster created.

Once the cluster is running we can start to deploy the resources from the YAML files with kubectl.

The command for do it is:

```
$ kubectl apply -f <YAML file>
```

After doing the command for all the files, we can check the status of the components with:

```
$ kubectl get [COMPONENT] -n [NameSpace]
```

```
* by default the NameSpace is the 'default'
```

Figure 18: App Web Kubernetes Components Deployed

```
$ kubectl get all

NAME                                    READY   STATUS    RESTARTS   AGE

pod/mongo-express-78fcf796b8-cc4sd      1/1     Running   0          82s

pod/mongodb-deployment-5ffd964db4-7kff7 1/1     Running   0          86s

pod/myweb-84d5b4bbb9-shmlc              1/1     Running   0          76s

pod/myweb-84d5b4bbb9-t7wpd              1/1     Running   0          76s


NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE

service/kubernetes            ClusterIP   10.96.0.1       <none>        443/TCP     39m
```

```
service/mongo-express-service   ClusterIP   10.101.62.114   <none>        8081/TCP    82s
service/mongodb-service         ClusterIP   10.99.156.99    <none>        27017/TCP   86s
service/myapp-service           ClusterIP   10.97.102.172   <none>        8080/TCP    76s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mongo-express       1/1     1            1           82s
deployment.apps/mongodb-deployment  1/1     1            1           86s
deployment.apps/myweb               2/2     2            2           76s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/mongo-express-78fcf796b8       1         1         1       82s
replicaset.apps/mongodb-deployment-5ffd964db4  1         1         1       86s
replicaset.apps/myweb-84d5b4bbb9               2         2         2       76s
```

An alternative to the get command is LENS. As explained before it is a user interface complement for Kubernetes. After installing LENS, it auto-detects the minikube cluster and shows us the information related.

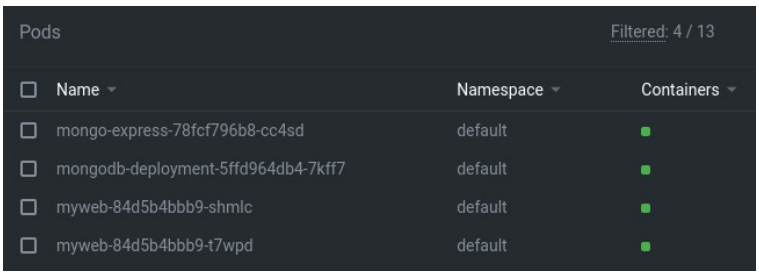


Figure 19: Minikube Cluster Info in LENS

Before accessing our app using an Ingress Controller, we need to configure a LoadBalancer to give the cluster an external IP address. Here is where the addons of Minikube became useful, especially the MetalLB one, which will give any external service of the cluster an IP after some configuration.

```
$ minikube addons enable metallb
$ minikube addons configure metallb
-- Enter Load Balancer Start IP: 192.168.49.10
-- Enter Load Balancer End IP: 192.168.49.20
✅  metallb was successfully configured
```

```
* we configured this IP because the cluster is in 192.168.49.2
```

After that we can use HELM to install the Nginx Ingress Controller to our cluster following Nginx Documentation[16].

```
$ helm repo add nginx-stable https://helm.nginx.com/stable
$ helm repo update
$ helm install my-release nginx-stable/nginx-ingress
```

Finally we can deploy the Ingress rules after adding one key annotation at the matadata field. Allowing the Nginx Ingress Controller to acknowledge him and configure himself.

```
metada:
   annotations:
       kubernetes.io/ingress.class: "nginx"
```

And if we check the Service components of the cluster:

Figure 20: App Web Kubernetes Services

```
$ kubectl get svc
NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
kubernetes                 ClusterIP      10.96.0.1       <none>          443/TCP
mongo-express-service      ClusterIP      10.101.62.114   <none>          8081/TCP
mongodb-service            ClusterIP      10.99.156.99    <none>          27017/TCP
my-release-nginx-ingress   LoadBalancer   10.98.46.192    192.168.49.10   80:31780/TCP,443:30119/TCP
myapp-service              ClusterIP      10.97.102.172   <none>          8080/TCP
```

We can see how the Nginx Ingress Controller(NxIC) is configured as a LoadBalancer and has an external IP. Which we can redirect the requests of the host stated at the Ingress rules if we configure our host hosts lists, to that IP. The NxIC will acknowledge our requests and act as a proxy server.

To be sure that the NxIC is forwarding to the right services we can get more information with:

```
$ kubectl describe ingress myapp-ingress
Name:             myapp-ingress
Namespace:        default
Address:          192.168.49.10
```

```
Default backend:  default-http-backend:80

Rules:

  Host               Path  Backends

  ----               ----  --------

  myapp.ex.com

                     /   myapp-service:8080 (172.17.0.2:3000,172.17.0.7:3000)

  mongodb.com

                        mongodb-service:27017 (172.17.0.8:27017)

  mongo-express.com

                     /   mongo-express-service:8081 (172.17.0.9:8081)

Annotations:         kubernetes.io/ingress.class: nginx
```

With all this configuration in place, we can access our web-app the same way as we did with Docker at the first scenario, but this time using a single-node Kubernetes cluster deployed by minikube.

4.4. Microk8s

After creating the necessary resources and testing them, it is time to reaffirm our compromise with k8s and use a multi-node cluster to extract and use all the functionalities available.

To help us simulate a multi-node environment, we will use Linux Containers (LXC) to represent various servers.

It also worth to remark that as well as we are seeing more ways of deploying Kubernetes, the characteristics and complexity of these deployments change. Converting these characteristics into another variable to take into account when choosing a type of deployment.

4.4.1 Prerequisites

Microk8s have similar requirements as the last scenarios. We need an Ubuntu 20.04 machine with at least 20 gigabytes of disk space and 4 gigabytes of memory plus an internet connection, information from Microk8s website[17].

With that we are going to download the LXD software and Microk8s.

4.4.2 Objective

Test the functionality of a multi-node cluster using Microk8s and the solution previously acquired. Reinforcing the idea of wide compatibility, high availability and scalability of Kubernetes.

Additionally, discuss the benefits of out-of-the-box Kubernetes deployments.

4.4.3 Implementation

First of all we need the LXC containers up and running:

```
$ snap install lxd
$ lxd init
$ lxc launch ubuntu:20.04 mk8s-node1
$ lxc launch ubuntu:20.04 mk8s-node2
$ lxc launch ubuntu:20.04 mk8s-node3
```

After the LXC creation we need a profile for the nodes in order to run microk8s:

```
$ lxc profile create microk8s
$ wget https://raw.githubusercontent.com/ubuntu/microk8s/
  master/tests/lxc/microk8s.profile -O microk8s.profile
$ cat microk8s.profile | lxc profile edit microk8s
$ lxc profile assign mk8s-node1 default,microk8s
```

Now we need to get inside the containers, start microk8s and create the cluster:

```
$ lxc shell mk8s-node1
mk8s-node1$ sudo snap install microk8s --classic
mk8s-node1$ microk8s.start
mk8s-node1$ microk8s add-node
From the node you wish to join to this cluster, run the following:
microk8s join 10.213.88.185:25000/4749bcbfee41c325afb9a68caf64c7a9
```

Executing the command 'microk8s join ...' inside the other two nodes, will add them to the cluster.

Now we can see the nodes of the cluster, using the kubectl instance that comes with microk8s.

```
$ microk8s.kubectl get nodes
NAME        STATUS   ROLES    AGE   VERSION
```

```
mk8s-node1    Ready    <none>   28m   v1.19.5
mk8s-node2    Ready    <none>   28m   v1.19.5
mk8s-node3    Ready    <none>   28m   v1.19.5
```

Now that we have our cluster ready we can start deploying our Kubernetes resources as we did with Minikube.

```
$ microk8s.kubectl apply -f <YAML file>
```

And if we use *'microk8s.kubecctl get all'* we can see again all the components started, as in figure [18].

But now we have more than one node and if we remember, the deployment of our web-app has more than one replica, meaning that we have created more than one application pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myweb
  labels:
    app: myweb
spec:
  replicas: 3
```

Figure 21: Number of Web-App Replicas

Let's see where they have been deployed:

```
$ microk8s.kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     NODE
mongo-express-797845bd97-65c4v    1/1     Running   0          5m      mk8s-node2
mongodb-75fb6cb767-h6njk          1/1     Running   1          5m      mk8s-node1
myweb-app-749c4b5d9b-brv2g        1/1     Running   0          5m      mk8s-node2
myweb-app-749c4b5d9b-z6c4r        1/1     Running   1          5m      mk8s-node3
myweb-app-749c4b5d9b-z2xvg        1/1     Running   1          5m      mk8s-node3
```

From the output of the command, we can see how the Pod got distributed around the available nodes by Kubernetes. It decided taking into account the available resources of each node, in our case the three nodes are the same, that is why the deployment has ended with a balanced workload across the cluster.

As before, we need to deploy our Ingress Rules and the Nginx Ingress Controller(NxIC) to complete our product and be able to access it from the internet. The rules are deployed with the command 'kubectl apply -f <file>' as the other components.

For the installation of NxIC we have two options, both involving add-ons of Microk8s.

Option 1

- Enabling HELM3 add-on,which installs HELM version 3, and use it to install NxIC from the creators site[16].

```
$ microk8s enable helm3
$ microk8s helm3 repo add nginx-stable https://helm.nginx.com/stable
$ microk8s helm3 repo update
$ microk8s helm3 install my-ingress-controller nginx-stable/nginx-ingress
```

Option 2

- Enabling Ingress add-on which enables the NxIC:

```
$ microk8s enable ingress
```

No matter which option we choose, in the end, the result will be the same.

At the end of all this, we will end up with the same Kubernetes components and functionality.

4.5. Kubeadm

Having the Kubernetes resources cemented and tested, we want to focus this configuration on installing and configuring Kubernetes without using predefined tools or setups, as Minikube or Microk8s.

And again, to help us simulate a multi-node environment, we will use Linux Containers (LXC) to represent various servers.

4.5.1 Prerequisites

The requisites are the ones from Kubeadm and Kubernetes, as they are the software we are gonna install. They can be fount at the Kubernetes documentation [11].

The more important is the need of one or more machines running one of these Linux like distribution:

- Ubuntu 16.04+
- Debian 9+
- Red Hat Enterprise Linux (RHEL) 7
- Fedora 25+

- CentOS 7
- HypriotOS v1.0.1+
- Flatcar Container Linux

Important, we need to have it installed Docker or any CRI at the node previously.

4.5.2 Objective

Configure and deploy a Kubernetes cluster with the Master-Slave architecture using the deployment tool Kubeadm. As well as using Flannel as our Container Network Interface for cluster networking.

Once the cluster is up and running, deploy our pre-designed web-app with Prometheus monitoring to add some graphical information about it complementing the LENS IDE.

4.5.3 Implementation

First of all we need the LXC containers up and running:

```
$ snap install lxd
$ lxd init
$ lxc launch ubuntu:20.04 k8s-master
$ lxc launch ubuntu:20.04 k8s-slave1
$ lxc launch ubuntu:20.04 k8s-slave2
```

After the LXC creation we need a profile for the nodes, for Kubernetes we need to add the following lines to a configuration profile:

```
config:
  linux.kernel_modules: ip_tables,ip6_tables,netlink_diag,nf_nat,overlay
  raw.lxc: "lxc.apparmor.profile=unconfined\nlxc.cap.drop= \nlxc.cgroup.devices.allow=a\
  nlxc.mount.auto=proc:rw sys:rw"
  security.privileged: "true"
  security.nesting: "true"
```

```
$ lxc profile create k8s
$ cat k8s.profile | lxc profile edit k8s
$ lxc profile assign k8s-master default,k8s
$ lxc profile assign k8s-slave1 default,k8s
$ lxc profile assign k8s-slave2 default,k8s
```

Now we need to get inside the containers to start the configuration of the cluster. Inside each LXC container, execute the following commands. Some commands and repositories are extracted from Kubeadm documentation[18].

The next commands need to be replied for every node:

```
$ lxc shell k8s-master
```

Add Kuberntes apt repository

```
k8s-master$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

k8s-master$ apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

Install Kubeadm, Kubectl and Kubelet

```
k8s-master$ apt install -y kubeadm=1.20.0-00 kubelet=1.20.0-00 kubectl=1.20.0-00
```

At the node we want to be the master:

The next command is used to list and pull the images that kubeadm requires.

```
k8s-master$ kubeadm config images pull
```

Now is time to initialized the cluster.

```
k8s-master$ kubeadm init --pod-network-cidr=10.244.0.0/16

--ignore-preflight-errors=all
```

- --pod-network-cidr flag, indicates a range of IP addresses for the pod network. If set, the control plane will automatically allocate Classless Inter-Domain Routing(CIDR) for every node.
- --ignore-preflight-errors flag,indicates what to due with the list of checks that this commands runs and whose errors will be shown as warnings. Value 'all' ignores errors from all checks.

Now is time to configure a Container Network Interface(CNI) for the cluster.

We are gonna see how to install Flannel and Calico. Either one is valid.

- There is how to install Flannel:

```
k8s-master$ kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

As we have seen, as Flannel is more simple than Calico, requires lest resources to install it at the cluster.

Now if we check the pods running at 'kube-system' name-space, we will see a 'flannelid' pod at each node:

```
k8s-master$ kubectl get pods -n kube-system
NAME                    READY   STATUS    AGE   NODE
. . .
kube-flannel-ds-s9r92   1/1     Running   13m   worker1
kube-flannel-ds-tnn7t   1/1     Running   13m   worker2
kube-flannel-ds-xdqxd   1/1     Running   16m   master
. . .
```

Checking the LXC containers from the machine hosting them, we can see how the CNI-plugin created one interface and Flannel created another. Flannel will use his to do the container networking:

```
k8s-master$ lxc ls
+-------------+---------+-----------------------+
|   NAME      |  STATE  |          IPV4         |
+-------------+---------+-----------------------+
| k8s-master  | RUNNING | 10.244.0.1 (cni0)     |
|             |         | 10.244.0.0 (flannel.1)|
|             |         | 172.17.0.1 (docker0)  |
|             |         | 10.213.88.170 (eth0)  |
+----------+---------+-----------------------+
| k8s-worker1 | RUNNING | 10.244.1.1 (cni0)     |
|             |         | 10.244.1.0 (flannel.1)|
|             |         | 172.17.0.1 (docker0)  |
|             |         | 10.213.88.121 (eth0)  |
+-------------+---------+-----------------------+
| k8s-worker2 | RUNNING | 10.244.2.1 (cni0)     |
|             |         | 10.244.2.0(flannel.1) |
```

```
|             |         | 172.17.0.1 (docker0)    |
|             |         | 10.213.88.70 (eth0)     |
+-------------+---------+-------------------------+
```

- And this is how to install Calico:

```
k8s-master$ kubectl create -f
https://docs.projectcalico.org/manifests/tigera-operator.yaml
k8s-master$ wget https://docs.projectcalico.org/manifests/custom-
resources.yaml -O calico.yaml
```

** Important: We need to change the field 'CIDR' to match the IP pool we set at the command 'kubeadm init'. In our case 10.244.0.0/16*

```
k8s-master$ kubectl apply -f calico.yaml
```

Now if we check the pods running at 'calico-sysem' namespace:

```
k8s-master$ kubectl get pods -n calico-system
NAME                                       READY    STATUS    NODE
calico-kube-controllers-546d44f5b7-htv5m   1/1      Running   k8s-master
calico-node-lv9fp                          1/1      Running   k8s-worker1
calico-node-qvqhc                          1/1      Running   k8s-worker2
calico-node-sh7j5                          1/1      Running   k8s-master
calico-typha-5c6cbd6dc9-h6212              1/1      Running   k8s-worker1
calico-typha-5c6cbd6dc9-tjmx2              1/1      Running   k8s-master
calico-typha-5c6cbd6dc9-whr59              1/1      Running   k8s-worker2
```

Also, if we check the LXC containers from the machine hosting them, we can see how the interface used by Calico to do the container networking is created and available:

```
$ lxc ls
+-------------+---------+-------------------------------+
|   NAME      |  STATE  |             IPV4              |
+-------------+---------+-------------------------------+
```

```
| k8s-master  | RUNNING | 192.168.219.64 (vxlan.calico)  |
|             |         | 172.17.0.1 (docker0)           |
|             |         | 10.213.88.121 (eth0)           |
+-------------+---------+--------------------------------+
| k8s-worker1 | RUNNING | 10.244.2.15 (vxlan.calico)     |
|             |         | 172.17.0.1 (docker0)           |
|             |         | 10.213.88.248 (eth0)           |
+-------------+---------+--------------------------------+
| k8s-worker2 | RUNNING | 10.244.1.34 (vxlan.calico)  |
|             |         | 172.17.0.1 (docker0)           |
|             |         | 10.213.88.180 (eth0)           |
+-------------+---------+--------------------------------+
```

Some observations to do are, the process 'kube-proxy', defined at Kubernetes architecture, hits the status running once a CNI is applied. He is in charge of the communication between components and without a CNI cannot do that. Another one is that when using Calico, as he is a more complex tool, takes more for a node to join the cluster as he has to create and configure more elements.

Continuing with the implementation of the setup, it's time to create the join command to execute at the terminal of the nodes we want to add to the cluster.

```
k8s-master$ kubeadm token create --print-join-command

kubeadm join 10.213.88.228:6443 --token vguzas.gvds6jy4pxo3x0g2 -
-discovery-token-ca-cert-hash
sha256:9a52a74610b17b51695c6ff37fc083b0f52b28eaba8a36fcd21bb3784741f135
```

- --print-join-command flag is the reason why the 'token' command instead of printing only the token, printed the full 'kubeadm join' using the token.

At the slaves nodes:

To join the actual running cluster we need to execute the command created by the 'kubeadm token create' adding the flag '--ignore-preflight-errors=all'.

```
k8s-slave1$ kubeadm join 10.213.88.228:6443 --token vguzas.gvds6jy4pxo3x0g2
--discovery-token-ca-cert-hash
```

```
sha256:9a52a74610b17b51695c6ff37fc083b0f52b28eaba8a36fcd21bb3784741f135
--ignore-preflight-errors=all
```

Once we got the confirmation of the success of the joining process, we can check the status of the nodes.

```
k8s-master$ kubectl get nodes

NAME         STATUS   ROLES         AGE    VERSION

k8s-master   Ready    master        5m     v1.20.0

k8s-slave1   Ready    <none>        4m     v1.20.0

k8s-slave2   Ready    <none>        1m     v1.20.0
```

If we remember, master nodes have characteristic processes. We are going to verify that the master node does indeed have them at the namespace *'kube-system'*.

```
k8s-master$ kubectl get pods -n kube-system --field-selector
spec.nodeName=k8s-master

NAME                                READY   STATUS    RESTARTS   AGE

etcd-k8s-master                     1/1     Running   0          20m

kube-apiserver-k8s-master           1/1     Running   0          20m

kube-controller-manager-k8s-master  1/1     Running   0          20m

kube-scheduler-k8s-master           1/1     Running   0          20m

. . . . .
```

And effectively, the Master contains all the processes it has to. Furthermore, if we start deploying the Kubernetes resources as we did in previous scenarios, we will see how any application pods will be deployed at the master node.

```
k8s-master$ kubectl get all -o wide

NAME                                    READY   STATUS         IP           NODE

pod/mongo-express-78fcf796b8-shbdf        1/1     Running        10.244.2.2   k8s-slave2

pod/mongodb-deployment-5ffd964db4-gztgw   1/1     Running        10.244.1.2   k8s-slave1

pod/my-release-nginx-ingress-7866ff67-v64kb  1/1     Running        10.244.2.7   k8s-slave2

pod/myweb-84d5b4bbb9-4gqk5                1/1     Running        10.244.1.4   k8s-slave1
```

```
pod/myweb-84d5b4bbb9-d5g2j          1/1     Running   10.244.2.3   k8s-slave2
pod/myweb-84d5b4bbb9-jpmmx          1/1     Running   10.244.1.3   k8s-slave1
```

After the main pods are deployed, we need to install the NxIC again and deploy the ingress rules.

After that, we will add to the cluster the Prometheus monitoring tool we mentioned before at the 'Kubernetes complements' section in this document.

To do so, we will install it via HELM, and access it to see the metrics it offers by default. As well as how interacts with LENS, another complement we have been using during the realization of this entire project.

Let's install Prometheus:

```
$helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts

$helm repo add stable https://charts.helm.sh/stable

$helm repo update

$helm install prometheus prometheus-community/kube-prometheus-stack
```

Now if we check the services Prometheus created, we want to access the one who is handling the connections to the Prometheus component Grafana, which is the application which makes the graphics from the metrics Prometheus get.

```
$ kubectl get svc
NAME                                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
. . .
prometheus-grafana                      ClusterIP   10.111.232.91    <none>        80/TCP
prometheus-kube-prometheus-operator     ClusterIP   10.104.190.54    <none>        443/TCP
prometheus-kube-prometheus-prometheus   ClusterIP   10.108.99.92     <none>        9090/TCP
prometheus-prometheus-node-exporter     ClusterIP   10.107.187.221   <none>        9100/TCP
. . .
```

To access the Grafana service we can create an ingress rule for it, or simply add a 'host' to the Ingress file we already have:

```
- host: mygrafana.com

     http:

       paths:
```

```
      - path: /
        backend:
          serviceName: prometheus-grafana
          servicePort: 80
```

With the new Ingress Rule deployed, and the pertinent change to our hosts file, pointing 'mygrana.com' to the IP of the NxIC, we can access it:

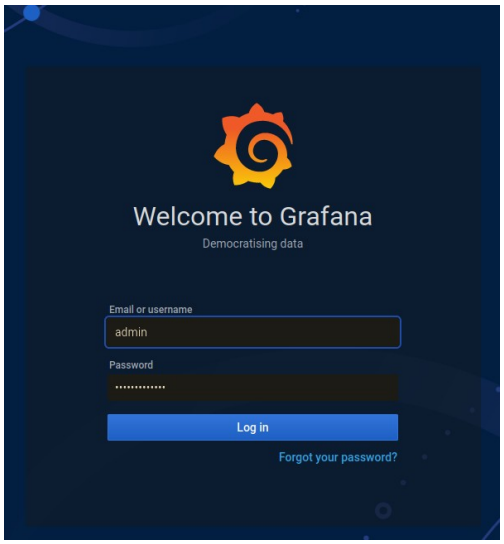


Figure 22: Grafana Login

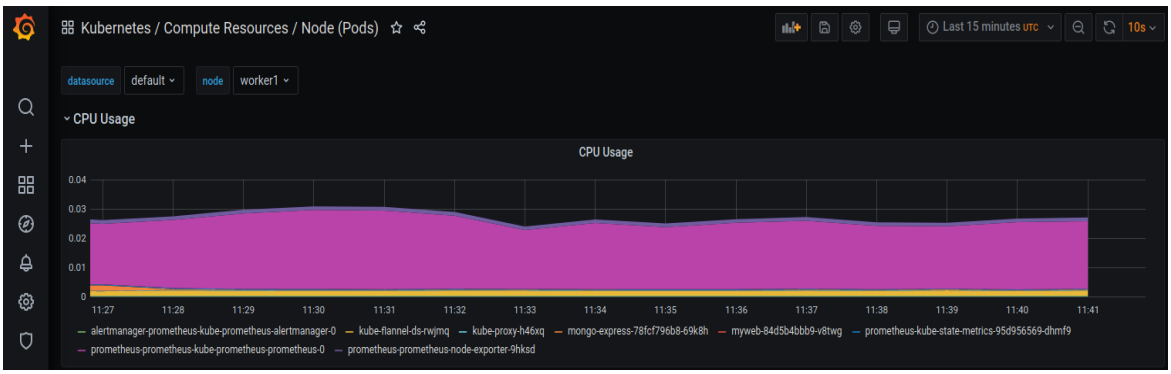


Figure 23: Grafana Worker Node CPU Usage

Figure 24: Grafana CPU Usage Graphic Colors code

After checking Prometheus and Grafana if we enter LENS we would see new graphical information available to us. Like Cluster wide or Node exclusive resource status.

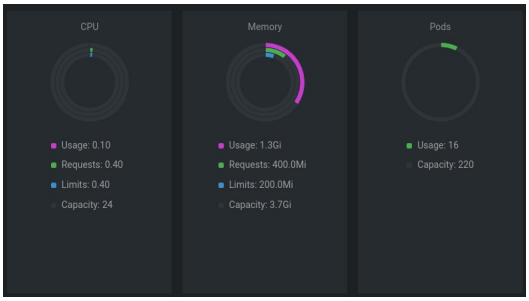


Figure 25: Cluster Wide Resource Chart

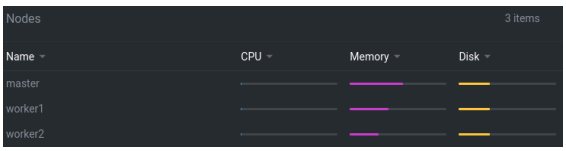


Figure 26: Node resources status

With that, we can conclude our implementation of this scenario.

5. Results

The final result of this thesis, after being through alleys of Docker, Kubernetes and related software. Is the obtention of three main clusters, Docker, Microk8s and Kubeadm. And another one that could be considerer for practice environments, Minikube.

In this section, we can find the analysis of the results of the previous cases we implemented.

5.1. Docker & Docker-Compose

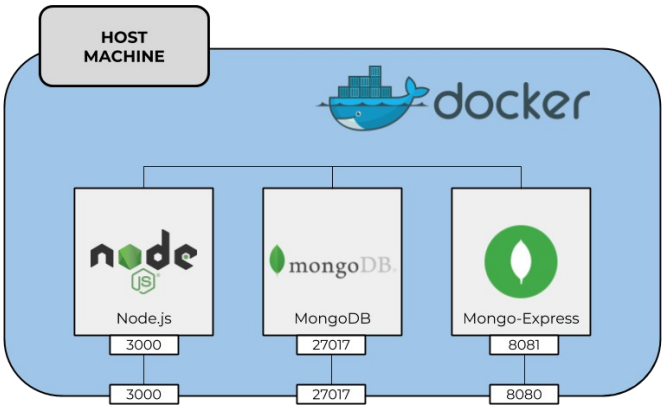


Figure 27: Docker Scenario

One of the most important purposes stated at the beginning of the document was to be able to demonstrate the significant impact of containers technologies in the process of development and implementation of applications and services.

Only with the part using Docker we have already demonstrated the effectiveness in reducing the time invested in the product's deployment in new environments and the broad compatibility of the containers. With the images from docker hub and after the containers, we cut off drastically the process of installation and configuration of MongoDB, Mongo-Express and Node.js framework.

Only MongoDB installation on Ubuntu machines takes around five commands plus all the configuration needed afterwards. Meanwhile, with Docker, only takes us three 'run' commands plus one for creating the docker network to have the complete setup up and running. Moreover, we are talking about not very complex applications.

Apart from that, we do not need to do different installation processes for each OS, such as installing MongoDB at windows or macOS. Because if Docker is installed, containers can be created and started without extra configuration.

At the second part, even though Docker simplifies the workload, the command syntax can be pretty tedious, and the management of more than two containers can get tricky. And there is where Compose comes to shine. Allowing the user to configure, run and manage multiple containers at once, making even more straightforward the task Docker already simplified.

Finally, to reinforce the wide compatibility concept, there is the execution of docker-compose file from Windows PowerShell at a Windows machine with Docker installed.

```
PS C:\Users\user\Desktop\app-web-compose> docker-compose up -build -d
```

After that and because Windows version of Docker comes with DockerDesktop, a user interface for Docker, the command's results can be seen there or just accessing *localhost* on ports *8080* o *3000* for mongo-express or web-app respectively.

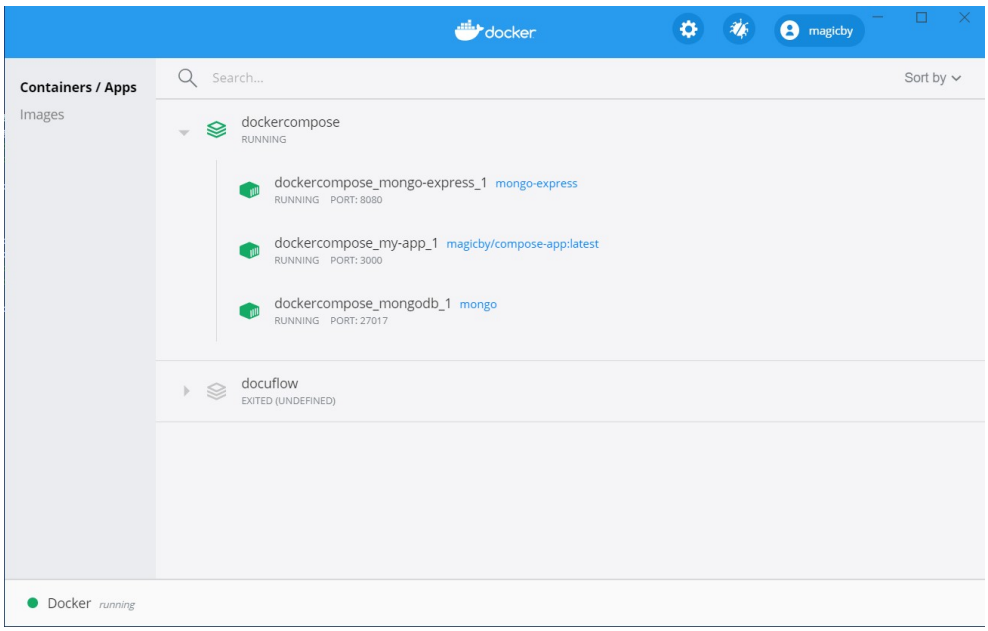


Figure 28: Docker Desktop Windows

5.2. Minikube

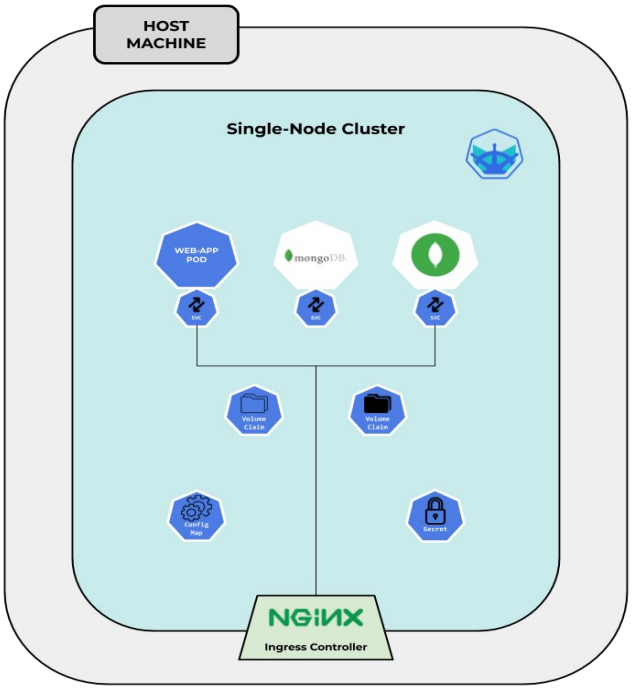


Figure 29: Minikube Cluster with K8s Components

Minikube offers a quick installation of a single-node k8s cluster, with simple commands and the same requisites as Docker.

The setup simplicity and architecture is the perfect development environment for creating and testing containerized applications and services without testing them at the production environment. It is like we have a little 'Kubernetes Development Kit' available within few commands.

Thanks to this, we have continued our transformation journey to turn our app into a containerized application and architecture.

This scenario helped create the Kubernetes foundation of our web-app. After testing the Kubernetes resources created, we can assure that using this technology is the right way to improve our deployment capacity in bigger environments.

But Minikube is just that, a simple K8s package. Its significant disadvantage is the incapacity of establishing and deploy to a multi-node cluster. That is why we needed the next scenario.

5.3. Microk8s

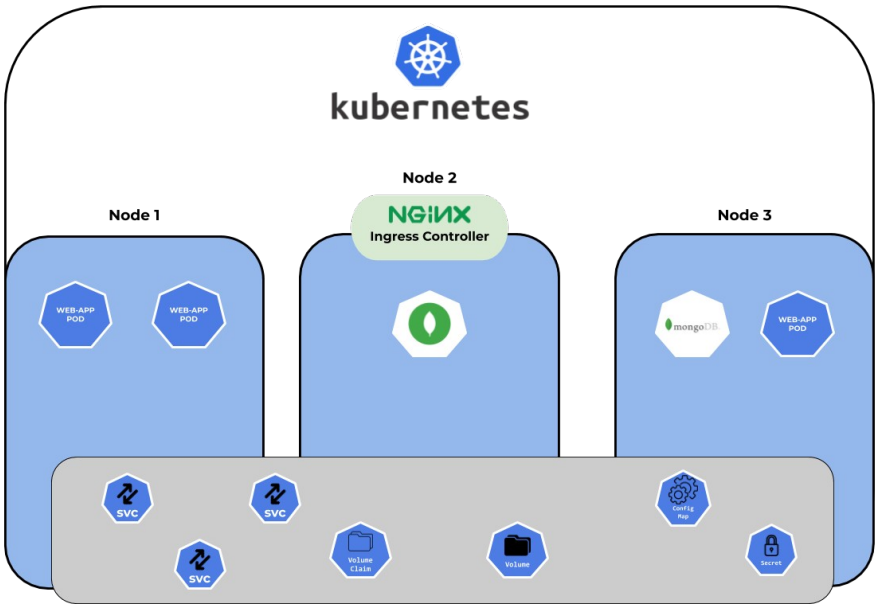


Figure 30: MicroK8s Cluster

Even though we finished with the same Kubernetes components as the scenario using Minikube. With Microk8s, we had our first contact with multi-node clusters and achieved a milestone.

Having the capacity to having more than one node offers us the capacity to scaling our application and increase cluster resources. With the Kubernetes features, we can do this without shutting down the cluster, as if we add more nodes k8s processes are smart enough to reorganize the workloads.

Furthermore, of course, we have the capacity of working a bigger application demand and workloads, enabling us to take more significant projects.

Apart from the advantages of having a multi-node cluster, Mircok8s provides us with a lightweight installation with useful add-ons as we have seen. Related to the options to install the NxIC, Option 2 is relatively quick and straightforward to use. However, Option 1 allows us to do a more personalized installation, as we can choose which version to use of the NxIC and have more customization on it.

But like all tools, Microk8s is not perfect. Customization is one of the counterparts of this type of deployments, as we do not know how the add-ons are precisely configured, and we have to rely on the creator's criteria. Also, Microk8s pre-installs some of the basics Kubernetes components, such as the API-Server or the Container Network Interface (CNI), which it uses Calico.

Another counterpart from our point of view is how Microk8s implements the architecture of Kubernetes. Instead of giving the master role to a node and

initializing the master processes inside, he starts the processes as snap daemons, that can be checked with 'microk8s.inspect':

```
$ microk8s.inspect
Inspecting services
  Service snap.microk8s.daemon-cluster-agent is running
  Service snap.microk8s.daemon-containerd is running
  Service snap.microk8s.daemon-apiserver is running
  Service snap.microk8s.daemon-apiserver-kicker is running
  Service snap.microk8s.daemon-control-plane-kicker is
running
  Service snap.microk8s.daemon-proxy is running
  Service snap.microk8s.daemon-kubelet is running
  Service snap.microk8s.daemon-scheduler is running
  Service snap.microk8s.daemon-controller-manager is running
  Copy service arguments to the final report tarball
```

At virtualized environments like at the LXC containers we are using, these processes can have problems booting after a shutdown or boot after Microk8s installation. And that has given us to many headaches during the implementation of the setup.

So, after seeing all of this, what if we do not want to use Calico? Maybe we want to try Flannel. What if we want to establish a multi-master cluster and test it. That is what the next setup is about.

5.4. Kubeadm

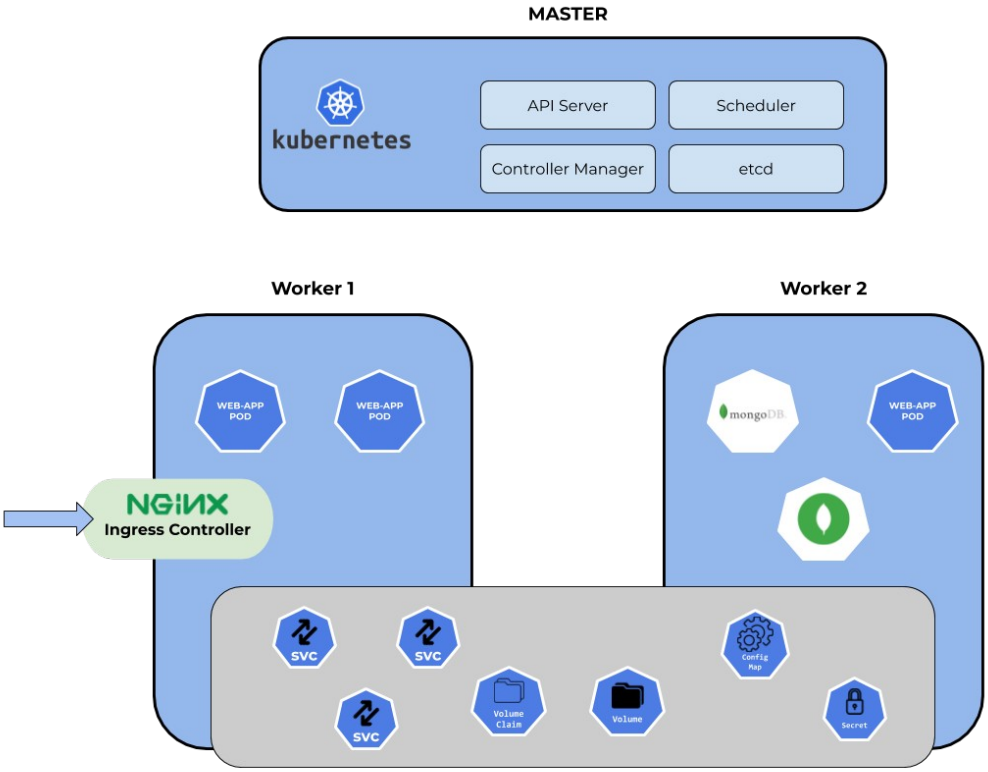


Figure 31: Kubeadm Kubernetes Cluster

Once we get used to the development of applications at containerized environments, we found out that the deployment of Kubernetes architecture itself is also a challenge. Deciding how to that is also a vital part of designing the architecture around the product we are designing.

In the previous setup, we've seen Microk8s and how he implements the Kubernetes architecture.And Kubeadm has proven to be less user friendly, and with more commands to know. However, on the other hand, it is a more clean Kubernetes installation, with the architecture expected. It has more potential and power if used correctly. Not having pre-installed components or add-on gives the user the freedom to choose what to install and how in the cluster. Plus the installation, starting of a cluster and node joining is way faster than with microk8s.

But on the other hand this increments the time needed to search, configure and install every component. And that could become tedious if it is not automated, like with shell scripts, or tools such as HELM are used. We've seen that HELM allows us to install applications composed of various Kubernetes resources like a package with just the command install. And that can reduce drastically the time spent deploying the resources.

Coming back to Kubeadm, we will give him the winner trophy is there was one. The simplicity and speed from zero to have the cluster running for us was clearly superior to microk8s.

6. Budget

This project has not required any prototype of hardware in order to be developed. The software and tools used are open-source programs or license-free. The primary resources of the project were me and my desktop machine with Ubuntu OS.
From the equipment point of view, the machine used needs to be medium to large range in performance. As virtualization is quite demanding on computers and we wanted to have various LXC containers running at once.

Equipment	Commercial Price	Quantity	Subtotal
Medium-Large range desktop Com	750.00 €	1	750.00 €
		TOTAL	750.00 €

Figure 32: Equipment Summary

The project counts on one principal worker plus an advisor. The worker is listed as a junior engineer and paid 15€/h for working a total of 504 hours, equivalent to the 18 ECTS on which the thesis project stands for, and each ECTS equals 28 hours. The advisor has computed a total amount of 10 hours, 1 hour for every meeting we had, plus 5 hours of assessment. And he is paid 35€/h. The taxes are considered at 18%.

Workers	Salary/Hour	Total Hours	Cost	Cost with taxes
Junior Engineer	12.00 €	504	6048.00 €	7,136.64 €
Project Advisor	35.00 €	15	525.00 €	619.50 €
			TOTAL	7,756.14 €

Figure 33: Summary of Personal Salaries

With that the total budget of the project is **8506.14€**.

As an extra, and as technologies studied at this project are heavily related with cloud service providers there is the cost of having a three nodes Kubernetes cluster at cloud provider Linode[19]. The cost had been calculated using the pricing tool available at their website.

Equipment	Price/Month	Quantity	Months	Subtotal
4GB 2Cores 80GB SSD	20.00 €	2	12	480.00 €
80GB Data Storage	8.00 €	1	12	96.00 €
Node Balancer	10.00 €	1	12	120.00 €
			TOTAL	696.00 €

Figure 34: Hosting Summary

7. Conclusions

First of all, I want to expose how I feel during the realization of this project. From the beginning, this project allowed me to study and analyze one of the most active technology flows, which are the containerized applications and micro-service architectures. Mora exactly Docker and Kubernetes.

At the start, I did not know a lot about Docker or Kubernetes, and I had just discovered that Kubernetes existed. So I was afraid that the development of the project would become a bumpy ride. However, it was more like a journey of a child in a toy store in the end.

As I was investigating Docker, I realized the great potential of containers and their influence on the development of services. And as soon as I started with Kubernetes, I was astonished of what it makes possible and the range of features it has. I kept studying Kubernetes, and I could not choose exactly what functionality, feature or service to base my project on. I was like a kid wanting all the toys on the store.

But time kept ticking, so taking into account my little initial knowledge, I decided to carry out the project focused as a transformation journey. Adapting the deployment of a web-app and its components to the world of Docker and Kubernetes.

To be fair, I never also worked with LXC containers, and now, after the project, I can say without a doubt they are a powerful tool that helps to recreate scenarios otherwise impossible like having multi a Kubernetes multi-node cluster. And that was key in the realization of this project, as, without LXC, we could not implement our scenarios.

They are powerful and complex to configure right, making necessary a large formation period before having an acceptance level.

Another conclusion is that each Set-up we ran has shown that each technology has its benefits and cons. Showing that there is no perfect solution for how to develop, deploy and maintain a service. Which means we have to choose the one more suitable for the resources we have and projects requirements.

Docker and derivates are handy for developers that are starting with containerized services. Also, because of how they are designed, they are only suitable at small environments, where one or two machines are more than enough to deal with the workload and fulfil the requisites.

Next, Kubernetes, which is not a substitute or a rival of Docker, it is a container manager which relies on container runtime to be installed at the machines in order to be able to create and run them. So Kubernetes is a tool to manage a large number of containers and surged because of the great success of

Docker that caused an increase in the number of containers running at microservice architectures.

With that in mind, it is a powerful tool to use and implement microservices architectures based on containers at big and dynamic environments. All this potential cames with the cost of the complexity. With that project, we acquired a wider knowledge of what Kubernetes is, but believe me when I say it could be totally possible to realize a thesis for each of the points viewed at the 'State of the Art'. Kubernetes is a long journey that needs experience and time to build enough confidence in working with him.

Following the Set-ups implemented at the current project, they got in common, that all of them are improvements to the traditional way of conceiving application and services. Cutting the time lost in configuring software packages at the final environment, or having the implicit wider compatibility comes with container technologies. We have seen how installing Docker, and with just one command, we have a fully deployed and functional application at any OS that supports Docker.

Furthermore, this should draw our attention to software developers, as adapting a software package to be deployed with containers, either using Docker or K8s, is not an easy task. This is causing many of the more used services, like GitLab, is still being developed to be deployed with Kubernetes.

Before finishing and regarding the various types of Kubernetes implementations, we have seen. All the flexibility and features they provide amused me since the start. They reinforce the idea that Kubernetes is a potent tool for deploying services and a vast base of users and companies behind it that does not stop developing and creating new functionalities.

Finally, and as a personal contribution, I would like to say that this project has generated an interest in this type of technologies. And I will use it as a starting point to continue researching and learning, whether as a personal, professional or academic purpose.

8. Future Development

This project sets the basic knowledge of Docker and Kubernetes. From here, there are a lot of project and research possibilities, how networking work with clusters, the multiple types of CNI's, Kubernetes infrastructure using cloud providers, creation of custom resources and more. This means that they are a long journey that this project just has started.

Focusing on the technologies, the future is plenty of continuous development and updates for them, as the creators of each have confirmed. The community also has a lot of research and development to do to bring the most used services and software packages to the container format so they can be deployed using Docker and Kubernetes.

An example of that continuous development is the version 1.23.0+ of Kubernetes that will be deployed in late 2021. This version will drop the Docker support for the Container Runtime Interface(CRI) of Kubernetes because Docker was the most used container runtime. Kubernetes added a feature,'dockershim', in their CRI to enable it to use Docker. Others container runtime supported by Kubernetes does not need that feature. Despite being removed in the future, Docker has announced the support of 'dockershim' as a stand-alone Kubernetes complement in case anyone wants to have Docker as the runtime still. This removal will cause better performance and less overhead during the communication with the container runtime. Besides the removal, there is no need to worry as the existing images will keep working.

An indirect cause of the change will be the increase in the use of the 2nd most popular container runtime Containerd[20]. Containerd was part of Docker. They extracted it as a separate component to run it without all the extra features that make Docker so easy to use for the user but unnecessary for the processes.

Containerd also forms part of the Cloud Native Computing Foundation projects. It is conceived as an industry-standard container runtime created to be as simple, robust and portable as possible.

Maybe this is the opportunity for the next thesis to be based on Containerd and Kubernetes and explore the newer versions' potential.

Another point for future development could be the deployment of Kubernetes clusters using cloud provides, such as Amazon Web Services or Google Cloud. And use them to create a Platform as a Service(PasS) or Software as a service(SaaS) architecture that are increasing their popularity these days.

References

1: Docker Inc. Docker Prerequisites. [Online] Available. 2020: https://docs.docker.com/engine/install/. [Accessed: 02 September 2020]

2: Canonical Ltd. Microk8s Requirements. [Online] Available. 2020: https://microk8s.io/docs. [Accessed: 17 November 2020]

3: Canonical Ltd. Linux Containers. [Online] Available. 2020: https://linuxcontainers.org/. [Accessed: 10 November 2020]

4: Docker. DockerHub. Online Available. 2020: www.docker.com. [Accessed: 07 September 2020]

5: Docker Inc. Dockerfile Reference. [Online] Available. 2020: https://docs.docker.com/engine/reference/builder/. [Accessed: 09 September 2020]

6: Docker Inc. Docker CLI Reference. [Online] Available. 2020: https://docs.docker.com/engine/reference/commandline/docker/. [Accessed: 10 September 2020]

7: The Kubernetes Authors. What is Kubernetes . [Online] Available. 2020: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed: 15 September 2020]

8: Red Hat Inc. Flannel Docs. [Online] Available. 2020: https://docs.projectcalico.org/getting-started/kubernetes/quickstart. [Accessed: 10 October 2020]

9: Tigera. Calico Docs. [Online] Available. 2020: https://www.projectcalico.org/. [Accessed: 10 October 2020]

10: The Kubernetes Authors. Kubectl Commands. [Online] Available. 2020: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands. [Accessed: 17 October 2020]

11: The Kubernetes Authors. Kubeadm. [Online] Available. 2020: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. [Accessed: 24 November 2020]

12: Helm Authors. HELM Documentation. [Online] Available. 2020: https://helm.sh/docs/intro/. [Accessed: 26 October 2020]

13: Mirantis Inc.. LENS Documentation. [Online] Available. 2020: https://docs.k8slens.dev/. [Accessed: 26 October 2020]

14: The Kubernetes Authors. Minikube Documentation. [Online] Available. 2020: https://minikube.sigs.k8s.io/docs/start. [Accessed: 30 November 2020]

15: The Kubernetes Authors. Kubectl Install. [Online] Available. 2020: https://kubernetes.io/docs/tasks/tools/install-kubectl/. [Accessed:10 November 2020]

16: F5 , Nginx Inc. Nginx Ingress Controller. [Online] Available. 2020: https://docs.nginx.com/nginx-ingress-controller/installation/installation-with-helm/. [Accessed: 12 November 2020]

17: Canonical Ltd. Microk8s Requirements. [Online] Available. 2020: https://microk8s.io/docs. [Accessed: 17 November 2020]

18: The Kubernetes Authors. Kubeadm Install. [Online] Available. 2020: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. [Accessed: 24 November 2020]

19: Linode Inc. Linode Home Web. [Online] Available. 2020: https://www.linode.com/. [Accessed 09 January 2021]

20: Containerd Authors. Containerd. [Online] Available. 2020: https://containerd.io/. [Accessed 08 January 2021]