Ca' Foscari University of Venice

Zamperla Analytics Service: design and deployment of a microservice architecture with Kubernetes

Computer Science and Information Technology

Master's Degree program

Stefano Sello - 864851

AY 2022-23

Advisors: Prof. Pietro Ferrara, Dr. Gianluca Caiazza

Department of Environmental Sciences, Informatics and Statistics
Ca' Foscari University of Venice

Abstract

This thesis introduces Kubernetes, a software that falls into the category of container orchestrators, and it applies it to the deployment of a data collection and processing application developed for the company Zamperla, a supplier of thrilling rides for amusement parks. Rides produce huge amounts of data when they are in operation. Remotely collecting, storing, and processing such data is a computational challenge that requires the adoption of modern software architectures based on the microservice pattern, and to replicate and orchestrate them through technologies like Kubernetes. The work is based on the candidate's experience gained during the thesis project, which involved using technologies such as Docker, RabbitMQ, and, precisely, Kubernetes, applied to a microservices architecture project developed by third-party actors within the academic realm.

Contents

Contents ii

1 Introduction 1

2 *Docker* and *Kubernetes* Overview 3
- 2.1 Docker and containerized applications 3
- 2.2 Kubernetes and microservices 4
- 2.3 Kubernetes basic concepts 7

3 Development environment 10
- 3.1 *Docker* Setup 11
- 3.2 *Kubernetes* setup 12
  - 3.2.1 *Minikube* installation and setup 13
  - 3.2.2 `kubectl` installation and setup 14
  - 3.2.3 *Kubernetes* `Hello World` 16

4 Design and Implementation of the infrastructure underlying the Zamperla Analytics application 20
- 4.1 System overview and application services 20
  - 4.1.1 *UI* component 21
  - 4.1.2 *BackEnd* component 22
  - 4.1.3 *DB* component 22
  - 4.1.4 *Data* component 23
- 4.2 Basic service/deployment setup: the UI component 24
  - 4.2.1 Setting up a `deployment` 24
  - 4.2.2 Setting up a `Service` 27
- 4.3 Adding persistent storage: the DB component 29
  - 4.3.1 Requesting storage: `PersistentVolume (PV)` 31

4.3.2 Using storage: `PersistentVolumeClaims (PVC)` . . . 33
4.3.3 `Pod` environment variables and database initialization: `ConfigMaps` 35
4.4 Summing up: BackEnd component and Data component . . 37
4.4.1 The *Data* component configuration 38
4.4.2 Project directories structure 39

5 Scaling jobs with RabbitMQ and Keda 41
5.1 Scaling data-intensive jobs: an application need 41
5.1.1 Message queues: how *RabbitMQ* helped to manage complexity 42
5.1.2 `Jobs`, i.e. *Kubernetes* disposable `Pods` 43
5.1.3 *Keda*: autoscaling Jobs 44
5.2 Jobs split-up: `reporter`, `postprocessor` & `sftp_downloader` 45
5.2.1 Step 1: startup script 45
5.2.2 Step 2: *Docker* image 48
5.2.3 Step 3: *Kubernetes Jobs* and *Keda* 52
5.2.4 Step 4: turn direct calls into messages 57
5.3 Different jobs require different priorities 59
5.3.1 Hierarchize *Kubernetes* objects with `PriorityClass` . 60

6 Infrastructure monitoring and observability 62
6.1 Prometheus: observability implementation in a *Kubernetes* cluster 63
6.1.1 Introduction to Prometheus 63
6.1.2 *Prometheus* setup in a *Kubernetes* environment 64
6.2 Next steps: *Graphana* and Alert System 69
6.2.1 Graphana: a visualization tool 69
6.2.2 Alert Manager: real-time alerting system 72

7 Conclusions 73

Bibliography 75

Chapter 1

Introduction

The goal of this master thesis concerns the design and implementation of a *Kubernetes* cluster able to properly execute, replicate, and orchestrate a *microservices* application. Such application collects and processes large amounts of data coming from several rides located in different amusement parks. The core microservices were previously developed mostly in *Python*, while the data elaboration core software was developed in *Rust*, and it was considered an untouchable black box. The aim of this project is the adoption of Kubernetes to build a scalable and reliable system able to process the requested jobs of data retrieval and manipulation in the most efficient, scalable, and effective way, with limited hardware resources. This thesis will discuss the reasons why *Docker* and *Kubernetes* were introduced in the first place, and the entire design process that took from the initial idea to the completed infrastructure. The architecture that is going to be explained consists of four main components, all described in detail in section 4.1, which are: a Web user interface, a backend application, which provides a data-access layer and an interface that provides cross-components interaction, a *PostgreSQL* service, used for data persistency, and a data manipulation component, which is the core of this system and responsible for those tasks that require to interact with data, such as the retrieval of remote data and the report generation task. All these services had initially their application code and their own `Dockerfile`. During the development of this project, a set of *Kubernetes* (explained in sections 4.2, 4.3 and 4.4) objects have been designed and developed in order to make each component up and running inside a *Kubernetes* cluster. Then, in chapter 5 the service dedicated to the data manipulation tasks has been divided into one Web service and three independent jobs, in order to improve scalability and fault-tolerance. Technologies like *RabbitMQ* and *Keda* have helped to achieve this goal. Finally, in chapter 6 it has been discussed how to set up an observability infrastructure to monitor the performances and the behavior of a system running in a *Kubernetes* cluster, in particular using *Prometheus* as the main application layer responsible for collecting and analyzing performance data.

Chapter 2

Docker and *Kubernetes* Overview

In recent years, the proliferation of Web applications, followed by huge growth in the complexity of the hardware infrastructures that these applications were backed by, has led to an incremental interest of the developer community in containerization technologies. Even if virtualization and resource isolation technologies were available since 1979 with `chroot`[1], the real game-changing software product that made more and more IT professionals switch to container-based applications was *Docker*, released in 2013[2].

2.1 Docker and containerized applications

Docker is a virtualization technology that allows users to execute virtual environments, called *containers* starting from a sort of *snapshot*, called *image*, which is built itself from a written declarative description of how this container should be made and what it should contain. Unlike standard virtual machines, which are abstractions of physical machines,*Docker* containers are an abstraction of the application layer that packages code and dependencies together.[3] The main advantages of adopting *Docker* as containerization technology were (and still are) the following ones[4]:

- Since containers include the minimum configuration needed to run the containerized application, **deployments are really fast**.
- Containers include both the operating system and the software components needed to run a containerized application, which means that

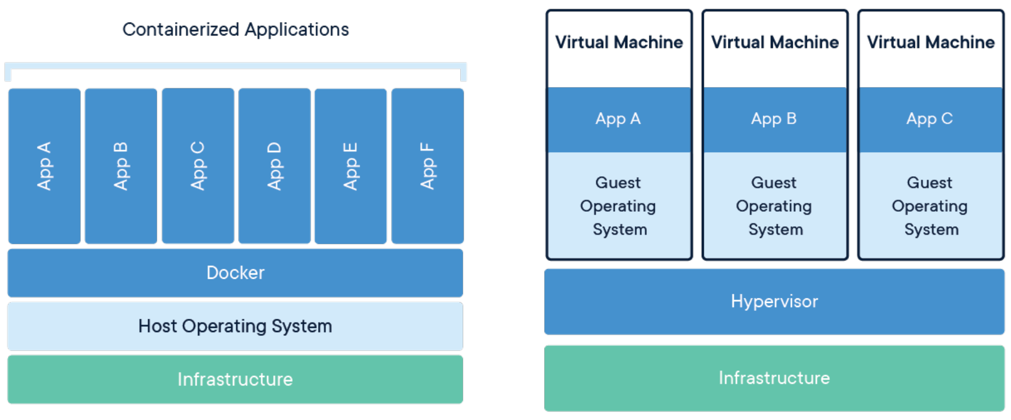


Figure 2.1: *Docker* and Hypervisor different approaches to virtualization

no OS/kernel constraint is required, and no other software except an installation of the *Docker* runtime is needed in any physical or virtual machine to run such application. Hence, **containerized software is extremely portable**.

- Using containers, developers can take advantage of the **version control system**, which allows simple version tracking (with changelog inspection) rollback operations.
- The built **containers** can be **shared among developers** thanks to public registries, making it easy to reuse popular and solid containers.

All these features make *Docker* one of the most widely used technologies in the development of modern applications, but it does not suffice if the objective is to deploy large-scale automatically scalable self-healing systems eventually made of dozens of different components. This is where *Kubernetes* comes into play.

2.2 Kubernetes and microservices

Kubernetes is a technology born to tame the complexity of large and complex applications. It is no coincidence that it was first developed by Google in 2014[5]. *Kubernetes* is a *container orchestrator*, i.e., a software able to automate most part of the operational effort required to run containerized services and workloads. Why do present-day software developers need software like this? Isn't it easier to provision physical servers in the "old but gold" manual way rather than learning a completely new technology that leads to almost the same results? The answer lies in the dramatic differences between the *monolithic* approach and the *microservices* approach to software architectures.

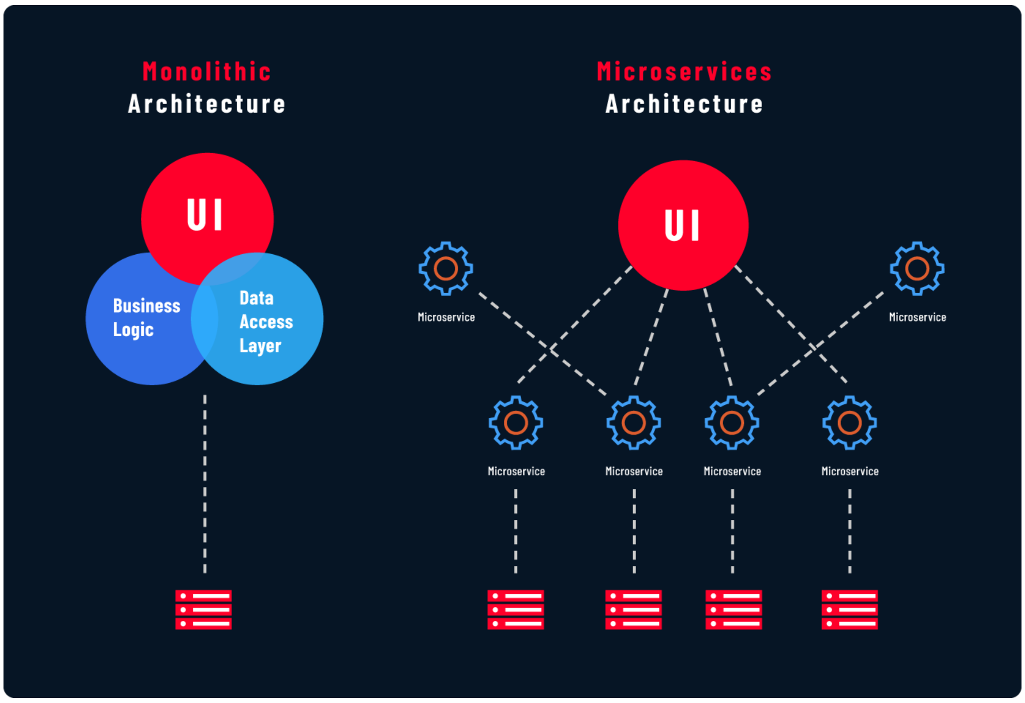


Figure 2.2: Monolithic architecture vs. microservices architecture

In *monolithic* software architectures all the modules that contribute to the correct execution of a program (such as data access layer, business logic, user interface, etc.) reside in a single program, which is deployed as a standalone application in a single machine. This type of approach has been widely adopted for many years and still is. But in recent years another approach has become popular: the *microservices* architectural pattern[6]. When using a microservices approach, every module needed for the application to work properly can be developed and deployed as a standalone service, possibly on different physical or virtual servers. Every *service* can access different databases and can potentially interact with every other service through lightweight protocols, such as `HTTP` or `AMQP`. There are several advantages that can lead a software developer to choose to adopt a microservices architecture instead of a monolithic approach. Some of the most important are the following[7].

- **Better maintainability:** When software applications grow in size and functionality, complexity tends to increase, and maintainability becomes harder to accomplish. Breaking a system into smaller, self-deployable, and standalone modules allows developers to test and develop each service independently from the other, knowing little or nothing about how other microservices work. Nevertheless, smaller codebases are better for readability and understandability, which makes them easier to maintain and update.
- **Improved scalability:** Frequently, there are a few components in a software application that require a higher amount of resource allocation, while other components require lower resources. If the application modules reside in microservices, developers can allocate resources and set up replicas and auto-scaling systems only for those modules that have a higher demand for resources and reliability.
- **Components replaceability:** Since every component interacts with the others using a specific formal protocol, each of these components can be replaced with another one that "talks the same language", i.e., uses the same protocol to interact with the sibling services. This possibility becomes useful when old legacy services need to be replaced with others developed with newer technologies or made available by third-party actors such as S.a.a.S. products.

Certainly, there are also some disadvantages when adopting microservices as an architectural choice. The main compromises that must be taken into account when transitioning to a microservice architecture are the following ones[8]:

- **Communication overhead:** Communication between modules happens in a network environment where messages and information exchange introduce an increased latency if compared to in-system calls of monolithic applications.
- **Increased operational complexity:** Managing and monitoring many applications in a microservices environment can be more complex than handling a single monolithic application, as it requires robust monitoring, logging, and error handling across different services.

Depending on the needs of the application and the previously made architectural choices, adopting a microservices architectural approach can be more or less suitable. In recent years, many products and methodologies have evolved in the direction of service-oriented architectures, enhancing the benefits and mitigating the disadvantages. For example, all the main players in the field of cloud computing services (Amazon with AWS, Google with Google Cloud, and Microsoft with Azure) are evolving their IaaS products, making microservices-designed architectures more and more appealing to software developers and DevOps engineers. Of course, *Kubernetes* is also one of the technologies that enabled this transition from monolithic architectures to service-based architectures.

Even if in the last couple of years microservice architectures have been involved almost everywhere for the creation of new products, there are now some perplexities about the adoption of *Kubernetes* and microservice solutions for every kind of application. Indeed there are some scenarios where microservices are useful and others where they bring a lot of useless complexity. There are big tech companies, like *Shopify*, which are deconstructing their monolith product into a more manageable componentized code base without involving the microservices paradigm, because not suitable for their needs.[9] There are authoritative and respected voices in the tech industry, like the one of David Heinemeier Hansson, creator of the widely-used Web framework *Ruby on Rails*, which say that the microservice pattern is useful and inevitable when it is used in a company of thousands of developers working on the same product but should be avoided as long as it can be when the team is relatively small, even if the product is big. For these cases, there exists the *Majestic Monolith*. [10] In other words, the microservices world is living its maximum in Gartner's Hype Cycles[11]: as a new technology adopted by the majority of the Big Tech companies, it is something that everyone wants to use and master. But soon the hype for this new software creation paradigm will fall, leaving room for the rationality to choose the best technology. It will initially seem to fall in complete disuse, due to the fact the disadvantages will be exaggerated and advantages will be belittled. Eventually, the adoption of this infrastructure philosophy will reach an equilibrium: large companies will continue to use microservice infrastructures because it is more convenient for realities with many developers; small companies will understand that maybe a more traditional monolith can do the same things being more maintainable, observable and easy to understand.

2.3 Kubernetes basic concepts

Kubernetes is a **container orchestration tool**, which means that it is responsible for executing containers and managing the resources behind them. *Kubernetes* comes with many handy features, such as container failure recovery, auto-scaling, and automatic resource allocation. But how is it done? What modules and components cooperate to achieve these goals? *Kubernetes* is a complex and articulated system. From a hardware point of view, it requires at least two different entities:

- **Control plane node** (or master node): a node of the cluster that contains all the modules needed to interact with the other nodes and with all the *Kubernetes* modules. More in-depth, *API server*, *Scheduler*, *Controller / Manager* and *etcd* are the main services installed on the master node, each of which is required to execute a part of the *Kubernetes* infrastructure. Application services are generally not executed on the control plane node.
- **Worker node(s)**: a node (or a set of nodes) responsible for the execution of the application containers. The master node communicates with the worker nodes in order to tell them to initiate a new container, recover an old one, and so on. Worker nodes also send to the master node information about the healthy or unhealthy state of services and the resource allocation possibilities.

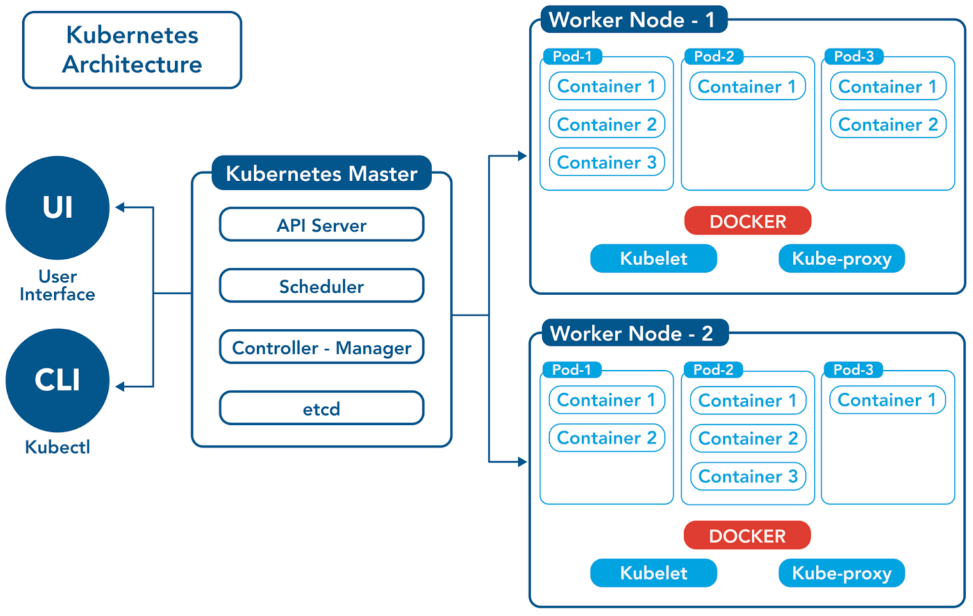


Figure 2.3: *Kubernetes* architecture components and interactions

Figure 2.3 shows which are the main modules required by *Kubernetes* to work properly and how they interact. A brief description of these modules follows[12].

Control plane modules

- **API server** (`kube-apiserver`) is the frontend server responsible for handling API requests.
- **etcd** is a persisted database where *Kubernetes* stores the information it needs: what nodes are part of the cluster, which resources exist in each node, etc.
- **Scheduler** (`kube-scheduler`) is the module that decides where the newly created pods should run.
- **Controller - Manager** (`Kube-Controller-Manager`) is responsible for running resource controllers, such as *deployments*.

Worker nodes modules

- **Kubelet** is the main worker nodes service which drives the container runtime to start the workloads scheduled for the node, monitoring also their status.
- **Kube-proxy** is the module responsible for the communication between pods and between the cluster and the Internet.
- **Docker** is the software that actually starts and stops containers and handles their communication. *Docker* is the most widely used software for this purpose, but *Kubernetes* supports also other *Container runtimes* like `rkt` or `CRI-O`.

All these modules allow developers to create objects through a RESTful API. Each object is created and managed within the perimeter of the cluster, taking the memory, storage, and computational power resources from the worker nodes that make the cluster infrastructure. The object manipulation request (creation, deletion, update, etc.) is sent to the *API server*, which stores or modifies the related object state information in the *etcd*. If the request involves the creation of new pods, the *Scheduler* is responsible for allocating the required resources. Finally, the *Controller-Manager* decides when and how allocated pods should start or stop. There exist many types of *Kubernetes* objects. The most widely used are *deployments*, *services*, *pods*, and *volumes*. Chapter 4 will explain in depth which types of objects have been used for this project and how they operate.

Chapter 3

Development environment

Info: For the sake of clarity, from now until the end of this elaborate the name of specific technologies will be written in *italic* (*Kubernetes*, *Docker*, *KEDA*, etc.), while the *Kubernetes* resource types will be written in `monospaced` *CamelCase* (`ConfigMap`, `Service`, ect.). This distinction is useful to discriminate between words that have a double meaning. For example, `Services` are *Kubernetes* objects, services are units of a microservice architecture.

In this chapter, we will go through the steps necessary to establish the basic development environment for running a *Kubernetes* cluster in a local machine. The list of installations and setups applies to all platforms, but the instructions provided in this thesis will focus on a *MacOS* environment. If there are differences between *Linux*-based operating systems and *MacOS*, notes will be included to point them out.

The conditions underlying the development of this project, in terms of operating system version and hardware specifications, are the following:

- **Operating System:** MacOS Ventura 13.4
- **CPU:** Apple M1
- **RAM:** 8 GB unified memory
- **Drive:** 256 GB SSD

3.1 *Docker* Setup

First Of all, the *Docker* runtime needs to be installed. For *MacOS* environments, the *Docker* development team provides a `.dmg` file that allows users to install *Docker Desktop* (as well as the *Docker* runtime) following some basic steps through a graphic user interface. Details and download links are provided on the following official Web pages: https://docs.docker.com/desktop/install/mac-install/. For *Linux* users there are different procedures depending on the *Linux* distribution. Almost every *Linux* distribution supports the installation of the *Docker* runtime through *Docker Desktop* (which is distributed through `.rpm` and `.deb` packages, available at the following official link: https://docs.docker.com/desktop/install/linux-install/). However, *Docker Desktop* comes with a limiting user agreement that does not allow free use of the software for certain categories of users. Due to these limitations, it may be more appropriate to opt for a standalone installation. Taking as an example a *Debian*-based distribution like *Ubuntu*, the installation requires the following steps:

1 Update the `apt` package index and install packages to allow `apt` to use a repository over `HTTPS`:

Command Line

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg
```

2 Add *Docker*'s official `GPG` key:

Command Line

```
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg
↪  | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

3 Set up the *Docker* repository:

Command Line

```
$ echo "deb [arch="$(dpkg --print-architecture)"
↪  signed-by=/etc/apt/keyrings/docker.gpg]
↪  https://download.docker.com/linux/ubuntu "$(.
↪  /etc/os-release && echo "$VERSION_CODENAME")" stable"
↪  | sudo tee /etc/apt/sources.list.d/docker.list >
↪  /dev/null
```

3 Update the `apt` package index:

Command Line

```
$ sudo apt-get update
```

4 Install the *Docker* engine:

Command Line

```
$ sudo apt-get install docker-ce docker-ce-cli
↪  containerd.io docker-buildx-plugin
↪  docker-compose-plugin
```

The whole procedure is fully described on the reference page. There are other available installation options, depending on the OS version, or the will to install a specific binary or a specific version of the engine, but these are most useful for this thesis.

3.2 *Kubernetes* setup

Setting up *Kubernetes* is a difficult task initially, especially for development purposes. First, we must choose between different *Kubernetes* distributions. *Kubernetes* is a tool designed mainly for production purposes. This means that it is hard to install and configure such a system on a development machine. Luckily, some products have been developed for this specific purpose: to provide a lightweight *Kubernetes* installation for local environments, allowing for enhanced development and debugging. Here, there are some alternatives:

- **Minikube:** the most widespread lightweight solution. It enables developers to run a single-node *Kubernetes* cluster on their local environment. It has a very good performance and is very easy to install. Here there is a reference to the project page: minikube.sigs.k8s.io
- **KIND (Kubernetes In Docker):** a tool originally developed to test *Kubernetes* itself that allows developers to run *Kubernetes* nodes into *Docker* containers. It can be useful to test infrastructures that need to be as close as possible to a production environment with many nodes. To learn more visit the project webpage: kind.sigs.k8s.io
- **k3s:** a lightweight *Kubernetes* distribution developed for production purposes, in particular for environments with low resource availability. Due to its lightweightness, it can also be used for development in local machines, but it is only available for *Linux* operating systems. Please visit the official project page for more details: k3s.io
- **Docker Desktop:** the already mentioned *Docker Desktop* provides a built-in *Kubernetes* solution that allows developers to interact with a *Kubernetes* cluster.

For this project, *Minikube* has been chosen since it has better performance in low-resource environments, it has a strong support community and, in general, it is easier to find guides and documentation based on it. Therefore, from now on, let us assume that *Minikube* is the underlying technology used to run the development single node *Kubernetes* cluster.

3.2.1 *Minikube* installation and setup

According to the official installation instructions, a local machine running *Minikube* requires at least:

- 2GB of free RAM
- 2 CPUs
- 20 GB of free storage
- an internet connection
- a preinstalled container manager (in this case we have already seen how to install *Docker*, but *Minikube* supports also other container and VM managers like *Podman*, *VirtualBox* and *Hyperkit*

To install and configure *Minikube* on *MacOS*, it suffices to execute the following commands:

1 Download and install the binary:

Command Line

```
$ domain=https://storage.googleapis.com
$ base=$domain/minikube/releases/latest
$ arch=darwin-amd64
$ curl -LO $base/minikube-$arch
$ sudo install minikube-$arch$ /usr/local/bin/minikube
```

Depending on the OS distribution and the desired *Minikube* version, there are different binaries made available. A whole list of *Minikube* executables can be found at https://github.com/kubernetes/minikube/releases.

2 Start your cluster:

Command Line

```
$ minikube start
```

That's it: now the single-node *Kubernetes* cluster underlined by *Minikube* is in execution on our local development machine.

3.2.2 `kubectl` installation and setup

Even if the development cluster is up and running, we can not yet interact with it, since a *client* for the *Kubernetes API Server* is missing. As explained in section 2.3, *Kubernetes* allows to manipulate its internal objects, such as `Pods`, `Services`, and `Volumes`, through a REST API. Requests to the latter are made to the *Kubernetes API Server*, and to execute these requests, a client able to handle this task is needed. Here `kubectl` comes in action. `kubectl` allows developers to interact with a *Kubernetes* cluster providing a simple *cli* interface that can be used to create, modify, get, and delete all *Kubernetes* objects in a cluster. The following are the steps required to install `kubectl`.

1 Download the latest version:

Command Line

```
$ arch=amd64
$ os=darwin
$ domain=https://storage.googleapis.com
$ baseurl=$domain/kubernetes-release/release
$ version=$(curl -s $baseurl/stable.txt)
$ curl -LO $baseurl/$version/bin/$os/$arch/kubectl
```

Note that, for different CPU architectures and operating systems, it suffices to change the values of `$arch` and `$os`.

2 Make the file executable

Command Line

```
$ chmod +x ./kubectl
```

3 Move the executable file into a `PATH` directory:

Command Line

```
$ mv kubectl ~/bin/kubectl
```

Assuming that there exists the directory `/bin` and that its path is included in `$PATH`.

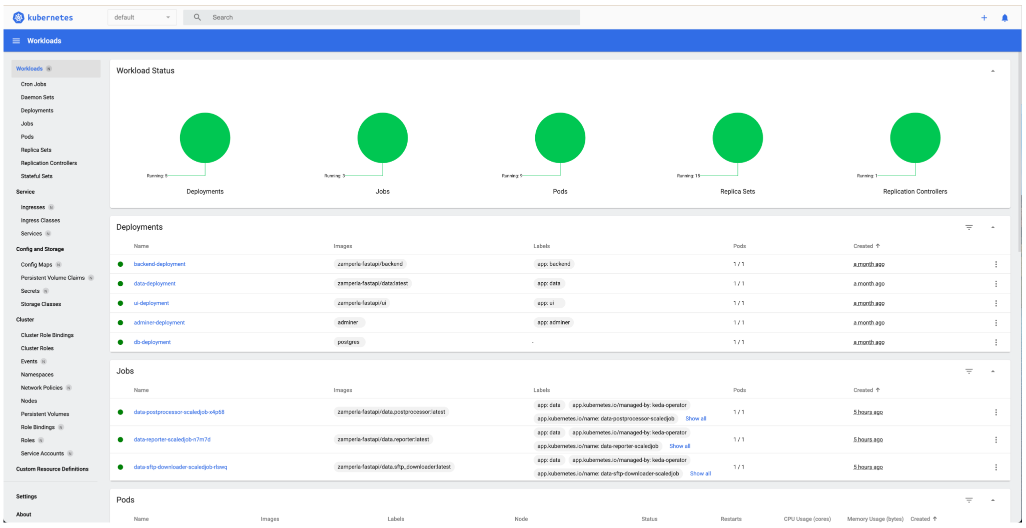
Executing the command `kubectl -h`, a usage suggestion should be obtained.

```
kubectl controls the Kubernetes cluster manager.

 Find more information at:
 ↪  https://kubernetes.io/docs/reference/kubectl/

Basic Commands (Beginner):
  create          Create a resource from a file or from stdin
  ...
```

3.2.3 *Kubernetes* `Hello World`

Now that all the components required to run a *Kubernetes* cluster are installed in the development machine, a good way to check if the whole system is working properly is that of executing some simple `kubectl` commands to see if the cluster reacts as expected.

The first command to test is `minikube dashboard`, which will open in the browser a *Kubernetes* dashboard containing a lot of useful information. If this command succeeds, the cluster is proven to be up and running.



Deployments

	Name	Images	Labels	Pods	Created ↑
●	backend-deployment	zamperla-fastapi/backend	app: backend	1 / 1	a month ago
●	data-deployment	zamperla-fastapi/data:latest	app: data	1 / 1	a month ago
●	ui-deployment	zamperla-fastapi/ui	app: ui	1 / 1	a month ago
●	adminer-deployment	adminer	app: adminer	1 / 1	a month ago
●	db-deployment	postgres	-	1 / 1	a month ago

Jobs

	Name	Images	Labels	Pods	Created ↑
●	data-postprocessor-scaledjob-x4p68	zamperla-fastapi/data.postprocessor:latest	app: data app.kubernetes.io/managed-by: keda-operator app.kubernetes.io/name: data-postprocessor-scaledjob Show all	1 / 1	5 hours ago
●	data-reporter-scaledjob-n7m7d	zamperla-fastapi/data.reporter:latest	app: data app.kubernetes.io/managed-by: keda-operator app.kubernetes.io/name: data-reporter-scaledjob Show all	1 / 1	5 hours ago
●	data-sftp-downloader-scaledjob-rlswq	zamperla-fastapi/data.sftp_downloader:latest	app: data app.kubernetes.io/managed-by: keda-operator app.kubernetes.io/name: data-sftp-downloader-scaledjob Show all	1 / 1	5 hours ago

Pods

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑

Figure 3.1: Minikube dashboard view

Other operations useful to try out:

A Create a `Deployment`:

Command Line

```
$ kubectl create deployment nginx --image=nginx
$ kubectl describe deployment/nginx
```

```
Name:                    nginx
Namespace:               test
CreationTimestamp:       Sun, 03 Sep 2023 17:27:05 +0200
```

```
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx
Replicas:               1 desired | 1 updated | 1 total | 0 available
↪  | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      False   MinimumReplicasUnavailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:  <none>
NewReplicaSet:   nginx-748c667d99 (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  15s   deployment-controller  Scaled up
  ↪  replica set nginx-748c667d99 to 1
```

Info: The command executed creates a `Deployment` named `nginx` where the related `Pods` execute a *Docker* container based on the `nginx` *Docker* image. The command `describe` prints a summary of the queried resource.

B Expose the `nginx` deployment through a `Service`:

Command Line

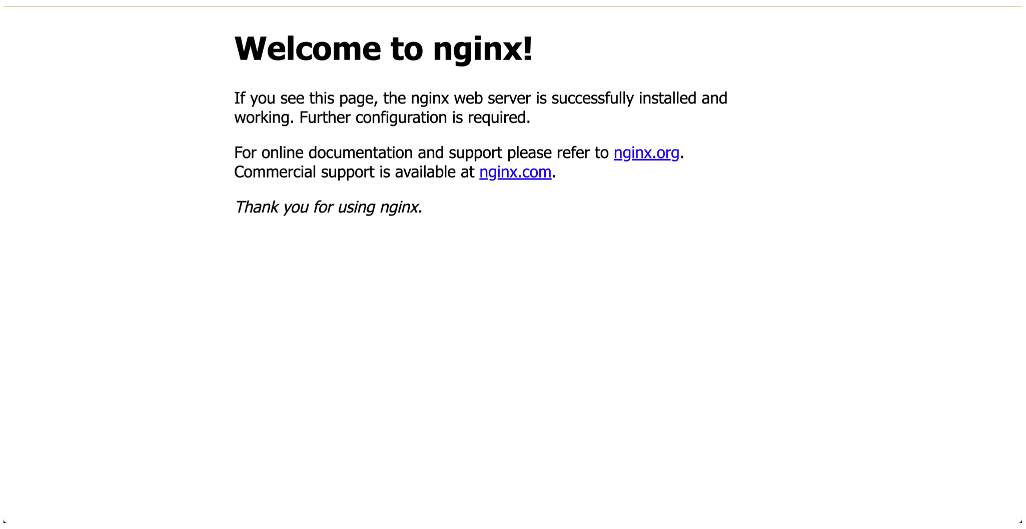
```
$ kubectl expose deployment/nginx --type="NodePort"
↪ --port 80
$ kubectl describe service/nginx
```

```
Name:                     nginx
Namespace:                test
Labels:                   app=nginx
Annotations:              <none>
Selector:                 app=nginx
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.108.98.225
IPs:                      10.108.98.225
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32063/TCP
Endpoints:                10.244.11.64:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

C Run a proxy to access the created service through an `HTTP` request:

Command Line

```
$ minikube service nginx
```



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 3.2: Nginx courtesy page

These are the main commands that allow a software developer to interact with a *Kubernetes* cluster to deploy a simple *Docker* container and to make the service it is running reachable from a development machine.

> **Info:** In this small tutorial the `nginx` *Docker* image, which was used for convenience, is a pre-built *Docker* image made available directly by *Minikube* since it is often used to run examples and tutorials. To use a custom *Docker* image, a user can decide to opt for a remote image hosted in an online *Docker* repository or build the image locally, with the `docker build` command. In this second case, it is necessary at first to tell *Docker* to use the `minikube`'s *Docker* environment, since otherwise the image will not be visible inside the `minikube` context. In this scenario, the instruction to run is `eval $(minikube -p minikube docker-env)`. Every image built after the execution of this command (considering the same shell session) will be built inside the `minikube` environment and will then become suitable to run inside *Kubernetes* pods.

Chapter 4

Design and Implementation of the infrastructure underlying the Zamperla Analytics application

4.1 System overview and application services

The *Zamperla* Web application, designed and implemented by Secura Factors (a spin-off of Ca' Foscari University of Venice), is a Web application based on microservices and written mostly in *Pyhton*. This Web application aims to collect data from the `PLC` nodes, store it in a database, and produce periodic reports. These `PLC` nodes collect data from physical sensors recording events on different rides inside an amusement park. The general idea is that a park has several rides; each ride communicates with a specific `PLC` node that collects some data about the ride it controls. Collected data is composed of a set of values associated with parameters like the time of a run, the time a ride has been in use, the number and type of eventual errors, etc. An access node within the perimeter of the park collects data coming from the different `PLC` nodes and makes them available to external services through a predefined protocol that can be a simple `SFTP` server or a more complex proprietary solution since the developed application allows for expansions based on the nature of the chosen protocol/system. Then a service running inside the application perimeter collects these data by downloading `.zip` files containing `.json` files that encode data coming from the different sensors. A service decodes the information contained in these `.json` files and saves it in a *PostgreSQL* database, which will then be used to process data and generate general reports. The application itself consists of 4 separate services: the *UI* component, the *BackEnd* component, the *DB* component, and the *Data* component.

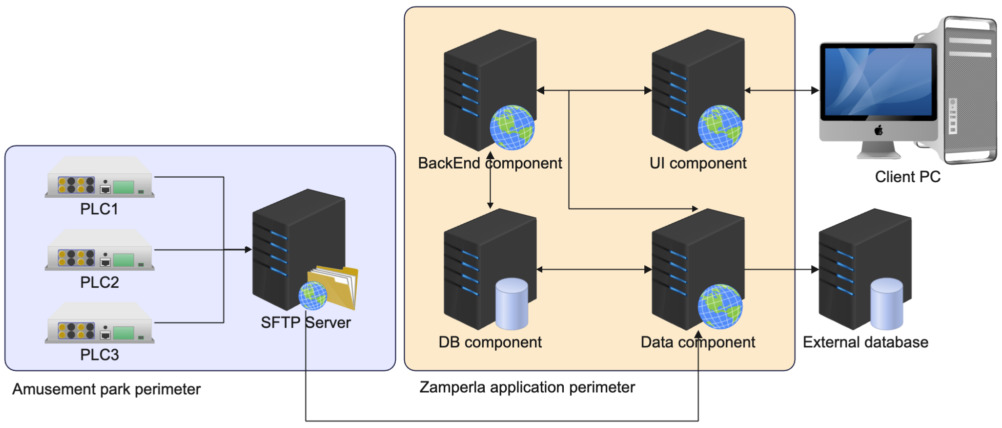
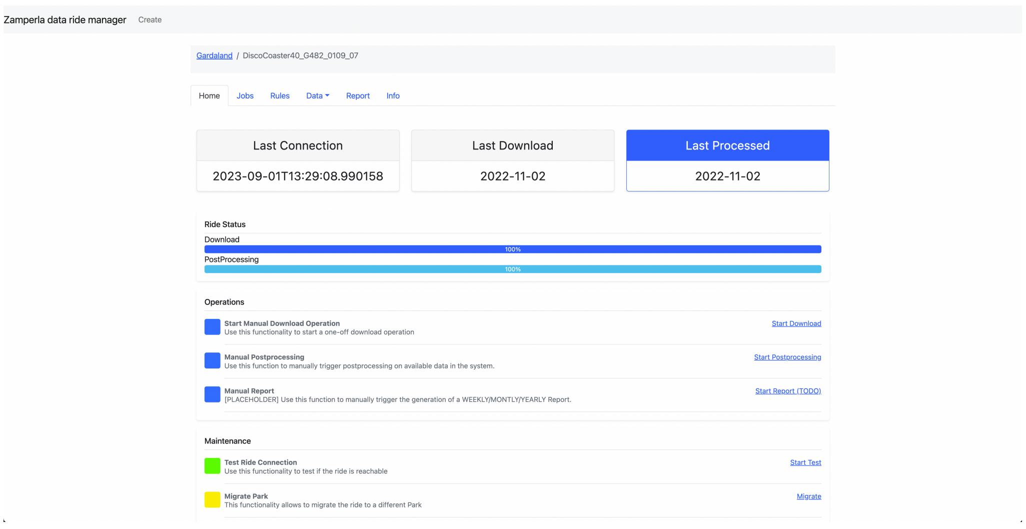


Figure 4.1: Graphic representation of the *Zamperla* infrastructure

4.1.1 *UI* component

The *UI* component is maybe the simplest service of the list. It consists of a *Flask* application written in *Python* that provides a graphic user interface that allows the end user to set up the general configuration of the system and add parks and rides, making it possible to set up scheduled `Jobs` or interact with the collected data in a simple and intuitive way. This component enables operations to be initiated through a browser event, such as submitting a form, but the service itself does not execute them. When an action is triggered, the service gathers the relevant information and sends it to the back-end service, which is responsible for managing it. It is useful to emphasize that the front-end software does not make use of any *JavaScript* framework nor of any additional technology like *Websockets* or *RTMP*, which makes it easier to maintain and provision the *Docker* container that will run it.


Zamperla data ride manager
Create
Gardaland / DiscoCoaster40_G482_0109_07
Home
Jobs
Rules
Data
Report
Info

Last Connection: 2023-09-01T13:29:08.990158

Last Download: 2022-11-02

Last Processed: 2022-11-02

Ride Status

Download 100%

PostProcessing 100%

Operations

Start Manual Download Operation
Use this functionality to start a one-off download operation
Start Download

Manual Postprocessing
Use this function to manually trigger postprocessing on available data in the system.
Start Postprocessing

Manual Report
[PLACEHOLDER] Use this function to manually trigger the generation of a WEEKLY/MONTLY/YEARLY Report.
Start Report (TODO)

Maintenance

Test Ride Connection
Use this functionality to test if the ride is reachable
Start Test

Migrate Park
This functionality allows to migrate the ride to a different Park
Migrate

Figure 4.2: The "ride" view of the graphic user interface made available by the *UI* service

4.1.2 *BackEnd* component

The *BackEnd* component is also very simple: its purpose is to receive requests from the *UI* component and forward them to the *DB* component if the requested operation is a simple CRUD operation, or to the *Data* service if the request concerns the scheduling and management of the data-intensive jobs.

4.1.3 *DB* component

The *DB* component is a service running an instance of `PostgreSQL` server. Figure 4.3 shows a graphic representation of the schema of the database. There are 8 relations:

- `Park`: the relation representing a physical park, with a certain geographic collocation and timezone.
- `Ride`: a ride, part of a `Park`. A `Park` can have many associated `Ride` records, but not vice versa.
- `Rules`: records of this relation represent a **execution rule**, that is, when a job should be started. It expresses relationships with both `Ride` and `Park`.
- `apscheduler_jobs`: a support relation containing auxiliary `APScheduler`-specific information about the execution of a job. `APScheduler`

is the *Python* library used to schedule routines.

- `Connection_mechanisms`: different ways a job can connect to a PLC gateway. Each `Ride` can have one or no connection mechanisms (the default is `SFTP`).
- `Settings`: a generic relation to store connection parameters and other types of configuration.
- `Downloader` and `Job_status`: auxiliary tables used to store the state of different jobs running. The *UI* service accesses this information to get the end user up-to-date on job status.

This database instance primarily interacts with the *BackEnd* service and the *Data* component both for readings and writings.

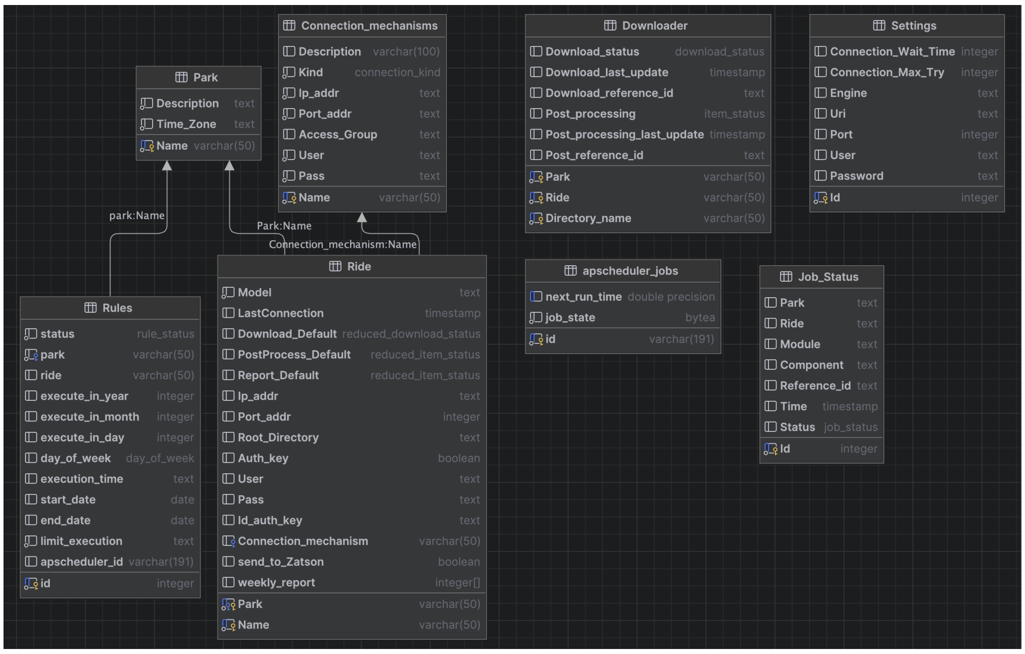


Figure 4.3: A graphic representation of the application database schema

4.1.4 *Data* component

This component is the main service responsible for the management of data coming from available rides. It includes all the logic required to interact with park gateways to retrieve data, as long as the algorithms needed to process these data and to generate reports consequently. This service is a

Flask application written in *Python*, which handles requests coming from the *Backend* service. However, it also incorporates a *Rust* software, developed by the Zamperla engineering team, which is responsible for the actual processing of data. In fact, *Rust* is a more efficient programming language than *Python*, and using this piece of software to run the post-processing tasks makes the application significantly faster. This service also makes use of *APScheduler*, a *Python* library that allows scheduling the execution of a job at a specific time: as an example, the download of the data collected by the sensors of a ride x can be scheduled to be executed once every week, at 9:00 AM on Mondays. However, this behavior is not very efficient because when the execution of different jobs overlaps, the application can run out of resources. This is one of the reasons for deciding to split this service into different execution units: one main unit responsible for handling the `HTTP` requests and for scheduling *jobs*, and 3 other units: one per job type (*download*, *postprocessing*, *reporting*). The actions taken to get these 4 different containers out of the initial one are described later in this thesis.

4.2 Basic service/deployment setup: the UI component

The *UI* component has been chosen as a starting point for the implementation of the *Kubernetes* infrastructure underlying the subject application for two main reasons: the first is that it is a relatively simple service consisting of just one *Flask* application serving `HTTP` requests. The second is that it provides a user interface, so it is simpler to check that the container is working properly and also provides a way to interact with the other services, making it trivial to test the system's functioning. The general idea here is to start with a simple `Service/Deployment` setup: a *Kubernetes* `Deployment` will spawn a predefined number of *Kubernetes* `Pods`, each of which will run an instance of the *UI* service. A *Kubernetes Service* will serve requests directed to these instances, providing an entry point for `HTTP` requests.

4.2.1 Setting up a `deployment`

As seen in chapter 2, a `Deployment` is an object of the *Kubernetes* domain that manages the execution and the updates of a set of `Pods`, which run containers based on the same *Docker* image. In *Kubernetes* objects can be managed thanks to the `.yaml` files, which represent the specifications of the state that the object should reflect. These configuration files follow strict conventions described in the *Kubernetes* reference document. The configuration file of the majority of resources that can be managed through the *Kubernetes* API contains at least 4 top-level properties:

- `[string] apiVersion`: expresses the version of the *Kubernetes* API that must be used to update the resource accordingly with the syntax adopted in the configuration file. As will be addressed later in this thesis, *Kubernetes* APIs can be extended, so it is mandatory to express which version should be used to guarantee proper management of the managed resource.
- `[string] kind`: specifies the kind of resource to be managed.
- `[object] metadata`: a composed field that contains context information about the resource that will be managed.
- `[object] spec`: the specification of the resource to be managed represents the state that the resource should obtain and maintain, as long as the cluster physical resources allow it. Its inner fields and information depend on the `kind` of resource to be managed.

The file `_k8s/zamperla-app/UI/deployment.yml` contains the `YAML` representation of the configuration of the *UI* service deployment. Here there are a few points to underline: the resource that is managed by this piece of `YAML` code is of type `Deployment`, named `ui-deployment` and created in the `default` namespace. *Kubernetes* allows the creation of different namespaces to supply an additional layer of isolation but, since there will be only one main application running in the cluster, it is useless to create a reserved namespace for it. It would only make `kubectl` commands more verbose since custom namespaces need to be specified in many cases. The usage of namespaces will make sense later in this thesis when supporting resources not directly related to the main application operation will be required (for instance, in chapter 5 a different namespace will be used to host the *KEDA* supporting components). The resource will be labeled with meta tags `[app="zamperla-app"]` and `[tier="ui"]`. Meta tags are useful for identifying resource associations. The deployment will have the following specifications: it will manage a single `Pod`, and the `Pod` that will be managed by this resource will be labeled with the same metadata of the deployment (as the property `spec.template.metadata.labels` specifies). Now, if the cluster already has a `Pod` labeled as this `deployment` expects, it would be sufficient to apply the configuration file to make the whole system work. In many cases, the properties of the `Pods` managed by the top-level resource are not predetermined. Therefore, the `spec.template` property provides the ability to define the characteristics that `Pods` should have.

_k8s/zamperla-app/UI/deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ui-deployment
  namespace: default
  labels:
    app: zamperla-app
    tier: ui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zamperla-app
      tier: ui
  template:
    metadata:
      labels:
        app: zamperla-app
        tier: ui
    spec:
      containers:
      - name: ui
        image: zamperla-fastapi/ui
        imagePullPolicy: Never
        resources:
          limits:
            cpu: 500m
            memory: 500Mi
        ports:
        - containerPort: 5000
```

In this case, `Pods` should be labeled with the values contained in the `spec.template.metadata.labels` property and should execute a container named `ui` based on the `zamperla-fastapi/ui` *Docker* image, having resources limited to 500 Mebibytes of RAM and the computational power of 500 millicpus. Also, the pods will expose port `5000` to the cluster, to make such a port available to services. A noticeable fact about this configuration is the value of the property `spec.template.spec.containers.imagePu` llPolicy: here its value is set to Never only for development purposes since the *Docker* image zamperla-fastapi/ui has been built only in the local *Docker* repository, and therefore it should not be pulled from a remote repository.

ℹ **Info:** The term **millicpu** is a name used to express a unit of measure for computational power. It represents a thousandth of the computational power of 1 standard CPU. Its symbol is m and it is used in Kubernetes specification files when there is the need to bind the CPU resource that a pod can (or should) request inside a cluster.

Opening a terminal prompt on the project directory, to apply this configuration file it suffices to launch the following command-line command:

Command Line

```
$ kubectl apply -f _k8s/zamperla-app/UI/deployment.yml
```

It is a best practice to query the *Kubernetes* API to retrieve the status of the newly created resource. The following command will accomplish this purpose:

Command Line

```
$ kubectl get deployment ui-deployment
```

```
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
ui-deployment   1/1     1            1           30s
```

4.2.2 Setting up a Service

To access the applications hosted in each Pod, a Service object must be created. This *Kubernetes* object provides a stable, discoverable endpoint for accessing a group of Pods, and offers load balancing, service discovery, and flexibility in how the application is exposed both internally and externally within the cluster. The configuration file of the Service object associated with the *UI* component has the following content:

_k8s/zamperla-app/UI/service.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ui-service
spec:
  selector:
    app: zamperla-app
    tier: ui
  type: NodePort
  ports:
  - port: 80
    nodePort: 30080
    targetPort: 5000
```

Probably the most interesting part of this configuration file is related to the `spec.type` property. This property defines the behavior that the `Service` should have. The *Kubernetes* API provides four different types of `Services`: `NodePort`, `ClusterIP`, `LoadBalancer`, and `ExternalName`. `ClusterIp` is used when the `Service` should only be accessible from other entities within the cluster. `LoadBalancer` allows the `Service` to take advantage of an external *Load Balancer* service, which is mainly used when the cluster is hosted in a cloud infrastructure such as *Amazon Web Services* or *Google Cloud*, which provides their solutions for this type of product. `ExternalName` is used if the application hosted in the `Pods` linked to the service has DNS software that can make the application respond to a CNAME entry. `NodePort` is suitable for most cases, as it makes the Web service reachable from the IP address of any of the cluster nodes on a predefined port. This is the behavior that best meets the needs of the UI component. Within a `Service` of type `NodePort`, a port mapping must be explicitly defined. In this case, the "source" port, which is the port exposed by the `Pods` in execution, is `5000`, and the port exposed by the `Service` is port `80`, which maps with port `30080` on each node of the cluster. In conclusion, the *UI* component will be available on every cluster node's IP at port `30080`. This would be true in a production cluster, but in a development environment cluster running on *Minikube*, it is also necessary to tell *Minikube* to expose the service to the outside world. The command that will accomplish this task, finally making the *UI* component interactive through a Web browser, is the following:

Command Line

```
$ minikube service ui-service
```

```
|-----------|------------|-------------|----------------------------|
| NAMESPACE |    NAME    | TARGET PORT |            URL             |
|-----------|------------|-------------|----------------------------|
| default   | ui-service |          80 | http://192.168.49.2:30080  |
|-----------|------------|-------------|----------------------------|
 Starting tunnel for service ui-service.
|-----------|------------|-------------|------------------------|
| NAMESPACE |    NAME    | TARGET PORT |          URL           |
|-----------|------------|-------------|------------------------|
| default   | ui-service |             | http://127.0.0.1:63665 |
|-----------|------------|-------------|------------------------|
 Opening service default/ui-service in default browser...
```

At this point, a browser tab should open showing the user interface of the *UI* service. Note also that *Minikube* does not expose the service directly on the node IP, but starts a tunnel to `127.0.0.1`, mapping the node exposed port to an arbitrary port of `localhost`. This is due to an internal implementation of *Minikube* that allows the product to maintain a coherent behavior across different operating systems.

4.3 Adding persistent storage: the DB component

The *DB* component is a simple `PostgreSQL` instance, but configuring this kind of service to run inside a *Kubernetes* instance hides some difficulties. First, let us look at its `Deployment` and `Service` configuration files. For this specific resource, file `_k8s/zamperla-app/DB/deployment.yml` represents the content of the manifest file of the `Deployment` related to the resource that is going to be created. The latter is indeed very similar to the one used to set up the *UI* component, but here there are a few criteria to underline.

1. In this case, having just one replica is mandatory since having multiple replicas would require a multi-instance database setup, with all the complexity a redundant database system requires, related in particular to the data synchronization problem.
2. The image being run inside the `Pods` related to this `Deployment` is no longer a locally built image, but the official `PostgreSQL` image,

retrieved from a default remote *Docker* repository. The `spec.template.spec.containers.imagePullPolicy` property is consequently set to `"IfNotPresent"` since the image could not be present in the local machine the first time.

3. The `containerPort` value is `5432`, the default port for a `PostgreSQL` service.

_k8s/zamperla-app/DB/deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-deployment
  # ...
spec:
  replicas: 1
  # ...
  template:
    # ...
    spec:
      containers:
        - name: db
          image: postgres
          imagePullPolicy: "IfNotPresent"
          ports:
            - containerPort: 5432
          # ...
```

Nevertheless, file `_k8s/zamperla-app/DB/service.yml` represents the content of the manifest file of the related `Service`. Similarly to what happened for the `Deployment` manifest, the `Service` specifications are quite similar to those of the *UI* service; but instead of a `Service` of type `NodePort`, a `ClusterIp Service` is used, since it is not necessary (nor convenient) to make the `Pods` reachable from outside the cluster. The challenging part of this configuration, however, is still missing, and it is the setup of *persistent storage* able to persist database information even if the `Pod` running the *DBMS* instance dies or restarts.

_k8s/zamperla-app/DB/service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: db-service
  namespace: default
  labels:
    app: zamperla-app
    tier: db
spec:
  type: ClusterIp
  ports:
    - port: 5432
  selector:
    app: zamperla-app
    tier: db
```

4.3.1 Requesting storage: `PersistentVolume (PV)`

A `Pod` running a *Docker* container can die for different reasons: the application in execution inside the container fails, the cluster runs out of resources, etc. When a `Pod` dies, all data inside the container that are not included in the build of the *Docker* image are lost. This is a problem when there is a need, for example in *DBMS* services, to persist additional data. The same problem happens also when a container is executed in a `docker compose` environment, or as a standalone *Docker* running instance: in these cases, the problem is easily solvable by attaching a *volume* to the running container. This way every data that will be saved inside the directory in which the volume is mounted will live regardless of the state of the container. In *Kubernetes* the problem can be solved using **Persistent Volumes**. Like the official documentation says:

> A *PersistentVolume (PV)* is a piece of storage in the cluster that has been provisioned by an administrator [...]. It is a resource in the cluster just like a node is a cluster resource.[13]

In practice, a `PersistentVolume` is a reserved piece of storage that lies at a predefined path in the host node and will store the data from `Pods` that use it. Here is the configuration file of the `PersistentVolume` used to store data related to the PostgreSQL instance running within the *DB* component:

```
_k8s/zamperla-app/DB/persistent-volume.yml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pg-volume
  namespace: default
  labels:
    app: zamperla-app
    tier: db
spec:
  storageClassName: manual
  capacity:
    storage: 3Gi
  accessModes: [ReadWriteMany]
  hostPath:
    path: "/data/zamperla-app/db-volume"
```

Most of the properties defined in this file are obvious. The key points of this configuration are:

- spec.storageClassName: defines the provisioning policy. In this case, it is a local storage provisioned by the system administrator. Other available provisioners, depending on the nature of the cluster, could be VsphereVolume, AzureFile, and so on.
- spec.capacity.storage: defines the storage size requirement. In this case, it has been set to 3Gi. This is not sufficient for production environments, but it is more than appropriate for development purposes.
- spec.accessModes: defines a list of modes that Pods can use to access this resource. In this case, it is set to ReadWriteMany, which means that every Pod that requests it can read and write in this reserved store location.
- spec.hostPath.path: defines the location inside the host node where data will be managed. For example, if a Pod writes something in the PersistentVolume created from this configuration, such data will be available at the host node at this location. If there are more nodes in

a cluster, during the provisioning process it is necessary to specify in which node the storage should be allocated.

Now it is possible to create the `PersistentVolume` resource where database data will be saved:

Command Line

```
$ kubectl apply -f
↪  _k8s/zamperla-app/DB/persistent-volume.yml
$ kubectl get pv pg-volume
```

```
NAME      CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM ...
pg-volume 3Gi      RWX          Retain         Bound        ...
```

The `kubectl get` command shows that the `PersistentVolume` has correctly been initialized and bound to the node resource allocation. Also, the `RECLAIM: Retain` piece of information indicates that the data are not erased when the `PersistentVolume` resource is destroyed: launching the command

Command Line

```
kubectl delete -f
↪  \_k8s/zamperla-app/DB/persistent-volume.yml}
```

will delete the *Kubernetes* resource, but not the data that were previously stored at the node location associated with the `PV`. This is the default behavior applied when no other *retain policy* is specified.

4.3.2 Using storage: `PersistentVolumeClaims (PVC)`

However, to allocate storage resources with `PersistentVolumes` is not sufficient to allow a `Pod` to write to a persistent storage location. In fact, a `PersistentVolumeClaim` is required. According to the official *Kubernetes* website:

> A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory).[13]

The content of the configuration file for the *DB* `PersistentVolumeClaim` follows.

_k8s/zamperla-app/DB/persistent-volume-claim.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pg-volume-claim
  namespace: default
  labels:
    app: zamperla-app
    tier: db
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 3Gi
```

As can be seen, most of the configuration simply doubles the properties of the `PersistentVolume` that is going to be used. Note on the association between a `PV` and a `PVC`: when the `PVC` is created, a *control loop* in the *control plane* searches for compatible `PVs` that can be attached. When a suitable `PV` is found, the latter is exclusively bound to the requesting `PVC`. The match happens when a `PV` with the same characteristics of - and **at least** the amount of available memory requested by - the requesting `PVC` is found. Otherwise, the `PVC` creation throws an error. Finally, a `Pod` should request to use the `PVC` created adding some properties to its configuration file. Note that the same `PVC` can be used in different deployment configurations, as it provides a sort of "*shared entry point*" to a specific store location in node storage. Referring to file `_k8s/zamperla-app/DB/deployment.yml`, the following changes occur.

_k8s/zamperla-app/DB/deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  # omitted
spec:
  #omitted
  template:
    spec:
      containers:
        # ...
        volumeMounts:
          - mountPath: /var/lib/postgresql/data
            name: dbdata
      volumes:
        - name: dbdata
          persistentVolumeClaim:
            claimName: pg-volume-claim
```

The added configurations express the requested `PVC` and the container path at which the volume bounded to the `PVC` will be mounted. The result now is that every information that the *DB* component will write to `/var/lib/postgresql/data` will be permanently saved to the `/data/zamperla-app/db-volume` location of the hosting node, and vice versa everything written by third party actors to the node volume location will become available inside the *DB* running container.

4.3.3 `Pod` environment variables and database initialization: `ConfigMaps`

The *DB* service utilizes a standard `PostgreSQL` image to operate correctly, which necessitates some initial configurations. Specifically, two configurations are required. The first one is the setup of predefined environment variables, which will store information regarding the database user, the *database* name, and the user password. The second one is the running of an initialization script, which will create the appropriate database schema. To perform these tasks, `ConfigMaps` will be used. Referring to the already cited *Kubernetes* documentation:

> A ConfigMap is an API object that lets you store configurations for other objects to use. Unlike most *Kubernetes* objects that have a spec, a ConfigMap has `data` and `binaryData` fields.[14]

The first `ConfigMap`, used to set environment variables inside the `PostgreSQL` *Docker* container, has the following content:

_k8s/zamperla-app/DB/secrets.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-secret
  namespace: default
  labels:
    app: zamperla-app
    tier: db
data:
  POSTGRES_DB: backend
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: admin
  PGDATA: /var/lib/postgresql/data
```

To use this *ConfigMap* content inside a container, the following directives should be added to the container's spec:

```yaml
envFrom:
  - configMapRef:
      name: db-secret
```

This way, the environment variables defined in the `ConfigMap` will be available inside the running *Docker* container. A note about environment variables that contain secrets: it is not a good idea to version `ConfigMap` configuration files with a versioning tool like `git` if they contain sensitive information, such as passwords or SSH keys. In these cases, it should be better to create `ConfigMaps` directly on the production nodes or to use different tools for secret obfuscation. The *Kubernetes* API, for example, provides the `Secret` API object to handle the storage of secret information, but since the original `git` repository had secrets versioned in clear, being an academic project more than a production-ready application, secrets have been threatened accordingly. However, this is an important improvement that would be great to develop with future revisions.

A ConfigMap has also been used to contain the SQL code to execute during the database initialization phase. In this case, the data property of the init-db.yml configuration file had only one key, i.e., initdb.sql. Its value is a multiline string representing the content of the SQL initialization file. Unlike a ConfigMap used as an environment variable store, however, this resource needs to be mounted similarly to a PVC. The configuration file for the *DB* Deployment needs to be changed in the following way:

```
_k8s/zamperla-app/DB/deployment.yml
spec:
  template:
    spec:
      containers:
        # ...
        volumeMounts:
          # ...
          - mountPath: /docker-entrypoint-initdb.d
            name: db-initdb
      volumes:
        # ...
        - name: db-initdb
          configMap:
            name: db-initdb-config
```

This configuration will mount a file named initdb.sql, with the content specified in the ConfigMap configuration file, inside the /docker-entrypoint-initdb.d directory. By default, the first time that the PostgreSQL container starts, it will run every SQL script inside this folder to initialize the database instance. The changes made by the initialization script to the database will be stored inside the /var/lib/postgresql/data container directory, which is persistent due to the bounded PV. Now the container is ready to run the PostgreSQL server and start accepting connections.

4.4 Summing up: BackEnd component and Data component

Most of the *Kubernetes* features used for this project have been presented during the previous chapters. The setup of the remaining components mainly consists in configuring those features properly, to get the desired behavior out of each component execution. The *BackEnd* component setup, in particular, is nothing more than what has already been presented with the *UI* component and the *DB* component: it consists of a `Flask` application that serves `HTTP` API requests and, depending on the nature of the request, forwards it to the database, to retrieve data, or to the *Data* component, when the purpose is the management of scheduled jobs. Its configuration consists of 3 files: `deployment.yml` for the `Deployment` specifications, `service.yml` for the `Service` configuration, and *configs.yml*, which contains the configuration of a `ConfigMap` aimed at setting the environment variables needed by the running container (mainly the connection credentials for the database connection and the *URL* of the *Data* service).

4.4.1 The *Data* component configuration

Differently from the *BackEnd* component, even if there are no new *Kubernetes* features that come into action, the configuration of the `Data` component is quite more complex. As already mentioned, it consists of a *Flask* application with a *Rust* executable responsible for the post-processing phase of the data threat process. This component was initially set up as a service running a single instance of its *Docker* image, but it has evolved into a more complex system, which will be described in depth in the next chapter. For now, it is important to note that this service uses many `PVs` and `PVCs`, to keep the generated files tidy. To give an idea of the volumes that need to be mounted, an extract of the `deployment` configuration file follows.

_k8s/zamperla-app/Data/deployment.yml (volumeMounts only)

```yaml
volumeMounts:
  - mountPath: /app/config.cfg
    name: data-config-volume
    subPath: config.cfg
  - mountPath: /app/logs
    name: data-logs-volume
  - mountPath: /app/z_acq
    name: data-z-acq-volume
  - mountPath: /app/zamperla_data
    name: data-zamperla-data-volume
  - mountPath: /app/Storage
    name: data-storage-volume
```

4.4.2 Project directories structure

From a folder structure perspective, each component has a separate directory where the configuration files are located. All component configuration folders are contained in an application root folder called `zamperla-app`, which contains configurations strictly related to the provisioning of the application components. Some configurations do not change the state of the objects that relate to the application services, like those required for the management of the observability infrastructure: those files will have a separate root folder at the same level as the `zamperla-app` folder. It is also useful to note that all application-related configurations can be applied at the same time with a single command, which is the following:

Command Line

```
$ kubectl apply -Rf _k8s/zamperla-app
```

When running recursive commands like this, it is important to pay attention to the order of execution. `kubectl` follows an alphabetical order, which can cause errors. For instance, `PVs` must be created before `PVCs`, and `PVCs` must be created before `Deployments` that require them.

```
_k8s
└ zamperla-app
   ├ BackEnd
   │  ├ configs.yml
   │  ├ deployment.yml
   │  └ service.yml
   ├ Data
   │  ├ configs.yml
   │  ├ deployment.yml
   │  ├ persistent-volume-claim.yml
   │  ├ persistent-volume.yml
   │  └ service.yml
   ├ DB
   │  ├ deployment.yml
   │  ├ init-db.yml
   │  ├ persistent-volume-claim.yml
   │  ├ persistent-volume.yml
   │  ├ secrets.yml
   │  └ service.yml
   └ UI
      ├ configs.yml
      ├ deployment.yml
      └ service.yml
```

Chapter 5

Scaling jobs with RabbitMQ and Keda

5.1 Scaling data-intensive jobs: an application need

As already mentioned, the *Data* application component is responsible for the download of sensor data, the post-processing elaboration of those data, and the generation of reports containing useful insights about the performance of the various rides. To achieve these goals, this service uses a *Python* library called *APScheduler*, which helps to schedule the execution of each job at a user-defined time. However, different jobs may be executed at the same time, and this represents a problem in terms of allocated resources. A system that can execute an arbitrary amount of work should be able to scale consequently, but this is not the case for the *Data* component, at least as it has been implemented to this point. The proposal, then, is that of splitting the component into four parts: a main part acting like an "*orchestrator*", which collects the `HTTP` requests and acts by consequence, and three job executors, which will handle one specific type of job at a time. Furthermore, the system should be able to execute an arbitrary (with the limits given by the physical resources) number of concurrent instances for each type of job, allowing for a higher degree of concurrency and, consequently, for a faster response to data operations requests. Designing a system capable of this type of behavior is not trivial: many actors should be taken into account and *Kubernetes* does not provide a standard method to scale `Pod` instances concerning custom metrics like the number of jobs that should be executed concurrently. The solution proposed with this project makes use of a *message queue*, of some `Job` *Kubernetes* objects, and of an *event-driven autoscaler* called *Keda*.

5.1.1 Message queues: how *RabbitMQ* helped to manage complexity

According to the official *AWS* documentation:

> A message queue is a form of asynchronous service-to-service communication used in serverless and microservices architectures. [15].

In practice, when working with message queues, a *Pub/Sub* mechanism comes into action, where *Pub/Sub* stands for "Publisher/Subscriber": a *publisher* publishes a message related to a certain topic on the message queue, a *subscriber* subscribes to this topic waiting for messages to arrive. When a message arrives, the subscriber *consumes* the message by reading it and performing any task that this message may require. If the task ends without errors, the message is deleted from the queue; otherwise, it is left on the top of the queue. Message queues follow a *FIFO* (*First In / First Out*) policy, so messages that arrive earlier are processed earlier. This project uses *RabbitMQ* as a message broker. Another widely used solution that has been taken into account is *Apache Kafka*, which is a well-known technology used by several tech companies. *Apache Kafka* is capable of handling more concurrent messages than *RabbitMQ*: Kafka can handle an order or millions of messages exchanged per second, while *RabbitMQ* needs more brokers to achieve this goal[16]. However, while *Kafka* uses a proprietary protocol to interact with consumers and producers, *RabbitMQ* uses the standard `AMQP` protocol to receive and publish messages, which makes things easier when it comes to integrating a message queue with a *Kubernetes* cluster. *RabbitMQ* is also capable of guaranteeing the quality of the arrival order of messages, and, differently from *Kafka*, a *RabbitMQ* consumer is aware of the status of a message (consumed/not consumed).[16] In conclusion, even if *Kafka* is faster, log-centric, and capable of handling a higher degree of parallelism, *RabbitMQ* provides more consistency guarantees out of the box. These are the main reasons for the decision to adopt *RabbitMQ* as a message broker instead of *Apache Kafka*.

RabbitMQ can be installed in a *Kubernetes* cluster like any other component configured during the development of this project: the *deployment* is similar to the one used for the *DB* component, but it does not use any external configuration or volume, and its `Pod` runs the official `rabbitmq` *Docker* image, exposing port 5672. The Service is essential and simply exposes port 5672 of the running container to the other Pods in execution inside the perimeter of the cluster.

5.1.2 Jobs, i.e. *Kubernetes* disposable Pods

To better understand how the definitive system will be built up and how the different actors of such a system will communicate, the concept of Job as a *Kubernetes* API object must be introduced. According to the - many times - already mentioned official *Kubernetes* documentation:

> A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As Pods complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e., Job) is complete. [17]

In other words, Jobs allow to start Pods that run containers having a single task to accomplish. When the task ends successfully, the host Pod dies. A Job configuration file is similar to a Deployment YAML:

example Job configuration file

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: csvtojson
spec:
  template:
    spec:
      containers:
      - name: csvtojson
        image: python:3
        command: ["python", "-c", "import csv,json;print
↪   json.dumps(list(csv.reader(open('csv_file.csv'))))",
↪   ">", "result.json"]
      restartPolicy: Never
```

The configuration file listed below creates a Job that converts a csv file named "csv_file.csv" into a json file named "result.json". Similarly, Jobs will be used in the context of this project to run tasks that can start and end autonomously, interacting only with the *message queue*.

5.1.3 *Keda*: autoscaling Jobs

Let us suppose that the infrastructure that is going to be shaped consists of:

1. A *message queue* based on *RabbitMQ*, which exposes a queue topic for each type of job that is going to be created
2. A *Data* component without the data management tasks, i.e. a Web service that serves requests coming from other services. The purpose of this unit is to schedule jobs concerning incoming requests. The jobs are scheduled with the help of *APScheduler*, but are not executed within this component. Instead, when the scheduler triggers an execution, a message is enqueued in the message queue related to the requested type of job.
3. For each type of task, a *Kubernetes* `Job` that will read a message from the queue, execute the relevant script, and, if necessary, add a message to the queue of the next type of task that should be run in the job sequence.

This is the main flow that should be executed every time a job is scheduled. But what happens if a `Job` of a certain type is requested but there is an old one still executing? The second job will have to wait until the first job is completed successfully. In this way, if the scheduler triggers many jobs, the system can enter a state of *Congestion Collapse*. This situation can be prevented, or at least its impact can be minimized, by utilizing an *autoscaler* to create a new `Pod` each time a message is received in a predetermined queue. For this project, *Keda* will be used as autoscaler software. The *Keda* official website reports:

> KEDA is a *Kubernetes*-based Event Driven Autoscaler. With KEDA, you can drive the scaling of any container in *Kubernetes* based on the number of events needing to be processed.[18]

Keda interacts with the low-level *Kubernetes* API to spawn new `Pods` every time a message comes to a queue. *Keda* APIs offer a straightforward approach to establishing a scaling system in which the metrics used to determine when and how to scale `Pods` can be based on the number and type of messages sent to a configurable message queue. Additionally, *Keda* is compatible with a variety of *Pub/Sub* and message queue technologies, such as *Apache Kafka*, *Redis*, and, fortunately, *RabbitMQ*.

5.2 Jobs split-up: `reporter`, `postprocessor` & `sftp_downloader`

This section describes how the initial *Data* component has been split into three different processing units: the `sftp_downloader` job, responsible for downloading data from the remote resources; the `postprocessor` job, which provides post elaboration and processing for the downloader data; the `reporter` job, which generates a summary report based on the processed information. The tasks division process took 4 different steps to complete:

1. Development of the startup *Python* script that will run in the *Docker* container related to each particular job.
2. Definition of a new *Docker* image to run for each type of job; the image used to execute a specific type of job should only contain, when possible, code and components strictly related to the process to run.
3. Definition and creation of the *Kubernetes* `Job` object responsible for the execution of the task.
4. Substitution of direct calls to task methods into the service code with the insertion of messages into the appropriate queue.

A complete description of these steps follows.

5.2.1 Step 1: startup script

Supposing that a task will run inside a *Docker* container and that this container will need a Python script to be defined as the *entrypoint* in the *Docker* image, 3 new Python scripts are needed: one for the information download process, one for the execution of the data postprocessing phase and one for the creation of the weekly report. Even if there are already some utility functions able to accomplish these tasks, such methods can not run as standalone executables and therefore need a wrapper script capable of:

1. reading messages sent to a given *RabbitMQ* queue
2. using the content of this message to get the information needed to carry out the designed task
3. executing the proper utility functions to obtain the expected result
4. sending a message to the *RabbitMQ* queue responsible for the management of the successive type of task that needs to be executed, if

necessary, encoding all the useful information that the next script will require

Taking the `downloader` job as an example, the utility function that needs to be called is `sftp_downloader`, which is a method contained in the helper file `Data/helpers/downloader/sftp.py`. Let us assume that the job will listen to the *RabbitMQ* queue due to the execution of `/usr/bin/amqp-consume`, an executable file shipped with the default installation of the *RabbitMQ* file. Executing this program with the right arguments allows the developer to call a *Python* script when a message is retrieved from the queue, passing the content of the message to the script as a command-line argument. The first point of the "*TODO*" list is then performed by a third-party actor: the script can ignore this part and assume that the command-line argument with which it is called contains the `rule_id` of the bounded rule that is going to be executed. The second point of this list, reading the information needed to properly run the task, can be easily accomplished: since messages are strings, the content of a message can be safely considered as a well-formatted `JSON` containing the id of the rule to execute (as well as a few other pieces of information that will be faced later in this chapter). Once the rule identifier has been obtained from the source, the script can acquire the context information of the park and the ride it needs to initiate the download for, by querying the `Rules` database relation. After that, the script is ready to call `sftp_downloader(...)`. Once the execution of this method ends, it is necessary to check if there are elements suitable for the post-processing phase, i.e. the configuration of the ride does not specify to skip the data post-process task and there are elements that have not been already post-processed. If these conditions apply, a message is sent to the `postprocessor` *RabbitMQ* queue, which will trigger the execution of the postprocessing procedure for the available data. The other jobs will have a similar approach. Both the `postprocessor` job and the `reporter` job will read the information they need from the command line argument that will be passed to the script by the `amqp-consume` executable. Both scripts will interpret this information as a well-formed `JSON` string, extracting from it the information they need to execute their main utility method and execute context operations like logging and updating the job status. A difference between the `reporter job.py` script and the others is that at the end of the execution of its main utility method, there will be no *RabbitMQ* queue to notify since it is the last phase of the data processing tasks chain. Back to the `downloader` task, taken as a general example, let us take a closer look at what such `job.py` *Python* script looks like.

Data/jobs/sftp_downloader/job.py

```
#!/usr/bin/env python
# various imports...
from helpers.downloader.sftp import sftp_downloader
#...

def notify_queue(data):
    # using Pika python library to send a new message to
    ↪  the post-process queue

def job(rule_id, ...):
    try:
        rule = get_rule(rule_id)
        ride = get_ride_info(rule.ride, rule.park)
        # ...
        sftp_downloader( ... )
        elem_to_process = get_to_process(rule.ride,
        ↪  rule.park)
        # check if there are elements to postprocess in
        ↪  the list elem_to_process
        if len(elem_to_process) != 0 and
        ↪  ride.PostProcess_Default != 'skip':
            start_date = min(elem_to_process)
            finish_date = max(elem_to_process)
            notify_queue({
                'operation_id': operation_id,
                'rule_id': rule_id,
                'start_date': start_date,
                'finish_date': finish_date,
                ...
            })
    # errors handling, job status update and logging

if __name__ == '__main__':
    args = json.loads(sys.stdin.readlines()[0])
    rule_id = args['rule_id']
    # ...
    job(rule_id, ...)
```

It is useful to know that these scripts will all be called invoking the following command, which is going to be used as the entrypoint of the *Docker* images that will run these processes:

Command Line

```
/usr/bin/amqp-consume --url=$BROKER_URL -q $QUEUE -c 1
↪  ./job.py
```

This command listens for events sent to the message queue named by `$QUEUE` and reachable at the URL given by `$BROKER_URL`. When a message arrives, `job.py` is called with the content of the message passed as a command-line argument.

5.2.2 Step 2: *Docker* image

The next step consists of building a *Docker* image for each different type of task. This *Docker* image should be able to execute its related *Python* script requiring as little as possible of the initial *Data* service code. To do this, it is necessary to analyze how the original *Data Docker* image was made. Since the original instance of the *Data*, service was responsible for everything related to the data manipulation tasks, from the download to the report generation, the initial *Data* component image included all code present within its folder. It was also responsible for building and executing the *Rust* code deputy to post-processing elaboration. Most of this code is no longer required, since the whole part concerning data manipulation is delegated to the single job *Docker* image. The *Data* service now has three main responsibilities:

1. collecting `HTTP` requests coming from the *BackEnd* component, which will contain the instructions to set up the job scheduling
2. scheduling jobs, with the help of *APScheduler*
3. sending the right data payload to the right queue when a job is triggered by the scheduler

These actions do not require the *Rust* software integration code, nor the different helper functions delegated to the download, postprocess, and report creation tasks.

The starting point of the development of each job's *Docker* image is the file below, which represents the original content of the *Data* component *Dockerfile*.

```
Data/Dockerfile.production
# STAGE 1: Build Rust app
FROM rust:1.68.2-bullseye
WORKDIR /Rustapp
COPY Rust .
RUN rustup install nightly
RUN cargo +nightly build --release

# STAGE 2: Build python service
FROM python:3.10.9-bullseye
WORKDIR /app
RUN apt-get -y update && apt-get install -yqq unzip
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY --from=0 ./Rustapp/target/release/..._TZ_004* .
COPY privateKey.pem .
COPY Report/zamperlacertificate.crt .
COPY ./*.py .
COPY helpers ./helpers
COPY Report/helpers ./helpers/Report/
COPY Rust/settings ./settings
COPY Report/AlarmTables ./AlarmTables

ENTRYPOINT ["uvicorn", "main:app", "--host", "0.0.0.0",
↪  "--port", "8081"]
```

This *Dockerfile* uses a feature called *Multi-stage Build*, which allows the construction of complex Dockerfiles capable of building *Docker* images that contain artifacts from different environments without making them too large in size and number of layers. The build process of the *Docker* image resulting from this Dockerfile produces a set of 2 stages:

1. the first FROM instruction defines the beginning of the first stage, responsible for the production of the executable file resulting from the

compilation and build of the *Rust* code contained in the Data/Rust folder.

2. The second FROM instruction represents the starting point of the second stage, which sets up the suited environment for the *Python* microservice execution.

The second stage starts with an official *Python* image and, at first, installs all pip project dependencies. Then it copies the release file obtained during the build process of the first stage to its root folder, obtaining the ability to run it without weighing down the final image release with all the *Rust* code and compilation/build products. Finally, the build process copies inside the image all the *Python* code available in the folder.

The image of the *Downloader* job does not need all these features. It is quite easy:

Data/jobs/sftp_downloader/Dockerfile.production

```
FROM python:3.10.9-bullseye
ENV BROKER_URL="amqp://guest:guest@rabbitmq-service:5672"
ENV QUEUE="sftp-download"

WORKDIR /app
RUN apt-get update && apt-get install -y curl
↪  ca-certificates amqp-tools unzip
↪ --no-install-recommends && rm -rf /var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY privateKey.pem .
COPY helpers ./helpers
COPY Report/helpers ./helpers/Report/
COPY jobs/sftp_downloader/job.py ./job.py

CMD /usr/bin/amqp-declare-queue --url=$BROKER_URL -q
↪  $QUEUE -d
CMD /usr/bin/amqp-consume --url=$BROKER_URL -q $QUEUE -c 1
↪  ./job.py
```

This Dockerfile is built from an official *Python* image, like the second stage of Data/Dockerfile.production and in addition, it defines $BROKER_URL and $QUEUE as environment variables. Moreover, it sets up the amqp-tools apt package, which includes a variety of utilities that make it easier to communicate with *RabbitMQ* servers. All *Python* code necessary to run the sftp_downloader task is also included in the build, within some files that are not strictly used by this procedure, but that are required due to strange dependencies still present in some parts of the code.

The postprocessor job has a more complex Dockerfile since its image also requires running the *Rust* executable, which contains the algorithm used to post-process the data. It shares the two-stage structure of the initial *Data* component image. Plus, along with the software included in the second stage, it also requires the amqp-tools apt package installed because it needs to interact with a *RabbitMQ* queue to get the information it needs to execute its job. The build process then is the following:

1. build a first image stage identical to the one seen with the initial *Data* component image, to get the data postprocessing software out of the *Rust* code
2. build a second stage based on an official *Python* image which also includes the amqp-tools toolbox
3. install in this second stage the required pip dependencies
4. copy the *Rust* executable from the first stage to the second
5. copy the remaining *Python* scripts from the local directory

The third and last image, the one related to the reporter job, is identical to the first one since most of the files are shared between all jobs. It only differs for the different job.py script, which is specific for each type of job. It is worth noting that, as an entrypoint, all tree images perform two program executions: at first, the amqp-declare-queue program is executed. It has the purpose of ensuring that a queue with the right name has been created. The second command, amqp-consume, listens for events in the queue related to its specific job. Without the execution of the first command, it would not work, as a blank *RabbitMQ* server does not come with a queue. This guarantees that there will always be a queue named appropriately and ready-to-accept listeners.

These Dockerfiles seen in this chapter could be lighted by providing a more granular separation of concerns in the application code. It is always a good idea to make the *Docker* images as small as possible, both in size and in build layers, since it significantly speeds up the interaction with an eventual repository. However, even if working on this aspect could be a strategic improvement occasion, it lies outside the scope of this thesis.

5.2.3 Step 3: *Kubernetes Jobs* and *Keda*

As already mentioned before, a `Job` in *Kubernetes* is an `API` object that provides a way to spawn `Pods` executing a program. After the successful end of the program, the job is terminated. The general structure of a `Job` configuration `YAML` file resembles that of a `Deployment`: there are 4 top-level properties, which are the same as requested by a `Deployment` file (`apiVersion`, `metadata`, `kind` and `spec`), where the `spec` attribute is a *key-value* object with a `template` key that contains the specifications of the `Pod` that will run within the `Job`, in the same way a `Deployment` defines the specifications of its related `Pods`. Additionally, a `Job` can define some other configurations. For instance, a developer can indicate the degree of parallelism required to execute a job (property `spec.parallelism`), the number of `Pods` that need to terminate successfully to consider the job accomplished (property `spec.completions`), and so on. The full list of features can be found on the official documentation page.

However, a `Job` itself is unable to scale its instances concerning the number of messages coming to a queue. *Kubernetes* has not been designed with this specific scope in mind and this case history is rare compared to many other cases. There exist some projects on GitHub aimed at obtaining this particular behavior, but the majority of them are not maintained and are based on obsolete technologies. Some of them were also taken into account for the development of this project, but within a few tries they turned out to be unsuitable for the project's sake. For example, onfido/k8s-rabbit-pod-autoscaler consists of a simple bash script polling the queue to know the number of messages and interacting with the *Kubernetes* `APIs` to scale up or down the number of `Pods`. The project is now archived, and since it was also difficult to configure, a better alternative was needed. Fortunately, a better alternative exists and it consists of *Keda*. *Keda* extends the *Kubernetes* `APIs` to provide some new types of objects, which act as wrappers for more native *Kubernetes* objects and allow to specify a scaling policy based on events. *Keda* allows scaling of `Deployment` and `Job` objects and therefore exposes an `API` to manage the wrapper objects for these types of resources. In particular, this project uses the `ScaledJob API`, which manages a resource that wraps the native `Job` *Kubernetes* object to make it easily scalable. To use *Keda*, however, it is necessary to deploy the *Keda* runtime in the *Kubernetes* cluster of interest. There are different ways to execute this task. In this project, the *Keda* runtime was deployed using an *Helm chart*. *Helm* is a *package manager* for *Kubernetes*: it provides a command-line interface that allows integration of a *Kubernetes* cluster with third-party plugins. These plugins, which consist of a collection of files that describe a related set of *Kubernetes* resources, are called *Charts*. The steps required to install the *Keda* runtime in a *Kubernetes* cluster using its *Helm chart* are the following.

1 Install the *Helm* package manager

Command Line

```
$ gitdomain=https://raw.githubusercontent.com
$ url=$gitdomain/helm/helm/main/scripts/get-helm-3
$ curl -fsSL -o get_helm.sh $url
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

2 Add the *Helm Keda* repository to the local *Helm* package list

Command Line

```
helm repo add kedacore https://kedacore.github.io/charts
```

3 Update the *Helm* package index

Command Line

```
helm repo update
```

4 Deploy the *Keda Helm* runtime in a reserved namespace

Command Line

```
helm install keda kedacore/keda --namespace keda
↪ --create-namespace
```

It is not mandatory to deploy the *Keda* runtime in a dedicated namespace, but it is advised by the official tutorials. This should limit the possibilities of corrupting in any way the *Keda* installation, by overwriting its components with custom resources.

Once the environment has been set up, the first `ScaledJob` can be configured. An example can help to comprehend how a `ScaledJob` can be configured. Hence, let us consider the `YAML` configuration file of the `sftp_downloader` `ScaledJob`.

_k8s/zamperla-app/Data/scaledjobs/sftp_downloader.yaml

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledJob
metadata:
  name: data-sftp-downloader-scaledjob
  namespace: default
  labels:
    app: zamperla-app
    tier: data-sftp-downloader
spec:
  jobTargetRef:
    template:
      spec:
        containers:
        # ...
    backoffLimit: 4
  pollingInterval: 2
  minReplicaCount: 1
  maxReplicaCount: 100
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  triggers:
  - type: rabbitmq
    metadata:
      queueName: sftp-download
      host: amqp://guest:guest@rabbitmq-service:5672
      mode: QueueLength
      value: "1"
```

The first interesting thing to note here is the `apiVersion` property: the file is not using an official *Kubernetes* `API` version, but it is using a *Keda* extension of such `API`. In this way, resources such as `ScaledJob` and `ScaledDeployment` can be handled within the context of a *Kubernetes* configuration file. The `metadata` property, on the contrary, does not contain anything different from what has already been described in the previous chapters. The `spec` property contains most of the aspects that characterize a `ScaledJob`:

- `jobTargetRef` contains the specs that define the `Job` to trigger and, for extension, the definition of the `template` describing its `Pods` and the containers they should run, as long as a `backoffLimit` quote, which expresses the number of `Pod` failures that can occur before considering the `Job` as failed.
- `pollingInterval` indicates the time, in seconds, to wait between each request made to the *RabbitMQ* server aimed at knowing how many messages need to be processed.
- `minReplicaCount` expresses the minimum number of active `Jobs`: in this case, at least one `Job` should always be ready to read from a message queue.
- `maxReplicaCount` expresses the maximum number of active `Jobs`: there should not be more than 100 active `Jobs` at the same time, since they would require too many resources. This threshold should be tuned in a production environment to maximize the number of jobs that can be executed concurrently without taking resources destined for other processes.
- `successfulJobsHistoryLimit` is the number of successful jobs to show when the `kubectl get job` is executed.
- `failedJobsHistoryLimit` is the number of failed jobs to show when the `kubectl get job` is executed.
- `trigger` contains the parameters to tune the scaling conditions to apply to the `ScaledJob`:
  - `type` expresses the type of trigger to use. In this case `rabbitmq` is used, as it reflects the technology adopted to implement the message queue.
  - `metadata.queueName` is the name of the *RabbitMQ* queue to listen for.
  - `metadata.host` reports the host address where the *RabbitMQ* instance is running.

  - metadata.mode indicates the type of metric to apply. There are 2 available metrics in *Keda* that allow defining a trigger condition: the QueueLength, which indicates the number of messages present in the queue as the value to which the trigger condition should be applied, and MessageRate, which uses the arrival rate of the messages instead. In this case, the most suitable alternative is QueueLength.
  - metadata.value is the threshold to apply to the metadata.mode metric in order to fire a new job. In this case, its value is 1, which means that "every new message adds a job instance to the jobs pool".

The *Keda API* supplies many other options, both in terms of job and trigger configuration. However, those listed in this file have been chosen as the most useful in the scenario of this specific project. Other possibilities will be examined in a production setting to refine the ScaledJob setup document to make the system as reliable and effective as possible.

The configuration files related to the reporter and postprocessor jobs are identical, except for the name of the *queue* to connect to, the container image to execute in each Pod, and the name of the object to create. Therefore, there is no need to report their content here. These files have been placed in the _k8s/zamperla-app/Data/scaledjobs folder. The resource can be created using the following command.

Command Line

```
kubectl apply -Rf _k8s/zamperla-app/Data/scaledjobs
kubectl get scaledjobs
```

```
NAME                MIN  MAX  TRIGGERS  READY  ACTIVE  PAUSED   AGE
data-postproce...   1    30   rabbitmq  True   True    Unknown  2m32s
data-reporter-...   1    30   rabbitmq  True   True    Unknown  2m32s
data-sftp-down...   1    30   rabbitmq  True   True    Unknown  2m32s
```

Querying the *API controller*, we obtain a list of the created ScaledJobs, which are both ready (meaning they are ready to scale, i.e. to create new job instances) and active (meaning that there is at least one job instance running since the minimum number of active jobs has been set to 1 there is one instance running for each type of job). The system, however, is not complete: as the last step, the service code should be altered so that instead of making direct calls, a message is placed in the appropriate queue.

5.2.4 Step 4: turn direct calls into messages

The last step missing to complete the lap is to physically replace function calls to `sftp_downloader`, `auto_postprocessing`, and `create_reports`, present in the *Data* service code, with the insertion of a message into the appropriate queue. There are mainly a couple of points where this substitution is an actual necessity. In fact, each of these 3 functions can be called synchronously from the frontend, through an `HTTP API` request, but synchronous requests are not a big issue from a performance point of view: since the graphic user interface made available by the *UI* component will be rarely used, it is unlikely that so many requests to overload the system will overlap. Therefore, in the first phase, those points where the task methods are called synchronously can be left out. The two main points where, instead, it is crucial to use an asynchronous execution passing through the message queue are:

- `POST /jobs/add/complete_process`
- `POST /jobs/add/sftp`

These two endpoints are handled by the `Data/main.py` file and the code line responsible for the scheduling and execution of such methods is similar to the following:

```
Schedule.add_job(task_function, trigger =
↪  CronTrigger_from_rule(id_rule), args=[...])
```

Here, the execution of `task_function` is scheduled, thanks to *APScheduler*, within the chronological condition given by `CronTrigger_from_rule(id_rule)` (which is a utility function that queries the database to retrieve information related to the rule of interest). Once the chronological condition is met, the execution of `task_function` is triggered. Instead of directly triggering the task function, the desired behavior is that of triggering the insertion of a message into a *RabbitMQ* queue. The following code fits with this necessity well.

```
def task_function_substitute(id_rule):
    # ... retrieving context information
    data = { "rule_id": id_rule, "operation_id":
    ↪  str(id_operation), "log_file_path": name_log_file }
```

```
queue_connection = pika.BlockingConnection(
↪  pika.ConnectionParameters('rabbitmq-service') )
queue_channel = queue_connection.channel()
queue_channel.basic_publish(exchange='',
↪  routing_key='queue-name', body=json.dumps(data))
```

This code basically initiates a connection with the *RabbitMQ* server and sends a message with a `JSON` body, containing basically the id of the rule, which is the critical piece of information, plus the `operation_id` and the `log_file_path`, which are context pieces of information used for outline tasks like logging, to the specified queue. Then, as described in the previous sections, a job will be triggered and will accomplish the task requested within the message body.

This last step completes the system roundtrip, which can now be fully described by the following sequence of actions:

1. The scheduling of a job is triggered by a user from the Web interface made available by the *UI* service.
2. The request is initially handled by the *Backend* component but then is forwarded to the *Data* service.
3. The *Data* service inserts in the database the information related to requests and then schedules the job with the help of *APScheduler*.
4. When the trigger condition is met, the *Data* service sends the information needed to process the job to the right *RabbitMQ* queue.
5. The arrival of the message to the queue triggers the execution of the right type of job, which consumes the message that executes its related task function. If many messages arrive at the same time, many `Pods` are spawned: one for each message. (The limit of concurrent `Pods` at the moment is set to 100, but it can be tuned to best meet the peculiarities of the production environment.)
6. Once the task has been accomplished, if the setting requires further data manipulation (for example, the execution of the `postprocessor` job after the end of its related `sftp_downloader` job), then a message is sent to the right *RabbitMQ* queue, and so on until the whole processing chain has been executed or a terminating condition is met (e.g. no new items have been downloaded and so there are no items to post-process)

7. every time a task changes its status (waiting, running, completed) the database is updated consequently so that the job status can always be displayed updated in the Web interface

5.3 Different jobs require different priorities

The resulting system includes a pool of jobs that get executed when precise events happen. However, the infrastructure that underlies the system is made up of limited physical resources. Thus, it is impossible to think that the system will be able to scale indefinitely: there is a concrete limit of `Pods` that can be executed concurrently. Since each `Pod` allocates some resources, this limit is given by the number of `Pods` with the sum of resource requests that saturates the system resource availability, whatever it is the kind of resource that comes to saturation: storage, computational power, or memory. The first limit that is reached determines that the system cannot execute any additional `Pod`. However, there is a hierarchy that defines which processes must always be alive, which processes should be alive whenever possible, and which processes can instead wait for resources to become available without any particular urgency. Table 5.3 defines the priority of each process, where 1 is the highest priority and 4 is the lowest.

Service	Priority
UI service	1
DB service	1
BackEnd service	1
Data service	1
`sftp_downloader` job	2
`postprocessor` job	3
`reporter` job	4

Web services have the highest priority since they are the main actors in the system: they are responsible for handling the user interaction and thus they should always be reachable and usable. Plus, it is unlikely that they will consume remarkable quantities of resources: the tasks they have to execute are easy and quite straightforward. Jobs, instead, have lower priority with respect to Web services, and do not share the same priority even between themself: the `sftp_downloader` job is the job with the highest priority, since without downloading data no processing task is possible. Then the `postprocessor` job has higher priority with respect to the `reporter` job, since without the post-processing phase no report generation can be made.

In the end, the `reporter` job has the lower priority. It is necessary to tell *Kubernetes* to give these `Pods` a higher or lower priority according to this hierarchy, to ensure a resource allocation compliant with these specifications. A particular type of *Kubernetes* resource, the `PriorityClass` resource, can help achieve this result.

5.3.1 Hierarchize *Kubernetes* objects with `PriorityClass`

Kubernetes provides a set of tools aimed at configuring in a more granular and specific way the scheduling logic applied by the scheduler to start resources. It exposes such tools through the `scheduling.k8s.io` API. One of the tools provided is precisely the `PriorityClass` API object. According to the official documentation:

> A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the name field of the PriorityClass object's metadata. The value is specified in the required value field. The higher the value, the higher the priority.[19]

In order words, priority classes can be assigned to `Pods` to alter the normal scheduling of resource allocations to encourage the execution of some `Pods` instead of others. For simplicity, and to exaggerate the difference between priority classes, the higher priority (the one associated with Web services, which in table 5.3 had value 1) will have the value 1000000 and will be named `always-up`. Its configuration file content is reported below.

_k8s/zamperla-priority-classes/always-up-priority-class.yaml

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: always-up
value: 1000000
globalDefault: false
```

Another interesting concept is that of the *Preemption Policy*. The default preemption policy for a priority class is `PreemptLowerPriority`, which means that, when the scheduler tries collecting resources to start a `Pod`, it can kill other `Pods` with a lower priority class (`Pods` without an explicitly defined priority class have a value of 0). This is the case for Web services that must always be up and running, even if some lower-priority Pods need to be killed to achieve the target. However, this is not the case for job priority classes, which will not be able to kill lower-priority Pods, since it would involve losing the work made by the running Pods. For example, if a new download job is started, but there are not enough resources to satisfy the request, postprocessor or sftp_downloader jobs should not be interrupted because their work otherwise would get lost. Then these priority classes will have the property preemptionPolicy set to Never. Table 5.3.1 contains a summary of the priority classes created to manage the scheduling of the Pods for this project.

Name	Value	Preemption Policy	Ref. to table 5.3
always-up	1000000	PreemptLowerPriority	1
high-priority	100000	Never	2
mid-priority	10000	Never	3
low-priority	1000	Never	4

Now priority classes can be added to the template of each Pod by setting the property template.spec.priorityClassName: <priority class name> in each deployment or ScaledJob configuration file. Obviously, resources like *RabbitMQ* Pods must have the highest available priority.

Chapter 6

Infrastructure monitoring and observability

Monitoring and observability are nowadays key points in the design and development of software infrastructures. The policies of many companies require a high level of system monitoring and observability from service providers, eventually accompanied by certifications released by experts and auditors. A system should be as reliable as it can be, preventing incidents and giving the possibility to inspect the reasons if one happens. **System Monitoring**, in the first place, is defined as "*the process of collecting and analyzing data about IT systems, including their performance, availability, and security*"[20], while **Software Observability** can be described as "*Software observability is the ability to understand the internal state of a software system based on its external outputs*"[21]. Thus, observability and monitoring are not the same thing, even if monitoring is necessary to achieve observability. With the rise of distributed systems and microservice architectures, observability has become more and more relevant for the maintainability, resilience, and dependability of software solutions, which explains the growing interest in observability technologies, even in academic research fields. However, observability can also be achieved using a set of strategies that do not require the necessary complexity of software solutions. The practices that are often involved in the observability processes are the collection of logs, the collection of metrics (which are numerical values that measure the performance of a system), and the monitoring of traces (i.e. sequences of events that show how requests flow through a system). Nevertheless, the collection of logs, metrics, and traces can be a tedious task for a developer team that works with large infrastructures. Fortunately, some software tools have been developed to automate these tasks. One of these is *Prometheus*, which is the main tool used in this project to achieve clear monitoring and a painless observability process.

6.1 Prometheus: observability implementation in a *Kubernetes* cluster

6.1.1 Introduction to Prometheus

Prometheus is an open-source framework that provides a way to implement monitoring and observability in distributed infrastructures. It also provides features that allow for easy interaction with *Kubernetes* clusters. Plus, it is gaining more and more popularity in recent years: for example, *7Pixel*, owner of the widely-used price comparator trovaprezzi.it, which is the company I worked in from January to August 2023, is going to replace *NewRelic*, a widespread closed-source software for system monitoring and observability, with *Prometheus* and *Grafana*. Describing in detail *Prometheus*, its whole set of features, and its inner functional specifications, is out of the scope of this thesis. However, some key points are worth mentioning.

- For **metric collection** *Prometheus* uses the *pull method*, which allows metrics retrieval over `HTTP` protocol. Thus, the services under observation should expose metrics at `/metrics` endpoint, which will be polled by *Prometheus* at regular intervals. There are cases when this method does not work properly, such as the case of short-lived processes that do not run enough time to allow *Prometheus* to scrape the metrics. In these cases, the metrics can be sent to *Prometheus* from the interested job through a service called *Pushgateway*.

- *Prometheus* exposes an `API` that allows software developers to develop extensions called ***Prometheus* Exporters**. Exporters can be used to export different types of metrics from any third-party product into the compatible *Prometheus* metrics format.

- In order to allow external services to query information, *Prometheus* exposes data through *PromQL*, which is a flexible query language. Third-party extensions, as well as the Prometheus user interface, can use this querying language to retrieve metrics from a *Prometheus* server.

- *Prometheus* uses *time-series databases*, particularly suited to manage data that are correlated with time, to store information. Plus, even if it stores data in the local storage by default, *Prometheus* also gives the possibility to set up remote replicas, to avoid single points of failure.

- **Alert manager** in *Prometheus* can be configured to offer real-time notifications when predetermined thresholds are reached for specific metrics. Notifications can be sent via emails or pub/sub messaging services.

It comes naturally that a *Prometheus* installation requires many components and is not as trivial as it may seem at first sight. Image 6.1 extensively represents a service infrastructure that is surrounded by a *Prometheus* setup, which resembles all the points listed above. In the following subsection the steps required to configure a monitoring architecture, similar to the one exposed in this image, will be described.

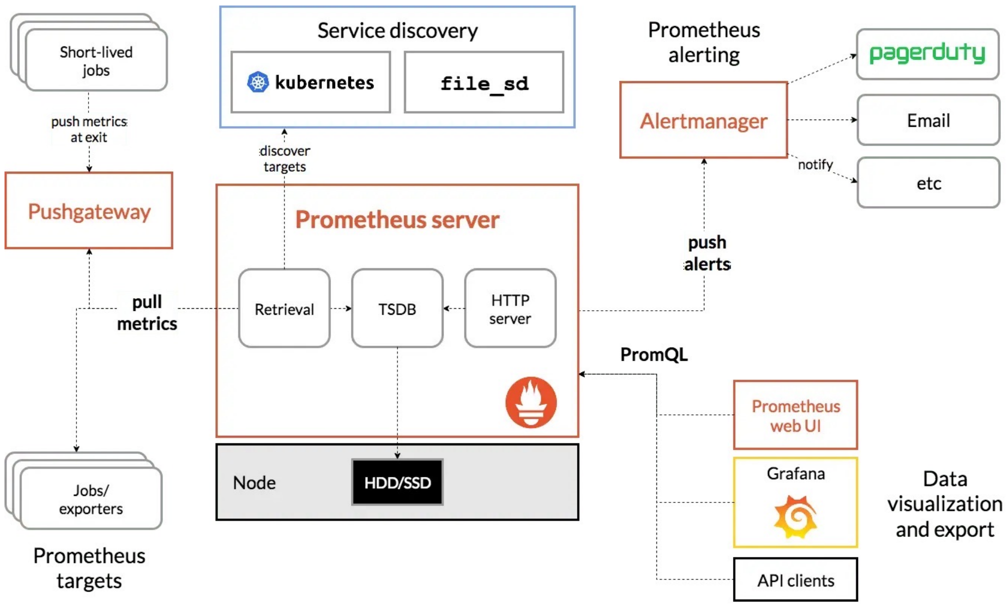


Figure 6.1: A graphic representation of a software architecture using Prometheus for monitoring

6.1.2 *Prometheus* setup in a *Kubernetes* environment

To set up *Prometheus* within the context of this project, the following steps are required.

1 First of all, let us create a separate namespace. In our case, the identity name of this namespace will be `zamperla-monitoring`. To do this, it suffices to execute the following command:

Command Line

```
kubectl create namespace zamperla-monitoring
```

This way, all the monitoring components that will be created within this namespace will be logically separated by the application resources.

2 Since the *Prometheus* server needs to query the *Kubernetes* API to retrieve metrics related to the cluster resources, an `RBAC` role should be defined. This step is needed because, otherwise, *Prometheus* would perform requests with the default role, which does not have enough permissions to retrieve data about *Kubernetes* object states. The manifest file of the `RBAC` role will look like the following.

_k8s/zamperla-monitoring/cluster-role.yml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources: ["nodes", "nodes/proxy", "services",
↪   "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
```

A `ClusterRole` defines a set of accessible resources within the modes these resources can be accessed, but it does not assign these permissions to an account. Thus, a `ClusterRoleBinding` object is used to assign these permissions to the default `ServiceAccount` of the `zamperla-monitoring` namespace, which means that all processes running inside the `zamperla-monitoring` namespace will grant the permissions defined in the `ClusterRole` manifest file.

_k8s/zamperla-monitoring/cluster-role-binding.yml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: zamperla-monitoring
```

The cluster role and the cluster role binding can be created using the `kubectl` command like any other *Kubernetes* resource.

3 *Prometheus* makes use of a configuration file called `prometheus.yaml`, which handles all the scrape specifications, service discovery details, storage locations, data retention configurations, etc. Furthermore, another file called `prometheus.rules` is used to configure the *Prometheus* alerting rules, which specify the metrics to watch and the thresholds that, if reached, will require the system administrator to be notified. These files will be mounted to the `Pod` directory `/etc/prometheus` through a `ConfigMap`. Describing in detail how *Prometheus* can be configured, its various features and preferences is outside the scope of this thesis and would require alone a whole book (which has also been written, though [22]). For this thesis, it is sufficient to say that the scrape configuration is a list of *jobs*, where a *job* in *Prometheus* is intended as the process that collects metrics from a set of endpoints of the same type. In the standard configuration used for this project, there are the following *jobs*:

- `kubernetes-apiservers`, which collects metrics from the *Kubernetes* API server
- `kubernetes-nodes`, which collects metrics exposed by the cluster nodes
- `kubernetes-pods`, which collects metrics from those `Pod`s that have the metadata attributes `prometheus.io/scrape` and `prometheus.io/port`

- `kubernetes-service-endpoints`, which collects metrics from those `Services` that have the metadata attributes `prometheus.io/scrape` and `prometheus.io/port`

The `/etc/prometheus/prometheus.yaml` is `YAML` file that contains:

- A `globals` section that specifies generic configurations. For example, in the configuration file used in this context, the `scrape_interval` (the time to wait between two adjacent polling requests) and the `evaluation_interval` are defined.
- A `rule_files` array containing the absolute paths of the alerting configuration files.
- A `scrape_configs` array of objects. Each object defines a *job* and contains the information needed to allow the *Prometheus* server to scrape the resource correctly - like endpoint, protocol, security certificates, etc.
- Many other context-specific configurations are available. A complete list of available features can be found at prometheus.io/docs/prometheus/latest/configuration/configuration.

4 The next step consists of creating the `Deployment` that will run the *Prometheus* server. It consists of a simple `Deployment` created in the `zamperla-monitoring` namespace. It has 1 replica and it makes use of the official *Prometheus Docker* image. The following arguments are passed to the entry point of the *Docker* image:

- `-storage.tsdb.retention.time=12h`: sets the metrics retention period to 12 hours, which means that after 12 hours metrics will get lost.
- `-config.file=/etc/prometheus/prometheus.yml`: indicates the path of the configuration file.
- `-storage.tsdb.path=/prometheus/`: indicates where to store the database data.

Moreover, the `ConfigMap` defined in step 3 is mounted as a volume, and an `emptyDir` volume is mounted to the `/prometheus` path, making *Prometheus* data persist until the `Deployment` is deleted. The attached file `_k8s/zamperla-monitoring/deployment.yml` reports the content of the *Prometheus* `Deployment` configuration file, which however lacks a persistent storage configuration, essential in a production environment.

_k8s/zamperla-monitoring/deployment.yml

```yaml
# ...
metadata:
  name: prometheus-deployment
  namespace: zamperla-monitoring
  # ...
spec:
  replicas: 1
  # ...
  template:
    # ...
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus
          args:
            - "--config.file=/etc/prometheus/prom.yaml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-server-conf

        - name: prometheus-storage-volume
          emptyDir: {}
```

5 The fifth and last step plans to define a *Kubernetes* `Service` to provide an access point for the Web UI component of Prometheus. Since the `Pod` exposes the 9090 port, it is sufficient to create a `NodePort Service` that exposes the 9090 port to the outside world at any port available in the node.

Command Line

```
kubectl expose deployment prometheus-deployment
 ↪ --type=NodePort --port=9090 --name=prometheus-service
 ↪ --namespace zamperla-monitoring
minikube service prometheus-service
```

Launching the `Service` will open the *Prometheus Web UI*, which is a powerful tool to visualize the time-based data collected by the software.

Now *Prometheus* is up and running, ready to collect metrics and expose data from its endpoints. If there is the need to collect metrics from running `Pods`, all `Deployment` configuration files should include the metadata properties `prometheus.io/scrape=true` and `prometheus.io/port=<exposed port>` to all `Pods`. For this use case, it is considered sufficient to monitor node resources only, since for the application domain events monitoring there is already a functional, properly working logging system that can handle this need.

6.2 Next steps: *Graphana* and Alert System

Prometheus is easily expandable with plugins and libraries that allow shaping the product on the infrastructure needs. It can also be used as-is, but the Web UI provided by *Prometheus* is quite complex to use, and data are often presented in a conflictual form. This is the main reason why it could be convenient to integrate *Prometheus* with *Graphana*.

6.2.1 Graphana: a visualization tool

Grafana is a free, open-source platform for monitoring and observability. It enables organizations to gain insights into the performance of their systems, applications, and infrastructure by visualizing and analyzing data from multiple sources. This tool is widely used in DevOps and is often combined with other tools such as *Prometheus*, *InfluxDB*, and *Elasticsearch*, that help create an omnicomprensiva monitoring experience with complete and powerful dashboards. In particular, the creation of dashboards makes it easy to have quick access to the most crucial performance indexes of a software architecture. In our infrastructure, *Graphana* can be configured as follows.

1 A `ConfigMap` should be set up to contain the contents of the configuration file for *Graphana*, similar to what was done for the *Prometheus* setup. This configuration is used to provide Graphana with the parameters it needs to connect to the data source, which in this case is *Prometheus*. Thus, the configuration file should be named `prometheus.yml` and should include the following:

```
/etc/grafana/provisioning/datasources/prometheus.yml
# ...
{
    "apiVersion": 1,
    "datasources": [
        {
           "access":"proxy",
            "editable": true,
            "name": "prometheus",
            "orgId": 1,
            "type": "prometheus",
            "url":
 ↪  "http://prometheus-service.monitoring.svc:8080",
            "version": 1
        }
    ]
}
```

The configuration file mainly expresses the URL to query and retrieve the time-series data from the *Prometheus* `Service`. The `_k8s/zamperla-monitoring/graphana-datasource-config.yml` file will contain a `data.prometheus.yml` property valued with the content of this file.

2 The next step requires the configuration of a `Deployment` that uses the official *Graphana Docker* image. This `Deployment` is very similar to the one created for the *Prometheus* installation. it pulls the image from the official *Graphana* repository and it mounts two volumes: one for the configuration file and one for those data that should be persisted. Also in this case the persistent volume will be mocked with an `EmptyDir` volume. This `Deployment` also makes use of 1 replica and exposes the `3000` port. Within the `Deployment`, a `Service` should be configured to make its `Pod` reachable by the browser: in this case, a `NodePort` `Service` forwarding the container port 3000 to the node port 32000. The *Graphana* Web UI can then be accessed at the specified node port. The credentials to enter the admin panel in a fresh Graphana installation are: `user: "admin", password: "admin"`.

3 The last step consists of setting up a *Graphana* dashboard to visualize *Prometheus* metrics. Fortunately, there are some templates ready to be imported and shared directly from the official *Graphana* website. As an example, the configuration hosted at grafana.com/grafana/dashboards/8588-1-kubernetes-deployment-statefulset-daemonset-metrics will be used. These are community-developed solutions that can help make the *Graphana* setup process easier. The import process is well described in the official *Graphana* documentation. In particular, the content of the `manage dashboards` section of the official documentation website is exactly the procedure that has been used for this project.

Once the dashboard has been imported we can visualize all the metrics that have been set up, such as CPU utilization and memory usage, from *Prometheus* in a clear, ordered, and easily understandable manner.

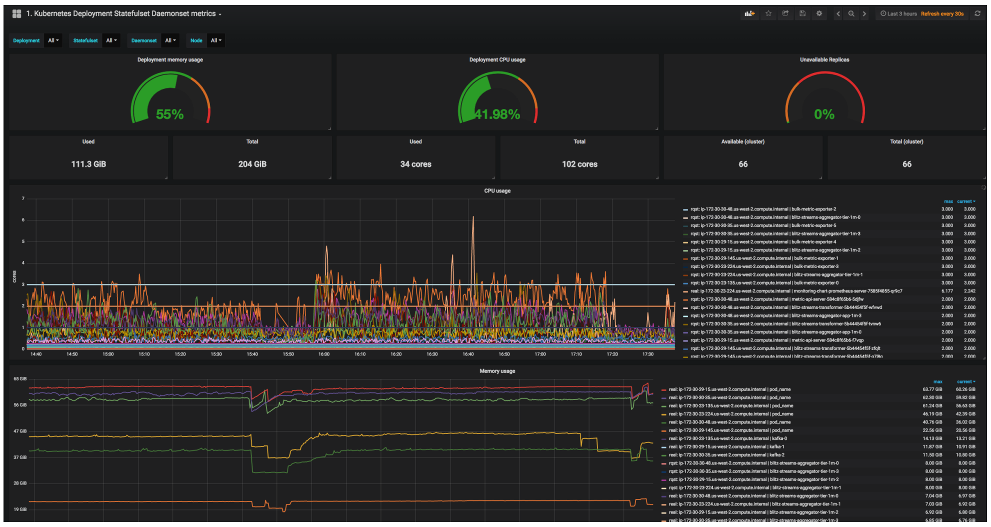


Figure 6.2: A Graphana dashboard showing some metrics

6.2.2 Alert Manager: real-time alerting system

Alert Manager has not been set up for this project, but it is an interesting feature that can be configured as a future improvement and that can surely be useful in many scenarios. It consists basically of watching for anomalies in the values reported by the metrics and notifying the system administrator, through different channels (email, pub/sub system, third-party application integration, etc.) of the occurrences of these anomalies. To set up an Alert Manager within a *Prometheus* installation monitoring a *Kubernetes* cluster, at first, the *Prometheus* instance should be properly configured to support an alert manager. In particular, 2 things are needed:

- a configuration file with the extension `.rule`
- an `alerting` section in the configuration file, which should set the values needed to connect to the alert manager (in particular, the property `alerting.alertmanagers[].targets` is an array of URLs containing the endpoints serving an alert system).

The Alert Manager needs a configuration file formatted following the Prometheus configuration file conventions. This file should be mounted as a `ConfigMap`, as seen for other setups. Then an instance of Alert Manager can be installed by setting up a `Deployment` that runs the official `prom/alertmanager:latest` *Docker* image. This `Service` should expose a port, reachable by the *Prometheus* server, which will be used to exchange messages with the latter. A detailed and descriptive configuration guide for alert managers is hosted on official *Graphana* website.

In conclusion, *Prometheus* and *Graphana* are powerful tools when it comes to setting up a complete and reliable observability ecosystem in a *Kubernetes* cluster and offer tons of configurations and customizations. However, the more customizable and powerful the tool is, the more it requires specific competencies to work on it. The result is worth the effort but requires an investment in time and staff training that cannot be taken for granted.

Chapter 7

Conclusions

The development of this project has been a challenging experience from many points of view: the project focuses on the design and development of a *Kubernetes* infrastructure for an application that involves many components and technical aspects, which needed to be handled with tools and methodologies often outside of my initial knowledge. However, it has been an engaging project to work on, and I learned a lot about the *Kubernetes* world, but also about the necessities of a growing reality like Zamperla, which applies technology at a higher level. The project is far from being finished: at least it misses a production setup, but also the way secrets are stored is not suitable for a final release. Moreover, there is still a bug in the logging procedure when it comes to scale jobs that I have not managed to fix yet. However, I am proud to say that the overall setup, except for some fine-tuning to do on the production infrastructure, is, in my opinion, reliable, well-organized, and easily maintainable. It is important to note that *Kubernetes* is a Google product, thus it has been developed to interact particularly well with Google Cloud, as well as any managed cloud I.A.A.S. products like Azure and AWS. A *Kubernetes* cluster that relies only on proprietary infrastructure is not a widespread solution and, therefore, it is harder to develop. However, the result is compelling and adaptable and can lead to a solid and reliable option. I would like to thank Professor Pietro Ferrara, who allowed me to work on this fantastic project and supported me throughout the development process. I would like to equally thank Dr. Gianluca Caiazza for the support, the helpful tips, and the interesting points of view on the discussed technologies and their applications. I hope that the work done with this thesis will help investigate the future involvement of *Kubernetes* for other university projects: many times new technologies are not adopted because the effort needed to learn how to use them is considered to be greater than the benefits provided. *Kubernetes*, at least in the working environments I have been in the last five years, is one of these. But now this powerful tool has been uncovered, at least in the academic context of Ca' Foscari, and hopefully, there will be many other projects to which this technology can be applied in the future.

Bibliography

[1] Bernstein, David, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. DOI: 10.1109/MCC.2014.51.

[2] Mobi, *Mobi*, https://github.com/moby/moby, 2013.

[3] Microsoft Community, *Containers vs. virtual machines*, https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm, 2023.

[4] The Red Hat Authors. “Advantages of using docker.” (2018), [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/7.0_release_notes/sect-red_hat_enterprise_linux-7.0_release_notes-linux_containers_with_docker_format-advantages_of_using_docker (visited on 05/18/2018).

[5] Burns, Brendan and Beda, Joe and Hightower, Kelsey and Evenson, Lachlan, *Kubernetes: up and running*. " O’Reilly Media, Inc.", 2022.

[6] Gos, Konrad and Zabierowski, Wojciech, “The comparison of microservice and monolithic architecture,” in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Apr. 2020, pp. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514.

[7] De Lauretis, Lorenzo, “From monolithic architecture to microservices architecture,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050.

[8] Smith, John and Johnson, Sarah, “Microservices vs. monoliths: A comparative study,” *Software Engineering Journal*, vol. 28, no. 3, pp. 237–245, 2020.

[9] K. Westeinde. “Deconstructing the monolith: Designing software that maximizes developer productivity.” (), [Online]. Available: https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity.

[10] D. H. Hansson. “The majestic monolith.” (), [Online]. Available: https://m.signalvnoise.com/the-majestic-monolith/.

[11] J. F. A. Linden, “Understanding gartner’s hype cycles,” May 30, 2003. [Online]. Available: http://ask-force.org/web/Discourse/Linden-HypeCycle-2003.pdf.

[12] Arundel, John and Domingus, Justin, *Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*. O’Reilly Media, 2019.

[13] The Kubernetes Authors. “Persistent volumes.” (), [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[14] The Kubernetes Authors. “Configmaps.” (), [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/.

[15] The Amazon Web Services Authors. “Amazon message queues.” (2023), [Online]. Available: https://aws.amazon.com/message-queue/.

[16] The Amazon Web Services Authors. “Rabbitmq vs. kafka: A comparison.” (), [Online]. Available: https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/.

[17] The Kubernetes Authors. “Jobs | kubernetes.” (2023), [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[18] The Keda Authors. “Keda | kubernetes event-driven autoscaling.” (2023), [Online]. Available: https://keda.sh/.

[19] The Kubernetes Authors. “Pod priority and preemption.” (2023), [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/.

[20] S. Y. Lee and A. C. Pang, “A survey of system monitoring tools for IT infrastructure,” *Journal of Network and Systems Management*, vol. 16, no. 4, pp. 417–447, 2008.

[21] P. R. Cook and A. J. Andrews, “On the definition of observability,” *ACM Transactions on Computer Systems*, vol. 38, no. 1, pp. 1–4, 2020.

[22] B. Brazil, R. Warren, and V. Marmol, *Prometheus Up and Running*. O’Reilly Media, 2019.