INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Orquestração de um pipeline de ferramentas para apoio ao ensino

RÚBEN LOURENÇO TEIXEIRA DE ALMEIDA
julho de 2022

POLITÉCNICO
DO PORTO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Orquestração de um pipeline de ferramentas para apoio ao ensino

RÚBEN LOURENÇO TEIXEIRA DE ALMEIDA
Junho de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Orchestrating a pipeline of tools to support teaching

Ruben Almeida

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Dr. Nuno Bettencourt

Porto, June 30, 2022

Dedicatory

To my family, friends and teachers that helped me unconditionally throughout this journey.

Abstract

The variety and nature of tools that are nowadays used on both academic and professional contexts have been increasing over the past few years. With the non stopping evolving cycle that technology suffers on a daily basis, this is a consequence that will be even more noticeable in a not so distant future. As a result, several problems have emerged. How to handle all of the crucial tools in a simple and reliable way? How to structure that process so it can scale, in order to apply it whenever the tools are being used by several people? Services specialization, such as microservices, taking advantage of the core tools features, allows the automation of almost all the needed configurations required.

This thesis focus on the design and implementation of a solution that enhances the ease and efficiency of creating and configuring working environments either of students or workers, being, however, more focused on the academic environment. To achieve this, a prototype tool has been developed, which consists in several services that when combined together are capable of creating and configuring a software development integration pipeline. The prototype is responsive to a certain configuration input and handles all the tasks needed in between the configuration steps, keeping the resulting pipeline always up to date.

Keywords: Microservices, Automation, GitLab, Jenkins, SonarQube

Resumo

A variedade e natureza das ferramentas que são hoje em dia utilizadas tanto em contextos académicos como profissionais têm vindo a aumentar nos últimos anos. Com o constante ciclo evolutivo que a tecnologia sofre diariamente, este aumento é um fator que será ainda mais percetível num futuro próximo. Por conseguinte, vários problemas têm surgido. Como lidar com todas as ferramentas cruciais de uma forma simples e fiável? Como estruturar esse processo para que possa ser escalado, a fim de o aplicar a situações em que as ferramentas estejam a ser utilizadas por várias pessoas? A especialização em serviços, surgindo aqui o conceito de microserviços, aproveitando algumas das funcionalidades oferecidas por parte das ferramentas, permite a automação de quase todas as configurações necessárias.

Esta tese tem como foco a conceção e implementação de uma solução que torne mais simples e eficiente o processo de criação e configuração de ambientes de trabalho, quer de estudantes quer de trabalhadores dando, no entanto, mais foco à vertente académica. Para o conseguir, foi desenvolvido um protótipo, que consiste em vários serviços que, quando combinados, são capazes de criar e configurar uma *pipeline* de integração de *software*. O protótipo tem como *input* uma determinada estrutura de dados e trata de todas as tarefas necessárias entre as etapas de configuração, mantendo assim a *pipeline* sempre atualizada.

Esta tese relata todo o processo envolvido na elaboração da solução final em cima descrita, estando inerentes as fases de estudo acerca dos conceitos fulcrais ao problema, análise de valor e de negócio, proposta de design, implementação e avaliação da solução desejada.

Palavras-Chave: Microsserviços, Automação, GitLab, Jenkins, SonarQube

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
CD	Continuous Delivery.
CI	Continuous Integration.
CPU	Central Processing Unit.
CRUD	Create, Read, Update and Delete.
CVCS	Centralized Version Control Systems.
DDD	Domain Driven Design.
DNS	Domain Name System.
DSL	Domain-specific language.
DTO	Data Transfer Object.
DVCS	Distributed Version Control Systems.
FEE	Fuzzy Front-end.
GPU	Graphics Processing Unit.
HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.
IP	Internet Protocol.
ISEP	Instituto Superior de Engenharia do Porto.
IT	Information technology.
JSON	JavaScript Object Notation.
K8S	Kubernetes.
LAPR	Laboratory / Project.
LDAP	Lightweight Directory Access Protocol.
LXC	Linux Containers.
NCD	New Concept Development.
NPD	New Product Development.
OAuth	Open Authorization.
OS	Operating System.
PaaS	Platform-as-a-Service.
RC	Remote Control.
REST	Representational state transfer.
RPA	Robotic Process Automation.
RQs	Research Questions.

TCP	Transmission Control Protocol.
TDD	Test Driven Development.
UDP	User Datagram Protocol.
UI	User Interface.
UML	Unified Modeling Language.
URL	Uniform Resource Locator.
VCS(s)	Version Control System(s).
VM	Virtual Machine.
VXLAN	Virtual Extensible Local Area Network.
YAML	Yet Another Markup Language.

Chapter 1

Introduction

Nowadays, on both academic and professional contexts, there is a vast amount of tools needed to perform day-to-day tasks. Maintaining those tools, in order them to be easily configurable and less error-prone is, sometimes, not an easy task.

However, with the current state of technology, automated approaches to address those issues are a reality and can be developed.

This chapter presents a detailed overview about the problem under study and its context, as well as the objectives and the approach used throughout the work.

1.1 Problem

Nowadays, either on academic or professional contexts there is an increasing amount of key indispensable tools that are required to complete daily common tasks.

Specially, in a technological environment such as a computer engineering students department, these key tools are literally everywhere. For instance, from *Git* to *Jenkins* and *SonarQube*, these are all tools that require manual actions, in this case from the teachers, to be configured and ready to be used. However, the configuration process involved is, sometimes, time-consuming and susceptible to errors. This factor can hamper the overall need of manage a variable group of students that can change in time, causing these tools to be updated with a certain frequency. Being, in most of the cases, the various tools interdependent, applying a change in one of them, means that all of the others need to be updated as well, causing the overall manual configuration time to increase exponentially. The constant need of change combined with the fact that, most of the times, the configuration processes are manual, is certain to say that this problem will get more severe as number of tools to manage increases.

This is a recurrent problem that becomes more visible at the end of the semester on the computer engineering department, when the final projects get started. This happens because at that time, students are organized into various groups and, for each one of the groups, a set of tools needs to be manually configured and integrated. This is necessary either to allow the students to accomplish their final projects, either to allow teachers to grade those projects at the end. However, as mentioned before, it is a very time-consuming task for the teachers, that can be even more complex when there are internal changes on the student groups after the start of the projects.

1.2 Goals

The aim of this work is to design and develop various components that when orchestrated correctly automate the environment setup and integration processes for a set of predefined tools as well as providing relevant information to manage and monitor those tools. To do so, it is also part of the goals to develop the entire infrastructure necessary to support the life-cycle and management of the various components stated before.

The idea behind these components is to ingest a well defined user input and transform it into meaningful data, allowing the automated configuration of the tools.

In addition, there is an underlying study about the performance, maintainability and reliability of the system in comparison with manual methods of configuration.

1.3 Research Questions

Based on the stated goals aforementioned, the following Research Questions (RQs) were derived:

RQ1 Would an automated tool be quicker and less error-prone when performing configurations tasks than a traditional manual approach?

- The aim of this question is to understand if an automated approach, on its overall, can be more trustworthy and quicker than performing the tools configurations manually.

RQ2 Is an automated approach viable in future in terms of maintainability and reliability, when the number of components to configure starts to increase or a manual approach is preferable?

- The aim of this question is to understand the benefits and drawbacks of automated and manual approaches when the number of tools to configure increases, in order to determine which approach is the best.

1.4 Research Methodology

In order to guide the investigation performed during the elaboration of the current thesis, the *Design Science Research* methodology was followed. This methodology is mainly applied to investigations where is intended to develop new solutions in order to fulfil a set of business requirements [14].

To enable understanding the problem inherent to the research questions various studies were made about the key concepts involved. Those studies were based on existing researches and papers that stated those same concepts from where the important information was extracted and properly treated and organized.

1.5 Hypothesis

Considering the research questions previously defined, the following hypothesis were developed and should be taken into consideration later on:

H1 An automated approach is definitely quicker and less error-prone when performing the configuration of various tools, specially when handling cascade changes.

H2 Even though an automated approach requires a more complex solution, its maintainability and reliability in the future, as the number of tools to configure increases, is not be compromised. The solution is kept simple and straight forward.

1.6 Work Plan

Having a well defined and structured working plan is crucial to manage any type of project, to the extend that it describes each and every step and tasks involved. However, it is important to keep in mind that this plan can suffer changes over the project lifecycle period.

Regarding the current work, the relevant information about each of its steps is here explained:

Exploratory Research Study an explore the key concepts within the domain of the problem, stating the current state of the art.

Contextualization of the problem Detailed analysis of the problem, stating its relevance and the value of its solution.

Analysis Here, an exploratory study is made so its possible to clearly understand the problem as well as the different possible approaches to it. The required use-cases are defined and an architectural model its established.

Design Here, an architectural proposal is elaborated as well as its possible alternatives, in order to fulfill the requirements previously established.

Development The implementation of the solution that aims to solve the problem at hand is done, taking into accounting the model defined on the previous step.

Validation The solution implemented is validated based on the success criteria previously defined, where its possible to point out future improvements and work to be done as well as the appropriated conclusions.

Documentation This is a continuous step that happens in parallel with all of the others and consists of writing the thesis. It functions as a means of transmitting and persist the gathered knowledge.

1.7 Thesis Structure

In order to make it more organized and easy to understand, this thesis was divided in several chapters.

The chapters that compose the current thesis are the following:

Chapter 1 (Introduction) 1 Presentation and contextualization of the problem under study in order to give a better understanding of the overall project and its objectives.

Chapter 2 (State of the Art) 2 Study and presentation about the various concepts and technologies that belong to the problem domain

Chapter 3 (Analysis) 3 Divided into business value and requirements engineering, it aims at presenting a value proposition to the solution as well as it constraints and technical requirements.

Chapter 4 (Design) 4 Presents the design adopted in order to conceive the solution as well as other design alternatives that could be considered.

Chapter 5 (Implementation) 5 Explanation about the implementation process, its requirements and the technological stack used.

Chapter 6 (Experiments) 6 Focuses on the solution evaluation and the validation of the obtained results.

Chapter 7 (Conclusion) 7 Presents the final gathered conclusions that can answer the research questions as well as a future perspective about the solution implemented and the problem itself.

Chapter 2

State of the Art

In this chapter, an overview regarding the different topics that surround the problem scope is provided.

It starts by presenting the technical related topics, where is firstly given an overview about software containerization and orchestration as well as different types of software architecture, since those are crucial points for any new application.

Right after, due to the nature of the problem, the concepts of *DevOps*, Continuous Integration (CI) and Continuous Delivery (CD) are described, being complemented with other important topics (Version Control System(s) (VCS(s)), Code Quality and Software Automation), in order to give a fully detailed comprehension of the concepts surrounding the problem.

Lastly, the business related topics are presented, such as Requirements Engineering. However, it is given a special focus to the Innovation Process and Value Proposition subjects.

2.1 Software Containerization

A recurring problem faced by developers is having code working successfully on a machine but not on other one due to the differences on the computing environment, if they exist [12].

According to the current technological trend, Microservice Architectures are rising in popularity among the developers. Microservices address various issues associated with Monolithic Architectures.

Microservices are the atomic unit of a Microservice Architecture, which is an architectural software style to design applications as sets of independently deployable services. In many cases, software applications are easier to develop and maintain if they are broken down into smaller components which work together. Each one of the component is developed independently, and the application is then the result of the integration of these individual components.

Virtualization is not the ideal option to handle these diverse challenges. In order to tackle such issues, the need of Containerization arises. Containerization makes it easier to deploy various applications using the same Operating System (OS) on a single Virtual Machine (VM) / Server. A Container uses OS level virtualization for deploying applications instead of creating an entire VM [12].

Containers allow enveloping applications and its respective dependencies inside its own environment. Therefore, it enables them to execute in isolation while sharing the same resources, mainly the OS. It promotes a rapid and lightweight application deployment, since the resources are utilized more effectively, by not wasting them on running separated OSs.

There are many Container runtimes, however Docker is the most popular open-source tool. Nowadays, there are many IT behemoths adopting Containerization technologies due to its easy deployment, scaling, and operations [12].

2.1.1 What is Docker

Docker is an open-source engine, that aims to automate the deployment of applications into containers [40]. It was developed by the team at *Docker, Inc*, also known as *dotCloud Inc*, a pioneer in the Platform-as-a-Service (PaaS) domain. Also, it was initially released by its team under the *Apache 2.0* license.

What makes Docker unique is the fact that it adds a deployment engine on top of a virtualized container execution environment. The way it was thought and designed allows it to be a lightweight and fast environment on which code can be executed, as well as an efficient workflow that enables that same code to quickly scale from a laptop to a test test environment and, in a final step, to production.

At the time of its development, Docker aimed at four end goals [40]:

- *Lightweight and easy to use tool* - Applications can be *Dockerized* in a short period of time. Docker relies on a copy-on-write model, thus making changes in an application is extremely fast. Later, these applications can be packed and placed into containers, that most of the times, only take a blink of an eye to launch. Besides, since it also removes the overhead that virtual machines cause, it makes containers highly performant.
- *Logical separation of responsibilities* - Docker was designed in a way that allows developers to focus on containerize their applications, while operations can focus on how to manage those containers. The consistency between the development environment and the deployment environment is also ensured, hence avoid problems such as "worked in dev, now an ops problem".
- *Fast and efficient development life cycle* - Reducing the time between the moment that code is being written and the moment its tested, deployed and used its one crucial feature provided by Docker. Docker aims to make applications easy to build and portable.
- *Enhance the usage of a service oriented architecture* - Docker is ruled by the principle that each application should be well defined and self-contained. This promotes the usage of a distributed model on where each application or service is represented by a network of containers. The processes of distribution, scalability and debug are, this way, directly eased.

Docker is composed by several components but the ones considered core are [40]:

- *Docker Engine*
- *Docker Images*
- *Docker Containers*
- *Registries*
- *Compose and Swarm*

2.1.1.1 Docker Engine

Docker Engine is the pair of Docker client and server [40]. Docker is considered a client-side application, on which the Docker client communicates with the Docker server or *daemon*, being the server the one responsible for all the work. Docker offers a command line client *docker* and a *RESTful* Application Programming Interface (API) named *dockerd* that facilitates the communication with the *daemon*. The Docker client and

daemon can both run on the same host or a local Docker client can be connected to a remote *daemon*, running on a different host. In order to get a better understanding about the previously mentioned concepts, Figure 2.1 shows the Docker architecture.

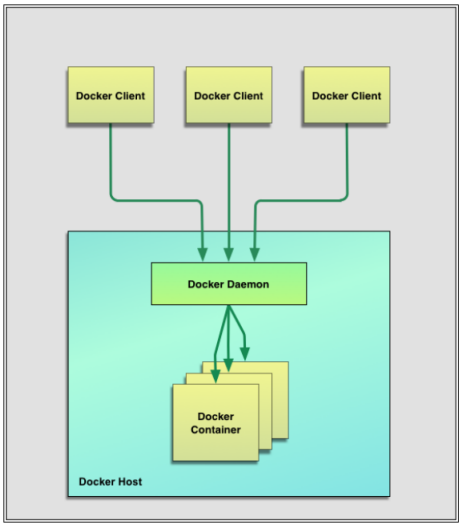

FIGURE 2.1: Docker architecture [40]

2.1.1.2 Docker Images

Docker Images represent the building blocks in the Docker domain since they are the constituent part of a container [40]. Images can be considered the result of the build process of Docker life-cycle. A layered based format, built from step-by-step instructions its how these images are created. In a more generalist approach, Docker images can be considered the source code that fuels the containers. Also, Docker images are highly portable, having the possibility to be stored, shared and updated.

2.1.1.3 Docker Containers

Docker helps in the process of building and deploying containers inside of which applications and services are packed. As mentioned before, images are the constituent part of a container so, in order to understand the container concept clearly, images can be seen as the packing aspect of Docker and containers as the execution aspect of Docker [40]. Metaphorically, Docker containers can be compared to real shipping containers, but instead of shipping goods they ship software. A good thing about containers is that Docker does not look at where they are shipped. This gives containers a huge portability and interchangeability, since they are as generic as possible. Also, Docker does not care about containers content, they can either have a web server or a database, in the end they are all treated the same way. With containers its possible to build self-contained test environments or replicate complex applications stacks, the amount of possible uses cases is infinite.

2.1.1.4 Registries

As previously mentioned, its possible to store Docker images and that is where the Registries urge [40]. Registries can either be public or private, having Docker itself a public registry for images, the *DockerHub*. Docker users are able to create an account on *DockerHub* and use it to store and share their own images there. Consequently, *DockerHub* contains a various images people have built and share over the time, images that represent crucial pieces when building software. On the other hand, private registries are, most of the times, used by companies since they allow an higher layer

of security (such as a *firewall*) that might be necessary when handling property source code or other sensitive information.

2.1.1.5 Compose and Swarm

In scenarios where multiple containers are being used, either on stacks or clusters, what in Docker terminology is called a swarm, two technologies are offered, Docker Compose and Docker Swarm [40].

Briefly explaining, Compose its used for running stacks of containers, such as an application stack (web server, application server and database) and Swarm allows creating clusters of containers, enabling executing scalable workloads (its considered an orchestration tool) [40].

2.1.2 Other Containerization Technologies

There are several technologies that enable the concept of containers. Although the most widely used one is Docker, there are plenty of other products available including [34]:

- *Linux Containers*
- *OpenVZ*

2.1.2.1 Linux Containers (LXC)

LXC, as the name suggests, is a lightweight container based virtualization tool that uses features of Linux *kernel*, mainly *namespaces* and *cgroups* [19]. While *namespaces* allow resource isolation, *cgroups* enable the resource management such as core count, memory limit, disk usage, etc. Since containers within LXC share the *kernel* with the OS, their file system and processes are visible and can be managed from the host OS.

2.1.2.2 OpenVZ

OpenVZ is a open source server automation and virtualization tool developed by *SWsoft* [17]. It allows the creation of various virtual private servers on a specific physical server so that way the hardware can be shared. Virtual private servers, unlike virtual machines run on the same operating system as the host machine.

Just as LXC, OpenVZ uses Linux *kernel* features such as *namespaces* and *cgroups*, in order to allow resource isolation and container resource management.

2.2 Container Orchestration

In recent times, containers have been in the spotlight [35]. They are characterized as standalone self-contained units that package software applications and their dependencies together. Hence, they are being widely used by several organizations to deploy their extensive set of applications such as web services, big data, internet of things, among others. This, in turn, has led to the seek and rise of container orchestration platforms. Designed to handle the deployment of containerized applications in large-scale clusters or servers, these type of systems are capable of running a tremendous amount of jobs across lots of machines.

Container orchestration systems allow the deployment of containerized applications on a shared cluster. They enable these applications execution and monitoring by transparently managing tasks and data deployed on a set of distributed resources.

Each application type has its own well defined characteristics and requirements such as high availability long-running jobs, deadline-constrained batch jobs, or latency-sensitive jobs. Most systems support multitenancy, as for example, scheduling applications belonging to various users on a shared set of compute resources, allowing for better resource

utilization. Thus, as applications are submitted for deployment, the orchestration system must place them as quick as possible on one of the available resources while, at the same time, taking into consideration its particular constraints in order to maximize the utilization of the compute resources and reduce the operational cost of the organization. Also, in some scenarios, these systems must achieve this while handling a considerably large number of compute resources, providing fault tolerance and high availability and promoting a balanced resource allocation [35].

2.2.1 What is Kubernetes

Developed by Google, Kubernetes was made open-source in the summer of 2014 [31]. However, after being handed over, the project is nowadays maintained by Cloud Native Computing Foundation. The name Kubernetes was originated by a Greek word that means *Helmsman*, the person who steers a ship. This association is reflected on the logo, as illustrated by Figure 2.2.

FIGURE 2.2: Kubernetes logo [31]

Still regarding the name, Kubernetes is often referred to as K8S, a shortened version of the name. The logic behind this is that the 8 replaces the number of characters in between the *K* and the *S* [31].

Kubernetes is known as an orchestrator, mainly oriented to containerized applications and micro-services orchestration. Putting it in simple terms, Kubernetes is a tool that enhances the deployment and maintenance processes of applications that are distributed as containers. Scaling, self-healing and load balancing are some of the various features that Kubernetes provides.

When talking about Kubernetes, it is almost mandatory to talk about Docker as well, since they are complimentary technologies. It is very common to see this combination across various projects, where Docker acts as the container run-time and Kubernetes and the container orchestrator. This means that Docker is responsible for starting, stopping or more generically managing the containers and Kubernetes orchestrates a set of host that run containers. In cases like this, Docker is the low-level technology that is orchestrated and managed by Kubernetes [31].

2.2.1.1 Masters and Nodes

A Kubernetes Cluster is composed by masters and nodes [31]. These are nothing more than Linux host running on physical servers or cloud instances.

The master is represented by a set of small services that are responsible for managing the cluster. Usually, all the master services run on a single host and they do not perform application workloads, in order to by entirely focused on managing tasks.

A Kubernetes master is made up by several sub-components that form the control plane of a cluster [31].

API server Represents, metaphorically, the brain of the Kubernetes cluster. It acts like the front-end of the control plane, exposing a *RESTful* API that is used to handle manifest files that later on are validated and deployed to the cluster itself.

Cluster Store Responsible for storing the cluster state and configurations.

Controller Manager Responsible for ensuring that the current state of the cluster is the desired one.

Scheduler Responsible for watching new workloads and assigning them to nodes.

Figure 2.3 illustrates the composition of a Kubernetes master (control plane), including all the sub-components previously mentioned.

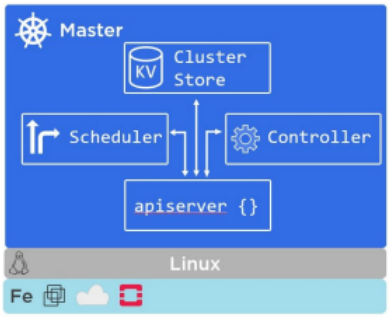

FIGURE 2.3: Kubernetes control plane [31]

On the other hand, the nodes are considerably simpler than masters and they are the ones actually responsible for running the workloads. Nodes are made up by the *kubelet*, *container runtime* and *kube-proxy* [31].

Kubelet Represents the main Kubernetes agent that runs on each cluster node. When installed on a Linux host it registers it as a cluster node.

Container runtime Responsible for performing container related operations, working together with the Kubelet. Docker is the most common container run-time used alongside Kubernetes.

Kube-proxy Responsible for managing and maintaining the network rules inside the node.

Figure 2.4 illustrates the composition of a Kubernetes master (control plane), including all the sub-components previously mentioned.

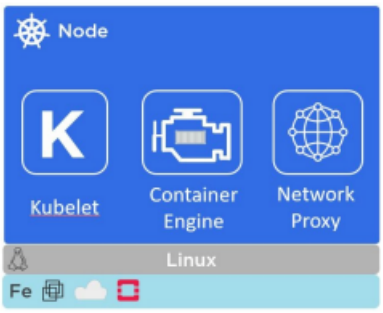

FIGURE 2.4: Kubernetes node [31]

2.2.1.2 Pods

Pods represent the atomic unit in the Kubernetes terminology [31]. Like mentioned before, Kuberentes runs containerized applications, however all the containers always run inside Pods, it is not possible to run a container directly on a Kubernetes cluster.

In the most simple scenarios, a single container is ran inside a Pod, however there are advanced scenarios where it is possible to run multiple containers inside a single Pod. It is important to keep in mind that when scalling down or up is required this a done by removing or adding pods, not by removing or adding the same container inside a Pod. Pods also represent the minimum unit of scaling in Kubernetes.

Regarding its life-cycle, Pods are mortal. They are born, they live and eventually they die. If, for some reason, they die unexpectedly, instead of trying to bring them back to life, Kubernetes starts new Pod instances, that basically are replicas of the ones that died [31].

2.2.1.3 Services

Services are used in order to provide a reliable networking endpoint for a set of Pods [31]. As mentioned before, Pods are mortal and can die. Either manually killed or replaced by new ones when something fatal happens, all these new Pods have totally different Internet Protocol (IP). This represent a problem since Pod IPs are not reliable, and it is where Services come to play a crucial function.

A Service is a Kubernetes object that provides stable Domain Name System (DNS), IP addresses, support to Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) and load-balancing across Pods. This mitigates the problem previously mentioned, enabling Pods to come and go but always keep a stable networking endpoint.

Summarizing Services, they are nothing more than a networking abstraction for multiple Pods that provides basic load balancing among them [31].

2.2.1.4 Deployments

Deployments play a crucial role on the constant need of rolling updates of a certain service by implementation a versioning approach [31]. They are considered the future of Kubernetes application management providing features such as versioning, rolling updates, concurrent releases and simple rollbacks.

Deployments are Representational state transfer (REST) objects in the Kubernetes API, which means that they can be defined in "Yet Another Markup Language (YAML)" or JavaScript Object Notation (JSON) manifest files and sent directly to the API server.

2.2.2 Minikube

Minikube is a single node lightweight Kubernetes cluster designed to ease the creation of learning environments. With minikube testing a Kubernetes setup on a local machine is a very straightforward and simple process [16].

To do so, minikube creates a virtual machine on the target host and then deploys there the single node Kubernetes cluster where it is possible to run container and pods.

2.2.3 Other Orchestration Technologies

Despite Kubernetes being the most popular tool used for Orchestration, there are plenty of other tools to do so.

On the Orchestration side, there are other tools such as [1]:

- *Docker Swarm*
- *Apache Mesos*
- *Cattle*

2.2.3.1 Docker Swarm

Docker Swarm is a clustering and scheduling tool for Docker containers [1]. It allows IT operators to manage a cluster of Docker nodes as a single system. This is an important aspect because it creates a cooperative group of machines that provide redundancy and enable the fail-over mechanism if one or more nodes experience an outage. The orchestrator is based on the master / slave model for which master dominate.

The master (known as Manager) is the node that is responsible for scheduling containers, whereas a slave (commonly known as Agent) is responsible for launching received containers. To connect containers hosted on different nodes under the same network, Docker Swarm offers an overlay network which exploits the Virtual Extensible Local Area Network (VXLAN) tunnel for creating a virtual network among hosts. The Manager node keeps tracks of the state of all nodes in a cluster(in the context of Docker Swarm this service is called discovery and is based on heartbeat mechanism that overlay networking module uses in determining whether a Docker daemon on a remote host in a cluster is still functioning). In cloud environments, elasticity is an important feature, Docker Swarm allows the fine-grained scalability of part of the service scaling one or more replicated services either up or down to the desired number of replicas [1].

2.2.3.2 Apache Mesos

Apache Mesos is a seminal open source project developed by the University of California, Berkeley [1]. The architecture consists of a master/slave design pattern in which the execution of tasks is delegated to slave nodes. The master process, running on a manager node of the cluster is responsible for managing and monitoring the whole cluster architecture. Therefore, it communicates with frameworks that aim at scheduling jobs on slave nodes. Mesos solutions are often developed by installing on top of a Mesos cluster an application-level management called Marathon. Marathon interacts with the master component providing orchestration functionalities to the whole Mesos cluster. Thus, in the case of slave faults, Marathon starts a new instance to guarantee the fault-tolerance. Mesos offers high-availability replicating master nodes in order to provide failover mechanisms in case of master failures. To achieve this, it depends on Apache Zookeeper that consists of an election algorithm which elects a new node to play a master role [1].

2.2.3.3 Cattle

Cattle is an orchestration engine powered by Rancher which is extensively exploited by Rancher users for creating and managing applications based on Docker containers [1]. One of the key reasons for its extensive adoption is its compatibility with standard Docker YAML file (also known as docker-compose) syntax and Docker commands. The architecture is based on a master / slave architectural pattern and the application deployment is based on the concept of “stack”. Each stack is a composition of “services” which are primarily docker images, characterized by application requirements such as scaling, health checks, service discovery links and configuration parameters [1].

2.2.3.4 Technologies Comparison

The orchestration engine structure across all platforms sits atop and consists of three layers [1]:

- *Resource Management*
- *Scheduling Management*
- *Service Management*

The resource management layer manages low-level resources such as memory, Central Processing Unit (CPU) / Graphics Processing Unit (GPU), disk space, volumes,

persistent volumes, port and IP [1]. It aims at maximizing utilization and minimizing interference between containers competing for resources. Table 2.1 reports the resources supported by the each one of the different alternatives.

Table 2.1: Resource Management Layer Comparison [1]

✓ (yes) P (partial)	Swarm	Mesos	Cattle	Kubernetes
Memory	✓	✓	✓	✓
CPU	✓	✓	✓	✓
GPU	-	P -	-	-
Disk Space	-	✓	-	-
Volume	✓	✓	-	✓
Persistence Volume	-	P	-	P
Port	✓	✓	✓	✓
IP	P	P	-	P

The scheduling layer targets at using cluster resources in an efficient manner [1]. It typically receives user-supplied indications as input and then decides how to place all containers. As stated on Table 2.2 this layer includes: placement (to directly control the scheduling decisions), replication/scaling (to express the number of microservice replicas), readiness checking (to include a container only when it is ready to answer), resurrection (to reenact fast long-lived processes whose job requires being always up and running; rescheduling to automatically restart and schedule crashed containers running on a failed node), rolling deployment (to automatically up/down-grades the application version) and co-location (to assert deployment constraints, such as to co-locate containers so to take advantage of local inter-process communication) [1].

Table 2.2: Scheduling Layer Comparison [1]

✓ (yes) P (partial)	Swarm	Mesos	Cattle	Kubernetes
Placement	✓	✓	✓	✓
Replication/ Scaling	✓	✓	-	✓
Readiness checking	-	✓	✓	✓
Resurrection	✓	-	✓	✓
Rescheduling	✓	✓	-	✓
Rolling Deployment	-	✓	✓	✓
Co-location	-	-	-	✓

Finally, the service management layer provides functional capabilities for building and deploying complex solutions [1]. As described on Table 2.3, it manages high-level aspects such as labels to attach metadata to container objects, groups / namespaces to isolate containers and support multitenancy, dependencies to express dependencies between microservices, load-balancing to divide incoming load and readiness checking to make the application available online only when it is ready to accept incoming traffic [1].

Table 2.3: Service Layer Comparison [1]

✓ (yes) P (partial)	Swarm	Mesos	Cattle	Kubernetes
Labels	✓	✓	✓	✓
Groups / Namespaces	-	✓	-	✓
Dependencies	-	✓	-	-
Load Balancing	-	P	✓	✓
Readiness checking	-	✓	-	✓

2.3 Monolithic Architecture

The monolithic architectural style has been widely used over the years in software development [2]. The term *monolithic* [1] is not new and as its definition suggests, alludes to the concept of uniqueness. In a software development perspective it means that all the components that composes an application such as user interface, business logic and database access, are placed together turning the application into a self-contained piece of software. However, saying that the software is self-contained also means that the various components that compose it are interconnected and interdependent. Therefore, a simple code change may affect the whole system, possibly breaking a major functionality in other modules, implying a constant test process over the entire system [2].

Implementing a system using a monolithic architectural style can be very demanding, whereas various challenges may urge during the development process [36].

Code complexity Usually, every time that a monolith grows its code complexity increases and the overall quality declines. This inevitably adds complexity to code understating and navigation, making it more favorable for security bugs and vulnerabilities to happen.

Deployment Complexity Some scenarios where a small code change is needed may result in the deployment of the entire system.

Dependency Hell Having a large number of dependencies makes the maintenance process extremely challenging. When updating a certain component, that component is most likely to directly or indirectly be a dependency of another part of the system, which can lead to an inconsistent system that does not behave as intended.

Scalability The various components of a monolith might have different request inbound than others. However to handle a situation where the traffic volume is higher, the entire application needs to be replicated, which results in resources wasting, thus directly affecting the system scalability.

Technology lock-in Due to the high coupling of the components, adopting new technologies in a monolith is an extremely hard and time-consuming tasks, being sometimes even not possible at all.

2.4 Microservices Architecture

The microservice architectural style began its rise in recent years and consists in a way of developing a single application that is composed by several smaller services [2]. These smaller services are distinct processes that are intended to be completely independent from each other, meaning that they can be developed and deployed independently. This

[1] According to its definition, *monolithic* stands for "(of an organization or system) large, powerful, indivisible, and slow to change".

is the same as saying that microservices are a way of segregating services so that they can be handled without any dependencies from other services in the context of design, development, deployment and upgrades [2].

The microservice architecture provides the necessary means for surpassing the drawbacks associated with a monolithic architecture, earlier explained. Also, in order to enhance the benefits and strengths of microservices, several patterns were developed over the years [36].

Independence Since microservices have well-defined boundaries, they are both deploy and operational independent. Therefore, the risk of causing side-effects when updating a certain service is drastically lowered.

Gradual roll-outs Whenever there is a new version of a certain microservice, it can be rolled out gradually in parallel with other existing versions. This way, the remaining microservices can gradually move their flow to the newer version.

Swift learning Being isolated pieces, when a change is required in a certain microservice it is not required to have a deep knowledge of the overall system implementation in order to perform the change.

Scalability Having different microservices handling separated parts of a complex system that may have different scaling requirements and request inbounds, allows scaling only the strictly necessary services in order to mitigate a possible higher request inbound volume. Instead of adding and removing instances of the entire application, it is possible to do that only for the microservices that are under heavy load.

No technology lock-in Since microservices are independent pieces of the application, developers are free to adopt a different technological stack when needed / required. The only constraint imposed when replacing a microservice is to maintain its interface contract, so that the order services do not notice any difference.

2.4.1 Decompose by Sub-domain Pattern

Domain Driven Design (DDD), as stated by Eric Evans on the book Domain-driven design [33], is an approach that helps building complex software systems, focused on the development of a object-oriented domain model. The model model is used in order to capture relevant nuances and particularities regarding the domain of the problem, defining a standardize vocabulary used by the team.

When using a microservice architectural style, DDD offers two core concepts that help alongside the process, sub-domains and bounded contexts.

DDD, instead of focusing on the development of a global domain model aims at developing multiple domain models, each one with an explicit sub-domain. Each sub-domain represent a part of the application's problem space and they can be identified by analyzing the business and identify the various areas of expertise [33].

On the DDD nomenclature, a bounded context represents the scope of a domain model. Each bounded context, when on the development phase, is most likely to represent a single or various services.

2.4.2 Asynchronous Messaging Pattern

When a messaging approach is implemented, services tend to communicate asynchronously with each other [33]. To do so, message-based applications rely on a *message broker*, that acts as a man-in-the-middle of the various services. Every time a client service wants to make a request, it just sends a message to the *message broker*, being the response also placed there when available. Because the communication process is asynchronously, the

client does not block waiting for a response. Instead, the client is already aware that the reply will not come right away.

2.4.3 API Gateway Pattern

An API gateway is a service that acts as the entry-point of an application from the external world [33]. Rather than holding business logic, the API gateway is responsible for tasks such as request routing, authentication, API composition, among many others. It represents an abstraction layer between external applications and the various microservices under a certain system.

One of the main advantages of using the API gateway pattern is that it encapsulates the internal structure of the application, avoiding its exposure to external systems. Rather than communicating directly to a specific service, clients always talk to the API gateway [33].

2.5 Spring Boot

The Spring framework is the most popular and widely used Java based framework [32], either for web and enterprise applications. It provides a rich ecosystem of projects addressing modern applications needs such as security, simplified access to both relational and non-relational databases, batch processing and integration with external networks. Since Spring is a very customizable and flexible framework, usually one application can be configured in multiple ways.

The main goal of the Spring framework [32] is to allow developers to quickly create Spring based applications without having to write all the boilerplate configuration that usually would be necessary.

The microservices architecture, in recent years, has become the elected architecture style for building large and complex application, being Spring Boot a great choice for building those types of application by taking adavantage of the various Spring Cloud modules [32].

2.6 Version Control Systems

According to its definition a Version Control System(s) (VCS(s)) is a specialized tool that enables developing teams to manage their source code as well as keeping track of the code development history [26].

Right now, there are two variants of these systems [15]:

- *Centralized Version Control Systems (CVCS)*
- *Distributed Version Control Systems (DVCS)*

Centralized version control systems lay on a client / server approach where users work with code that's on a single central repository. These types of systems require either merging or locking in order to handle the synchronization [15].

Due to its nature, centralized systems face the following challenges [15]:

- A network connection is required in order to work on the source code
- A single point of failure is an issue when using only one centralized server

Distributed version control systems allow users to work locally on a copy of a certain repository. The changes made can later be merged and pushed to the master repository. One of the biggest advantage of distributed version control systems is the fact that they enable users to work offline, which enhances flexibility [15].

Even though distributed systems have a smaller range of disadvantages than centralized systems, one of the most common complains is the fact that pessimistic locks are not available [15].

2.6.1 Background of Centralized and Distributed Systems

The concept of version control began as a local source code management activity, where a mainframe computer was connected to multiple terminals. To perform version control, this approach kept all the needed information on the source code itself and locking was used in order to avoid concurrent changes that could possibly interfere with the integrity of the system [15].

As time passed by and personal computers became more popular, version control systems had to evolve and become more sophisticated. Instead of using a local machine, these systems started to use a centralized server to perform the intended actions, being Concurrent Versions System the first centralized version control system as we see them today. Due to the lack of other alternatives, CVS was the standard system to use for a long time, however as time passed by a new and sharpened system appeared and became the new first choice for version control, the Subversion [15].

Despite of the advances that Subversion brought, a new paradigm for version control arrived when BitKeeper was released. BitKeeper was a distributed version control system that allowed developers to have their own working copy of a certain repository locally, on their machines. Even though BitKeeper was used on the early phases of Linux, at a certain point in time, due to antagonism of ideals, Linus Torvalds, the father of Linux, decided to create a new version control system to replace it. That marked the born and rise of Git, a distributed version control system that gained a huge popularity over the years. Also, at the time Git appeared, another distributed version control system was born, Mercurial. Even if not as popular as Git, Mercurial established its position among the users [15].

2.7 DevOps Methodology

Throughout the years software development methodologies have suffered an immense evolution, causing the urge of new techniques to enhance continuous delivery, client partnership, client satisfaction, communication with stakeholder, among many other factors [6]. To fulfil these needs, for a long period of time, the agile methodology was the way to go. However, in more recent years, a new methodology called DevOps was introduced. This new methodology aimed at strengthen the collaboration between the development team and the operations team, therefore mitigating unwanted constraints for product delivery, putting into action agile and lean concepts.

DevOps, in its core, its a software development technique that focus on bring closer IT professionals by prioritizing communication and collaboration thus, as mentioned before, allow products to be quickly delivered and distributed [6]. Also as mentioned before, this bound is focused specially on the development and operations teams, enabling an higher flexibility and accuracy as well as bringing more automation ideal to software, making it easier to manage complex problems and producing robust solutions. DevOps also aims at a regular interaction with business in order to keep track of its requirements, that might change over time. The main routines that DevOps targets are intensive cycles of testing, integration, development and deployment.

2.7.1 DevOps versus Agile

Since the DevOps methodology derives from some Agile principles, Table 2.4 shows what differentiates them, in order to get a clear understanding on what DevOps really is [6]:

Table 2.4: DevOps versus Agile Comparison

Characteristic	DevOps	Agile
Principles	Cross-Functional teams; Feedback loop; Automation; Flexible Scope;	Iterative sprint cycles; Assumes business requires may change; Close relationship between business, developers and testers; Flexible Scope;
Automation Level	High	Varies
Business Ownership of Project?	Yes	Yes
Response to Changes	Highly responsive-cross functional teams define business needs more precisely	Iterative delivery enables prioritization
Quality	Highly automated unit tests during development	Issues are identified at the end of each sprint
Risk	Decreases as product evolves	Decreases as product evolves
Customer Feedback	Continuous	After every sprint
Communication	Specs and design documents are a common communication method	Scrum is the most common method for Agile and all its ceremonies
Main Tools Used	Puppet, Chef, TeamCity ,OpenStack, AWS, among others	JIRA, Bugzilla, Kanboard, among others

2.8 What is CI / CD

In one hand, Continuous Integration (CI) is a software development practice that aims at constantly integrate and test changes that are made to a system while as they are being developed [25].

Having this integration phase approach is very valuable for the different people inside a development team since each individual is able to work on its own peace of the code of the project in isolation. The various pieces are only merged and properly tested once they are fully completed, whereby the idea that supports this is that the glue between the pieces should be defined well enough so that unite them raises no problem.

With Continuous Integration its usual to see developers commit small changes several times per day. Every time a change is committed, the Continuous Integration server pulls that change and runs tests in order to ensure, as the name indicates, if its everything stable when integrating that new piece.

If any errors occur during the test execution in the Continuous Integration phase a trigger is automatically pulled that stops the situation (often called a broken or red build). At this point nobody in the team should make any additional changes since the priority is to fix the so called broken build. If the person who triggered the failing build is not able to fix it in a viable period of time, the change should be reverted in the version control system so that the Continuous Integration job can run successfully again.

On the other hand, Continuous Delivery (CD) can be considered as the next step of Continuous Integration. While Continuous Integration integrates every commit that is made and runs tests to ensure that everything stays stable, Continuous Delivery ensures that the system as a whole is production ready on every commit. The changes that are applied to another environment other than production use the exact same processes and tools that are used the production environment. The benefit of making this a continuous process is that while performing changes, releasing them out to either production or not its a trivial task [25].

One big misconception that's often made is that Continuous Delivery means that every change is directly applied to production after passing the defined checklist of tests and other requirements (Continuous Deployment). Even tough this is a reality is some organizations, in the majority of them this doesn't apply. The main objective of Continuous Delivery is not to apply every change immediately but making sure that the change could be applied if wanted.

Continuous Delivery enables teams to decide of whether and when to apply a certain change to production, making them only depending on a business decision to proceed with the change to production or not. Also, the act of rolling out a change to production doesn't require the entire team to stop the development, having a project plan, or even having a maintenance window, its a process that just occurs since it was already multiple times performed and validated in testing environments [25].

2.8.1 CI / CD Tools

Nowadays, to fulfill the needs that teams may face, there are several tools that enhance the application of Continuous Integration and Continuous Delivery to projects. Some of the most popular tools are [8]:

- *Jenkins*
- *GitLab CI*
- *Bamboo*
- *TeamCity*

2.8.1.1 Jenkins

Jenkins, originally known as Hudson, is an open source Continuous Integration tool written in Java and highly extensible due to its plugin oriented architecture. Considered a leading platform in the market, Jenkins is widely used by various teams with different dimensions and characteristics, supporting a vast range of languages and technologies, including .NET, Ruby, Groovy, Grails, PHP, Java, among others [37].

Part of Jenkins popularity comes from its easy usage, having a simple user interface, intuitive and visual appealing, combined with the fact that its has a very low learning curve.

Besides being powerful and extensible, Jenkins is also very flexible, which makes it easily adaptable to various purposes. There are hundreds of open source plugins available and they provide cover everything from version control systems, build tools, code quality metrics, build notifiers, integration with external systems and much more.

Another crucial factor for Jenkins popularity comes from its community. Jenkins has a large and dynamic community. This results in a fast development pace, providing new releases almost weekly, targeting new features, bug fixes and plugin updates [37].

2.8.1.1.1 Build Jobs Build jobs are considered the standard currency of a Continuous Integration server [37].

A build job is a process that may include the compilation, testing, packaging, deployment or any other activity within a certain software project. Build jobs are highly configurable and can have multiple forms, from compiling and testing the code to packing it code and deploy it to production, the involved steps in a build job cover a wide range of possibilities.

2.8.1.1.2 Security Jenkins supports several security models and allows the integration with various user repositories. Despite security in Jenkins in smaller companies may not represent a large concern, since the only need is to prevent unidentified users to access the build jobs, in larger companies where there are stricter rules and only members with a certain role can modify the build jobs, usually, other requirements are needed. To fulfil these various requirements and possibilities Jenkins offers different approaches on how to implement security [37].

- Using Jenkins Built-in User Database
- Using an Lightweight Directory Access Protocol (LDAP) Repository
- Using Microsoft Active Directory
- Using Unix Users and Groups
- Integration with Other Systems

2.8.1.2 GitLab CI / CD

GitLab CI, offered by GitLab, is a CI tool that, with the help of scripts, enable building and testing automatically an application every time a commit is made. GitLab CD, also offered by GitLab, is a CD tool that after a certain application is built and tested, it enables it to be continuously deployed. If Continuous Delivery is being used, the deploy is a manual task. On the other hand, if Continuous Deployment is being used, the deployment is an automatic process, without Human intervention required [10].

2.8.1.3 Bamboo

Developed by Atlassian, Bamboo is a CI / CD tool that offers resilience, reliability and scalibility. Also, another Bamboo's strength is the out of the box integration with Atlassian services such as Bitbucket, Jira and Opsgenie [3].

2.8.1.4 TeamCity

TeamCity is a Java based CI / CD tool managed by JetBrains. TeamCity is a commercial product but the licensing allows to get started with it for free. Also, TeamCity offers a wide variety of plugins, either developed by JetBrains and its community, first-class support for various technologies and support for a vast range of version control systems such as Git, Mercurial and Subversion [22].

2.8.2 Infrastructure as Code

Over the last years Operations teams have been making use of scripts and other tools in order to fulfill the required day-to-day routines associated with the infrastructure maintenance. However, currently, with the increase of infrastructure complexity and the evolution of technology, automation is a must when is comes to infrastructure management [25].

In order to leverage the process of infrastructure maintenance, as mentioned before, automated configuration management tools become cutting edge technology over the past decade. CFengine was considered the *father* of these new line of technologies while other tools like Puppet and Ansible carried its legacy. These tools brought a whole different perspective on how infrastructure management was performed, allowing

computing resources allocation and configuration to be centralized and captured in files that could be easily maintained under source control systems.

Also, one massive advantage brought by these tools was the fact that software engineering practices and conventions such as Test Driven Development (TDD), CI, CD, among others, could be applied to the infrastructure itself.

So, in reality what is Infrastructure as Code? Infrastructure as code is an approach to combine cutting edge cloud technologies to dynamically managing infrastructure. The goal is to take care of infrastructure and the tools that help in its management as if they were part of a software system, using software engineering patterns and practices to manage any possible changes. The result of this in a robust infrastructure that's clear and well tested, making it easier to perform operation tasks and needed changes.

Infrastructure as Code lays its foundation on the capability of treating infrastructure elements as they were data, helping decoupling infrastructure from its hardware components by providing a programmatic interface to manage storage, servers, networks, among other components [25].

Summarizing Infrastructure as Code as a whole, its core objectives and guide lines are [25]:

- infrastructure should support and enable change, not representing an obstacle when modifications are needed
- teams should be able to provision and manage resources according their needs, without needing specialized people to do that for them
- whenever there are changes to the system, they should be easy and clear to perform, without causing pressure on teams
- rather than constantly trying to avoid failure, teams should be comfortable and able to recover from failure with ease
- solutions to problems, rather than be discussed in documents and meetings should be addressed via a possible implementation that has been tested and measured

2.9 Code Quality

Code Quality can be seen as a set of specifications and goals that needs to be fulfilled in order to meet the specified requirements of a certain system, thus its a subjective concept since different teams can have different definitions of quality depending on the context. In whatever form we see quality, its a crucial aspect in every system. Such as time and cost, quality is one of the main factors that determine the success, or not, of any software project. Besides, the effects of a software with bad code quality can be disastrous [38].

2.9.1 Code Analysis Tools

In order to track the quality of a certain system continuously, its crucial to use strategies that can easily and quickly retrieve the needed metrics and measures, otherwise constantly making the code assessment manually would be very costly [30]. Those desired strategies exist in the form of tools. These tools perform an in dept code static analysis and report their findings in a direct and clear way. Static analysis aims at evaluating a certain system's code using programming language specific rules and conventions, quality characteristics and paradigms, resulting in a possible list of the so called bugs and code smells [7].

Some code analysis tools that are widely used nowadays are [9]:

- *SonarQube*

- *Coverity*
- *Codacy*

2.9.1.1 SonarQube

SonarQube is one of the most used open source static code analysis tools nowadays, being frequently adopted in both academic and professional contexts [20]. SonarQube supports several programming languages but its especially known for being the best Java analyzer. Also, it allows the integration with CI / CD tools such as Jenkins, TeamCity and Bamboo as well as repository management services such as GitLab, Bitbucket and GitHub. It can be accessed by Sonar own server or downloaded and executed elsewhere. SonarQube performs the calculation of various metrics and verifies, taking into consideration a set of rules that can be adjusted, if the code is compliant with them or not. Every time a code violation is detected or a metrics is beyond the defined threshold, SonarQube raises an issue that can addressed over three main scopes [20]: *Reliability*, *Maintainability* and *Security*.

The reliability rules, also known as *bugs*, represent code violations [20]. On the other hand, maintainability related issues are known as *code smells*, that is the same as saying portions of code that makes it harder to read and modify. Finally the security issues, as the name indicates, are related to breaches in the code that can compromise it. SonarQube, apart from creating different types of issues, also categorizes them into severity levels, comprehending five levels [20]: *Blocker*, *Critical*, *Major*, *Minor* and *Info*.

The first two levels, Blocker and Critical, have the highest chance of impacting negatively the system or the developer's productivity. Due to its severity, SonarQube advices to fix these issues as soon as possible [24].

Major issues are more focused on the developers perspective since they can highly lower their productivity. Minor issues follow the same pattern of the major issues but the impact they cause on developers is lighter. Last but not least, the info level issues act like a fallback, representing all the issues that are neither bugs nor code smells. All the issues go through a life cycle that is internally managed by SonarQube. When they first are created by SonarQube, they are marked as open and after that can only follow one of two possible paths, either are manually closed (resolved) or fixed and acknowledged by SonarQube (closed) [24].

2.9.1.2 Coverity

Coverity is a product offered by Synopsys that aims at performing static code analysis. It enables detecting flaws and security issues in various programming languages. Also, these code analysis are made incrementally in the background giving developers real time feedback and results directly on the Integrated Development Environment [5].

2.9.1.3 Codacy

Codacy is an automated code review platform for that supports static code analysis for several programming languages. It allows the integration with other tools such as GitLab, Bitbucket, GitHub, among others, as well as selecting the desired rules to be applied and even defining totally new customized rules [39].

2.10 Robotic Process Automation

The Robotic Process Automation (RPA) is a recent approach that aims at automating business processes (Figure 2.5), avoiding users to repeating tasks that require manual input over and over again [41]. One of the biggest advantages that the RPA framework

brings is the fact that the communication with other applications is not made through the API but directly through the user interface, besides being very fast to implement.

FIGURE 2.5: Business processes that can be automated with RPA [41]

RPA significantly helps to reduce costs and increases the overall operational performance. Also, it is a fact that software robots, when properly configured, are almost certain to perform their tasks without any errors [41].

2.11 Selenium Web Driver

The Selenium Web Driver is the successor of Selenium Remote Control (RC), that in the meantime was officially deprecated [11]. Selenium Web Driver accepts input commands and sends them to a supported browser. Each browser supported by Selenium has a specific browser driver implementation. Breaking it down into steps, Selenium works according to the following sequence [11]:

- The driven listen to input commands from Selenium
- The received commands are converted into the specific browser native API
- The driver gathers the result of the command and sends it back to Selenium

Figure 2.6 illustrates the process previously mentioned.

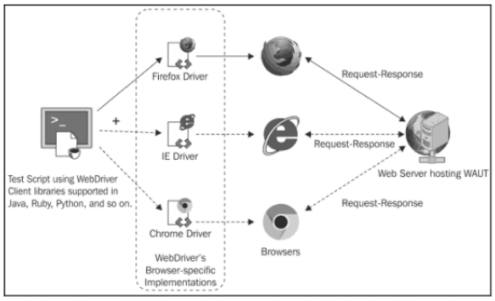

FIGURE 2.6: Selenium flow breakdown [11]

Having that said, Selenium can be used to create browser-based regression automation, scale and distribute scripts across browser, among other browser automation tasks [11].

2.12 Innovation Process

The innovation process can be divided into three distinct areas [18], following presented and illustrated on Figure 2.7.

- Fuzzy Front-end (FEE)
- New Product Development (NPD)
- Commercialization

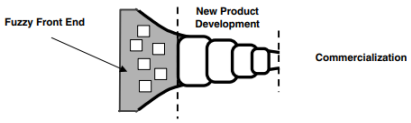

FIGURE 2.7: Innovation process [18]

The first area, the fuzzy front-end, usually is considered one of the main opportunities to improve the overall innovation process and includes the activities before the product development [18].

On the second part, the new product development, the ideas originated on the previous section are implemented. It is considered a very formal and structured process, in order to increase the value and success probability of high-profit concepts entering product development and commercialization phases. The main differences between fuzzy front-end and the new product development can be seen on Table 2.5 [18].

TABLE 2.5: Difference Between the Fuzzy Front End and the New Product Development Process [18]

	Fuzzy Front-end	**New Product Development**
Nature of Work	Experimental and usually chaotic	Organized and goal-driven with a defined plan
Commercialization Date	Unpredictable	Almost certainly
Funding	Variable	Budgeted
Revenue Expectations	Uncertain with lots of speculation	Predictable with increasing certainty as the product release gets closer
Activity	The research conducted by individuals and team aims at minimize risk and optimize potential	Multi-function product development team
Measures of Progress	Strengthened concepts	Milestone achievements

At last, the commercialization part, is when the product resulting from the development is taken to the market [18].

2.12.1 New Concept Development Model

Aiming to provide a common language and definition of the key concepts of the fuzzy front-end the New Concept Development (NCD) model was proposed [18], Figure 2.8 .

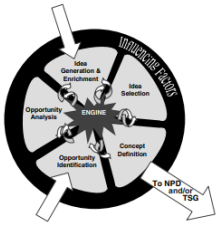

FIGURE 2.8: New concept development [18]

The new concept development model is composed by three key parts, presented below [18].

- The engine, that symbolizes the leadership, culture and business strategy of an organization that drives the five key elements that are controllable by the corporation.
- The inner areas, that define the five controllable elements (opportunity identification, opportunity analysis, idea generation and enrichment, idea selection, and concept definition) of the fuzzy front-end.
- The influencing factors, that basically are the organizational capabilities, the outer world and the enabling sciences that may be involved. These factors are considered uncontrollable by the corporation.

2.12.1.1 Opportunity Identification

This phase is usually where an organization identifies the available opportunities that it might be interested on pursuing. Both technological and business opportunities are taken into consideration. An opportunity can reflect an entirely new route for the business or simply an upgrade to an existing product. The essence of opportunity identification is the identification of a market or technology arena that a company may want to be part of [18].

2.12.1.2 Opportunity Analysis

In the opportunity analysis phase, an opportunity is assessed in order to validate is that is worth pursuing it. Usually additional information is needed so that the opportunity identification can be broken down into specific business and technology opportunities [18].

2.12.1.3 Idea Generation and Enrichment

The idea generation and enrichment phase is where the birth, development and maturation of a concrete idea happens, thus being considered an evolutionary process. The ideas, throughout their life-cycle, are built up, torn down, combined, reshaped, modified and upgraded [18].

2.12.1.4 Idea Selection

In most of the cases, organizations do not have any problems coming up with new ideas, however they tend to struggle in the idea selection phase, where they have to choose what ideas to purse in order to achieve the most business value. Selecting good ideas is critical to the success of an organization. The idea selection process involves various iterative activities that tend to go through multiple steps from opportunity identification, opportunity analysis, and idea generation and enrichment, several times [18].

2.12.1.5 Concept Definition

Concept definition represents the final element of the new concept development model and it is the only exit to the new product development. It requires the development of a business case that can be based on several factors such as market or customer needs, commercial and technical risk factors and size of opportunity [18].

2.13 Analytic Hierarchy Proces

Analytic Hierarchy Process (AHP) was developed by Dr. Thomas Saaty in 1980 as a tool to help solving technical problems [29].

AHP aims at quantifying relative priorities for a given set of alternatives on a scale driven approach. It deconstructs a complex problem to the level of pairwise comparisons and merges the results systematically.

They key steps involved on the AHP are the following:

- Analysis of overall problem in an hierarchical way, in order to dividing it into sub-problems that are easy to understand
- Assign priorities to elements on each level of decision making within the hierarchy
- Assign numerical values to the various elements
- Analysis and evaluation of possible problem solutions

2.14 Value Proposition

The value proposition can be seen as a global vision over a a bundle of products and services that create value for a specific Customer Segment. The value proposition also describes the reasons why the costumers prefer one company over another, since it provides the factors that distinguish one company from the others [27].

2.14.1 Value Proposition Canvas

In order to enhance the value of a certain product, two key factors should be taken into consideration [28]:

Customer Profile Describes a certain customer segment in a more structured and detailed way. It also breaks the customer into jobs, pains and gains. Figure 2.9 presents the customer profile and its divisions.

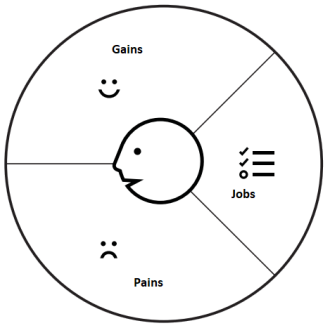

FIGURE 2.9: Customer profile [28]

Gains Describe the outcomes and benefits that the customers are seeking. Gains can be functional, social and emotional. [28].

Pains Describe anything that can annoy and bother the customer, before, during and after trying to get the job done. Also, pains describe risks and bad outcomes. [28].

Jobs Describe what customers are trying to get done in their work or live. It can be a task to complete, a problem to solve or a need to fulfil [28].

Value Map Describes the features of a certain value proposition in a more structured and detailed way. It also breaks the value proposition into products, services, pain relievers and gain creators. Figure 2.10 presents the value map and its divisions.

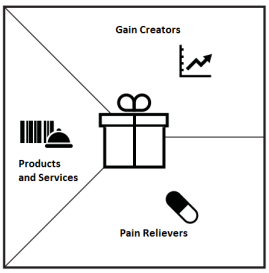

FIGURE 2.10: Value map [28]

Gain Creators Describe how a set of products and services create customer gains. Gains creators explicitly outline how the outcomes and benefits that the customer expects are going to be produced [28].

Pain Relievers Describe how the products and services alleviate certain customer pains. Pain relievers outline how the things that annoy and bother the customers are intended to be eliminated [28].

Products and Services Describe the various items that the business has to offer [28].

2.15 Requirements Engineering

In order to understand the concept of Requirements Engineering, first it's necessary to understand what is a requirement. Looking at it in a simple perspective a requirement can be seen as "something which a costumer needs". However, this definition is not suitable to all situations. There are several ways to define a requirement, but the most notable is Institute of Electrical and Electronics Engineers Standard 610 (1990) that states the following [21]:

1. A certain need or condition required by a user to either solve a problem or achieve an objective
2. A condition or capability that must exist in a certain system to satisfy a standard or other formal documents
3. A documented representation of a condition

As previously mentioned, these requirements are often written down in documents that summarize the requirements stage.

Having that said, if the requirements document can be seen as the end goal of the requirements stage, requirements engineering can be seen as the process of elaborating that document [21].

2.16 Summary

This chapter focused on a study about the concepts inherent to the problem under analysis. In this way, it allowed to deepen the knowledge about both technical and theoretical topics in order to support the rest of the work.

Once concluded the study, the next logical step is to analyze the problem in order to better shape it.

Chapter 3

Analysis

This chapter presents both value and requirements engineering analysis regarding the desired solution.

In order to understand some of the concepts that are going to be next presented, Chapter 2 provides a detailed walk-through about them (value analysis 2.12, 2.14 and requirements engineering 2.15).

3.1 Value Analysis

By using Peter Koen, New Concept Development (NCD) 2.12, it is possible to turn an opportunity into a concept, following all the key steps described on the previously explained model 2.12.1.

3.1.1 Opportunity Identification

As verified in studies previously done, it is possible to state that software maintenance and evolution are very time consuming processes [13].

Whenever a new project needs to be started, there are configuration tasks that need to be performed and, in some scenarios, they are manually performed. This is a very common issue in technological academic environments. Usually, every time the students are allocated into different groups, there is the need of configuring for each group a set of tools in order to provide the students with an working environment. This environments, despite most of the times being replicas of a certain template, they are one by one manually assigned and configured , which leads to a time-consuming and error-prone process.

In order to better understand the difficulties faced in real academic contexts and the needs that are felt, a survey was conducted among Instituto Superior de Engenharia do Porto (ISEP) teachers, who are usually responsible for performing manual configuration tasks, whenever necessary for students to do their work.

The survey, available in its entirety on appendix B, is composed by three multiple choice questions and one with open answer. Note that the survey focused on a specific course unit of the computer engineering degree, Laboratory / Project (LAPR).

Question 1: Do you think that configuring and integrating the tools required for the various student groups is a very time-consuming task? (Take into consideration the Laboratory / Project (LAPR) context)

Figure 3.1 presents the responses obtained for the first question of the survey.

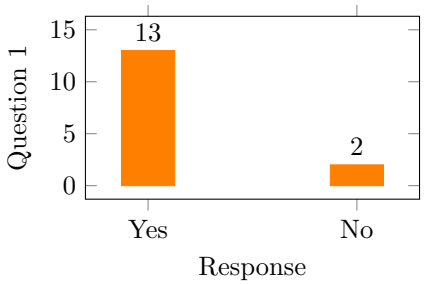

FIGURE 3.1: Results of Question 1

Question 2: Being the current process manual, performing a cascade change (move a student from one group to another), meaning that a modification in one tool would have to be reflected across all the other tools as well, is a demanding and time-consuming task, sometimes error-prone?

Figure 3.2 presents the responses obtained for the second question of the survey.

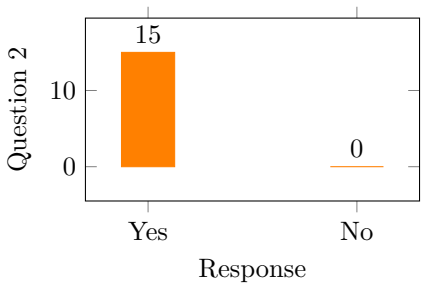

FIGURE 3.2: Results of Question 2

Question 3: Do you think that there would be immediate benefits if the teachers had a system that could automate the integration and configuration of tools that are required for the students work? (Take into consideration the Laboratory / Project (LAPR) context)

Figure 3.3 presents the responses obtained for the third question of the survey.

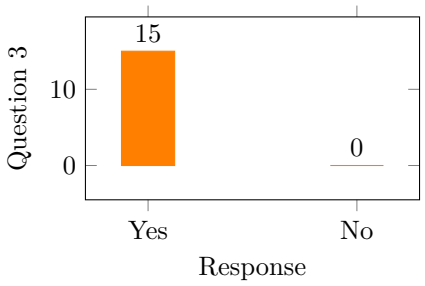

FIGURE 3.3: Results of Question 3

Question 4 What are the most commonly experienced problems when setting the needed tools for students? (Take into consideration the Laboratory / Project (LAPR) context)

Some of the most common problems faced by the teachers are the following:

- The manual configuration is prone to errors.
- Time consuming.
- Setting up project's particularities on each different platform.
- Time, internet connection, software and hardware installation and compatibility, licenses, etc.

Considering the results of the survey, urges an opportunity to implement a service that automates the configuration of various tools, with a special focus on academic environments.

3.1.2 Opportunity Analysis

The concept of automation is turning into a common path of the software development environment. Nowadays, more and more people are seeking to automate repetitive tasks in order to save time and avoid errors.

Specially on academic environments, those are crucial factors that can have an high impact when wrongly managed. Thinking on an academic environment, it is known that sometimes teachers have difficulties configuring the needed tools for students to be able to do their tasks. This happens because its either a time-consuming process, being sometimes hard to allocate time to do it, or just due to the constant changes happening on the tools, which makes it challenging to maintain them.

Having that said, it makes total sense to embrace the opportunity of creating an automation system that can help mitigating real problems that are currently being faced with an elevated frequency.

3.1.3 Idea Generation and Enrichment

After analysing and reflecting over the opportunity, the following ideas emerged:

- Develop a system which facilitates the configuration of a set tools from a standard user input, as well as enabling their monitoring.
- Among the available automation tools, build various prototypes and evaluate which one can offer a better performance.
- Combine the already available automation tools with a new system in order to provides extra features that may not exists yet or may not be adequate to the problem.

3.1.4 Idea Selection

In order to perform this step the Analytic Hierarchy Process (AHP) technique was used, which was already explain in previous chapters 2.13. More details about the following steps can be found on the appendix A.

3.1.4.1 Hierarchical Division

The first phase of this technique is to present the most relevant criteria for the decision making.

Therefore, to understand the relevance of each criteria, it is presented Table 3.1 with a matrix containing the various criteria and Table 3.2 with a normalized matrix of those same criteria.

Table 3.1: Pairwise Comparison Matrix

-	Performance	Maintainability	Reliability
Performance	1	$\frac{1}{5}$	$\frac{1}{5}$
Maintainability	5	1	1
Reliability	5	1	1
Sum	11	$\frac{11}{5}$	$\frac{11}{5}$

Table 3.2: Pairwise Comparison Matrix Normalized

-	Performance	Maintainability	Reliability	Weights
Performance	$\frac{9}{100}$	$\frac{9}{100}$	$\frac{9}{100}$	$\frac{9}{100}$
Maintainability	$\frac{9}{20}$	$\frac{9}{20}$	$\frac{9}{20}$	$\frac{9}{20}$
Reliability	$\frac{9}{20}$	$\frac{9}{20}$	$\frac{9}{20}$	$\frac{9}{20}$
Sum	1	1	1	1

3.1.4.2 Priority Definition

Once defined and prioritized the criteria, the next phase is to present and evaluate the different alternatives against each one of them.

- Performance (Table 3.3)
- Maintainability (Table 3.4)
- Reliability (Table 3.5)

Table 3.3: Priorities Regarding Performance

Performance	System from scratch	Using existing tools	Combine both	Weights
System from scratch	$\frac{71}{100}$	$\frac{9}{20}$	$\frac{81}{100}$	$\frac{33}{50}$
Using existing tools	$\frac{7}{50}$	$\frac{9}{100}$	$\frac{3}{100}$	$\frac{9}{100}$
Combine both	$\frac{7}{50}$	$\frac{9}{20}$	$\frac{4}{25}$	$\frac{1}{4}$

Table 3.4: Priorities Regarding Maintainability

Maintainability	System from scratch	Using existing tools	Combine both	Weights
System from scratch	$\frac{13}{20}$	$\frac{14}{25}$	$\frac{69}{100}$	$\frac{63}{100}$
Using existing tools	$\frac{13}{100}$	$\frac{11}{100}$	$\frac{2}{25}$	$\frac{11}{100}$
Combine both	$\frac{11}{50}$	$\frac{33}{100}$	$\frac{23}{100}$	$\frac{13}{50}$

Table 3.5: Priorities Regarding Reliability

Reliability	System from scratch	Using existing tools	Combine both	Weights
System from scratch	$\frac{13}{20}$	$\frac{14}{25}$	$\frac{69}{100}$	$\frac{63}{100}$
Using existing tools	$\frac{13}{100}$	$\frac{11}{100}$	$\frac{2}{25}$	$\frac{11}{100}$
Combine both	$\frac{11}{50}$	$\frac{33}{100}$	$\frac{23}{100}$	$\frac{13}{50}$

3.1.4.3 Global Priorities Comparison

The final step that is missing is to gather the previously determined priorities and compared them in order to conclude which alternative should be chosen (Table 3.6).

Table 3.6: Global Priorities Comparison

-	Performance	Maintainability	Reliability
System from scratch	$\frac{33}{50}$	$\frac{63}{100}$	$\frac{63}{100}$
Using existing tools	$\frac{9}{100}$	$\frac{11}{100}$	$\frac{11}{100}$
Combine both	$\frac{1}{4}$	$\frac{13}{50}$	$\frac{13}{50}$

Taken into consideration the studies presented on Table 3.6, it is clear to say that the alternative to follow is to build an automation "*system from scratch*", which maps to the idea "Develop a system which facilitates the configuration of a set tools from a standard user input, as well as enabling their monitoring".

3.1.5 Concept Definition

The final objective is to implement a system from scratch that can handle the configuration of various tools from a standardized user input and at the same time providing important metrics that enable the monitoring of those tools.

3.2 Value Proposition

This section aims at presenting the value proposition of the desired solution, mainly by focusing on academic contexts characteristics. In order to help elaborating the value proposition, the Value Proposition Canvas was used 2.14.1.

3.2.1 Value Proposition Canvas

In order to conceive the value proposition canvas, both the customer profile and the map value were elaborated, Figure 3.4.

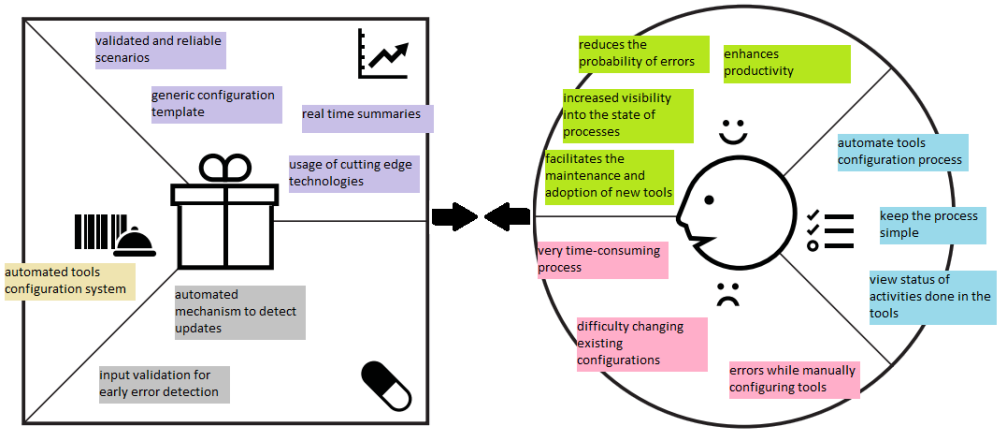

FIGURE 3.4: Value proposition canvas

By looking at the presented value proposition canvas it is possible to quickly verify the main points it targets. From pains and their relievers to gains and their contributors, the value proposition canvas shows in detail the perceived value over the desired solution.

3.3 Requirements Engineering

The focus of this section is to present the functional and non-function requirements as well as the core domain concepts, in order to get a better understanding on the overall solution.

3.3.1 Functional Requirements

Functional Requirements are used to capture and define the system's functionalities and behaviors [23]. These functionalities and behaviors are often represented as tasks, services or user stories that the system must implement in order to perform.

Therefore, in order to fulfill the required needs for the system, Table 3.7 presents the defined functional requirements.

Table 3.7: Functional Requirements

ID	Requirement
Bitbucket / GitLab / GitHub	
FR1	Creation of Git Repositories (from scratch or by forking an existing repository)
FR2	Add user(s) to repository
FR3	Manage users' permissions on a repository
FR4	Perform repository backup among different platforms
FR5	Perform repository download
FR6	Move users among repositories
FR7	Remove user from a list of repositories
Jenkins	
FR8	Creation of jobs
FR9	Add user(s) to job
FR10	Manage users' permissions on a job
FR11	Configure base steps
FR12	List passing and failing jobs
SonarQube	
FR13	Creation of projects
FR14	Add user(s) to project
FR15	Manage users' permissions on a project
FR16	Configure metrics rules
FR17	List passing and failing projects

3.3.2 Non-functional Requirements

Apart from functionalities and behaviours systems are also represented by their constraints and properties, being those the elements that compose the non-functional requirements. Performance, reliability, portability and robustness are examples of non-functional requirements and they are often subjectively evaluated due to its difficulty of testing. Non-functional requirements usually are more critical than functional requirements to the overall system success or failure or in other words, for the overall system quality [4].

The planned non-functional requirements for the system are presented on Table 3.8.

Table 3.8: Non-Functional Requirements

ID	Requirement
NFR1	All the developed services must be containerized and orchestrated
NFR2	The system must support batch processing of the creation and configuration phases of the various tools
NFR3	Creation and Configuration of tools must not take long than 5 seconds (per tool)
NFR4	The final solution must follow a microservices architecture

3.3.3 Domain Model

Understanding the domain of the problem is crucial to get a more robust and grounded knowledge of all the nuances involved.

Figure 3.5 illustrates the conceived domain model that represents the various bounded contexts. The domain model was elaborated according Eric Evans and is represented using Unified Modeling Language (UML).

The purpose of a bounded context representation is to later on assist on the microservices design. Note that the microservices approach on the solution design will be later explained with more detail.

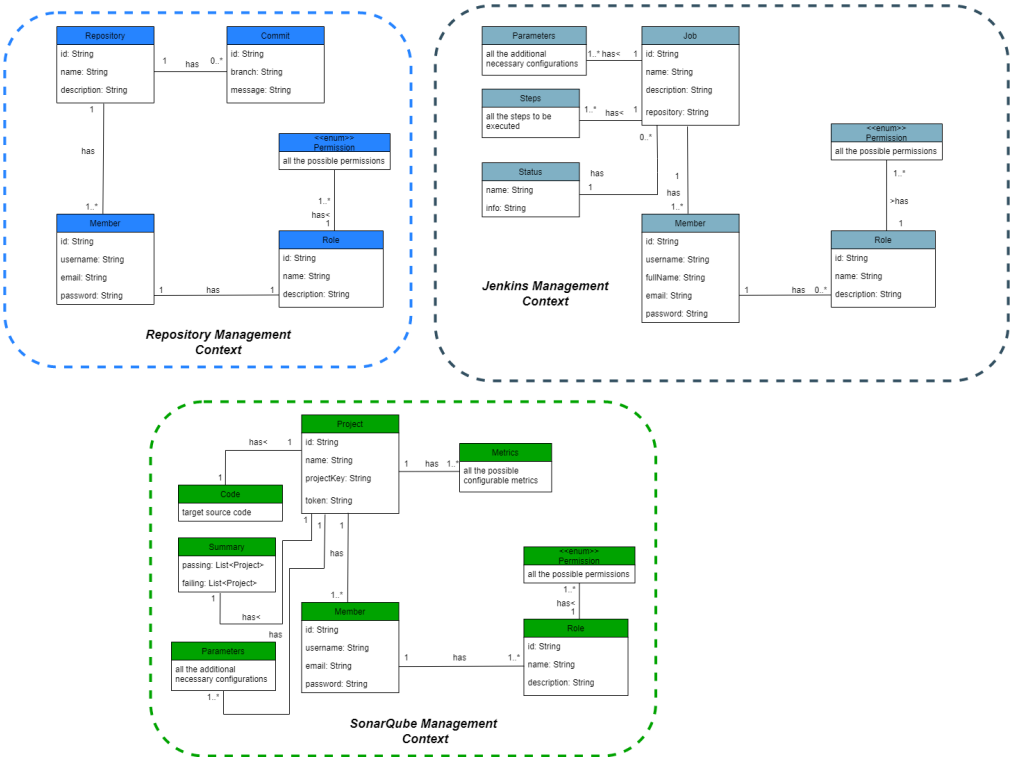

FIGURE 3.5: Domain Model organized by Bounded Contexts

The presented domain model is composed by the main following concepts:

Repository Represents the target object to be created and configured (Repository Context)

Commit Represents a commit on a specific repository

Job Represents the target object to be created and configured (Jenkins Context)

Project Represents the target object to be created and configured (SonarQube context)

Member Represents the base user across the different contexts

Role Represents a set a Permissions that can be assigned to a User or Group

Permission Represents a certain action that can be perform over the different contexts

Status Represents the internal state of a job

Summary Represents the overview of the status of the projects in terms of success and failure according to its respective definition

Parameters Represents any additional configurations that may be needed to fulfill the management of the Jobs and Projects

Steps Represents the tasks that the a certain Job need to execute

Code Represents a repository source code

Metrics Represents the analysis configuration to be performed on a certain project

3.4 Summary

This chapter focused on understanding the value of the proposed solution as well as the opportunity for action. Therefore, once determined those key concepts, both functional and non-functional requirements were defined, which aimed at satisfying all the needs inherent to the problem under study.

That said, with the requirements of the solution detailed, the next step is the development of its design.

Chapter 4

Design

In order to drive the implementation phase, its important to already have a well defined and thought software architecture, were different possible approaches are considered and debated.

This chapter aims at presenting the design proposal for the desired solution, including all the resulting artifacts that bring value and knowledge. It is important to keep in mind that all the design decisions seek fulfilling both functional and non-functional requirements on the best way possible.

4.1 Architecture

Deciding which architecture to adopt when developing a new system can, sometimes, be a very complex and demanding task. Fortunately there are plenty of strategies that can be used to help in this process.

As previously presented on section 3.3.3, when elaborating the domain of the solution, this was organized into bounded contexts or sub-domains. This division not only now helps with the architectural decisions but also promotes a microservices driven approach, as stated on the non-functional requirement NFR4 3.8.

Recalling Figure 3.5, it is possible to identify three major sub-domains: Repository Management, Jenkins Management and SonarQube Management. Having those sub-domains in mind, Figure 4.1 presents the architectural proposal for the desired solution, following a microservices approach.

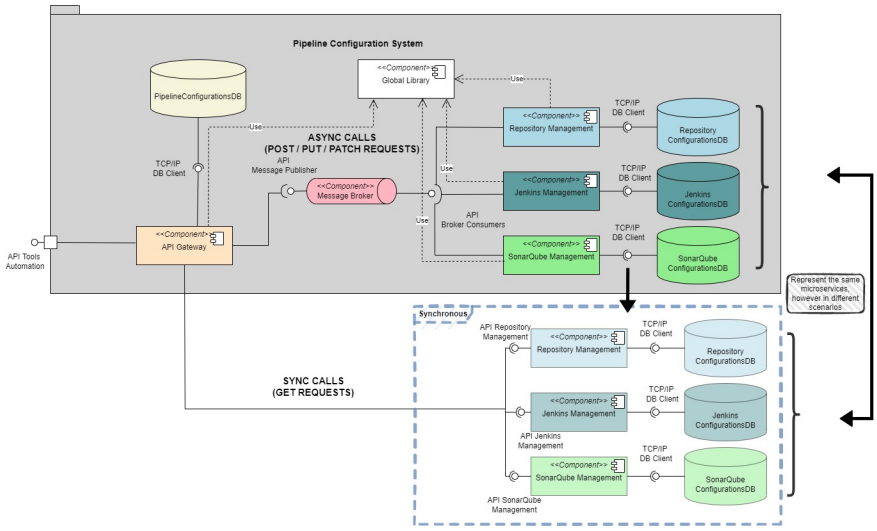

FIGURE 4.1: Tools configuration automation system architecture, components diagram

From Figure 4.1, it is visible that each bounded context originated a single contained component.

In terms of components, the following were proposed:

- **Repository Management** Responsible for handling all repository related operations (Create, Read, Update and Delete (CRUD)) as well as performing users management among the repositories
- **Jenkins Management** Responsible for handling all Jenkins related operations as well as performing users management among the jobs
- **SonarQube Management** Responsible for handling all SonarQube related operations as well as performing users management among the projects
- **API Gateway** Responsible for intercepting all the incoming requests and routing them to the respective service that can perform the requested task. Its main purpose is to have a centralized point to aggregate the requests and coordinate the actions.
- **Global Library** Responsible for providing common features and utilities shared across all the microservices.
- **Message Broker** Responsible for queuing and exchanging messages between the various components.

It is worth to note that besides using microservices, the system rely on a API Gateway component that introduces the API Gateway pattern, a common pattern used in microservices based applications. This component behaves as an entry point to the various microservices when they are trying to be accessed from external clients, performing tasks such as routing, authentication and monitoring. In this specific system it is a very useful pattern because the mentioned advantages it brings, before explained 2.4.3, are required to have.

Also, it is noticeable that each one of the components owns a dedicated database, which is another common pattern used amongst microservices contexts. For this solution, since there are different tools and specific microservices for each tool, it is important to seek an architecture that reduces the overall coupling, also because that is one of the main concerns that a microservices driven architecture tries to mitigate.

Another pattern that is reflected on the presented architecture is the Asynchronous messaging pattern. The API Gateway after receiving a request, depending on its purpose (need of read / write actions), places it on a queue that later on will be consumed triggering the required flow. The fact that the communication is made asynchronously, the client using the application does not block waiting for a reply, resulting in an higher flexibility as well as an higher availability since in situations of stress or failure the messages will simply be added to the queue and processed by the system when possible.

This architectural style can be considered hybrid since it is relaying on characteristics from both event driven and non-event driven architectures.

Regarding the system deployment proposal, Figure 4.2 describes how to various components will be organized. They are all going to coexist inside a common cluster, only having one public reachable point, the API Gateway.

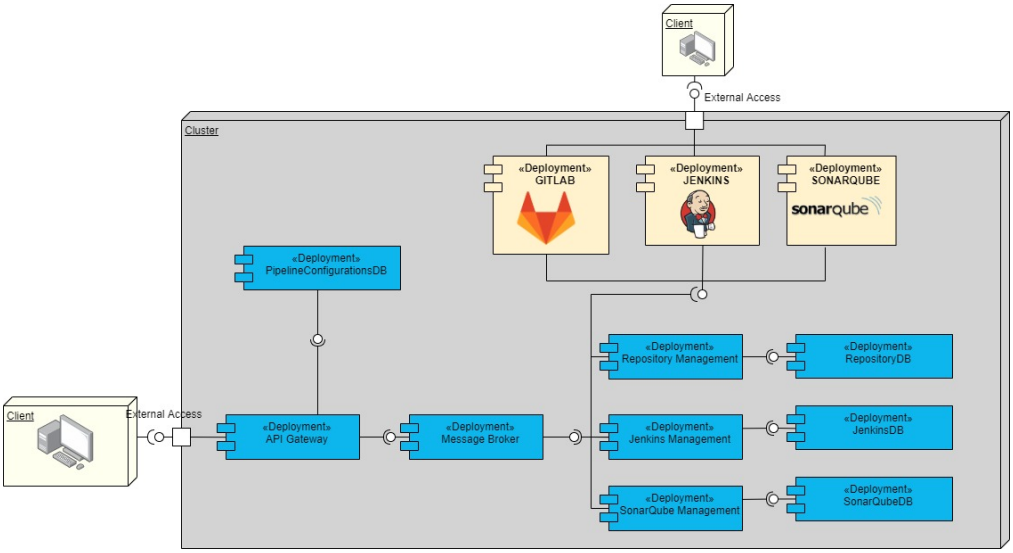

FIGURE 4.2: Tools configuration automation system Deployment Diagram

4.2 Alternative Architecture

One possible alternative for the previously presented architecture is a fully non-event oriented architecture. The majority of the above patterns are kept, such as the API Gateway and database per service, the only one that was abolished was the asynchronous messaging pattern. Instead of queuing the requests on the message broker, they are directly dispatched to the respective service and the response is returned synchronously. All the main components previously presented are also kept.

Figure 4.3 illustrates the alternative architecture proposed, where it is visible the direct communication amongst the services, without the usage of message brokers. Note that the communications between services rely on Hypertext Transfer Protocol (HTTP) / Hypertext Transfer Protocol Secure (HTTPS) requests.

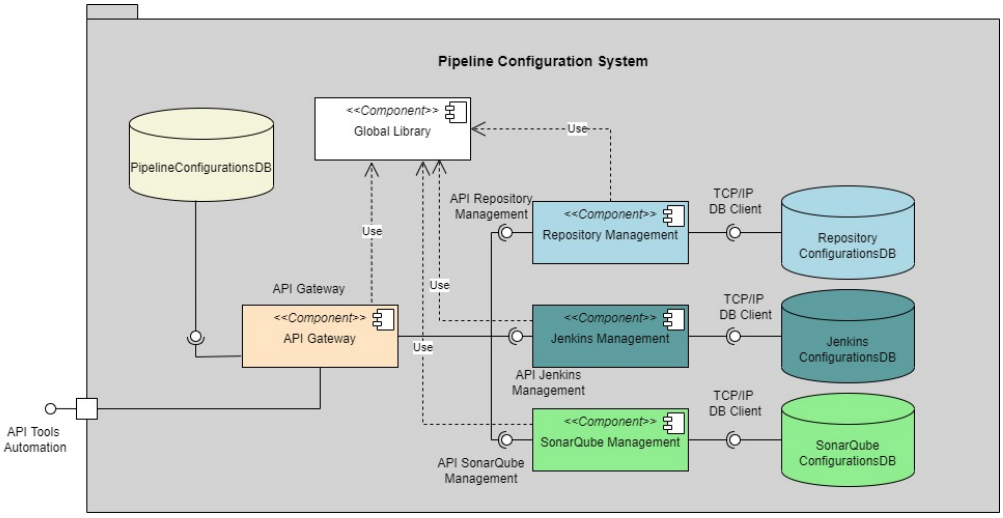

FIGURE 4.3: Tools configuration automation system alternative architecture, Components Diagram

4.3 Alternatives Comparison

Both architectures presented, rely on a microservice architectural style, supported by a separation in various services according to the business capabilities.

The major difference between both architectures is the introduction of a message broker component, which conducts the architecture who owns it to an event driven approach.

In order to understand the major benefits and drawbacks of each architectural proposal Table 4.1 presents the most important points that should be taken into consideration.

Table 4.1: Architectural Proposals Comparison.

Architectural Style	**Hybrid**	**Non-Event Driven**
Complexity	High	Low
Coupling	Low	High
Response Time	Low	High

Taking in consideration that part of the goals of the final solution lay around performance and maintainability, an hybrid architectural style is preferred.

4.4 Summary

Starting from the previously defined requirements, this chapter focused on the elaboration of the architecture to be followed by the solution as well as possible alternatives for it.

Thus, having an architectural base established, the next step is to implement it.

Chapter 5

Implementation

The conception of a working prototype is a very important phase, allowing to ensure that in the end there is a solution that actually matches and corresponds to the established needs and goals, respectively. Therefore, once concluded the design proposal for the desired solution, the following phase is to proceed to its implementation.

This chapter aims at presenting the implementation details of the solution taking into consideration the design decisions previously made, as well as the previously established requirements. To do so, the various components that constitute the solution will be exposed and explained.

Note that during the implementation process the requirements were prioritized in order to ensure that, even if not all of them were implemented or available at the end, the highest priority ones, thus more important to achieve a presentable and functional solution, were there.

5.1 Technological Stack

Before starting the implementation itself it is important to define what technologies are going to be used during that process. That selection can be impacted by various reasons, since previous knowledge, current trending or simply because some technologies have already establish their position in a certain field.

Note that most of the decisions that were made during the implementation were mainly based on the factors previously mentioned, the author experience and knowledge and current state of the art in terms of the most used technologies to accomplish a certain goal.

5.1.1 Microservices

In total, the solution is composed by four different microservices, each one with a very specific and well defined responsibility, in order to accommodate the requirement enumerated on section 3.3.

Each microservice was built using Java, more specifically Spring Boot 2.5.

The Gateway microservice is the entry-point to the entire solution, being responsible of orchestrating and coordinating the incoming requests as well as delegating them to the respective target microservices. Additionally, it allows aggregating all the solution provided features in a single resource.

The Repository, Jenkins and Sonar, as the name indicates, are microservices specialized on GitLab, Jenkins and SonarQube, respectively. Each one of these microservices provide the required functionalities, previously defined, associated with each tool.

Also, there is an additional component, Global, that works as a library providing generic implementations of features used across all the microservices.

In terms of communication between the microservices the usage of a message broker was preferred, where Apache Kafka was the one selected. By using an asynchronous communication style 4.1, then microservices are able to handle possible down-times gracefully without losing any requests made while they were not operable.

5.1.2 Infrastructure

To host the implemented solution, a Kubernetes cluster 2.2.1 was configured on top of a Virtual Machine (VM). Note that all the tools required by the solution (GitLab, Jenkins, SonarQube) are self-managed instances, which means that they are installed, administered and maintained internally, thus providing an higher flexibility.

All of the solution components, from the tools required to the microservices and message brokers are containerized objects, as defined on the non-functional requirement NFR1 3.8 that, in order to be installed on the cluster, were described as specific Kubernetes objects:

- Services
- Deployments
- Persistent Volume Claims

5.2 Requirements

After defined the technological stack to be used, the implementation is ready to be started. However, it is important to keep in mind that the implementation should take into consideration and follow all the requirements 3.3 and design decisions 4.1 previously established.

Note that, in order to support the implementation of the requirements, the targeted tools to be automated need to be configured in advance. The configuration of the various tools will be explained later on this chapter.

Also, during this current section 5.2 not all the implementation details will be presented, only the most important ones will get detailed and explain.

5.2.1 Virtual Machine and Kubernetes Cluster

As previously mentioned, the entire system was built on top of a Virtual Machine (VM) running a Kubernetes cluster.

In order boot a Virtual Machine (VM) already with a running Kubernetes cluster, minikube was selected 2.2.2. To do so, the instruction presented on the following code block is required to be executed 5.1.

```
1 minikube start --memory 11500 --cpus 4 --disk-size 40000 --vm
```

Listing 5.1: Instruction to boot a Virtual Machine (VM) with Kubernetes installed

The parameters visible on the previous code block 5.1, such as memory, cpu and disk space, are all related with the resources that the Virtual Machine (VM) will be given.

5.2.2 GitLab

Regarding GitLab, the first step is to install a self-managed instance on the running Kubernetes cluster.

To do so, the Kubernetes objects beforehand mentioned and illustrated on the following code blocks were defined.

```yaml
# gitlab.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: gitlab-config
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

Listing 5.2: Persistent volume claim object for GitLab

```yaml
# gitlab.yaml
apiVersion: v1
kind: Service
metadata:
  name: gitlab
spec:
  selector:
    app: gitlab
  ports:
    - name: "80"
      port: 80
      targetPort: 80
      nodePort: 30100
...
  type: NodePort
```

Listing 5.3: Service object for GitLab

```yaml
# gitlab.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gitlab
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gitlab
  template:
    metadata:
      labels:
        app: gitlab
    spec:
      containers:
        - name: gitlab
          image: gitlab/gitlab-ee:latest
          ports:
            - containerPort: 80
...
      volumes:
        - name: config
          persistentVolumeClaim:
            claimName: gitlab-config
```

Listing 5.4: Deployment object for GitLab

The Persistent Volume Claim object 5.2, as the name suggests, is an internal Kubernetes storage unit that will hold crucial data, in this case mainly related to configurations, preventing it to be lost.

The Service object 5.3 is mainly used as a networking endpoint for the Pods, providing a basic load balancing among them.

Lastly, the Deployment object 5.2, is the closed we have to a Pod representation, the Kubernetes atomic unit. There we can see how many Pods of GitLab we are going to have running (field *replicas*), what image of GitLab we are using (field *image*), among other important configurations, such as environment variables and volumes used.

Having all the necessary Kubernetes objects defined, once the code block 5.5 is applied, GitLab is installed on the cluster. Note that all the various Kubernetes objects were defined in a single file, otherwise would be necessary to execute the command 5.5 for each one of the files representing an object.

```
1 kubectl apply -f gitlab.yaml
```

Listing 5.5: Instruction to apply the various GitLab Kubernetes objects

After having a stable GitLab instance up and running, there are two main configurations that are required:

Communication The automation of GitLab actions will be entirely made using its Application Programming Interface (API), which requires a token to be generated, allowing external applications to communicate with the GitLab instance. Figure 5.1 illustrates this step.

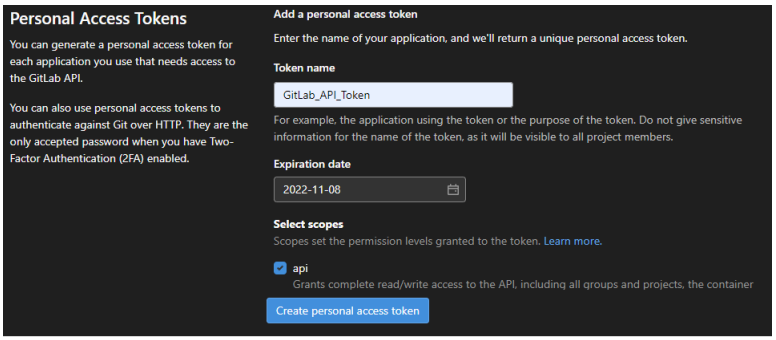

Figure 5.1: GitLab Application Programming Interface (API) token generation

Authentication Since GitLab will be the main source of true in terms of authentication, it is necessary to configure an application in order to allow other tools to use GitLab as an Open Authorization (OAuth) provider. Figure 5.2 illustrates this step.

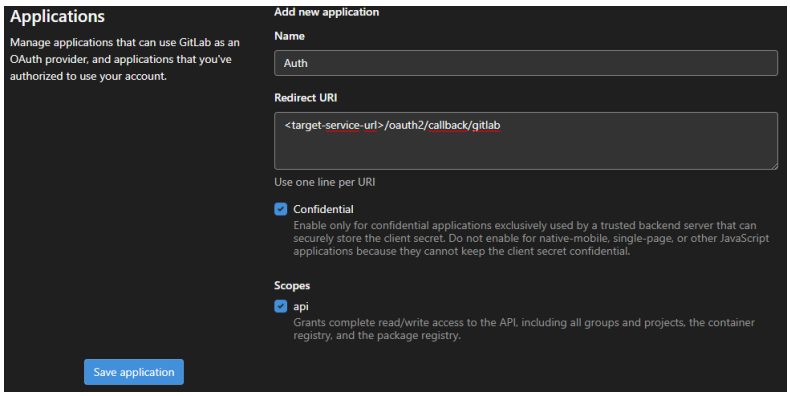

Figure 5.2: GitLab Open Authorization (OAuth) application creation

5.2.3 Jenkins

Similarly to GitLab 5.2.6, we need to install a self-managed Jenkins instance on the running Kubernetes cluster. To do so, and as explained before, the Kubernetes objects illustrated on the following code blocks 5.6, 5.7, 5.8 were defined.

```
# jenkins.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-config
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

Listing 5.6: Persistent volume claim object for Jenkins

```
# jenkins.yaml
apiVersion: v1
kind: Service
metadata:
  name: jenkins
spec:
  selector:
    app: jenkins
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30050
  type: NodePort
```

Listing 5.7: Service object for Jenkins

```
# jenkins.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkins
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jenkins
  template:
    metadata:
      labels:
        app: jenkins
    spec:
      containers:
        - name: jenkins
          image: jenkins/jenkins:lts
          ports:
            - containerPort: 8080
          imagePullPolicy: Always
          volumeMounts:
            - name: config
              mountPath: /var/jenkins_home
      volumes:
        - name: config
          persistentVolumeClaim:
            claimName: jenkins-config
```

Listing 5.8: Deployment object for Jenkins

Having all the required objects defined the code block 5.9 is applied.

```
kubectl apply -f jenkins.yaml
```

Listing 5.9: Instruction to apply the various Jenkins Kubernetes objects

Once the Jenkins instance is ready to be used, there are some configurations that are required:

Plugins In order to take full advantage of Jenkins and enhance its features some plugins are required to be installed. Figure 5.3 illustrates this step.

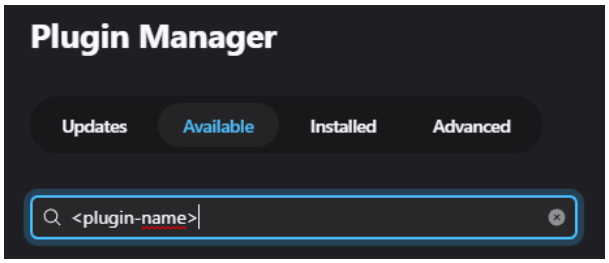

Figure 5.3: Jenkins plugins installation

Communication The automation of Jenkins actions will be entirely made using its Application Programming Interface (API), which requires a token to be generated, allowing external applications to communicate with the Jenkins instance. Figure 5.4 illustrates this step.

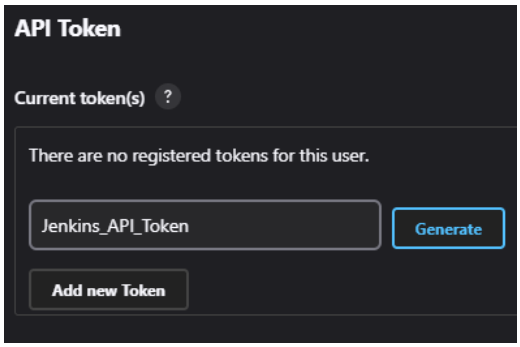

Figure 5.4: Jenkins Application Programming Interface (API) token generation

Credentials By its nature, Jenkins is required to communicate with various services, such as GitLab and SonarQube. In order to do so, it requires those services credentials to be configured, thus allowing the communication between them. Figure 5.5 illustrates this step.

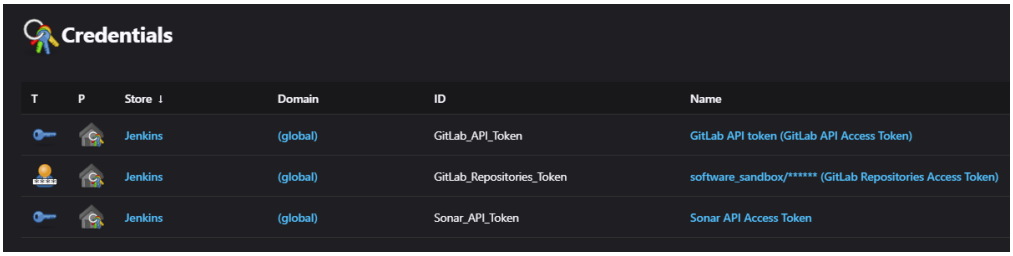

Figure 5.5: Jenkins credentials manager

GitLab Connection Having the GitLab credentials created, the connection itself needs to be configured. Figure 5.6 illustrates this step.

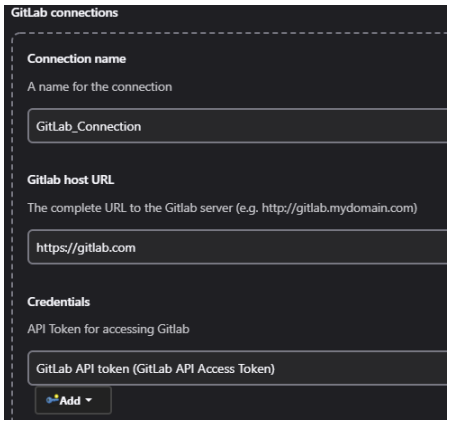

FIGURE 5.6: Jenkins GitLab connection configuration

SonarQubbe Connection Having the SonarQube credentials created, the connection itself needs to be configured. Figure 5.7 illustrates this step.

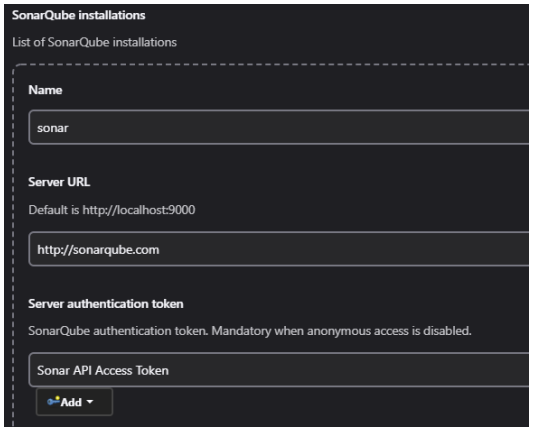

FIGURE 5.7: Jenkins SonarQube connection configuration

5.2.4 SonarQube

Regarding SonarQube, the initial setup is very similar to GitLab 5.2.6 and Jenkins 5.2.7, being the first step installing a self-managed instance on the running Kubernetes cluster. To do so, and as explained before, the Kubernetes objects illustrated on the following code blocks 5.10, 5.11, 5.12 were defined.

```
# sonarqube.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sonar-data
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 60Mi
```

LISTING 5.10: Persistent volume claim object for SonarQube

```yaml
# sonarqube.yaml
apiVersion: v1
kind: Service
metadata:
  name: sonarqube
spec:
  selector:
    app: sonarqube
  ports:
    - port: 9000
      targetPort: 9000
      nodePort: 30500
  type: NodePort
```

Listing 5.11: Service object for SonarQube

```yaml
# sonarqube.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sonarqube
spec:
  selector:
    matchLabels:
      app: sonarqube
  template:
    metadata:
      labels:
        app: sonarqube
    spec:
      containers:
        - name: sonarqube
          image: sonarqube:latest
          ports:
            - containerPort: 9000
          env:
            - name: SONAR_ES_BOOTSTRAP_CHECKS_DISABLE
              value: "true"
          volumeMounts:
            - name: data
              mountPath: /opt/sonarqube/data
...
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: sonar-data
```

Listing 5.12: Deployment object for SonarQube

Once again, having all the required objects defined the code block 5.13 is applied.

```
kubectl apply -f sonarqube.yaml
```

Listing 5.13: Instruction to apply the various SonarQube Kubernetes objects

Once the SonarQube instace is fully up and running, there are some configurations that are required:

Communication The automation of SonarQube actions will be entirely made using its Application Programming Interface (API), which requires a token to be generated, allowing external applications to communicate with the SonarQube instance. Figure 5.8 illustrates this step.

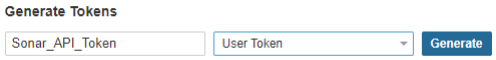

FIGURE 5.8: SonarQube Application Programming Interface (API) token generation

Authentication Since GitLab will be the main source of true in terms of authentication, it is necessary to configure SonarQube, enabling GitLab to be used as an Open Authorization (OAuth) provider. Figure 5.9 illustrates this step.

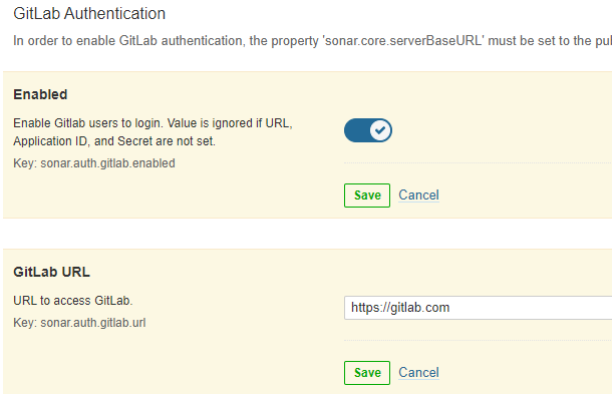

FIGURE 5.9: SonarQube Authentication with GitLab Open Authorization (OAuth) application Configuration

5.2.5 Gateway Microservice

This project, as the name suggests, implements the API Gateway pattern, being responsible for aggregating all the features provided by the system into a single entry-point, as well as orchestrating the requests among the various microservices. It exposes a Representational state transfer (REST) Application Programming Interface (API) making use of the OpenAPI Specification, so that the users can visualize the service provided resources.

The Gateway microservice implements an hybrid behaviour, communicating synchronously with the other microservices when receives GET requests, and asynchronously when received POST or PUT requests. However, for the overall solution goal, the asynchronous messaging behavior is the most important one.

As it is visible on the following code block 5.14, the Gateway microservice receives as input a *PipelineConfig* Data Transfer Object (DTO) 5.15, or in order words, an object from where the various tools configurations will derive.

```
1  //PipelineService.java
2  private void dispatchPipelineConfig(PipelineConfig pipelineConfig) throws
       AutomationException {
3          try {
4              this.repositoryTemplate.send(REPOSITORY_CONFIG_TOPIC,
       randomLong(), ComponentsMapper.mapRepository(pipelineConfig));
5              this.mongoUtilsPipeline.insertRecord(
6                      new GlobalPipelineConfigs(
7                              pipelineConfig.getName(),
8                              ComponentsMapper.mapJenkins(pipelineConfig,
       defaultRepositoryUrl),
9                              ComponentsMapper.mapSonar(pipelineConfig),
10                             ComponentsMapper.mapRepository(pipelineConfig)
11                     ), mongoTemplatePipeline);
12         } catch (Exception e) {
```

```
13              throw new AutomationException(e.getLocalizedMessage(),
    HttpStatus.INTERNAL_SERVER_ERROR.value());
14          }
15      }
```

Listing 5.14: *PipelineService* the class responsible for handling the user requests

```
1  //PipelineConfig.java
2  @Getter
3  @Setter
4  @AllArgsConstructor
5  @NoArgsConstructor
6  @Builder
7  public class PipelineConfig {
8
9      private String name;
10     private String description;
11     private String jenkinsTemplate;
12     private String forkUrl;
13     private List<PipelineUsers> users;
14
15 }
```

Listing 5.15: *PipelineConfig* the Data Transfer Object (DTO) that contains the generic pipeline configuration

However, looking closely to the code block 5.14, it is visible that only the repository related configurations are sent to the message broker (*Kafka*), while the remaining ones are aggregated into a new Data Transfer Object (DTO) and store on the database. This happens due to the dependency that the other tools, Jenkins and SonarQube, habe with the repository management tool chosen, GitLab.

Since Jenkins and SonarQube directly depend on GitLab, either for configuration or authorization purposes, the Gateway microservice cannot forward the configurations received to the microservices responsible for their configuration right way. Instead, it computes and stores them on a database. In parallel, is actively listing on a specific *Kafka* topic, waiting for a notification to be sent by the Repository microservice informing that the configuring process there is finished, as the code block 5.16 presents.

```
1  //PipelineKafkaConsumer.java
2  @KafkaListener(topics = PIPELINE_CONFIG_TOPIC, groupId = "pipeline.
       consumer")
3      public void consume(ConsumerRecord<Long, String> record) {
4          if (record.value() != null) {
5              LOGGER.info("KAFKA >> Received new Pipeline Configuration for
       {}", record.value());
6              try {
7                  consumeRecord(record);
8              } catch (Exception e) {
9                  LOGGER.error("KAFKA >> Error Processing Record");
10                 throw new RuntimeException(e);
11             }
12         } else {
13             LOGGER.error("KAFKA >> Invalid Record: null");
14         }
15     }
```

Listing 5.16: *PipelineKafkaConsumer* the class that contains the generic pipeline configuration

Every time a new notification is received, the Gateway microservice, as the code block 5.17 shows, fetchs back the respectively configuration from the database and then dispatches it for the remaining microservices, Jenkins and Sonar.

```java
//PipelineKafkaConsumer.java
private void consumeRecord(ConsumerRecord<Long, String> record) {
        GlobalPipelineConfigs config = fetchConfig(record.value());
        if (Objects.nonNull(config)) {
            this.sonarTemplate.send(SONAR_CONFIG_TOPIC, randomLong(),
    config.getSonarConfig());
            this.jenkinsTemplate.send(JENKINS_CONFIG_TOPIC, randomLong(),
    config.getJenkinsConfig());
        } else {
            LOGGER.error("ERROR >> No Configuration Found for Group {}",
    record.value());
        }}
```

LISTING 5.17: *PipelineKafkaConsumer* dispatching the remaining configurations to Jenkins and Sonar microservices

5.2.6 Repository Microservice

This project exposes a Representational state transfer (REST) Application Programming Interface (API) and consumes data from a centralized message broker. However, the asynchronous data consumption is the most important part. Whenever a new message arrives, containing repository configuration objects, this project consumes that message and communicates directly with the GitLab Application Programming Interface (API) in order to perform the necessary configurations requested.

As it is visible on the code block 5.18, this service is listening on a specific *Kafka* topic and, whenever a new message gets there, it will try to consume it in case the message is valid, according to the input the service is expecting.

```java
//RepositoryKafkaConsumer.java
@KafkaListener(topics = REPOSITORY_TOPIC, groupId = "repository.consumer")
    public void consume(ConsumerRecord<Long, RepositoryConfig> record) {
        if (record.value() != null) {
            LOGGER.info("KAFKA >> Received new Repository Configuration {}
    ", record.key());
            try {
                consumeRecord(record);
            } catch (Exception e) {
                LOGGER.error("KAFKA >> Error Processing Record");
            }
        } else {
            LOGGER.error("KAFKA >> Invalid Record: null");
        }}
```

LISTING 5.18: *RepositoryKafkaConsumer* the class responsible for consuming and handling incoming messages from *Kafka*

The *RepositoryConfig* object, presented on the following code block 5.19, is a Data Transfer Object (DTO) that contains all the specifications necessary to fulfill the various use cases that the microservice is responsible for.

```java
//RepositoryConfig.java
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class RepositoryConfig {
    private List<RepositoryDto> repositories;
    private List<PermissionsDto> permissions;
    private ModificationsDto modifications;
    private List<UserDto> users;
}
```

LISTING 5.19: *RepositoryConfig* the Data Transfer Object (DTO) that contains the various repository related requests configurations

At the consumption level, each one of the possible configuration requests are handled sequentially, as it is visible on code block 5.20

```
//RepositoryKafkaConsumer.java
private void consumeRecord(ConsumerRecord<Long, RepositoryConfig> record)
    {
        RepositoryConfig validatedPayload = validatePayload(record.value()
    );
        dispatchUsers(validatedPayload.getUsers());
        dispatchRepositories(validatedPayload.getRepositories());
        dispatchPermissions(validatedPayload.getPermissions());
        dispatchModifications(validatedPayload.getModifications());
    }
```

Listing 5.20: Sequential consumption of the various configurations

Since all the functional requirements followed a similar approach in terms of implementation, only one of them will be explained in detail. Having that in mind, the following explanation is the implementation of the functional requirement FR1, that was previously defined 3.7.

Taking a closer look at the code block 5.21, it is visible that the various repository configurations received are handled iteratively, fulfilling the non-functional requirement NFR2 3.8, calling a method, in the case, from the *GitLabService* class, since GitLab was the repository management tools chosen. Note that the repository, in case of having any initial members, they are also created and configured on this step.

One crucial step that happens on the code block 5.21 is the part when, after dispatching the received configurations, the service produces a message to another *Kafka* topic, in order to notify the completion of its tasks, allowing the other microservices (Jenkins and Sonar) to start configuring and integrating with the newly created repositories. As mentioned before 5.2.5 Jenkins and SonarQube directly depend on both GitLab repositories and GitLab users, either for configuration or authorization purposes so, the Repository microservice needs to send a notification when fully finishes a certain configuration, thus allowing the other microservices to perform their job as well.

```
//RepositoryKafkaConsumer.java
private void dispatchRepositories(List<RepositoryDto> repositories) {
        repositories.forEach(repositoryDto -> {
            try {
                Repository createdRepository = gitLabService.
    createRepository(DtoToDomainMapper.mapRepository(repositoryDto));
                LOGGER.info("Repository {} created successfully",
    repositoryDto.getName());
                dispatchMembers(createdRepository.getId(), Objects.nonNull
    (repositoryDto.getUsers()) ? repositoryDto.getUsers() : Collections.
    emptyList());
                this.pipelineTemplate.send(PIPELINE_TOPIC, randomLong(),
    repositoryDto.getName());
            } catch (HttpException | AutomationException | MongoException
    e) {
                LOGGER.error("ERROR >> Failed to configure repository: {}"
    , repositoryDto.getName());
            }});
    }
```

Listing 5.21: Sequential consumption of the various configurations

Taking a look at the service layer, two main actions are performed, as the code block 5.22 shows.

First, the call to the GitLab Application Programming Interface (API) is made, in order to programmatically create and configure the repository.

After that, in order to also keep a persistence layer on the microservice side, the created repository is stored on a *MongoDB* database. This is important to avoid unnecessary

calls to the GitLab Application Programming Interface (API) whenever small amounts of data are required. Instead, that data can easily be fetched from a database, reducing the overall complexity and error prone calls to the Application Programming Interface (API).

```
//GitLabService.java
@Override
    public Repository createRepository(Repository repository) throws
    HttpException, AutomationException, MongoException {
        Repository result = this.mapperRepositories.parseResponseBody(
                this.repositoryPost(repository, url, PROJECTS_PATH),
    Repository.class);
        commitSonarFile(result.getId(), result.getName());
    ((GitLabRepository) result).setWeb_url(concatUrl(repositoriesPath,
    result.getName()));
        mongoUtilsRepository.insertRecord(new RepositoryDbo().map((
    GitLabRepository) result),
                mongoTemplateRepository);
        return result;
    }
```

Listing 5.22: *GitLabService* creation of repositories method

5.2.7 Jenkins Microservice

This project also exposes a Representational state transfer (REST) Application Programming Interface (API) and consumes data from a centralized message broker. As the name indicates, this microservice main responsibility is to consume Jenkins related configurations and communicate with the Jenkins self-managed instance in order to dispatch them.

In order to fulfill one of the functional requirement, FR11 3.7, the Jenkins microservice allows the configuration of jobs base steps making use of seeder jobs, as the following code block 5.23 presents.

```
//JenkinsKafkaConsumer.java
private void dispatchSeeds(List<SeedDto> seeds) {
        seeds.forEach(seedDto -> {
            try {
                jenkinsService.createJenkinsSeedJob(
                        seedDto.getSeedName(),
                        seedDto.getSeedTemplate().getBytes(
    StandardCharsets.UTF_8)
                );
                LOGGER.info("Successfully created seed job {}",
                        seedDto.getSeedName());
            } catch (HttpException | AutomationException | MongoException
    e) {
                LOGGER.error("ERROR >> Failed to create seed job: {}",
    seedDto.getSeedName());
            }});
    }
```

Listing 5.23: *JenkinsKafkaConsumer* creation of Jenkins seed jobs

The only input needed to create this type of jobs is the name to give to the seeder job and the base steps that will be used as template. Note that the seeder jobs base steps can be expressed using Jenkins own *Job Domain-specific language (DSL)* plugin, allowing them to be defined in a programmatic but human readable style.

Regarding the functional requirement FR8 3.7, and as the code blocks 5.24 presents, the Jenkins microservice also allows the creation of ordinary jobs, targeting a certain repository. To do so, the seeder jobs previously mentioned come into action. Having a seeder job available, to create a new Jenkins job it is only necessary to trigger the seeder job with the missing configurations (job name and repository Uniform Resource Locator

(URL)). This then results on a new job to be automatically created and configured, thus inheriting the seeder job base steps.

```
1  //JenkinsKafkaConsumer.java
2  private void dispatchJobs(List<JobDto> jobs) {
3          jobs.forEach(jobDto -> {
4              try {
5                  jenkinsService.triggerJenkinsSeedJob(
6                          Objects.isNull(jobDto.getSeedTemplate()) ?
       DEFAULT_SEED_JOB : jobDto.getSeedTemplate(),
7                          CreateJobPayload.builder().jobName(jobDto.
       getJobName()) .repoUrl(jobDto.getRepoUrl()).build());
8                  LOGGER.info("Successfully created job {}", jobDto.
       getJobName());
9              } catch (HttpException | AutomationException | MongoException
       e) {
10                 LOGGER.error("ERROR >> Failed to create job: {}", jobDto.
       getJobName());
11             }});
12     }
```

Listing 5.24: *JenkinsKafkaConsumer* creation of ordinary Jenkins jobs

Both seeder and ordinary jobs are created communicating directly with the Jenkins instance Application Programming Interface (API), being then stored on an internal database as code block 5.25 illustrates.

```
1  //JenkinsService.java
2  public String triggerJenkinsSeedJob(String seedName, CreateJobPayload
       createJobPayload) throws AutomationException, HttpException,
       MongoException {
3          String handledResponse = jenkinsPost(null, url, JOB, seedName,
       TRIGGER_SEED, ADD_URL_PARAMS, JOB_PARAM, createJobPayload.getJobName()
       , CONCAT_URL_PARAMS,
4                  URL_PARAM, createJobPayload.getRepoUrl());
5          mongoUtilsJobs.insertRecord(new JobDbo().map(createJobPayload),
       mongoTemplateJenkins);
6          return handledResponse;
7      }
```

Listing 5.25: *JenkinsService* creation of job using the Jenkins and storage on the database

Note that the user management related requirements on Jenkins were not implemented due to the tight schedule and also because of Jenkins' own Application Programming Interface (API) restrictions. An alternative would be to use an automation tool to automate the manual inputs needed to perform the user management tasks directly on the browser, however this type of approach, in terms of maintainability, is not desirable, since the browser page is likely to be change with higher frequency, causing constant changes on the service as well.

5.2.8 Sonar Microservice

This project similarly to Repository and Jenkins microservices, also exposes a Representational state transfer (REST) Application Programming Interface (API) and consumes data from a centralized message broker. Its main responsibility is to consume SonarQube related configurations and communicate with the SonarQube self-managed instance in order to dispatch them.

As it is visible on the code block 5.26, like the other services, the Sonar microservice is actively listening on a *Kafka* queue, waiting for new configurations to dispatch.

```
//SonarKafkaConsumer.java
@KafkaListener(topics = SONAR_TOPIC, groupId = "sonar.consumer")
    public void consume(ConsumerRecord<Long, SonarConfig> record) {
        if (record.value() != null) {
            LOGGER.info("Kafka >> Received new Sonar Configuration {}",
    record.key());
            try {
                consumeRecord(record);
            } catch (Exception e) {
                LOGGER.error("Kafka >> Error Processing Record");
            }
        } else {
            LOGGER.error("Kafka >> Invalid Record: null");
        }}
```

LISTING 5.26: *SonarKafkaConsumer* actively listening for new configurations and dispatching them

One important remark on the Sonar microservice is the mechanism behind the functional requirement FR13 3.7.

Recalling the code block 5.22 it is visible that when the repository configuration is made a SonarQube related file is also committed. This file is very important because it will act as the bridge that connects GitLab and SonarQube. Looking at the code block 5.27 it is visible that the object forwarded to the service layer contain a project specific key that is the same one present on the file previously committed. On a first step, the project on SonarQube is created only with its members configured, not pointing to any repository. However, the first time a certain repository is built on Jenkins that will automatically trigger SonarQube that will bind the repository with the respective project, like previously said, by matching the project key.

```
//SonarKafkaConsumer.java
private void dispatchProjects(List<ProjectDto> projects) {
        projects.forEach(projectDto -> {
            try {
                sonarService.createSonarProject(
                        SonarProject.builder()
                                .name(projectDto.getName())
                                .key(projectDto.getKey())
                        .visibility(VisibilityLevel.valueOf(projectDto.
    getVisibility().toUperCase())).build());
                LOGGER.info("Successfully created project {}", projectDto.
    getName());
            } catch (HttpException | AutomationException | MongoException
    e) {
                LOGGER.error("ERROR >> Failed to create project {}",
    projectDto.getName());
            }
            dispatchMembers(projectDto.getKey(), Objects.nonNull(
    projectDto.getProjectsUsers()) ? projectDto.getProjectsUsers() :
    Collections.emptyList());
        });
    }
```

LISTING 5.27: *SonarKafkaConsumer* actively listening for new configurations and dispatching them

5.3 Summary

Taking into consideration the developed prototype, this chapter focused on its evaluation. This evaluation is a very important step since it is here that it can be concluded if the solution really meets the previously defined requirements, thus allowing for the corroboration of the elaborated hypotheses. For the evaluation, load tests and code analyzers were used.

Chapter 6

Experiments and Evaluation

This chapter, as the name suggests, aims at describing the experiments made to the final solution as well as its assessment based on those experiments.

Firstly the various hypothesis were identified, followed by the indicators and sources of information, and finally the evaluation methodology.

6.1 Hypothesis

Taken into consideration the context of the current work, two hypothesis were already previously elaborated 1.5.

Just to briefly recall the defined hypotheses, their meaning is presented again below.

H1 - Hypothesis related with the solution performance The implemented solution have an higher performance than any other manual approach.

H2 - Hypothesis related with the solution maintainability and reliability The implemented solution present reasonable levels of maintainability and reliability, even if the number of tools it has to automate increases.

6.2 Indicators

In order for an hypothesis to be accepted, it must be possible to test it. To do so, either for corroborating or refuting the hypothesis, is needed the definition of various evaluation indicators.

In the presented context of the desired solution, the following evaluation indicators were defined:

Performance Evaluate the final solution based on response time, inputs, different loads and other factors that might influence the performance

Maintainability Since the targeted tools to automate may suffer constant modifications over time, it is important to evaluate if it is easy to update the solution to quickly correct the flaws and adapt to the new changes.

Reliability Since the number of tools to keep track and automate or the load of requests may increase, it is important to assess if the overall is reliable.

The various evaluation indicators mentioned will mainly be assessed based on system metrics, static code analyzers such as SonarQube and load tests.

6.3 Methodology

An evaluation methodology reflects the way hypothesis will be verified. However, the methodology to use depends on the hypothesis under evaluation and the properties to be considered.

To the hypothesis related with the system overall performance, load tests should be preferred, since they are a very useful tool to assess performance. The expected result is that the solution will support a number of requests as if it were being used in a real academic or professional context, while maintaining its response time within a certain range.

To the hypothesis related with the system overall maintainability and reliability, static code analyzers are the preferred approach. Both in terms of maintainability and reliability, static code analyzers are useful tools to assess the status of the source code of the solution and how easy would be to update or change it. Here, it is expected that the solution presents reasonable results, ensuring that way its ease of maintenance.

6.4 Results

Once defined the hypotheses and the evaluation methodologies, the next step is to present the results achieved.

The following section aims at describing the outputs of the evaluation as well as their explanation.

6.4.1 Hypothesis related with the solution performance

The first hypothesis, which is mainly focused on performance, was assessed using load tests. These tests had the objective of simulating a scenario as close as possible to reality. Appendix C presents a more detailed overview about the tests steps.

The first test consisted on setting up the tools for five groups of students, each one with four elements (usually the size of a real class). The evaluation is presented individually for each tool and the obtained results are presented on the following tables. Table 6.1 presents the results for GitLab, Table 6.2 for Jenkins and Table 6.3 for SonarQube.

Table 6.1: Performance results for GitLab

Group	Time Required
1	3.6 seconds
2	2.4 seconds
3	2.9 seconds
4	2.1 seconds
5	2.3 seconds

Table 6.2: Performance results for Jenkins

Group	Time Required
1	0.5 seconds
2	0.03 seconds
3	0.04 seconds
4	0.04 seconds
5	0.02 seconds

Table 6.3: Performance results for SonarQube

Group	Time Required
1	1.4 seconds
2	1.3 seconds
3	1 seconds
4	1.2 seconds
5	1 seconds

The results presented above, as expected, support the hypothesis that the implemented solution provides a much higher level of performance than the traditional manual process.

Just for a matter of compassion, the manual process was measured in minutes, while the implemented solution only took a few seconds to perform its tasks. However, it is important to remember that Jenkins has the highest performance level due to the lack of the user management feature, which lowers the total time it requires. Nevertheless, it is still possible to verify that an automated approach, like the implemented solution provides way higher performance levels.

The second test, in order to consolidate the first one, consisted on making a cascade change by moving an element from one group to another (with different permissions). This forces the configurations to be updated across all the tools. Note that Jenkins user management is not currently supported by the solution. Table 6.4 presents the obtained results.

Table 6.4: Performance results for a cascade change

Tool	Time Required
GitLab	0.6 seconds
SonarQube	0.3 seconds

Once again the hypothesis is corroborated, thus allowing to safely state that the implemented solution, in terms of performance, is far superior to the traditional manual process.

6.4.2 Hypothesis related with the solution maintainability and reliability

The second hypothesis, which focuses on the factors maintainability and reliability, was assessed using one of the tools automated by the solution, SonarQube.

The evaluation was performed by breaking down the various components of the solution and analyse each one of them under various rules.

In order to understand the following analysis, Table 6.5 should be taken into consideration.

TABLE 6.5: Analysis results evaluation

Result	Evaluation
A	Acceptable (✓)
B	Acceptable (✓)
C	Acceptable (✓)
D	Not Acceptable (✗)
E	Not Acceptable (✗)

Gateway Microservice The Gateway microservice presents acceptable results both for maintainability and reliability, however the reliability could be improved since it is very close to a non acceptable evaluation. Figure 6.1 illustrates the achieved results for the Gateway microservice.

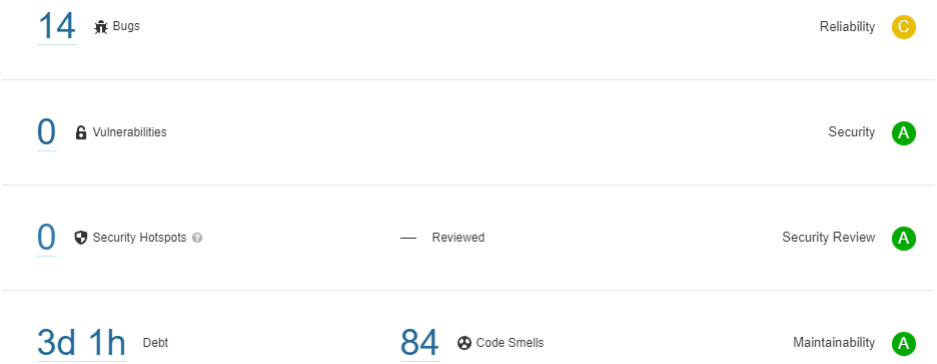

FIGURE 6.1: Gateway microservice SonarQube analysis results

Repository Microservice The Repository microservice presents acceptable results both for maintainability and reliability, however the reliability could be improved since it is very close to a non acceptable evaluation. Figure 6.2 illustrates the achieved results for the Repository microservice.

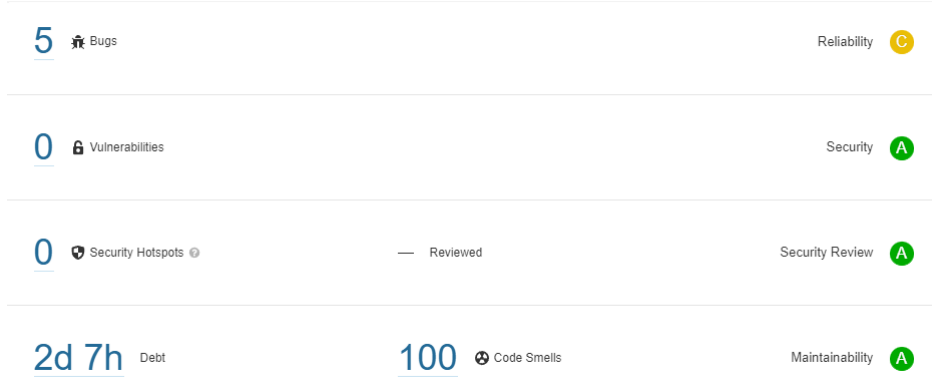

FIGURE 6.2: Repository microservice SonarQube analysis results

Jenkins Microservice The Jenkins microservice presents acceptable results both for maintainability and reliability. Figure 6.3 illustrates the achieved results for the Jenkins microservice.

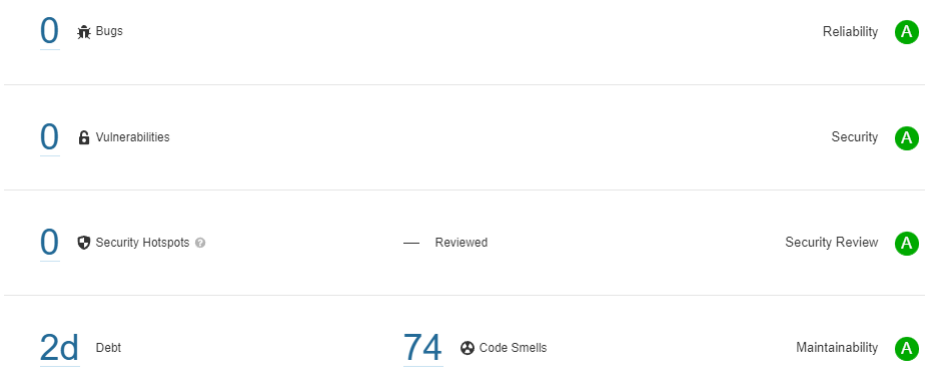

FIGURE 6.3: Jenkins microservice SonarQube analysis results

Sonar Microservice The Sonar microservice presents acceptable results both for maintainability and reliability, however the reliability could be improved since it is very close to a non acceptable evaluation. Figure 6.4 illustrates the achieved results for the Sonar microservice.

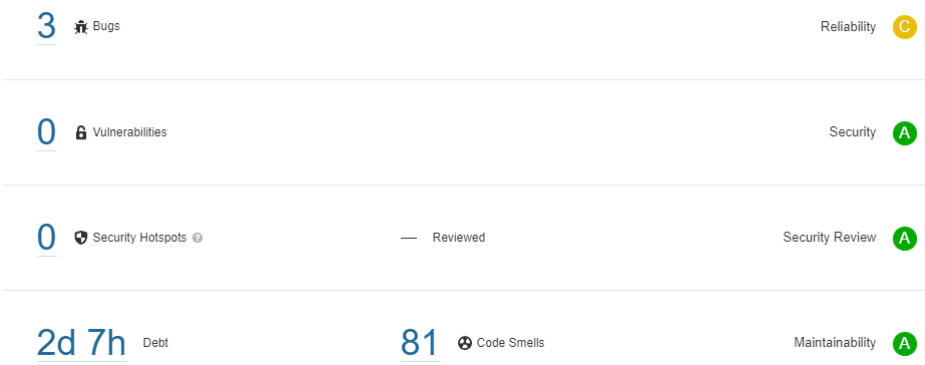

FIGURE 6.4: Sonar microservice SonarQube analysis results

The results previously presented corroborate with the hypothesis that despite the complexity that an automation system can have and despite the number of tools it is handling, if well designed and implemented, its maintainability and reliability are kept simple and high, respectively.

Note that, the reliability on the Gateway, Repository and Sonar microservices was indirectly impacted due to the tight schedule in which the solution was developed.

6.5 Summary

Starting from the previously defined requirements and architectural model, this chapter focused on detailing the main aspects inherent to the implementation of the solution. From the configuration of the necessary infrastructure and tools to the implementation of the functionalities themselves, all the steps involved were carefully presented and explained.

This is followed by an analysis of the solution in order to validate whether it actually meets and solves the requirements and problems identified, respectively.

Chapter 7

Conclusion

This dissertation allowed to address a problem already known in technological environments as well as to develop a possible solution to mitigate or even eliminate it.

In a first phase, the concepts involved around the desired solution were studied in order to understand what is the current state of the art, Chapter 2, is regarding similar situations and, consequently, which best practices should be followed. In addition, a value analysis, Chapter 3, was also performed around the problem in order to identify the best opportunity to be followed.

The study made it possible to shape the initial problem in order to define its main focus, which ended up directing it to a problem currently felt in academic environments of a technological nature.

An integration and configuration automation system for tools used in academic environments was the solution conceived and on which most of the work done in this dissertation is focused. The need to perform this automation allowed to realize the effort and complexity of the current process, which is done manually. Therefore, there was a whole process of design, Chapter 4, and implementation, Chapter 5, of a prototype in order to try to solve the problem under study, as well as the verification and evaluation of its results, Chapter 6. In this way, it was possible to realize if the solution designed responded to the difficulties brought by the problem.

7.1 Research Questions

At the beginning, the problem under study was broken down into simpler questions in order to be able to provide an answer to the uncertainties associated with it.

RQ1: Would an automated tool be quicker and less error-prone when performing configurations tasks than a traditional manual approach?
After concluded the solution development, it is possible to verify that the performance it provides to configure various tools is way superior than the traditional manual process. Besides, in addiction to be less error-prone than the manual process, it also provides early error detection, which means that if the input data given to the system is invalid, it will be reject and not propagate in cascade amongst the tools that the system is responsible to configure.

RQ2: Is an automated approach viable in the future in terms of maintenance and reliability, when the number of components to configure starts to increase or a manual approach is preferable?
Once the solution was completed and, taking advantage of one of the tools it handles, it was possible to evaluate but above all to corroborate that the maintainability and reliability factors are not compromised.

An automated approach if well designed and implemented, such as the one developed, regardless of the number of tools to be integrated and configured, the

maintainability and reliability factors will always remain linear, not representing a problem as the solution evolves.

7.2 Contributions

The overall solution and all the work made towards it represents a contribution to the better understand about real problems faced by technological academic environments and what can be made to mitigate them.

Since the work was based on three main tools, namely GitLab, Jenkins and SonarQube, the resulting solution also contributes to the understanding about the limitations that this tools APIs' have, while also allowing to understand all the backbone necessary to configure and integrate them from scratch. In addiction to the mentioned tools, all the other services that are part and support the solution were also built from scratch, thus contributing to understand the evolving process of creating a new software system.

Regarding the implemented prototype itself [1], it is an important step to encourage the adoption of this type of automation systems on real academic environments, since the benefits this systems bring are immediately visible. Not only they make the teachers' work easier, but they also reduce the possibility of human error. All of this combined can make some of the ordinary processes of technological academic environments much more robust and flexible to changes.

7.3 Limitations

Even tough the main goals of this work were successfully achieved, there are some limitations that should taken into consideration:

Technological Discrepancies The overall idea of the solution is to automate the configuration and integration of a set of tools in order to ease that process on academic environments. This automation was decided to be made using the respective tools API's. However, some of the tools API's lack functionalities, resulting in some actions only being executable by the User Interface (UI). This led to some of the initial requirements not being able to be fully developed.

External Hardware Dependency The developed solution was hosted inside a Virtual Machine (VM) created on a local machine, meaning that the resources available were limited. This represents a bottleneck since it directly affects how the system behaves in heavy load situations, since it might run out of resources, or even make some ordinary tasks take longer than expected. The best approach to this situation would be to migrate the solution to one of the various cloud providers existent on the market. However, it is important to keep in mind that such a migration would mean monetary costs.

7.4 Future Work

In the same sense, although the main objectives were achieved, a set of future work tasks were identified in order to continue the solution and improve it:

Jenkins User Management Jenkins was the tool where more difficulties urged when integrating with its Application Programming Interface (API). That combined to the tight schedule led to that feature not implemented. However, the Jenkins service is very maneuverable and flexible, making a future integration to support users management very simple. Therefore, one of the future tasks would be to

[1] The implemented prototype is considered an open-source project and its source code can be found on GitLab: `https://gitlab.com/software_sandbox/pipeline-automation.git`

provide the Jenkins service a way to support user management, for example, by using a Robotic Process Automation (RPA) tool.

SonarQuqbe Metrics and Quality Gates Due to the tight schedule and the amount of time invested on exploring the various tools API's SonarQube metrics and quality gates configurations were not automated, requiring manual actions to be performed. Since one of the main goals it to automate the full process, a future task required is to complement the solution allowing it to perform those configuration steps.

Expand the Overall Range of Features Since the implemented solution is a prototype, despite providing the main features required, some of them could be more tailored to certain specific needs. Additionally, some of the provided features only allow a simple configurations, not taking full advantage of the versatility offered by the different tools API's. As such, in the future, the solution should be upgraded in order to provide more features and enhance the ones that already has.

Move Towards the Cloud A large-scale adoption of the solution would imply very strict performance ratings, and would increase the overall resources needed to be able to safely scale the various services, thus being able to respond to larger load volumes. That, currently, cannot be achieved by using a local machine. Therefore, in the future will be necessary to move the solution towards cloud computing, thus giving it more freedom in terms of resources as well as robustness, performance and reliability.

Real Life Scenarios Even though the solution developed took into consideration real life use cases, there is nothing better than test it on a real computer engineering department, in order to assess its features and understand what additional requirements would be necessary to fulfill the day-to-day. As such, a future task would be to perform a trial run of the solution in a real academic environment.

7.5 Personal Remarks

The motivation provided by the problem under analysis was, without a doubt, one of the key factors throughout this dissertation that, in general, met the main objectives initially proposed.

In addition, the work carried out allowed the drawing of important conclusions and future actions that will certainly bring a positive impact to technological academic environments, particularly for teachers and students, since the problem addressed is a difficulty that is felt nowadays. Therefore, it is hoped that the prototype developed can evolve and serve as a starting point for a solution that may be adopted in the near future.

Bibliography

[1] Isam Mashhour Al Jawarneh et al. "Container orchestration engines: A thorough functional and performance comparison". In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–6.

[2] Gaurav Aroraa. *Building Microservices with. NET Core 2.0: Transitioning monolithic architectures using microservices with. NET Core 2.0 using C# 7.0*. Packt Publishing Ltd, 2017.

[3] *Bamboo Continuous Integration and Deployment Build Server*. URL: https://www.atlassian.com/software/bamboo (visited on 02/27/2022).

[4] Lawrence Chung et al. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media, 2012.

[5] *Coverity SAST Software | Synopsys*. URL: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html (visited on 02/27/2022).

[6] Ms. Renuka Narsingh Ram Dr. Shiv Kumar Goel. "A Comparative Study of Agile & Devops Methodology". In: (2019). DOI: http://doi.org/10.22214/ijraset.2019.6439.

[7] Neil A Ernst et al. "What to Fix? Distinguishing between design and non-design rules in automated tools". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2017, pp. 165–168.

[8] Sendy Ferdian et al. "Continuous Integration and Continuous Delivery Platform Development of Software Engineering and Software Project Management in Higher Education". In: *Jurnal Teknik Informatika dan Sistem Informasi* 7.1 (2021).

[9] Shikha Gautam. "Comparison of Java Programming Testing Tools". In: *International Journal of Engineering Technologies and Management Research* (2018), pp. 66–76.

[10] *GitLab CI/CD | GitLab*. URL: https://docs.gitlab.com/ee/ci/ (visited on 02/27/2022).

[11] Unmesh Gundecha and Satya Avasarala. *Selenium webdriver 3 practical guide: End-to-end automation testing for web and mobile browsers with selenium webdriver*. Packt Publishing Ltd, 2018.

[12] Sanjay Hardikar, Pradeep Ahirwar, and Sameer Rajan. "Containerization: Cloud Computing based Inspiration Technology for Adoption through Docker and Kubernetes". In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. IEEE. 2021, pp. 1996–2003.

[13] David Hearnden et al. "Automating software evolution". In: *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE. 2004, pp. 95–100.

[14] Alan R Hevner et al. "Design science in information systems research". In: *MIS quarterly* (2004), pp. 75–105.

[15] My Högblom and Viktor Green. "Version Control Systems in Corporations: Centralized and Distributed-An explorative case study into the corporate use of version control systems". In: (2015).

[16] *IEEE Xplore Full-Text PDF:* URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=9761317 (visited on 06/26/2022).

[17] Pavan Sutha Varma Indukuri. "Performance comparison of Linux containers (LXC) and OpenVZ during live migration". PhD thesis. Doctoral dissertation, Master's thesis, Blekinge Institute of Technology, Sweden, 2016.

[18] Peter A Koen et al. "Fuzzy front end: effective methods, tools, and techniques". In: *The PDMA toolbook* 1 (2002), pp. 5–35.

[19] Ákos Kovács. "Comparison of different Linux containers". In: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE. 2017, pp. 47–51.

[20] Valentina Lenarduzzi et al. "Are sonarqube rules inducing bugs?" In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 501–511.

[21] Linda A Macaulay. *Requirements engineering*. Springer Science & Business Media, 2012.

[22] Manoj Mahalingam. *Learning Continuous Integration with TeamCity*. Packt Publishing Ltd, 2014.

[23] Ruth Malan, Dana Bredemeyer, et al. "Functional requirements and use cases". In: *Bredemeyer Consulting* (2001).

[24] Diego Marcilio et al. "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 209–219.

[25] Kief Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[26] Kıvanç Muşlu et al. "Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 334–344. ISBN: 9781450327565. DOI: 10.1145/2568225.2568284. URL: https://doi.org/10.1145/2568225.2568284.

[27] Alexander Osterwalder and Yves Pigneur. *Business model generation: a handbook for visionaries, game changers, and challengers*. Vol. 1. John Wiley & Sons, 2010.

[28] Alexander Osterwalder et al. *Value proposition design: How to create products and services customers want*. Vol. 2. John Wiley & Sons, 2015.

[29] I Palcic and B Lalic. "Analytical Hierarchy Process as a tool for selecting and evaluating projects." In: *International Journal of Simulation Modelling (IJSIMM)* 8.1 (2009).

[30] Reinhold Plösch et al. "A method for continuous code quality management using static analysis". In: *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE. 2010, pp. 370–375.

[31] Nigel Poulton. "The Kubernetes Book". In: *Independently published* (2017).

[32] K Siva Prasad Reddy. *Beginning Spring Boot 2: Applications and microservices with the Spring framework*. Apress, 2017.

[33] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[34] Rizki Rizki, Andrian Rakhmatsyah, and M Arief Nugroho. "Performance analysis of container-based hadoop cluster: Openvz and lxc". In: *2016 4th International Conference on Information and Communication Technology (ICoICT)*. IEEE. 2016, pp. 1–4.

[35] Maria A Rodriguez and Rajkumar Buyya. "Container-based cluster orchestration systems: A taxonomy and future directions". In: *Software: Practice and Experience* 49.5 (2019), pp. 698–719.

[36] Ekaterina Shmeleva et al. "How Microservices are Changing the Security Landscape". In: (2020).

[37] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O'Reilly Media, Inc.", 2011.

[38] Diomidis Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[39] *The DevOps Intelligence Platform - Codacy | Codacy*. URL: https://www.codacy.com/ (visited on 02/27/2022).

[40] James Turnbull. *The Docker Book: Containerization is the new virtualization.* James Turnbull, 2014.

[41] Solomiya Yatskiv et al. "Improved method of software automation testing based on the robotic process automation technology". In: *2019 9th International Conference on Advanced Computer Information Technologies (ACIT).* IEEE. 2019, pp. 293–296.

Appendix A

Analytic Hierarchy Process Steps

The bellow files present all the steps made while using the AHP method.

AHP

Pairwise comparisons

	Performance	Maintainability	Reliability
Performance	1.00	0.20	0.20
Maintainability	5.00	1.00	1.00
Reliability	5.00	1.00	1.00
SUM	11.00	2.20	2.20

Normalized comparison matrix and estimated weights

	Performance	Maintainability	Reliability	Weights
Performance	0.09	0.09	0.09	0.09
Maintainability	0.45	0.45	0.45	0.45
Reliability	0.45	0.45	0.45	0.45
Sum	1.00	1.00	1.00	1.00

Consistency Test

1.00	0.20	0.20	0.09	0.27
5.00	1.00	1.00	0.45	1.36
5.00	1.00	1.00	0.45	1.36

STEP2

3
3
3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

STEP3 3

STEP4 Consistency | CI | 0

STEP 5 CR=CI/RI | 0

Matrices with alternatives

Performance	System from scratch	Using existing tools	Combine both
System from scratch	1.00	5.00	5.00
Using existing tools	0.20	1.00	0.20
Combine both	0.20	5.00	1.00
Sum	1.40	11.00	6.20

Maintainability	System from scratch	Using existing tools	Combine both
System from scratch	1.00	5.00	3.00
Using existing tools	0.20	1.00	0.33
Combine both	0.33	3.00	1.00
Sum	1.53	9.00	4.33

Reliability	System from scratch	Using existing tools	Combine both
System from scratch	1.00	5.00	3.00
Using existing tools	0.20	1.00	0.33
Combine both	0.33	3.00	1.00
Sum	1.53	9.00	4.33

Matrices with normalized alternatives

Performance	System from scratch	Using existing tools	Combine both	Weights
System from scratch	0.71	0.45	0.81	0.66
System using existing tools	0.14	0.09	0.03	0.09
Combine both	0.14	0.45	0.16	0.25
				1

Maintainability	System from scratch	Using existing tools	Combine both	Weights
System from scratch	0.65	0.56	0.69	0.63
System using existing tools	0.13	0.11	0.08	0.11
Combine both	0.22	0.33	0.23	0.26
				1

Reliability	System from scratch	Using existing tools	Combine both	Weights
System from scratch	0.65	0.56	0.69	0.63
System using existing tools	0.13	0.11	0.08	0.11
Combine both	0.22	0.33	0.23	0.26
				1.00

Matrices with global priorities

	Performance	Maintainability	Reliability	
System from scratch	0.66	0.63	0.63	0.09
Using existing tools	0.09	0.11	0.11	0.45
Combine both	0.25	0.26	0.26	0.45

System from scratch	0.64
Using existing tools	0.10
Combine both	0.26

Appendix B

Orchestrating a pipeline of tools to support teaching - Survey

The bellow files present the results of the survey performed amongst Instituto Superior de Engenharia do Porto (ISEP) teachers, in order to support the opportunity identification 3.1.1.

Do you think that configuring and integrating the tools required for the various student groups is a very time-consuming task? (Take into consideration the LAPR context)	Being the current process manual, performing a cascade change (move a student from one group to another), meaning that a modification in one tool would have to be reflected across all the other tools as well, is a demanding and time-consuming task, sometimes error-prone?	Do you think that there would be immediate benefits if the teachers had a system that could automate the integration and configuration of tools that are required for the students work? (Take into consideration the LAPR context)	What are the most commonly experienced problems when setting the needed tools for students? (Take into consideration the LAPR context)
Yes	Yes	Yes	
Yes	Yes	Yes	
No	Yes	Yes	
			1. When a student loose their group partners he has group members becomes a big problem. 2. Some Erasmus students don't have all disciplines as well some portuguese students. 2. If members of a group is not productive it will affect all group grades
Yes	Yes	Yes	Team Management (credentials and authorization); setting up project's particularities on each different platform; having global dashboards;
Yes	Yes	Yes	Time, internet connection, sw/hw installation/compatibility, it/is literacy, licences
Yes	Yes	Yes	
Yes	Yes	Yes	Getting institutional support.
Yes	Yes	Yes	The manual configuration is prone to errors.
No	Yes	Yes	

Yes	Yes	Yes	Time consuming
Yes	Yes	Yes	Sometimes too many tools to manage
Yes	Yes	Yes	At first glance, they may seem like confusing tools with similar functions. Perhaps proposing a practical case of configuration and use, oriented to the basic functionality of the tools, could contribute to help you take more advantage of their use.
Yes	Yes	Yes	Multiple operating systems, development environments and system configurations across the students difficults the setup of IDEs and required APIs, needing special help to perform those task, delaying the development of the project and exercices.
Yes	Yes	Yes	
Yes	Yes	Yes	

Appendix C

Solution Performace Tests

The bellow images present a partial output generated by the solution when performing the load tests, in order to assess its performance.

Figures C.1 and C.2 present a partial output at the time of the initial setup from the Repository application, Figure C.3 from Jenkins application, Figures C.4 and C.5 from Sonar application and finally Figure C.6 from both Repository and Sonar applications when moving a user.

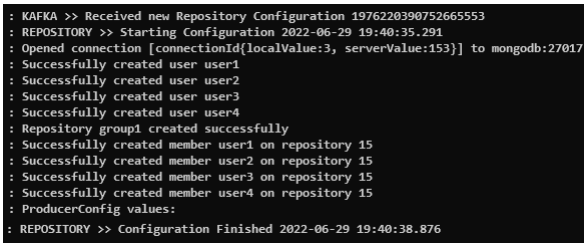

FIGURE C.1: Repository application partial output

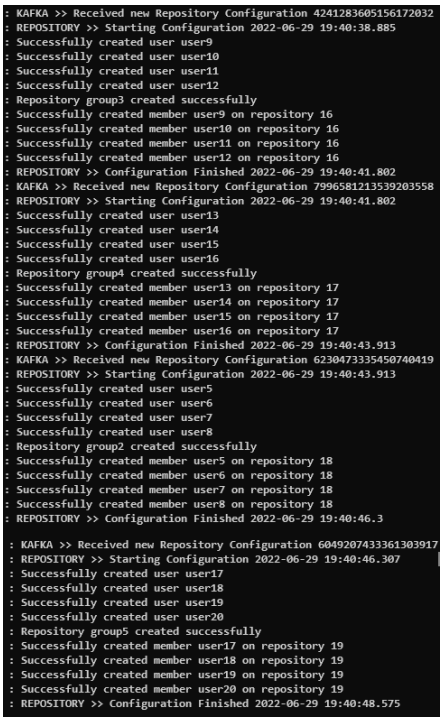

FIGURE C.2: Repository application partial output, continuation

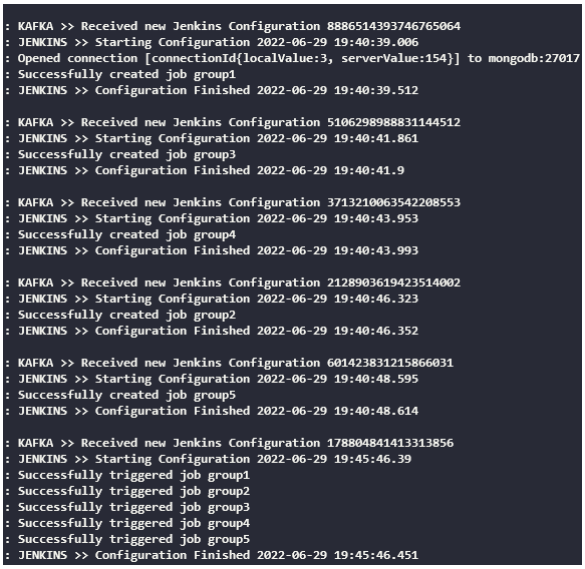

FIGURE C.3: Jenkins application partial output

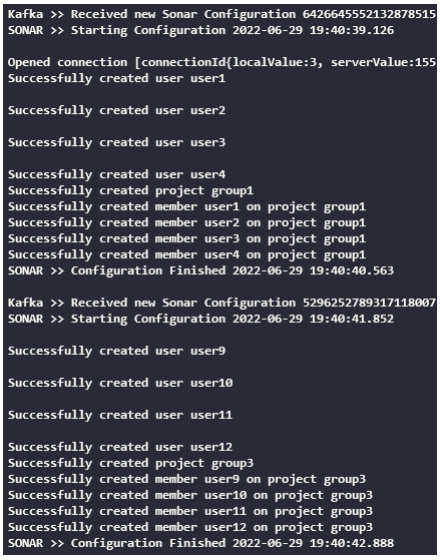

FIGURE C.4: Sonar application partial output

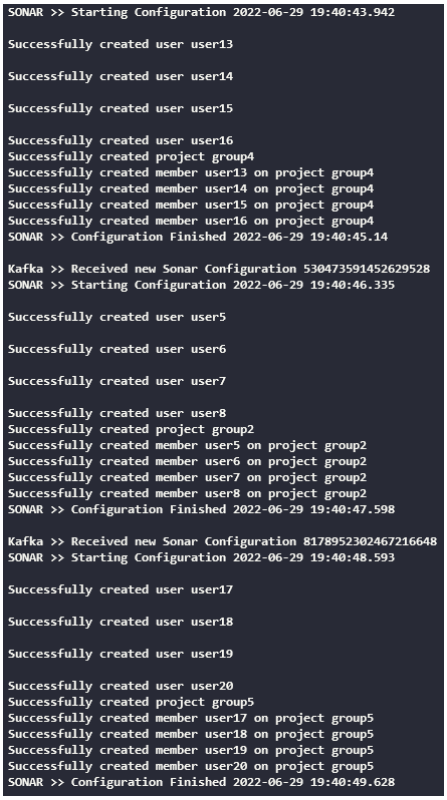

FIGURE C.5: Sonar application partial output, continuation

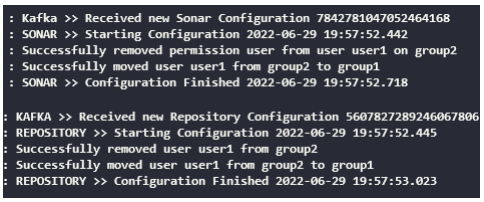

FIGURE C.6: Repository and Sonar applications partial output when moving a user and changing its permissions