Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Andrade Alves

Software defined applications: A DevOps approach to monitoring

February 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Andrade Alves

Software defined applications: A DevOps approach to monitoring

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António Luís Pinto Ferreira de Sousa

February 2022

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

This document marks the end of a five-year journey that made a broad and positive impact on both my life, and of the people who are closest to me. Reaching this milestone and getting ready for a new journey, requires showing my gratitude to all those who have supported and inspired me to end this cycle.

First of all I would like to thank my supervisor, Professor António Sousa, for all the help, support and advice during the past two years, and for always driving me to new and exciting challenges.

To Professor Vítor Fonte, and Professor João Marco, for all the advice and guidance while developing new projects at INESC TEC.

To my fellow researchers at the Mobile ID project, a sincere word of appreciation for the hard work, and team effort to complete all of our goals.

To the friends I have made during the past five years at UMinho, a special thank you for all the unique moments we have experience together and that have contributed to who I am as a person at the end of this chapter.

To my friend Afonso, for the many reflections, coffees and sincere friendship ever since high school.

To my girlfriend, for the unfailing support, care and continuous encouragement throughout my years of study, and specially through the most challenging times.

Lastly, but not least, I would like to express my gratitude to my family, for supporting me and encouraging me throughout this endeavour. To both of my parents, I am grateful for the opportunities they have provided and for showing me the meaning of sacrifice, hard work and integrity in order to achieve my goals. This recognition extends to my brother, as well as my grandparents.

To all of you who made this journey possible, thank you.

ABSTRACT

DevOps presents a mix of agile methodologies that allow an application's release cycle to be shortened. This translates into a faster delivery of value to the stakeholders.

However, the value creation chain does not finish at the end of that cycle. It is necessary to monitor the artifacts produced at a system level, and at the application level, in order to ensure the compliance of the functional and non functional requirements.

Today, there seems to be a clear separation between the monitoring process and the application development process. As the development and operations processes have merged in DevOps, this dissertation pretends to investigate how to integrate several aspects of monitoring into the regular lifecycle of an application's development.

The inclusion of external services further emphasizes the need to include an observability component into an infrastructure.

The main goal of this dissertation is to develop a solution for the deployment of an infrastructure using state-of-the-art technologies and frameworks, while also providing observability to the system and to the applications running on it.

To do so, it required the investigation of the methodologies and concepts that are the base of the software development lifecycle, focusing on the latter stages of that process: the deployment, and monitoring phases.

These methodologies and concepts were complemented with the study of state-of-the-art technologies and frameworks that aim to ease the burden of setting up an infrastructure quickly and with the necessary tools to evolve it after the initial setup and with each new software release. Furthermore, it also involved the research of tools that enable the collection of metrics from applications, as well as processing such data and displaying it in useful ways for operators and stakeholders.

In this context, this dissertation aims to provide a solution for the deployment of MobileID applications at INESC TEC, using the Mobile Driving Licence as the primary case study. The proposed design and implementation with a container orchestration framework and CI/CD pipelines, enables faster development of different MobileID applications, while also providing continuous monitoring to the deployments.

With this implementation, it was possible to assess how container orchestration frameworks provide greater flexibility to applications, and how this observability can be augmented with the use of dedicated monitoring systems.

KEYWORDS DevOps, Software Development, Application Deployment, Monitoring, Continuous Integration, Continuous Delivery, Virtualization, Container Orchestration.

RESUMO

DevOps baseia-se na utilização de um conjunto de metolodogias ágeis que permitem encurtar o ciclo de desenvolvimento de uma aplicação de forma a que as alterações efetuadas pelos programadores se traduzam no valor desejado pelas partes interessadas. No entanto, a criação de valor não termina na parte final desse ciclo. É necessário monitorizar os artefactos produzidos tanto a nível de sistema, como a nível aplicacional, de forma a garantir o cumprimento de requisitos funcionais e não funcionais.

Todavia, parece existir uma separação entre o processo de monitorização e o processo de desenvolvimento de aplicações. Tal como os processos de desenvolvimento e de operações se uniram no conceito de *DevOps*, pretende-se também investigar como será possível integrar vários aspetos de monitorização no ciclo normal de desenvolvimento de uma aplicação.

O principal objetivo desta dissertação é desenvolver uma solução de operacionalização de infraestruturas de suporte a aplicações com recurso às tecnologias e ferramentas mais adequadas. Esta solução deverá ser acompanhada, em paralelo, por mecanismos de observabilidade dessa infraestrutura e das aplicações que nela são executadas.

Para isso, foi necessária a investigação de metodologias e conceitos que formam a base do processo de desenvolvimento de *software*. O foco esteve nas partes finais do processo: a fase de *deployment* e a de monitorização.

Estas metodolodogias e conceitos foram complementados com o estudo de tecnologias e ferramentas que pretendem facilitar o processo de montar uma infraestrutura rapidamente, bem como permitir a evolução da arquitetura inicial consoante os subsequentes lançamentos de aplicações.

Para além disso, também envolveu a pesquisa de ferramentas que permitem extrair e armazenar métricas de aplicações, bem como processar essa informação e disponibilizá-la em formato útil quer para operadores, quer para outras partes interessadas.

Neste contexto, esta dissertação pretende desenvolver uma solução que permita efetuar o *deployment* de aplicações de Identidade Digital no INESC TEC, utilizando a Carta de Condução Móvel como caso de estudo. A arquitetura proposta, e a respetiva implementação com recurso a um orquestrador de *containers* e *pipelines* de CI/CD, permite o desenvolvimento mais ágil de novas aplicações de Identidade Digital, e proporciona monitorização contínua a cada iteração do desenvolvimento.

A partir do resultado prático obtido, foi possível aferir de que forma os orquestradores de *containers* permitem melhorar a observabilidade de aplicações, e de que forma ela pode ser aumentada com recurso a sistemas dedicados de monitorização contínua.

PALAVRAS-CHAVE *DevOps*, Desenvolvimento de Software, *Deployment* de Aplicações, Monitorização, Integração Contínua, Entrega Contínua, Virtualização, Orquestradores de *Containers*.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACL Access Control List. 54
AMQP Advanced Message Queuing Protocol. 28
API Application Programming Interface. 16, 26, 28, 30, 31, 44, 47, 57, 61, 62

CA Certificate Authority. 48
CCTA Central Computing and Telecommunications Agency. 6
CI Continuous Integration. 20
CI/CD Continuous Integration/Continuous Delivery. 41, 54, 57, 64
CLI Command Line Interface. 18
CNCF Cloud Native Computing Foundation. 43, 44
CNI Container Network Interface. 52
CPU Central Processing Unit. 12, 13, 17, 23, 24, 25, 36, 43, 61

DHCP Dynamic Host Configuration Protocol. 28
DNS Domain Name System. 13, 16, 28, 38, 51
DSL Domain-Specific-Language. 18, 19

HCL Hashicorp Configuration Language. 22, 44, 49, 54, 55, 59
HTTP Hypertext Transfer Protocol. 24, 25, 28, 31, 44, 57, 61
HTTPS Hypertext Transfer Protocol Secure. 31

I/0 Input/Output. 12, 23
IaaS Infrastructure as a Service. 14
IaC Infrastructure as Code. 14, 21, 33, 42
ICMP Internet Control Message Protocol. 28
IP Internet Protocol. 14, 16, 47, 51, 52, 53, 56
IT Information Technology. 6, 17, 18, 32
ITIL Information Technology Infrastructure Library. 4, 6, 17, 37, 38

JSON JavaScript Object Notation. 19, 26, 36, 45, 56
JVM Java Virtual Machine. 19

KVM Kernel-based Virtual Machine. 14

mDL Mobile Driving Licence. 29, 31
mTLS Mutual Transport Layer Security. 48, 49, 52

NFS Network File System. 28
NNTP Network News Transfer Protocol. 25

OSI Open Systems Interconnection. 38

PaaS Platform as a Service. 12
PKI Public Key Infrastructure. v, 31, 33, 36, 43, 44, 45, 47, 48, 49
POP Post Office Protocol. 25
PQL Puppet Query Language. 19

RAM Random Access Memory. 12, 17
RBAC Role-Based Access Control. 22
REST Representational State Transfer. 15, 30

SAN Storage Area Network. 22
SLA Service Level Agreement. 24, 39, 61
SLI Service Level Indicator. 24
SLO Service Level Objective. 24
SMS Short Message Service. 31
SMTP Simple Mail Transfer Protocol. 25
SSH Secure Shell. 19, 31, 46, 47
SSL Secure Sockets Layer. 19
SVC Service Value Chain. 8
SVS Service Value System. 6, 7

TTL Time to Live. 50

URL Uniform Resource Locator. 55, 56, 60

VCS Version Control System. 36, 42, 57
VM Virtual Machine. 11, 12
VPN Virtual Private Network. 31, 36, 38

XP Extreme Programming. 20

YAML YAML Ain't Markup Language. 15, 20, 36

1

INTRODUCTION

DevOps presents a mix of agile methodologies that allowed the application's release cycle to be shortened. This translates to a faster delivery of value to the stakeholders, and quick iterations on an application's source code.

However, the value creation chain does not finish at the end of that cycle. It is necessary to monitor the artifacts produced at a system level, and at the application level, in order to ensure the compliance of the functional and non functional requirements.

Today, there seems to be a clear separation between the monitoring process and the application development process. As the development and operations processes have merged in DevOps, this dissertation pretends to investigate how to integrate several aspects of monitoring into the regular lifecycle of an application's development.

The inclusion of external services further emphasizes the need to include an observability component into an infrastructure. These type of components in an architecture can result in a number of errors or other unexpected situations that could prove difficult to diagnose without appropriate monitoring tools.

1.1 MOTIVATION

The main motivational factor behind this dissertation's work was the development process for a proof of concept for the Mobile Driving Licence according to the ISO/IEC FDIS 18013-5 international standard. This standard specifies the interfaces in an implementation of a driving licence associated with a mobile device.

The process of developing the issuing authority infrastructure for the project required the manual deployment of every updated version on a server by an operator. This task was repeated for every new version, and therefore it was a task that could be targeted for automation, in order to prevent configuration drifts and reduce human errors.

As each deployment was used by multiple users and developers, and issues started to be raised, it highlighted the need to preemptively detect any anomaly. As these situations occur on a regular basis on any released version, it also motivated the design and implementation of a system that could be used to monitor deployments regardless of the application that was targeted.

Finally, the project started to branch out to generic Mobile Documents, in order to target different use cases beyond the Mobile Driving Licence. As new applications are built on top of the existing infrastructure, the manual management of the increasing number of services becomes unfeasible for a modestly sized team. Adopting a

framework that relieves the burden of managing those services manually is crucial to increase the velocity of development.

1.2 GOALS

The main goal of this dissertation is to develop a solution for the deployment of an infrastructure using state-of-the-art technologies and frameworks, while also providing observability to the system and to the applications running on it.

This requires the study of DevOps tools and their relationship with the release cycle; how software is currently delivered; the suitability of containers versus virtual machines and how to orchestrate multiple applications within a single infrastructure.

It also involves the exploitation of tools that allow the retrieval of telemetry data from applications; the study of processes that allow monitoring to co-exist with the continuous development of applications; the operation of tools that automatically manage an application's elasticity; and how to design and implement monitoring for the infrastructure and its applications in order to meet agreed targets.

1.3 DOCUMENT STRUCTURE

This dissertation is composed of 6 Chapters, including this introductory one.

Chapter 2 discusses the dissertation's body of knowledge, exposing software development and DevOps practices, as well as tools and frameworks that apply those theoretical practices. The Chapter also focuses on the monitoring phase of the software development lifecycle, in order to understand how to integrate observability in applications.

Chapter 3 presents the case study that motivated the development of this dissertation's work. It starts by describing the technical specifications that drove the Mobile ID project, and continues with the design of the backend components. It ends by specifying the project's infrastructure and its limitations as well as the deployment process for each component and its drawbacks.

Chapter 4 proposes an architecture for three components of the case study: the general infrastructure, the monitoring infrastructure, and the deployment process. It starts with the technical requirements for each component, continues with their architectural overview, and ends with the deployment process for services deployed in the new infrastructure.

Chapter 5 describes the implementation process of the design proposed in Chapter 4. It starts by discussing the selected tools among the ones described in Chapter 2, and continues with the steps that compose the automated deployment of the designed infrastructure. It ends by evaluating the final system, how it improves the observability of the case study, and how container orchestration systems contribute to the monitoring process in the software development lifecycle.

Finally, Chapter 6 summarizes the main conclusions and contributions provided by the developed system. Furthermore, it proposes improvements that could be developed in the future to benefit the system and the developers that interact with it.

2

STATE OF THE ART

The first step in meeting the main goal of developing a solution for the deployment and configuration of an infrastructure is to identify and review literature that presents the most relevant concepts related to software development, deployment and configuration, as well as application and system monitoring. These concepts will include both technical definitions, along with technical methodologies. These methodologies will be used as the main driver for the proposed design on Chapter 4.

To complement this research, there will be a presentation of a set of tools and technologies whose aim is to facilitate the application of the researched methodologies. The discussion of this set is crucial in the decision making process of which tools will be used during this dissertation's work, as reasoned in Section 5.1. As a result of this discussion, and the proposed design on Chapter 4, it will be possible to implement a proof of concept system as described on Chapter 5.

The Chapter starts with an introduction to the software development practice in Section 2.1, and how it evolved into the agile methodologies that are broadly used nowadays. It continues with an introduction of the ITIL practices and which subset is related to this dissertation's goals. This first section ends with the activities that make up the deployment phase of the development lifecycle.

Section 2.2 introduces virtualization concepts and a comparison is made between the two major options for virtualizing applications: virtual machines and containers. This section ends with the description of the components of three popular container orchestration tools used in the market: Docker Swarm, Nomad and Kubernetes.

Section 2.3 presents the principles of continuous delivery, and includes a description of three technologies that promote the principles of configuration management: Chef, Puppet and Ansible. It also presents the benefits of adopting version control systems to manage systems' configurations.

In Section 2.4, there is a contextualization of the continuous integration practice, and how it relates with the principle of treating infrastructure as code. This section ends with the presentation of the open source tool Terraform that enables the provisioning of components of an infrastructure.

Finally, Section 2.5 describes the principles that should guide a monitoring solution for an application. It breaks down monitoring according to its type, and presents metrics that should be recorded for most applications. This section ends with the discussion of three open source tools that enable the collection of telemetry data in applications and infrastructure: Nagios Core, Prometheus and Elastic Stack.

2.1 SOFTWARE DEVELOPMENT

Software development is referred to as a "process by which user needs are translated into a software product. This involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. These activities can overlap or be performed iteratively." [94].

According to Sommerville, every software development process must include four core activities [146]:

- Software specification;
- Software design and implementation;
- Software validation;
- Software evolution.

In a waterfall process, the author suggests that these activities can be represented in 5 different processes: requirements specification, software design, implementation, integration and maintenance. Each process requires a different set of activities, skills and people, creating independent teams of development and operations tasks.

However, this siloed approach has shown to be the cause of problems around software development in the industry, and thus new approaches based on DevOps principles have become widespread [86].

2.1.1 *DevOps*

DevOps as a movement emerged around 2009, a time when operations and development teams alerted about the problems that arose from their organizational and functional separation [31]. That year, at the O'Reilly Velocity Conference, John Allspaw and Paul Hammond presented "10+ Deployer per Day: Dev and Ops Cooperation at Flickr", which highlighted the contentious nature of Development and Operations during software deployment [114].

DevOps is considered a set of practices that automates and integrates the processes between software development and operations teams, increasing an organization's ability to deliver applications and services at high velocity [16, 13].

These practices include adopting an agile approach to project management [51]. Delivering software in smaller batches allows the continuous evaluation of the project's requirements, plans and results, providing teams with the necessary information to respond to rapid changes. This involves adopting team frameworks such as Scrum and Kanban [120].

Scrum is a framework that creates a team environment where there are three major steps that are continuously repeated [141]:

1. The Product Owner prioritizes the work for a problem into a Product Backlog.

2. The Team turns a selected part of the work into an Increment of value during a Sprint.

3. The Team and the stakeholders inspect the result and adjust accordingly to the next Sprint

Kanban is the Japanese word for "billboard", and was originally developed by Toyota as a scheduling system to promote their lean manufacturing process [135]. It consists of a board that helps visualize workflow, and is broken up into three parts: to do, in progress and done [137].

Another principle of DevOps is "shifting left" [29]. By moving a variety of tests to the development process, instead of in a separate test or quality-assurance stage, developers can find and fix bugs faster due to the rapid feedback.

DevOps also advocates implementing automation into the software development pipeline [119]. This can be achieved through continuous integration and delivery, which includes automated testing of pushed changes and subsequent deployment to the end users. This can include unit testing, integration testing, end-to-end testing and performance testing.

Another important aspect of DevOps is making the infrastructure easier to rebuild than to repair [102]. This further reduces the number of ways in which the infrastructure can drift from the expected state into snowflakes.

Adopting a DevOps approach enables faster deployments which provide faster customer feedback. According to Nicole Forsgren, "elite teams deploy 208 times more frequently and 106 times faster than low-performing teams" [60].

2.1.2 *ITIL*

ITIL was first developed by the Central Computing and Telecommunications Agency (CCTA) at the end of the 1980's in Great Britain [147]. The goal of this agency was to develop recommendations for the provision of good quality IT services at a reasonable cost, culminating in the catalogue of best practices for organizations known today as ITIL.

ITIL 4 is the most recent edition of this set of practices, and it was published in February 2019 [19]. It is a completely revised framework in comparison to the previous version, and it is structured in two major components: the four dimensions model and the Service Value System (SVS).

The four dimensions of service management are representative of perspectives that should be applied to every service, and they are:

- Organizations and people;
- Information and technology;
- Partners and suppliers;
- Value streams and processes.

The first dimension involves the formal structures within an organization, including roles and responsibilities, as well as company culture; the second dimension is related to how information and tools can help create value,

both to individual service value streams as well as general service management; the third dimension covers the relationships with other organizations that are involved in the chain of delivering an organization's services; the fourth dimension introduces the activities and workflows the organization undertakes to enable value creation for the stakeholders.

The key element of the SVS is the Service Value Chain, which is "an operating model for service that covers all the key activities required to effectively manage products and services" and is depicted in Figure 1.

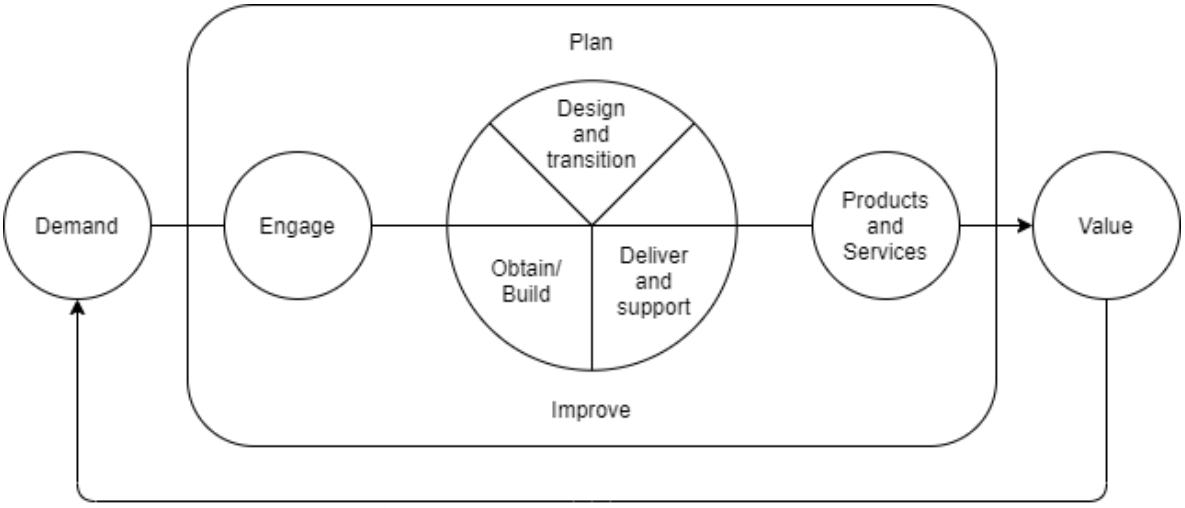

Figure 1: ITIL's Service Value Chain

This chain includes six activities that represent the steps taken to produce value: plan, engage, improve, design and transition, deliver and support, and obtain/build.

The plan activity aims to produce a set of strategic, tactical and operational plans that outline the vision and general direction for all the products and services throughout the organization; the engage activity is focused on understanding the stakeholder needs, and fostering a continuous engagement relationship with all of the organization's stakeholders; the improve activity ensures that the information and knowledge gathered in other activities is used to properly ensure the improvement of products, services and practices across the organization; the design and transition activity is concerned with meeting the stakeholder's expectations for quality, costs and time to market; the deliver and support activity ensures that the products and services are delivered and supported according to the agreements and specifications set between the organization and the stakeholders; the obtain/build activity is responsible for making service components available at the time and in the place they are needed along the service value chain, meeting agreed specifications.

The SVS includes a set of management practices that range from general management, service management and technical management. Each of these practices is defined as a "set of organizational resources designed for performing work or accomplishing an objective", and is subject to the four dimensions model. From these set of practices, there are four that are relevant to the relationship between DevOps and monitoring: two from the service management, and two from the technical management.

Monitoring and event management

The goal of the monitoring and event management practice is to monitor service components and keep a record of a select number of events throughout time. As a service management practice, its main contribution to the SVC resides in the improve, the design and transition, and the deliver and support activities. It improves the value chain as the observation of the environment contributes to proactively improving its stability; the monitoring data provides information about design decisions, as well as the transition status in a given environment; and it guides how an organization should manage support of events.

Release management

Release management is the practice of making "new and updated features and services available for use". As a service management practice, its main impact on the SVC is on the design and transition activity. It ensures that new or updated features are made available to customers in a supervised way.

Deployment management

The purpose of the deployment management practice is to "move new or updated software or any other component to one or more environments", such as production, testing or staging. As a technical management practice, it contributes to the SVC in both the design and transition, and the obtain/build activities. The deployed changes in a DevOps environment are commonly automated with a toolchain for continuous delivery, integration and deployment.

Infrastructure and platform management

The infrastructure and platform management practice includes the provisioning of "the technology needed to support activities that create value for the organization and its stakeholders", while also enabling "monitoring of technology solutions available to the organization". As a technical management practice, it has an emphasis on the design and transition, and the obtain/build activities. The information provided about the technology can benefit the final product or service, as well as the components that need to be obtained along the SVC.

2.1.3 *Software Deployment*

According to Carzaniga et al. [33], all the activities that make a software system available to use are referred to as software deployment. There is a total of 8 activities described, and they are releasing, installing, activating, deactivating, updating, adapting, deinstalling and dereleasing.

On the other hand, Group [71] points at 5 activities as being part of the deployment process: installation, configuration, planning, preparation and launch. It sees the process as uni-directional, and does not take into account the cyclic nature of the deployment process proposed by the previous author.

Release

The releasing activity includes the operations that are needed in order for the system to be packaged and distributed. The package shall have enough metadata to describe the resources on which the system depends. In contrast, Group [71] states that packaging the software is a precondition to the process of deployment.

Installation

Both authors present the installation activity. The former considers it the most complex of the deployment activities, because it requires the packaged software to be inserted and configured into the consumer site. The latter mostly follows the definition of the former, but makes it clear that this activity is not related to inserting the packaged software onto the computers on which it will execute. It is solely brought into a repository under the control of the deployment operator, in order to apply deployer defined policies'.

Activation

The activation activity covers the operations that are required to start up the software system. This can require external services or processes to be started before the software system itself is activated. This activity seems similar to the launch activity of the deployment process defined in Group [71]. In this activity, the application is brought to an executing state using the resources laid out in the package's metadata previously set.

Deactivation

As for the deactivation activity, it is described as the reverse of the activation and it encompasses all the operations required to bring the system into a halt.

Update

This activity is considered a special case of installation, as it also requires the transfer and configuration of the packaged software into the target environment. However, it is less complex as most of the dependencies have been resolved and can be performed while the previous version of the software is still active.

Adaptation

Adaptation is very similar to the update activity, as it implies a modification of a previously installed system. However, it differs in the event that triggers the activity. The update is triggered by the release of a new version by the software developers, whereas the adaptation is triggered by changes in the target environment, such as adding a new virtual machine to a cluster or removing a physical data volume.

Deinstallation

The deinstallation activity removes the packaged software from the target environment. This is usually done when the system is no longer required, and may involve reconfiguring other systems which are dependent on the one being deinstalled.

Derelease

As for the dereleasing activity, it is considered the final activity of the deployment process of a software system. The system is marked as obsolete and no further support is provided by the producer.

The diagram in Figure 2 depicts the relationship between the deployment activities proposed by Carzaniga et al. [33]. It begins with the release phase, followed by either decommissioning the system (with the derelease phase) or installing it on the consumer site. After this installation, the system needs to be started, which is possible with the activation activity. After being activated, the system can be deactivated to perform three different activities: to adapt, when the target environment is changed; to update, when a new version is released by the software developers; or to be deinstalled, when the system is no longer necessary. It is clear that any change to be made to the running system requires the system to not be activated. However, this unavailability can be solved with continuous delivery practices.

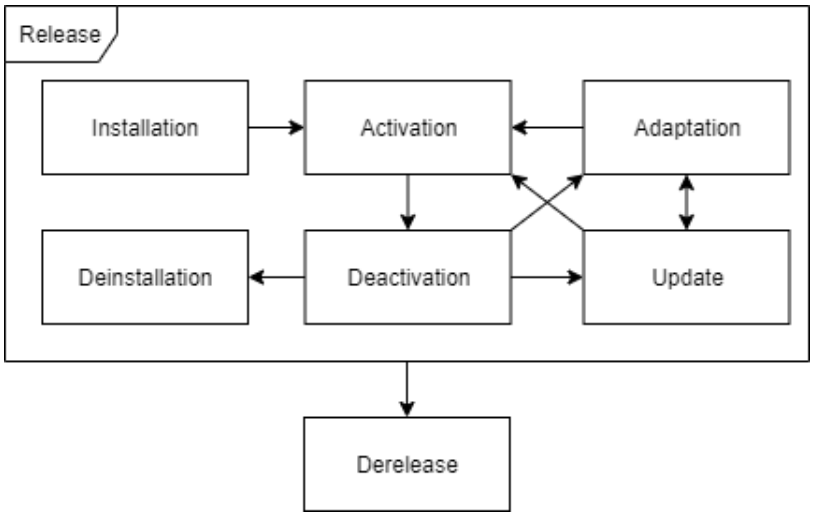

Figure 2: Activities of the deployment process

2.2 VIRTUALIZATION

Virtualization has been a topic of research for over 40 years [63], with the micro programming of processors being shaped by the task of emulation.

The author further emphasizes that the "emulation capabilities in a high speed processor, are examples to his more general approach to emulation". He concludes predicting that the "trend should be heightened in the future".

Pearce et al. [131] summarizes system virtualization as the "use of an encapsulating software layer that surrounds or underlies an operating system and provides the same inputs, outputs and behavior that would be expected from physical hardware".

Currently, there are two main options for adopting virtualized systems into the software deployment lifecycle: virtual machines and containers [97].

2.2.1 *Hardware Virtualization*

As each software system can be considered an independent component, it makes sense to deploy each component on its own physical computer. To set up the system, one would have to find and manage a physical space, power, cooling, network connectivity and then install an operating system and software dependencies before the application itself. This would then lead to several issues regarding the scalability, security or redundancy of the system, as each of these would require adding more computers. This process can take a very long time, and requires a considerate amount of resources.

This approach could also mean wasting resources, as an application could stand idle for 99% of the time and barely use any of the available resources. However, when the application would perform its more resource-intensive tasks, all the computer's power could be used, and therefore a very powerful machine would be put to waste during most of its lifetime. According to Magazine [110], x86 servers are running at 12% utilization of its operating system, which is considered highly inefficient.

Virtualization relies on software to simulate hardware functionality and create a virtual computer system [144]. This can be achieved using a Virtual Machine (VM), which is an isolated software container with an operating system and application inside. However, this machine cannot interact directly with the underlying physical computer.

Between the virtual machine and the physical computer resource, lies the hypervisor [22]. It is a thin software layer that dynamically allocates the computing resources to each virtual machine according to their configuration and needs [142].

The use of this abstraction provides 4 benefits: partitioning, by dividing the resources of a system between all the virtual machines running on the hypervisor; isolation, both in security, performance or failures from the underlying hardware or software; encapsulation, due to the possibility of saving the state of a virtual machine as a file; hardware independence, because a virtual machine can be provisioned in any physical server with the presence of a hypervisor.

Resorting to virtual machines also presents some drawbacks. Abstracting the system often includes a penalty in CPU, network and I/O performance. This can be further emphasized when dealing with over-provisioning of virtual resources. Security may also be at risk, in case a malicious user has access to the physical resources on which virtualized resources are running. Having multiple virtual machines running on a single physical resource

also increases the dependability on that machine. If a single machine fails, multiple virtual ones may fail in consequence [136].

There's also a drawback which is common with the physical infrastructure approach. When deploying an application to a virtual machine, if the application does not use most of the resources made available to it for the majority of its lifecycle, it is wasting those resources [97].

2.2.2 *Containers*

Containers are isolated and lightweight silos for running applications on the host operating system [145]. They use operating system features to isolate the processes, networking and file-system, so that it appears as a self-contained server environment [119].

The usage of namespaces creates a layer of isolation for the container from the other running processes on a system. From the container's perspective, a global resource appears as its own isolated instance [100]. Docker, a popular container engine, uses for its Linux implementation the namespaces relative to process isolation (pid), network interfaces (net), inter-process communication (ipc), file-system mount points (mnt) and kernel version identifiers (uts) [50].

Control groups (cgroups) allow the allocation of computational resources, such as CPU, RAM, Disk I/0 or network bandwidth, among groups of processes. This provides the ability to limit the resources used by each container and prioritise one over another [101].

Each virtual machine can have a different operating system installed, as the abstraction is at the hardware level. Multiple virtual machines can be run on top the same hypervisor, and they emulate the underlying hardware.

Container instances, on the other hand, share the underlying operating system kernel. This is why containers cannot run different operating systems other than their host system. They can, however, run different distributions of the same OS. This is the case of Linux distributions, where different distributions share the same Linux kernel [119].

A container image can be much smaller than a VM image because it doesn't need to include a full operating system.

In comparison to virtual machines, containers can be created and shut down very quickly. This is due to the fact that they only require the processes that make up the application to be started, and not booting up an entire virtual machine and initializing its operating system.

Another benefit of containers is that they include all the dependencies of an application into a self-contained piece of software, therefore improving portability.

2.2.3 *Container Orchestration*

With the rise in popularity of containerization systems like Docker, virtualizing applications became a bigger topic of interest to Platform as a Service (PaaS) providers [34]. As micro-services architectures rely on a large number

of very small applications connected with each other, the demand for an automated process for managing and orchestrating containers has emerged.

These frameworks allow the dynamic deployment, automation, scaling, monitoring and managing of multiple containers. This allows distributing workloads across hosts through a software layer that abstracts the complexity of the cluster, displaying the infrastructure as a pool of resources. These resources can be network topologies, storage volumes, secrets or application configurations and containers.

Docker Swarm

Docker Swarm is a cluster manager and orchestrator embedded in the Docker Engine [50].

A Swarm is composed of multiple physical or virtual nodes that are running the Docker daemon in swarm mode. These nodes can play two different roles simultaneously: managers or workers [11].

Managers are responsible for dispatching units of work to worker nodes. These units of work are named tasks. They also perform cluster management functions that are necessary to maintain the desired state of the swarm, such as electing a leader. This leader is selected among the manager nodes using the Raft consensus algorithm [124].

Worker nodes are responsible for receiving and executing tasks pushed from the manager nodes. An agent is executed in every worker node to report the current state of its assigned tasks to the manager.

A task consists of a Docker container and commands to run inside the container. Once it starts in a node, it cannot be moved to a different node. However, this doesn't mean that the functionality provided by that task stops being available in case the node fails or the container exits. This is possible due to the Service abstraction.

A Service is the definition of the tasks, and it is the core structure of the Swarm. It is defined by a container image and by the commands that shall be executed inside the running containers. It can be further configured with network connectivity, attached volumes, external services dependencies, memory and CPU restrictions, number of replicas and rolling update policy.

There can be two types of services: replicated or global. With a replicated service, a number of identical tasks are run within the swarm. With a global service, each node in the swarm runs one task of that service.

The services in the swarm are assigned a Domain Name System (DNS) entry, which provides internal load balancing in the cluster. There can also be external load balancing: a service can have an exposed port and it will be accessible externally in any node in the swarm.

Nomad

Nomad is an orchestration framework for containerized or non-containerized applications [108]. Its 1.0 version was released in October 2020 but it launched its first version alongside Kubernetes in 2015 [46].

A Nomad cluster can have machines running as two different types: Clients or Servers [75].

A Nomad Client is a machine in the cluster where tasks can be executed. The Nomad agent runs on this machine and it is responsible for monitoring any tasks assigned by the Servers.

A Nomad Server is a machine responsible for accepting Jobs, managing Nomad Clients and computing where tasks should be placed. There can be multiple Servers within a cluster in order to tolerate machine failures. As in Docker Swarm, the Raft algorithm is used to determine which Server is the leader.

The workloads that run in the cluster are named Jobs. A Job describes a desired state, and it is Nomad's responsibility to ensure that the actual state matches the declared desired state.

A Job may have multiple groups, and each group may have multiple tasks. A Group is a set of tasks that run on the same client node. A Task is the smallest execution unit of work in Nomad.

Each Task requires a Driver to execute and to provide resource isolation. Nomad includes four built-in enabled drivers: Docker, Java, QEMU and isolated exec.

The Docker driver provides Docker workflows on Nomad, adding port mapping and container images downloading.

The Java driver provides the ability to execute Java applications that are packaged into Jar files.

The QEMU driver provides a virtual machine runner that can utilize Kernel-based Virtual Machine (KVM) to exploit hardware virtualization features and improve performance.

The Isolated exec is used to execute a command for a task with the isolation primitives of the operating system. It is mostly used to call scripts or other wrappers.

In contrast to Kubernetes, Nomad does not provide service discovery nor secrets management. Instead, it suggests integrating other tools such as Consul and Vault which are also developed by HashiCorp.

Kubernetes

Kubernetes is an open-source platform for orchestrating and managing containerized applications and services. It was developed and introduced by Google on the 6th of June of 2014 as the successor of Borg, which was Google's container-oriented internal cluster manager [117, 152].

Kubernetes automates the deployment, scaling, load balancing, logging, and monitoring of containerized applications. It also provides features of an Infrastructure as a Service (IaaS), such as enabling a range of user preferences and configuration flexibility. This configuration can be set declaratively, which allows treating the Infrastructure as Code (IaC) with all the corresponding benefits.

POD A pod is the smallest deployable unit of computing in a Kubernetes cluster [103]. Each pod can have one or more containers, and they are deployed on the same node. As they are tightly coupled with one another, containers in the same pod share the network namespace and file-system volumes. A unique IP address is assigned to each pod.

A pod is designed to be an ephemeral entity, and its lifecycle is created and managed by a workload resource such as a Deployment, StatefulSet, DaemonSet or Job.

NODE A Kubernetes cluster consists of a set of nodes that run containerized applications. By having more than one node, Kubernetes can distribute the workload across them according to the desired state.

A node can be any type of machine, either physical or virtualized. In each one of them, three components can be found.

The kubelet is an agent responsible for managing pods, by making sure that pods are healthy and running properly. It warrants that each pod is as described by its specification.

The kube-proxy acts as a network proxy, and it implements part of the Service concept. It is the component responsible for maintaining network rules on the nodes, and what allows Pods to communicate with components inside and outside of the cluster.

The container run-time is a software layer required in every node, and it is responsible for communicating with the kernel, setting up cgroups, SELinux Policy and App Armor rules [112].

CONTROL PLANE To control the cluster, a control plane is used and it is composed of four different components that can run on any node. A fifth component may be present, but only if the cluster is running on a public cloud provider. When the cluster is running on premises, the cloud-controller-manager is not present.

The kube-apiserver exposes the Kubernetes API using a REST interface. This allows the configuration and validation of the cluster objects such as pods, services and controllers.

The etcd component works as a backing store for the cluster's data. It is a highly-available key value store [57].

The kube-scheduler watches the non-assigned Pods in the cluster and takes the decision as to where they will run. In order to take that decision, this component takes into account each Pod's resource requirements, software, hardware or policy constraints, affinity specifications, data locality, inter-workload interference and deadlines.

The kube-controller-manager is the component responsible for managing Controller objects. These include the Node Controller, which handles cases such as a node going down; the Replication Controller, which is responsible for managing the correct number of pods according to its template; the Endpoints Controller, which joins service objects and pods; and the Service Account & Token Controllers, that create accounts to access the cluster's namespaces.

DEPLOYMENT A Deployment allows describing the desired state of an application. It uses a YAML Ain't Markup Language (YAML) manifest to describe which container images the application uses, how many pods should be deployed and how the application should be updated [25].

This Kubernetes object makes the update and deployment process automated and repeatable, and it is entirely managed by the Kubernetes backend. The update can be performed using two different strategies: recreate or rolling update. The former removes all the running pods before creating new ones, while the latter won't remove old pods until there are enough new pods available to meet a specified threshold. The first approach creates downtime during the update, while guaranteeing that there won't be two different versions of a deployment running at the same time. The second one guarantees no downtime during the update process, but may cause issues for dependable services due to having 2 different versions of the same application running at the same time.

SERVICES A Service is an abstraction to expose applications running on pods. Each pod has a unique IP address, and as they are ephemeral, this address is not suitable for long-lived connections.

The Service allows decoupling an individual pod IP address from the functionality they provide, by offering an endpoint API that lets the Service be accessibly inside and outside the cluster.

This also provides load balancing capabilities, through a DNS service inside the Kubernetes cluster that acts as a proxy. With this approach, any pod in the cluster can reach another Service through its name and namespace.

There are 4 types of Services: ClusterIP, which makes the Service only reachable from within the cluster; NodePort, that exposes the Service on each of the cluster's node IP at a static port, making the Service reachable from outside the cluster; LoadBalancer, which exposes the Service using a cloud provider's specific load balancer; and ExternalName, that maps a Service to a DNS name that is verified outside the cluster.

VOLUMES The concept of Volume used by Kubernetes was first introduced by Docker [50]. Volumes are stored in the host filesystem in an area controlled by Docker. This means that, during regular operation, other processes should not be capable of modifying these files. This mechanism allows persisting data beyond the container's lifetime.

Kubernetes borrows this concept from Docker, and expands it with multiple types of volumes. This abstraction allows all the containers within a Pod to use a volume independently from its implementation. Some common implementations are block storage volumes such as AWS Elastic Block Store [8], OpenStack Cinder [126] and GCE persistent disk [68]; or networked filesystems such as GlusterFS [85], nfs [154] or Ceph [35].

2.3 CONTINUOUS DELIVERY

When deploying software systems, they are not seen as being fixed and never-changing [27]. As the application's code is regularly updated by the development team, the system itself will require updates to its configuration and environment.

Each new deployment will introduce changes to the running systems, and any of these changes might become a reliability or security issue. According to a survey made to 800 global businesses, human error is considered to be the top issue negatively impacting reliability by 80% of the firms [47]. Therefore, a system must be set in place in order to prevent such issues from arising.

2.3.1 *Configuration Management*

Configuration can be described as a human-computer interface for modifying system behaviour [27]. As it is an interface that involves human interaction, it is prone to errors. Changing a single configuration option can have dramatic changes on functionality, and it is therefore critical to ensure that these changes are documented, and can be rolled back if they are proven to be harmful.

Configuration management is considered a process by which all artifacts relevant to a project, and the relationships between them, are stored, retrieved, uniquely identified and modified [86]. It is a way to make sure that a system performs as expected to changes made over time [79]. An interesting comparison is made by Bichard [28], when stating that configuration management can be thought of as being like an always up-to-date inventory for the technology assets of a given project, and a single source of truth. This process is also part of ITIL, as Service Asset and Configuration Management [99]. The goal is to maintain information about the configuration items required to deliver an IT service, including their relationships.

This technical approach was first developed by the United States Department of Defense during the 1950s, in order to track changes in the development of complex systems [140]. After various iterations, a guidebook was published in 2001 that named this technical system as configuration management [123].

Software configuration management is also defined as the "process of applying management throughout the software life cycle to ensure the completeness and correctness of software configuration items" [94].

Given its pivotal place in a software system, configuration management should be treated with an engineering approach, in order to have confidence that any change to the system's configuration is safe. Beyer et al. [27] present three criteria that must be met to consider a change to be safe:

- Ability to be deployed gradually, avoiding an all-or-nothing change.
- Ability to roll back the change, if it is found to be dangerous.
- Automatic rollback if the change leads to loss of operator control.

The first criteria fits with one of the principles of the Agile Manifesto, which recommends delivering working software frequently [24]. By adopting configuration management, teams are able to deliver their software safely and keep their agile velocity high. The second one prevents the change from endangering the system that is already working and providing value to its users. The third one is specially important in changes that prevent the system from being accessible at all to its operators. This can happen in situations such as when a firewall rule is updated and it starts to block access to those who are responsible for managing the system. In these cases, the ability to rollback the change automatically is of paramount importance due to the operator's inability to enact further changes to the system.

In a micro-services architecture [6], configuration management is specially useful for managing metadata such as the specification of hardware resource needs for the service, be it CPU, RAM or persistent storage; secrets such as passwords or private keys; and endpoints to external services, such as databases or other domains [30].

According to Team [149], employing these processes of configuration management provides many benefits, such as:

- Reducing the risk of outages and security breaches.
- Quicker restoration of service.
- Enhanced system and process reliability.

- Clear status accounting.
- Cost reduction of IT services.

To properly configure all the elements in the infrastructure, open source and commercial tools are used, and some of the most popular ones are presented in the following Sections.

Chef

Chef is an open-source configuration management tool that automates the configuration, deployment and management of an infrastructure [37]. It operates with a master-agent architecture, meaning that clients pull the necessary configuration data from a central master server.

It consists in three major components: Chef Infra Server, Chef Infra Client and Chef Workstation.

Chef Infra Server is a system that holds the cookbooks, policies and metadata associated with each registered note. Its front-end is written with Erlang [56], which is known for its capability of performing in highly concurrent, fault-tolerant and distributed environments. Every Chef Infra Client interacts with this front-end.

Chef Infra Client is an agent that runs on every node managed by Chef. Each client periodically pulls the Chef Infra Server with the necessary configuration data to update itself. It then performs all the steps required to bring the node into the described state. Ohai [38] is the tool used to collect the system configuration data, which is then used by the Chef Infra Client when applying cookbooks.

Chef Workstation is the system where the operator interacts with the Chef infrastructure. This system has all the tools required to interact with the Chef Infrastructure, namely the chef and knife command line tools, which are used to interact with the Chef Infra Server; testing tools such as ChefSpec and Cookstyle, which allow unit testing resources locally and automatically linting cookbooks; and an instance of the Chef Infra Client, which is what is used by the CLI tools to interact with the Chef Infra Server.

Chef is cloud agnostic and therefore works with any type of nodes, such as:

- Server - an active device connected to a network that can run the Chef Infra Client and have it communicate with the Chef Infra Server;
- Cloud - a cloud-based node that is hosted in popular cloud providers such as AWS [17], OpenStack [127], Google Compute Engine [40] or Microsoft Azure [20];
- Virtual machine - a virtual node that runs as a software implementation, with a similar behavior to the server type;
- Network device - any device (such as a switch or a router) that is being managed by a Chef Infra Client;
- Container - software abstraction that allows a single operating system to host many containers.

Chef interprets the infrastructure with text files named cookbooks written with Ruby [4] as the reference language and extended Domain-Specific-Language (DSL) for certain resources. They define scenarios and can be used to group and organize recipes. Each recipe is a Ruby script that specifies which resources are required and in which order they should be provisioned.

Puppet

Puppet is an open source configuration management tool that aids with the management and automation of servers [134]. It is structured in a master-agent architecture like Chef, but it differs in that the configuration files in Puppet are declarative, while in Chef they are procedural. This means that in Puppet the files describe the desired state of the system, instead of the steps required to achieve that state. They are similar in that they both get their configuration data using a pull mechanism. These files are written in Puppet's DSL called Puppet Code.

Puppet is comprised of five major components: Puppet Server, Puppet agent, Facter, Hiera and PuppetDB.

The Puppet Server is the primary node which manages the configuration information of the agent nodes in the infrastructure. It runs on the Java Virtual Machine (JVM) as a Ruby and Clojure application. This server also needs to configure itself, and therefore it also runs a puppet agent.

The Puppet agent is a daemon whose job is to retrieve the local machine's configuration definitions from the Puppet Server and apply it. This communication between the agent and the server is secured with SSL using a certificate authority that is part of the Puppet Server.

Facter is Puppet's inventory tool, and it is the tool responsible for collecting metrics about the agent node. These metrics are called facts, and they can be classified as core facts, custom facts or external facts. Core facts are the built-in facts that come pre-installed with Facter; custom facts run Ruby code in order to produce a value; and external facts return values from pre-defined static data, or as the result of an executable script or program. These facts are then sent to the Puppet Server in a Puppet code file named a manifest. With this information, the Puppet Server then compiles a catalog, which is a JSON document that describes the desirable state of an agent.

Hiera is a tool used by Puppet for two purposes: to store the configuration data in key-value pairs, and to look up the data needed for a particular module in a node when compiling the catalog. Developing code that serves different purposes but can be configured using site-specific variables in external configuration data files improves the code's re-usability.

PuppetDB is where all the generated data by Puppet, such as facts, catalogs or reports, is stored. It is backed by PostgreSQL [72] and it provides a searchable database of every resource being managed on any node. Puppet Query Language (PQL) is the query language designed to explore the records in the PuppetDB.

Ansible

Ansible is an open source configuration management tool that is also capable of acting as a deployment and orchestration tool [82]. Its architecture is different from Puppet and Chef, as Ansible is an agentless tool that uses the push model, instead of the master-agent pull architectures of the former tools. To do so, Ansible uses SSH, which is a remote management framework that is already pre-installed on most Linux distributions [84].

Its style can be considered a hybrid model, as it allows both procedural and declarative configuration [83]. It can be procedural in that Ansible executes each task in order, one at a time and against all the desired hosts. It can also be seen as declarative, as some Ansible modules check if the desired final state has already been

achieved, and consequently exit without executing any tasks. These modules are called idempotent because running them once or multiple times results in the same outcome.

To configure the machines, Ansible uses YAML files called playbooks [81]. Each playbook comprises one or more plays. Each play is then made of one or multiple tasks. A task consists on a call to an Ansible module that fulfills a specific job. A set of tasks can be organized into reusable units of code called roles. These bring a similar benefit to the use of Hiera in a Puppet system, as they allow applying common configurations in different scenarios by simply changing site-specific variables. These tasks and roles are executed on hosts which form the inventory. The inventory can be further grouped to allow finer-grained execution of plays.

2.3.2 *Version Control*

In order to apply the principles of configuration management, a version control system is often used [119].

Version control systems are a mechanism for keeping multiple versions of files [86]. They also allow teams of engineers to collaborate on such files.

They are a very powerful tool because by keeping everything in version control, anyone is able to reproduce a particular state of the software system at any given time [98]. It also adds accountability for every change, by keeping a record of when the change was made, who made it, what it consisted of and for what reason was it made.

One of the most popular tools for managing versions is Git [65], which accounts for 70% of all the search interest among 5 different version control systems [42]. This choice is very popular due to several characteristics.

Being a distributed version control system, each developer gets a full working copy of the repository. Consequently, if there is a failure in any of the servers which hold the data, the information can still be retrieved using any of the client repositories [36].

Another benefit of having the full history of the repository locally is that it allows offline inspection of previous versions of a file and comparisons between commits [12]. This avoids the cost associated with performing such operations over the internet.

2.4 CONTINUOUS INTEGRATION

Continuous Integration (CI) is considered an automatic process which encompasses steps such as running unit and acceptance tests, compiling code, verifying code coverage and building deployment packages [59]. In Extreme Programming (XP), continuous integration implies that teams build multiple times per day, committing code into the repository whenever possible [95].

According to Ståhl and Bosch [148], it improves release frequency and predictability, improves communication and increases developer productivity. CI was considered by Wells [153] as a "pay me now or pay me more later" activity, because the burden of integrating small changes throughout the project is significantly smaller than the burden of integrating large changes near the end of the project.

Several tools can be used to enable this practice, such as Bamboo [14], Jenkins [96], Travis CI [150], CircleCI [39] or GitLabCI [66]. They enable fast feedback when a code change causes a test to fail or when it cannot produce a resulting build [119].

2.4.1 *Infrastructure as Code*

Infrastructure as Code (IaC) is an approach that handles infrastructure automation based on software development practices [119].

It emerged as a way to solve the problem of environment drift in the release pipeline [73]. This drift helped create snowflake servers, which represent a configuration that cannot be replicated automatically and thus burden the task of maintenance.

By treating infrastructure as code, a team is able to commit changes regularly and have those changes be tested and deployed, when used in conjunction with a continuous delivery and integration approach. This also decouples the infrastructure from the operations team, as everything is self-documented in the source files and there's full traceability of the changes each file suffered and thus improving accountability.

2.4.2 *Provisioning*

Provisioning is the process of setting up the infrastructure, and it can involve different types of provisioning [80].

Provisioning tools can be used to set up any of these types of provisioning, and most infrastructure vendors offer their own platform-specific tools. Google developed Cloud Deployment Manager [67], Amazon has AWS CloudFormation [7] and Microsoft created Azure Resource Manager [115]. There is also a popular open source tool called Terraform [76].

Server provisioning

Provisioning a server involves all the operations required to create a new machine and set it to a working state. This can include setting up the actual hardware in a data center, but is usually more concerned with installing and configuring its operating system and the connections to storage or network providers.

Network provisioning

Network provisioning includes setting up connectivity among users, containers, servers and other devices. It involves the creation and management of networking routes, load balancing pools or firewall rules [119].

Storage provisioning

Provisioning storage is the process of providing disk space to be used by applications and services running on the servers.

This can include setting up a Storage Area Network (SAN), which provides block-level network access to storage [23].

User provisioning

User provisioning involves creating, managing and maintaining a user's rights and privileges within the infrastructure. It is a process that can include both the operations team and human resources.

This can include setting up a Role-Based Access Control (RBAC), which is comprised of permissions, roles, groups and users [151].

2.4.3 *Terraform*

Terraform was first released by HashiCorp in 2014, and it is written using the Go programming language [76].

It is cloud-agnostic and has compatibility with the largest infrastructure providers: Amazon [17], Google Cloud [40], Azure [20], Alibaba Cloud [70], Oracle [128], and Openstack [127]. It also leverages capabilities to interact with cloud-agnostic tools such as Kubernetes.

Terraform allows describing any part of the infrastructure with configuration files using the domain-specific language Hashicorp Configuration Language (HCL). These files can be version controlled, creating a history of changes to the infrastructure.

With the configuration files, Terraform generates an execution plan that describes the steps it will execute to reach the described state.

After reviewing the plan, the final step in a Terraform workflow is applying the proposed changes to the infrastructure, provisioning the required resources.

2.5 APPLICATION MONITORING

Monitoring is described as the activity of collecting, processing, aggregating and displaying real-time quantitative data about a system [26]. Cotton [45] breaks monitoring down in five different types: infrastructure monitoring, log analysis, distributed tracing, application performance monitoring and real user monitoring, but emphasizes that the first one remains at the center of any monitoring strategy.

2.5.1 *Desirable requirements*

An ideal monitoring strategy has four desirable requirements [27]. First of all, it shall be fast, because how long it takes for the system to notify the operator when something goes wrong can have a substantial impact on the length and severity of the failure. Also, slow data can lead the operator to act on incorrect data. Second of all, it shall allow calculations to be performed over the metrics that have been collected. Supporting statistical functions can be particularly useful because trivial operations may mask bad behavior. Thirdly, it shall allow the

data that is collected to be displayed in useful arrangements to the people who consume such information. These can include graphs for time-series data, and tables or charts for more structured data. Last of all, it shall alert the people responsible for the system proportionally to the size of the event. These alerts can include e-mails, displaying an indicative message on some dashboard, or even a direct message to the person responsible for the failing part of the system.

2.5.2 *Sources of data*

Monitoring data can come from four major sources [86]. Via the hardware, by monitoring voltages, temperatures, fan speeds or peripheral health; via the operating system, by monitoring memory usage, swap usage, disk space, I/0 bandwidth or CPU usage; via middleware, by monitoring the usage of database connection pools, or information on the number of connections or response time; and finally via the application itself, by connecting to hooks that enable visibility to metrics that are relevant to both the operations teams and the stakeholders.

This collected data can be broadly classified in two categories: logs or metrics. Metrics are numerical measurements that represent attributes and events, and are typically collected via data points at regular time intervals. These can be attributes that range from latency in a service or database to the number of open file descriptors [125]. Logs are an append-only records of events.

Metrics are used at Google to drive alerts and dashboards, given their real-time nature, whereas logs are commonly preferred to find the root cause of an issue as they tend to produce more accurate data than metrics [27].

2.5.3 *Types of monitoring*

Monitoring an application can be done using two types of strategies. There isn't a clear consensus on the boundaries of each of these types, but the terminology is common, with Dilhasha [49] sharing the same view as Beyer et al. [26] and Rogers [138] with a different approach to the former.

Black-box monitoring

Black-box monitoring can be classified as the testing of externally visible behavior, just as a user would see it [26]. This type of monitoring indicates the general availability of the system and is symptom-oriented, as it represents active problems, in contrast to predicting future problems [49].

According to Rogers [138], black-box monitoring is referred to as monitoring with a focus on areas such as disk space, CPU usage, load averages or memory usage. It seems to include system specific metrics, and not application-related metrics. It does, however, in contrast to the definition presented before, require that the system must be known with more detail, as getting system specific metrics requires instrumented access to those systems.

White-box monitoring

White-box monitoring is a type of monitoring based on metrics exposed by the internals of the system, which includes logs, interfaces or HTTP handlers that emit internal statistics [26]. This type of monitoring depends on the possibility to inspect the internals of the system, and therefore requires instrumentation. In contrast to black-box monitoring, it allows the detection of imminent problems, or unexpected behavior [138].

Given the need for instrumenting the source code, white-box monitoring requires developers to take their share of responsibility in monitoring their applications, whereas black-box monitoring is mostly the responsibility of systems administrators or DevOps engineers.

By using a mix of white-box and black-box monitoring, one can both detect a problem and find its cause, which enables the development of a solution. By recognizing the symptoms using black-box monitoring, one can inspect the data provided by the white-box monitoring strategy to help diagnose and fix the issue.

2.5.4 *Goals*

The monitoring system that is setup should be able to answer two main questions: what is broken and why is it broken. Addressing the former reveals the symptom, while the latter indicates a possible intermediate cause [26].

Another important aspect of the monitoring system is making sure that the system is meeting the Service Level Agreement (SLA) set between the stakeholders and the product owner. This contract includes the consequences of meeting or missing the Service Level Objective (SLO) they contain. SLOs are values, or range of values, for a service level and are measured through a Service Level Indicator (SLI). These are a quantitative measure of an aspect of the level of service that is provided.

Beyer et al. [26] further defines four metrics that are considered common goals for every application: latency, traffic, errors and saturation.

Latency is related to the time taken to service a request, measured from the moment it arrived to the system to the moment a response was sent; traffic is a measure of the amount of demand being put on the system, and can be measured with metrics such as the number of requests per second; errors is the rate of requests that fail, and can be measured by monitoring the status code of an HTTP response; saturation is a measure of the system utilization, and can be measured by the amount of resources being used in a system, such as memory or CPU utilization.

2.5.5 *Nagios Core*

Nagios Core is an open source monitoring tool that was first launched in 1999 [55]. Nagios is a recursive acronym that stands for "Nagios Ain't Gonna Insist on Sainthood", which is a reference of the the previous version of the software developed by Ethan Galstad named NetSaint [64]. It is aimed at monitoring the infrastructure, instead of the application level.

Nagios Core provides several features to monitor an infrastructure [54]: a web interface to view network status, log files and event history; monitoring of network services using HTTP, SMTP, POP3, NNTP or ping; monitoring of system resources such as CPU load, disk usage, memory usage or running processes; notifications for when a problem is detected in a service or host.

In order to configure a Nagios monitoring solution, multiple objects shall be defined.

A host is used to define a physical or virtual server that resides in the infrastructure. These hosts can then be grouped into host groups, in order to simplify the configuration of similar hosts.

Hosts can have three different states: Up, Down and Unreachable. When Nagios checks the status of the hosts, it can detect changes in state and trigger pre-defined event handlers and send out notifications.

A service is used to define a metric that can be extracted from something that is running on a host. These services can also be grouped into service groups, to further simplify managing similar services. A service is pinged using a provided command, and its status is inferred from the response status. These checks occur at regular intervals according to the service definitions, following a pull model.

Services can have four different states: Ok (o), Warning (w), Unknown (u) and Critical (c). These can be used in order to apply different policies according to a state change in a service.

2.5.6 *Prometheus*

Prometheus is an open source systems monitoring tool initially developed at SoundCloud in 2012 [132]. It collects data via a pull model over HTTP. This provides some advantages:

- Prometheus can run on any machine when developing changes;
- Easier detection if a target is down;
- A target can be inspected manually with a web browser.

It is designed as a system to collect and process metrics, and not as an event logging system.

Prometheus forms an ecosystem of multiple components: the Prometheus server, client libraries, a push gateway, service exporters and an alert manager.

Prometheus server

The Prometheus server scrapes metrics from instrumented targets and stores the samples locally. These are time series data samples, and they can be manipulated with aggregations to produce new time series. This data can then be visualized using tools such as Grafana [105].

Client libraries

Client libraries are used to instrument application code. There are officially supported libraries for Go [1], Java [2], Scala [5], Python [3] and Ruby [4]. These libraries implement the Prometheus metric types: counters, gauges, histograms and summaries.

Metric types

Counters are cumulative metrics that represent single monotonically increasing counters. They can be used to represent metrics such as the number of tasks completed, number of errors or number of served requests.

Gauges are metrics that represent a numerical value that can arbitrarily go up and down. They can represent metrics such as memory usage, or number of concurrent requests.

Histograms are sampled observations that are counted in buckets. These buckets can be configured and they track both the number of observations and the sum of the observed values. Each bucket has an upper bound.

Summaries are similar to histograms in that they sample observations. However, they also provide configurable quantiles over a sliding time window.

Push gateway

The push gateway is a tool to allow ephemeral jobs to expose metrics to Prometheus [133]. Given the ephemeral nature of these jobs, they use a push model to send metrics to this gateway. This avoids the risk of them not existing long enough to be pulled by Prometheus. The gateway then exposes these metrics to Prometheus.

Service exporters

Service exporters are tools and libraries that are used to instrument systems with Prometheus metrics. These systems can range from databases, to hardware, issue trackers, messaging systems, storage systems, web servers and service APIs.

Alertmanager

Alertmanager is the component responsible for handling alerts sent by the Prometheus server. These alerts can be categorized into groups to reduce them to a single notification. This component is also capable of suppressing notifications based on certain criteria, such as other alerts already being fired.

After receiving alerts from the Prometheus server, Alertmanager is capable of processing and forwarding them to external systems such as e-mail or generic webhooks that can connect to popular messaging apps like Slack [143] or Mattermost [111].

Grafana

Grafana is an open source analytics and monitoring solution recommended to visualize data stored in Prometheus [105].

With Grafana, it is possible to visualize all the metrics stored in Prometheus with informative dashboards that can be defined with JSON files [106]. This approach enables these dashboards to be checked into version control systems and therefore take advantage of the principles of Infrastructure as Code.

2.5.7 *Elastic Stack*

The Elastic Stack is composed of four different tools: Elasticsearch, Kibana, Beats and Logstash [52]. These tools work together to provide analytics and visualization to an infrastructure or application.

Elasticsearch

Elasticsearch was the first tool developed by Elastic and launched in 2010 [21]. It is a distributed search and analytics engine and is considered the heart of the Elastic Stack [53].

Elasticsearch is efficient in storing and indexing any type of data, be it structured or unstructured. It is used within the Stack as the storage component of the data collected and aggregated by Logstash and Beats.

Kibana

Kibana is an open source analytics and visualization tool developed by Elastic [53]. It was designed to use Elasticsearch as a data source, and it offers the possibility to manage the entire Elastic Stack.

Kibana displays data in several ways: using Discover, Dashboards, Canvas or Maps.

Discover is a tool used to freely explore the data available in Elasticsearch. It is used to meet one or two goals: to get a general overview of what is happening with something that is being stored in Elasticsearch or to find an answer to a specific question.

Dashboards are a collection of panels used to analyze data. These panels can provide information using a diverse set of types: areas, bars, lines, pies, donuts, tree maps, goals, gauges, metrics, tag clouds, data tables, heat maps, stacked bars, or stacked areas.

Canvas are a data visualization and presentation tool to mix live data from Elasticsearch with colors, images, or text. This allows exporting data to a more visually compelling report.

Maps are a tool to create maps that interpret geographical data at scale. The data structure that supports this type of visualization is GeoJSON [32]. The maps produced by this tool can then be embedded in dashboards.

Beats

Beats are an open source data export tool that acts as agents on the desired servers in order to feed an Elasticsearch cluster [53]. Each of these agents monitors a specific aspect of a system or application.

There are several types of Beats: Auditbeat, Filebeat, Functionbeat, Heartbeat, Journalbeat, Metricbeat, Packetbeat and Winlogbeat.

Auditbeat is a shipper that audits the activity of users and processes in systems [53]. It can also detect changes to files that could be classified as security policy violations.

Filebeat is a shipper that monitors the log files on specified locations [53]. This data can then be sent directly to Elasticsearch, Logstash, Kafka [9] or Redis [109].

Functionbeat is a shipper that is deployed in serverless environments in common cloud providers such as Amazon or Google [53]. In Google it can be deployed as a Google Cloud Function [40], and in AWS [17] as an AWS Lambda service. The events processed by this agent are then sent to Elasticsearch.

Heartbeat is an agent that periodically checks the status of a service [53]. It differs from Metricbeat in that it can provide information about a service's reachability.

Journalbeat is a shipper that forwards log data from systemd journals [53]. These log events are then forwarded to either Elasticsearch or Logstash.

Metricbeat is a shipper that collects metrics from both the operating system on which it is running and services running on that server [53]. There is a large variety of modules that provide metrics from popular services such as MongoDB [118], Nginx [58], PostgreSQL [72], Redis [109], Zookeeper [10] and even Prometheus.

Packetbeat is a network packet analyzer that provides application monitoring and performance analytics [53]. It captures network traffic between application server and correlates requests with responses. It supports protocols such as ICMP, DHCP, DNS, HTTP, AMQP, Redis and NFS among others. The captured correlated transactions are then inserted directly into Elasticsearch or a queue created with Redis and Logstash.

Winlogbeat is an agent that ships Windows event logs using Windows APIs [53]. These events can range from application events, hardware events, security events or system events. They are also sent directly to Elasticsearch or through Logstash.

Logstash

Logstash is an open source data collection engine [53]. The goal of Logstash is to dynamically treat data from various sources and normalize it in order to be further analyzed. A usual installation of Logstash receives data from one or more Beats, transforms it with optional filters, and then sends it to an Elasticsearch cluster. This data can then be visualized with a tool such as Kibana.

Logstash works as an event processing pipeline. Each pipeline worker handles batches of events from a central queue, applying configured filters and running those events through outputs such as Elasticsearch.

3

CASE STUDY

In the last Chapter, the literature that is relevant to the goals of this dissertation was presented, as well as some agile methodologies that should guide the software development, deployment and configuration process. As a result, there is now a body of knowledge, tools and frameworks that can be used to target a concrete case study.

In this Chapter, the focus will be on examining Mobile ID, the case study that motivated the development of this dissertation's work. The assessment of the status of the platform before the dissertation's work is crucial to understand the design choices made on Chapter 4 and the concrete implementation on Chapter 5.

This Chapter starts by presenting the technical specifications that guided the Mobile ID project in Section 3.1.

Section 3.2 describes the backend design of the case study, detailing the purpose of each component as well as its dependencies.

Section 3.3 details the infrastructure on which the backend was deployed, describing the distribution of the components. The Section ends with a discussion of the limitations that the current infrastructure presents.

Finally, Section 3.4 outlines the deployment process for each of the components of the backend. For each of the processes, there is a discussion on its drawbacks and in which contexts they can become problematic.

3.1 MOBILE IDENTIFICATION

The ISO/IEC FDIS 18013-5 is an International Standard that specifies the interfaces in an implementation of a driving licence associated with a mobile device [87]. These include the interface between a Mobile Driving Licence (mDL) and a mobile driving licence reader, and the interface between a mobile driving licence reader and the issuing authority infrastructure.

Besides the interfaces, this document also specifies a *mdoc* as a "document or application that resides on a mobile device or requires a mobile device as part of the process to gain access to the document or application". This concept can be applied to different areas of mobile identification beyond a mobile driving licence. However, this standard does not propose a similarly generic structure for the issuing authority infrastructure.

An effort to generalize both the interfaces and the data structures is underway with the ISO/IEC CD 23220 document series, which specifies the building blocks for identity management via mobile devices. This series includes six different parts: Part 1 is related to generic system architectures of mobile eID systems [93]; Part 2 to data objects and encoding rules for generic eID systems [88]; Part 3 to protocols and services for the issuing

phase [89]; Part 4 to protocols and services for operational phase [90]; Part 5 to trust models and confidence level assessment [91]; and Part 6 to mechanisms for use of certification on trustworthiness of secure area [92].

3.2 BUSINESS ARCHITECTURE

The MobileID team at INESC TEC has developed a proof of concept implementation of the ISO 18013-5 standard, which includes both the issuing authority infrastructure and two mobile applications: one for a mobile driving licence holder, and another for verifying and reading mobile driving licences. The analysis and subsequent design proposal is primarily aimed at the issuing authority infrastructure components.

The issuing authority infrastructure has been implemented as a monolithic Django [61] project, and it has been partitioned into several apps, as depicted in Figure 3. For each app, a short summary is presented as well as its external dependencies, which will prove useful to verify if the proposed solution is compatible with the current system implementation.

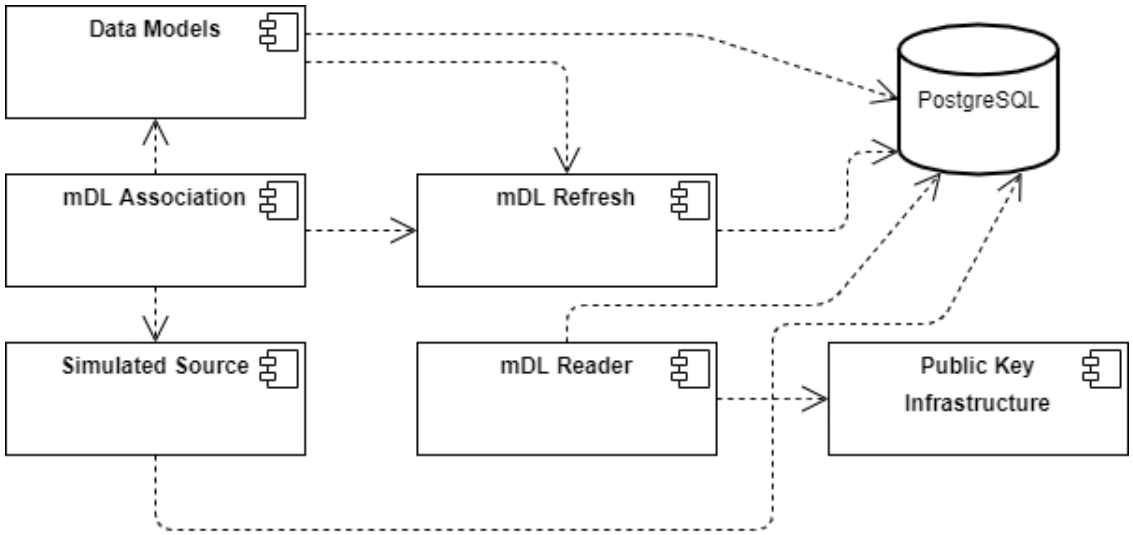

Figure 3: Current business architecture

The Data Models app contains the data models of a simulated database of driving licences. This includes driving privileges, inhibitions, and other biographic information. It connects with the mDL Refresh app through a REST API whenever a change is performed to an attribute of any driving licence holder. It also connects to a PostgreSQL [72] database to persist the data.

The mDL Association app holds most of the business logic of the system, such as associating driving licences with a mobile device, refreshing the status of a mDL, and online retrieval of attributes. It connects with the mDL Refresh app through a REST API whenever a new mobile driving licence is associated with a mobile app. It also depends on the Data Models app to retrieve information about a holder's driving licence.

The mDL Reader app is responsible for managing authorized mDL readers and the attributes they are authorized to retrieve from mDLs. It connects to a PostgreSQL database to persist the data, and to a Vault PKI to manage and issue certificates [78].

The mDL Refresh app allows the verification of the status of any mDL. It connects to a PostgreSQL database to persist the data and does not depend on any other app.

The Simulated Source app emulates the behavior of the Mobile Digital Key, a platform provided by the Portuguese State that links a cellphone number to a document number, allowing the retrieval of attributes of any citizen pending his authorization [130]. This app also connects to a PostgreSQL database to persist the data.

3.3 INFRASTRUCTURE ARCHITECTURE

The current infrastructure that supports the business architecture described in Section 3.2 is depicted in Figure 4. It consists of three virtual machines that are deployed on physical machines on-premises.

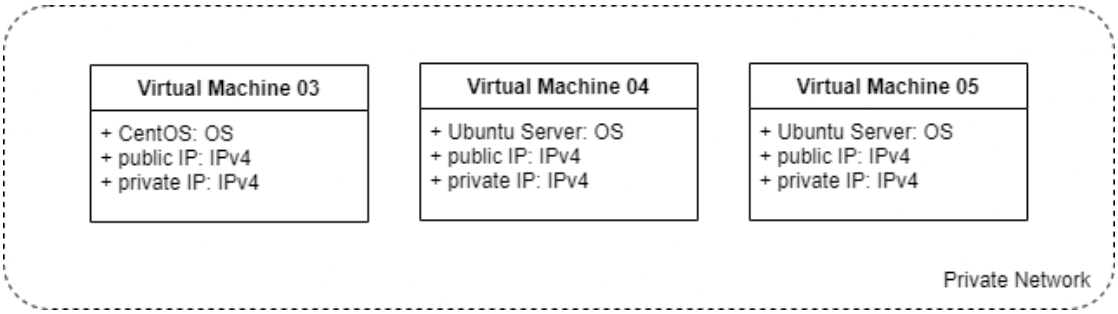

Figure 4: Current infrastructure

After deploying the machines, they can be accessed and configured by SSH while connected to a Virtual Private Network (VPN) through port 22. The only other available ports to the public Internet are ports 80 and 443, which are the default ports for HTTP and HTTPS traffic.

The deployment of the components on the infrastructure is depicted on Figure 5. There are two Django components, which represent different releases of the proof of concept. They are represented with **v1** and **v2**, meaning version 1 and version 2 respectively. The decision as to where to deploy this component was determined by the development team by the available machines at the time of deployment and the requirement to clearly separate the concerns of the two proofs of concept. Each of those components has also access to a local PostgreSQL relational database management system, as well as a web interface developed with Flask [129].

On Machine 05 there is an instance of Vault deployed, in order to provide Public Key Infrastructure (PKI) capabilities to the app. It also includes a minimal setup of Prometheus, which collects metrics from the Django app as well as from other external services. These external services include an SMS communication API, whose balance is actively monitored in order to issue notifications to a chat system through a web hook. These collected metrics are then displayed with Grafana dashboards.

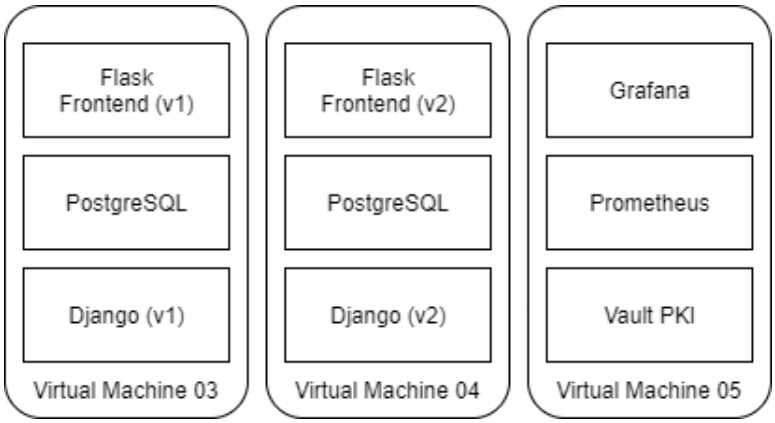

Figure 5: Current distribution of components on the infrastructure

This current infrastructure, as well as the distribution of the components present some limitations. The failure of any of the machines can be catastrophic, as there is no backup strategy automated by the development team. If a machine fails, data could be lost and its components will be unavailable until a new one is deployed and configured with new instances of the components. The recovery of data is reliant on the local backups made by the team or the nature of the failure of the underlying infrastructure, which is outside the team's scope of action.

3.4 DEPLOYMENT PROCESS

The process for deploying the virtual machines and the private network is manual, as it involves sending a ticket to an IT support team that will create a machine fitting the desired requirements. The private network is already deployed, and only requires registering addresses for the newly added virtual machines. This process can be slow and prone to issues, as it can take up to forty eight hours and relies on human resources for its correct execution.

The deployment of the Django, PostgreSQL and Flask components is also manual. Each of these components has its own Git repository, where all the project files are tracked. Sensitive information such as database credentials and secret keys are not stored in a version control system. Instead, they are saved on the machine in which that information is required and in local backups that are the responsibility of the development team. The process of applying an update varies according to the component.

For the Django project, the process is as depicted on Figure 6. This process already has a continuous integration system, which automatically verifies unit tests that are part of the source code and analyses the code for any syntax errors. However, every time a new change needs to be deployed, a team member with access to the desired machine must pull the updated files on that machine and restart the Django system service. If the change is more significant, it may require manual intervention on sensitive information, such as when a connection to a new external service is required and its credentials need to be set up on the corresponding deployment. After pulling the code and complete necessary manual changes, the system daemon is restarted and the new version is ready to be used.

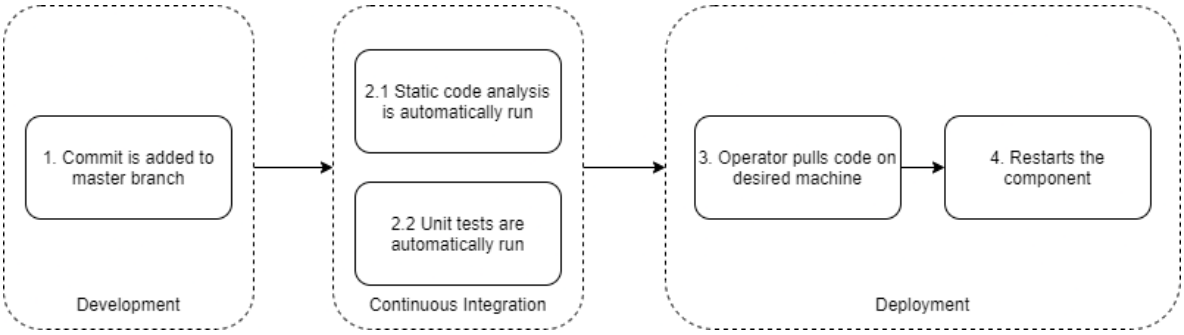

Figure 6: Current software lifecycle for the Django project

In case the new deployment causes issues, the rollback to a previous version is very difficult. This happens because after deploying a new change, the previous version is only stored in the version control system. When a new deployment requires a significant change (such as when it applies database migrations), rolling it back can require a significant amount of manual work, and it might even be more efficient to apply a new patch on top of the upgrade, instead of rolling back to a known working state.

As for the monitoring components on Machine 05, the deployment is already semi-automated and its process is depicted on Figure 7. The Grafana and Prometheus configurations are stored on a Git repository, following the IaC approach as laid out on Section 2.4.1. After committing new changes to the repository, a GitLab Runner automatically builds a new Docker image based on the public images provided by both Grafana and Prometheus. This image is then uploaded to the GitLab Container Registry. This process is already automated through a GitLab CI pipeline, whereas the deployment step is still manual. The image is pulled by a member of the development team with access to the machine, and the container is restarted with the latest built version. Because the resulting images are stored in the Container Registry, the process of rolling back unwanted or problematic changes is easier than with the Django project, as it only requires changing the image used for the component.

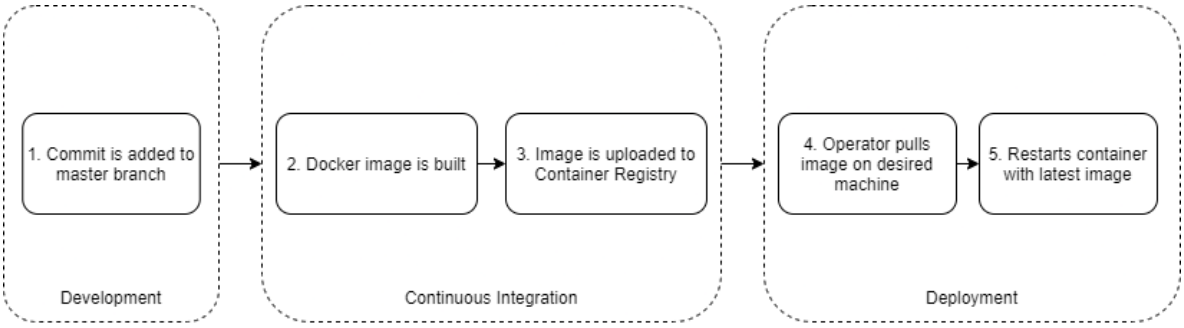

Figure 7: Current software lifecycle for the monitoring components

The process of deploying the Vault PKI is similar to the monitoring components process. Every time the configuration is changed, those files are added to a Git repository. The result of this commit is not processed by

any continuous integration pipeline. Instead, the code is pulled by an operator on the desired machine and the resulting state is managed through a Terraform Configuration whose state is stored locally.

4

PROPOSED DESIGN

In the previous Chapter, the case study that motivated this dissertation's work was described and analyzed. There was a discussion on the shortcomings of the design across its infrastructure, and its deployment process.

As a result of that discussion, and with the analysis of the agile methodologies researched in Chapter 2, this Chapter proposes an architecture for three components in the case study: the general infrastructure, the monitoring infrastructure, and the deployment process. These architectures were designed without a specific platform or tool in order to provide flexibility during the implementation phase, where there will be a selection of the most appropriate tool for each of the components.

This Chapter starts with the technical requirements for the implementation of the three components, specified according to its area of focus in Section 4.1.

Section 4.2 details the architectural overview of the physical infrastructure of the system. This platform-independent architecture was designed by abstracting the underlying cloud service deployment model, and meeting the requirements of Section 4.1.

Section 4.3 describes the architectural overview of the monitoring infrastructure of the system. This design creates a logical distinction between the components that are part of the monitoring system, but already adopts the best practices presented in Chapter 2.

Finally, Section 4.4 proposes the deployment process for services deployed in the infrastructure.

4.1 REQUIREMENTS

In order to design and implement a proof of concept that improves the current system, a list of requirements was collected. It was based on the limitations and challenges identified during the analysis in Chapter 3, and the best practices and methodologies as researched in Chapter 2. The resulting list of requirements focuses on the infrastructure and monitoring layers of the system, as well as the deployment process.

4.1.1 *Infrastructure requirements*

1. The infrastructure must be described by text files that can be version controlled.
2. The infrastructure must allow the failure of at least one node at any moment.

3. The infrastructure must be accessible without the use of a Virtual Private Network (VPN).
4. The infrastructure must be created and destroyed with a single executable action.
5. The infrastructure must be capable of balancing its services automatically according to its nodes' load.
6. The number of nodes in the infrastructure must be configurable.
7. The infrastructure must allow network connection between all the nodes and from the Internet.
8. The tools used to deploy the infrastructure must be open source.

4.1.2 *Monitoring requirements*

1. The CPU and memory usage of the physical nodes of the infrastructure must be monitored.
2. The monitoring tools must automatically collect deployed services' metrics.
3. The monitoring tools must allow the configuration of rules-based alerts.
4. The monitoring tools must permanently store the collected metrics.
5. The monitoring tools must be configurable by text files that can be version controlled.
6. The monitoring tools must be capable of importing dashboards specified in JSON or YAML formats.
7. The tools used to monitor the infrastructure and its services must be open source.

4.1.3 *Deployment requirements*

1. The user must be able to logically isolate deployed services on the infrastructure.
2. The user must be able to deploy services on one or more nodes.
3. The user must be able to connect to a Public Key Infrastructure (PKI).
4. The user must be able to automate the deployment of services specified in a remote VCS.
5. Only authorized users are allowed to deploy services on the infrastructure.

4.2 INFRASTRUCTURE ARCHITECTURE

The infrastructure is the foundational level on which the services and tools will be deployed on. Therefore, it must be designed taking into consideration the requirements specified in Section 4.1, and making sure that it

is capable of providing the necessary computing resources to the software layers that are deployed on it. This activity is part of the Infrastructure and platform management, a technical management practice of ITIL 4.

The proposed solution is platform independent, as it relies on abstracting the underlying cloud service deployment model: whether it is a private, public, community or hybrid cloud [113]. The platform independent architecture is depicted on Figure 8.

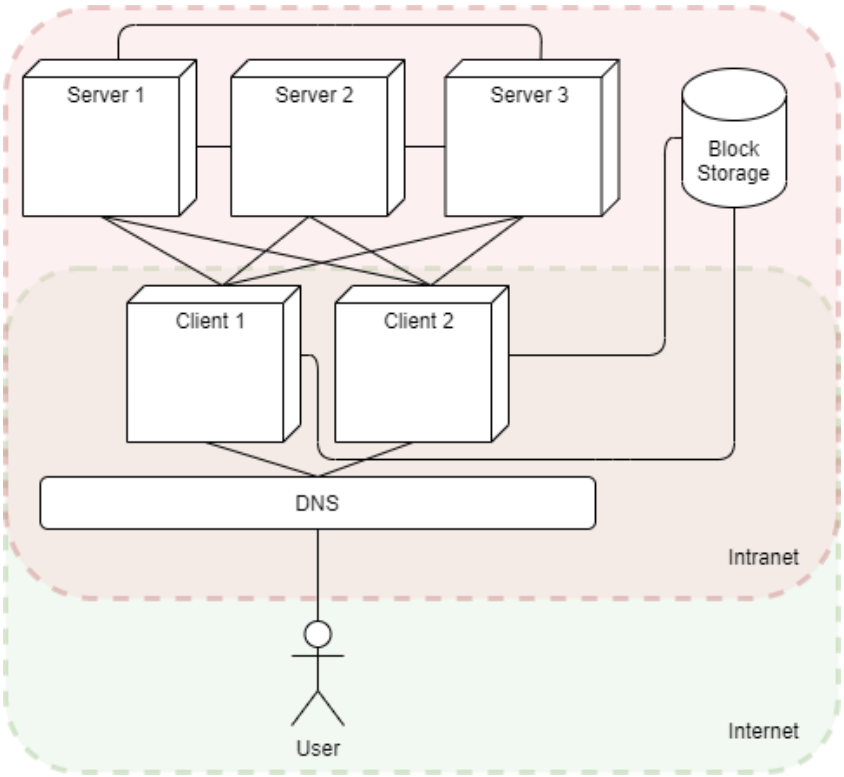

Figure 8: Infrastructure architecture

This proposed architecture is based on the minimum common requirements of the container orchestration tools researched on Section 2.2.3. Nomad and Consul require an odd number of server nodes and an adequate number of client nodes. As they use Raft for replicating state throughout the servers, a quorum of servers is required for its correct operation [124]. By having three nodes as servers, the cluster can withstand the failure of one of the servers, as consensus can still be reached with two active nodes. Having a higher number of server nodes could improve the fault tolerance of the infrastructure. However, not only would it require more processing time to replicate state along the increased number of servers, it would also incur in higher billing costs.

The number of clients is set to two, providing fault tolerance to the services running on the infrastructure in case one of the client nodes fails. The choice of having two client nodes is based on the current load of the infrastructure, as it only has nine different applications which do not have a consistent high load. If that load increases, more nodes should be readily made available in order to load balance the traffic throughout the client

nodes. The proposed design does not take into consideration the possibility of running services on the server nodes, even though it is a feature of tools like Docker Swarm and Kubernetes.

In order to allow mobility of services between the client nodes, they both have access to a shared block storage volume. Attaching the volume dynamically when a service is booted on any of the machines provides greater flexibility and fault tolerance to the system. If further storage requirements are set in the future, the size and speed of this volume can be adapted accordingly.

All of the nodes, including the block storage volume, are deployed on a Virtual Private Network (VPN). This limits outside traffic and helps prevent unauthorized access. The only machines that are available to the general Internet are the client nodes, as the users need to be able to reach them in order to use the deployed services. In order to provide Open Systems Interconnection (OSI) Layer 4 load balancing across the client nodes, the Domain Name System (DNS) is used. There are other possible options that could be adopted in the future. One of them is to use a cloud provider's own load balancing solutions: Google Cloud's Load Balancing [41], Digital Ocean's Load Balancer [48], Azure Load Balancer [116] or AWS's Elastic Load Balancing [18]. These products automatically distribute traffic across multiple targets, are especially tailored to their own products and they also provide load balancing capabilities at Layer 7 in the OSI model. However, they also aggravate the operating cost of the infrastructure and create a dependency on a specific cloud provider's implementation. Another option would be to include another set of nodes configured to operate a load balancing application such as NGINX [121], HAProxy [74] or Traefik [107]. These nodes would be exposed to the public Internet and they would allow removing the direct connection of the client nodes to the Internet. However, it would increase the operating cost of the infrastructure due to the larger number of nodes and subsequent increase in maintenance-related activities. Given the current traffic expectations, the adopted solution fits the requirements and is flexible enough to adopt other solutions later on if the circumstances demand it.

4.3 MONITORING ARCHITECTURE

In order to meet the requirements specified in Section 4.1, a set of monitoring components need to be configured and deployed on top of the infrastructure architecture proposed in Section 4.2. However, this design could be applied in a different infrastructure, as it only requires computational instances capable of running its components. The goal of this design is to improve and formalize the monitoring and event management of the case study. This activity is a service management practice of ITIL 4 and is depicted on Figure 9.

The proposed architecture is comprised of three sets of components: Services, Collection, and Visualization.

Services

The Services set includes two types of components: the services themselves, and a service discovery component. The former are the software applications that will be deployed on the infrastructure, while the latter is a software component that automatically detects software applications that are deployed on the infrastructure's network. As the number of instances of each service and their location can dynamically change according to a set of restrictions, a service discovery component is required in order to reach an instance of the desired service.

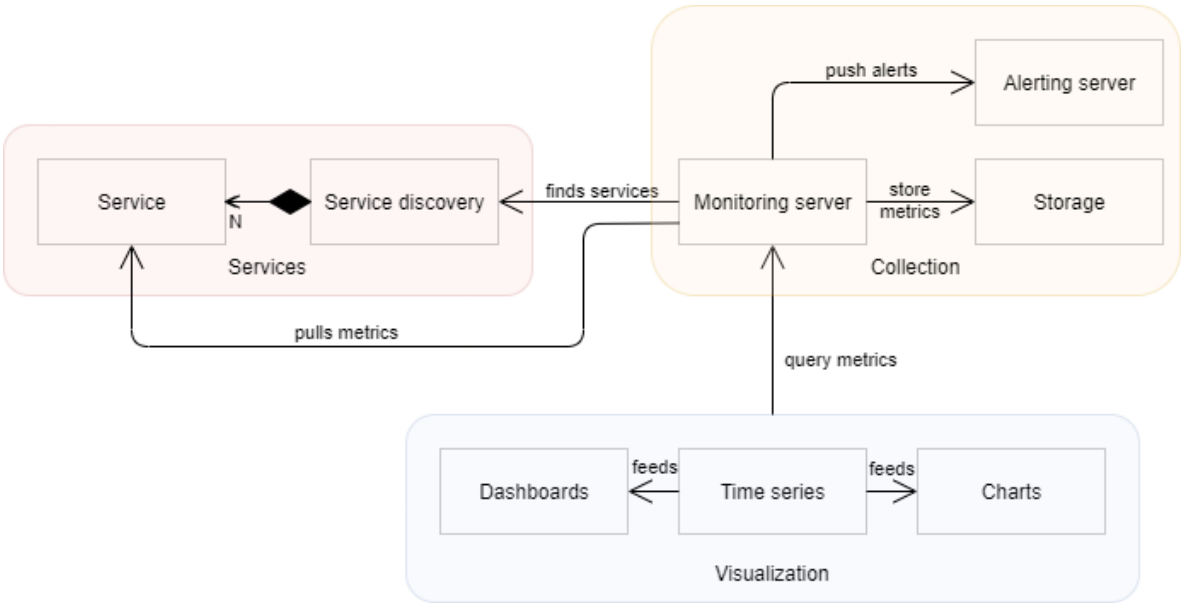

Figure 9: Monitoring architecture

Collection

The Collection set includes three types of components: the monitoring server itself, an alerting server and a storage medium.

The monitoring server is a software component that can fulfill four distinct use cases: to find deployed services, to pull metrics from those services, to push alerts according to a set of rules, and to store the collected metrics. The first use case is relevant in order to decouple the monitoring server's configuration from the deployed services. By having a generic interface that finds target services, the server can dynamically add and remove monitoring targets without manual configuration. The second use case is the primary use case of any monitoring server that uses the pull architecture for monitoring targets. Having a list of services to pull metrics from, the server can process the results and act upon them for the third use case. It requires a set of rules which are defined by the operator. If a metric (or set of metrics) meets the criteria determined by the rules, the server will push an alert to the alerting server component. The fourth use case is important to analyze the behavior of the monitored services throughout time, in order to detect patterns and prevent future unavailability. It is also useful to produce regular reports based on retrieved data, and to assess whether the system is complying with the Service Level Agreement (SLA).

The alerting server acts as a complement to the monitoring server. It decouples the alerting business logic from the alerting actions. This component is responsible for routing the alerts that are pushed by the monitoring server to a range of channels such as email, chat platforms, phone calls or messaging. This server also provides notification rate limiting, silencing and alert dependencies on top of the pushed alerts by the monitoring server.

The storage medium is a time-series database that stores the collected metrics by the monitoring server. This database shall be adapted to the format of metrics that suits an efficient storage and retrieval of data by the monitoring server.

Visualization

Finally, the Visualization set is comprised of three different components: the time-series data, dashboards and charts. The time-series data is the result of querying the metrics collected by the monitoring server. These are then fed as the basis of dashboards and charts that provide useful information about the status of the system, and its performance throughout periods of time. These components must also be flexible enough in order to allow graphing information across different aggregations of metrics in real time.

4.4 DEPLOYMENT PROCESS

The goal of both the architectures proposed in the previous Sections is to promote a deployment process that fulfills the criteria of having readily available a set of metrics and useful information about any new service deployment. With the capabilities provided by this infrastructure, the deployment process for each software service is depicted in Figure 10.

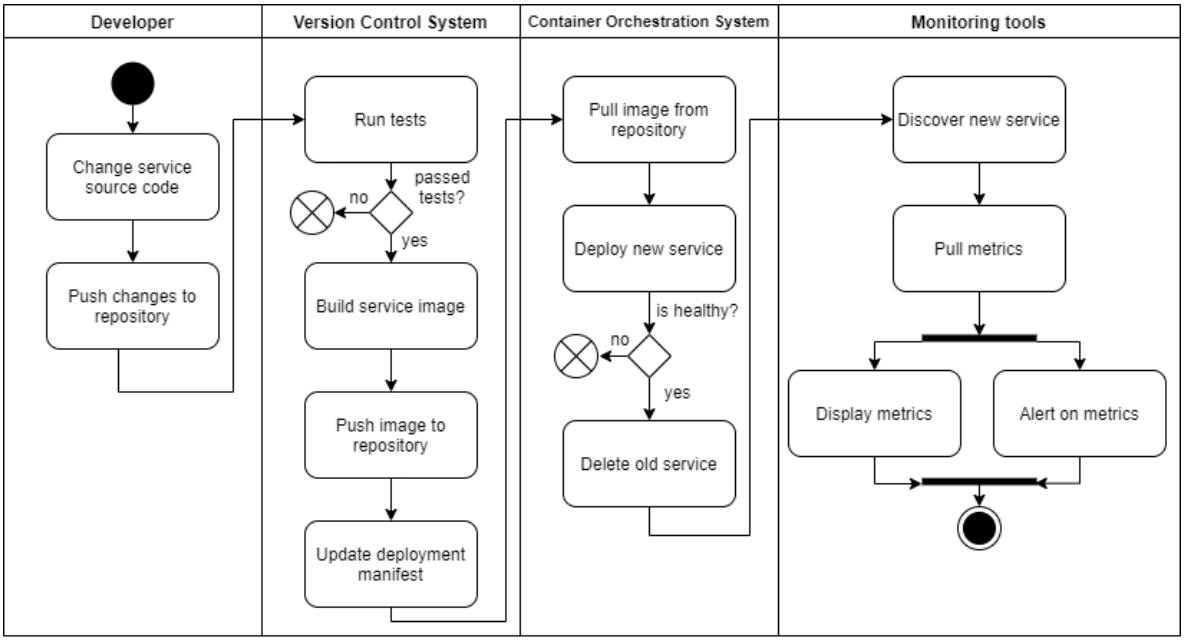

Figure 10: Deployment process for services in the infrastructure

The process is separated in four different stages, according to the responsible actors: the Developer, Version Control System, Container Orchestration System and Monitoring Tools. The first stage is the only one where the changes are manual, whereas the remaining three are fully automated.

The process begins with the developer applying changes to a service's source code. This source code is then pushed to the version control system, which has an embedded CI/CD tool. This tool is responsible for automatically executing a series of tests present in the source code, which range from unit tests to static code analysis. If those tests pass, a new service image is built with the manifest that is part of the source code. This image is then pushed to an image repository, which will be used by the Container Orchestration System. It pulls the uploaded image and deploys the service according to its deployment manifest, which describes the number of replicas of the service, the computing requirements and the dependencies of other services. If it manages to deploy the service correctly, it removes the older version of the service. After detecting the new service through the service discovery component, the monitoring server starts pulling metrics from the new version of the deployed service. These metrics are then displayed through visualization tools, and can be used to create new specific alerts.

5

IMPLEMENTATION

In the last Chapter, the architecture of the implemented system was presented and a platform independent model was discussed for each of the components: the system infrastructure, the monitoring infrastructure and the deployment process for services.

This Chapter aims to describe the implementation process of those architectures. It begins with a discussion on the selected tools for the system, as a result of the analysis in Chapter 2, and proceeds to describe the deployment process that was automated to create a system according to the requirements elicited in the previous Chapter.

Initially, Section 5.1 presents the reasoning behind the selection of the tools used for the system implementation.

Finally, Section 5.2 details the steps in each of the deployment stages to obtain the system implementation. This Section ends with an evaluation of the resulting monitoring components, assessing if the system improves the observability of the case study and in which ways do container orchestration systems contribute to the monitoring process.

5.1 TOOLS SELECTION

The selection of tools used to implement the design proposed in Chapter 4 was driven by the list of requirements in Section 4.1 and the methodologies and technologies researched in Chapter 2.

5.1.1 *Version Control System*

A Version Control System (VCS) is going to be used in order to store and version control the source code for the infrastructure and the systems that run on it, following the IaC principles. As seen on Section 2.3.2, Git is the most popular choice among the available options. As the team that developed the case study presented in 3 has used the GitLab platform as its VCS, this will also be the used platform to manage the configuration code for the infrastructure.

The use of this platform also provides other useful features, such as a continuous integration server that has a native integration with the version control system. This server provides automated builds, tests, automatic deployments and storage of artifacts beyond source code, such as Docker images.

5.1.2 *Container Orchestration*

The Container Orchestration tool is one of the main components of the proposed system, as it provides a framework that allows describing the deployment of all the services. The chosen tool for the implementation was Nomad.

Using Docker Swarm was too restrictive, as it does not provide such a rich feature set in comparison to both Nomad and Kubernetes. It does not provide auto scaling capabilities, its logging features are very limited, and its adoption is very small in the industry. It also does not have a configurable Public Key Infrastructure (PKI) system, which is a key service for the case study. However, its simplicity and reduced feature set would make it the fastest to deploy, as the process is the simplest in comparison to the other two tools. Likewise, Docker Swarm does not provide out-of-the-box monitoring tools, which increases the complexity in setting up an observable infrastructure.

Another option would be to use Kubernetes. According to a survey from the Cloud Native Computing Foundation (CNCF) in 2020, Kubernetes is the most popular Container Orchestration tool, with 91% of respondents reporting some usage of the tool [62]. Being the most popular option, it is also the framework with the most native tools available for various purposes, which range from monitoring to packaging applications with Helm [15]. However, its setup and configuration is also the most complex of the analyzed tools. The official documentation even advises handing off some of the configuration process to turnkey cloud solutions [103]. If that is not possible, a number of tools have been developed to automatically configure and deploy a Kubernetes cluster in a public cloud provider, such as Kops [104]. This tool not only assists in the setup and configuration of a cluster, but also provisions the necessary cloud infrastructure. It does not, however, help with the process of setting up a cluster in a hybrid or on-premises infrastructure.

Lastly, Nomad was the chosen tool due to some advantages it provides over the other two analyzed options. Firstly, it is the simplest tool to install. It is a single binary and requires no external services for coordination or storage. Secondly, it integrates natively with already used tools such as Vault, as well as Consul for service discovery. Thirdly, it supports workloads besides containers, which provides further flexibility to the cluster. Finally, it also includes out-of-the-box a set of monitoring capabilities which enhance the observability onto the cluster, such as memory and CPU usage of both nodes and services.

5.1.3 *Provisioning and configuration*

To automate the process of provisioning and configuring the infrastructure as well as its services, it is necessary to use a mix of the tools analyzed in both Chapter 2.3 and Chapter 2.4.

Ansible will be used due to its agent-less architecture, reducing the number of components on each machine as they do not need to have any software related to configuration management running. It will be used to configure the system in a way that is predictable and repeatable, to minimize the possibilities of creating snowflake environments. It is also compatible with various types of infrastructure, ranging from public cloud to on-premises, giving it a broad range of application and adaptability. It will not, however, be used to provision infrastructure due to limitations of its procedural behavior. Procedural code for provisioning infrastructure does not capture its state, therefore requiring taking into account in what state is the infrastructure at any moment to avoid duplicating resources or end up with a snowflake.

Terraform will be another used tool in two different contexts: to configure specific software components, and to provision public cloud infrastructure.

The software components that can be configured through Terraform are Nomad and Vault: the container orchestrator, and the secret management tool which provides PKI capabilities. Terraform is supported by both components, is developed by the same company (Hashicorp) and it is the recommended tool to configure logical aspects of their operation. An alternative would be to create a shell script which invokes their HTTP APIs and manually manages the state. This contrasts with the benefits of the declarative approach of Terraform, as the code always reflects the latest state of the configuration and wraps the HTTP calls in a common language (HCL).

In order to provision cloud infrastructure, Terraform will also be used due to its broad compatibility with the largest public cloud providers, as well as the other benefits of treating the cloud infrastructure as code as discussed in Section 2.4.1. Having a common language reflecting the infrastructure of a specific cloud provider also benefits the transition between them, as there is no need to adopt a different tool to provision the desired resources. This allows the creation of an abstraction layer between provisioning the infrastructure and the configuration of the orchestrator and monitoring tools.

5.1.4 *Monitoring*

In order to provide continuous monitoring to the deployed applications, there needs to be a combination of the tools analyzed in Chapter 2.5 as well as specific capabilities of the chosen orchestrator tool in Section 5.1.2.

Prometheus was the chosen tool to collect and store metrics as time series data, fulfilling the role of the monitoring server in the proposed architecture at Section 4.3. It is the only monitoring tool that has reached the "graduated" level according to the Cloud Native Computing Foundation (CNCF), as it is geared towards cloud-based solutions and micro-services. It integrates natively with the container orchestrator tool as well as with its service mesh. It also has broad compatibility with development frameworks and languages, with twenty client libraries and hundreds of exporters of metrics from software tools, making it a flexible choice for the collection of metrics from deployed applications.

It also has an advantage over the other tools: an integration with the Consul Catalog API that retrieves targets from the service mesh. This API implements the Service discovery component in Figure 9, decoupling the monitoring configuration from the observable targets.

The monitoring server will be complemented with Alertmanager, as it is the recommended tool for sending alerts when using Prometheus.

In terms of storing the collected metrics, the default local storage will be used due to its custom, highly efficient format. This choices simplifies the deployment, and reduces the overall cost of the system when in comparison to distributed and remote storage alternatives.

For the visualization components of the monitoring architecture, Grafana was deemed the best fit. It is the recommended visualization tool by Prometheus, as it supports querying Prometheus servers for metrics. Kibana, another visualization tool, only supports querying Elasticsearch instances. This contrasts with the other data sources that can be used by Grafana, such as Graphite, InfluxDB and also Elasticsearch. Grafana is also compatible with configuration as code principles, as it is configurable with *ini* files and its dashboards can also be stored and version controlled in JSON files.

5.2 IMPLEMENTATION ARCHITECTURE

The complete deployment of the implemented system is split in four stages: the provisioning of the infrastructure, the deployment and configuration of the PKI, the deployment and configuration of the container orchestrator and the deployment and configuration of the monitoring services. These four stages allow the consequent integration of a software component in a resilient platform with readily available metrics. They are depicted in Figure 11, which also presents the main components that each stage requires and produces.

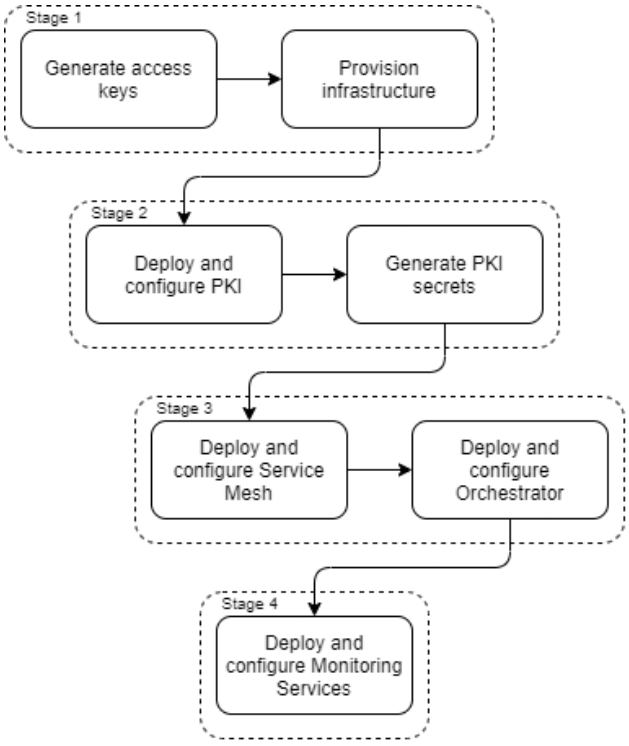

Figure 11: Implementation architecture in stages

5.2.1 *Infrastructure provisioning*

The first step in the deployment of the system is to generate an SSH key pair. This pair of keys will be used to access the provisioned machines, instead of using password-based authentication.

The next step in the deployment is the provisioning of the necessary infrastructure to meet the demands of the architecture proposed in Section 4.2.

Two different providers were selected: one public cloud provider, and one local environment. The goal of developing two different infrastructure providers was to demonstrate that this stage of the deployment process can be adapted to any provider without changing the following stages.

DigitalOcean

The chosen public cloud provider for the infrastructure was DigitalOcean, due to the availability of credits and its clear and predictable pricing model. A Terraform provider for DigitalOcean was used in order to deploy the necessary resources: one Virtual Private Cloud, five Droplets and one Block Storage Volume. The mapping between these resources and the proposed design in Figure 8 is in Table 1.

Name	**Quantity**	**Product**	**Description**
Server	3	Basic Droplet	2GB of RAM, 1 core and 50GB of storage
Client	2	Basic Droplet	1GB of RAM, 1 core and 25GB of storage
Intranet	1	Virtual Private Cloud	Private network isolated from the public internet
Block Storage	1	Block Storage Volume	5GB of persistent storage

Table 1: DigitalOcean resource mapping

These resources were then bundled into a Terraform Module, which can be configured with the variables described in Table 2.

Name	**Default value**	**Description**
vpc_name	mid-network	Name for the Virtual Private Cloud
vpc_ip_range	10.10.10.0/24	IP Range for the Virtual Private Cloud
region	lon1	Region to deploy the resources
num_servers	3	Number of servers
num_clients	2	Number of clients
num_volumes	1	Number of block volumes
volume_size	1	Size in GB for each volume
project_name	mid	Unique project name in Digital Ocean account
ssh_key_name	mid-ssh-key	SSH key name as uploaded in Digital Ocean
access_token	(not applicable)	Digital Ocean Personal Access Token

Table 2: DigitalOcean Terraform Module input variables

The use of this module requires manual input. It is necessary to generate a personal access token within Digital Ocean in order to use its API. It is also required to upload the public key of the generated SSH key pair to access the deployed resources.

Each of the five deployed virtual machines is tagged according to its expected use: the servers are tagged with *server* and the clients are tagged with *client*. There is also another tag named *vault* which is used to determine in which machine the Vault instance is deployed.

After using this module, it is possible to retrieve the virtual machines' public IP addresses in order to proceed with further configuration.

Vagrant

In order to test the infrastructure in a local environment and avoid any billing costs, Vagrant was chosen as the tool to automatically configure and provision virtual machines. It allows the configuration of the infrastructure in a *Vagrantfile*, which follows the Ruby language's syntax. The file describes the necessary resources: the five virtual machines, and a private network. This implementation does not include a block storage volume, as it is only intended to be used for development and testing purposes. As in DigitalOcean, this approach also groups the machines according to their use: *server*, *client* and *vault*. This approach allows the next stages to be independent from the chosen infrastructure provider, as they can retrieve the same information.

5.2.2 *Public key infrastructure deployment and configuration*

After provisioning the infrastructure, an instance of Vault is deployed and configured in order to provide the cluster with PKI capabilities. This instance is also used by the Mobile Identification system as described in Chapter 3 to provide certificates to the mDL Association and mDL Reader apps.

The first step in this stage is to deploy the Vault instance, and the process is depicted in Figure 12. All the activities are carried out by an Ansible playbook, whose tasks related to the deployment of the PKI are grouped logically into an Ansible role. The inventory used to apply this playbook is retrieved from the previous stage from the machines that are tagged with *vault*. After this process, a container running Vault is running on the selected machine, and the initial credentials are stored locally on a folder only accessible with root permissions. These credentials include the Vault root token and the unseal keys [78]. The instance stores its encrypted data locally on the machine, making it a critical point of failure. This shortcoming can be addressed with a Vault deployment in multiple instances, or with a network file system.

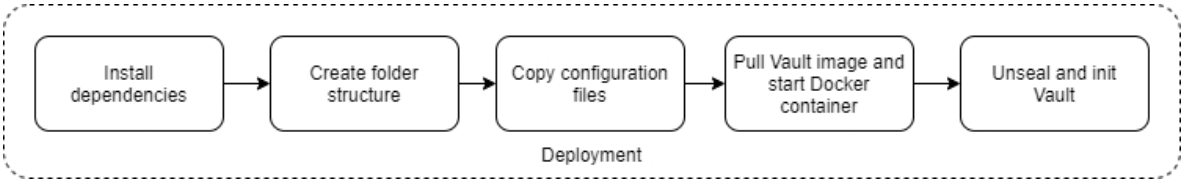

Figure 12: Activity diagram of the PKI deployment process

After completing the deployment process, the next step is to proceed with the configuration of the Vault instance as per Figure 13. These activities are described in a Terraform configuration, which requires two environment variables: *VAULT_TOKEN* and *VAULT_ADDRESS*. The former is the root token which was generated in the deployment process, and the latter is the address of the machine on which Vault was deployed. Both these variables can be retrieved from the output of the previous step.

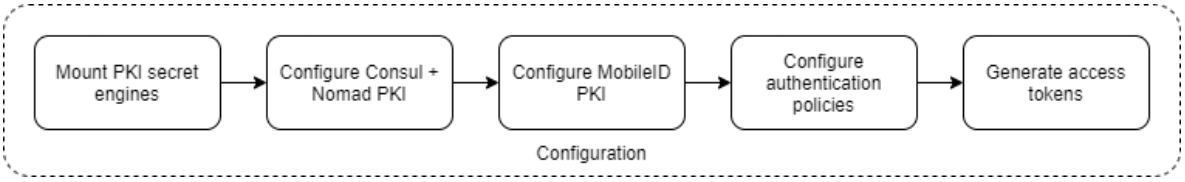

Figure 13: Activity diagram of the PKI configuration process

Each of the activities in the process is grouped into its own Terraform module.

The Consul and Nomad PKI secret engines provide two different certificate authorities. In each of these, there is a trust chain that complements the CA certificate with an intermediate authority. This authority has a Vault role which enables the automatic provisioning of mTLS certificates to both Consul and Nomad.

The next module to be configured is the MobileID PKI Module. This generates two different certificate chains: one for the *mdoc* readers, and one for the Issuing Authority. An overview of both chains is depicted in Figure 14.

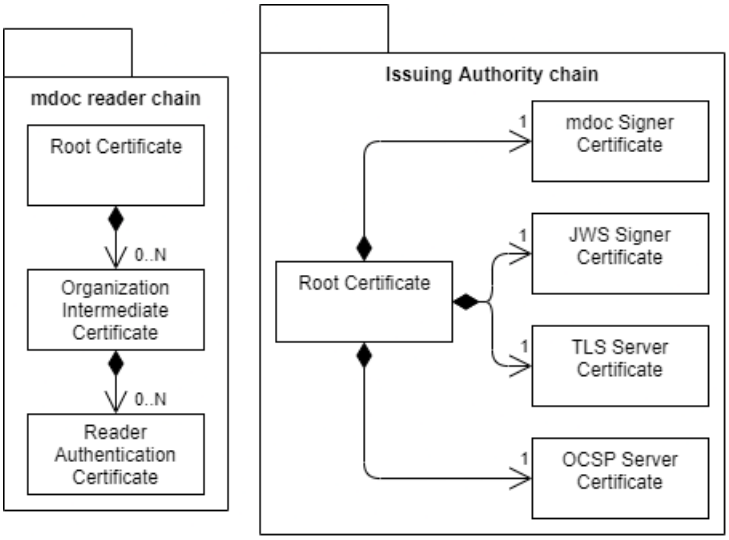

Figure 14: Class diagram of the MobileID PKI

The first is composed of a single root certificate, whose chain of trust will be expanded during the operation of MobileID instances through the dynamic creation of organizations of authorized readers. Each of these organizations will have an intermediate authority that lays under the original root certificate. This intermediate authority will issue individual certificates to each authorized reader for them to present when requesting a *mdoc* to a *mdoc* holder.

The second is composed of a root certificate, which signs four leaf certificates. The specification of each certificate is stored in the Terraform module as a Vault role according to the Annex B of the ISO 18013-5 specification. This allows the MobileID instances to automatically issue new certificates by connecting to Vault and requesting a new certificate from the desired role.

After completing the configuration of the MobileID secret engine, the next step in the configuration process of the PKI is the configuration of authentication policies. These policies limit the scope of access of the applications that interact with Vault. This follows the principle of least privilege, which states that every program must be able to access only the information and resources that are necessary for its legitimate purpose.

To limit the use of the root token to this step in the configuration process, an administrator user is created with a given password, which has full access to the mounted secret engines, and can mount additional engines.

A Vault *Approle* is created for the Django backend. This results in a pair of information: a *secret ID*, and a *role ID*. With both of these credentials, the Django backend is able to login to the Vault instance and get an access token. This token is short-lived and only provides enough permissions to interact with both chains as seen on Figure 14. This is possible due to the applied policy described in an HCL file. It describes which secret engines the *Approle* is allowed to access and edit.

Finally, two policies are configured for the Nomad cluster: one for the management of Vault tokens, and another for the issuing and renewal of mTLS certificates. The first is useful for the regular operation of the orchestrator, as it allows jobs submitted to Nomad to request tokens from Vault. This feature can be integrated into the Django backend, instead of using the process involving the *Approle*. The second policy is necessary for the *consul-template* daemon that periodically renews the mTLS certificates to access the Vault instance with only the necessary permissions.

After completing these tasks, the Vault token for Nomad is created, as well as the tokens to be used by the *consul-template* daemon to update the mTLS certificates for the Consul and Nomad clusters.

5.2.3 *Container orchestration deployment and configuration*

The next stage in the system implementation is the deployment and configuration of the container orchestration system.

This system is comprised of two interacting tools: Consul and Nomad. Consul is the service mesh that connects the jobs that run in the container orchestrator Nomad. Both of these tools require certificates from the PKI in order to have mTLS enabled.

Consul-template daemons configuration

The first step in this stage is the configuration of the *consul-template* daemons that will periodically update the mTLS certificates in both the client and the server machines. This daemon requires two sets of files: configuration files, and template files.

A configuration file declares, for each certificate and private key, three attributes: the template file, where should the daemon place the generated secret and which command to execute when updating the secret. An example configuration file can be seen below for the Nomad server:

```
template {
  source      = "/opt/consul-template/templates/nomad/ca.crt.tpl"
  destination = "/opt/nomad/tls/ca.pem"
  command     = "sudo systemctl reload nomad"
}

template {
  source      = "/opt/consul-template/templates/nomad/server.crt.tpl"
  destination = "/opt/nomad/tls/server.pem"
  command     = "sudo systemctl reload nomad"
}

template {
  source      = "/opt/consul-template/templates/nomad/server.key.tpl"
  destination = "/opt/nomad/tls/server-key.pem"
  command     = "sudo systemctl reload nomad"
}
```

Listing 5.1: Consul-template configuration file for Nomad server

This configuration file declares three files that should be handled by the daemon. The first is related to the root certificate of the trust chain of Nomad, while the last two are the server certificate and private key that encrypt the traffic between Nomad servers.

A template file describes in the *Go-template* format how to generate secrets from a Vault secret engine. The source field in each template block in the configuration file describes the location of the corresponding template file. An example template file can be seen below for the Nomad server certificate:

```
{{ with secret "nomad/inter-ca/issue/nomad-global" "common_name=server.global.
    nomad" "ttl=24h" "alt_names=localhost" "ip_sans=127.0.0.1"}}
{{ .Data.certificate }}
{{ end }}
```

Listing 5.2: Consul-template template file for the Nomad server certificate

This template file declares that the *consul-template* daemon shall access the secret engine with path *nomad/inter-ca/issue/nomad-global* and issue a certificate with the given attributes and time to live.

The configuration files are different for Consul and Nomad, and they also differ from client to server.

In order to deploy the daemons, an Ansible role named *consul-template* was created. This role copies the configuration files for the servers and clients to the respective machines, and ensures that the *systemd* service for the *consul-template* daemon is healthy and running. This daemon is responsible for periodically updating the files if their TTL is close to expiring.

Consul deployment

The next step in this stage is to deploy a Consul cluster so that Nomad nodes (clients and servers) can automatically connect to each other, and so that jobs deployed on the Nomad cluster can connect with each other through Consul's service mesh.

The Consul cluster has an architecture that is similar to Nomad's: it consists of an odd number of servers and any number of clients. Given the proposed infrastructure in Figure 8, it shall have servers running on machines tagged *server* and clients running on machines tagged *client*.

The deployment process for deploying the Consul cluster is depicted in Figure 15. The activities in the diagram are encapsulated with an Ansible role, which executes each activity as a task that requires the successful completion of the previous task.

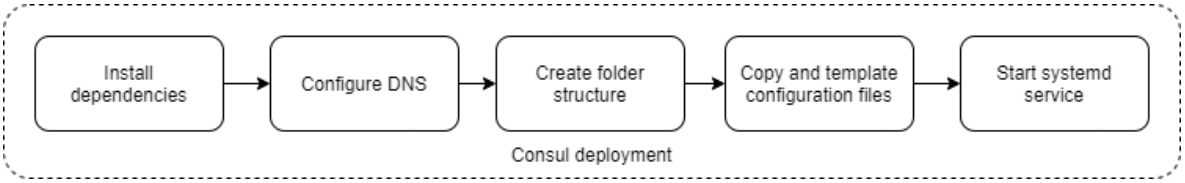

Figure 15: Activity diagram of the Consul deployment process

Given that Consul acts as a service mesh, configuring the machine's DNS is valuable so that registered services can be reached according to the Consul's routing rules. By routing domains that end in *.consul* to Consul, any service that is deployed through Nomad and registered in Consul can be reached through its service name, instead of its transient assigned IPs and ports.

The Consul's configuration files are written according to the *Jinja2* template format [139]. This allows Ansible to automatically replace the desired variables with automatically collected values. An example configuration file for the Consul server is found below:

```
verify_incoming = true
verify_outgoing = true
verify_server_hostname = true
ca_file = "/opt/consul/tls/ca.pem"
cert_file = "/opt/consul/tls/server.pem"
key_file = "/opt/consul/tls/server-key.pem"
auto_encrypt {
    allow_tls = true
}
ui_config {
    enabled = true
}
encrypt = "{{ consul_encryption_key }}"
server = true
node_name = "{{ consul_node_name }}"
bind_addr = "{{ consul_bind_address }}"
```

```
retry_join = ["{{ consul_server_address }}"]
bootstrap_expect = {{ num_servers }}
enable_syslog = true
data_dir = "/opt/consul/data"
ports {
    grpc = 8502
}
connect {
    enabled = true
}
client_addr = "{{ consul_bind_address }} 127.0.0.1"
```

Listing 5.3: Consul configuration file for server

In this configuration file, there are five different variables that are replaced during the execution of the Ansible role. The consul encryption key is an automatically generated string; the consul node name is the machine's name on which the server is being deployed; the consul bind address is the private IP address of the machine; the consul server address is the private IP address of the first server to be deployed; and the number of servers matches the number of provisioned servers.

After populating the templates and copying all the necessary files, a systemd job is created and executed in both servers and clients, completing the deployment of the Consul cluster.

Nomad deployment

Having the Consul cluster configured and deployed, it is possible to deploy and configure Nomad. Figure 16 depicts all the activities related to the deployment process for Nomad.

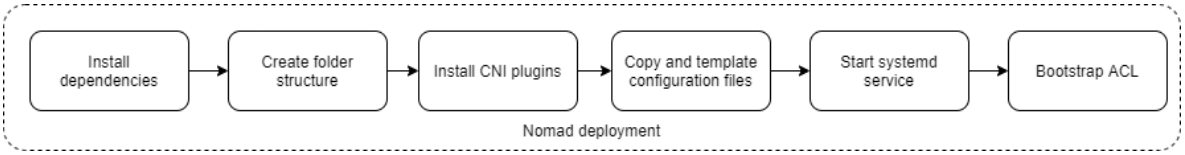

Figure 16: Activity diagram of the Nomad deployment process

Like in the previous deployment processes, this one is also encapsulated by an Ansible role. This role goes through all the necessary tasks, and ensures that they are completed successfully before moving onto the next task.

One of the activities in this process is the installation of CNI plugins. These are a requirement for Nomad in order to properly configure the network namespaces used to enable Consul Connect. This feature provides service-to-service connection authorization and encryption using mTLS, securing inbound and outbound traffic between services automatically.

A similar step in the deployment process of Nomad to the process of Consul is the templating used to automatically fill the necessary variables for Nomad's configuration files. The template for Nomad's client nodes is as follows:

```
name = "{{ inventory_hostname }}"
addresses {
    rpc = "{{ nomad_private_address }}"
    serf = "{{ nomad_private_address }}"
    http = "0.0.0.0"
}
consul {
    address = "{{ nomad_private_address }}:8500"
}
advertise {
    http = "{{ nomad_private_address }}"
}
client {
    enabled = true
}
tls {
    rpc = true
    ca_file = "/opt/nomad/tls/ca.pem"
    cert_file = "/opt/nomad/tls/client.pem"
    key_file = "/opt/nomad/tls/client-key.pem"
    verify_server_hostname = true
}
plugin "docker" {
    config {
        volumes {
            enabled = true
        }
        allow_privileged = true
    }
}
```

Listing 5.4: Nomad configuration file for client

In this template there are two variables that are filled during the execution of the Ansible role. The *inventory_hostname* is the machine's name on which the client is being deployed; the nomad private address is the private IP address of the machine. The client is exposed to the public network, while its connection to the servers is established through the private network. The addresses of the Nomad servers are retrieved automatically through the Consul client that is already running on the machine. As the servers are deployed before the clients, they register themselves as Consul services. When the clients are being deployed, they query the Consul client for the IP addresses of the *nomad-server* service.

After copying and templating all the configuration files, the systemd service is started on each machine, and the Nomad cluster is considered operational. However, it lacks the security and organizational benefits of an ACL system.

The ACL system is kickstarted in a recently created cluster through a Bash script that returns a management token. This token is stored locally on a folder which is only accessible with root permissions, and will be used in the configuration step for the Nomad cluster.

Nomad configuration

With the Nomad cluster operational and with its ACL system enabled, it is necessary to configure a basic set of services in the cluster. Those services are part of the general namespace, as depicted on Figure 17, and are described with an HCL file according to Nomad's syntax.

The Fabio [122] service is a load balancer for applications that are deployed and managed by a Consul service mesh. As all the jobs deployed on Nomad are registered in Consul, no extra configuration is necessary for those jobs to be accessible through this load balancer.

A GitLab Runner is also deployed on the cluster in order to provide CI/CD capabilities to GitLab repositories that store code intended to be tested and deployed in the cluster. This job requires a unique token that is provided when registering a runner with GitLab.

The Prometheus and Grafana jobs are the other general services that are deployed automatically in the cluster, and they are discussed further in Section 5.2.4.

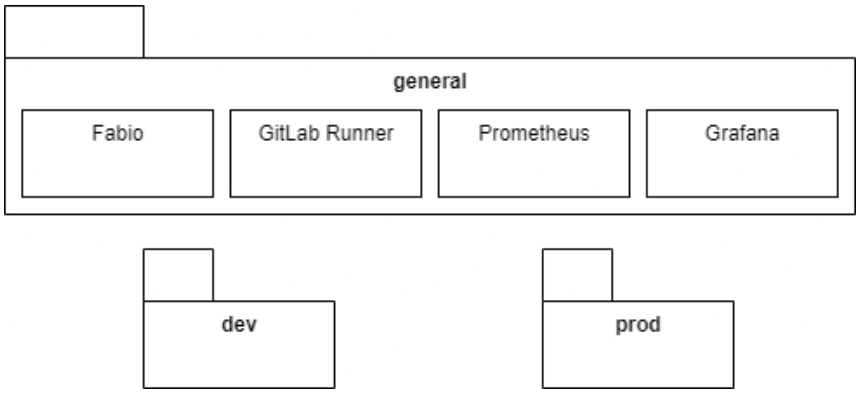

Figure 17: Class diagram of Nomad's namespace configuration

The configuration of the cluster is done with a Terraform module, which requires two environment variables: the address of the Nomad cluster, and a management token. Both of these inputs are retrieved from the deployment process. This Terrafom module also creates an ACL token that can be used to authorize other systems to deploy onto the *dev* or *prod* namespaces. Both are created for the deployment and categorization of the business components of the MobileID project.

5.2.4 *Monitoring deployment and configuration*

Two of the services that are automatically configured in the Nomad cluster are Prometheus and Grafana, as depicted on Figure 17. These services implement the proposed design as discussed in Section 5.1.4.

The Prometheus service job is specified in an HCL file, and its configuration template is automatically filled by a template stanza. This block in the configuration file uses a template renderer that populates attributes from environment variables, Consul data, or Vault secrets. This file also specifies the name of the service to be registered in Consul and the URL path the load balancer should use to redirect external traffic to Prometheus.

The Prometheus configuration file allows the specification of a set of targets and parameters describing how to scrape them, with the section *scrape_config*. Two jobs were defined for this section: one to retrieve metrics from the Nomad cluster itself, and another to retrieve metrics from services registered in Consul. The latter allows Prometheus to automatically collect metrics from any application deployed in the Nomad cluster and registered in Consul. Its specification in the configuration file can be seen below.

```
scrape_configs:
  - job_name: 'consul_sd_metrics'

    consul_sd_configs:
    - server: 'consul.service.consul:8500'

    relabel_configs:
    - source_labels: ['__meta_consul_service']
      regex: '(.*)-sidecar-proxy'
      action: drop
    - source_labels: ['__meta_consul_tags']
      regex: '(.*)prometheus(.*)'
      action: keep
    - source_labels: ['__meta_consul_service']
      target_label: service
      replacement: '$1'
    - source_labels: ['__metrics_path__', '__meta_consul_service']
      regex: '(.*);(.*)'
      replacement: '/$2$1'
      target_label: '__metrics_path__'

    scrape_interval: 5s
    metrics_path: /metrics
    params:
      format: ['prometheus']
```

Listing 5.5: Prometheus scrape configuration for Consul services

This configuration includes some automatic relabeling of meta labels provided by Consul. The services registered with the *sidecar-proxy* labels will be ignored, and only the services with the tag *prometheus* will be scraped.

The *metrics_path* label is updated with Consul's service name, so that Prometheus can use the load balancer to access the service.

The Prometheus instance itself exposes metrics at */metrics*, and therefore it can also be scraped using this strategy as depicted on Figure 18. This Figure also highlights the labels that were added during the relabeling phase, as well as the IP address of the instance on which Prometheus is running.

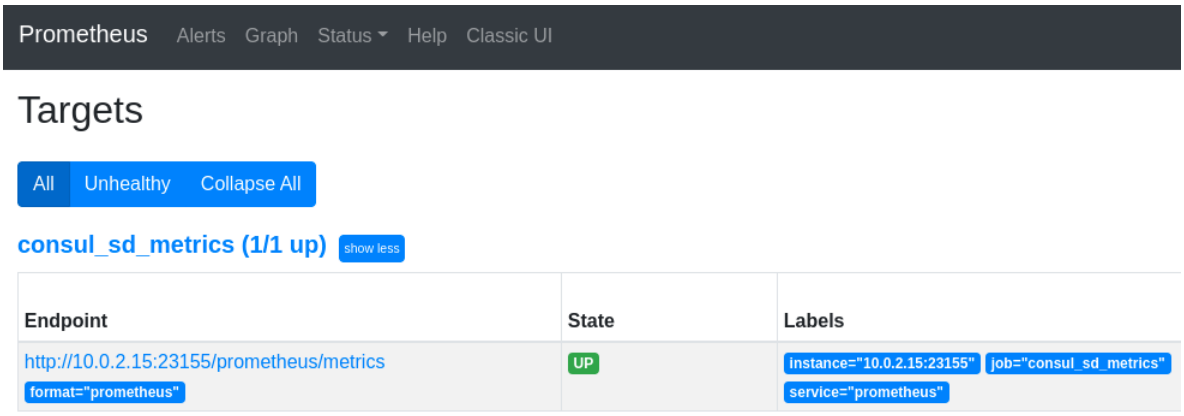

Figure 18: Automatic service discovery in Prometheus' web interface

The metrics collected from the registered services can then be queried and displayed with Grafana. As the type of metrics, and their respective name can vary from application to application, Grafana is automatically configured to have Prometheus as its data source. Furthermore, the job specification of Grafana allows the automatic configuration of dashboards that can be developed for specific services. An example for the configuration of a dashboard that automatically displays useful information about the status of the Nomad cluster can be seen below.

```
artifact {
  source = "github.com/alves-luis/nomad-monitoring/grafana/dashboards"
  destination = "local/dashboards/"
}
```

Listing 5.6: Artifact stanza for Nomad job

One or more of these stanzas can be added to Grafana's job file for Nomad, so it can download the artifacts and automatically map them to the configuration path inside the container. The contents on the source URL are the JSON files that describe the desired Grafana dashboard, and the destination indicates the path inside the container where the files will be placed.

5.2.5 *Service integration*

With the monitoring components properly configured, and an operating container orchestration system, it is possible to integrate workloads to the cluster that are continuously monitored. The collected metrics are useful both for maintaining the service as well as for improving it for its stakeholders.

In order to deploy a service to the cluster, the repository which stores its source code must be configured to interact with Nomad. As declared in Section 5.1.1, the VCS that will interact with Nomad is GitLab. As the cluster already includes a service which runs GitLab CI jobs, the cluster's HTTP private API endpoint is available to that container. In order to deploy and manage jobs in the cluster, the GitLab Runner requires two variables that should be added to the repository's CI/CD variables: *NOMAD_ADDR* and *NOMAD_TOKEN*. These variables will be automatically added to the environment of the CI/CD pipelines that are executed by GitLab CI.

Having the repository configured, it is now necessary to add two files to its root: one that specifies the CI/CD pipeline, and another that specifies the Nomad job.

CI/CD pipeline

In GitLab, the pipeline file has the name *.gitlab-ci.yml*. In this file, the steps depicted in Figure 10 in the VCS column are described. An implementation of this column for the MobileID backend is as follows:

```
stages:
  - test
  - build
  - deploy

test_static:
  stage: test
  image: python:3.7.7-buster
  script:
    - pip3 install flake8
    - flake8 .
  tags:
    - mid
  allow_failure: true

test_integration:
  stage: test
  image: python:3.7.7-buster
  variables:
    POSTGRES_PASSWORD: "password"
    POSTGRES_DB: "mdoc"
    POSTGRES_USER: "admin"
  services:
    - name: postgres:12.3
      alias: db
  script:
```

```
    - pip3 install -r requirements.txt
    - cp .env.ci .env
    - python3 manage.py migrate
    - python3 manage.py test -v 2
  tags:
    - mid

build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - mkdir -p /kaniko/.docker
    - echo "{\"auths\":{\"$CI_REGISTRY\":{\"username\":\"$CI_REGISTRY_USER\",\"
        password\":\"$CI_REGISTRY_PASSWORD\"}}}" > /kaniko/.docker/config.json
    - /kaniko/executor --context $CI_PROJECT_DIR --dockerfile $CI_PROJECT_DIR/
        Dockerfile --destination $CI_REGISTRY_IMAGE/backend:$CI_COMMIT_SHORT_SHA
  tags:
    - mid

deploy:
  stage: deploy
  image: $CI_REGISTRY_IMAGE/nomad:latest
  script:
    - envsubst '${CI_COMMIT_SHORT_SHA} ${CI_REGISTRY_IMAGE}' < mdoc.nomad > job.
        nomad
    - cat job.nomad
    - nomad validate job.nomad
    - nomad plan job.nomad || if [ $? -eq 255 ]; then exit 255; else echo "
        success"; fi
    - nomad run job.nomad
  tags:
    - mid
```

Listing 5.7: GitLabCI pipeline for backend service

The pipeline is comprised of three different stages: test, build and deploy.

At the test stage, there are two parallel steps: one for executing static analysis of the Python code according to the Flake8 tool [43]; and the other for executing the tests developed with Django's testing framework [61]. The static analysis step only warns the developer of potential issues, as it does not block the execution of the rest of the pipeline in case of failure.

As for the build stage, the pipeline uses a Kaniko image in order to build the project's image from its Dockerfile without requiring a Docker daemon [69]. This artifact is then pushed to the repository's image registry tagged with the commit revision hash.

After completing the two previous stages, the pipeline proceeds to deploy the built image to Nomad, by replacing the variables *CI_COMMIT_SHORT_SHA* and *CI_REGISTRY_IMAGE* in the Nomad job definition file stored in the repository. This file is validated with Nomad, and its scheduler is invoked with a dry-run to determine what will happen after the file is submitted. If there is an error determining the plan's results, the pipeline is aborted. If there isn't any error, the job is updated and the new version is allocated on Nomad.

Nomad job definition

The other file that should be present in the source code repository is the Nomad job definition file. This file is specified in HCL and describes the schema of the service that will be deployed on Nomad. A definition file for the MobileID backend is as follows:

```
job "mdoc-backend-job" {
  datacenters = ["dc1"]
  type = "service"

  group "mdoc-backend-group" {
    count = 1
    network {
      port "http" {
        to = "8000"
      }
      port "db-port" {
        to = "5432"
      }
    }
    task "mdoc-backend" {
      driver = "docker"
      config {
        image = "{$CI_REGISTRY_IMAGE}/backend:{$CI_COMMIT_SHORT_SHA}"
        ports = ["http"]
      }
      service {
        port = "http"
        tags = ["urlprefix-/mdoc-backend", "mdoc-backend"]
        check {
          name     = "alive"
          type     = "http"
          path     = "/"
          interval = "10s"
          timeout  = "2s"
        }
      }
      template {
        data = <<EOH
          DBNAME = "{{key "service/mdoc-backend-db/name"}}"
```

```
          DBUSER = "{{key "service/mdoc-backend-db/user"}}"
          DBPASSWORD = "{{with secret "secret/mdoc-backend-db-password"}}{{.Data.
              value}}{{end}}"
          DBHOST = "db"
          DBPORT = "5432"
          DATABASE = "postgres"
        EOH

        destination = "secrets/.env"
        env = true
      }
    }
    task "mdoc-db" {
      driver = "docker"
      config {
        image = "postgres:13.4"
        ports = ["db-port"]
      }
      template {
        data = <<EOH
          POSTGRES_DB = "{{key "service/mdoc-backend-db/name"}}"
          POSTGRES_USER = "{{key "service/mdoc-backend-db/user"}}"
          POSTGRES_PASSWORD = "{{with secret "secret/mdoc-backend-db-password"
              }}{{.Data.value}}{{end}}"
        EOH

        destination = "secrets/.env"
        env = true
      }
    }
  }
}
```

Listing 5.8: Nomad job definition for MobileID backend

This definition file includes two tasks inside a group: the Django backend and the PostgreSQL database. The image was built during the pipeline, and it is pulled automatically by Nomad from the image registry when allocating a new job.

The mdoc-backend service is registered in Consul with the *service* stanza, allowing Prometheus to automatically scrape metrics exposed by Django with the helper library *django-prometheus* [44]. This registration also allows the load balancer to map the URL path */mdoc-backend* to this service, so that it can be accessed outside the cluster.

Both tasks use the template stanza in order to retrieve secrets from Vault. This avoids storing sensitive information such as database credentials in the source code repository, and thus strengthening the security of the implementation.

After being deployed successfully, the two containers are available on any client node of the cluster.

5.2.6 *Monitoring evaluation*

After completing the deployment and configuration of the cluster that automatically collects and displays information about the state of deployed services, it is important to reflect on how this improves the observability of such services.

The container orchestration system includes a basic set of monitoring tools that enable real-time observation of the service's physical resource usage. This can be seen on Figure 19, which is a screenshot of Nomad's web interface.

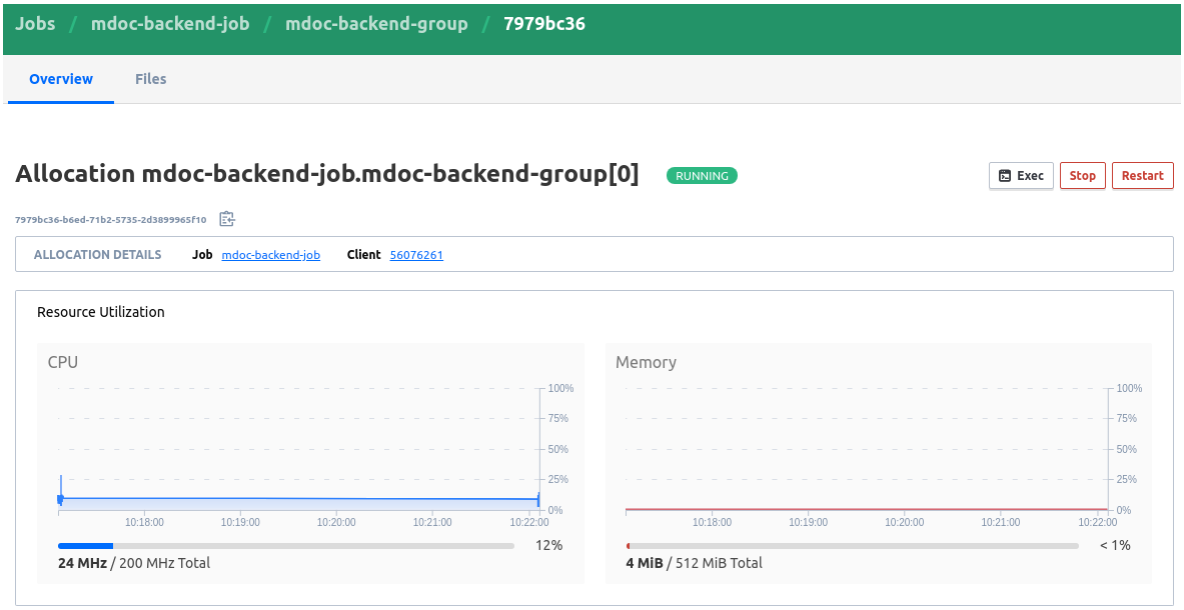

Figure 19: Service monitoring in Nomad's web interface

This panel displays two charts that track both the CPU and memory usage. This information can be used by the operator to assess if there are enough resources to meet the service's demand, and add new nodes if necessary. It can also be used by third-party tools such as Nomad's Autoscaler [77]. This tool can interact with Nomad's HTTP API to automatically determine the number of replicas that the service should have in order to meet a defined SLA.

Furthermore, this interface also provides real-time log streaming from both *stdout* and *stderr*, as well as a file system explorer for the service's jobs. That can be useful to understand the state of the service without directly connecting to the machine, providing greater flexibility on the tools used to assess the status of a service.

Moreover, this solution also improves the observability of the nodes that support the logical infrastructure of the cluster. Figure 20 is a screenshot of one of the infrastructure nodes' monitoring page of Nomad's web interface. Like in the service page, this panel also displays two charts that track both the CPU and memory usage.

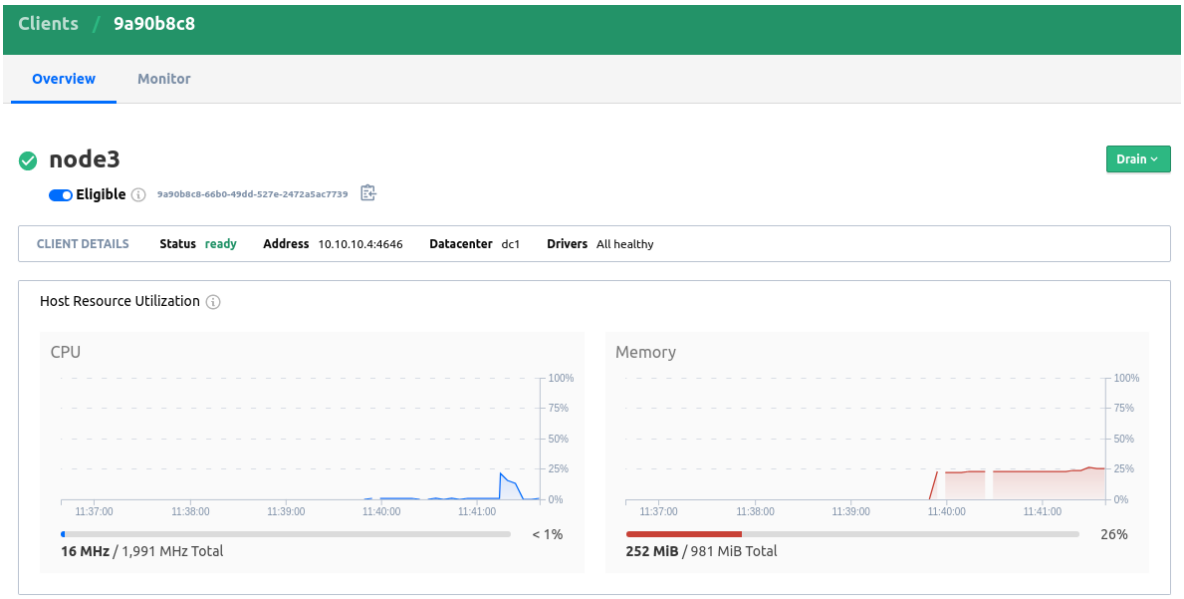

Figure 20: Node monitoring in Nomad's web interface

Although this set of monitoring tools built in Nomad expands the observability of both nodes and services, the configuration of Prometheus, Grafana and Alertmanager enhances the solution. It provides collection and handling of metrics that are exported by the services, and the services can provide those metrics by including libraries that are generally available for the most popular frameworks. These libraries expose a metrics endpoint that is discovered by Consul and consumed by Prometheus. With that endpoint, it is possible to analyze detailed metrics for a service such as the *mdoc-backend*: the frequency of requests by endpoint of the API, the distribution of requests throughout a time series, the frequency of status codes by endpoint of the API, when was the service unreachable and for how long the service was unavailable.

This detailed information about the service can then be used in three different ways: to configure a specific dashboard in Grafana, that displays useful information for an operator; to be queried using PrometheusQL in order to obtain data to drive operational or development decisions; to be processed automatically by Alertmanager in order to page the operators in case specific thresholds are met.

6

CONCLUSION AND FUTURE WORK

Throughout this dissertation, the design and development of an infrastructure that supports multiple services and provides observability of itself and its services was described. This was the main goal of the document and, as such, this practical work requires a conclusion that summarizes the outcomes of the exploratory research, while also addressing potential developments and future work that could improve this dissertation's contribution.

The first step in addressing the main goal of this dissertation was the study of DevOps and its methodologies, and in which ways they influenced the existing tools. The focus was on container orchestration frameworks, and monitoring systems.

All of the analyzed container orchestration frameworks were developed with the intention of streamlining the process of deploying, managing and scaling workloads. This increases the velocity with which teams can deploy new applications, and enables the fast feedback loop that DevOps advocates. Besides creating this self-contained execution environment for applications, these frameworks also provide basic observability to them, and to the cluster itself. However, they have some limitations, as they abstract the services that are executed on the framework and therefore cannot provide the extensive capabilities that dedicated monitoring systems do.

The monitoring systems that were studied during this dissertation also highlight how the paradigm has shifted from monolithic applications deployed on-premises to micro-services deployed on the cloud. As the responsibility of monitoring infrastructures has been shifted to monitoring services, new monitoring systems have been developed that focus more on the applications and its internal metrics rather than in the infrastructure itself. By collecting and analyzing the information about how an application performs through monitoring, it is possible to improve and launch new products or features that increase value for the stakeholders.

The analysis of the case study of this dissertation was valuable in order to understand the starting point of the final implementation. The automation of the deployment process, as well as the increased observability of the services are crucial improvements in the system. As there is no need for an operator to manually deploy each component, the process becomes less prone to errors and does not rely on a specific member of the team to execute a set of tasks. This also allows a more efficient use of human resources that can be deployed in activities that are not so easily automated. The automated process also strengthens the security of the deployment, as it adopts the principle of least privilege that abstracts the storage of secrets from the applications through the use of secret management tools.

This implementation also demonstrates how easily monitoring can be incorporated in the development process of applications through the use of client libraries that export metrics. These libraries are common for most of the development frameworks, and they don't need extensive configuration in order to be integrated with Prometheus.

As to future work, there is the potential of developing a tool that automatically detects if a deployed service has any metrics available to export. This can be useful to incorporate in a CI/CD pipeline so that developers can be warned about the lack of observability of their services, and be encouraged to integrate metrics exporters.

Another useful tool would be the automatic configuration of the service stanza in the Nomad configuration file. While intuitive, this file is still verbose and requires proper specification in order to automatically register the service in Consul and retrieve its metrics.

BIBLIOGRAPHY

[1] Go programming language, 2021. URL https://golang.google.cn/. Online; accessed 20-january-2021.

[2] Java programming language, 2021. URL https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html#. Online; accessed 20-january-2021.

[3] Python programming language, 2021. URL https://www.python.org/. Online; accessed 20-january-2021.

[4] Ruby programming language, 2021. URL https://www.ruby-lang.org/en/. Online; accessed 20-january-2021.

[5] Scala programming language, 2021. URL https://scala-lang.org/. Online; accessed 20-january-2021.

[6] A. Akbulut and H. G. Perros. Software versioning with microservices through the api gateway design pattern. In *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 289–292, 2019. doi: 10.1109/ACITT.2019.8779952.

[7] Amazon. Aws cloudformation, 2021. URL https://aws.amazon.com/cloudformation/. Online; accessed 16-january-2021.

[8] Amazon. Amazon elastic block store, 2021. URL https://aws.amazon.com/ebs/. Online; accessed 17-january-2021.

[9] Apache. Apache kafka, 2017. URL https://kafka.apache.org/. Online; accessed 11-october-2021.

[10] Apache. Zookeeper, 2020. URL https://zookeeper.apache.org/. Online; accessed 11-october-2021.

[11] Aqua. Docker swarm, 2021. URL https://www.aquasec.com/cloud-native-academy/docker-container/docker-swarm/. Online; accessed 28-january-2021.

[12] Atlassian. Why git?, 2020. URL https://www.atlassian.com/git/tutorials/why-git. Online; accessed 26-december-2020.

[13] Atlassian. What is devops | atlassian, 2021. URL https://www.atlassian.com/devops/what-is-devops. Online; accessed 12-january-2021.

[14] Atlassian. Bamboo continuous integration and deployment build server, 2021. URL https://www.atlassian.com/software/bamboo. Online; accessed 20-january-2021.

[15] Helm Authors. Helm, 2021. URL https://helm.sh/. Online; accessed 30-august-2021.

[16] AWS. What is devops?, 2021. URL https://aws.amazon.com/devops/what-is-devops/. Online; accessed 12-january-2021.

[17] AWS. Amazon web services (aws) - cloud computing services, 2021. URL https://aws.amazon.com/. Online; accessed 1-february-2021.

[18] AWS. Elastic load balancing - amazon web services, 2021. URL https://aws.amazon.com/elasticloadbalancing/. Online; accessed 1-august-2021.

[19] Axelos. *ITIL® Foundation: ITIL 4 edition*. TSO (The Stationery Office), part of Williams Lea, 2019.

[20] Microsoft Azure. Cloud computing services | microsoft azure, 2021. URL https://azure.microsoft.com/en-us/. Online; accessed 1-february-2021.

[21] Shay Banon. You know, for search, 2010. URL https://www.elastic.co/blog/you-know-for-search. Online; accessed 30-january-2021.

[22] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL https://doi.org/10.1145/1165389.945462.

[23] Richard Barker and Paul Massiglia. *Storage area network essentials*. Veritas. John Wiley & Sons, Nashville, TN, November 2002.

[24] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Principles behind the agile manifesto, 2001. URL https://agilemanifesto.org/principles.html. Online; accessed 12-december-2020.

[25] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml ain't markup language: Version 1.2, 2009. URL https://yaml.org/spec/1.2/spec.html. Online; accessed 10-january-2021.

[26] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[27] Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara, and Stephen Thorne. *The Site Reliability Workbook*. O'Reilly Media, Inc., 1st edition, 2018.

[28] Lou Bichard. Configuration management: What is it and why is it important?, 2020. URL https://www.plutora.com/blog/configuration-management. Online; accessed 12-december-2020.

[29] Kristian Bjerke-Gulstuen, Emil Wiik Larsen, Tor Stålhane, and Torgeir Dingsøyr. High level test driven development – shift left. 2015.

[30] Ian Buchanan. Configuration management, 2020. URL https://www.atlassian.com/continuous-delivery/principles/configuration-management. Online; accessed 23-december-2020.

[31] Ian Buchanan. History of devops | atlassian, 2021. URL https://www.atlassian.com/devops/what-is-devops/history-of-devops. Online; accessed 12-january-2021.

[32] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. The geojson format. RFC 7946, RFC Editor, August 2016.

[33] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf. *A Characterization Framework for Software Deployment Technologies*. University of Colorado - Department of Computer Science, 1998.

[34] Emiliano Casalicchio. Autonomic orchestration of containers: Problem definition and research challenges. 2017.

[35] Ceph. Ceph documentation, 2021. URL https://docs.ceph.com/en/latest/. Online; accessed 18-january-2021.

[36] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2020.

[37] Chef. Chef web docs, 2021. URL https://docs.chef.io/. Online; accessed 19-january-2021.

[38] Chef. chef/ohai, 2021. URL https://github.com/chef/ohai. Online; accessed 30-january-2021.

[39] Inc. Circle Internet Services. Continuous integration and delivery - circleci, 2021. URL https://circleci.com/. Online; accessed 20-january-2021.

[40] Google Cloud. Compute engine: Virtual machines (vms) | google cloud, 2021. URL https://cloud.google.com/compute. Online; accessed 1-february-2021.

[41] Google Cloud. Cloud load balancing | google cloud, 2021. URL https://cloud.google.com/load-balancing. Online; accessed 1-august-2021.

[42] Rhode Code. Version control systems popularity in 2016, 2016. URL https://rhodecode.com/insights/version-control-systems-2016. Online; accessed 26-december-2020.

[43] Ian Stapleton Cordasco. Flake8 documentation, 2016. URL https://flake8.pycqa.org/en/latest/manpage.html. Online; accessed 10-october-2021.

[44] Uriel Corfa. django-prometheus, 2021. URL https://github.com/GoogleContainerTools/kaniko. Online; accessed 12-october-2021.

[45] Bob Cotton. Why git?, 2017. URL https://blog.freshtracks.io/monitoring-is-dead-long-live-observability-235b62f4d1d1. Online; accessed 20-november-2020.

[46] Armon Dadgar. Hashicorp nomad, 2020. URL https://www.hashicorp.com/blog/nomad-announcement. Online; accessed 25-january-2021.

[47] Laura DiDio. Itic 2017-2018 global server hardware, server os reliability survery, 2018. URL https://www.ibm.com/downloads/cas/YGLRKEEKt. Online; accessed 9-December-2020.

[48] DigitalOcean. Digitalocean load balancers, 2021. URL https://www.digitalocean.com/products/load-balancer/. Online; accessed 1-august-2021.

[49] Fathima Dilhasha. Observability & monitoring — part 02, 2018. URL https://medium.com/the-devops-journey/observability-monitoring-part-02-d4d81b67c09a. Online; accessed 29-november-2020.

[50] Docker. Docker documentation, 2020. URL https://docs.docker.com. Online; accessed 9-january-2021.

[51] T. Dybå and T. Dingsøyr. Agile project management: From self-managing teams to large-scale development. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 945–946, 2015. doi: 10.1109/ICSE.2015.299.

[52] Elastic. Elastic stack: Elasticsearch, kibana, beats & logstash | elastic, 2021. URL https://www.elastic.co/elastic-stack. Online; accessed 30-january-2021.

[53] Elastic. Elastic stack and product documentation, 2021. URL https://www.elastic.co/guide/index.html. Online; accessed 30-january-2021.

[54] Nagios Enterprises. Table of contents . nagios core documentation, 2021. URL https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/toc.html. Online; accessed 30-january-2021.

[55] Nagios Enterprises. What is nagios?, 2021. URL https://www.nagios.org/about/. Online; accessed 30-january-2021.

[56] Erlang. Erlang programming language, 2021. URL https://www.erlang.org/. Online; accessed 20-january-2021.

[57] etcd. etcd, 2020. URL https://etcd.io/. Online; accessed 9-january-2021.

[58] Inc. F5 Networks. Nginx | high performance load balancer, 2021. URL https://nginx.org/. Online; accessed 11-october-2021.

[59] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. 2015.

[60] Nicole Forsgren. The 2019 accelerate state of devops - elite performance, productivity, and scaling, 2019. URL https://cloud.google.com/blog/products/devops-sre/the-2019-accelerate-state-of-devops-elite-performance-productivity-and-scaling. Online; accessed 13-january-2021.

[61] Django Software Foundation. Django | documentation, 2021. URL https://docs.djangoproject.com/en/3.2/. Online; accessed 10-october-2021.

[62] The Cloud Native Computing Foundation. Cncf survey 2020, 2021. URL https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Online; accessed 30-august-2021.

[63] Samuel H. Fuller, Victor R. Lesser, C. Gordon Bell, and Charles Kaman. Microprogramming and its relationship to emulation and technology. In *Conference Record of the 7th Annual Workshop on Microprogramming*, MICRO 7, page 151–158, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450374217. doi: 10.1145/800118.803855. URL https://doi.org/10.1145/800118.803855.

[64] Ethan Galstad. What does nagios mean?, 2009. URL https://support.nagios.com/knowledge-base/view-faq/?id=52&catid=35&faq_id=2. Online; accessed 30-january-2021.

[65] Git. Git, 2021. URL https://git-scm.com/. Online; accessed 1-february-2021.

[66] GitLab. Set up automated ci systems with gitlab | gitlab, 2021. URL https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/. Online; accessed 20-january-2021.

[67] Google. Cloud deployment manager, 2021. URL https://cloud.google.com/deployment-manager. Online; accessed 16-january-2021.

[68] Google. Storage options | google cloud, 2021. URL https://cloud.google.com/compute/docs/disks. Online; accessed 16-january-2021.

[69] Google. kaniko - build images in kubernetes, 2021. URL https://github.com/GoogleContainerTools/kaniko. Online; accessed 11-october-2021.

[70] Alibaba Group. Alibaba cloud, 2021. URL https://eu.alibabacloud.com/en. Online; accessed 1-february-2021.

[71] Object Management Group. *Deployment Configuration of Component-based Distributed Applications Specification*. OMG, 2004.

[72] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database, 2021. URL https://www.postgresql.org/. Online; accessed 30-january-2021.

[73] Sam Guckenheimer. What is infrastructure as code?, 2017. URL https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code. Online; accessed 16-january-2021.

[74] HAProxy. Haproxy - the reliable, high performance tcp/http load balancer, 2021. URL https://www.haproxy.org/. Online; accessed 1-august-2021.

[75] Hashicorp. Documentation | nomad by hashicorp, 2020. URL https://www.nomadproject.io/docs. Online; accessed 26-january-2021.

[76] HashiCorp. Introduction to terraform, 2021. URL https://www.terraform.io/intro/index.html. Online; accessed 15-january-2021.

[77] Hashicorp. Nomad autoscaler, 2021. URL https://github.com/hashicorp/nomad-autoscaler. Online; accessed 15-october-2021.

[78] Hashicorp. Vault by hashicorp, 2021. URL https://www.vaultproject.io/. Online; accessed 30-january-2021.

[79] Red Hat. What is configuration management?, 2020. URL https://www.redhat.com/en/topics/automation/what-is-configuration-management. Online; accessed 9-december-2020.

[80] Red Hat. What is provisioning?, 2021. URL https://www.redhat.com/en/topics/automation/what-is-provisioning. Online; accessed 16-january-2021.

[81] Red Hat. Continuous integration and delivery with ansible, 2021. URL https://www.redhat.com/rhdc/managed-files/ContinuousDelivery-WhitePaper.pdf. Online; accessed 26-january-2021.

[82] Red Hat. Ansible in depth, whitepaper, 2021. URL https://www.ansible.com/hubfs/pdfs/Ansible-InDepth-WhitePaper.pdf. Online; accessed 24-january-2021.

[83] Red Hat. Intro to playbooks, 2021. URL https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#tasks-list. Online; accessed 25-january-2021.

[84] Red Hat. Connection methods and details, 2021. URL https://docs.ansible.com/ansible/latest/user_guide/connection_details.html. Online; accessed 25-january-2021.

[85] Red Hat. Glusterfs documentation, 2021. URL https://docs.gluster.org/en/latest/. Online; accessed 17-january-2021.

[86] Jez Humble and David Farley. *Continuous delivery : reliable software releases through build, test, and deployment automation*. Addison-Wesley, 6th edition, 2010.

[87] ISO 18013-5:2021. Personal identification — iso-compliant driving licence — part 5: Mobile driving licence (mdl) application. Standard, International Organization for Standardization, Geneva, CH, September 2021.

[88] ISO/IEC AWI TS 23220-2. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 2: Data objects and encoding rules for generic eid systems. Standard, International Organization for Standardization, Geneva, CH, July 2019.

[89] ISO/IEC AWI TS 23220-3. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 3: Protocols and services for issuing phase. Standard, International Organization for Standardization, Geneva, CH, July 2019.

[90] ISO/IEC AWI TS 23220-4. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 4: Protocols and services for operational phase. Standard, International Organization for Standardization, Geneva, CH, July 2019.

[91] ISO/IEC AWI TS 23220-5. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 5: Trust models and confidence level assessment. Standard, International Organization for Standardization, Geneva, CH, July 2019.

[92] ISO/IEC AWI TS 23220-6. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 6: Mechanism for use of certification on trustworthiness of secure area. Standard, International Organization for Standardization, Geneva, CH, May 2020.

[93] ISO/IEC DIS 23220-1. Cards and security devices for personal identification — building blocks for identity management via mobile devices — part 1: Generic system architectures of mobile eid systems. Standard, International Organization for Standardization, Geneva, CH, November 2021.

[94] ISO/IEC/IEEE 24765:2017(E). Systems and software engineering – vocabulary. Standard, International Organization for Standardization, Geneva, CH, September 2017.

[95] Ron Jeffries. What is extreme programming?, 2011. URL https://ronjeffries.com/xprog/what-is-extreme-programming/. Online; accessed 15-january-2021.

[96] Jenkins. Jenkins, 2021. URL https://www.jenkins.io/. Online; accessed 20-january-2021.

[97] A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, 2015. doi: 10.1109/ICACEA.2015.7164727.

[98] Heike Jurzik. Version control systems and continuous deployment tools: A perfect fit, 2019. URL https://deploybot.com/blog/version-control-systems-and-continuous-deployment-tools-a-perfect-fit. Online; accessed 22-december-2020.

[99] Stefan Kempter. Service asset and configuration management, 2020. URL https://wiki.en.it-processmaps.com/index.php/Service_Asset_and_Configuration_Management. Online; accessed 14-december-2020.

[100] Michael Kerrisk. namespaces - overview of linux namespaces, 2020. URL https://man7.org/linux/man-pages/man7/namespaces.7.html. Online; accessed 9-january-2021.

[101] Michael Kerrisk. cgroups — linux manual page, 2020. URL https://man7.org/linux/man-pages/man7/cgroups.7.html. Online; accessed 31-january-2021.

[102] Gene Kim, Patrick Debois, John Willis, and Jez Humble. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016. ISBN 1942788002.

[103] Kubernetes. Kubernetes documentation, 2020. URL https://kubernetes.io/docs/home/. Online; accessed 9-january-2021.

[104] Kubernetes. kops - kubernetes operations, 2021. URL https://github.com/kubernetes/kops. Online; accessed 30-august-2021.

[105] Grafana Labs. Grafana, 2021. URL https://grafana.com/. Online; accessed 30-january-2021.

[106] Grafana Labs. Dashboard json model, 2021. URL https://grafana.com/docs/grafana/latest/dashboards/json-model/. Online; accessed 30-january-2021.

[107] Traefik Labs. Traefik, the cloud native application proxy, 2021. URL https://traefik.io/traefik/. Online; accessed 1-august-2021.

[108] Yishan Lin. Keynote - aligning principles to nomad 1.0, 2020. URL https://www.hashicorp.com/resources/keynote-aligning-principles-to-nomad-1-0. Online; accessed 25-january-2021.

[109] Redis Ltd. Redis, 2021. URL https://redis.io/. Online; accessed 11-october-2021.

[110] Mission Critical Magazine. Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space, November 2010. URL https://www.missioncriticalmagazine.com/articles/83726-gartner-says-efficient-data-center-design-can-lead-to-300-percent-capacity-growth-in-60-percent-less-space. Online; accessed 21-december-2020.

[111] Mattermost. Incoming webhooks, 2021. URL https://docs.mattermost.com/developer/webhooks-incoming.html. Online; accessed 30-january-2021.

[112] Scott McCarty. A practical introduction to container terminology, 2018. URL https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction. Online; accessed 9-january-2021.

[113] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical report, National Institute of Standards and Technology (NIST), September 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[114] Steve Mezak. The origins of devops: What's in a name?, 2018. URL https://devops.com/the-origins-of-devops-whats-in-a-name/. Online; accessed 12-january-2021.

[115] Microsoft. Azure resource manager, 2021. URL https://azure.microsoft.com/en-us/features/resource-manager/. Online; accessed 16-january-2021.

[116] Microsoft. What is azure load balancer?, 2021. URL https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview. Online; accessed 1-august-2021.

[117] Ron Miller. Four years after its release, kubernetes has come a long way, 2018. URL https://techcrunch.com/2018/06/06/four-years-after-release-of-kubernetes-1-0-it-has-come-long-way/?guccounter=1. Online; accessed 9-january-2021.

[118] Inc. MongoDB. Mongodb: The application data platform, 2021. URL https://www.mongodb.com/. Online; accessed 11-october-2021.

[119] Kief Morris. *Infrastructure as Code - Managing Servers in the Cloud*. O'Reilly, 2016.

[120] J. R. Neve, K. Godbole, and R. Neve. Productivity and process improvement using 'scaled agile' approaches: An emphasized analysis. In *2017 International Conference on Inventive Computing and Informatics (ICICI)*, pages 793–798, 2017. doi: 10.1109/ICICI.2017.8365245.

[121] NGINX. Http load balancing | nginx plus, 2021. URL https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/. Online; accessed 1-august-2021.

[122] Education Networks of America. Fabio load balancer, 2021. URL https://fabiolb.net/. Online; accessed 10-december-2021.

[123] The United States Department of Defense. Military handbook - configuration management guidance, 2001. URL https://www.acqnotes.com/Attachments/MIL-HDBK-61A%20(SE)Configuration%20Management%20Guidance.pdf. Online; accessed 23-december-2020.

[124] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *Stanford University*, 2014.

[125] OpenCensus. Opencensus, 2020. URL https://opencensus.io/. Online; accessed 25-november-2020.

[126] OpenStack. Cinder, 2021. URL https://wiki.openstack.org/wiki/Cinder. Online; accessed 17-january-2021.

[127] OpenStack. Open source cloud computing infrastructure - openstack, 2021. URL https://www.openstack.org/. Online; accessed 1-february-2021.

[128] Oracle. Cloud infrastructure | oracle, 2021. URL https://www.oracle.com/cloud/. Online; accessed 1-february-2021.

[129] Pallets. Welcome to flask - flask documentation, 2010. URL https://flask.palletsprojects.com/en/2.0.x/. Online; accessed 30-january-2021.

[130] AMA Agência para a Modernização Administrativa. A chave móvel digital, 2021. URL https://www.autenticacao.gov.pt/web/guest/a-chave-movel-digital. Online; accessed 30-january-2021.

[131] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Comput. Surv.*, 45(2), March 2013. ISSN 0360-0300. doi: 10.1145/2431211.2431216. URL https://doi.org/10.1145/2431211.2431216.

[132] Prometheus. Overview | prometheus, 2021. URL https://prometheus.io/docs/introduction/overview/. Online; accessed 30-january-2021.

[133] Prometheus. prometheus/pushgateway, 2021. URL https://github.com/prometheus/pushgateway. Online; accessed 30-january-2021.

[134] Open Source Puppet. Welcome to puppet 7.3.0, 2021. URL https://puppet.com/docs/puppet/7.3/puppet_index.html. Online; accessed 22-january-2021.

[135] Nor Azian Abdul Rahmana, Sariwati Mohd Sharifb, and Mashitah Mohamed Esac. Lean manufacturing case study with kanban system implementation. 2013.

[136] Jessie Reed. Physical servers vs. virtual machines: Key differences and similarities, 2018. URL https://www.nakivo.com/blog/physical-servers-vs-virtual-machines-key-differences-similarities/. Online; accessed 3-january-2021.

[137] Max Rehkopf. What is a kanban board?, 2020. URL https://www.atlassian.com/agile/kanban/boards. Online; accessed 14-january-2021.

[138] Keith Rogers. Black box vs. white box monitoring: What you need to know, 2018. URL https://devops.com/black-box-vs-white-box-monitoring-what-you-need-to-know/. Online; accessed 28-november-2020.

[139] Armin Ronacher. Welcome to jinja2, 2008. URL https://jinja2docs.readthedocs.io/en/stable/. Online; accessed 10-december-2021.

[140] John. Roum, Christopher. A study of the department of defense configuration management policies and procedures as applied to the fa-18 strike/fighter program., 1987. URL https://calhoun.nps.edu/handle/10945/22513.

[141] Scrum.org. What is scrum?, 2021. URL https://www.scrum.org/resources/what-is-scrum. Online; accessed 14-january-2021.

[142] Jordan Shamir. 5 benefits of virtualization, 2020. URL https://www.ibm.com/cloud/blog/5-benefits-of-virtualization. Online; accessed 2-january-2021.

[143] Slack. Sending messages using incoming webhooks, 2021. URL https://api.slack.com/messaging/webhooks. Online; accessed 30-january-2021.

[144] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.173. URL https://doi.org/10.1109/MC.2005.173.

[145] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273025. URL https://doi.org/10.1145/1272998.1273025.

[146] Ian Sommerville. *Software engineering*. Addison-Wesley, 9th edition, 2011.

[147] John S Stewart. How itil started, 2013. URL https://internationalbestpracticeinstitute.wordpress.com/2013/02/11/how-itil-started/. Online; accessed 20-february-2021.

[148] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. 2013.

[149] UpGuard Team. Service asset and configuration management, 2020. URL https://www.upguard.com/blog/5-configuration-management-boss. Online; accessed 15-december-2020.

[150] GMBH Travis CI. Travis ci - test and deploy with confidence, 2021. URL https://www.travis-ci.com/. Online; accessed 20-january-2021.

[151] M. Uddin, S. Islam, and A. Al-Nemrat. A dynamic access control model using authorising workflow and task-role-based access control. *IEEE Access*, 7:166676–166689, 2019. doi: 10.1109/ACCESS.2019.2947377.

[152] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[153] Don Wells. Integrate often, 1999. URL http://www.extremeprogramming.org/rules/integrateoften.html. Online; accessed 15-january-2021.

[154] Arch Wiki. Nfs, 2021. URL https://wiki.archlinux.org/index.php/NFS. Online; accessed 17-january-2021.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. Este trabalho é financiado por fundos nacionais através da FCT – Fundação para a Ciência e a Tecnologia, I.P., no âmbito do projeto UIDB/50014/2020.