

A DevOps approach to infrastructure on demand

CÉSAR JOSÉ COSTA PINHEIRO

Junho de 2022

A DevOps approach to infrastructure on demand

César Pinheiro

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

Supervisor: Dr. Nuno Bettencourt

Porto, June 30, 2022

Dedictory

I dedicate this work to everyone that has helped me throughout my academic journey, especially to my mother for her continuous support since the very beginning, to my girlfriend and her continuous offers of help to lessen my burdens and all my friends for making this journey so much more pleasant.

Abstract

As DevOps grows in importance in companies, there is an increasing interest in automating the process of building and deploying infrastructure, having as an objective reduce the complexity for non DevOps engineers and making it so that infrastructure is less error prone, which is not the case when doing it manually.

This work aims to explore how to build a solution that allows to manage infrastructure on demand while supporting specific services that are relevant for git profiles analysis, such as Sonarqube and Jenkins.

Firstly, this work starts by introducing its context, the problem that the solution is trying to solve and the methodology used to develop the solution.

On the State of the Art various topics are presented in order to give all the information needed to understand the implementation of the solution, including concepts such as DevOps and Automation, while going over specific technologies such as GraphQL, Docker, Terraform and Ansible.

A value analysis was also done to explore what are the main concerns for stakeholders when managing their infrastructure and to define the value of the solution being developed.

Lastly, the solution was implemented making use of various technologies and with scalability in mind that would allow it to grow in the amount of services supported with minimum changes.

The work is interesting for someone that is interested in DevOps, Infrastructure-as-Code and automation in general.

Keywords: DevOps, Infrastructure-as-Code, Automation, GraphQL, Terraform, Ansible

Resumo

Com o crescimento da importância de DevOps em empresas existe um interesse acrescido em automatizar o processo de construir e de dar deploy de infra-estrutura, tendo como objectivo reduzir a complexidade para engenheiros menos proficientes em DevOps, e construir infra-estrutura que é menos propensas a erros, o que não acontece quando feito manualmente.

Este trabalho visa implementar uma solução capaz de gerir infra-estrutura a pedido e ao mesmo tempo suportar serviços específicos relevantes para a análise de perfis git, como por exemplo Sonarqube e Jenkins.

Em primeiro lugar, este trabalho começa por introduzir o seu contexto, o problema que a solução está a tentar resolver e a metodologia utilizada para desenvolver a solução.

No estado da arte são apresentados vários tópicos com a finalidade de fornecer toda a informação necessária para compreender a implementação da solução, incluindo conceitos como DevOps e automação, são também exploradas tecnologias específicas como GraphQL, Docker, Terraform e Ansible.

Foi também feita uma análise de valor para explorar quais são as principais preocupações das partes interessadas na gestão das infra-estruturas das suas empresas e para definir o valor da solução que está a ser desenvolvida.

Finalmente, a solução foi implementada, recorrendo a várias tecnologias e tendo em mente a escalabilidade da solução que permitiria crescer na quantidade de serviços suportados requerendo alterações mínimas.

O trabalho é interessante para alguém que esteja interessado em DevOps, Infraestrutura como código e automatização em geral.

Keywords: DevOps, Infrastructure-as-Code, Automation, GraphQL, Terraform, Ansible

Contents

List of Figures	xiii
List of Tables	xv
List of Source Code	xvii
List of Acronyms	xix
1 Introduction	1
1.1 Problem	1
1.2 Objectives	2
1.3 Hypotheses	2
1.4 Work Plan and Methodology	2
1.5 Thesis Structure	3
2 State of the Art	5
2.1 DevOps Methodology	5
2.1.1 Pipelines	6
2.1.2 Infrastructure as Code	7
2.1.3 Automation	9
2.2 Virtualization of Applications	9
2.2.1 What is Docker	11
2.3 Software Design	12
2.3.1 REST and GraphQL	12
2.3.2 Design Principles	14
2.4 Value Analysis	17
2.4.1 New Concept Development Model	17
2.4.2 Analytic Hierarchy Process	18
2.5 Existing Solutions	19
2.5.1 Bamboo Server	19
2.5.2 Buddy	20
2.5.3 Harness	20
2.5.4 Octopus Deploy	20
2.5.5 Vagrant	21
2.5.6 Cycloid	21
2.6 Summary	22
3 Analysis	23
3.1 Requirement Analysis	23
3.2 Value Analysis and Proposition	24
3.2.1 New Concept Deployment	24

3.2.2	Perceived Value	25
3.2.3	Value Proposition	26
3.2.4	Analytic Hierarchy Process	26
3.3	Summary	29
4	Design	31
4.1	Domain	31
4.2	Architecture	32
4.3	Internal components	33
4.4	Requirements	33
4.4.1	Configuration Management	33
4.4.2	Deployment Management	36
4.4.3	Deployment Observation	38
4.5	Summary	39
5	Implementation	41
5.1	The Structure	41
5.2	The Querier	43
5.2.1	Models	44
5.2.2	Queries	44
5.3	The Server	45
5.3.1	Start Command	45
5.3.2	GraphQL API	47
5.3.3	Configuration Management	48
5.3.4	Deployment Management	50
5.4	The Observer	51
5.4.1	Start command	52
5.4.2	The Worker	52
5.5	Template files	54
5.6	Encapsulating the Services	55
5.7	Automation	57
5.8	Summary	58
6	Evaluation	59
6.1	Process	59
6.2	Ease of use	59
6.3	Deployment time	61
6.4	Deletion time	61
6.5	Summary	61
7	Conclusion	63
7.1	Summary	63
7.2	Achieved	63
7.3	Limitations and Future Work	63
7.4	Contributions	64
7.5	Final remarks	64
	Bibliography	65
	Appendixes	66

A	AHP Analysis	67
B	Deployment creation	69
C	Sonarqube template files	71
D	Jenkins template files	75

List of Figures

2.1	Example of a CI/CD pipeline, extracted from <i>CI/CD Pipeline</i> (2021)	6
2.2	How Docker uses containers, extracted from <i>What is a Container?</i> (2019)	11
2.3	The Innovation Process (Koen et al. 2002)	17
2.4	The NCD Model (Koen et al. 2002)	18
3.1	Value Proposition Canvas	26
3.2	Hierarchical Decision Tree - Determining Priorities.	27
4.1	Domain Model	32
4.2	Architecture Diagram	32
4.3	Internal components diagram	33
4.4	Read global configuration	34
4.5	Read use case configuration	34
4.6	Add use case configuration	34
4.7	Update use case configuration	35
4.8	Delete use case configuration	35
4.9	Read service configuration	35
4.10	Add service configuration	36
4.11	Update service configuration	36
4.12	Delete service configuration	36
4.13	Read all deployments	37
4.14	Read one deployment	37
4.15	Create a new deployment	38
4.16	Delete an existing deployment	38
4.17	Observe deployments	39
5.1	GraphQL Playground	47
6.1	GraphQL API add use case mutation	60
6.2	GraphQL API create deployment mutation	60
6.3	Logging statements in the terminal	61
6.4	Logging statements in the terminal	61

List of Tables

3.1	Non-Functional Requirements	24
3.2	Functional Requirements	24
3.3	Tool's benefits and sacrifices per stakeholder	26
3.4	Scale of importance	28
3.5	Criteria pairwise comparison	28
3.6	Priority Vector	28
3.7	Alternatives' composite priority	29

List of Source Code

5.1	Root command	41
5.2	Querier interface	43
5.3	Querier New function	44
5.4	Deployment Model	44
5.5	Preloading helper function	45
5.6	Deployment retrieval method	45
5.7	Server command	46
5.8	Terraform installation function	46
5.9	<i>gqlgen</i> configuration file	47
5.10	Create Deployment Resolver	48
5.11	Read Configuration	49
5.12	Add Use Case to Configuration	49
5.13	Update Use Case Configuration	49
5.14	Delete Use Case Configuration	50
5.15	Read all deployments	50
5.16	Delete deployment	51
5.17	Observer command	52
5.18	Observer worker Start method	52
5.19	Observer failure handler	53
5.20	Observer success handler	53
5.21	Observer worker method to update the instance	54
5.22	Server Dockerfile	55
5.23	Server start script	56
5.24	Database Dockerfile	56
5.25	docker-compose file	56
5.26	Makefile	57
7.1	Create deployment	69
7.2	main.tf	71
7.3	ansible.yaml	73
7.4	inventory	75
7.5	main.tf	75
7.6	ansible.yaml	77
7.7	settings.yaml	78
7.8	plugins.yaml	79
7.9	inventory	79

List of Acronyms

AHP	Analytic Hierarchy Process.
AI	Artificial Intelligence.
API	Application Programming Interface.
AWS	Amazon Web Services.
CD	Continuous Deployment.
CFM	Continuous Feedback & Monitoring.
CI/CD	Continuous Integration & Continuous Delivery.
CI	Continuous Integration.
CLI	Command Line Interface.
CP	Continuous Planning.
CRUD	Create Read Update Delete.
CR	Consistency Ratio.
CT	Continuous Testing.
DIP	Dependency Inversion Principle.
DTO	Data Transfer Object.
EC2	Elastic Compute Cloud.
FFE	Fuzzy Front End.
HCL	HashiCorp Configuration Language.
HTTP	Hypertext Transfer Protocol.
IDE	Integrated Development Environments.
IP	Internet Protocol.
ISP	Interface Segregation Principle.
IT	Information and Technology.
IaC	Infrastructure as Code.
LSP	Liskov Substitution Principle.
NCD	New Concept Development.
NPD	New Product Development.
OCP	Open-Closed Principle.
OOP	Object-Oriented Programming.
ORM	Object-Relational Mapping.
OS	Operating System.
QFD	Quality Function Deployment.
REST	Representational State Transfer.
SDK	Software Development Kit.
SD	Sequence Diagram.
SQL	Structured Query Language.
SRP	Single Responsibility Principle.
SSH	Secure Shell.
SaaS	Software as a Service.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.

URL Uniform Resource Locator.
VM Virtual Machine.

Chapter 1

Introduction

In a demanding and competitive market, where each year there are more solutions and more professionals in the Information and Technology (**IT**) area, it is important to find the best talent that can boost your product to the next level.

To accomplish this, companies have developed multiple strategies in order to determine whether a professional is or not a good fit technically for their company, some create algorithmic problems that require discipline and practice to solve, others require the candidate to create a small project in order to see how they actually do code development. What both methods have in common is that both require manual validation and are prone to inherent biases.

One thing is certain, in order to determine whether a profile matches the opportunity, there is a need to make sure that their technical skills match a certain criteria, and that the criteria does not change based on the interviewer. One way to do this would be the automation of the technical skill evaluation, where it is possible to determine a profile's technical skills (to a certain level) based on their repositories and contributions to projects. This would allow the retrieval of metrics that could prove to be crucial information on how that profile works and how their technical skills have evolved in a certain time-frame.

The capacity of a system to evaluate profiles based on repository analysis is something that would require a complex system which would need to behave differently depending on the load and number of repositories being analysed. This research focuses on how to make sure that such system has the right infrastructure that allows it to serve its users without losing any availability when the loads are bigger while not wasting resources when the loads are smaller. It is also important to determine how such a system should communicate with each of its internal services in order to make it perform well.

This chapter serves the purpose of providing context of what is being researched, the problem that is being solved, the objectives of the research being done as well as the hypotheses and the approach taken to accomplish the objectives that were established.

1.1 Problem

The internet is growing each passing day, having grown from in both users and websites in an astronomical level. In 2010 there were approximately 200 million live websites and 2 billion users, while in January of 2022 there are approximately 1.9 billion websites and 5.1 billion users (*Internet Live Stats* 2022). This growth brings opportunities but also new problems that need to be solved by software engineers.

One of the problems that this growth brings is bringing value to a large user base, especially when the value that is being provided requires a high processing power as is the case of repository analysis. A website tries to bring value to its users, and this value must be available to the users at all times.

With the usage of cloud solutions such as *Amazon Web Services (AWS)*, *Google Cloud* and *Azure* it is possible to solve the availability issues, but these services require extensive configuration in order to have a service up and running with the expected behaviour. So is it possible to have repository analysis tools available on demand? Taking into consideration this question a *DevOps* approach to managing infrastructure is to be researched and implemented.

1.2 Objectives

With this project it is intended the conception of a solution to managing a product's infrastructure. To accomplish this, a service is to be built that should be capable of:

- Orchestrating repository analysis tools
- Deploying repository analysis tools to the cloud
- Encapsulating repository analysis tools

1.3 Hypotheses

In order to determine whether the project was successfully completed it is required that hypothesis are set on the work. So with that in mind, the hypothesis are presented:

- Is the solution capable of deploying individual services (triggering deployments to the cloud as needed)?
- Is the solution capable of orchestrating individual services (serving instances of the services as needed)?
- Is the solution capable of encapsulating individual services (using existing solutions is the service capable of encapsulating the services in order to work with them)?

1.4 Work Plan and Methodology

A work plan and work methodology is important to a project's management, given that it describes the different phases of the project that allow the accomplishment of the different tasks. The current phases of the work plan and the methodology to accomplish it are the following:

- Conception - this phase includes tasks related to the problem, its contextualization and the state of the art.
 - Research and contextualize the problem - try to understand the relevance of the problem (Chapter 1).
 - Research about the DevOps methodology, automation and similar work that was done previously - study the current state of the art of these different fields (Chapter 2).

- Analysis - at this phase a study is done on how to approach the problem and both functional and non-functional requirements are defined and an analysis on the value of the solution being developed (Chapter 3).
- Design - this phase consists of defining the architecture of the solution and it includes the research and preparation of an approach to apply in the practical context 4.
- Development - the solution is developed based on the previous phase, along with its experimentation (Chapter 5).
- Evaluation - the solution is evaluated based on the expected results, meaning that if ways to improve the solution are found they should be registered as well as conclusions about the solution 6.
- Documentation - this phase includes the completion of the thesis as a way to share the knowledge gained throughout the development of the solution.

1.5 Thesis Structure

The thesis follows a set structure, and each chapter has a given purpose. This chapter, as mentioned previously, tries to provide context on what the problem being solved is and the objectives of the research being done and solution being implemented.

On Chapter 2, the state of the art is described, where the crucial concepts to the research are explored and presented. In this chapter the objective is to provide as much information as needed about the technical concepts that are necessary in order to fully understand the chapters that come next. Existing approaches to the problem are also presented.

On Chapter 3 consists of the analysis of the problem, where a value analysis is done as well as the requirements analysis (both functional and non-functional).

Chapter 4 goes over the design of the solution to the problem, making sure that each decision that is being done has a reason to be that way and document it. In this chapter there will also be diagrams in order to more easily visualize how the solution will look like at the end of its implementation.

On Chapter 5 the whole development process is documented in order to provide information on how the final solution was constructed. Images of the different stages of development and the state of the product will also be present in order for the progress to be apparent and easily seen.

Finally, for Chapter 6, the objective is to evaluate the implemented solution and validate whether it accomplishes all the established objectives. It will also document the experience of developing the solution, where it went well, where it went wrong, while also evaluating whether it fits the expected results that were documented on this chapter.

Chapter 2

State of the Art

This chapter provides a state of the art about the DevOps framework, Infrastructure-as-Code, the topic of software design is also briefly explored, as well as virtualization of software applications and automation.

Additionally, this chapter provides some insight on existing solutions, namely products for workflow automation and management like Bamboo Server and Vagrant, and full-on Software as a Service (**SaaS**) products such as Cycloid.

2.1 DevOps Methodology

In order to possess an advantage in cloud platforms, there is a need to design applications so that they are decoupled from physical resources. When considering decoupled architectures especially in cloud design for a platform as a service and infrastructure as a service, there is a need to perceive the efficiency of deployments, the deployment stages involved for an application, and the correct utilization of the underlying cloud resources, this way of thinking and optimization of cloud computing resources saves money to companies (Agrawal and Rawat 2019).

DevOps is a system of thinking with a primary concern for developing, deploying and operating high-quality software. If development, deployment and operation can be considered as a pipeline for code to go through, then DevOps tries to look at that pipeline from a holistic perspective. DevOps is a compound of development (Dev) and operations (Ops), resulting in a union of people, processes, and technologies to continually provide value to customers. A DevOps culture along with its practices and tools provides the ability for teams to better respond to customer needs, increase confidence in the applications that are built and makes business goals achievable faster.

A suitable methodology to implement DevOps is through the usage of Continuous Planning (**CP**), Continuous Integration (**CI**), Continuous Testing (**CT**), Continuous Deployment (**CD**), and Continuous Feedback & Monitoring (**CFM**). Each of these phases are not separated by boundaries, nor are limited in terms of only being able to start a given phase when another ends (Agrawal and Rawat 2019).

Continuous Planning consists of establishing a vision over what is the ultimate work goal of a project, defining a set of functionalities giving each iteration value, the criteria to be fulfilled and the end. Its continuous given that it should address changes and evolutions according to a continuous improvement process, based on Continuous Feedback & Monitoring.

Continuous Integration is the automation of processes such as review, validation, testing and alerting of the value that is built along the iterations of a project. This means that for each iteration of value delivered, in this case software, it should be automatically tested in order to ensure that it works well and then published in a service that integrates the value with the rest of the project, this phase also touches the Continuous Testing phase in which the value is being tested continuously based on different approaches such as unit tests, integration tests, functional tests, acceptance tests, quality analysis of the code, and regression tests (Arachchi and Perera 2018).

The deployment of software is a process that consists of multiple steps and the more steps there are in this process the more it is prone to human error. Promoting its automation through tools and scripts is the goal of Continuous Deployment, where the whole process is automated which looks to lessen the failure due to manual tasks executed by humans.

Continuous Feedback & Monitoring is a phase which should be permanent and be applied throughout the entire cycle of DevOps. This phase consists of monitoring, analysing and measuring all that displays the current status overview of the application and its infrastructure, including all its dependencies. This process should evolve according to the results, where over each iteration of feedback the project is adjusted accordingly (Arachchi and Perera 2018).

2.1.1 Pipelines

A DevOps pipeline is one of the most crucial parts of the DevOps process. This term is used to discuss the tools, processes, and automation frameworks used to build software artifacts, it can be thought as a sequence of events or jobs that can be executed to accomplish a given task. Jenkins is an example of an open-source automation software which allows its users to build their own pipelines (Arachchi and Perera 2018).

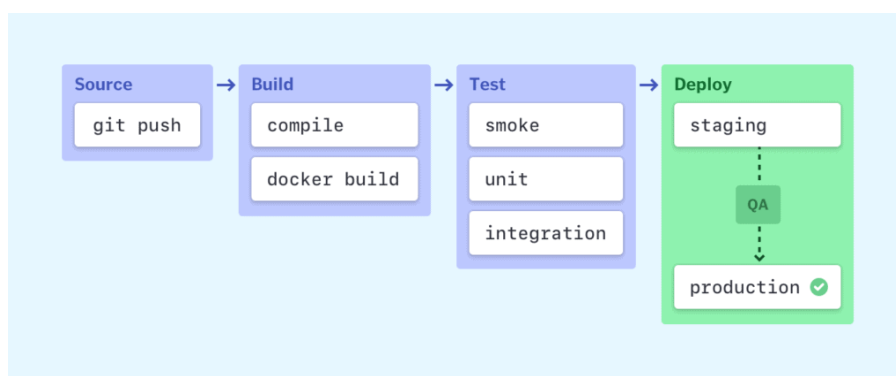


Figure 2.1: Example of a CI/CD pipeline, extracted from *CI/CD Pipeline* (2021)

Pipelines can be split into two different concepts, Stages and Steps. A pipeline is formed of multiple stages which in turn contain a series of steps. These stages are used to visualize the pipeline process, while each step inside a given stage represent a task that should be executed by the pipeline. An example of such a pipeline can be seen on Figure 2.2 (Arachchi and Perera 2018).

2.1.2 Infrastructure as Code

Infrastructure design can be defined as the software life-cycle phase which defines and configures the software infrastructure that is required for a given software. Infrastructure design typically entails a tiring manual process of installation and configuration scripts needed to, among others:

- Instantiate and link the required machines for the software to run;
- Install and configure the required software and middleware for a Virtual Machine (**VM**);
- Instantiate and run the needed services for the software to be operated.

Virtualization, cloud, containers, and server automation should simplify **IT** operations work. There should be less time and effort spent provisioning, configuring, updating and maintaining services. Problems need to be quickly detected and resolved, and systems should be consistently configured and up to date. DevOps promotes the use of a typical software development notion, it promotes the usage of *source code* for infrastructure design and management as well. This means that the entire set of scripts, automation and configuration code can be expressed using the same standard language, this practice is called *infrastructure-as-code*. The purpose of Infrastructure as Code (**IaC**) is to re-use successful and common software development practices to speed up software operations (Artac et al. 2017; Morris 2016).

Through the usage of Infrastructure as Code teams expect to achieve the following (Morris 2016):

- Infrastructure supports and enables change, rather than being an obstacle;
- Changes to the systems are routine;
- **IT** staff spends their time on valuable tasks instead of repetitive ones;
- Teams are able to recover from failures quickly and easily;
- Improvements are made continuously;
- Solutions to problems are proven through implementation and experimentation.

Terraform

Terraform is an open source **IaC** tool created by HashiCorp and written in the Go programming language to provision infrastructure. This tool can be used to deploy infrastructure from a laptop, a server or any computer at all. Terraform enables developers to use a high-level configuration language called HashiCorp Configuration Language (**HCL**) to describe the desired end state of the cloud for running an application. Because Terraform uses a simple syntax, it can provision infrastructure across multiple cloud and on-premise data centers, and can efficiently re-provision infrastructure in response to configuration changes (Brikman 2019).

Under the hood, Terraform makes Application Programming Interface (**API**) calls on the behalf of the user to one or more providers such as **AWS**, Azure, Google Cloud, DigitalOcean, etc. That means that Terraform gets to leverage the infrastructure those providers are already running for their **API** servers, as well as the authentication mechanisms that were already being used with those providers by the user, such as API keys (Brikman 2019).

Terraform uses a declarative approach to provisioning servers, meaning since its configuration declares the wanted end state, Terraform figures out how to get that end state, Terraform will also be aware of any state it created in the past. Therefore, if the configuration previously deployed 10 servers and the requirements changed and the end state should be 15 servers, this can be changed in the configuration and Terraform will figure out it only needs to deploy 5 more servers (Brikman 2019).

Terraform has a few core concepts, namely:

- **Variables:** key-value pairs used by Terraform modules to allow customization.
- **Module:** a folder with Terraform templates where the configurations are defined.
- **Provider:** a plugin used to interact with **APIs** of services and access their related resources.
- **Resources:** refers to a block of one or more infrastructure objects (e.g. compute instances, virtual networks, etc.) which are used in configuring and managing the infrastructure.
- **State:** consists of cached information about the infrastructure managed by Terraform and its related configurations
- **Data Source:** is implemented by providers to return information on external objects to Terraform.
- **Output Values:** are return values of a Terraform module that can be used by other configurations.
- **Plan:** it is one of the stages where Terraform determines what needs to be created, updated or destroyed to move from the current state to the end state.
- **Apply:** it is one of the stages where Terraform applies the changes in the current state of the infrastructure in order to move to the desired end state.
- **Destroy:** it is one of the stages where Terraform destroys all remote objects managed by a particular Terraform configuration.

Ansible

Ansible is an open source automation tool that enables the automation of provisioning, configuration management, application deployment and orchestration processes. Unlike more simplistic management tools, Ansible can be used to automate the installation of software, daily tasks, provisioning of infrastructure, patching of systems and enable the sharing of automation across organizations (Hochstein and Moser 2017).

Ansible works by connecting to what is to be automated and pushing programs that execute instructions that would have to be done manually. These programs utilize Ansible modules which are executed over Secure Shell (**SSH**) by default, and once complete are removed.

Ansible has a few core concepts, namely:

- **Control node:** the machine from which the ansible Command Line Interface (**CLI**) tools (ansible-playbook, ansible, ansible-vault) are run.
- **Managed nodes:** also referred to as the "hosts", these are the target devices that will be managed by Ansible.

- **Inventory:** a list of managed nodes provided by one or more "inventory sources". The inventory can specify information specific to each node, like Internet Protocol (**IP**) addresses. It is also used for assigning groups, that allow for bulk variable assignment.
- **Playbooks:** contain Plays, which are the basic unit of Ansible execution and are written in YAML.
- **Plays:** this object maps hosts to tasks. The Play contains variables, roles and an ordered list of tasks and can be run repeatedly.
- **Roles:** they are a limited distribution of reusable ansible content such as tasks, handlers, variables, plugins, templates and files.
- **Tasks:** the definition of "actions" to be applied to the managed host.
- **Handlers:** a special variation of a Task that can only execute when notified by a previous task which resulted in a "changed" status.
- **Modules:** the code or binaries that Ansible copies and executes on each managed node to accomplish the action defined in each task.
- **Plugins:** there are pieces of code that expand Ansible's core capabilities.

2.1.3 Automation

Automation is used as a term to describe technological applications where the human input is minimized as much as possible. These applications include business process automation, **IT** automation, personal applications such as home automation and more. The usage of sets of processes that are repeated increase the productivity and efficiency in software engineering while also reducing human errors (*What is automation?* | IBM 2021).

The usage of automation allows developers, operators, testers and other stakeholder in DevOps to automate the tasks performed in the creation and deployment of software. Activities such as integration, testing, building and delivering software can be automated in order to reduce delays and risks given that such tasks are time-consuming and error-prone if done manually.

Delivering new releases can be complicated work for many applications, it could mean setting up and configuring web servers, or fixing errors so that a given release can run properly. Such activities are hard to accomplish manually and can lead to errors given the manual aspect of that process leading to delays and additional expenses. These problems primarily affect the operations team, possibly creating an increasing tension between operations and development teams. For these reasons, DevOps relies on automation heavily. One of the core concepts in DevOps automation are deployment pipelines, which represent the process of getting software under development from version control to the production environment. DevOps optimizes this process by automating every step of it, avoiding delays during the development process and achieving continuous practices, such as Continuous Delivery, Continuous Integration, and Continuous Monitoring (Mohammad 2018).

2.2 Virtualization of Applications

Everything starts and ends with hardware, in order to run an application, there is the need for some real hardware. This hardware includes actual physical machines, with certain computing

capabilities, memory, and local persistent storage. In addition to hardware there is the need for some shared persistent storage and to hook up all the machines by setting up the networking in order for them to find and talk with each other.

The virtualization of applications is a process that deceived the application into believing that it interfaces directly with an operating system's capacities, when it does not. To achieve this it is required for a virtualization layer to be inserted between the application and the Operating System (**OS**). This layer, or framework, must run the application's subsets virtually and without impacting the subjacent **OS**. This virtualization layer replaces a portion of the run-time environment typically supplied by the **OS**, diverting files and registry log changes to a single executable file. By diverting the processes into one file instead of many, the application easily operates on a different device, and incompatible applications can now run adjacently (Portnoy 2012).

Desktop virtualization is also used in conjunction with application virtualization, it consists of the abstraction of the physical desktop environment and its related software from the end-user device that accesses it.

There are two main ways to virtualize applications: through the usage of Virtual Machines, and through the usage of Containers.

A **VM** is software that replicates the functions of a computer. It can be seen as a computer created by another computer. **VMs** can execute applications and programs without the need to use physical hardware, they are isolated from the rest of the machines that hosts it and behaves as if it is the only operating system on it (Portnoy 2012).

A container is a lightweight virtualization architecture that allows the deployment of individual applications inside portable and isolated environments. By isolating the application from the external host environment, containers enable frictionless application deployment. A single container might be used to run anything from a small micro-service or process to a larger application. Inside a container all the necessary executables, binary code, libraries, and configuration files are stored, these are stored in common ways so that they can be run anywhere, whether it be on a desktop or the cloud (Portnoy 2012).

Compared to the server or virtual machine approaches, containers do not contain operating system images. This makes them more lightweight and portable, with significantly less overhead. In larger application deployments, multiple containers may be deployed as one or more container clusters. Such clusters might be managed by a container orchestrator.

Examples of such container orchestration tools are:

- Kubernetes by Google;
- Fleet by CoreOS;
- Apache Mesos by The Apache Software Foundation;
- Helios by Spotify;
- Centurion by New Relic.

2.2.1 What is Docker

Docker is a project created by a team at *Docker, Inc* (formerly dotCloud Inc). This project resulted in the development of an open-source engine that automated the deployment of applications into containers.

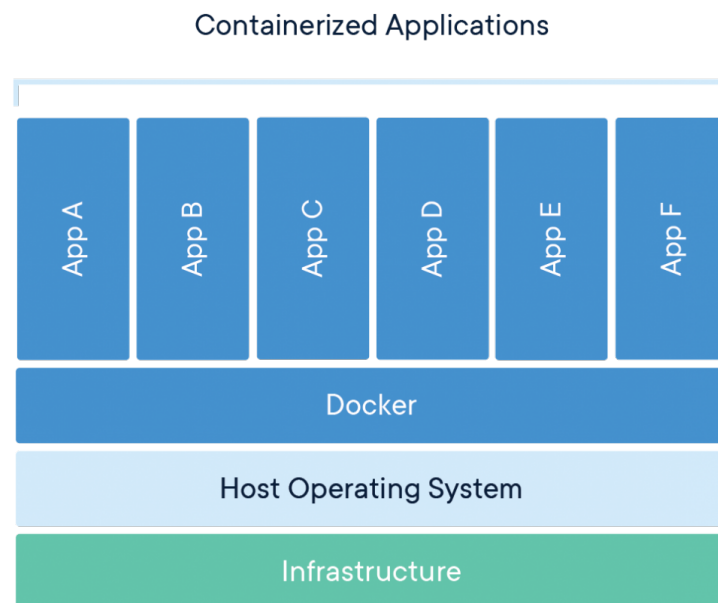


Figure 2.2: How Docker uses containers, extracted from *What is a Container?* (2019)

Docker makes use of virtualized container execution environments by adding an application deployment engine on top of it (Figure 2.2). It was designed to provide a lightweight and fast environment in which anyone is able to run their code as well as an efficient workflow to get the code from any computer to a test environment and then into a production environment (Turnbull 2014).

Docker proposes the following:

- An easy and lightweight way to model reality;
- A logical segregation of duties - developers care about making their applications run in the containers and operations cares about managing the containers;
- Fast and efficient development life cycle;
- Encourages service orientated architecture.

Docker is composed of the following components:

- Engine - a client-server application where the client talks with the Docker daemon which does all the work;

- Images - Images are the "build" part of Docker's life cycle and can be considered as the "source code" of the containers, they are highly portable and can be shared, stored, and updated;
- Registries - There are two types of registries: public and private; *Docker, Inc.* operates the public registry for images, called the Docker Hub, which contains images that other people have built and shared
- Containers - containers allow the packaging of applications and services, they are launched from images and can contain one or more running processes.

2.3 Software Design

By software design it is possible to understand the process, as well as all the documents resulting from the process, of turning requirements into executable code.

Software architecture design can be understood as the development process of going from existing requirements and possibly some already designed components to the software architecture — producing all appropriate architecture documentation.

For assessing user requirements, a document with all requirements is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages. Software design is the first step in the software design life cycle, which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements that were registered in the requirements document (R. Martin 2018).

Software design yields three levels of results:

- Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain;
- High-level Design - The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other;
- Detailed Design - Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

2.3.1 REST and GraphQL

Both Representational State Transfer (**REST**) and GraphQL are strategies that aim to agilize the process of data exchange through the network. Data exchange can be understood as the process of taking data structured under a source schema and transforming it into a target schema, so that the target data is an accurate representation of the source data, allowing this data to be shared between different systems.

Web services are purpose-built web servers that support the needs of a given site or application. Clients make use of **APIs** to communicate with web services. Both strategies can be used to facilitate and define patterns for this communication.

REST

REST is an acronym for *Representational State Transfer*. It is an architectural style devised with the premise that developers use the standard Hypertext Transfer Protocol (**HTTP**) methods (GET, POST, PUT and DELETE) to query and mutate resources represented by a Uniform Resource Identifier (**URI**) on the Internet.

A resource can be thought as a big dataset that describes a collection of entities of a given type. **REST** is agnostic in terms of the format used to structure the response data from a resource. JSON is the most popular data format, even though there is the possibility to use other data formats such as XML and CSV (Masse 2011).

Web Services which **APIs** are designed based on the REST architectural style are referred to as **REST** gIsAPIs. **REST APIs**, to be considered RESTful, must comply with a set of constraints, those being:

- Must have a client-server architecture.
- Must have a uniform interface.
- Must be stateless, meaning that no session information is retained by the receiver.
- Must have cacheability, meaning that responses must define themselves as either cacheable or non-cacheable.
- Must be a layered system, meaning that the client can't tell whether it is connected directly to the end server or an intermediary.

REST APIs provide the following advantages:

- Is easy to understand and learn, due to its simplicity and broad use.
- Eases the organization of complicated applications.
- High loads can be managed with the help of **HTTP** proxies and cache.
- Usage of standard **HTTP** verbs to retrieve data and make requests.
- Brings flexibility by not being bound by a given data format.

Although there are benefits there are also draw-backs to the usage of the **REST** architectural style to design **APIs**:

- Requires more effort to implement as the number of parameters increases.
- Lack of state, since most applications require stateful mechanisms it burdens the client with maintaining the state.
- Lack of security, since **REST** does not impose security.

GraphQL

GraphQL is a technology that originated from Facebook (more recently known as Meta) and that is now open-source. GraphQL is a query language for **APIs**, it is also a runtime for

fulfilling queries. Given that GraphQL is a specification, it is inherently language agnostic, resulting in implementation of this specification in various languages.

The underlying mechanism for executing queries and mutations is the POST **HTTP** verb. This means that GraphQL clients written in different languages can communicate between each other. As the name implies, GraphQL is intended to represent data in a graph. A graph is defined according to a schema language that is particular to GraphQL, developers use the schema language to define the types as well as the query and mutation operations that will be published by a GraphQL **API** (Porcello and Banks 2018).

Although there are no constraints when building a GraphQL **API**, there are some principles/guidelines that should be taken in consideration when building its services, those being:

- Should be hierarchical.
- Should be product centric.
- Should have strong typing.
- Should provide client-specified queries.

GraphQL **APIs** provide the following advantages:

- Clients are able to dictate exactly what they need from the server.
- Requires less effort to implement **API** queries.
- Clients are able to retrieve multiple resources in a single request.
- It is strongly typed, which allows clients to know exactly what data is available and in what form it exists.

There are also draw-back to the usage of GraphQL **APIs**, those being:

- Queries always return an **HTTP** status code of 200, regardless of its success or failure.
- Lack of built-in caching support.
- Unnecessary complexity for projects where the data is relatively consistent over time.

2.3.2 Design Principles

Design principles are widely applicable laws, guidelines, biases and design considerations which are applied with discretion and result from the accumulated knowledge and experience from various professionals. Design principles in software are the result of the accumulated knowledge of established engineers in the field and help making the software being designed more maintainable and scalable.

In the world of Object-Oriented Programming (**OOP**), there are many design guidelines, patterns or principles. Five of these principles are usually grouped together and are known as **SOLID**. While each of these five principles describes something specific, they overlap as well. In general, SOLID helps manage code complexity. It leads to more maintainable and extensible code (R. C. Martin 2002).

The design principles that will be described try to make it so that software does not become hard to maintain. Some of the symptoms of rotting design are:

- **Rigidity**: implementing a small change is difficult since it is likely to translate into a cascade of changes.
- **Fragility**: any change tends to break the software in many places, even in areas not conceptually related to the change.
- **Immobility**: it is not possible to reuse modules from other projects or within the same project since these modules contain too many dependencies.
- **Viscosity**: when changes are needed, developers will prefer the easier route even if it breaks existing design.

Single Responsibility Principle

The Single Responsibility Principle (**SRP**) states that a class should have one and only one reason to change, meaning that a class should have only one job. This can also be understood as cohesion and is crucial for a scalable software solution.

It is important to separate responsibilities since each responsibility is an axis of change. When the requirements change, that change will manifest through a change in responsibility amongst classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change, creating coupling. This can impair or inhibit the ability of the class to meet all responsibilities. This kind of coupling leads to fragile designs that break in unexpected ways when changed (R. C. Martin 2002).

In the context of the **SRP**, a responsibility is "a reason for change", if more than one motive for changing a class arises, then that class has more than one responsibility.

Open-Closed Principle

The Open-Closed Principle (**OCP**) states that objects or entities should be open for extension but closed for modification, meaning a class should be extensible without modifying the class itself.

Following the **OCP** means that the behaviour of the modules can be extended as the requirements of the application change to satisfy those changes and the extension of its behaviours does not result in changes to the source of the module itself, meaning it remains untouched. This can be done through the usage of abstraction, by abstracting the implementation details from a class it is possible to make it open for extension since it won't be concerned with the implementation details but with the behaviour it expects from the abstraction (R. C. Martin 2002).

Conformance to this principle is what yields the greatest benefits claimed for object oriented technology. Although it is not a good idea to apply rampant abstraction to every part of the application. Rather, it requires dedication from the developers to apply abstraction only to the parts of the program that exhibit frequent change.

Liskov Substitution Principle

The Liskov Substitution Principle (**LSP**) states that subtypes must be substitutable for their base types, meaning that every class or derived class should be substitutable for either their base or parent class.

The importance of this principle is obvious when considering the consequences of violating it. In the presence of a function f that takes, as its argument, reference to some base class B , if there is a derivative D of B which, when passed to f causes f to misbehave, then D violates the **LSP**. Then D is Fragile in the presence of f . If changes are done to f in order to accommodate D then f now violates the **OCP** because it is not closed to all the various derivatives of B (R. C. Martin 2002).

When **OCP** is in effect, applications are more maintainable, reusable and robust. The **LSP** is one of the prime enablers of the **OCP**, it is what allows a module, expressed in terms of a base type, to be extensible without modification. That substitutability must be something that developers can depend on implicitly. The contract of the base type has to be well understood if not explicitly enforced by the code.

Interface Segregation Principle

The Interface Segregation Principle (**ISP**) states that a client should never be forced to implement an interface that it does not use, or client should not be forced to depend on methods that they do not use.

This principle aims to deal with the disadvantages of complex interfaces. Classes that have complex interfaces are classes whose interfaces are not cohesive. Meaning the interfaces can be broken up into groups of methods where each group serves a different set of clients. The **ISP** acknowledges the existence of objects that require non cohesive interfaces, however it suggests that clients should not know about them as a single class, only being aware of the base classes that have cohesive interfaces (R. C. Martin 2002).

Complex classes cause bizarre and harmful couplings between clients. When one client forces a change on a complex class, all other clients are affected. Thus, clients should only have to depend on method that they actually call. This can be achieved by splitting a complex interface into many client-specific interfaces.

Dependency Inversion Principle

The Dependency Inversion Principle (**DIP**) states that entities must depend on abstractions, not on concretions. Also stating that the high level module must not depend on the low level module, but they should depend on abstractions. This principle is an enabler of decoupling.

There are implications when high-level modules depend on the lower level modules, changes to those lower level modules can have direct effect on the higher level modules and can force them to change. It should be the other way around, meaning that the high-level modules should be the ones influencing the low-level modules. The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details (R. C. Martin 2002).

The high level modules are the ones that should be reusable, meaning that these should depend on abstractions instead of concrete implementations of these low-level modules, since by depending on the low-level modules it becomes very difficult to reuse those high-level modules. The **DIP** makes it so that the dependency structure is inverted such that both details and policies (high-level modules) depend on abstractions.

2.4 Value Analysis

Value analysis is a systematic application of established techniques to identify the functions of a product or component and to provide the desired functions at the lowest total cost. It is a creative approach to eliminate unnecessary costs which add neither to quality nor to the appearance of a given product.

2.4.1 New Concept Development Model

The innovation process (figure 2.3) is the process of coming up with an idea and develop, test, and commercialize it.

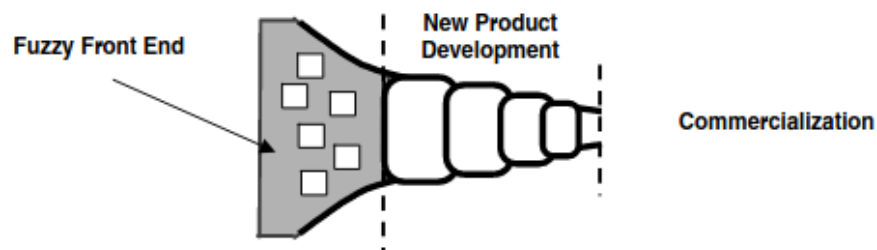


Figure 2.3: The Innovation Process (Koen et al. 2002)

The "Fuzzy Front End" is the first stage of the innovation process, and is the part where the opportunities are identified, analyzed, and validated, and the concept is developed, prior to entering the product development phase.

The New Concept Development (**NCD**) Model, defined by Peter Koen (Koen et al. 2002), is a model defined with common language and terminology which aims to help optimize activities in the Fuzzy Front End (**FFE**), resulting in a higher number of profitable concepts entering the New Product Development (**NPD**).

The **NCD** model is a nonlinear process and consists of three parts which are represented in figure 2.4:

- the uncontrollable influencing factors.
- the controllable engine that drives the activities in the **FFE**.
- the five activity elements of the **NCD**.

The **NCD** model is represented as a circular shape because the ideas are expected to flow, circulate, and iterate between and among all the five activity elements.

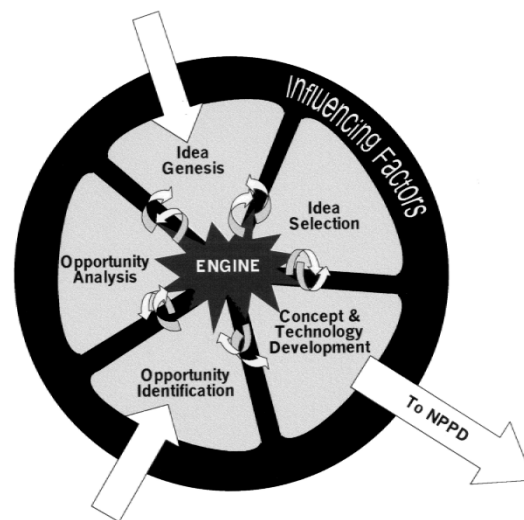


Figure 2.4: The NCD Model (Koen et al. 2002)

The influencing factors are the factors that affect the entire innovation process and influence the concept and its viability, namely the organization capabilities, the outside world and the enabling sciences and technologies.

The engine represents the factors that drive the five key elements that are controllable by the corporation, for example the leadership, culture, and business strategy of the organization.

Lastly, there is the inner spoke area which defines the five key elements that are controllable by the organization, namely opportunity identification, opportunity analysis, idea generation and enrichment, idea selection and concept definition.

2.4.2 Analytic Hierarchy Process

The Analytic Hierarchy Process (**AHP**) is a multi-criteria decision method, developed by Thoma L. Saaty in 1980. AHP represents an accurate approach to quantifying the weights of decision criteria, and can be used with qualitative, as well as quantitative criteria. **AHP** aims to divide the problem in hierarchic decision levels, facilitating its comprehension.

The first step of the **AHP** method is to build the hierarchic decision tree, with three levels representing the problem, the criteria, and the alternatives, respectively.

The second step is the establishment of priorities between the elements of each hierarchy level, using a comparison matrix, using a given scale that should be defined.

The next step consists of obtaining the relative priority of each criteria. This is done by normalizing the comparison matrix, and then the priority vector should be obtained by calculating the arithmetical average of the values in each line of the normalized matrix.

With the results obtained in the previous step it is necessary to evaluate the consistency of the relative priorities, which is done by calculating the consistency ratio. This is done with the aim of measuring how consistent the judgments are in relation to large samples of completely random judgements. These judgements are based on the assumption that the decision maker is rational, so that if A is preferred to B, and B is preferred to C, then A is preferred to C.

After creating a consistent criteria comparison matrix the same process must be done on a comparison table for each criteria, considering the selected alternatives. This is done until all the criteria's comparison matrix are consistent.

The last step of the Analytical Hierarchy Process is to obtain the alternatives' priority composite. This can be done by multiplying the priority vector of each criteria comparison matrix with the criteria's relative priority. The alternative with the highest relative priority represents the **AHP**'s final result.

2.5 Existing Solutions

The automation of the whole DevOps workflow to the end-user, making it as simple as possible to build, test and deploy a given software product is something important for smaller organizations that cannot have access to DevOps engineers.

There are currently multiple software solutions that permit the automation of these workflows. The different stages of the DevOps workflow were mentioned previously and they are the following:

- Continuous development
- Continuous integration
- Continuous testing
- Continuous monitoring and feedback
- Continuous delivery
- Continuous deployment

2.5.1 Bamboo Server

Bamboo Server helps DevOps and Continuous Integration & Continuous Delivery (**CI/CD**) teams streamline software development and delivery using the power of automation, integrations, and workflow management. It is part of the Atlassian ecosystem of products and can be used in conjunction with code pipelines in Bitbucket. It also has a wide range of automated tasks for build, test, and deployment use cases (*Bamboo Server* 2021).

It includes features such as:

- Can be installed in local servers but can require some set-up effort;
- Integration with all Atlassian products natively and third-party apps such as **AWS**, Docker and CodeDeploy at various DevOps stages;
- Work with all major coding languages and building platforms, being available on Mac, Windows, as tarball files and as a ZIP archive;
- Provide free support for the first 12 months while also benefiting from a community of more than 3 million global Atlassian users;
- Being suitable for large deployments up to 1000 remote agents and unlimited local agents.

2.5.2 Buddy

Buddy is a DevOps automation tool primarily meant for **CI/CD** workflows. Buddy has a library of more than 150 automated tasks and actions ready for use, and combines user experience with performance into one solution for teams that want to introduce **CI/CD** and accelerate the development lifecycle of the software being developed (*Buddy* 2021).

It includes features such as:

- Can be deployed on a **SaaS** model with free set-up for up to 5 projects and 120 pipeline runs per month;
- Offer a library of integrations covering all major cloud providers, developer tools, and collaboration platforms;
- Support deployments for most programming languages. It can also be used for application development on **AWS**, Google, Azure, Kubernetes, and others;
- A growing community of users along with robust learning materials;
- Sufficient scalability to support deployments of every size.

2.5.3 Harness

Harness is an end-to-end software delivery platform that enables automation at key stages of the DevOps lifecycle. Not only does it aid in the continuous integration and continuous delivery processes, but it also manages cloud costs and helps automate feature delivery, making use of Artificial Intelligence (**AI**) to simplify these processes (*Harness* 2021).

It includes features such as:

- The possibility to benefit from use-case-specific deployment, with on-premise and **SaaS** options as needed;
- Integration with all major development environments, with support for infrastructure provisioning scripts;
- Communities for Harness made available on Slack and online;
- Feature Flags, which uses machine learning and automation to speed up feature pipelines, enable standardization, and reduce the risk for each feature release;
- Adaptation to enterprise needs and use cases such as frequent releases, **CI/CD**, and cloud cost management.

2.5.4 Octopus Deploy

Octopus Deploy applies automated workflows to streamline even the most complex software deployments. It uses automated deployment tools for developers and releases managers and automated runbooks to reduce efforts for operations teams (*Octopus Deploy* 2021).

It offers features such as:

- Can be quickly deployed as a service for up to 5000 deployment targets, and it supports unlimited deployments on local servers;

- Native connection with major **CI** servers, operations tools, container platforms, etc., and it is possible to integrate custom applications through **APIs**;
- Support for all major development environments like Java, .NET, Node.js, Python, and others;
- Slack communities and discussion forums with active participation from users;
- A dedicated offering called Octopus Deploy for Enterprise which connects multiple teams, platforms, and software releases across the organization;
- Ease of use and automation prowess, it is possible to quickly get tutorial guidelines for all major languages, build servers, and package repositories.

2.5.5 Vagrant

Vagrant by HashiCorp helps the deployment of standardized workflows irrespective of dev, ops, design, or any other role. It makes production processes more consistent by driving reusability for packages and configurations. Vagrant addresses a singular use case, which it solves very effectively. It is possible to automate production workflows across environments to reduce inconsistency as well as scripting efforts (*Vagrant* 2021).

It includes features such as:

- A quickstart installation for all major operating systems and VMware virtual machines;
- Integration with configuration management systems such as Ansible, Chef, Docker, Puppet, and Salt to enable consistency across production pipelines;
- Availability on macOS, Windows, Linux, Debian, Centos, and ArchLinux, along with popular code editors, Integrated Development Environments (**IDE**), and browsers;
- A large user community, thanks to the HashiCorp forum. There are user groups, event organizers, and the dedicated HashiCorp discuss the platform;
- Being ready for deployment at scale.

2.5.6 Cycloid

Cycloid is an open DevOps and hybrid cloud collaboration platform. It's an enterprise-grade tool that helps teams upskill and work together, regardless of skill set, expertise, or technology. It attempts to bring all necessary tools together, providing end-to-end project frameworks that will help guide and facilitate a DevOps revolution in an organization (*Hybrid Cloud DevOps / Cycloid* 2021).

It includes features such as:

- Can be deployed on a **SaaS** model;
- Integration with all major cloud providers such as **AWS**, Google, Azure, and others;
- A **CLI** and an **API** for developers to use across the organization to interact with the service and integrate in their own tools;
- Tools to visualize the product's infrastructure and its cost management;
- A growing library of documentation and material to get started with the service.

2.6 Summary

On this chapter an overview of different theoretical topics was synthesized which will prove to be important to the understanding of the following chapters.

The DevOps methodology was studied, along with concepts such as Pipelines, Infrastructure as Code and Automation. This section also goes over specific technologies used in the practice of Infrastructure as Code such as Terraform and Ansible, providing some information on how these work.

A section dedicated to the Virtualization of Applications is also present, providing information about the concept of virtualization through different methods such as virtual machines and containers. In this section an overview of what is Docker is also done along with its benefits and core concepts.

On the Software Design section, it is provided a summary of what consists software design, strategies used to agilize data exchange and some design principles that should be taken in consideration when building solutions. The strategies mentioned in this section are **REST** and **GraphQL**, and each one of these is described along with their constraints, advantages and disadvantages. The design principles mentioned are the ones under the SOLID acronym, which provide value in building maintainable and clean solutions.

The Value Analysis section provides information regarding the **NCD** model and the **AHP** which are used to determine the value of the solution being developed and the priority when developing the solution.

Finally, there is the section containing existing solutions, where an overview of the different solutions that achieve something similar is done. On this section solutions such as Harness and Vagrant are covered, containing information on what these solutions are capable of.

Chapter 3

Analysis

The development of an infrastructure on demand deployment tool is an attempt at providing a way for people to easily deploy repository analysis to the Cloud with minimum configuration. The current reality is that DevOps is a complex area which requires attention on different topics such as availability, performance and reliability. These topics are affected by how the services are deployed, where they are deployed, if they are scaling based on the load of requests or not, etc.

Tasks like processing of large quantities of data can affect these services, which means that strategies should be developed to address difficulties that arise. The tool developed attempts to solve by default these kinds of problems while allowing its users to customize which tool to deploy and what functionalities to include depending on their use case.

This section is split into two smaller sections, one focused on the analysis of requirements for the solution being developed, and one for the analysis of the value that is envisioned for the solution. The requirements analysis section tries to identify all the functional and non-functional requirements that the solution developed must satisfy. The value analysis section makes use of the New Concept Development Model (**NCD**) to identify and analyze the opportunity in the market, with this information the perceived value of the solution is documented. This section also includes a value proposition of the solution as well as a value analysis using the Quality Function Deployment (**QFD**) method.

3.1 Requirement Analysis

Both functional and non-functional requirements were collected in order to better understand the expectations about the final desired service. The expectations in terms of non-functional requirements for the infrastructure on demand deployment tool being developed are presented in the table 3.1 while the functional requirements are presented in the table 3.2.

Identification	Non-Functional Requirement
NFR1	Reliability (Availability): the tool developed must work in a stable way regardless of the conditions that may affect it
NFR2	Supportability (Flexibility): the tool should be flexible , meaning it should support different environments as much as possible
NFR4	Performance (Speed): the tool should provide fast deployments, meaning it should not take more than 20 minutes to take a deployment request and have it up and running
NFR5	Usability: the tool developed should be easy to interact with, meaning it shouldn't have any complications, and should have a simple and friendly user interface to make the interaction easier
NFR6	Portability: the tool should be accessible from any machine or system
NFR7	Security: the tool should be secure against attacks

Table 3.1: Non-Functional Requirements

Identification	Functional Requirement
FR1	Configuration: the tool should allow the management its global configuration
FR2	Deployment: the tool should allow users to create deployments
FR3	Observing: the tool should observe the deployments done and notify the end user

Table 3.2: Functional Requirements

3.2 Value Analysis and Proposition

In this section there is a focus on the value that is envisioned for the solution that will be developed, using the **NCD** Model, the opportunity will be identified and analyzed. Once the opportunity is identified and analyzed, the perceived value, the value proposition and its analyzation will be defined using the Analytic Hierarchy Process (**AHP**) method.

3.2.1 New Concept Deployment

The innovation process is the process of coming up with an idea and develop, test, and commercialize it. The New Concept Development Model aims to help optimize activities in the Fuzzy Front End, hopefully resulting in a higher number of profitable concepts entering the New Product Development.

Opportunity Identification

Since infrastructure is something that is becoming increasingly important, it makes sense to make its management easier, faster and more efficient, as well as less error and bug prone. One solution would be the use of a management tool that would handle all of it with minimum configuration, such as the tool being developed.

By creating a solution for infrastructure management there would be several benefits that could be achieved, for instance:

- Reducing the need for as many DevOps engineers in a company.
- Easier process to create and manage deployments.
- Flexibility to migrate infrastructure from one Cloud Provider to another with minimum effort.
- Developers can focus on development instead of on infrastructure.

Opportunity Analysis

Although an infrastructure management tool may be the possible solution to making infrastructure management easier and more efficient, it is necessary to further analyze and define the opportunity.

The major issues that are commonly associated with Infrastructure as a Service in cloud systems are virtualization and multi-tenancy, resource management, network infrastructure management, data management, **APIs**, interoperability etc (Manvi and Shyam 2014). All these problems can be minimized by introducing some kind of abstraction in the form of a management tool that provides an interface for the users to interact with these services and configure them.

3.2.2 Perceived Value

To realize the value of using an infrastructure management tool, it is necessary to identify the stakeholders, how can they benefit from the solution and what sacrifices they need to go through to use it. This can be seen on the table 3.3.

Stakeholders	Benefits	Sacrifices
Developers	<p>The solution makes it so that developers don't require as much DevOps knowledge in order to set up deployments.</p> <p>The solution creates a complexity abstraction regarding deployments to cloud providers facilitating the process.</p>	The solution requires developers to learn a new syntax to configure the deployments in it.
Companies	<p>The solution makes it so that companies don't need as many highly specialized DevOps engineers to set up repository analysis tools.</p> <p>The solution agilizes the process of creating workflows and managing projects.</p>	Requires the onboarding of new developers to a tool not broadly used.

Table 3.3: Tool's benefits and sacrifices per stakeholder

3.2.3 Value Proposition

To define the value proposition its necessary to identify the product, gain creators, pain relievers, gains, pains, and customer jobs, thus developing a value proposition canvas (figure 3.1).

The following value proposition represents the value to be delivered by using a solution to create infrastructure on demand.

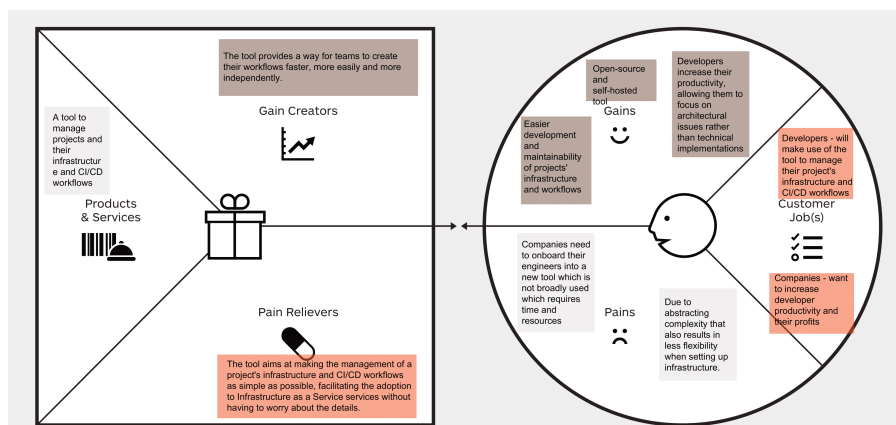


Figure 3.1: Value Proposition Canvas

3.2.4 Analytic Hierarchy Process

The Analytic Hierarchy Process (**AHP**) represents an accurate approach to quantifying weights of decision criteria. It aims at dividing the problem in hierarchy decision levels, facilitating its comprehension. In this case the **AHP** method aims to determine what is the highest priority when managing infrastructure.

The first step to determine this is to build an hierarchical decision tree, which can be seen on figure 3.2, with three levels representing the problem, the criteria, and the alternatives, respectively.

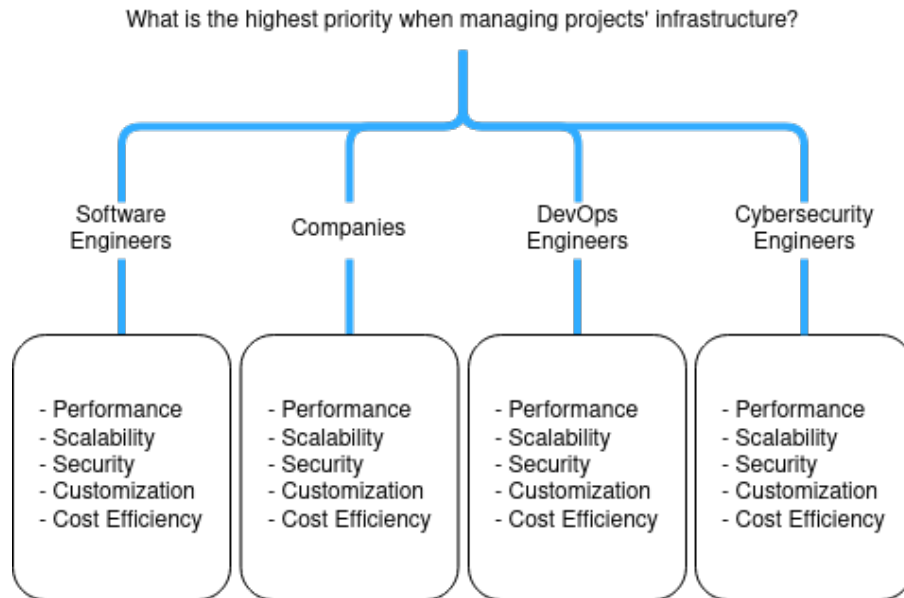


Figure 3.2: Hierarchical Decision Tree - Determining Priorities.

On the hierarchical decision tree, the first level represents the problem, which in this case is "What is the highest priority when managing projects' infrastructure?".

The second level represents the criteria that will be used in order to judge the priorities, in this case the Stakeholders that will use the product will be used as criteria in order to make this judgement, those being:

- Software Engineers - The developers that code the projects and interact with the tool to manage its infrastructure.
- Companies - The companies that make use of the solution.
- DevOps Engineers - The engineers with the DevOps knowledge that can use the solution to facilitate their work.
- Cybersecurity Engineers - The engineers that handle security in the company and make sure that their software is secure.

The third level represents the alternatives, in this case these are the priorities that should be taken in consideration, namely:

- Performance
- Scalability
- Security
- Customization
- Cost Efficiency

The second step is the establishment of a priority between each hierarchic level. The scale used to establish it can be seen on table 3.4.

Importance Level	Definition
1	Equal Importance
3	Weak Importance
5	Important
7	Very Important
9	Crucial

Table 3.4: Scale of importance

The pairwise comparison of the defined criteria can be seen in table 3.5.

	Soft. Eng.	Companies	DevOps Eng.	Cyb. Eng.
Soft. Eng.	1	1/3	1	3
Companies	3	1	3	5
DevOps Eng.	1	1/3	1	3
Cyb. Eng.	1/3	1/5	1/3	1
Sum	5 1/3	1 7/8	5 1/3	12

Table 3.5: Criteria pairwise comparison

After defining the pairwise comparison of the criteria it is necessary to obtain the relative priority of each criteria, this is done by normalizing the comparison matrix, and then obtaining the priority vector by calculating the arithmetical average of the values. This can be seen on table 3.6.

Software Engineers	0.2009
Companies	0.5193
DevOps Engineers	0.2009
Cybersecurity Engineers	0.0789

Table 3.6: Priority Vector

Once the priority vector is defined, the evaluation of the consistency of the relative priorities is done through the calculation of the Consistency Ratio (**CR**). If the **CR**'s value is superior to 0.1 then the judgements are not trustworthy. In this case the obtained CR was of 0.0161422, assuring the consistency of the comparison matrix.

Finally, to obtain the alternatives' priority composite, the priority vectors of each criteria's comparison matrix is multiplied with the criteria's relative priority. This can be seen on table 3.7.

	S.E.	Comp.	D.E.	C.E.	Vector	Priority
Performance	0.2539	0.1298	0.1952	0.0631	0.2009	0.162594
Scalability	0.5230	0.1298	0.1952	0.1650	0.5193	0.224698
Security	0.0827	0.3424	0.0737	0.4418	0.2009	0.244048
Customization	0.0702	0.0557	0.4624	0.1650	0.0789	0.148931
Cost Eff.	0.0702	0.3424	0.0737	0.1650	-	0.219727

Table 3.7: Alternatives' composite priority

In this case the alternative with the highest relative priority is Security, meaning that it should be the highest priority when designing the tool given the stakeholders that will be using it.

3.3 Summary

Throughout this chapter it was possible to identify the functional and non-functional requirements for the solution, namely the concerns regarding the performance of the solution, its security and its ease of use.

The solution's value and value proposition were properly detailed to synthesize its merit and lay out the means through which the user can capitalize from it. This proved that the solution is viable and important for the current market where DevOps is growing in importance and is increasingly expensive to hire good DevOps engineers.

It was necessary to evaluate the concerns regarding a tool capable of managing a company's infrastructure, to do that the various stakeholders and their concerns were used inputs for a multi-criteria decision model in order to determine which concern is the priority when building the solution. The resulting conclusion was that security is the highest priority when choosing a solution to manage the infrastructure and the full **AHP** analysis can be seen on the appendix A.

Chapter 4

Design

This section explains the design of the proposed solution. It is expected that all future approaches follow the design developed and documented here, however as the code evolves, there can be some discrepancies, but whenever possible the design must be followed, or as similar as possible. The section starts by going over the domain of the solution being developed in order to grasp the context of the solution being developed. Afterwards the solution's high level architecture is designed based on the domain previously defined, the sub-domains identified and based on the requirements analyzed on chapter 3. Once the architecture is defined an overview of what each service will look like internally is documented, where each component of a given service and their responsibility is explained. Finally, based on the sub-domains identified, an overview of each use case is given with a description and a diagram representing the flow that is to be expected of the services in order to achieve the use case.

4.1 Domain

The domain can be analysed on the Figure 4.1 and is composed of the following classes:

- **Deployment** - represents a deployment request of a set number of machines for a given service based on the global configuration
- **Instance** - represents a machine that is being deployed
- **Credential** - represents a credential that will be used to access the different services
- **Configuration** - represents the global configuration of the tool
- **Use Case Configuration** - represents the configuration of a given use case
- **Service Configuration** - represents the configuration of a given service for a specific use case
- **Plugin Information** - represents the information of a given service plugin

The Domain Model was built based on the Unified Modeling Language (**UML**) notation and *Craig Larman's* guidelines.

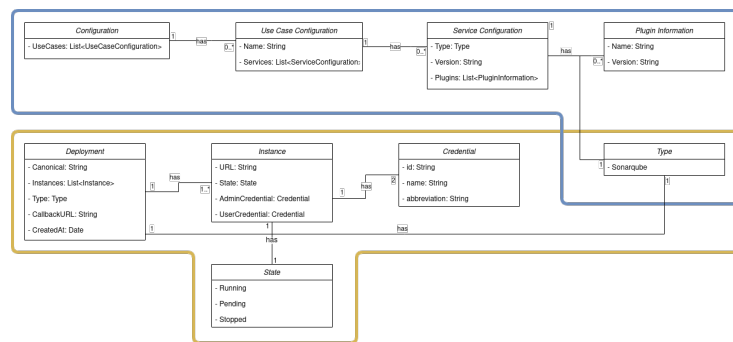


Figure 4.1: Domain Model

4.2 Architecture

The solution will be developed based on a monolithic architecture. In order to make it scalable the domain will be split in several sub-domains, where each sub-domain corresponds to a different part of the business.

Based on this approach the following sub-domains were identified and isolated:

- Configuration management
- Deployments Management

Each of these sub-domains will be isolated from each other in the code, although the architecture will be the one seen on Figure 4.2.

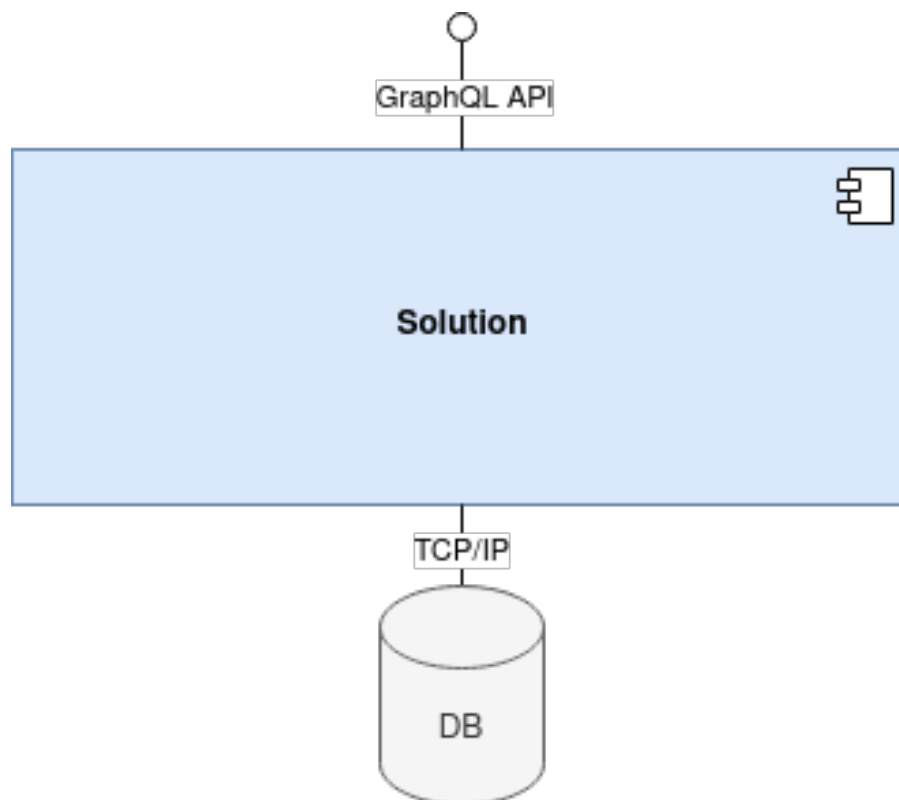


Figure 4.2: Architecture Diagram

4.3 Internal components

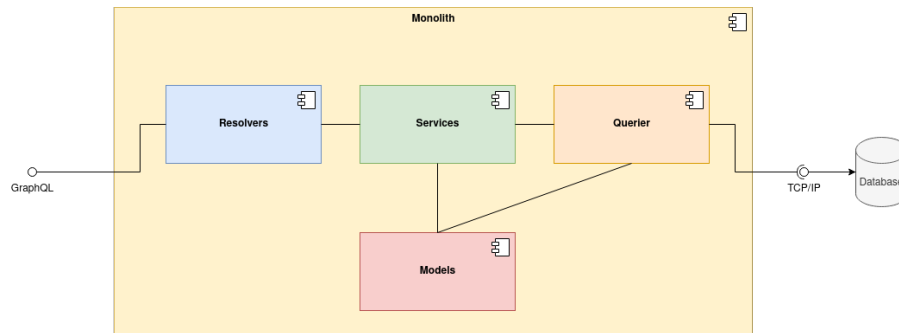


Figure 4.3: Internal components diagram

It can be observed on figure 4.3 that the service will follow an internal structure, where the service will expose a GraphQL **API**, and will be composed of 4 smaller components, namely "Resolvers", "Models", "Services" and "Querier":

- **Resolvers** - The resolvers are responsible for resolving the GraphQL requests and calling the adequate service in order to do the actual business logic;
- **Models** - Models represent the domain, and map the entities of this domain to actual objects that will be used across the different components;
- **Services** - Services are responsible for implementing the business logic and make use of the querier to manage the models;
- **Querier** - The querier is responsible for creating the isolation between persistence layer and business logic layer, serving as an interface for the services to interact with the persistence layer.

4.4 Requirements

This section presents the design of the functional requirements' process, using Sequence Diagram (SD).

4.4.1 Configuration Management

For this requirement it is necessary that the tool provides the users the means to manage the configuration. In order to do this it is necessary that the users are able to create, update, delete and retrieve the configuration. This includes the configuration for each use case and for each service. The respective diagram for the global configuration can be seen in the figure 4.4. For the configuration of a specific use case we can see the figures 4.5, 4.6, 4.7 and 4.8, and for the configuration of a specific service we have the figures 4.9, 4.10, 4.11 and 4.12.

The reading of the global configuration (figure 4.4) consists of reading all the use case configurations in the database as well as the corresponding service configurations associated.

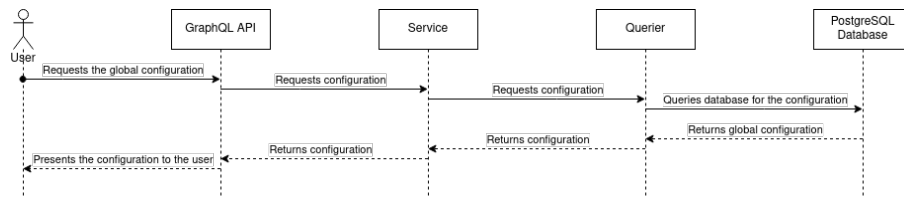


Figure 4.4: Read global configuration

The reading of a use case configuration (figure 4.5) retrieves a specific use case configuration from the database along with all its associated service configurations.

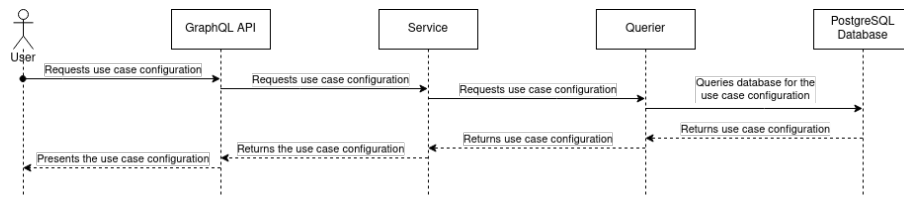


Figure 4.5: Read use case configuration

The process of adding a use case configuration (figure 4.6) will add the specific use case configuration along with all the service configurations associated to the global configuration, providing a way for users to add use cases to the configuration.

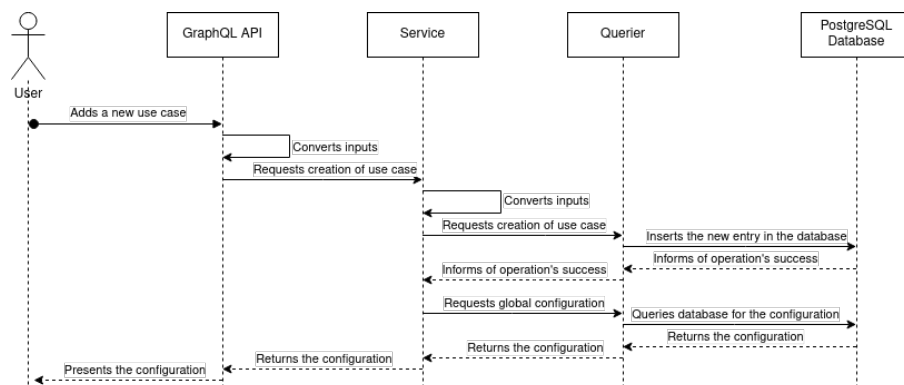


Figure 4.6: Add use case configuration

Updating a given use case configuration (figure 4.7) will update all of the services of the given use case configuration, meaning that all its services will be replaced by the new ones, providing a way for users to update all services of a given use case.

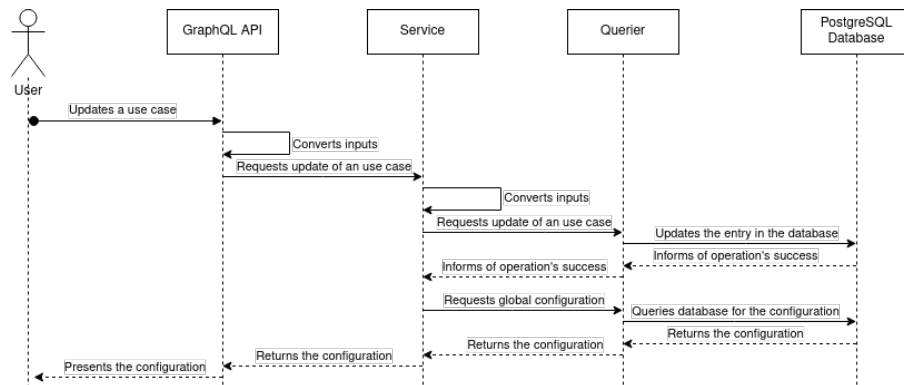


Figure 4.7: Update use case configuration

The deletion of a use case configuration (figure 4.8) consists of the removal of the use case from the global configuration, along with all its service configurations.

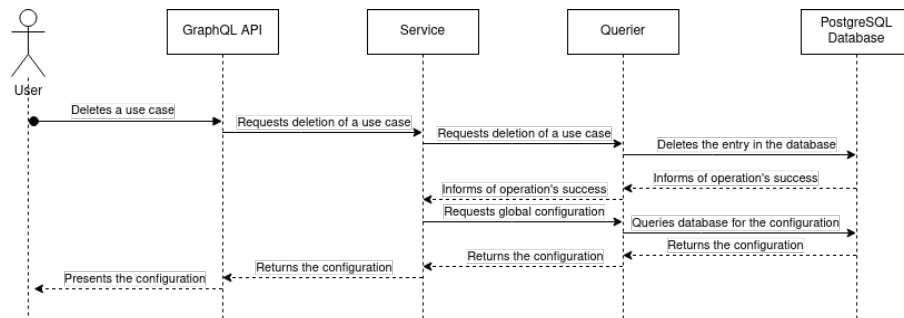


Figure 4.8: Delete use case configuration

The reading of a service configuration (figure 4.9) retrieves a specific service configuration from a given use case from the database along with all its associated information.

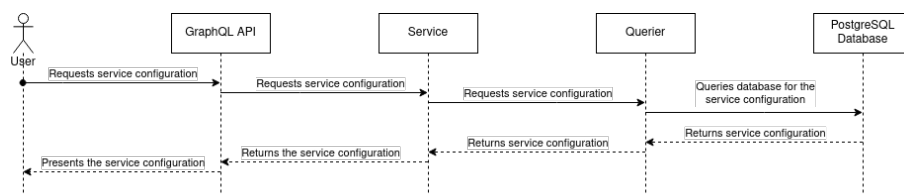


Figure 4.9: Read service configuration

Adding a service configuration (figure 4.10) consists of the insertion of a service configuration on a given use case, so that the use case can be used to make deployments of the given service.

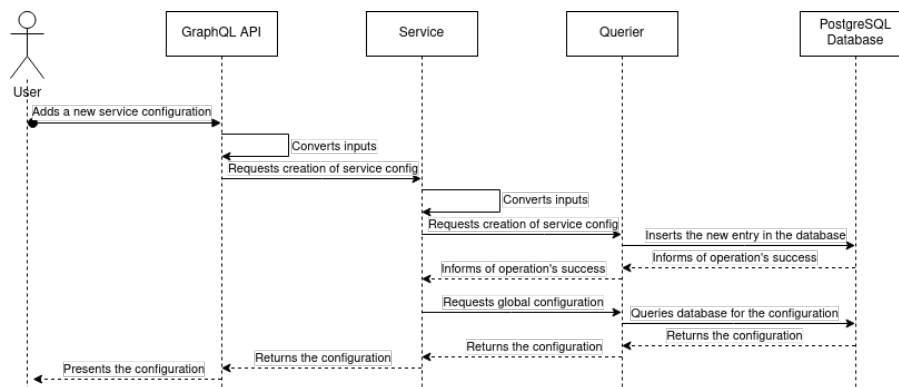


Figure 4.10: Add service configuration

Updating a service configuration (figure 4.11) will update an existing entry in the database, where the user can change which version to use and which plugins to include.

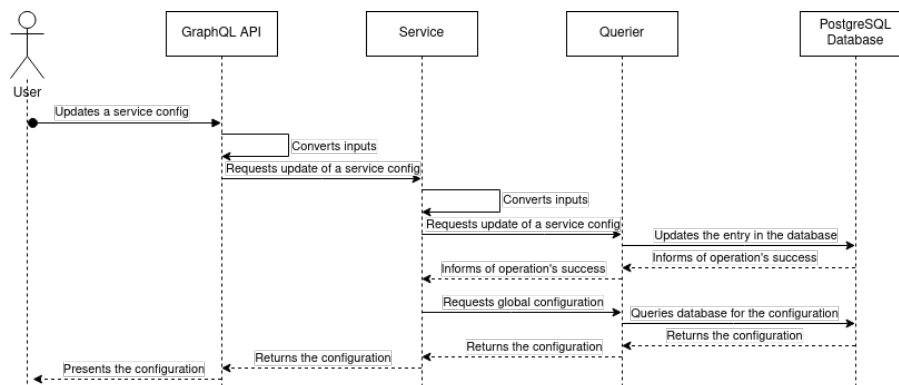


Figure 4.11: Update service configuration

The deletion of a service configuration (figure 4.12) consists of the removal of the configuration of a specific service from a given use case.

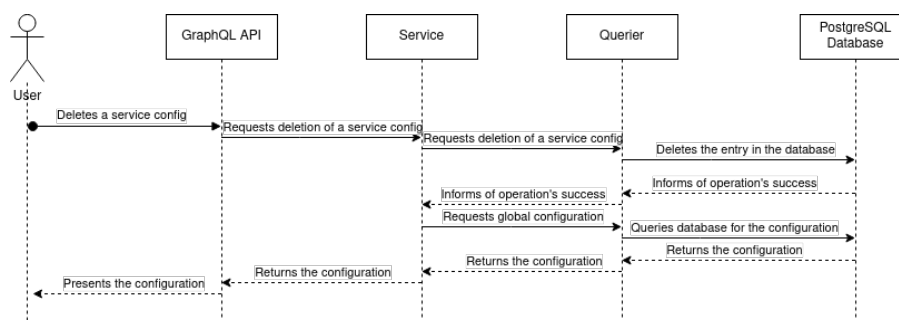


Figure 4.12: Delete service configuration

4.4.2 Deployment Management

The deployments represent an actual deployment done to the Cloud. The tool provides a way for users to read all deployments, read a specific deployment, create a new deployment

and delete an existing deployment. The diagrams representing these interactions can be found on figures 4.13, 4.14, 4.15 and 4.16.

The reading of all deployments (figure 4.13) retrieves all deployments that are stored in the database and returns them to the requesting user.

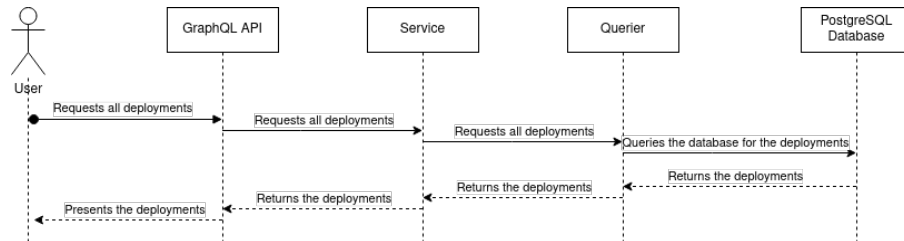


Figure 4.13: Read all deployments

Reading an existing deployment (figure 4.14) retrieves a given deployment based on his canonical name, returning it to the user.

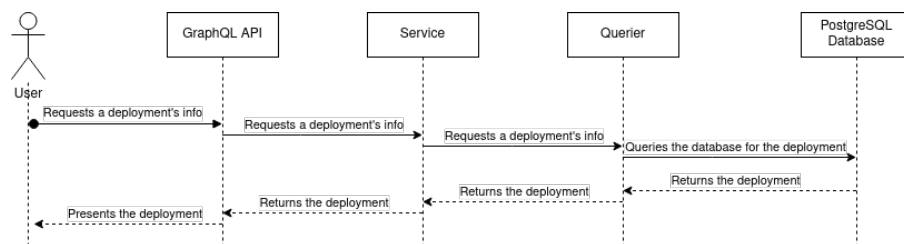


Figure 4.14: Read one deployment

The creation of a new deployment (figure 4.15) will first verify that the configuration that the user requested to be used actually exists, once that verification is done it will retrieve the template files for the requested service then for each service it load all the necessary data for the deployment. Once that is done it will start an asynchronous process for each deployment that will be responsible for building the directory where the files for that specific deployment will live, it will also do the interpolation of the template files so that they can be written on the folder of the specific deployment, once the files are written it will make use of Terraform to do the deployment using the newly generated files, once the deployment is done it will retrieve the public **IP** addresses of the instances created and persist those instances in the database so that they can be accessed by the observer.

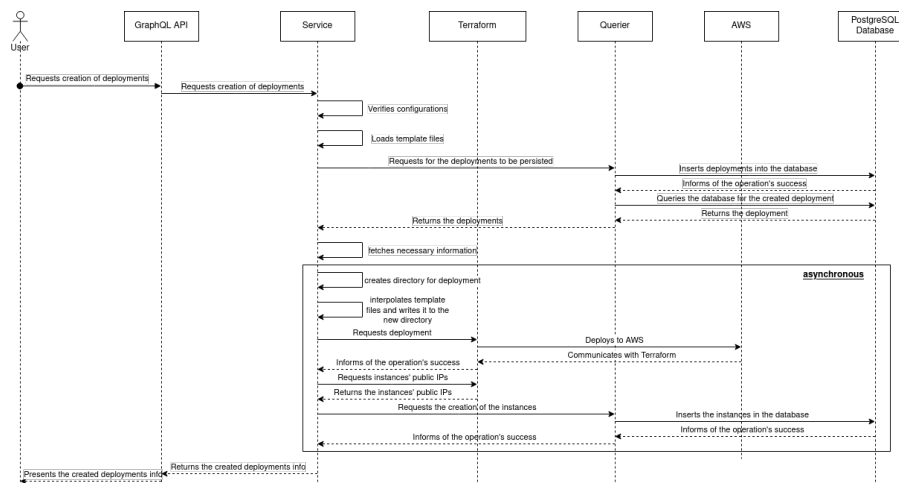


Figure 4.15: Create a new deployment

The deletion of an existing deployment (figure 4.16) will make use of Terraform to destroy the deployment in the Cloud, then it will delete the directory of that deployment and finally it will delete it from the database.

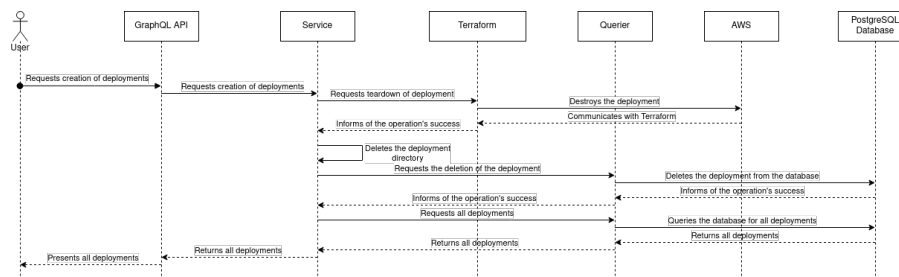


Figure 4.16: Delete an existing deployment

4.4.3 Deployment Observation

In order for the users to be notified when a given service is available to be connected to there needs to be some kind of observability implemented so that when these services are up we send the relevant data to the user. The diagram representing this flow can be seen on figure 4.17.

The observation of the deployments (figure 4.17) consists of a process that every minute will fetch all deployments that are still pending from the database and verify the connection to the different instances, in case the connection cannot be established it will verify whether the service has been pending for longer than a certain amount of time, if it is the deployment is deleted so that it does not consume any more resources, otherwise a user is created on the instance and the information is sent to the end user so that he can connect to that same instance.

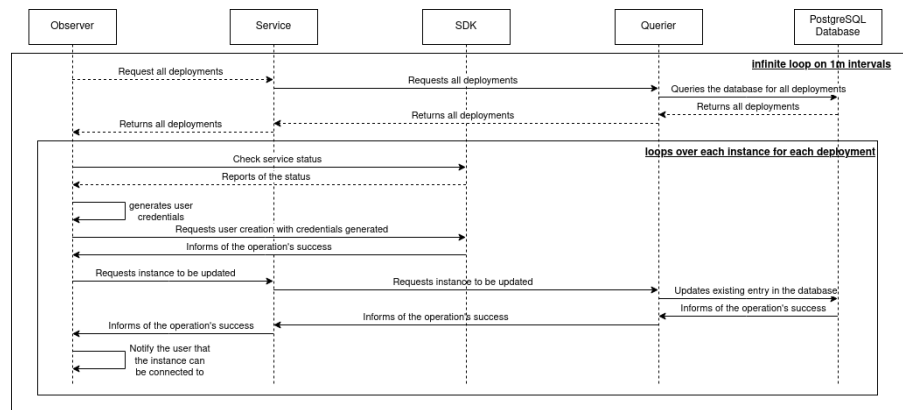


Figure 4.17: Observe deployments

4.5 Summary

Throughout this chapter the domain of the solution was documented, referencing the different domain entities that made sense to have in the solution to accomplish the functional requirements established on the previous chapter, a domain model containing all these domain entities and their respective attributes was also created in order to more easily visualize the associations between each entity.

In this chapter the solution's architecture was also mentioned, having been made the decision to go forward with a monolithic architecture that would have two main components, the server and the observer, which would interact with a database. The monolith's internal architecture was also documented, having four main components, the resolvers, the models, the services and the querier.

Finally it was also documented each functional requirement's use case along with a diagram describing the process of how the different components of the solution would interact to achieve the use case.

Chapter 5

Implementation

This section presents the development process of the proposed solution. This includes the process of developing the services, the process of encapsulating the solution and the process through which the service is able to do dynamic deployments.

Note that the development of the solution was done using the following technologies:

- Go as the programming language.
- Cobra as the framework for the **CLI**.
- gqlgen as the tool to serve the GraphQL **API**.
- GORM as the Object-Relational Mapping (**ORM**) to interact with the database.
- PostgreSQL as the Database System used.
- Docker in order to encapsulate the solution.
- Terraform so that the Infrastructure could be deployed to the Cloud.
- Ansible as the tool to configure the instances in the Cloud.

5.1 The Structure

The solution was built as a **CLI** since it would allow to start different services using different commands from the same monolithic solution. This solution exposes three commands:

- **server** - command that starts the GraphQL **API** that handles configuration management and deployment management.
- **observer** - command that starts the Observer responsible for doing the last steps of the instance configuration and notifying the user once it is ready.
- **fetch** - command used to fetch all necessary static information including app versions and plugins.

It is also worth mentioning that the root command, code listing 5.1, is the one responsible for registering any common flags across commands and registering the sub-commands so that these can be called from the root.

```
1 // RootCmd is the root command of the application to
2 // which all the other subcommands belong to
3 var RootCmd = &cobra.Command{
4     Use:     "lift",
5     Short:   "Lift's CLI",
```

```

6 Long: "Lift's full featured Command Line Interface",
7 }
8
9 func init() {
10 // Debug flag
11 RootCmd.PersistentFlags().Bool("debug", false, "Run the service in debug mode")
12 viper.BindPFlag("debug", RootCmd.PersistentFlags().Lookup("debug"))
13
14 // Terraform flags
15 RootCmd.PersistentFlags().String("terraform_exec_path", "", "Path to the terraform
    executable")
16 viper.BindPFlag("terraform_exec_path", RootCmd.PersistentFlags().Lookup("
    terraform_exec_path"))
17
18 // Database configuration flags
19 RootCmd.PersistentFlags().String("db_host", "localhost", "The host where the
    database lives")
20 viper.BindPFlag("db_host", RootCmd.PersistentFlags().Lookup("db_host"))
21 RootCmd.PersistentFlags().String("db_user", "user", "The database user to connect
    with to the database")
22 viper.BindPFlag("db_user", RootCmd.PersistentFlags().Lookup("db_user"))
23 RootCmd.PersistentFlags().String("db_password", "password", "The database user's
    password")
24 viper.BindPFlag("db_password", RootCmd.PersistentFlags().Lookup("db_password"))
25 RootCmd.PersistentFlags().String("db_name", "lift", "The database name")
26 viper.BindPFlag("db_name", RootCmd.PersistentFlags().Lookup("db_name"))
27 RootCmd.PersistentFlags().Int("db_port", 5432, "The port in which the database is
    listening on")
28 viper.BindPFlag("db_port", RootCmd.PersistentFlags().Lookup("db_port"))
29
30 RootCmd.AddCommand(
31     serverCmd,
32     observerCmd,
33     fetchCmd,
34 )
35 }

```

Listing 5.1: Root command

The directory containing almost all of the logic is the **internal** directory, this directory is the "internal" logic directory and contains several directories that will each contain logic that contributes to the whole functionality of the solution, namely:

- **config** - has the model that represents the configuration of the solution.
- **db** - has all the logic related to interacting with the database, and includes all the database models and querying logic.
- **fetcher** - contains the logic around fetching version and plugin information from the internet related to the different services that can be deployed.
- **graph** - holds the logic around the GraphQL **API**, which includes schemas, query and mutation resolvers, models generated and the GraphQL handler for the server.
- **models** - contains the domain objects used across the business logic as well as the Data Transfer Object (**DTO**).
- **observer** - has the logic of the Observer worker, which is responsible for listening for working instances in order to configure them and notify the user.
- **sdk** - holds custom Software Development Kit (**SDK**) created in order to more easily interact with the different types of services **APIs**.

- **services** - contains the business logic of the solution, meaning that all the logic around deployments and configuration lives here.
- **terraform** - has the Terraform worker logic created in order to perform the application and teardown of deployments.
- **utils** - has a multitude of utility functions that are used across the solution.

The project was structured with scalability in mind, so that the support of new services can easily be done by creating a few directories and constants across the project. To achieve support of a new service it is required the creation of a template folder with the name of the service, a fetcher for the service that queries for the service's application and plugin versions (if needed), and to add the service as a constant on the GraphQL schema and domain models. This way the code can be extended without a need for a lot of modification in an attempt to fulfil the Open-Closed Principle.

5.2 The Querier

The Querier is responsible for the interaction with the Database, meaning it abstracts the interaction with the Database from the service. It supports a multitude of operations on the different models that are used to make changes to the Database.

The Querier shown on the code listing 5.2 is the interface used to abstract the implementation details from the service layer, so that any structure that satisfies the interface can be used as a Querier for the Service, this is done in an attempt to fulfil the Dependency Inversion Principle, making it so that a different Querier implementation can be used in the future.

```

1 type Querier interface {
2     // Configuration CRUD operations
3     GetConfiguration(ctx context.Context) (*Configuration, error)
4     GetUseCaseConfiguration(ctx context.Context, uc string) (*UseCaseConfiguration,
5         error)
6     GetServiceConfiguration(ctx context.Context, uc string, service uint) (*
7         ServiceConfiguration, error)
8     CreateUseCaseConfiguration(ctx context.Context, newUC UseCaseConfiguration) (*
9         UseCaseConfiguration, error)
10    CreateServiceConfiguration(ctx context.Context, newS ServiceConfiguration) (*
11        ServiceConfiguration, error)
12    UpdateConfiguration(ctx context.Context, updatedC Configuration) error
13    UpdateUseCaseConfiguration(ctx context.Context, updatedUC UseCaseConfiguration)
14        error
15    UpdateServiceConfiguration(ctx context.Context, updatedS ServiceConfiguration)
16        error
17    DeleteUseCaseConfiguration(ctx context.Context, uc string) error
18    DeleteServiceConfiguration(ctx context.Context, uc string, service uint) error
19
20    // Deployment operations
21    GetAllDeployments(ctx context.Context) ([]*Deployment, error)
22    GetDeploymentByCanonical(ctx context.Context, can string) (*Deployment, error)
23    CreateDeployment(ctx context.Context, newD Deployment) (*Deployment, error)
24    UpdateDeployment(ctx context.Context, updatedD Deployment) error
25    DeleteDeployment(ctx context.Context, can string) error
26
27    // Instances operations
28    BatchCreateInstances(ctx context.Context, instances []Instance) error
29    UpdateInstance(ctx context.Context, updatedI Instance) error
30 }

```

Listing 5.2: Querier interface

The creation of the Querier is done through the *New* function (code listing 5.3) that accepts a configuration. Based on that configuration it will generate the connection string used to connect to the PostgreSQL Database. Using GORM, the **ORM** used in the project, it will open the connection to the database, with the connection open we call the method *AutoMigrate* with the different models that we want to support in the database, this will make sure all the models are correctly structured. Once the migration is complete we return the querier structure with the connection which will be used to interact with the database.

```

1 type querier struct {
2     db *gorm.DB
3 }
4
5 func New(cfg *config.Config) Querier {
6     conn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s sslmode=disable",
7         cfg.DBHost, cfg.DBPort, cfg.DBUser, cfg.DBPassword, cfg.DBName)
8     db, err := gorm.Open(postgres.Open(conn), &gorm.Config{})
9     if err != nil {
10         log.Fatal(err)
11     }
12     db.AutoMigrate(
13         &Deployment{}, &Instance{}, &Credential{},
14         &UseCaseConfiguration{}, &ServiceConfiguration{}, &PluginInformation{},
15     )
16     return &querier{db}
17 }
18

```

Listing 5.3: Querier New function

5.2.1 Models

The models (code listings 5.4 and ??) make use of one of the features of Go, *struct embedding* which allows the embedding of the attributes of a struct into another struct, to embed the *gorm.Model* struct, this is required so that GORM can use the model as a database model.

There is also the usage of field tags which allows the communication to the GORM **ORM** of relationships between tables and specifics about a field such as if it is an index, unique, etc. The code listing 5.4 shows one such case where tags are used to declare a one-to-many relationship between "Deployment" and "Instance", and the "Canonical" field has the tag which tells GORM that it is going to be an unique index.

```

1 type Deployment struct {
2     gorm.Model
3
4     Canonical string 'gorm:"uniqueIndex"'
5     Type      uint
6     Instances []Instance 'gorm:"foreignKey:DeploymentCanonical;references:Canonical"'
7     CallbackURL string
8 }

```

Listing 5.4: Deployment Model

5.2.2 Queries

The queries were implemented making use of the GORM functionalities. On the code listing 5.5, it is possible to see a helper function that makes use of the *Preload* method to eager

load the foreign keys of a Deployment into the Deployment model itself. This makes it so that instead of having to implement complex queries it is possible to simply call the helper method which will allow the retrieval of the sub-fields of the Deployment with no complexity.

```
1 func preloadDeployment(db *gorm.DB) *gorm.DB {  
2     return db.Preload("Instances.AdminCredential").Preload("Instances.UserCredential")  
3 }
```

Listing 5.5: Preloading helper function

On the code listing 5.6 it is possible to see the usage of the *preloadDeployment* helper function which is prepended to the *First* method, which is GORM's way of retrieving the first occurrence of a given entity that satisfies a given query, in this case that has the canonical passed as the last argument.

```
1 func (q *querier) GetDeploymentByCanonical(ctx context.Context, can string) (*  
    Deployment, error) {  
2     db := q.db.WithContext(ctx)  
3  
4     var deployment Deployment  
5     res := preloadDeployment(db).First(&deployment, "canonical = ?", can)  
6     if res.Error != nil {  
7         return nil, fmt.Errorf("deployment not found: %w", res.Error)  
8     }  
9     return &deployment, nil  
10 }
```

Listing 5.6: Deployment retrieval method

GORM contains a multitude of methods that provide ways to query, create, update and delete entities in the database, its usage was crucial to the implementation of the different methods of the Querier interface, making it so that the development speed was improved by a lot.

It is worth mentioning that although **ORMs** are helpful by making the development process faster for smaller projects and projects that are starting out they normally do not scale for more complex projects that require more complex queries, sometimes resulting in performance issues when handling a lot of data since these queries might not be optimized for the problem that is being solved.

5.3 The Server

The server is responsible for the handling of configuration and deployment management, meaning it supports all the Create Read Update Delete (**CRUD**) operations around Use Case Configurations and Service Configurations, while also supporting operations such as reading, creating and deleting deployments.

This section will go over the implementation of the server command, the GraphQL **API** and these two different components, along with their difficulties and the end solution.

5.3.1 Start Command

Since the solution was built as a **CLI**, the entry-point for the server is a command called "server", this command (code listing 5.7) will start the server and run the GraphQL **API** on port 8080 by default, resulting in a graphical playground that can be used to interact with

the server and make the wanted queries and mutations. This graphical playground is the result of the "gqlgen" library which provides a handler for this playground, making it so that serving it is easy.

```

1 var serverCmd = &cobra.Command{
2     Use:     "server",
3     Short:   "Starts the server",
4     Long:    "Starts the GraphQL server API with the given configuration",
5     RunE:    func(cmd *cobra.Command, args []string) error {
6         // Load the environment variables
7         err := env.Load(".env")
8         if err != nil {
9             log.Println("no .env file found: %w", err)
10        }
11
12        cfg := config.New(viper.GetViper())
13        utils.InstallTerraform(cfg)
14        repo := db.New(cfg)
15        s := services.New(repo, cfg)
16
17        router := mux.NewRouter()
18
19        router.Handle("/", graph.NewPlaygroundHandler("/query"))
20        router.Handle("/query", graph.NewHandler(s))
21
22        log.Printf("connect to http://localhost:%d/ for GraphQL playground", cfg.Port)
23        return http.ListenAndServe(fmt.Sprintf(":%d", cfg.Port), router)
24    },
25 }
26
27 func init() {
28     // Server flags
29     serverCmd.PersistentFlags().Int("port", 8080, "The port for the server to listen on")
30     viper.BindPFlag("port", serverCmd.PersistentFlags().Lookup("port"))
31 }

```

Listing 5.7: Server command

Note that one of the difficulties of the implementation of the server was concerning the guarantee that the machine that is running the server has Terraform installed, since it is a dependency when doing the actual deployments. To do this an utility function was created (code listing 5.8) that will make sure to install a temporary version of Terraform on the host machine and registering it on the configuration of the server so that it can be used in the future to do deployments.

```

1 // in case no terraform executable path was provided we will install it
2 // and pass the new executable path to the config
3 func InstallTerraform(cfg *config.Config) {
4     if cfg.TerraformExecPath == "" {
5         // we will install version 1.2.1 of Terraform
6         installer := &releases.ExactVersion{
7             Product: product.Terraform,
8             Version: version.Must(version.NewVersion("1.2.1")),
9         }
10
11         execPath, err := installer.Install(context.Background())
12         if err != nil {
13             log.Fatal("error installing Terraform: %w", err)
14         }
15         cfg.TerraformExecPath = execPath
16     }
17 }

```

Listing 5.8: Terraform installation function

5.3.2 GraphQL API

The GraphQL **API** was built using the *gqlgen* library, this library is based on a Schema first approach, meaning the GraphQL schema is defined first and then the Go code is generated that can be used to start the actual server and resolve the queries and mutations. This library can also be configured in terms of where the code is generated and where the schema is defined, this can be seen on the code listing 5.9 which contains the YAML configuration used for the generation of the Go code.

```

1 # Where are all the schema files located? globs are supported eg  src/**/*.graphqls
2 schema:
3   - internal/graph/schema/*.gql
4
5 # Where should the generated server code go?
6 exec:
7   filename: internal/graph/generated.go
8   package: graph
9
10 # Uncomment to enable federation
11 # federation:
12 #   filename: graph/generated/federation.go
13 #   package: generated
14
15 # Where should any generated models go?
16 model:
17   filename: internal/graph/models_gen.go
18   package: graph
19
20 # Where should the resolver implementations go?
21 resolver:
22   layout: follow-schema
23   dir: internal/graph
24   package: graph
25   filename_template: "{name}.resolvers.go"
26
27 # Optional: turn on use ' + "\"" + 'gqlgen:"fieldName"' + "\"" + ' tags in your models
28 # struct_tag: json
29
30 # Optional: turn on to use []Thing instead of []*Thing
31 omit_slice_element_pointers: true

```

Listing 5.9: *gqlgen* configuration file

As mentioned before, the use of the "gqlgen" library is responsible for serving a playground graphical user interface to use the GraphQL **API**, this graphical user interface can be seen on figure 5.1.

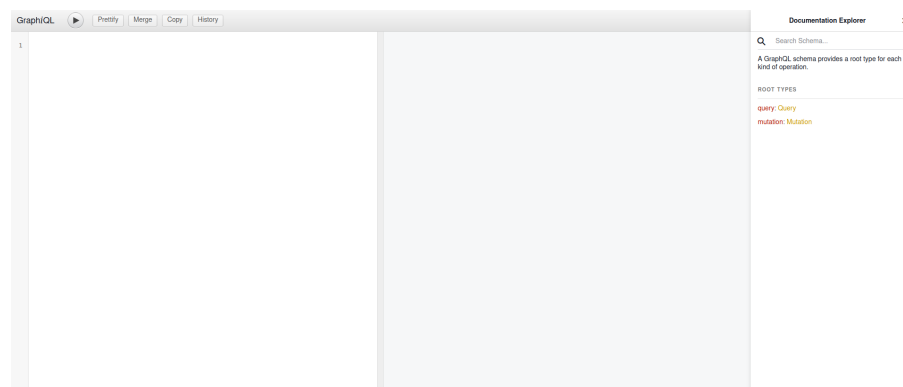


Figure 5.1: GraphQL Playground

The Schema is defined on the *internal/graph/schema* folder, and contains all of the Schema definitions needed for the user to interact with the services, this includes type definitions for data, inputs and also queries and mutations. The definition of the schema makes it so that the Go code generated contains all the necessary type definitions and makes it so that it is only required to implement the resolvers for each query and mutation defined in the schema.

The resolvers are responsible for resolving the queries and mutations requested by the user. This allows so that the responsibilities are split, and the resolvers are only concerned about receiving the user requests and resolving them, meaning executing the actual logic. On the code listing 5.10 the resolver converts the input into a **DTO** that the service can comprehend, once it receives the response back from the service the resolver resolves the model that it got from the service into a valid GraphQL model generated based on the Schema defined previously.

```

1 func (r *mutationResolver) CreateDeployments(ctx context.Context, input
  NewDeployments) ([]Deployment, error) {
2   deployments, _, errors := r.s.CreateDeployment(ctx, input.toDTO())
3   if len(errors) > 0 {
4     fullErr := fmt.Errorf("failed to create the deployment: ")
5     for _, err := range errors {
6       fullErr = fmt.Errorf("%s\n%w", fullErr, err)
7     }
8     return nil, fullErr
9   }
10
11   nds := make([]Deployment, len(deployments))
12   for i, deployment := range deployments {
13     nd := Deployment{
14       Canonical: deployment.Canonical,
15       Instances: make([]Instance, len(deployment.Instances)),
16       Type:      deployment.Type,
17     }
18
19     for j, instance := range deployment.Instances {
20       nd.Instances[j] = Instance{
21         State: DeploymentState(instance.State),
22       }
23     }
24     nds[i] = nd
25   }
26   return nds, nil
27 }

```

Listing 5.10: Create Deployment Resolver

5.3.3 Configuration Management

The server is also responsible for the configuration management, meaning it handles all the **CRUD** operations around the configuration's use cases and service configuration. Since the configuration itself does not require too much logic, all operations for both configuration of use cases and services is fairly straightforward only needing to use the querier to interact with the database and make the changes or query for data. Examples are provided in order to showcase the flow for each one of the **CRUD** operations.

Retrieving the global configuration (code listing 5.11) is achieved by calling the `GetConfiguration` method of the querier, resulting in the object that should be returned to the user.

```

1 // ReadConfiguration reads the whole configuration
2 func (s *Service) ReadConfiguration(ctx context.Context) *models.Configuration {
3     cfg := &models.Configuration{}
4     dbConfig, err := s.repo.GetConfiguration(ctx)
5     if err != nil {
6         return cfg
7     }
8     cfg.FromDB(dbConfig)
9     return cfg
10 }

```

Listing 5.11: Read Configuration

The creation of a use case configuration (code listing 5.12) consists of simply persisting the model in the database and then querying the newly changed configuration so that it can be returned to the user.

```

1 // AddConfigurationUseCase adds a new usecase to the configuration
2 func (s *Service) AddConfigurationUseCase(ctx context.Context, ucconfig *dtos.
    NewUseCaseConfiguration) (*models.Configuration, error) {
3     _, err := s.repo.CreateUseCaseConfiguration(ctx, *ucconfig.ToDB())
4     if err != nil {
5         return nil, fmt.Errorf("failed to add the use case %s to the configuration: %w",
            ucconfig.Name, err)
6     }
7
8     // fetch updated global configuration
9     dbgc, err := s.repo.GetConfiguration(ctx)
10    if err != nil {
11        return nil, fmt.Errorf("failed to retrieve the global configuration: %w", err)
12    }
13    gconfig := &models.Configuration{}
14    gconfig.FromDB(dbgc)
15    return gconfig, nil
16 }

```

Listing 5.12: Add Use Case to Configuration

Updating a use case configuration (code listing 5.13) can be done by simply calling the querier with the updated entry. Once that is done the global configuration is retrieved so that the updated configuration is returned to the user.

```

1 // UpdateConfigurationUseCase updates a specific usecase in the configuration
2 func (s *Service) UpdateConfigurationUseCase(ctx context.Context, usecase string,
    ucconfig *models.UseCaseConfiguration) (*models.Configuration, error) {
3     ucconfig.Name = usecase
4     err := s.repo.UpdateUseCaseConfiguration(ctx, *ucconfig.ToDB())
5     if err != nil {
6         return nil, fmt.Errorf("failed to update the usecase %s in the configuration: %w",
            usecase, err)
7     }
8
9     // fetch updated global configuration
10    dbgc, err := s.repo.GetConfiguration(ctx)
11    if err != nil {
12        return nil, fmt.Errorf("failed to retrieve the global configuration: %w", err)
13    }
14    gconfig := &models.Configuration{}
15    gconfig.FromDB(dbgc)
16    return gconfig, nil
17 }

```

Listing 5.13: Update Use Case Configuration

The deletion of a use case configuration (code listing 5.14) is done by making the request to the querier which will make the changes on the database and then querying again for the newly updated global configuration.

```

1 // DeleteConfigurationUseCase deletes a specific usecase from the configuration
2 func (s *Service) DeleteConfigurationUseCase(ctx context.Context, usecase string) (*
  models.Configuration, error) {
3     err := s.repo.DeleteUseCaseConfiguration(ctx, usecase)
4     if err != nil {
5         return nil, fmt.Errorf("failed to delete usecase %s from the configuration: %w",
           usecase, err)
6     }
7
8     // fetch updated global configuration
9     dbgc, err := s.repo.GetConfiguration(ctx)
10    if err != nil {
11        return nil, fmt.Errorf("failed to retrieve the global configuration: %w", err)
12    }
13    gconfig := &models.Configuration{}
14    gconfig.FromDB(dbgc)
15    return gconfig, nil
16 }

```

Listing 5.14: Delete Use Case Configuration

5.3.4 Deployment Management

The server also takes care of the management of the deployments, this includes fetching the currently active deployments, the retrieval of a specific deployment based on its canonical (unique identifier), the creation of a new deployment and the deletion on an existing one. The implementation of the deployment management required resolving more concerns since the deployments needed to execute code that would deploy one or more instances to a given cloud provider, in this case **AWS**.

The retrieval of deployments (code listing 5.15), similarly to the retrieval of the global configuration, simply calls the querier in order to fetch the data and returns it to the user and takes care of the conversion from database models to domain models that can be returned.

```

1 // ReadAll retrieves all deployments
2 func (s *Service) ReadAllDeployments(ctx context.Context) []*models.Deployment {
3     dbds, err := s.repo.GetAllDeployments(ctx)
4     if err != nil {
5         return nil
6     }
7     deployments := make([]*models.Deployment, len(dbds))
8     for i, dbd := range dbds {
9         deployment := &models.Deployment{}
10        deployment.FromDB(dbd)
11        deployments[i] = deployment
12    }
13    return deployments
14 }

```

Listing 5.15: Read all deployments

One of the main concerns when developing the logic around creating a deployment was how to make it dynamic and how to make it fast. The first one is achieved by making use of template files which live in a specific path that is accessed based on the service that is being created, this allows so that new services can be added to the solution without requiring many

changes to the code. The last one is achieved by making use of Go's concurrency model, creating goroutines that will perform the heavy lifting in parallel, making it so that multiple deployments can be done at the same time without affecting the time it takes to deploy all instances, only taking as long as the slowest deployment.

The process of creating a deployment itself (appendix B) starts by validating the configuration that the user wants to use, if the configuration exists it is cached in order to access it when doing the actual deployment. Once complete the paths to the template files that will be used to generate the deployment files are retrieved and the deployment is persisted with no instances, since the deployment hasn't yet happened. The interpolator is then loaded, the interpolator is responsible for replacing the data in the template files for actual data specific for each deployment such as the version of the service to use, the plugins and the amount of instances that should be deployed. With the interpolator available the goroutines are created which will start the heavy lifting of building all the specific deployment files, using the interpolator, in their correct location, running the Terraform code that will do the deployment, retrieve the public IP addresses of the instances created and persist the newly deployed instances. When all the asynchronous processes are over the errors are logged.

The deletion of a given deployment (code listing 5.16) will delete a given deployment from the path where the deployment files for that deployment are stored, making use of Terraform it will teardown the deployment's resources making it so that no resources are wasted.

```

1 // Delete deletes a deployment with the given canonical
2 func (s *Service) DeleteDeployment(ctx context.Context, dcan string) ([]*models.
   Deployment, error) {
3     // create the Terraform worker which will delete the deployments
4     tfw := terraform.NewWorker(s.config.TerraformExecPath)
5
6     deploymentDir, err := utils.BuildDeploymentFolderPath(dcan)
7     if err != nil {
8         return nil, fmt.Errorf("could not build path to deployment %s: %w", dcan, err)
9     }
10
11     err = tfw.Teardown(deploymentDir)
12     if err != nil {
13         return nil, fmt.Errorf("could not teardown deployment %s: %w", dcan, err)
14     }
15
16     err = os.RemoveAll(deploymentDir)
17     if err != nil {
18         return nil, fmt.Errorf("could not delete deployment files folder: %w", err)
19     }
20
21     err = s.repo.DeleteDeployment(ctx, dcan)
22     if err != nil {
23         return nil, fmt.Errorf("failed to delete deployment %s: %w", dcan, err)
24     }
25
26     deployments := s.ReadAllDeployments(ctx)
27     return deployments, nil
28 }

```

Listing 5.16: Delete deployment

5.4 The Observer

The observer is responsible for observing the deployments that are still pending on a set interval, making sure they are either up and running so that it can configure them, or taking

too long to be available and tearing them down.

This section will go over the implementation of the worker behind the observer.

5.4.1 Start command

Similarly to the server the entry-point for the observer worker is the "observer" command which is responsible for creating the worker that will do the job of listening for changes in the deployments, configuring them and notifying the user (code listing 5.17).

```

1 var observerCmd = &cobra.Command{
2     Use:     "observer",
3     Short:   "Starts the observer worker",
4     Long:    "Starts the observer worker responsible for notifying the user when
5             instances are up",
6     RunE: func(cmd *cobra.Command, args []string) error {
7         // Load the environment variables
8         err := env.Load(".env")
9         if err != nil {
10             log.Println("no .env file found: %w", err)
11         }
12
13         cfg := config.New(viper.GetViper())
14         utils.InstallTerraform(cfg)
15         repo := db.New(cfg)
16         s := services.New(repo, cfg)
17         ow := observer.NewWorker(s, cfg)
18
19         log.Printf("observer worker started")
20         return ow.Start()
21     },
22 }
```

Listing 5.17: Observer command

5.4.2 The Worker

The worker's *Start* method (code listing 5.18) is the main loop that will periodically iterate over the existing deployments in the database. It will loop on 1 minute intervals, which can be easily configured by changing the constants present in the code. This main loop will, for each deployment's instance that is still pending, check its connection by communicating with the instance's public **IP** address and handle the instance differently depending on the result.

```

1 func (w *Worker) Start() error {
2     tfw := terraform.NewWorker(w.cfg.TerraformExecPath)
3
4     for {
5         log.Println("Started iterating over deployments...")
6         ctx := context.Background()
7         deployments := w.s.ReadAllDeployments(ctx)
8         now := time.Now()
9         for _, d := range deployments {
10             // check whether we already exceeded the time limit
11             exceededLimit := now.Sub(d.CreatedAt) > limit
12
13             for _, i := range d.Instances {
14                 if i.State != models.Pending {
15                     continue
16                 }
17
18                 ok := w.checkConnection(i, d.Type)
```

```

19         if !ok {
20             err := w.handleFailedConnection(d.Canonical, exceededLimit, tfw)
21             if err != nil {
22                 log.Println(err)
23             }
24             continue
25         }
26
27         err := w.handleSuccessfulConnection(i, *d)
28         if err != nil {
29             log.Println(err)
30         }
31     }
32 }
33 time.Sleep(delay)
34 }
35 }

```

Listing 5.18: Observer worker Start method

If the instance is not currently available to connect and it exceeded the limit set for time that it can take to be up (configured to 15 minutes) then the whole deployment is deleted (code listing 5.19).

```

1 func (w *Worker) handleFailedConnection(canonical string, exceededLimit bool, tfw *
   terraform.Worker) error {
2     log.Println("Handling failed connection")
3     if !exceededLimit {
4         return nil
5     }
6
7     go func() {
8         _, err := w.s.DeleteDeployment(context.Background(), canonical)
9         if err != nil {
10             log.Println(err)
11         }
12     }()
13     return nil
14 }

```

Listing 5.19: Observer failure handler

In the case that the connection goes through then the instance needs to be configured (code listing 5.20), meaning the instance needs to be set up depending on the service type, this can include the creation of a user for Sonarqube or the creation of a job for Jenkins. This is achieved by making use of the custom **SDKs** created to abstract the implementation details of each service's **API** and consist of making **HTTP** calls to the **API** of the instance that was started.

```

1 func (w *Worker) handleSuccessfulConnection(i models.Instance, d models.Deployment)
   error {
2     log.Println("Handling successful connection")
3
4     // set the instance's admin credentials
5     adminUsername, adminPassword := utils.GetAdminCredentials(d.Type, i.URL)
6
7     i.AdminCredential = models.Credential{
8         Username: adminUsername,
9         Password: adminPassword,
10    }
11
12    userCreds, err := w.setup(i, d)
13    if err != nil {

```

```

14     return fmt.Errorf("could not set up instance: %w", err)
15 }
16
17 // update the instance to hold credentials and new state
18 i, err = w.updateInstance(i, d.Canonical, userCreds)
19 if err != nil {
20     return fmt.Errorf("could not update instance: %w", err)
21 }
22
23 // obfuscate the admin credentials
24 i.AdminCredential = models.Credential{}
25
26 // notify the user that the service requested is available
27 data, err := json.Marshal(i)
28 if err != nil {
29     return fmt.Errorf("could not parse data into json: %w", err)
30 }
31
32 r := strings.NewReader(string(data))
33 _, err = http.Post(fmt.Sprintf("%s/%s", d.CallbackURL, d.Canonical), "application/
    json", r)
34 if err != nil {
35     return fmt.Errorf("could not call user's service: %w", err)
36 }
37 return nil
38 }

```

Listing 5.20: Observer success handler

Once the instance is set up it is required that the instance itself is updated in the database so that in the future the worker is aware that the instance is already up and running and has been configured correctly. To do this the *updateInstance* method is called which will set the instance's user credentials, change the instance's status to running and persist that information (code listing 5.21).

```

1 func (w *Worker) updateInstance(i models.Instance, dcan string, userCreds *models.
    Credential) (models.Instance, error) {
2     i.UserCredential = *userCreds
3     i.State = models.Running
4
5     err := w.s.UpdateInstance(context.Background(), dcan, &i)
6     if err != nil {
7         return i, fmt.Errorf("could not persist user credential: %w", err)
8     }
9     return i, nil
10 }

```

Listing 5.21: Observer worker method to update the instance

Once everything has been correctly persisted and configured, then the user is notified that the instance can be connected to and sends the instance's information so that the user can persist any data he needs about the instance's location and credentials.

5.5 Template files

The template files are one of the core features of the solution, they are what allows the solution to make the deployments in a dynamic way that provides a way for the code to not be concerned about the implementation details of the deployment itself.

The variables in the template files can be identified by the use of parenthesis and the dollar sign, one example of such variable is "(\$ version \$)".

There are currently three restrictions when creating new template files for a new service:

- The folder containing the template files should have the name of the service and should be within the "templates" folder.
- The entry-point of the deployment should be a Terraform file called "main.tf".
- The Terraform file should provide the public **IP** addresses of the deployed instances as output.

The core template files used to deploy the Sonarqube and Jenkins instances can be seen on appendixes C and D respectively. The main idea behind their implementation is the creation of resources on **AWS** to configure the instance and the instance itself, there is also a different kind of resource, a "null resource" which is responsible for running the Ansible code in the instance that was deployed based on the ansible files present in the template files.

The resources responsible for configuring the instance in this case are **AWS** key pairs and **AWS** security groups, the later normally is configured to expose two ports in the instance, port 22 for **SSH** connections and the port specific to the service, for Sonarqube it is port 9000 and for Jenkins it is port 8080.

The instance itself consists of an Elastic Compute Cloud (**EC2**) instance of different size depending on the service that is being deployed and its requirements. Lastly the "null resource" copies files from the host machine into the instance that is being deployed, these files include utility and Ansible files, and will run a script that will update the instance, install Ansible and run the Ansible playbook on the instance on **AWS**.

As part of the requirements previously mentioned the Terraform file will also output the instances public **IP** addresses.

5.6 Encapsulating the Services

So that the solution can be started in a quick fashion, independently of the machine that will be running it, the encapsulation of the solution was done. This was achieved by making use of *Docker* and *docker-compose*, which allow the setup of containers that will run the solution without worrying about the host machine.

The server's Dockerfile (code listing 5.22) will do the following:

1. build the binary of the solution.
2. copy the binary into the "lift" directory in the container.
3. copy the necessary files from the host machine into the container, including the start script, the ".env" file, the template files and the public and private **SSH** keys.
4. run the start script (code listing 5.23) with the database username, password and host which will be used to connect to the database.

```
1 FROM golang:alpine as builder
2 RUN mkdir /build
3 ADD . /build/
4 WORKDIR /build
5 RUN go build -o main .
6
7 FROM alpine
```

```

8
9 RUN mkdir /lift
10 COPY --from=builder /build/main /lift/
11 COPY ./env /lift/
12 COPY ./templates/ /lift/templates/
13 COPY ./static/keys/lift.pub /lift/.ssh/id_rsa.pub
14 COPY ./static/keys/lift /lift/.ssh/id_rsa
15 COPY ./static/ /lift/static/
16 COPY ./scripts/start.sh /lift/
17 RUN chmod +x /lift/start.sh
18
19 WORKDIR /lift
20
21 ARG db_user
22 ARG db_password
23 ARG db_host
24
25 ENV DB_USER=${db_user}
26 ENV DB_PASSWORD=${db_password}
27 ENV DB_HOST=${db_host}
28
29 CMD ./start.sh --db_user ${DB_USER} --db_password ${DB_PASSWORD} --db_host ${DB_HOST}
  }
```

Listing 5.22: Server Dockerfile

```

1 #!/bin/sh
2
3 /lift/main server "$@" &
4 /lift/main observer "$@" &
5 wait
  
```

Listing 5.23: Server start script

The database's Dockerfile (code listing 5.24) will just move an Structured Query Language (**SQL**) script into the `"/docker-entrypoint-initdb.d"` container location, this will make it so that the SQL commands in that script are run on start up of the database, allowing the bootstrap of the database from the script. In this case there's only a line which will grant the privileges to the database user.

```

1 FROM postgres:14.3-alpine
2
3 ADD ./init.sql /docker-entrypoint-initdb.d
  
```

Listing 5.24: Database Dockerfile

With the Dockerfiles complete *docker-compose* is used to set up the services on the host machine (code listing 5.25). It describes two services, the "server" service and the "psql_db" service. The "server" service takes build arguments for the Dockerfile, and those are the database username, the database password and the database host which will be used on the "start.sh" script. It is also possible to see that this service depends on the "psql_db" service, which represents the database used by the solution, and the same database username and password will be used as environment variables for the container to create the non-root user.

```

1 version: '3.8'
2
3 services:
4   server:
5     container_name: lift_server
6     restart: always
7     network_mode: "host"
8     build:
  
```

```

 9      context: .
10      args:
11        - db_user=${DB_USER}
12        - db_password=${DB_PASSWORD}
13        - db_host=127.0.0.1
14      dockerfile: Dockerfile
15      depends_on:
16        - psql_db
17
18      psql_db:
19        container_name: lift_db
20        build:
21          context: db
22          dockerfile: Dockerfile
23        restart: always
24        environment:
25          POSTGRES_USER: ${DB_USER}
26          POSTGRES_PASSWORD: ${DB_PASSWORD}
27          POSTGRES_DB: ${DB_NAME}
28        ports:
29          - '5432:5432'
30        volumes:
31          - db-data:/data/db
32
33      volumes:
34        db-data:

```

Listing 5.25: docker-compose file

5.7 Automation

In order to make the development and testing process easier there was the use of a Makefile (code listing 5.26), which provides various targets that are used.

```

1  include .env
2
3  build-sql-init:
4    @./db/setup.sh ${DB_USER} ${DB_NAME}
5
6  gqlgen:
7    @go run github.com/99designs/gqlgen generate
8
9  generate-all: gqlgen
10   @go generate ./...
11
12  teardown:
13   @docker-compose down
14   @-./scripts/teardown.sh lift
15
16  start: teardown generate-all build-sql-init
17   @docker-compose --env-file ./env build --no-cache
18   @docker-compose --env-file ./env up
19
20  db-cli:
21   @docker exec -it lift_db psql -d ${DB_NAME} -U ${DB_USER} -h localhost -p 5432

```

Listing 5.26: Makefile

The "build-sql-init" target will run a script that will take as input the database username and the database name and generate the **SQL** *init* script for the database based on a template script.

The "gqlgen" target will generate the GraphQL **API**'s related code, so that when there are changes in the schema these can be easily converted into code. The "generate-all" target will regenerate all the code in the solution, meaning that if new constants are introduced or the GraphQL schema is changed then the "generate-all" target can be called to regenerate all the previously generated files.

The "teardown" target will take down the services started by *docker-compose* and make sure that there are no left-over resources being consumed by calling the "teardown.sh" script which deletes docker images related to the solution.

The "start" target will first call the "teardown" target, regenerate all the code and build the **SQL** init script, then it will build the Docker images based on the docker-compose file created, and once that is done it will run the images as containers.

Finally the "db-cli" target is used as a means to connect to the database directly to investigate any issues that may arise or confirm data changes while developing.

5.8 Summary

In this chapter an overview of the implementation process was done and some code snippets of the developed solution are also provided. This chapter provides an overview of the solution's structure, the querier component, the server component, the observer component, the template files, the encapsulation process and the automation used on the solution.

The solution's structure section goes over the commands exposed by the **CLI** and their purpose, having also some information regarding the structure of the directory containing the core business logic.

Afterwards, on the querier's section, examples are provided of the interactions with the database through the usage of GORM and how it was crucial to boost the development speed.

On the server's section, the server's logic is analysed, going over the logic behind the deployments and the design decisions that were made in order to achieve better performances on these, the GraphQL **API** built is also described along with the technology used to achieve it.

The observer's section gives special attention to how the instances that are being deployed are checked for connection and how they are handled depending on that result.

The section regarding the template files provides information regarding some of the requirements needed for each file and some of the similarities found between template files across different services, being the result of some decisions regarding the cloud provider being used.

Afterwards, an overview of how the solution was encapsulated is also provided, going over some of the files used to configure that encapsulation and some of the technologies used such as Docker and docker-compose.

Finally, a small section is dedicated to the automation done in the solution to facilitate the development process, this automation consists of a Makefile that is used to accomplish repetitive tasks.

Chapter 6

Evaluation

This section goes over the evaluation of the solution developed by making use of the metrics that were retrieved internally by the solution. These metrics will be used in order to track the evolution of the solution, its performance, and if it answers the research hypotheses. The evaluation was done through an empiric approach.

It is expected that the solution is capable of deploying repository analysis tools, orchestrating them and encapsulate them in an efficient way.

6.1 Process

The evaluation process of the research hypotheses is split into six phases:

- Creation of a use case configuration.
- Creation of a deployment that uses that configuration.
- Track how long the deployment took.
- Request its deletion.
- Track how long it took.
- Verify that it was correctly deleted.

These steps should be done once the first version of the solution is completed, and once every improvement is done in order to evaluate its progress throughout the development process.

6.2 Ease of use

The ease of use of the solution was one of the big concerns when developing the solution, it was important to make sure that the configuration of the deployments was easy to do. The end result is the GraphQL **API** which requires a mutation to add a new Use Case which can hold multiple types of services with a set of plugins that will be added to the instance deployed (figure 6.1), the process is easy and means that the user only requires to do the configuration once and after that a various amount of deployments can be done with that configuration as a base.



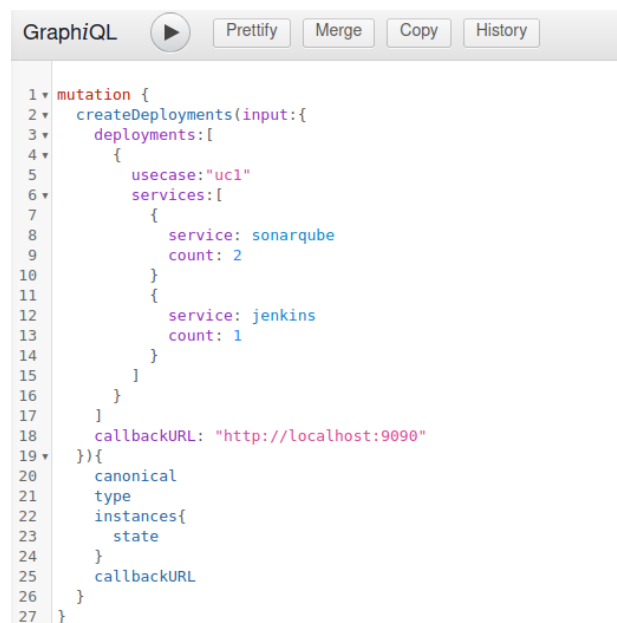
```

1 mutation {
2   addUseCaseConfiguration(input: {
3     name: "uc1"
4     services: [
5       {
6         type: sonarqube
7         version: "9.2.0.49834"
8         plugins: [
9           {
10            name: "mulesoft"
11            version: "0.5.1"
12          }
13        ]
14      }
15      {
16        type: jenkins
17        version: ""
18        plugins: []
19      }
20    ]
21  }) {
22    usecases {
23      name
24      services {
25        type
26        version
27        plugins {
28          name
29          version
30        }
31      }
32    }
33  }
34 }

```

Figure 6.1: GraphQL API add use case mutation

Once the configuration is set up, new deployments can be easily done with just a simple mutation delegating which use case to use, although more use cases are allowed as well, and which type of service to deploy and how many instances to deploy (figure 6.2). In order to not have hanging requests the user also defines a callback Uniform Resource Locator (**URL**) which will be used to inform of the instance's that are ready to be connected with, providing a way for users to automate the process of registering instances.



```

1 mutation {
2   createDeployments(input: {
3     deployments: [
4       {
5         usecase: "uc1"
6         services: [
7           {
8             service: sonarqube
9             count: 2
10          }
11          {
12            service: jenkins
13            count: 1
14          }
15        ]
16      }
17    ]
18    callbackURL: "http://localhost:9090"
19  }) {
20    canonical
21    type
22    instances {
23      state
24    }
25    callbackURL
26  }
27 }

```

Figure 6.2: GraphQL API create deployment mutation

6.3 Deployment time

The solution itself has a limit of time that it allows for a deployment to take, being capped at 15 minutes, if the deployed instance is not reachable in 15 minutes it will be deleted. The time measuring process consisted of logging the time at the moment of the start of the deployment process and the time at the moment of the notification to the user.

After creating a deployment with two Sonarqube instances and one Jenkins instance it is possible to verify through the logs on figure 6.3 that the Sonarqube instances took 4 minutes and 17 seconds and 4 minutes and 19 seconds respectively and the Jenkins instance took 6 minutes and 23 seconds, proving that the solution can deploy and configure services and notify the user within the 15 minutes limit.

```
server_1 | 2022/06/21 15:27:22 Canonical -> uci-sonarqube-62f9a809d126f182277415f77d1906d3 | Start Time -> 2022-06-21 15:27:22.029830805 +0000 UTC m=+108.157456172
server_1 | 2022/06/21 15:27:22 Canonical -> uci-jenkins-1e2e63456b81c436ba0aa73055f72279 | Start Time -> 2022-06-21 15:27:22.036398451 +0000 UTC m=+108.164023828
server_1 | 2022/06/21 15:27:37 Started iterating over deployments...
server_1 | 2022/06/21 15:28:37 Started iterating over deployments...
server_1 | 2022/06/21 15:29:37 Started iterating over deployments...
server_1 | 2022/06/21 15:30:37 Started iterating over deployments...
server_1 | 2022/06/21 15:31:37 Started iterating over deployments...
server_1 | 2022/06/21 15:31:37 Checking connection on URL 3.249.255.171
server_1 | 2022/06/21 15:31:38 Handling successful connection
server_1 | 2022/06/21 15:31:39 Canonical -> uci-sonarqube-62f9a809d126f182277415f77d1906d3 | End Time -> 2022-06-21 15:31:39.824780879 +0000 UTC m=+365.952409592
server_1 | 2022/06/21 15:31:39 Checking connection on URL 54.194.54.181
server_1 | 2022/06/21 15:31:40 Handling successful connection
server_1 | 2022/06/21 15:31:41 Canonical -> uci-sonarqube-62f9a809d126f182277415f77d1906d3 | End Time -> 2022-06-21 15:31:41.350148875 +0000 UTC m=+367.47777588
server_1 | 2022/06/21 15:32:41 Started iterating over deployments...
server_1 | 2022/06/21 15:33:41 Started iterating over deployments...
server_1 | 2022/06/21 15:33:41 Checking connection on URL 34.250.12.13
server_1 | 2022/06/21 15:33:45 Handling successful connection
server_1 | 2022/06/21 15:33:45 Canonical -> uci-jenkins-1e2e63456b81c436ba0aa73055f72279 | End Time -> 2022-06-21 15:33:45.891738178 +0000 UTC m=+492.019366891
```

Figure 6.3: Logging statements in the terminal

6.4 Deletion time

The time a deployment takes to delete is also an important metric since it is not wanted for a deployment to take too long to be deleted since that translates into resources being used when they should not. A similar approach to the deployment time was used, where logging was added before and after the deletion is complete having taken 1 minute and 25 seconds for the Sonarqube deployments and 53 seconds for the Jenkins deployment (figure 6.4), proving to be a quick process.

```
server_1 | 2022/06/21 16:11:08 Canonical -> uci-sonarqube-1c33f54fe9baf3095411f7e538adf90e | Start Time -> 2022-06-21 16:11:08.546914579 +0000 UTC m=+843.209471699
server_1 | 2022/06/21 16:11:19 Started iterating over deployments...
server_1 | 2022/06/21 16:12:19 Started iterating over deployments...
server_1 | 2022/06/21 16:12:33 Canonical -> uci-sonarqube-1c33f54fe9baf3095411f7e538adf90e | End Time -> 2022-06-21 16:12:33.310914614 +0000 UTC m=+927.973471704
server_1 | 2022/06/21 16:13:19 Started iterating over deployments...
server_1 | 2022/06/21 16:13:20 Canonical -> uci-jenkins-34294c1530984bbccbfbb4126f5fc018c | Start Time -> 2022-06-21 16:13:20.644725195 +0000 UTC m=+975.307282295
server_1 | 2022/06/21 16:14:13 Canonical -> uci-jenkins-34294c1530984bbccbfbb4126f5fc018c | End Time -> 2022-06-21 16:14:13.646086696 +0000 UTC m=+1028.308643796
```

Figure 6.4: Logging statements in the terminal

6.5 Summary

To properly evaluate the research hypotheses proposed on chapter 1, an empiric approach was used, where multiple instances of different services were requested for deployment.

The first hypothesis regarded whether the solution was capable of deploying the instances to the cloud, making them accessible in the Internet.

Upon evaluation, it is possible to conclude that, in regards to the hypothesis mentioned, it is clear that it was fulfilled. The solution was able to finalize the deployment processes and notify a user whenever an instance was ready, providing the link to the instance that could be accessed in order to interact with the instance.

The second hypothesis regarded whether the solution was capable of orchestrating the deployed instances, making it so that these could be removed or configured as needed.

After evaluating in regards to this hypothesis, it is possible to conclude that the solution partially fulfils this hypothesis. The solution is able to create, configure and remove resources from the cloud, but it does not support the scaling of a specific deployment or the reconfiguration of a deployment after completed.

Finally, the third hypothesis regarding the encapsulation of the services deployed, can be considered as partially fulfilled. The solution provides a way to isolate implementation details about the service being deployed from the code, it makes use of Terraform and Ansible to encapsulate that logic. Although the solution does not provide a complete encapsulation regarding the operating system that is being used for the instance, since the template files are crafted for the specific **OS** where the service will be deployed.

Chapter 7

Conclusion

This chapter provides a summary of the results of the solution developed, depicting the obtained results, limitations, and future improvements. Finally, this chapter goes over a reflection of the final result.

7.1 Summary

The solution involved an extensive study of different technologies such as GraphQL, Terraform, Ansible, Bash Scripting, Sonarqube and Jenkins throughout its development.

The solution developed consists of a GraphQL API capable of managing and configuring instances of Sonarqube and Jenkins on the cloud, more specifically on AWS. The deployment process was fairly quick taking no more than 10 minutes for either instance to be up and ready to be connected to by a user.

The solution developed is dynamic and has growth potential in the amount of services that can be configured and deployed, meaning it can grow to accommodate more services other than Sonarqube and Jenkins with minimal changes to the code of the solution.

7.2 Achieved

The main goal of this work was to build a solution capable of deploying infrastructure on-demand, which was achieved having also implemented the removal of such resources used in the deployments. Mechanisms were also implemented to make the deployment process asynchronous, making it so that multiple deployments can happen asynchronously in a single request, achieving better deployment times as a result.

7.3 Limitations and Future Work

The solution developed is limited in terms of technologies, as mentioned before it requires the entry point of the deployment to be a Terraform file, meaning that if the deployment should be done with a different technology some extra logic needs to be added, making the code less maintainable.

Due to time constraints the Jenkins deployment solution is also lacking in security, not requiring any credentials to interact with it, which can be improved in the future in its configuration.

For future work it would also be interesting to add more code analysis tools such as *Gitinspector* and more auxiliary services such as *ElasticSearch*.

7.4 Contributions

All the code produced for the sake of the solution's development is publicly available at the Github repository under an MIT license so that it can be used by other people for their own purposes and to improve the state of infrastructure on demand in the industry.

7.5 Final remarks

This work provided a way for the author to explore some of the concepts that were lectured in the master's degree in Software Engineering, namely infrastructure, DevOps and automating processes.

It provided a challenge to learn new technologies such as Terraform and Ansible in more depth, which are technologies that will prove useful in a professional environment since both of these are trending technologies in the DevOps space.

The work also provided an opportunity to study the **AWS** cloud provider and its services such as **EC2** instances. It allowed the learning in more depth of services such as Sonarqube and Jenkins, and how to configure them in a new instance from scratch.

Bibliography

- Agrawal, Prashant and Neelam Rawat (2019). "Devops, A New Approach To Cloud Development amp; Testing". In: *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. Vol. 1, pp. 1–4. doi: 10.1109/ICICT46931.2019.8977662.
- Arachchi, S.A.I.B.S. and Indika Perera (2018). "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management". In: *2018 Moratuwa Engineering Research Conference (MERCon)*, pp. 156–161. doi: 10.1109/MERCon.2018.8421965.
- Artac, Matej et al. (2017). "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 497–498. doi: 10.1109/ICSE-C.2017.162.
- Bamboo Server (2021). url: <https://www.atlassian.com/software/bamboo>.
- Brikman, Y. (2019). *Terraform: Up & Running: Writing Infrastructure as Code*. O'Reilly Media. isbn: 9781492046875. url: <https://books.google.pt/books?id=57ytDwAAQBAJ>.
- Buddy (2021). url: <https://buddy.works/>.
- CI/CD Pipeline (2021). url: <http://semaphoreci.com/blog/cicd-pipeline>.
- Harness (2021). url: <https://harness.io/>.
- Hochstein, L. and R. Moser (2017). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media. isbn: 9781491979778. url: <https://books.google.pt/books?id=h5YtDwAAQBAJ>.
- Hybrid Cloud DevOps | Cycloid (2021). url: <https://www.cycloid.io/>.
- Internet Live Stats (2022). url: <https://www.internetlivestats.com/>.
- Koen, Peter A et al. (2002). "FuzzyFrontEnd: Effective Methods, Tools, and Techniques LITERATURE REVIEW AND RATIONALE FOR DEVELOPING THE NCD MODEL". In.
- Manvi, Sunilkumar S. and Gopal Krishna Shyam (2014). "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey". In: *Journal of Network and Computer Applications* 41 (1), pp. 424–440. issn: 10958592. doi: 10.1016/J.JNCA.2013.10.004.
- Martin, R.C. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Martin, Robert C. Prentice Hall. isbn: 9780134494166. url: <https://books.google.pt/books?id=8ngAkAEACAAJ>.
- Martin, Robert C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. 1st. Prentice Hall. isbn: 0135974445; 9780135974445.
- Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media. isbn: 9781449319908. url: <https://books.google.pt/books?id=eABpzyTcJNIC>.
- Mohammad, Sikender Mohsienuddin (2018). "Streamlining DevOps Automation For Cloud Applications". In: pp. 955–959.

- Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud*. Safari Books Online. O'Reilly Media. isbn: 9781491924396. url: <https://books.google.pt/books?id=BIhRDAAAQBAJ>.
- Octopus Deploy* (2021). url: <https://octopus.com/>.
- Porcello, E. and A. Banks (2018). *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media. isbn: 9781492044864. url: <https://books.google.pt/books?id=q5NoDwAAQBAJ>.
- Portnoy, M. (2012). *Virtualization Essentials*. Essentials. Wiley. isbn: 9781118240175. url: <https://books.google.pt/books?id=0kgBf8UUsa8C>.
- Turnbull, J. (2014). *The Docker Book: Containerization Is the New Virtualization*. Amazon Digital Services LLC. isbn: 9780988820203. url: <https://books.google.pt/books?id=4xQKBAAAQBAJ>.
- Vagrant* (2021). url: <https://www.vagrantup.com/>.
- What is a Container?* (2019). url: <https://www.docker.com/resources/what-container>.
- What is automation? | IBM* (2021). url: <https://www.ibm.com/topics/automation>.

Appendixes

A AHP Analysis

Pairwise Comparison						
	Software Engineers	Companies	DevOps Engineers	Cybersecurity Engineers		
Software Engineers	1	1/3	1	3		
Companies	3	1	3	5		
DevOps Engineers	1	1/3	1	3		
Cybersecurity Engineers	1/3	1/5	12/30	1		
SOMA	5 1/3	1 7/8	5 1/3	12		
Matriz de comparação normalizada e pesos estimados						
(Divido cada elemento da matriz de comparação pelo total da coluna respectiva)						
	Software Engineers	Companies	DevOps Engineers	Cybersecurity Engineers	Pesos	
Software Engineers	0.1875	0.1786	0.1875	0.2500	0.2009	
Companies	0.5625	0.5357	0.5625	0.4167	0.5193	
DevOps Engineers	0.1875	0.1786	0.1875	0.2500	0.2009	
Cybersecurity Engineers	0.0625	0.1071	0.0625	0.0833	0.0789	
	1.0000	1.0000	1.0000	1.0000		
Teste da consistência						
STEP1	1	1/3	1	3	0.2009	4/5
	3	1	3	5	0.5193	2 1/8
	1	1/3	1	3	0.2009	4/5
	1/3	1/5	1/3	1	0.0789	1/3
STEP2	4.0395					
	4.0802					
	4.0395					
	4.0151					
STEP3	4.0436					
STEP4	Consistência	IC	($\lambda_{\max} - n$)/ $n - 1$	0.01452799264		
STEP5	CR=IC/RI		0.01614221404			
Matrizes com alternativas						
Software Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	
Performance	1	1/5	3	5	5	
Scalability	3	1	5	7	7	
Security	1/3	1/5	1	1	1	
Customization	1/5	12/30	1	1	1	
Cost Efficiency	12/30	1/7	1	1	1	
Total	4.7333	1.6857	11.0000	15.0000	15.0000	
Companies	Performance	Scalability	Security	Customization	Cost Efficiency	
Performance	1	1	1/3	3	1/3	
Scalability	1	1	1/3	3	1/3	
Security	3	3	1	5	1	
Customization	1/3	12/30	1/5	1	1/5	
Cost Efficiency	3	3	1	5	1	
Total	8.3333	8.3333	2.8667	17.0000	2.8667	
DevOps Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	
Performance	1	1	3	1/3	3	
Scalability	1	1	3	1/3	3	
Security	1/3	1/3	1	1/5	1	
Customization	3	3	5	1	5	
Cost Efficiency	1/3	1/3	1	1/5	1	
Total	5.6667	5.6667	13.0000	2.0667	13.0000	
Cybersecurity Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	
Performance	1	1/3	1/5	1/3	1/3	
Scalability	3	1	1/3	1	1	
Security	5	3	1	3	3	
Customization	3	1	1/3	1	1	
Cost Efficiency	3	1	1/3	1	1	
Total	15.0000	6.3333	2.2000	6.3333	6.3333	

Matrizes com alternativas normalizadas

Software Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	Pesos
Performance	2/9	1/8	2/7	1/3	1/3	0.2539
Scalability	5/8	3/5	4/9	4/9	4/9	0.5230
Security	0	1/8	1/9	0	0	0.0827
Customization	0	1/9	1/9	0	0	0.0702
Cost Efficiency	0	1/9	1/9	0	0	0.0702
Total	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Companies	Performance	Scalability	Security	Customization	Cost Efficiency	Pesos
Performance	1/8	1/8	1/9	1/6	1/9	0.1298
Scalability	1/8	1/8	1/9	1/6	1/9	0.1298
Security	3/8	3/8	1/3	2/7	1/3	0.3424
Customization	0	0	0	0	0	0.0557
Cost Efficiency	3/8	3/8	1/3	2/7	1/3	0.3424
Total	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DevOps Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	Pesos
Performance	1/6	1/6	2/9	1/6	2/9	0.1952
Scalability	1/6	1/6	2/9	1/6	2/9	0.1952
Security	0	0	1/9	1/9	1/9	0.0737
Customization	5/9	5/9	3/8	1/2	3/8	0.4624
Cost Efficiency	0	0	1/9	1/9	1/9	0.0737
Total	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Cybersecurity Engineers	Performance	Scalability	Security	Customization	Cost Efficiency	Pesos
Performance	0	0	1/9	0	0	0.0631
Scalability	1/5	1/6	1/7	1/6	1/6	0.1650
Security	1/3	1/2	4/9	1/2	1/2	0.4418
Customization	1/5	1/6	1/7	1/6	1/6	0.1650
Cost Efficiency	1/5	1/6	1/7	1/6	1/6	0.1650
Total	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0.00	0.00	0.59	0.69	1.12	1.24	1.33	1.41	1.48	1.48	1.51	1.48	1.56	1.57	1.58			

Teste da consistência

STEP1					STEP2					STEP3					STEP4					STEP5				
1	1/5	3	5	5	0.2539	1.308926	5.156071	5.039563527	0.009890881809	0.00883114447														
3	1	5	7	7	0.5230	2.681349	5.127055																	
1/3	1/5	1	1	1	0.0827	0.412374	4.988699																	
1/5	1/7	1	1	1	0.0702	0.348642	4.962995																	
1/5	1/7	1	1	1	0.0702	0.348642	4.962995																	
1	1	1/3	3	1/3	0.1298	1.038556	8.000852	7.804778435	0.7011946088	0.626066615														
1	1	1/3	3	1/3	0.1298	1.038556	8.000852																	
3	3	1	5	1	0.3424	2.834676	8.279850																	
1/3	1/3	1/5	1	1/5	0.0557	0.359777	6.462487																	
3	3	1	5	1	0.3424	2.834676	8.279850																	
1	1	3	1/3	3	0.1952	1.258988	45125448	6.262661777	0.3156654443	0.2818441467														
1	1	3	1/3	3	0.1952	1.258988	6.451254																	
1/3	1/3	1	1/5	1	0.0737	0.425906	5.782577																	
3	3	5	1	5	0.4624	3.165323	6.845644																	
1/3	1/3	1	1/5	1	0.0737	0.425906	5.782577																	
1	1/3	1/5	1/3	1/3	0.0631	0.491552	79078418	8.35224398	0.8380609949	0.7482687455														
3	1	1/3	1	1	0.1650	1.452614	8.801596																	
5	3	1	3	3	0.4418	3.342398	7.565646																	
3	1	1/3	1	1	0.1650	1.452614	8.801596																	
3	1	1/3	1	1	0.1650	1.452614	8.801596																	

Matrizes com prioridades globais

	Software Engineers	Companies	DevOps Engineers	Cybersecurity Engineers	Si'
Performance	0.2539	0.1298	0.1952	0.0631	0.2009
Scalability	0.5230	0.1298	0.1952	0.1650	0.5193
Security	0.0827	0.3424	0.0737	0.4418	0.2009
Customization	0.0702	0.0557	0.4624	0.1650	0.0789
Cost Efficiency	0.0702	0.3424	0.0737	0.1650	
Performance	0.1625940966				
Scalability	0.224698601				
Security	0.2440480996				
Customization	0.1489316053				
Cost Efficiency	0.2197275975				

B Deployment creation

```

1 // Create creates a deployment with the inputs given
2 func (s *Service) CreateDeployment(ctx context.Context, nds *dtos.NewDeployments) (
3     deployments []*dtos.CreatedDeployment, warnings, errors []error) {
4     warnings = make([]error, 0)
5     errors = make([]error, 0)
6
7     configurations, errors := s.loadAndVerifyConfigurations(ctx, nds)
8     if len(errors) > 0 {
9         return nil, nil, errors
10    }
11
12    templates, errors := s.loadTemplateFiles()
13    if len(errors) > 0 {
14        return nil, nil, errors
15    }
16
17    tfw := terraform.NewWorker(s.config.TerraformExecPath)
18
19    // For each individual deployment persist its information
20    // and start a goroutine that will do the deployment logic
21    wg := sync.WaitGroup{}
22    deployments = make([]*dtos.CreatedDeployment, 0)
23    for _, nd := range nds.Deployments {
24        for _, ns := range nd.Services {
25            canonical := utils.BuildDeploymentCanonical(nd.UseCase, ns.Service)
26            // was previously validated
27            st, _ := models.TypeString(ns.Service)
28
29            // We persist the deployment with no instances since we
30            // don't know their URLs yet
31            d, warning := s.persistDeployment(ctx, canonical, st, nds.CallbackURL)
32            if warning != nil {
33                warnings = append(warnings, warning)
34                continue
35            }
36
37            // We create the DTO with the count number of instances
38            // on a Pending state
39            cd := &dtos.CreatedDeployment{
40                Canonical: canonical,
41                Type: d.Type.String(),
42                Instances: make([]*dtos.CreatedInstance, ns.Count),
43                CallbackURL: nds.CallbackURL,
44            }
45
46            for i := 0; i < ns.Count; i++ {
47                cd.Instances[i] = dtos.CreatedInstance{
48                    State: models.Pending.String(),
49                }
50            }
51
52            deployments = append(deployments, cd)
53
54            intpl, err := s.loadInterpolator(nd.UseCase, ns.Service, canonical, ns.Count,
55                configurations)
56            if err != nil {
57                errors = append(errors, err)
58                continue
59            }
60
61            // we start goroutines that will do the actual deployments
62            // the deployment consists of:
63            // 1. Read the template files
64            // 2. Replace what needs to be replaced in the template file
65            // 3. Persist the resulting file

```

```

64 // 4. Run the terraform using the terraform worker with the resulting files
65 wg.Add(1)
66 go func(ns dtos.NewService) {
67     defer wg.Done()
68     dctx := context.Background()
69
70     // we retrieve the path to the deployment folder
71     pathToDir, err := utils.BuildDeploymentFolderPath(cd.Canonical)
72     if err != nil {
73         errors = append(errors, fmt.Errorf("could not build deployments path: %w",
err))
74         return
75     }
76
77     // we create the directories for the deployment files
78     err = os.MkdirAll(pathToDir, 0755)
79     if err != nil {
80         errors = append(errors, fmt.Errorf("error creating directory for
deployment %s: %w", cd.Canonical, err))
81         return
82     }
83
84     // we retrieve all the filepaths for the files that need
85     // interpolation and need to be created on the deployment files folder
86     filepaths, ok := templates[ns.Service]
87     if !ok {
88         errors = append(errors, fmt.Errorf("no template founds for the service %s"
, ns.Service))
89         os.RemoveAll(pathToDir)
90         return
91     }
92
93     err = s.interpolateAndWriteFiles(filepaths, canonical, ns.Service, pathToDir
, *intpl)
94     if err != nil {
95         errors = append(errors, err)
96         os.RemoveAll(pathToDir)
97         return
98     }
99
100     cleanup := func() {
101         log.Printf("cleaning up failed deployment %s\n", cd.Canonical)
102         os.RemoveAll(pathToDir)
103         s.DeleteDeployment(dctx, cd.Canonical)
104     }
105
106     // once all files are interpolated and created we do the
107     // deployment logic using the terraform worker
108     err = tfw.Deploy(pathToDir)
109     if err != nil {
110         errors = append(errors, fmt.Errorf("error executing deployment %s: %w", cd
.Canonical, err))
111         cleanup()
112         return
113     }
114
115     // with the deployment done we fetch the outputs
116     // which contain the Public IPs of the deployments
117     deploymentURLs, err := tfw.GetIPs(pathToDir)
118     if err != nil {
119         errors = append(errors, fmt.Errorf("error retrieving the public IPs of the
deployment %s: %w", cd.Canonical, err))
120         cleanup()
121         return
122     }
123
124     // create the instances with their public IP and a Pending state

```

```

125     instances := make([]db.Instance, len(deploymentURLs))
126     for i, durl := range deploymentURLs {
127         dbi := db.Instance{
128             DeploymentCanonical: d.Canonical,
129             State:                uint(models.Pending),
130             URL:                   durl,
131         }
132         instances[i] = dbi
133     }
134
135     // persist the new instances
136     err = s.repo.BatchCreateInstances(dctx, instances)
137     if err != nil {
138         errors = append(errors, fmt.Errorf("error updating deployment %s's URL: %w",
139             cd.Canonical, err))
140         cleanup()
141         return
142     }
143 }
144 }
145
146 // Wait for all the deployment tasks to be over
147 // and check for errors in order for them to be logged
148 go func() {
149     wg.Wait()
150     for _, err := range errors {
151         log.Println(err)
152     }
153     for _, w := range warnings {
154         log.Println(w)
155     }
156 }()
157 return deployments, warnings, nil
158 }

```

Listing 7.1: Create deployment

C Sonarqube template files

```

1 terraform {
2     required_providers {
3         aws = {
4             source = "hashicorp/aws"
5             version = "3.74.0"
6         }
7     }
8 }
9
10 provider "aws" {
11     profile = "default"
12     region = "eu-west-1"
13 }
14
15 resource "aws_key_pair" "(${ name })" {
16     key_name = "(${ name })-key-pair"
17     public_key = file("${var.public_key}")
18 }
19
20 resource "aws_security_group" "(${ name })" {
21     name = "(${ name })-sg"
22
23     ingress {
24         description = "Access Sonarqube"
25         from_port = 9000

```

```

26     to_port      = 9000
27     protocol     = "tcp"
28     cidr_blocks  = ["0.0.0.0/0"]
29 }
30
31 ingress {
32     description = "SSH from everywhere"
33     from_port   = 22
34     to_port     = 22
35     protocol    = "tcp"
36     cidr_blocks = ["0.0.0.0/0"]
37 }
38
39 egress {
40     from_port   = 0
41     to_port     = 0
42     protocol    = "-1"
43     cidr_blocks = ["0.0.0.0/0"]
44 }
45
46 tags = {
47     Name = "($ name $)-sg"
48 }
49 }
50
51 resource "aws_instance" "($ name $)" {
52     count = "($ count $)"
53     // retrieved from https://cloud-images.ubuntu.com/locator/ec2/
54     // and it refers to an eu-west-1 Ubuntu 18.04 LTS amd64 machine
55     ami           = "ami-0ce48dd7b483b8402"
56     instance_type = "t3.large"
57     key_name      = aws_key_pair.($ name $).key_name
58
59     security_groups = [aws_security_group.($ name $).name]
60
61     tags = {
62         "Name" = "($ name $)-${count.index}"
63     }
64 }
65
66 resource "null_resource" "run_ansible" {
67     count = "($ count $)"
68
69     connection {
70         type = "ssh"
71         user = "ubuntu"
72         host = aws_instance.($ name $)[count.index].public_ip
73         private_key = "${file("${var.private_key}")}"
74         agent = false
75         timeout = "3m"
76     }
77
78     provisioner "file" {
79         source = "./files"
80         destination = "/tmp"
81     }
82
83     provisioner "file" {
84         source = "./ansible"
85         destination = "/tmp"
86     }
87
88     provisioner "remote-exec" {
89         inline = [
90             "chmod +x /tmp/files/run-ansible.sh",
91             "chmod +x /tmp/files/db-setup.sh",
92             "/tmp/files/run-ansible.sh"

```

```

93     ]
94   }
95 }
96
97 output "public_ips" {
98   description = "Public IPs of the instances"
99   value = "${aws_instance.($ name $).*.public_ip}"
100 }

```

Listing 7.2: main.tf

```

1  ---
2
3  - name: Configure Sonarqube
4    hosts: local
5    connection: local
6    become: true
7    tasks:
8      - name: Update Repository Cache
9        retries: 2
10       delay: 10
11       apt:
12         update_cache: true
13
14     - name: Install PostgreSQL
15       apt:
16         name: "{{ item }}"
17         state: present
18       with_items:
19         - acl
20         - postgresql
21         - postgresql-contrib
22         - libpq-dev
23         - python3-psycopg2
24
25     - name: Install SonarQube Requirements
26       apt:
27         name: "{{ item }}"
28         state: present
29       with_items:
30         - openjdk-11-jdk
31         - fontconfig-config
32         - libfreetype6
33         - zip
34         - unzip
35
36     - name: Strip carriage returns from scripts to ensure this works from Windows
37       VMs
38       replace:
39         path: "/tmp/files/db-setup.sh"
40         regexp: "[\r]$"
41         replace: ""
42
43     - name: Create postgres user and DB for SonarQube
44       become: yes
45       become_user: postgres
46       command: "/tmp/files/db-setup.sh {{ sonar_db_pass }}"
47
48     - name: Create the sonar user for running the SonarQube services
49       user:
50         name: sonar
51         comment: System user for running SonarQube
52
53     - name: Download SonarQube
54       get_url:
55         url: "{{ sonar_download_url }}"
56         dest: "/srv/sonarqube-{{ sonar_version }}.zip"

```

```

56
57 - name: Extract SonarQube
58   unarchive:
59     src: "/srv/sonarqube-{{ sonar_version }}.zip"
60     dest: "/srv"
61     copy: no
62     owner: sonar
63
64 - name: Link this version of sonarqube as the server SonarQube version
65   file:
66     src: "/srv/sonarqube-{{ sonar_version }}"
67     dest: "/srv/sonarqube"
68     state: link
69     owner: sonar
70
71 - name: Configure SonarQube Port
72   lineinfile:
73     path: "/srv/sonarqube/conf/sonar.properties"
74     regexp: "^sonar.web.port="
75     insertafter: "^#sonar.web.port="
76     line: "sonar.web.port=9000"
77
78 - name: Configure SonarQube DB username
79   lineinfile:
80     path: "/srv/sonarqube/conf/sonar.properties"
81     regexp: "^sonar.jdbc.username="
82     insertafter: "^#sonar.jdbc.username="
83     line: "sonar.jdbc.username={{ sonar_db_user }}"
84
85 - name: Configure SonarQube DB password
86   lineinfile:
87     path: "/srv/sonarqube/conf/sonar.properties"
88     regexp: "^sonar.jdbc.password="
89     insertafter: "^#sonar.jdbc.password="
90     line: "sonar.jdbc.password={{ sonar_db_pass }}"
91
92 - name: Configure SonarQube DB connection string
93   lineinfile:
94     path: "/srv/sonarqube/conf/sonar.properties"
95     regexp: "sonar.jdbc.url=jdbc:postgresql://localhost/sonar"
96     insertafter: "^#sonar.jdbc.url=jdbc:postgresql://localhost/sonar"
97     line: "sonar.jdbc.url=jdbc:postgresql://localhost/sonar"
98
99 - name: Configure SonarQube to run as the sonar user
100   lineinfile:
101     path: "/srv/sonarqube/bin/linux-x86-64/sonar.sh"
102     regexp: "RUN_AS_USER=sonar"
103     insertafter: "^#RUN_AS_USER="
104     line: "RUN_AS_USER=sonar"
105
106 - name: Install the sonarqube plugins
107   get_url:
108     url: "{{ item }}"
109     dest: /srv/sonarqube/extensions/plugins
110   with_items:
111     - "{{ groups['sonar_plugins'] }}"
112
113 - name: Set Elasticsearch requirements
114   sysctl:
115     name: vm.max_map_count
116     value: 524288
117     state: present
118     reload: yes
119
120 - name: Copy the SonarQube service configuration file
121   copy:
122     src: "/tmp/files/sonarqube.service"

```

```

123     dest: "/etc/systemd/system/sonarqube.service"
124
125   - name: Configure OS security limits for the sonar user
126     copy:
127       src: "/tmp/files/sonarqube.limits"
128       dest: "/etc/security/limits.d/99-sonarqube.conf"
129
130   - name: Configure kernel level limits for Elasticsearch
131     copy:
132       src: "/tmp/files/sonarqube.sysctl"
133       dest: "/etc/sysctl.d/99-sonarqube.conf"
134
135   - name: Enable the SonarQube service
136     systemd:
137       state: started
138       enabled: yes
139       daemon_reload: yes
140       name: sonarqube

```

Listing 7.3: ansible.yaml

```

1 [local]
2 localhost
3
4 [local:vars]
5 sonar_version=($ version $)
6 sonar_download_url=($ download_url $)
7 sonar_db_name=sonar
8 sonar_db_user=sonar
9 sonar_db_pass=($ db_pass $)
10
11 [sonar_plugins]
12 ($ plugin_urls $)

```

Listing 7.4: inventory

D Jenkins template files

```

1 terraform {
2   required_providers {
3     aws = {
4       source = "hashicorp/aws"
5       version = "3.74.0"
6     }
7   }
8 }
9
10 provider "aws" {
11   profile = "default"
12   region = "eu-west-1"
13 }
14
15 resource "aws_key_pair" "($ name $)" {
16   key_name = "($ name $)-key-pair"
17   public_key = file("${var.public_key}")
18 }
19
20 resource "aws_security_group" "($ name $)" {
21   name = "($ name $)-sg"
22
23   ingress {
24     description = "Access Jenkins"
25     from_port = 8080
26     to_port = 8080

```



```

27     protocol      = "tcp"
28     cidr_blocks   = ["0.0.0.0/0"]
29 }
30
31 ingress {
32     description = "SSH from everywhere"
33     from_port   = 22
34     to_port     = 22
35     protocol    = "tcp"
36     cidr_blocks = ["0.0.0.0/0"]
37 }
38
39 egress {
40     from_port   = 0
41     to_port     = 0
42     protocol    = "-1"
43     cidr_blocks = ["0.0.0.0/0"]
44 }
45
46 tags = {
47     Name = "($ name $)-sg"
48 }
49 }
50
51 resource "aws_instance" "($ name $)" {
52     count = "($ count $)"
53     // retrieved from https://cloud-images.ubuntu.com/locator/ec2/
54     // and it refers to an eu-west-1 Ubuntu 18.04 LTS amd64 machine
55     ami           = "ami-0ce48dd7b483b8402"
56     instance_type = "t2.micro"
57     key_name      = aws_key_pair.($ name $).key_name
58
59     security_groups = [aws_security_group.($ name $).name]
60
61     tags = {
62         "Name" = "($ name $)-${count.index}"
63     }
64 }
65
66 resource "null_resource" "run_ansible" {
67     count = "($ count $)"
68
69     connection {
70         type = "ssh"
71         user = "ubuntu"
72         host = aws_instance.($ name $)[count.index].public_ip
73         private_key = "${file("${var.private_key}")}"
74         agent = false
75         timeout = "3m"
76     }
77
78     provisioner "file" {
79         source = "./files"
80         destination = "/tmp"
81     }
82
83     provisioner "file" {
84         source = "./ansible"
85         destination = "/tmp"
86     }
87
88     provisioner "remote-exec" {
89         inline = [
90             "chmod +x /tmp/files/run-ansible.sh",
91             "/tmp/files/run-ansible.sh"
92         ]
93     }

```

```

94 }
95
96 output "public_ips" {
97   description = "Public IPs of the instances"
98   value = "${aws_instance.${name}.public_ip}"
99 }

```

Listing 7.5: main.tf

```

1  ---
2
3  - name: Configure Jenkins
4    hosts: local
5    connection: local
6    become: true
7    tasks:
8      - name: Ensure dependencies are installed.
9        apt:
10         name:
11           - curl
12           - apt-transport-https
13           - gnupg
14           - 'fontconfig'
15           - 'openjdk-11-jre'
16         state: present
17
18      - name: Add Jenkins apt repository key.
19        apt_key:
20         url: "https://pkg.jenkins.io/debian-stable/jenkins.io.key"
21         state: present
22
23      - name: Add Jenkins apt repository.
24        apt_repository:
25         repo: "deb http://pkg.jenkins.io/debian-stable binary/"
26         state: present
27         update_cache: true
28
29      - name: Download specific Jenkins version.
30        get_url:
31         url: "https://mirrors.jenkins.io/debian/jenkins-{{ jenkins_version }}_all.deb"
32         dest: "/tmp/jenkins-{{ jenkins_version }}_all.deb"
33
34      - name: Install our specific version of Jenkins.
35        apt:
36         deb: "/tmp/jenkins-{{ jenkins_version }}_all.deb"
37         state: present
38
39      - name: Ensure Jenkins is installed.
40        apt:
41         name: jenkins
42         state: present
43
44      # Configure Jenkins init settings.
45      - include_tasks: settings.yaml
46
47      # Make sure Jenkins starts, then configure Jenkins.
48      - name: Ensure Jenkins is started and runs on startup.
49        service: name=jenkins state=started enabled=yes
50
51      - name: Wait for Jenkins to start up before proceeding.
52        uri:
53         url: "http://localhost:8080/cli/"
54         method: GET
55         return_content: "yes"
56         timeout: 5
57         body_format: raw

```

```

58     follow_redirects: "no"
59     status_code: 200,403
60     register: result
61     until: (result.status == 403 or result.status == 200) and (result.content.find
("Please wait while") == -1)
62     retries: "5"
63     delay: "60"
64     changed_when: false
65     check_mode: false
66
67 - name: Get the jenkins-cli jarfile from the Jenkins server.
68   get_url:
69     url: "http://localhost:8080/jnlpJars/jenkins-cli.jar"
70     dest: "/opt/jenkins-cli.jar"
71     register: jarfile_get
72     until: "'OK' in jarfile_get.msg or '304' in jarfile_get.msg or 'file already
exists' in jarfile_get.msg"
73     retries: 5
74     delay: 10
75     check_mode: false
76
77 - name: Remove Jenkins security init scripts after first startup.
78   file:
79     path: "{{ jenkins_home }}/init.groovy.d/basic-security.groovy"
80     state: absent
81
82 # Update Jenkins and install configured plugins.
83 - include_tasks: plugins.yaml

```

Listing 7.6: ansible.yaml

```

1 ---
2 - name: Check if jenkins_init_file exists.
3   stat:
4     path: "{{ jenkins_init_file }}"
5   register: jenkins_init_file_stat
6
7 - name: Ensure jenkins_init_file exists.
8   file:
9     path: "{{ jenkins_init_file }}"
10    state: touch
11    mode: 0644
12    when: not jenkins_init_file_stat.stat.exists
13
14 - name: Modify variables in init file.
15   lineinfile:
16     dest: "{{ jenkins_init_file }}"
17     insertafter: '^{{ item.option }}='
18     regexp: '^{{ item.option }}=\\\"{{ item.option }} '
19     line: '{{ item.option }}=\"${ item.option } {{ item.value }}\"'
20     state: present
21     mode: 0644
22   with_items:
23     - option: JAVA_ARGS
24       value: "{{ jenkins_java_options }}"
25   register: jenkins_init_prefix
26
27 - name: Ensure jenkins_home {{ jenkins_home }} exists.
28   file:
29     path: "{{ jenkins_home }}"
30     state: directory
31     owner: jenkins
32     group: jenkins
33     mode: u+rw
34     follow: true
35
36 - name: Set the Jenkins home directory.

```

```

37 lineinfile:
38   dest: "{{ jenkins_init_file }}"
39   regexp: '^JENKINS_HOME=.*'
40   line: 'JENKINS_HOME={{ jenkins_home }}'
41   mode: 0644
42   register: jenkins_home_config
43
44 - name: Immediately restart Jenkins on init config changes.
45   service: name=jenkins state=restarted
46   when: jenkins_init_prefix.changed

```

Listing 7.7: settings.yaml

```

1 ---
2 # Update Jenkins so that plugin updates don't fail.
3 - name: Create Jenkins updates directory.
4   file:
5     path: "{{ jenkins_home }}/updates"
6     state: directory
7     owner: jenkins
8     group: jenkins
9     mode: 0755
10
11 - name: Download current plugin updates from Jenkins update site.
12   get_url:
13     url: "https://updates.jenkins.io/update-center.json"
14     dest: "{{ jenkins_home }}/updates/default.json"
15     owner: jenkins
16     group: jenkins
17     mode: 0440
18     changed_when: false
19     register: get_result
20     until: get_result is success
21     retries: 3
22     delay: 2
23
24 - name: Remove first and last line from json file.
25   replace: # noqa 208
26     path: "{{ jenkins_home }}/updates/default.json"
27     regexp: "1d;$d"
28
29 - name: Install Jenkins plugins using password.
30   jenkins_plugin:
31     name: "{{ item }}"
32     jenkins_home: "{{ jenkins_home }}"
33     state: "latest"
34     timeout: "30"
35     updates_url: "https://updates.jenkins.io"
36     url: "http://localhost:8080"
37     with_dependencies: "true"
38     with_items: "{{ groups['jenkins_plugins'] }}"
39     retries: 3
40     delay: 2
41
42 - name: Restart Jenkins
43   service:
44     name: jenkins
45     state: restarted

```

Listing 7.8: plugins.yaml

```

1 [local]
2 localhost
3
4 [local:vars]
5 jenkins_version=($ version $)

```

```
6 jenkins_home=/var/lib/jenkins
7 jenkins_init_file=/etc/default/jenkins
8 jenkins_java_options=-Djenkins.install.runSetupWizard=false -Dhudson.security.csrf.
  DefaultCrumbIssuer.EXCLUDE_SESSION_ID=true
9
10 [jenkins_plugins]
11 git
12 bitbucket
13 bitbucket-build-status-notifier
14 build-timeout
15 credentials
16 branch-api
17 ldap
18 credentials-binding
```

Listing 7.9: inventory