João Miguel Mendes Gonçalves

Automatic Deployment Solution for Multi-Cloud Environments

Dissertation in the context of the Master in Informatics Engineering, specialization in Communications, Services and Infrastructures, advised by Prof. Dr. Karima Velasquez and Prof. Dr. David Abreu, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September, 2023

Department of Informatics Engineering

João Miguel Mendes Gonçalves

Automatic Deployment Solution for Multi-Cloud Environments

Dissertation in the context of the Master in Informatics Engineering, specialization in Communications, Services and Infrastructures, advised by Prof.Dr. Karima Velasquez and Prof. Dr. David Abreu, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September, 2023

This work is funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

Acknowledgements

To all the professors of the Department of Informatics Engineering of University of Coimbra, who helped and provided learning over this master's degree period.

To my supervisors Professor Karima Velasquez and Professor David Abreu for their availability, collaboration and guidance throughout both semesters. A special thanks to Professor Marília Curado for presenting and suggesting this internship proposal.

Finally, to my family and friends for the support, encouragement and guidance through my academic journey.

Abstract

The following report develops the subject of Automatic Deployment Solution for Multi-Cloud Environments. The main objective of this solution is to perform the automatic deployment of an application to a Multi-Cloud environment each time the application is updated. Automating the deployment process of the application will replace manual and time-spending procedures that need to be done to perform the application's deployment. A Waterfall Model with Staged Delivery methodology was used for the development of this work, by dividing the development process into incremental steps, thus allowing the consolidation of concepts and also allowing defining stages to perform the implementation and validation of the solution. The work starts with a literature review on Cloud Computing concepts in order to gain knowledge of Multi-Cloud environments. Then, a theoretical analysis of Multi-Cloud management tools and Automatic Deployment tools is performed to determine the tools to be used in the solution. After the theoretical segment, follows the practical segment of the work, where the solution is designed, implemented and validated. The general architecture of the solution is presented, transmitting that the general idea of the solution is to combine a Multi-Cloud management tool with an Automatic Deployment tool to perform the automatic deployment of an application to the Cloud. Then, the incremental steps done in order to develop the solution are presented, by describing their definition and implementation. The validation of the Automatic Deployment Solution is performed in the last step, by using a Multi-Cloud management tool with an Automatic Deployment tool to automatically deploy an application to the Cloud. Finally, the conclusions taken from the work done are presented, together with the following steps to be done regarding the usage of this solution.

Keywords

Automatic Deployment, Automation, Cloud, Multi-Cloud Environments, Multi-Cloud Management Tools.

Resumo

O presente relatório desenvolve o tema Solução de Implementação Automática em Ambientes *Multi-Cloud*. O objetivo principal desta solução é realizar a implementação automática de uma aplicação para um ambiente *Multi-Cloud* sempre que a aplicação seja atualizada. Automatizar o processo de implementação da aplicação vai substituir processos manuais que consomem bastante tempo que precisão de ser feitos para realizar a implementação da aplicação. A metodologia *Waterfall Model with Staged Delivery* é utilizada para o desenvolvimento deste trabalho, ao dividir o processo de desenvolvimento em passos incrementais, permitindo assim a consolidação de conceitos e a definição de etapas para realizar a implementação e validação da solução. O trabalho começa com uma análise literária sobre os conceitos de Computação na *Cloud* de modo a adquirir conhecimento sobre ambientes *Multi-Cloud*. De seguida, é realizada uma análise teórica sobre Ferramentas de Gestão de *Multi-Clouds* e Ferramentas de Implementação Automática, para determinar quais as ferramentas a serem utilizadas na solução. Depois da componente teórica, segue-se a componente prática do trabalho, onde a solução é desenhada, implementada e validada. É apresentada a arquitetura geral da solução, transmitindo que a ideia geral da solução é combinar uma Ferramenta de Gestão de *Multi-Cloud* e uma Ferramenta de Implementação Automática para realizar a implementação automática de uma aplicação para a *Cloud*. De seguida, são apresentados os passos incrementais realizados para desenvolver a solução, ao descrever a sua definição e implementação. A validação da Solução de Implementação Automática é realizada no último passo, ao utilizar uma Ferramenta de Gestão de *Multi-Cloud* e uma Ferramenta de Implementação Automática para implementar automaticamente uma aplicação na *Cloud*. Para concluir, são apresentadas as conclusões retiradas do trabalho realizado e são indicados os próximos passos a serem feitos relativamente à utilização desta solução.

Palavras-chave

Implementação Automática, Automação, *Cloud*, Ambientes *Multi-Cloud*, Ferramentas de Gestão de *Multi-Clouds*.

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Problem . . . 3
1.3 Objectives . . . 3
1.4 Contributions . . . 4
1.5 Methodology . . . 4
1.6 Working Plan . . . 5
1.7 Document Structure . . . 6

2 Cloud Computing Paradigm **7**
2.1 Cloud Computing Characteristics . . . 8
2.2 Cloud Service Models . . . 8
2.2.1 Infrastructure as a Service . . . 9
2.2.2 Platform as a Service . . . 10
2.2.3 Software as a Service . . . 10
2.2.4 Everything as a Service . . . 10
2.3 Deployment Models . . . 11
2.3.1 Private Clouds . . . 11
2.3.2 Public Clouds . . . 12
2.3.3 Community Clouds . . . 12
2.3.4 Hybrid Clouds . . . 13
2.3.5 Multi-Clouds . . . 16
2.4 Multi-Cloud vs. Hybrid Cloud . . . 17
2.5 Summary . . . 19

3 Management Tools for Multi and Hybrid Clouds **21**
3.1 Management Tools Analysis . . . 21
3.1.1 Terraform . . . 22
3.1.2 Anthos . . . 24
3.1.3 Elastic Kubernetes Service (EKS) . . . 27
3.1.4 Ansible . . . 29
3.2 Tools Comparison . . . 31
3.3 Summary . . . 34

4 Automatic Deployment Tools **35**
4.1 Continuous Integration / Continuous Delivery . . . 35
4.2 Kubernetes . . . 36
4.3 GitHub Actions . . . 37
4.4 Summary . . . 38

5 Problem Analysis **41**
5.1 Problem Description 41
5.2 Requirements Analysis 42
5.2.1 Functional Requirements 42
5.2.1.1 User Stories 42
5.2.1.2 High-level requirements 43
5.2.2 Non-Functional Requirements 43
5.3 Risk Analysis 44
5.4 Summary 46

6 Proposed Solution **47**
6.1 General Architecture 47
6.2 GitHub Actions Workflows 48
6.3 Scenarios 53
6.3.1 Scenario 1 55
6.3.1.1 Description 55
6.3.1.2 Implementation and validation 56
6.3.2 Scenario 2 58
6.3.2.1 Description 58
6.3.2.2 Implementation and validation 59
6.3.3 Scenario 3 63
6.3.3.1 Description 63
6.3.3.2 Implementation and validation 64
6.3.4 Scenario 4 65
6.3.4.1 Description 65
6.3.4.2 Implementation and validation 66
6.4 Summary 69

7 Conclusion **71**

Acronyms

ALB Altice Labs.

API Application Programming Interface.

AWS Amazon Web Services.

BYON Bring Your Own Node.

CAPEX Capital Expenditure.

CDNaaS Content Distribution Network as a Service.

CI/CD Continuous Integration/Continuous Delivery.

CLI Command Line Interface.

CMP Cloud Management Platform.

CPU Central Processing Unit.

CSP Cloud Service Provider.

DaaS Desktop as a Service.

DBaaS Database as a Service.

DEI Department of Informatics Engineering.

DNS Domain Name System.

DNSaaS DNS as a Service.

EC2 Elastic Compute Cloud.

ECR Elastic Container Registry.

EKS Elastic Kubernetes Service.

ELB Elastic Load Balancer.

ENI Elastic Network Interface.

GB Gigabytes.

GCP Google Cloud Platform.

GKE Google Kubernetes Engine.

HCL Hashicorp Configuration Language.

HTML Hyper Text Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPs Hypertext Transfer Protocol Secure.

IaaS Infrastructure as a Service.

IAC Infrastructure as Code.

IAM Identity and Access Management.

IP Internet Protocol.

IPN Pedro Nunes Institute.

IT Information Technology.

IT - Aveiro Telecommunications Institute of Aveiro.

JSON JavaScript Object Notation.

KVM Kernel-based Virtual Machine.

LAN Local Area Network.

LBaaS Load Balancing as a Service.

LTS Long Term Support.

MaaS Monitoring as a Service.

MoSCoW Must have, Should have, Can have, Won't have.

NIST National Institute of Standards and Technology.

OPEX Operational Expenditure.

OS Operating System.

PaaS Platform as a Service.

SaaS Software as a Service.

SOA Service-Oriented Architecture.

SSH Secure Shell.

UC University of Coimbra.

URL Uniform Resource Locator.

VCS Version Control System.

VM Virtual Machine.

VPC Virtual Private Cloud.

VPN Virtual Private Network.

WAN Wide Area Network.

WB Web-Based.

WSL Windows Subsystem for Linux.

XaaS Everything as a Service.

YAML Yet Another Markup Language.

List of Figures

1.1 Gantt chart for 1st semester 5
1.2 Gantt chart for 2nd semester 5

2.1 Cloud model organization 7
2.2 Cloud service models 9
2.3 Hybrid architecture patterns 15

3.1 Terraform workflow 23
3.2 Anthos architecture diagram 26
3.3 Amazon EKS setup process 28
3.4 Amazon EKS architecture 28
3.5 Ansible's architecture 30

4.1 CI/CD flow 36

6.1 General Architecture 48
6.2 main.yml events 49
6.3 main.yml *test* job 49
6.4 main.yml *deployment* job 50
6.5 delete_scenario.yml 51
6.6 start_minikube_cluster.yml 52
6.7 delete_minikube_cluster.yml 52
6.8 stop_minikube_cluster.yml 53
6.9 Scenario 1 56
6.10 Mongo Express browser page 58
6.11 Scenario 2 59
6.12 Scenario 2 workflows in GitHub 61
6.13 Mongo Express browser page 61
6.14 Self-Hosted Runners of Scenario 2 62
6.15 Scenario 3 63
6.16 Self-Hosted Runners of Scenario 3 64
6.17 Scenario 3 workflows in GitHub 65
6.18 Scenario 3 Mongo Express browser page 65
6.19 Scenario 4 66
6.20 Scenario 4 Ansible Playbook execution 68
6.21 Scenario 4 workflows in GitHub 69
6.22 Scenario 4 deployed in a Kubernetes cluster running in DEI's Cloud 69

List of Tables

2.1 Multi-Cloud vs. Hybrid Cloud 19

3.1 Multi-Cloud and Hybrid Cloud environment management tools comparison 33

5.1 User Stories 42
5.2 Risk analysis 45

6.1 Kubernetes objects 55

Chapter 1

Introduction

This report was developed within the scope of the Dissertation in Communications, Services and Infrastructures, in order to create an Automatic Deployment Solution for Multi-Cloud Environments.

The concept of "Cloud" has gained more relevance and visibility throughout the years. To an ordinary person, the Cloud is mainly associated with the storage of files and photos on the Internet, so they can backup the files that are on their personal devices, and so they can access those files and photos anytime and anywhere. This idea of the Cloud defined by ordinary people is not wrong but is barely anything regarding the Cloud's full capabilities and deployment models.

Cloud computing consists of having access to a wide range of computing resources and services through a network connection. These resources and services can be gathered and installed by the final user, applying a more private approach, or rented by an external provider that is responsible for managing and building the infrastructure, resources, and services that are going to be provided, applying a more public and pay-as-you-use model.

More recently, Cloud computing has become an integral part of many businesses. For instance, from a startup company perspective, it may be more beneficial to adopt a pay-as-you-use model at the beginning of their business, because by doing so they do not have to worry about guessing the capacity they might need, do not have to spend money buying the equipment and running and maintaining the infrastructure and datacenters, they only have to pay for the resources they use, and finally, they can go global in a matter of minutes.

At a company's most advanced stage of their business when they already have profit and funding, they can opt to migrate to a Private model, becoming responsible for all the infrastructure, resources, and services needed. This approach might be more expensive but allows companies to be fully independent and have total control over their infrastructure.

Another possible scenario is when companies choose a Hybrid or Multimodel by merging Cloud models, so they can have access and combine the best characteristics, features, and solutions that each Cloud provides.

Nowadays, many companies may tend to use a Multi-Cloud model approach instead of the other available Cloud models. This is due to the fact that by opting for a Multi-Cloud model companies can combine the best of other models and also avoid a vendor lock-in restriction, which may affect companies that opt for Public and Hybrid Cloud models.

Moreover, another reason why companies move to either a Hybrid or Multi-Cloud model is that certain types of health and financial data have restrictions as to where they must be stored. In addition, the rise of new management technologies, like Cloud management tools, makes it possible to manage these complex environments [47].

The main benefit of using Multi-Cloud management tools is the ability to leverage the best features of different Clouds. Some examples of Multi-Cloud management tools are Terraform [64], Google Anthos [27], Amazon Elastic Kubernetes Service (EKS) [12], and Ansible [6]. In spite of the management tools available, companies find it difficult to choose the best management tool for their business and needs.

The process of choosing a management tool may become difficult and indecisive in some cases. This process is usually composed of the evaluation of metrics, characteristics, and functionalities of the available tools, and the assessment of the end-user requirements. The choice of the management tool may vary from end user to end user, because each user may have different requirements and a tool may be better suited for one scenario than another.

Management tools can be used together with other tools, such as GitHub Actions [24], in order to convert some manual and time-spending procedures into automatic processes. The main objective of this work is to design and develop a framework for automatic deployment in Cloud and Multi-Cloud environments.

1.1 Motivation

The work presented in this report is incorporated into the POWER project. This project aims to create an innovative portfolio of products and services, mostly based on Cloud and cognitive technologies. It is structured into the following five subprojects: Subproject 1: New Technology Integration; Subproject 2: Future Networks; Subproject 3: Future Operations; Subproject 4: Future Services; and Subproject 5: Data Business and 360 Monetization. The partners that work together in the POWER project are the University of Coimbra (UC) [4], Pedro Nunes Institute (IPN) [3], the Telecommunications Institute of Aveiro (IT - Aveiro) [2], and Altice Labs (ALB) [1]. While the UC, IPN and IT - Aveiro are mostly oriented to the research field, ALB is oriented to the production of solutions for other companies and for the market, but also supports the research field in order to have better results for their solutions.

All the researchers who work on the subprojects need to perform experiments in order to validate their proposals and hypotheses. Before starting the execution

of those experiments, the researchers need to first configure the testbeds that will be used. However, the goal of the researchers is to perform their experiments and not to waste time investigating how to deploy the testbeds and the configurations needed to do so. Besides being used for the POWER's subprojects' testbeds, this framework can also be used as a basis for the deployment of projects that are being developed by ALB, such as FOCUS and OpenPlay.

It was determined that the deployment of the testbeds was going to be performed in a Multi-Cloud environment due to ALB's interest in the advantages of this type of Cloud deployment model. Some advantages that satisfy ALB's interests are the possibility of using multiple Cloud providers thus avoiding a vendor lock-in, the higher amount of availability zones that are covered, and the ease of managing some business costs.

1.2 Problem

Taking into consideration the motivation previously mentioned, the problem identified is how to perform an automatic deployment in different Clouds. In order to accomplish automatic deployment, a Continuous Integration/Continuous Delivery (CI/CD) tool needs to be used to automate some steps of the deployment that were being performed manually and directly in the terminal of the machine where the deployment was being executed. Without an automatic deployment, the user had to deploy all the components of his testbeds manually. This might not be a problem on a smaller scale testbed, but on a larger scale, it can become impractical and time-consuming, thus the need for automatic deployment rather than manual deployment. Also, another point to consider is that the user might not have knowledge of deployment tools, so performing an automatic deployment would also solve this issue.

1.3 Objectives

The main objective of this work is to design and develop an automatic deployment solution for Cloud and Multi-Cloud environments, which incorporates the use of management tools for Multi-Clouds and a CI/CD tool to automate processes. This objective aims to the automatic deployment of applications both on-premises and on a Cloud Service Provider (CSP).

To accomplish this objective, a combination of specific goals was set to be achieved first. The first goal is to identify the requirements to distinguish a Multi-Cloud from a Hybrid Cloud. The second goal consists of the identification and analysis of the management tools used for the management of the environment used. Finally, the third goal was to validate the automatic deployment using some scenarios and Clouds provided by some of the project's members, namely ALB and the Department of Informatics Engineering (DEI) of the UC.

1.4 Contributions

Taking into consideration the objectives and goals described above, this report has produced the following contributions:

- An analysis on the Cloud computing paradigm, focusing on its characteristics, service models, deployment models and in the differences between Multi-Clouds and Hybrid Clouds. This analysis is presented in Chapter 2.
- The identification of currently existing tools for Multi-Cloud and Hybrid Cloud environments management, which were provided in the initial phase of this work. The tools identified are presented in Section 3.1.
- A comparative analysis between the Multi-Cloud and Hybrid Cloud environments management tools that were identified, considering a set of metrics and characteristics related to Multi-Cloud support. This contribution is portrayed in Section 3.2.
- The proposal of a solution for automatic deployment in Cloud and Multi-Cloud environments. This proposition is presented in Chapter 6.
- The configuration of workflows that can be used as pipeline templates for CI/CD. The workflows are explained in Section 6.3.
- The documentation of the steps followed to deploy each scenario provided by ALB. The process followed is presented in Section 6.3.

1.5 Methodology

A Waterfall Model with Staged Delivery approach was used for this internship's work. This methodology was chosen because the process of developing the final version of the solution is divided into incremental steps, thus allowing the definition and consolidation of concepts and also allowing defining stages to perform the implementation and validation of the solution. The flexibility brought by the Staged Delivery allows solving the problems associated with the Waterfall Model, such as not having feedback on the work done and cascading bugs. The feedback on each incremental step will come from the POWER and internship periodic meetings. Furthermore, due to the feedback provided, the Staged Delivery will also allow adjusting the incremental steps if needed. In this way, it is granted that the final version of the solution can meet and satisfy the current interests of the POWER project instead of only considering the initial plan. In addition, the tests performed in each incremental step will allow to detect potential cascading bugs.

1.6 Working Plan

The work presented in this report started with a literature review during the 1st semester. The review consisted of first studying Cloud Computing concepts in order to get knowledge on Multi-Cloud environments, presented in Chapter 2, and then performing a theoretical analysis on Multi-Cloud management tools, presented in Chapter 3. After the literature review was concluded, it was planned the technological work to be performed during the 2nd semester. Figure 1.1 presents the summary of the work done during the 1st semester.

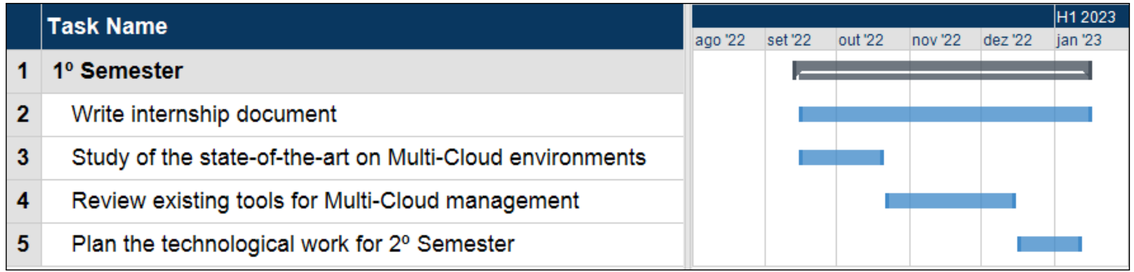


Figure 1.1: Gantt chart for 1st semester

The 2nd semester consisted of the practical segment of this work. This segment was divided into four incremental steps to implement and validate the framework. Each step consisted of first learning the practical concepts and then applying them to perform and deliver each step. The first step consisted of learning the technology that was going to be used, the second step consisted of introducing a CI/CD tool, the third step introduced the deployment in a Kubernetes cluster and the fourth step introduced a Multi-Cloud management tool. The incremental steps done during the 2nd semester are presented in Chapter 6. Figure 1.2 presents the summary of the work done during the 2nd semester.

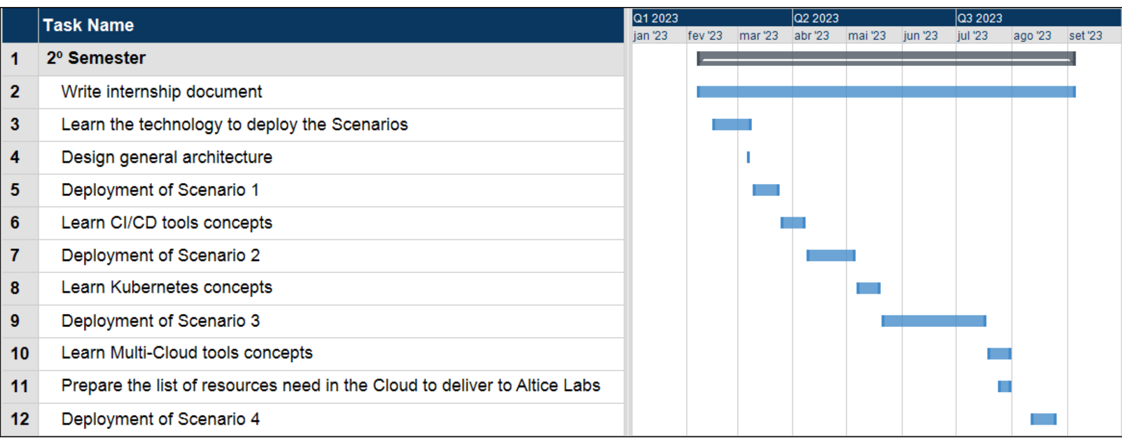


Figure 1.2: Gantt chart for 2nd semester

1.7 Document Structure

The remaining of the document is organized as follows:

- **Chapter 2 -** Provides an overview of the Cloud computing paradigm, including its characteristics, deployment models, service models and a comparative analysis between Multi-Cloud and Hybrid Cloud;
- **Chapter 3 -** Reviews and describes the analyzed Multi-Cloud and Hybrid Cloud environments management tools;
- **Chapter 4 -** Provides an overview of tools used for automatic deployment.
- **Chapter 5 -** Presents an analysis of the problem, including its description, requirements and risks;
- **Chapter 6 -** Describes the proposed solution to solve the problem at hand, including its general architecture, workflows, scenarios, implementation and validation;
- **Chapter 7 -** Depicts the main conclusions, a summary of the sections addressed in the report and the steps that can be done in the future.

Chapter 2

Cloud Computing Paradigm

This chapter provides an overview of cloud computing, as well as the related fundamental theoretical concepts to the understanding of the subject covered in this internship.

The National Institute of Standards and Technology (NIST) defines Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction" [49]. Some examples of configurable computing resources are networks, servers, storage, applications or services.

With regard to the Cloud model organization, it is composed of five Cloud computing characteristics, four service models, and five deployment models (or Cloud types). Figure 2.1 represents the Cloud model organization previously described with the concepts that will be covered throughout the present chapter.

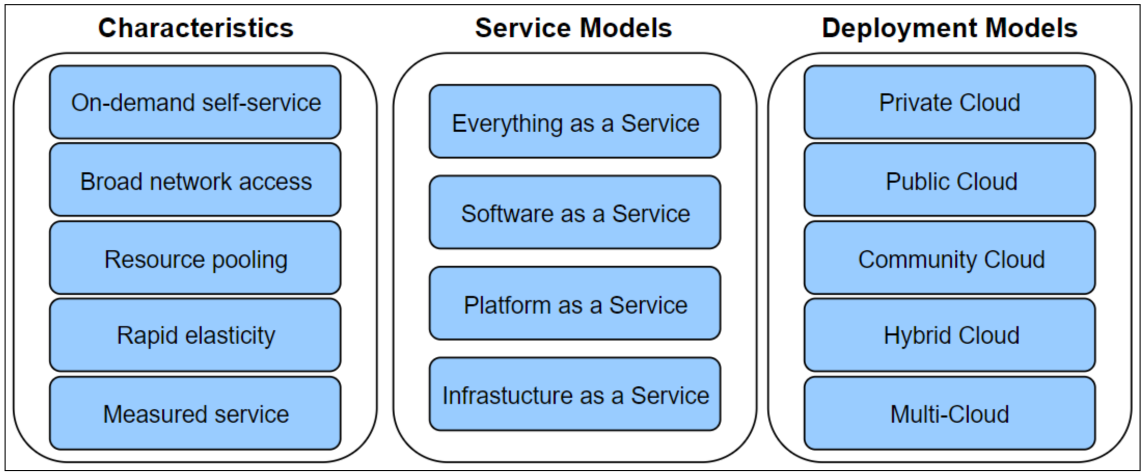


Figure 2.1: Cloud model organization

2.1 Cloud Computing Characteristics

The Cloud computing model is composed of five essential characteristics. These characteristics are: On-demand self-service, Broad network access, Resource pooling, Rapid elasticity, and Measured service.

The *On-demand self-service* characteristic affirms that a consumer can provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider [49]. Considering a start-up context, most users begin by using limited resources and may increase them over time due to their business growth, and this methodology allows them to request more resources all by themselves, without the inconvenience of personally interacting with the service provider every time they need to increase their resources [63].

Broad network access states that "capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms", such as mobile phones, tablets, laptops and workstations [49].

The next characteristic is *Resource pooling* and declares that "the provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand" [49]. Some examples of resources are storage, processing, memory and network bandwidth. The main advantages of resource pooling are the increased robustness against component failures, better ability to handle localized outbreaks in traffic, and a maximized utilization [71].

Rapid elasticity asserts that resources can be dynamically provisioned and deprovisioned, and in some cases automatically, to scale rapidly proportionally with demand [49]. From the customer's perspective, the resources available often seem to be unlimited and can be booked in any quantity at any time. Elasticity is normally associated with infrastructures, therefore with the Infrastructure as a Service (IaaS) Cloud service model. Elasticity is beneficial because of the need to keep the gap between demand and capacity as small as possible so that it is not necessary to refuse requests due to a lack of resources [48].

The last characteristic is *Measured service* and it states that Cloud systems automatically control and optimize the appropriated resource usage to the type of service, such as storage, processing, and bandwidth. Resource usage can be monitored, controlled and reported, providing transparency for both the provider and the consumer [49].

2.2 Cloud Service Models

This section introduces an analysis of each Cloud service model. There are three main types of Cloud service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). More recently, a new

type of Cloud service model has been introduced, which is the Everything as a Service (XaaS). Figure 2.2 [62] illustrates the Cloud service models pyramid, showing some examples of services offered from each Cloud service model and the main users of each service offered.

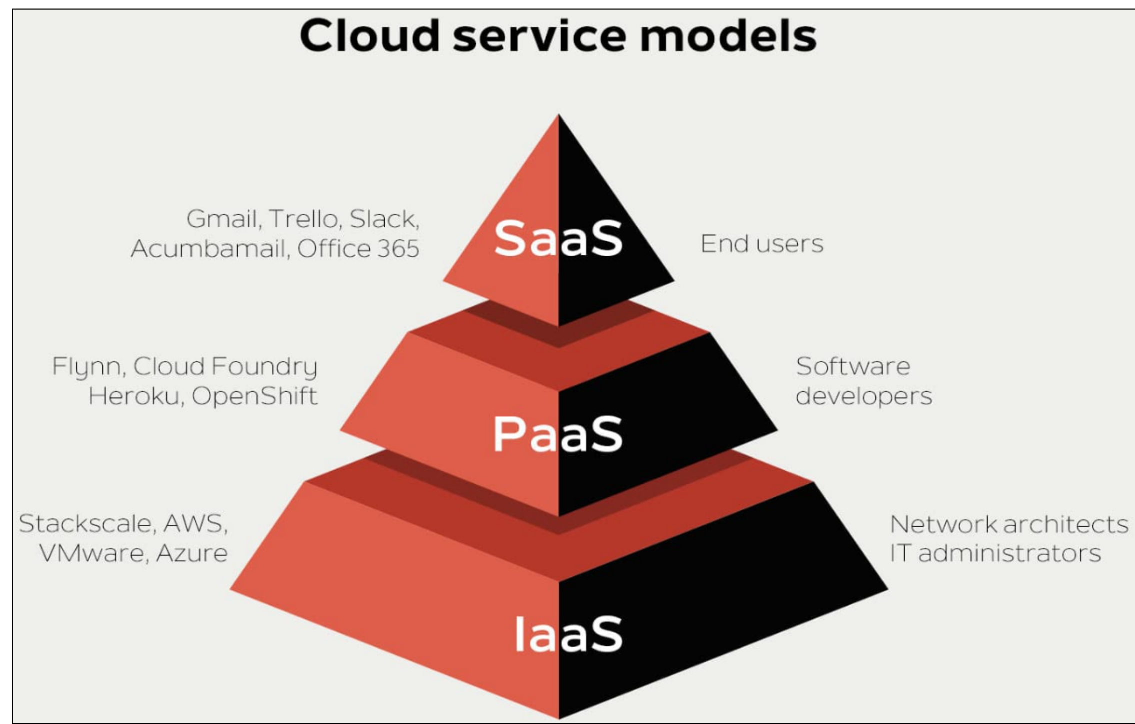


Figure 2.2: Cloud service models [62]

2.2.1 Infrastructure as a Service

The first service model is IaaS and provides the consumer with the capability to provision processing, storage, networks and other computing resources where the consumer is able to deploy and run arbitrary software, such as Operating Systems (OSs) and applications. In this service model, the consumer does not have any kind of control over the underlying Cloud infrastructure, only has control over OSs, storage and deployed applications [49].

Regarding the management of the infrastructure, the responsibility falls to the Cloud Service Provider (CSP) and, as previously noted, the consumer has no control or management permissions over the hardware. As for renting, an Application Programming Interface (API) or dashboard is made available to the client so that he will be able to rent the infrastructure. This service model is the typical deployment model of Cloud storage providers [57]. Public Cloud providers like Amazon Web Services (AWS), Microsoft Azure and Google Cloud are examples of IaaS [55].

2.2.2 Platform as a Service

The Platform as a Service (PaaS) is the model where the consumer is provided with the capability to deploy onto acquired applications created using programming languages, libraries, services and tools supported by the provider. In this service model, the consumer is not able to manage or control the underlying Cloud infrastructure including network, servers, OSs or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment [49].

In other words, it can be stated that the hardware and an application-software platform are provided and managed by a Cloud service provider, but the user has to handle the apps running on top of the platform and the data the app needs. Developers and programmers primarily use this service model [57]. Some examples of PaaS are AWS Elastic Beanstalk, Heroku and Red Hat OpenShift [55].

2.2.3 Software as a Service

In the Software as a Service (SaaS) model the consumer is provided with the capability to use the provider's applications running on a Cloud infrastructure. The applications are accessible from various clients' devices through either a web browser or a program interface. In this service model, the consumer does not manage or control the underlying Cloud infrastructure or individual application capabilities, with the exception of some user-specific application configuration settings [49].

The SaaS model incorporates a number of unique characteristics. Some examples are that allows the consumer to benefit from the vendor's latest technological features without the costs associated with software updates and upgrades; it allows each consumer to opt either to share access to the software with other consumers, thus enabling shared total costs, or decide to be a single tenant, thus providing greater control and security; and also lets users access the vendor's software through the Internet on a "pay-as-you-use" basis, rather than licensing, installing and maintaining software on client's computers or servers [32].

2.2.4 Everything as a Service

The Everything as a Service (XaaS) is a more recent type of service model that has been gaining more relevance over time due to the fact that numerous service models have been proposed in the form of "as a Service" or "aaS".

Despite the options of "aaS" being infinite, some papers regarding the XaaS concept organize the "aaS" options into various types. Some types of "aaS" are the following: *Traditional services*, which are provided either by individual people directly with concrete actions, or nominally by institutions/society but still implemented by real people who interact with end users, as "Music as a Service"; *Network services*, which are applications running at the network application layer

and above which are based on application layer network protocols for provision of capabilities, as "Internet Protocol (IP) Networks as a Service"; *Services in programming/modelling*, which are used in the context of programming and OSs in a computer system by a process to response to users' requests, as "Data Access as a Service"; *General Service-Oriented Architecture (SOA) services*, which follows the design pattern of SOA in which distinct pieces of software provide application functionality as services to other applications via a protocol, as "Models as a Service"; *Web services*, that refers to software functions provided over the Web, as "Training as a Service"; and finally *The Cloud era*, that aims to leverage utility and consumption of computing resources related to Public, Private or Hybrid Cloud infrastructures such as the previously mentioned IaaS, PaaS and SaaS [72] [73].

To reinforce the idea of XaaS, more "aaS" examples can be Desktop as a Service (DaaS), Monitoring as a Service (MaaS), DNS as a Service (DNSaaS), Database as a Service (DBaaS), Load Balancing as a Service (LBaaS), and Content Distribution Network as a Service (CDNaaS) [61].

2.3 Deployment Models

This section presents a description of each available deployment model. The five main models of Cloud computing are Private Clouds, Public Clouds, Community Clouds, Hybrid Clouds, and Multi-Clouds.

2.3.1 Private Clouds

The Private Cloud is defined as "computing services offered either over the Internet or a private internal network and only to select users instead of the general public" [50].

This Cloud infrastructure is provisioned for exclusive use by a single organization with multiple consumers, like business units [57]. In terms of ownership and maintenance, Private Clouds may be owned, managed, and operated by the organization, a third party, or some combination of them [49]. Private Clouds are also known as internal or corporate Clouds, and some benefits they give to business includes self-service, scalability, elasticity [50], and normally a higher degree in terms of control, privacy, and security through the companies' firewall.

A situation where security needs to be extremely well managed is when companies have sensitive data stored in their Cloud and this one is managed by a third party. In this case, the companies need to give the third party access to their Cloud for them to manage and maintain it, but also need to restrict that access to a certain level.

On the other hand, Private Clouds are more expensive in terms of costs, being that still require significant upfront Capital Expenditure (CAPEX) and Operational Expenditure (OPEX). Other concerns are that the elasticity feature is normally bounded by existing hardware resources and that the company's Informa-

tion Technology (IT) department is held responsible for the cost, accountability and management of the Cloud, thus Private Clouds require the same of staffing, management, and maintenance expenses as a traditional datacenter ownership [50].

Private Clouds are capable of delivering two types of Cloud service models. The first one is IaaS, which allows the use of the infrastructure resources such as compute, network and storage. The other is PaaS, which allows companies to deliver everything from simple Cloud-based applications to sophisticated-enabled enterprise applications [50].

2.3.2 Public Clouds

The Public Cloud environment is normally created from an IT infrastructure not owned by the end user but by a CSP, and it is provisioned for open use by the general public, being shared with multiple organizations using the public Internet [57] [49] [70]. Regarding ownership and maintenance, Public Clouds may be owned, managed, and operated by a business, academic, or government organization, or some combination of them.

Usually, Public Clouds have less privacy and security when compared to Private Clouds, but there are cases where Public Clouds may be safer than Private Clouds. An example is when the IT department responsible for the Private Cloud management is not as reliable, capable and qualified as the IT department in charge of the Public Cloud management.

In order to provide a Public Cloud infrastructure, the CSP uses groups of datacenters that are partitioned into Virtual Machines (VMs) and shared by the end users [70]. This deployment model uses a "pay per use" basis, meaning that end users only pay for the resources they consume. A real-life situation where is proven that this characteristic is actually convenient is in a start-up company context. Normally, a start-up company does not have enough budget to purchase, manage, and maintain its own Private Cloud at the beginning of its business, therefore they choose to use a Public Cloud environment in an initial phase where the CSP is held responsible for all management and maintenance of the system, and then, on a latter phase invest and migrate to a Private Cloud environment [51].

Another point in favour of Public Clouds when compared to Private Clouds is that elasticity is typically higher, as Public Cloud providers own a bigger amount of resources to provide. Some examples of Public Cloud providers are Alibaba Cloud, AWS, Google Cloud, IBM Cloud, and Microsoft Azure [57].

2.3.3 Community Clouds

The Community Cloud infrastructure is "provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns" [49], like universities or libraries. With regard to ownership and maintenance,

Community Clouds may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them [49].

About its characteristics, it shares some of the advantages of Private Clouds, like control, flexibility and customization, with some advantages of Public Clouds, such as more elasticity and fewer costs. The costs are shared among the community and regarding security is potentially less secure than a Private Cloud, but still more secure than a Public Cloud, even though it depends more on the quality of the Cloud management services than on the deployment model [61].

In respect of Community Cloud's advantages, the first is that the costs of setting up a communal Cloud versus an individual Private Cloud can be cheaper due to the division of costs among all participants. The second is that the management of the Community Cloud can be outsourced to a Cloud provider. The advantage here is that the provider would be an impartial third party that is bound by contract and that has no preference for any of the clients involved other than what is contractually mandated. The third is that the tools residing in the Community Cloud can be used to leverage the information stored to serve consumers and the supply chain [32].

Regarding the drawbacks, some examples are that it costs more than a Public Cloud and that a fixed amount of bandwidth and data storage is shared among all community members [32].

2.3.4 Hybrid Clouds

A Hybrid Cloud is a more complex type of Cloud, due to its composition consisting of two or more distinct Clouds, such as Private, Public, or Community. Also, for an environment to be considered a Hybrid Cloud, it is mandatory that it has a Private Cloud. The most common Hybrid Cloud example is the combination of a Private and Public Cloud environment, like an on-premises datacenter, and a Public Cloud computing environment, as AWS, Google Cloud, and Microsoft Azure [31]. Each Cloud remains a unique entity but is bound together with other Cloud by standardized or proprietary technology that enables data and application portability among them [49] [32]. The connection between the Clouds can be established through Local Area Networks (LANs), Wide Area Networks (WANs), Virtual Private Networks (VPNs), and/or APIs [57].

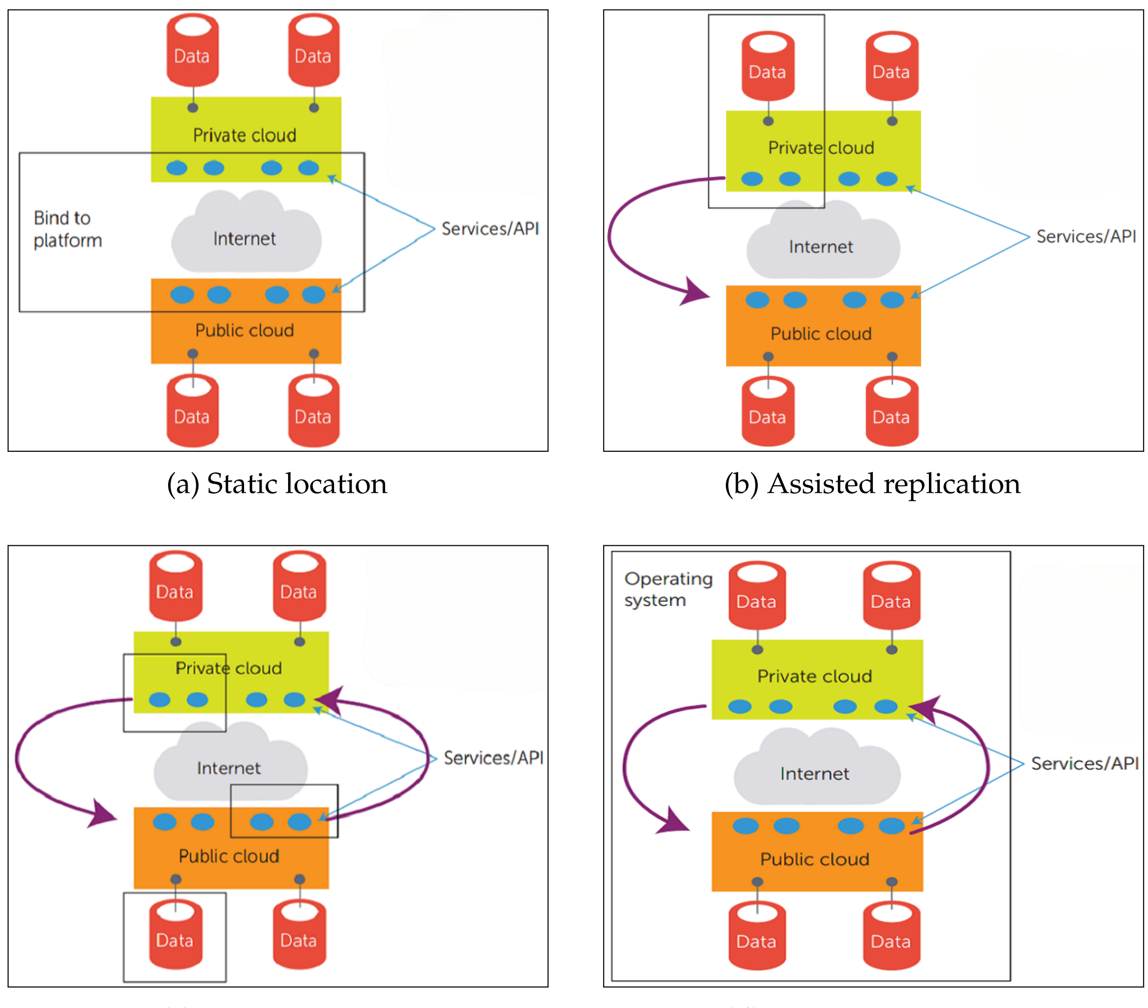
A very useful feature supported by Hybrid Clouds is Cloud bursting, which is the ability to auto-expand the use of Public Cloud resources [47]. This helps enterprises achieve cost optimizations by maintaining a bare minimum IT resources in-house, as they know there is a backup in place, in case there is a sudden spike in user demand [20].

Nowadays, from a business point of view, companies tend to migrate to Hybrid Clouds due to several factors. First, simple Clouds might not provide the resources and services needed to cover the requirements that big companies de-

mand. Second, the rise of new management technologies allows them to manage and control more complex environments in a simpler way. An example is having just an interface that covers various functions. Third, the need to overcome security and governance challenges is essential to place some workloads on-premises and some off-premises. Fourth, using a computing strategy based on Hybrid Clouds is cheaper and also adds some benefits such as agility. Finally, certain types of sensitive data, such as financial and health-related data, have limitations as to where they must be stored, being more secure to store these types of data on-premises [47].

Through the last few years, many Hybrid Cloud innovative approaches have been developed. According to David S. Linthicum [47] the emerging hybrid architecture patterns can be organized into four categories. The first one is *static location*, which refers to architectures where the workload's location is tightly bound to Private or Public Clouds, as it is visible in Figure 2.3a with each service/API affixed to its Cloud. This means it is difficult to migrate workloads between Private and Public Clouds. Currently is the most typical architecture. The following category is *assisted replication* and refers to architectures where some workloads can be replicated from Private to Public Clouds, or the other way around. In Figure 2.3b is illustrated an entire workload located in a Private Cloud that is to be replicated in a Public Cloud. Due to this architecture's limited use of abstraction, the workloads replication needs to be performed with some technological assistance, such as code and/or interfaces. The third architecture is *automigration* and refers to the code or entire workloads moving between Private and Public Clouds, through human intervention or through automated processes, by using some abstraction. In Figure 2.3c is presented the automatic movement of the data and/or workloads between a Private and Public Cloud. The final architecture is *dynamic migration* and refers to moving workload instances between Private and Public Clouds as if both existed in the same virtual OS. In Figure 2.3d is shown that the workflows can move from Cloud to Cloud as if they were on the same server. This last category is the functional objective of Hybrid Cloud computing [47].

To assemble a Hybrid Cloud and move to a hybrid architecture, there are some steps that need to be followed to implement and operate such a complex and distributed technology. Firstly, is necessary to ensure the appropriate security mechanisms that cover both Private and Public Clouds. Also, logging and auditory systems need to be functional as well. A good security approach that fits in here is an Identity and Access Management (IAM) approach. Furthermore, the requirements for the Hybrid Cloud that will be developed need to be analyzed, estimated and understood. Moreover, Private and Public Clouds candidates need to be selected taking into consideration the known compatibility of each candidate. After that, follows the testing phase in order to assure that the Hybrid Cloud assembled meets the requirements. Still related to testing, it is a good practice to implement a noncritical application on the Hybrid Cloud, to try to find any issues and fix them quickly. In addition, a Cloud Management Platform (CMP) or a CSP are needed to manage the Hybrid Cloud resources. At last, a usage-based accounting system is needed for show-backs and chargebacks, as well as to limit the use of resources [47].



(a) Static location

(b) Assisted replication

(c) Automigration

(d) Dynamic migration

Figure 2.3: Hybrid architecture patterns [47]

2.3.5 Multi-Clouds

A Multi-Cloud is defined as Cloud systems in which applications are hosted as chunks among a heterogeneous network of different Clouds. In other words, it can be said that is a Cloud system where different Cloud networks are merged for different roles. According to [36], Multi-Cloud's components are all unique Cloud systems and not deployment models, as happens with Hybrid Clouds. A Multi-Cloud consists of the combination of two or more Clouds, but for an environment to be considered a Multi-Cloud it is mandatory that it has at least two Public Clouds from different CSPs.

When choosing what type of Cloud to use, there are several motivations that encourage companies to opt for a Multi-Cloud approach. One motivation is due to legal constraints. This happens when a company needs to operate a part of its business in one Cloud situated in a specific location, but due to possible law restrictions in that location, the company needs to perform the storage of the needed data in a different region or Cloud [36]. Other motivation that leads companies to opt for a Multi-Cloud is when they have planned to expand their infrastructure in the future to either enhance productivity, efficiency and security, meet 24×7 requirements for high data availability, or when the companies have a global customer base and want to guarantee that their customers do not have latency issues while accessing their services [39]. At last, one of the most important motivations is the convenience of combining the resources from different Clouds in order to use the best resources and services of each CSP in only one environment.

According to Petcu [53], there are various reasons why the services and resources from multiple Clouds are needed simultaneously. The first reason is to deal with the peaks in service and resource requests by using external ones. The second reason is to optimize costs or improve the quality of services. Another reason is enabling the ability to react to new CSPs' offers. Other important reason is to avoid being dependent on only one provider, thus avoiding a vendor lock-in. Moreover, there is also the possibility to ensure backups in different CSPs in order to deal with disasters or scheduled inactivity [53] [36].

Petcu [53] proposes a list of technical requirements for a Multi-Cloud. These technical requirements are organized into three groups. The first set of requirements belongs to the *development group* and some of them are the following: offer a resource and service management software, offer services that are Cloud vendor agnostic, offer an interface for describing functional and non-functional requirements of the clients, and support the application portability between the connected Clouds. The second set of requirements belongs to the *deployment group* and some requirements listed are: maintaining the particularities of various Clouds, not imposing any constraints on the connected Clouds, supporting the connection with the top Cloud providers, supporting the application relocation between Clouds, being able to deploy on Private Clouds to enable testing, debugging or privacy, and the ability to use automated procedures for deployments. The last group is the *execution group*, where the requirements are: offering a monitoring service for the deployed applications, allowing the management of

the full life-cycle of the deployed applications, and allowing a dynamic allocation of resources or mechanisms for self-adaptation [53].

2.4 Multi-Cloud vs. Hybrid Cloud

As previously mentioned, Multi-Clouds and Hybrid Clouds are Clouds that are composed of the combination of more than one Cloud. A typical Multi-Cloud integrates at least two different Clouds of the same type, while a Hybrid Cloud blends two or more Clouds of different types having at least one Private Cloud. In addition, in a Multi-Cloud environment, an organization utilizes multiple Public Cloud services, while a Hybrid Cloud environment is usually used to orchestrate a single IT solution [68].

Regarding vendor disparities between Multi-Clouds and Hybrid Clouds, while Multi-Clouds include multiple Clouds from different vendors that provide multiple Cloud services, Hybrid Clouds only have one vendor associated. In an industry context, a Multi-Cloud can bring benefits such as enabling a company to use best-in-class services for each app or task, reducing the risk of vendor lock-in, and allowing better business planning by opting for the most affordable services [39].

Considering combination differences, while Multi-Clouds are composed at least of a combination of Public + Public Clouds, Hybrid Clouds are composed of two or more Clouds of different types, having at least one Private Cloud in the combination. A Cloud can be named either a Multi-Cloud or Hybrid Cloud when the Clouds' combination consists of, at least, a Public + Public + Private Clouds mixture. Using fruits as a context for a simpler comparison is like a Multi-Clouds was at least composed of a combination of two types of apples (Public + Public), and Hybrid Clouds were at least composed of a combination between apples and grapes (Public + Private) [39] [68].

Other topics that are important to compare are the costs, availability and inter-Cloud workloads. As for the cost contrasts, in Multi-Clouds a company may not be responsible to pay for datacenters or in-house systems, while in Hybrid Clouds a company normally faces significant expenditure due to the costs associated with its Private Cloud. With regard to availability, while in Multi-Clouds high availability is one of the driving factors due to the CSPs' availability, in Hybrid Clouds high availability is dependent on the in-house teams. At last, about the inter-Cloud workloads, while in a Multi-Cloud setup, different Clouds can manage different tasks, in Hybrid Clouds different components work simultaneously together to run a single IT solution [39].

Regarding the similarities, both Multi-Clouds and Hybrid Clouds are advisable for the storage of sensitive data, therefore in both cases, sensitive data storage is subject to infrastructural design and business requirements. The security is also reliant on the system's underlying architecture, moreover, in both cases, the Cloud providers can take the responsibility to safeguard the infrastructure from external threats and attacks. Another similarity is the emphasis on regulatory

compliance, and in this case, even though Hybrid Clouds setups collaborate with Cloud providers that adhere to laws, regulations, guidelines and specifications, Multi-Clouds are slightly superior in this aspect due to the multiple vendors characteristic, which can be beneficial because some laws from a specific vendor may be more valuable for a certain type of business. The final similarity is that is very complex to migrate data between each other, being that this task can be time-consuming and challenging. In a Multi-Cloud setup, data has to be migrated to multiple different types of Clouds, while in a Hybrid Cloud setup, the migration is to Public Clouds of different vendors, requiring even more time, resources and skills to perform it [39].

The following Table 2.1 resumes the comparison between Multi-Clouds and Hybrid Clouds. The (✓) mark indicates that the Cloud has that characteristic and the (✗) mark states that the Cloud does not have that characteristic.

Characteristics	**Multi-Cloud**	**Hybrid Cloud**
Clouds from different vendors	✓	✗
Multiple Public Cloud services	✓	✓
Always includes a Private Cloud	✗	✓
Can include a Private Cloud	✓	✓
Private + Public combination	✗	✓
Public + Public combination	✓	✗
Public + Public + Private combination	✓	✓
High costs associated	✗	✓
High availability	✓	✓
Store sensitive data	✓	✓
Infrastructural security	✓	✓
Emphasis on regulatory compliance	✓	✓

Complex Cloud migration	✓	✓

Table 2.1: Multi-Cloud vs. Hybrid Cloud

2.5 Summary

In this chapter, a global overview of the Cloud model organization was presented. At first, it was presented its five computing characteristics, namely the on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. Secondly, its four service models were depicted, as IaaS, PaaS, SaaS and XaaS. Finally, it was provided its five deployment models, which are Private Clouds, Public Clouds, Community Clouds, Hybrid Clouds, and Multi-Clouds.

The final section of this chapter performed a comparison between Multi-Clouds and Hybrid Clouds. The main difference is that Multi-Clouds are composed of at least two Public Clouds from different vendors, while Hybrid Clouds need to have at least a Private Cloud integrated.

Chapter 3

Management Tools for Multi and Hybrid Clouds

This chapter is composed of the presentation of some Multi-Cloud and Hybrid Cloud management tools. The sections that compose this chapter are the following: Management Tools Analysis and Tool Comparison. The first section presents the investigation performed for each management tool addressed in this report. The tool analyzed are Terraform [64], Google Anthos [27], Amazon Elastic Kubernetes Service (EKS) [12] and Ansible [6]. The other section of this chapter provides a comparison summary between the management tools addressed in the previous section. To perform the comparison, a set of features is used as comparison criteria.

3.1 Management Tools Analysis

This section presents an analysis of some Multi-Cloud and Hybrid Cloud environment management tools. The management tools addressed are Terraform [64], Anthos [27], EKS [12] and Ansible [6].

Beyond the tools that are going to be addressed, there are also other tools available to perform centralized management of Cloud environments. Some examples are Cloudify [19], Puppet [54] and Chef [18]. In this case, the comparison will only be performed between Terraform, Anthos, EKS and Ansible [6]. The reason for this is because Altice Labs (ALB) had an interest in having an analysis of Anthos, EKS and Ansible, which are recent tools that they do not use and have little knowledge, and compare them with a tool they are already using, which is Terraform.

A standard structure was defined in order to perform a similar and balanced analysis of the management tools in hand. To begin with, the definition of the tool is provided. Then, it is described how the tool works and how it is composed. Finally, it is presented the characteristics of the tool.

3.1.1 Terraform

Terraform is an Infrastructure as Code (IAC) that allows the build, change, and version of both Clouds and on-premises resources safely and efficiently. The resources are defined in human-readable configuration files that can be versioned, reused, and shared. Terraform can manage low-level components like compute, storage and networking resources, as well as high-level components like Domain Name System (DNS) entries and Software as a Service (SaaS) features [35]. Terraform provides the foundation for Cloud and on-premises infrastructure automation using IAC approach to provisioning Cloud infrastructure and services [34]. It is classified as an orchestration tool because it is designed to provision the servers themselves, leaving the job of configuring those servers to other tools [15].

It uses its own high-level configuration language known as Hashicorp Configuration Language (HCL), or optionally JavaScript Object Notation (JSON), in order to detail the infrastructure setup. Furthermore, it encourages a more declarative style where the user writes code that specifies his desired end state and the IAC tool itself is responsible for figuring out how to achieve that state [15]. Also, it is able to manage multiple Cloud providers and even cross-Cloud dependencies by means of special plugins called providers. Moreover, it is backed by large communities of contributors and is well documented, both in terms of official documentation and community resources such as blog posts and StackOverflow questions [15]. Terraform supports continuous delivery by applying configuration updates, which allows adding, removing or modifying resources, and when resource arguments cannot be updated, the existing resource will be replaced by a new one instead. However, Terraform does not support recovery actions, since any errors need to be addressed manually. Another limitation of Terraform is its lack of compatibility with other tools, such as VMware, because it is dependent on Application Programming Interfaces (APIs) to use other tools [65].

Regarding how Terraform works, it creates and manages resources on Cloud platforms and other services through their APIs. Terraform uses providers to be enabled to work with virtually any platform or service with an accessible API. Thousands of providers have already been written by HashiCorp and the Terraform community in order to manage many different types of resources and services [35]. In terms of the core Terraform's workflow, it consists of four stages [35]:

- **Write:** The resources are defined by ourselves, which may be across multiple Cloud providers and services. An example is to create a configuration to deploy an application on Virtual Machines (VMs) in a Virtual Private Cloud (VPC) network with security groups and a load balancer;
- **Plan:** Terraform creates an execution plan that describes the infrastructure that will create, update or destroy based on the existing infrastructure and the written configuration;
- **Apply:** On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. An example is when

the user updates the properties of a VPC and changes the number of VMs in that VPC, Terraform will recreate the VPC before scaling the VMs;

- **Destroy:** Terraform decommissions the services and infrastructure configured, so it unwinds everything that has been done. In some sense, is a specialized version of **Apply**. An example is when the user creates a staging or test environment and then needs to destroy it.

Figure 3.1 illustrates Terraform's workflow stages previously described. The writing stage is represented by the file where the configurations are written, the planning stage is represented by Terraform because it is responsible for the planning, and the apply stage is represented by the Cloud and servers thus is there where the configurations are going to be applied.

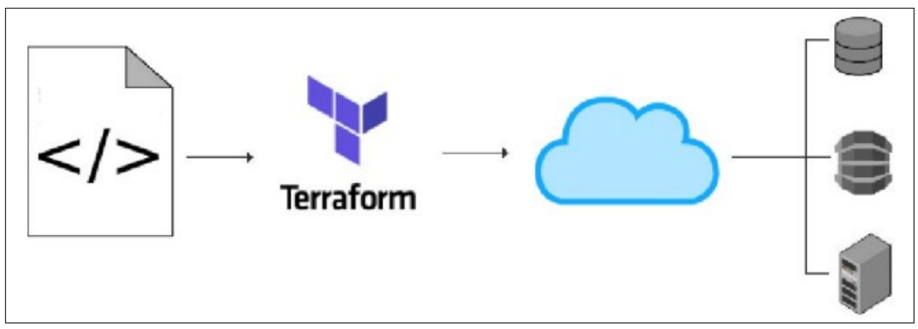


Figure 3.1: Terraform workflow [17]

Terraform can be used by a singular individual practitioner who uses it locally, and by multiple team members that collaborate to manage the same infrastructure. In the single-user case, the user uses Terraform independently without the multiple users overwrite issues. In the multiple team members case, there is the need to make sure the team has a constant view and the need to avoid multiple changes in parallel. As an individual, the user is still writing his Terraform configurations and planning locally, but now he pushes what he made locally into a Version Control System (VCS), as GitHub or Bitbucket, to manage and coordinate as it is done for source control. Then the Terraform Enterprise is used to solve state management issues, to make sure there is only one run at a time, and to keep variables centrally and encrypted.

Terraform has some characteristics that allow it to solve infrastructure challenges. The characteristics are the following [35]:

- Terraform allows to **manage any infrastructure** by using already written providers for the platforms and services we use, or by using a provider written by ourselves. Also, Terraform takes an immutable approach to infrastructure, reducing the complexity of upgrading or modifying services and infrastructure, and reducing the likelihood of configuration drift bugs [15];

- The user can **track his infrastructure** through a plan generated by Terraform itself, and before modifying his infrastructure Terraform prompts him for his approval. A state file is used to keep track of the user's real infrastructure, which acts as a source of truth for his environment. Terraform uses the state file to determine the changes to make to the user's infrastructure so that it will match his configuration;

- The user is able to **automate changes**. Terraform configuration files are declarative, meaning that they describe the end state of the user's infrastructure. There is no need to write step-by-step instructions to create resources because Terraform handles the underlying logic. Also, Terraform builds a resource graph to determine resource dependencies and creates or modifies non-dependent resources in parallel, allowing Terraform to provision resources efficiently;

- The capability to **standardize configurations**. Terraform supports reusable configuration components named modules that define configurable collections of infrastructure, saving time and encouraging best practices. The user has the possibility to use publicly available modules from the Terraform Registry, or write his own;

- The ability to use Terraform in a **collaborative** way. As the user's configuration is written in a file, he can commit it to a VCS and use Terraform Cloud to efficiently manage Terraform workflows across teams.

3.1.2 Anthos

Anthos is Google's managed applications platform, which allows users to run Kubernetes and other workloads consistently across on-premises datacenters and multiple Public Clouds. It may be one of the first Multi-Cloud platforms backed by a major Cloud provider, with native support for on-premises deployments and Google Cloud, Amazon and Azure Cloud environments. In order to use this tool tho run their workflows the users need to pay for a subscription [52] [30].

Anthos focuses on three key capabilities. The first one is **Multi-Cloud container orchestration** and states that Anthos runs on both existing virtualized infrastructures and bare-metal servers, enabling the administration of Kubernetes clusters both on-premises or in the Cloud. The second capability is **Automating policies**, which affirms that Anthos' configuration manager enforces enterprise-level policies across Multi-Cloud deployments, ensuring constant observation and security enforcement. The third one is **Modernizing security** and declares that Anthos enables integrating security throughout an application's develop-build-run cycle and also creates a defence-in-depth security model that employs a broad selection of security controls consistently across all environments [52].

Regarding how Anthos is built, at its heart is **Google Kubernetes Engine (GKE)**, which performs some activities like the management of Kubernetes clusters and dependent applications, the monitoring of applications and switching

loads between on-premises and Cloud. With GKE users can reserve Internet Protocol (IP) addresses via Google Cloud Virtual Private Network (VPN), allocate compute resources to a cluster and scale up or scale down the deployment in accordance with the demands. Moreover, GKE gives the users the ability to manage all resources using built-in dashboards and gain insights into the functionality of applications using Google Cloud (Stackdriver) Monitoring and Logging services [52].

The next Anthos building block is **GKE On-Prem**, also known as Anthos clusters on VMware. It is a software that brings GKE to on-premises datacenters, allowing to create, manage and upgrade Kubernetes clusters in an on-premises environment. It runs on top of VMware vSphere 6.5, but Google is working on supporting additional hypervisors including Hyper-V and Kernel-based Virtual Machine (KVM) [52] [66].

The following building block is **Anthos Config Management**, which is responsible to help to deploy Kubernetes across a range of environments. It lets the user simultaneously configure and maintain multiple clusters, and rapidly develop applications across hybrid container environments. Also, it supports Kubernetes-native configuration formats, such as Yet Another Markup Language (YAML) and JSON, to manage a large number of clusters simultaneously [52].

Another building block is the networking component of Anthos, entitled **GKE Hub**. Its goal is to connect Google Cloud Services Platform, other Cloud providers and on-premises clusters' data. GKE lets the user combine and access all data from across a Multi-Cloud deployment, and view and manage all Kubernetes clusters on a single panel.

The final building block is **Istio**. It allows the user to connect Google Cloud Platform (GCP), third-party Clouds, databases and other components into a single service mesh, supporting load balancing, monitoring of large numbers of clusters and traffic management. In other words, Istio allows to manage communication among the different clusters as well as deployments and health checkers [22].

Figure 3.2 represents the diagram of Anthos' architecture. In it can be visualised the previously detailed blocks that compose Anthos, in which layer they are situated and if they run on-premises or in the Cloud. Also, it is worth mentioning that the GKE On-Prem is situated on the on-premises side, while the GKE is situated on the Public Cloud side.

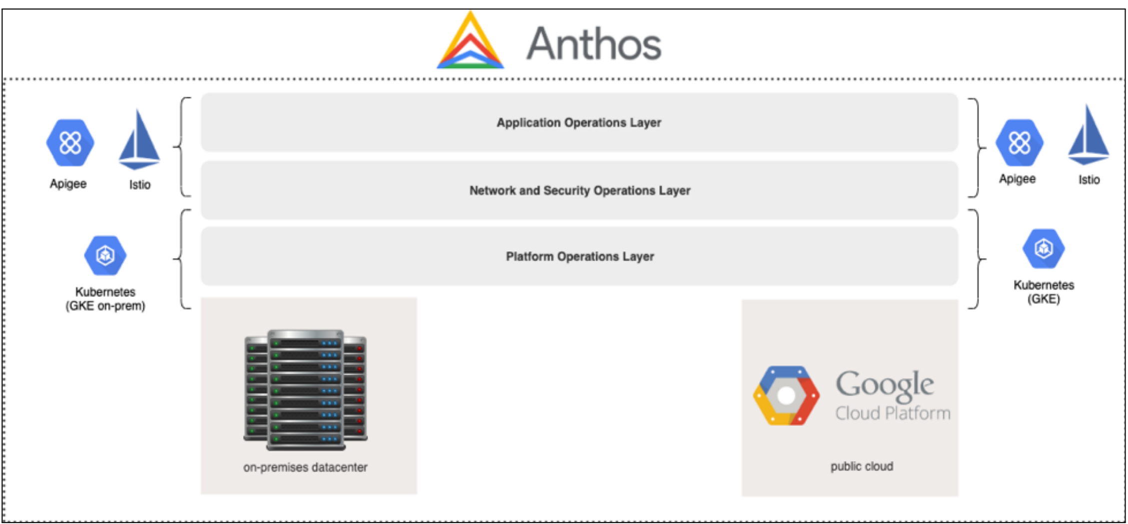


Figure 3.2: Anthos architecture diagram [52]

With respect to the features, Google Anthos has the following characteristics [52] [28]:

- Google Anthos has **Multi-Cloud support**. It runs on any Public Cloud and is simple to migrate between Clouds because Kubernetes is supported by all Cloud providers. Google Anthos also has partnerships with many hardware makers, like Cisco, Dell or Intel, so it can run with the majority of the on-premises hardware available;

- With Anthos, the user has the ability to **monitor** and improve the Cloud application's performance. It provides visibility into the performance, uptime, and overall health of Cloud-powered applications. Metrics and logs can be collected, and dashboards and views may be used to monitor both the platform and applications;

- **Anthos for VMs** supports development teams that want to standardize on Kubernetes but have existing workloads running on VMs that cannot be easily containerized. It also provides a fit assessment tool to identify the VMs that are better candidates to shift. This assessment provides compatibility recommendations, identifies the issues that must be resolved before migration, and provides an Hyper Text Markup Language (HTML) output in order to ease the view and analysis of the results [29];

- Besides Anthos, Google also developed a Multi-Cloud analytics tool named **BigQuery Omni**. This tool was created in order to allow a centralized analysis of data in an environment with multiple providers. Another reason why this tool was created is that other analytics tools are often elementary for Multi-Cloud analytics. Without BigQuery, users would typically be required to paste the data of the different Clouds they are using to their primary Cloud analytics tool [67];

- As previously mentioned, Anthos is **compatible with on-premises systems and other Cloud service providers**, like Amazon Web Services (AWS) and

Azure. Also, one major benefit due to the underlying open-source technology is that the users are **not forced to utilise a specific Cloud vendor** [37];

- Anthos can be used to transfer workloads to a GKE container, and to **migrate workloads from other providers**, like Azure, AWS or VMware VMs [37].

3.1.3 Elastic Kubernetes Service (EKS)

Amazon EKS is a managed service used to run Kubernetes on AWS without needing to install, operate and maintain a personal Kubernetes control plane or nodes. EKS runs and scales the Kubernetes control plane across multiple AWS availability zones to ensure high availability [13]. On-premises, EKS provides a consistent Kubernetes solution and totally compatible with integrated tools and simple deployment to AWS Outposts, VMs or bare metal servers [10]. AWS Outposts solutions allows the users to extend and run native AWS services on-premises [11]. Traditionally, Amazon EKS is used for deployment across hybrid environments.

The applications running on EKS are fully compatible with applications running on any standard Kubernetes environment, regardless they are running in on-premises datacenters or Public Clouds. This means that any standard Kubernetes application can easily be migrated to EKS without any code modification. EKS is integrated with many AWS services in order to provide scalability and security for the users' applications. Some examples of the capabilities provided are AWS Elastic Container Registry (ECR) for container images, and also Amazon VPC for isolation. As it runs updated versions of the open-source Kubernetes software, the users are able to use all of the existing plugins and tooling from the Kubernetes community [13].

Regarding how it works, to start with EKS the first step that needs to be done is to provision an Amazon EKS cluster in the AWS Management Console or with the AWS Command Line Interface (CLI). This EKS cluster is provisioned through AWS which creates a master node or control plane. The next step consists of deploying managed or self-managed Amazon Elastic Compute Cloud (EC2) nodes, also known as worker nodes. They will be connected to the control plane for management and scaling control. After the cluster is ready, the user can configure the Kubernetes tools that he wants to use, like *kubectl* or *eksctl*, in order to communicate and configure his cluster. After that, the user is able to deploy and manage workloads on his Amazon EKS cluster the same way he would with any other Kubernetes environment. In addition, the user can also use the AWS Management Console to view information about his workloads [13] [59].

Figure 3.3 illustrates the general overview for creating a Kubernetes cluster with Amazon EKS. Firstly, an Amazon EKS cluster is provisioned. The following step is the deployment of Amazon EC2 nodes. After that, the Kubernetes tools that will be used are configured. At last, applications can finally be deployed on the cluster configured. Also, Figure 3.4 provides a visual representation of the EKS architecture, showing two VPCs, one with the worker nodes, and the other

with the control plane. A network load balancer and an Elastic Network Interface (ENI) are used to provide communication between the VPCs.

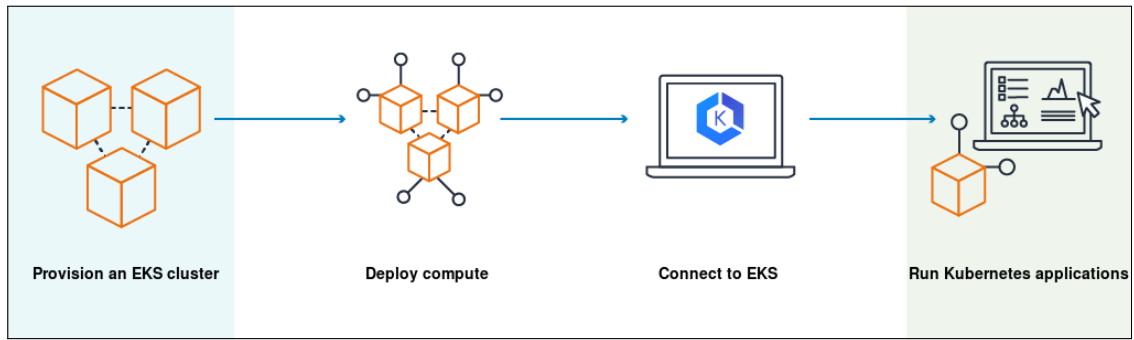


Figure 3.3: Amazon EKS setup process [13]

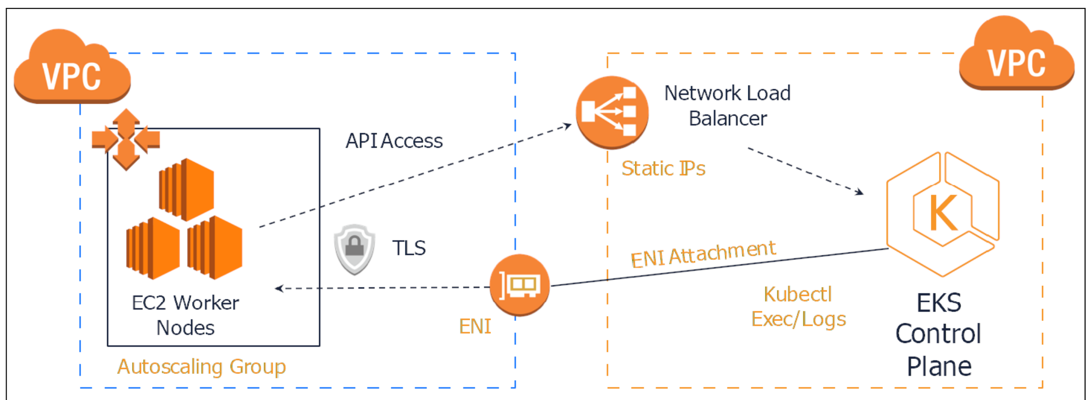


Figure 3.4: Amazon EKS architecture [59]

According to Amazon's documentation, EKS has the following characteristics [5] [9]:

- By using EKS, the user has **access to other AWS services**, like AWS Identity and Access Management (IAM) for authentication, AWS ECR for container images and AWS Elastic Load Balancer (ELB) for load distributions;
- As mentioned, EKS provides a scalable and highly-available Kubernetes **control plane** running across multiple AWS availability zones. It runs the Kubernetes control plane across the availability zones in order to secure high availability, and also automatically detects and replaces unhealthy control plane nodes;
- Typically, EKS is used for **hybrid deployments**. It can be used on AWS Outposts to run containerized applications requiring low latencies to on-premises systems. With this feature, users can manage containers on-premises with the same ease as they manage their containers in the Cloud;

- EKS supports **running worker nodes from different Operating Systems (OSs)**, like Windows worker nodes alongside Linux worker nodes. This allows the users to use the same cluster for managing applications on either OS.

3.1.4 Ansible

Ansible is an open-source Information Technology (IT) automation tool that automates provisioning, configuration management, application deployment, workflow orchestration and many other IT processes. Ansible automates the management of remote systems and controls their desired state. Ansible has the ability to manage either Linux or Windows systems but the management of Windows systems may have some limitations because it is a recent feature [56] [7].

Ansible's architecture consists of three main components. The first one is the **control node**, which is the system where Ansible is installed and where the user runs Ansible commands. The requirements for this node are that needs to have any UNIX-like machine with Python installed. This includes Debian, Ubuntu, macOS and Windows under a Windows Subsystem for Linux (WSL) distribution since Windows without WSL is not natively supported as a control node. The second one is the **managed nodes**, which are the remote systems that Ansible controls and is where the configurations are going to be performed. These nodes' requirements are to have Python installed to run Ansible library code and to have the Secure Shell (SSH) service running. The third component is the **inventory**, which is a list created by the user in the control node that contains the managed nodes IPs or hostnames. It is possible to group multiple IPs or hostnames, thus grouping nodes with different ends, like web servers or databases [7].

Figure 3.5 illustrates Ansible's architecture with its components. In it, is presented the control node with both Ansible installed and the inventory, and also are shown the connections from the control node to the managed nodes. These connections are performed via SSH, so there is no need to have any agents installed in the managed nodes [56] [7].

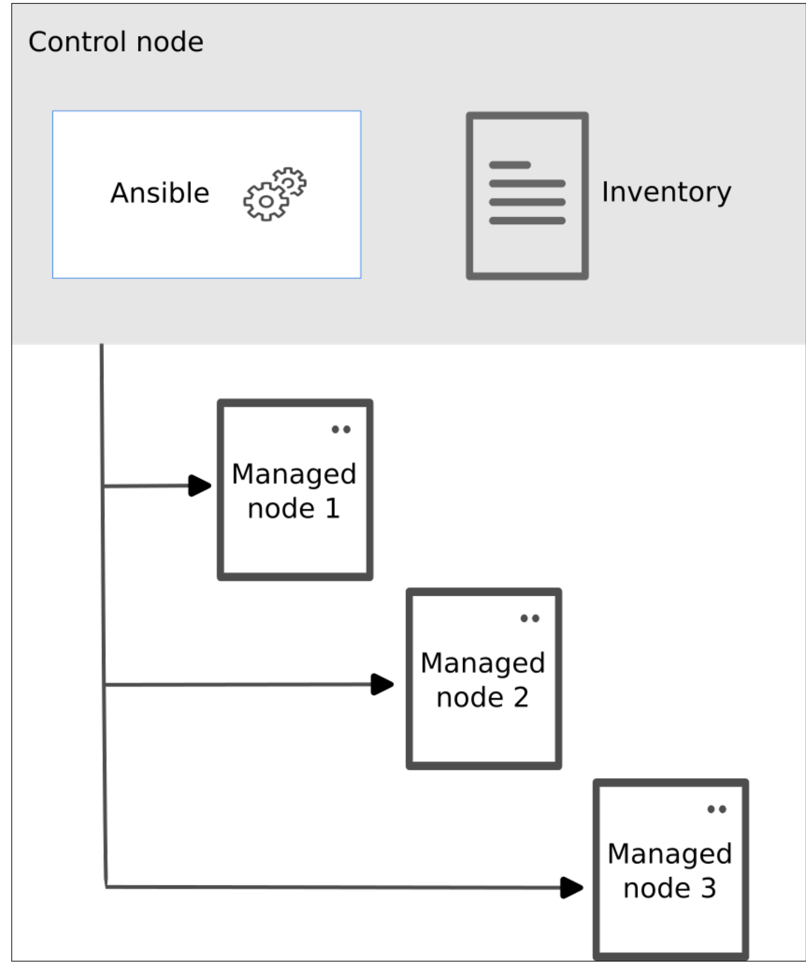


Figure 3.5: Ansible's architecture [7]

Ansible works with instructions written in YAML files named **Ansible Playbooks**. These playbooks are composed of one or more sets of instructions, each one called an **Ansible Play**. Each Play is at least composed of two components: the **hosts**, which specifies the nodes that will be configured, and the **tasks**, which represents the actions to be performed. A task contains its own description and a module, making sure that each module is executed with certain arguments. A **module** is a small program that does one small specific task, like running a command, creating or copying a file, installing or starting an Nginx server, or starting a Docker container. Ansible Playbooks can contain in a single file all the configurations to perform in the managed nodes or can import configurations and tasks written in files that are organized in a directory structure. In this case, the files are called **Ansible Roles**.

To begin using Ansible, the user has to make sure the control node and the managed nodes meet their requirements. Next, the user has to install Ansible in his control node. Afterwards, the user has to generate a key pair in his control node and then copy the public key to the nodes he wants to manage. After the connection part of the setup is done, the user needs to create a working directory in his control node. In this directory, the user must create the inventory file and also a file named "ansible.cfg". This "ansible.cfg" is the file where the user specifies some Ansible default configurations, such as specifying which inventory file

and which key Ansible has to use when running Ansible commands. After this procedure is done, the user can create and run his Ansible Playbooks from his working directory.

Ansible's features are what turn Ansible into a very powerful and accessible tool. The features linked with Ansible are the following [60] [8]:

- Ansible's automation has a significant **team impact** by allowing teams to save time and be more productive, by eliminating repetitive tasks and also by reducing the number of mistakes and errors that could happen by doing tasks manually;
- Ansible is an **open-source** and **free** tool and is also very **simple to set up and use**, not requiring the user to have any special coding skill or learn any coding language in specific;
- Ansible is a Cloud **agnostic** tool, thus it can be used to automate any machine no matter its location. The machines can be located either on-premises or in any type of Cloud or Cloud Service Provider (CSP);
- Since the connections between the control node and the managed nodes are done via SSH, Ansible is considered to be an **agentless** tool, because there is no need to install or update an agent in the managed nodes.

3.2 Tools Comparison

In this section is performed a comparison between each management tool previously described. Most of the features used were proposed by [65]. Is important to note that despite being similar, the first three features are different and complement each other. The features used for the comparison are the following:

- **Multi-Cloud support:** Supporting multiple Cloud providers, by offering a Cloud abstraction layer which hides differences and avoids the need for provider-specific customisation causing the vendor lock-in issue.
- **Cross-Cloud support:** Enhances the Multi-Cloud feature by allowing to distribute component instances of a single application over multiple Cloud providers. The advantages of this feature are allowing the selection of the best-fitting Cloud providers on a per component instance basis, optimising costs, improving the quality of services and leveraging the application's availability as it introduces resilience against the failure of individual Cloud providers.
- **Interoperability approach:** The ability to develop applications that combine resources that can interoperate, or work together from multiple Cloud providers, hence taking advantage of specific features provided by each provider.

- **Integration:** Being able to enable the use of servers not managed by a Cloud platform or VMs on unsupported Cloud providers, in order to support more advanced Infrastructure as a Service (IaaS)/Platform as a Service (PaaS) services.

- **Access:** This feature captures what interfaces a Cloud resource orchestration framework uses to interact with Cloud resources. There are three types of interfaces supported: CLI, Web-Based (WB) dashboard, and WB API.

- **Free use:** This feature specifies if the tool is free to use, thus not requiring the user to pay any kind of subscription or plan.

After the management tools analysis and considering the features just described, Table 3.1 was created with the goal of presenting the comparison between the tools mentioned before. In order to gather the information required to understand if a feature is supported by a tool, the article [65] and official documentation of each tool were used.

The (✓) mark indicates that the tool supports that feature and the (✗) mark states that the tool does not support that feature. In the Multi-Cloud support and Access features, are provided examples of what each tool supports regarding the feature at hand. The Cross-Cloud support, Interoperability approach and Free use features are simple "support" or "does not support" questions, therefore no examples are listed. Regarding the Integration feature, Terraform supports external IaaS/PaaS but does not allow the user to bring his own node, thus the (✗) on the Bring Your Own Node (BYON). On the other hand, Anthos, EKS and Ansible support both options.

Cloud features	Terraform	Anthos	EKS	Ansible
Multi-Cloud support	✓(AWS, GCP, Azure, OpenStack, Oracle, vSpheres and more than 30)	✓(Google Cloud, Amazon, Azure)	✗(AWS)	✓(Any type of Cloud and CSP)
Cross-Cloud support	✓	✓	✗	✓
Interoperability approach	✗	✓	✗	✓
Integration	✓External IaaS/PaaS services ✗BYON	✓	✓	✓
Access	✓(CLI, WB Dashboard, WB API)	✓(CLI, WB Dashboard, Kubernetes API, Anthos Multi-Cloud API)	✓(CLI, WB Dashboard)	✓(CLI)
Free use	✗	✗	✗	✓

Table 3.1: Multi-Cloud and Hybrid Cloud environment management tools comparison

3.3 Summary

This chapter presented four Multi-Cloud management tools. The first one is Terraform, which is an IAC tool focused on building, changing, and versioning both Cloud and on-premises resources. The second one is Anthos, which allows running Kubernetes and other workloads across on-premises datacenters and multiple Clouds. The third tool is EKS, which is a service used to run Kubernetes on AWS without needing to install and maintain a personal Kubernetes control plane or nodes. The last tool is Ansible, which is a Cloud agnostic open-source and free automation tool focused on the management of remote systems.

A comparison between these tools was performed in this chapter's last section. For this analysis, the following features were used: Multi-Cloud support, Cross-Cloud support, Interoperability approach, Integration, Access and Free use. **This analysis led to the conclusion that Ansible is the most complete tool out of the four tools compared**. The results of the comparative analysis can be consulted in Table 3.1.

Chapter 4

Automatic Deployment Tools

This chapter is composed of the presentation of tools used for automatic deployment. The sections presented in this chapter are the following: Continuous Integration/Continuous Delivery (CI/CD), Kubernetes and GitHub Actions.

4.1 Continuous Integration / Continuous Delivery

CI/CD is a method of frequently delivering apps to customers by automating the stages of app development. Even though CI/CD stands for continuous integration and continuous delivery, another concept attributed to CI/CD is continuous deployment. This method is seen as a solution to the problems caused to development and operations teams when integrating new code into already running apps. Also, CI/CD is known for introducing automation and continuous monitoring throughout the lifecycle of apps, from integration and testing phases to delivery and deployment. These practices taken together are referred to as a CI/CD pipeline [58].

The "CI" stands for continuous integration, which is an automation process for developers. It is considered a successful "CI" when code changes to an app are regularly built, tested and merged into a shared repository. Continuous integration is a good solution to the problem of having various branches of an app in development at once that might conflict with each other [58].

The "CD" either stands for continuous delivery or for continuous deployment, which are automation processes in further stages of a pipeline. Continuous delivery means that the changes made by a developer to an application are automatically tested and uploaded to a repository, where they can be deployed to a production environment by the operation team. The continuous delivery purpose is to ensure that the deployment of new code takes as minimal effort as possible. Regarding continuous deployment, it refers to automatically releasing the changes made by a developer from the repository to production, where the application is usable by customers. This concept's objective is to avoid overloading operations teams with manual processes that slow down app delivery [58].

Figure 4.1 sums up a typical CI/CD flow. The flow begins with continuous integration by building, testing and merging code changes, then follows the continuous delivery by automatically releasing the newly updated code into a repository, and finally follows the continuous deployment by automatically deploying the updated version of the app to production for the customers to use.

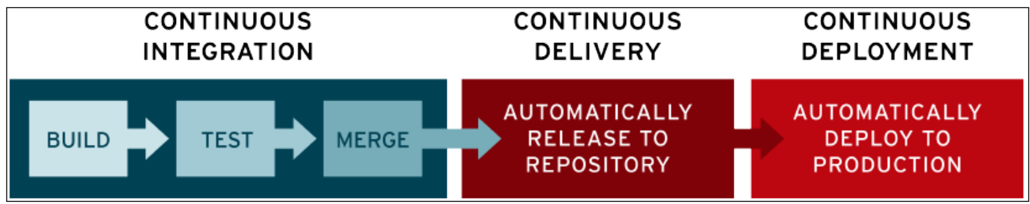


Figure 4.1: CI/CD flow [58]

There are CI/CD tools that help teams to automate their development, deployment and testing. While there are tools that handle all the CI/CD concepts, there are other tools that are specialized either in the integration concept or in the delivery and deployment concepts. Some examples of CI/CD tools are GitHub Actions [24], GitLab [26] and Bitbucket [14]. In addition, teams often use CI/CD tools together with other DevOps tools in order to strengthen their production processes. Automation or orchestration tools like Ansible [6], Chef [18], Docker [21] and Kubernetes [42] are not considered CI/CD tools, but they often appear in many CI/CD workflows [58].

4.2 Kubernetes

Kubernetes, also known as K8s, is an open-source container orchestration tool used for automating deployment, scaling and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Also, Kubernetes gives the freedom to take advantage of on-premises or any type of Cloud infrastructure since it can be run in any virtual or physical environment [42].

Kubernetes brings orchestration features to containerized applications, such as automated rollouts and rollbacks, service discovery (gives Internet Protocol (IP) addresses and Domain Name System (DNS) names to pods) and load balancing across pods, storage orchestration, self-healing, secret and configuration management, and horizontal scaling. Kubernetes components are defined by manifests, which are Yet Another Markup Language (YAML) files containing the detailed configuration of the resources. Some examples of Kubernetes components are namespaces, deployments, pods, services and replica sets [42].

To run Kubernetes it is needed to install a Kubernetes cluster, which is a set of nodes configured to run containerized applications. These clusters are more

lightweight and flexible than Virtual Machines (VMs), so they allow for applications to be more easily developed, moved and managed. Also, unlike VMs, Kubernetes containers are not restricted to a specific Operating System (OS), thus they are able to share OSs and run anywhere. For a production environment, it is a good practice for the cluster to be distributed across multiple worker nodes, while for a testing environment, the cluster's components can all run in the same node [69].

Regarding its composition, Kubernetes clusters are formed with one master node and a number of worker nodes. The master node has the job of controlling the state of the cluster, by deciding which applications are running and their corresponding container images, and also by choosing in which worker node should new application components be created. Besides the master node's main obligation is to control the cluster, it can also be configured to be possible to create new components in it. The worker nodes have the responsibility to run the jobs assigned to them by the master node.

Since a Kubernetes cluster needs a considerable amount of resources in each node, there is a possibility to quickly set a local cluster in a machine with lower settings and with any type of system. To do this, it is used a tool called Minikube [44]. The purpose of this tool is to help new users and students have their first interaction with Kubernetes technology, focusing on making it easy to learn and develop for Kubernetes. To use Minikube, the user's machine needs to have at least the following settings: 2 Central Processing Units (CPUs), 2 Gigabytes (GB) of free memory, 20GB of free disk space, Internet connection and a container or VM manager, such as Docker, Hyper-V or VirtualBox. After installing and starting the Minikube cluster in their machine, the user just needs to write and then apply the manifest files of the components he wants to create in his local cluster [44] [43].

4.3 GitHub Actions

The CI/CD tool analysed in this work is GitHub Actions. This tool was selected due to a previous study performed in the context of the POWER project that compared GitHub Actions with GitLab and Bitbucket and concluded that GitHub Actions was the best CI/CD tool out of the three.

GitHub Actions is the CI/CD tool provided by GitHub. This platform allows the automation of building, testing and deployment pipelines. With this tool, the user can create workflows that build and test pull requests to a repository, or deploy merged pull requests to production. Beyond DevOps, GitHub Actions lets create and run workflows to appropriately label issues that someone made in a repository. To use this tool, it is just needed to have a GitHub repository [25].

GitHub Actions is a tool that brings important benefits to its users. The first one is that assuming that the user is using a GitHub repository to store his code, there is no need to integrate a third-party CI/CD tool with GitHub. The second benefit is that the setup of a pipeline is considerably simple, thus requiring almost

no effort from a user. Finally, since it is a tool designed for developers, there is no need for additional DevOps members in teams whose dedication is to set up and maintain a CI/CD pipeline in a developers project [38].

The machines that run the workflows are called runners. There are two types of runners: the GitHub Runners and the Self-Hosted Runners. A GitHub Runner is a VM provided by GitHub that has the job of simply running workflows. These VMs can either be Linux, Windows, or macOS. These runners are normally used to run building, testing and merging workflows. A Self-Hosted Runner is a machine owned by a user that has to be configured to communicate with GitHub so it can run workflows from the user's repository. This machine can either be the user's personal computer or VM, or a machine located in a datacenter or in the Cloud. To configure a machine as a Self-Hosted Runner, first, the user has to access his repository settings, then click on the "Actions" option listed and then click on the "Runners" option that will appear. After, the user has to click on the "New self-hosted runner" button, then select the image and architecture of his machine and finally copy and run in the machine the commands that will appear after choosing the image and architecture. When configuring the machine, the user can assign a label to the machine, in order to distinguish that machine from other Self-Hosted Runners from that repository. After configuring the machine, it will appear listed in the "Runners" section [25] [23].

The structure of a GitHub Actions workflow consists of events and jobs. An event is what specifies when the workflow will trigger. There are many events possible that can trigger a workflow but the most common ones are: on "push", which is when a push is made to the repository; on "pull_request", which is when a pull request is made to the repository; and on "workflow_dispatch", which allows triggering the workflow via the "Actions" tab of the repository. A job is what is going to be executed in the runners. Each job is composed of a set of steps and by specifying in which runner the job will run. A step can either be a shell command or an "action", which is a custom application for the GitHub Actions platform done by other users that can be reused by others. Steps are executed in order and are dependent on each other. The option that specifies where the job will run is called "runs-on", and its values can be the OSs of the runners provided by GitHub or the labels assigned to a specific Self-Hosted Runner [25].

4.4 Summary

This chapter presented a basis for automatic deployment. It began by introducing the CI/CD concept by stating that is a method of continuously delivering apps to users by using automation to reduce the effort needed to integrate updates and deliver and deploy the final version of the application. After introducing the CI/CD concept, it makes a description of tools normally used in CI/CD. The first tool described was Kubernetes, which is a container orchestration tool used for automating deployment. Its great advantage is being lighter than VMs, thus easing the development, movement and management of applications. The other tool presented was GitHub Actions, which is a CI/CD tool provided by

GitHub that has the goal of automating processes by making use of workflows. Its main benefit is releasing the user from the effort and inconvenience of integrating a third-party CI/CD tool with GitHub to access his repository.

Chapter 5

Problem Analysis

This chapter is composed of an analysis of the problem that motivated the development of this report. It contains the Problem Description, the Requirements Analysis and the Risk Analysis. The first section provides the problem's description, explaining the main challenges regarding the problem presented. The following section presents the requirement analysis, including the identification of both functional and non-functional requirements. The last section provides the risks that can happen during the implementation of the proposed solution.

5.1 Problem Description

Nowadays, the demand for resources to deploy applications and services in the Cloud is increasing in order to enhance the value and expand the reach of the solutions that are being deployed. Taking into consideration the high amount of resources demanded, a single Cloud Service Provider (CSP) might not have the capacity to supply all the resources or features required by companies to fulfil their needs, so to solve this limitation, companies have realized that relying on various CSPs and combining the resources provided by them can become beneficial for their business.

The process of managing and configuring various resources and features provided from different CSPs can become a problem if not done properly. To begin with, it must be ensured that the resources supplied from different places can interoperate and be combined, hence taking full advantage of every resource provided. In addition, managing a huge amount of resources and deploying on different Clouds in a non-automated way can become time-consuming, because each Cloud may have its own deployment procedure. In other words, is like if a system administrator is following a procedure to deploy in Google Cloud, but then has to follow a completely different procedure, and in some cases more complex, to deploy the same product in a different Cloud. The logical solution to solve this issue is to use a framework for the automatic deployment in Cloud and Multi-Cloud environments, with minimum touch management as possible. With this solution, a system administrator could simply press a button to perform a

deployment, regardless of the Cloud where he wants to deploy.

A method to achieve this solution would be to use a combination of Multi-Cloud management tools, to manage and configure the resources from multiple CSPs, with a Continuous Integration/Continuous Delivery (CI/CD) tool, in order to automate as much of the deployment procedure as possible.

5.2 Requirements Analysis

This section performs the identification of the functional requirements and non-functional requirements related to the problem previously described.

5.2.1 Functional Requirements

This section addresses the functional requirements that fall under the scope of the present problem.

5.2.1.1 User Stories

Table 5.1 presents the generic requirements identified in a User Story format. The requirements listed are seen from a system administrator's point of view, thus following the train of thought of the problem's description. The first column indicates what the user intends, while the second column indicates the user's objective.

As a System Administrator, I want...	**So that...**
automate the procedure to perform configurations and deployments	I can reduce the effort needed for these tasks and focus on other assignments.
to use a tool able to manage multiple environments	I don't have to configure a specific tool for a specific environment.
to use a tool that is Cloud agnostic	I freely choose the environments and providers I need without any kind of restrictions.
to use a scalable tool	I can add additional computing resources to my environments without worrying about its configuration effort.

Table 5.1: User Stories

5.2.1.2 High-level requirements

This section lists the requirements that define how the framework must be structured. Some requirements are more important than others, so to define their prioritization is used the Must have, Should have, Can have, Won't have (MoSCoW) prioritization method, which is a four-step approach to prioritize requirements, from most to least important. The four steps of this method are the following [16]:

- **M - Must have:** this step identifies the requirements that are necessary for the successful completion of the project;
- **S - Should have:** this second category specifies which requirements are important but not mandatory to complete the project without impacting it;
- **C - Can have:** this category defines which requirements have a smaller impact when left out of the project;
- **W - Won't have:** this final step includes all the requirements not needed for the project's time frame.

The requirements considered in the development of the automatic deployment solution in Multi-Cloud environments are the following:

1. The framework **must use** a combination of a Multi-Cloud tool with a CI/CD tool;
2. The framework **must be able** to simplify and automate complex procedures by allowing to perform them with just a few steps;
3. The framework **must be able** to deploy to any type of on-premises or Cloud environment;
4. The deployments performed by this framework **must be done** in a Kubernetes cluster;
5. The Kubernetes cluster used **should have** a minimum of two nodes;
6. The framework used **could be able** to install addons in the Kubernetes cluster.

5.2.2 Non-Functional Requirements

This section addresses the non-functional requirements that fall under the scope of the present problem.

- **Security:** Security of the applications deployed due to the information and data stored. The infrastructure and tools must be secure to ensure there is no information or data leak. Security is divided into three attributes:

  - **Confidentiality:** this attribute is related to the absence of protected data by parties unauthorized to access the data. In this case, the credentials to access the tools must be only known by its users;
  - **Integrity:** is related to the improper modification of the systems used. This can be a problem in a Multi-Cloud environment, due to the combination of different systems and resources used;
  - **Availability:** it concerns the readiness to provide what was deployed. In order to guarantee high availability, a backup system must be used beyond the deployed application;
- **Usability:** The user of the Multi-Cloud tool must have management and network knowledge, programming background, and orchestration experience.

5.3 Risk Analysis

Table 5.2 lists the risks identified for the development of the framework. The first column has a description of each Risk. The second column indicates the level of Impact of each risk on a zero (0) to five (5) scale, with zero (0) being "no impact" and five (5) being "high impact". The third column declares the Likelihood of each risk on a zero (0) to five (5) scale, with zero (0) being "not likely" and five (5) being "very likely". The fourth column indicates the Severity of each risk, being the result of the multiplication between the Impact and the Likelihood. The fifth column declares the Consequence of each risk. The last column provides a Mitigation Plan for each risk identified.

Risk	Impact	Likelihood	Severity	Consequence	Mitigation Plan
Not having the budget to pay for a CSP.	4	4	16	Not being able to test the framework in a Public Cloud.	1) Ask for funding from the project POWER partners; 2) Test the framework in on-premises and Private Cloud environments.
Not having the budget to subscribe to a Multi-Cloud tool.	3	4	12	Not being able to test a specific Multi-Cloud tool.	1) Ask for funding from the project POWER partners; 2) Use free tools in the framework.
The resources needed can not be supplied by the provider used.	5	1	10	Not being able to test the framework in a Cloud environment.	Test the framework in an on-premises environment.
Total failover of the provider.	4	1	4	Lose the environment and the applications deployed in there.	Test the framework in an on-premises environment.
Total failover on-premises.	4	2	8	Lose the workflow files, configurations and applications that are running on-premises.	Have a backup of the workflow files in a repository or in an external storage device, to have a recovery point to be able to continue the testing of the framework on another machine.

Table 5.2: Risk analysis

The first risk listed is related to the subscription to a CSP in order to use a Public Cloud for testing purposes. In case there is no budget for a subscription, a possible mitigation plan consists of asking for funding from the project POWER partners. Other possible mitigation consists of performing the testing of the framework in on-premises and Private Cloud environments. This risk occurred during the framework implementation and in Section 6.3.4 is presented the mitigation plan followed.

The second risk describes a similar scenario to the first risk, where there is no budget to use a Multi-Cloud tool. This risk can be mitigated by asking for funding from the project POWER partners, or by using a free tool in the framework. This risk occurred during the framework implementation and in Section 6.3.4 is presented the mitigation plan followed.

The third risk describes a scenario where the provider used is not able to supply the resources needed to test the framework in a Private Cloud environment. In this case, the mitigation plan consists of testing the framework in an on-premises environment

The fourth risk identifies a scenario where occurs a failover in the provider, leading to the loss of the environment and the applications already deployed. The mitigation plane consists of making use of an on-premises environment to perform the framework tests.

The last risk consists of a situation where could happen a problem that could lead to the malfunction of the on-premises environment that was being used for testing. A plan to workaround and prevent this risk consists of having a backup of the files and procedure followed to perform the framework testing, in order to have a recovery point and continue the tests using other hardware.

5.4 Summary

The first section of the chapter provided a description of the problem at hand, pointing out the reasons why the problem should be solved and a method to achieve a possible solution. The second section offered an analysis of the requirements, first by presenting more generic requirements by using User Stories from a system administrator's point of view, and then by listing and prioritizing more specific requirements by using the MoSCoW method. This section also mentioned the non-functional requirements that were not crucial for the framework function. The last section assessed the risks that could possibly occur during the testing of the framework and also provided a mitigation plan in order to be possible to workaround them.

Chapter 6

Proposed Solution

This chapter presents the proposed solution to solve the problem previously described. The sections that compose this chapter are the following: General Architecture, GitHub Actions Workflows and Scenarios. The first section overviews the general architecture of the solution. The second section presents the workflows that were configured to automate the deployment. The last section describes the scenarios provided by Altice Labs (ALB) and also presents the process of implementing and validating each scenario.

6.1 General Architecture

The general architecture of the solution proposed in this chapter has the objective of automatically deploying a testbed application in the Cloud via GitHub. For this, a set of workflows was configured, whose purpose is to contain the commands and configurations needed to deploy a specific testbed application.

As is presented in Figure 6.1, the general idea of the solution is to have a deployment team responsible for the configuration of the tools that will be used for automation and Cloud management. After the tools are configured, they will be used by a development team that will use a Multi-Cloud tool to manage their environment and will also use GitHub Actions to automatically deploy their testbed applications in the Cloud.

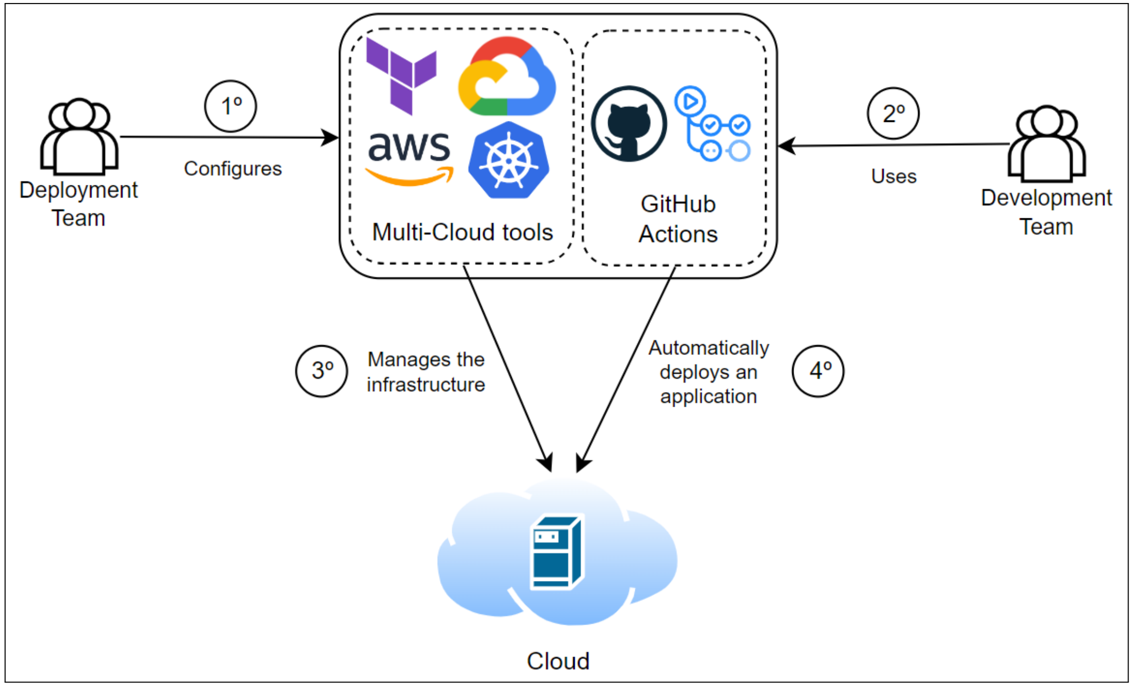


Figure 6.1: General Architecture

6.2 GitHub Actions Workflows

A set of GitHub Actions workflows were configured in order to perform the automation of manual processes. The workflows configured had the objective of deploying and deleting the components of an application and automating the starting, stopping and deleting of a Minikube cluster. The workflows configured are the following:

- main.yml;
- delete_scenario.yml;
- start_minikube_cluster.yml;
- stop_minikube_cluster.yml;
- delete_minikube_cluster.yml;

The main.yml workflow is the main workflow of the ones created because it is the one that applies the manifest files to deploy the Kubernetes components that compose the application. As shown in Figure 6.2, this workflow gets triggered when a push is made to the main branch of the repository, by using the "*push*" option. It can also be manually triggered through GitHub by using the "*workflow_dispatch*" option. Also, this workflow is composed of two jobs. The first one is named "*tests*" and it is responsible for the validation of the manifest files by using the Kubeval tool. In Figure 6.3 it is possible to verify that this job runs in a GitHub Runner. This eliminates the process of installing Kubeval manually in the local Virtual Machine (VM). By running this job in a GitHub Runner, the Kubeval

tool can be easily set up by using the "*lra/setup-kubeval@v1*" action, without being necessary to write anything in the runner's console, such as a password, due to administrator permissions. This job uses the "*actions/checkout@v3*" action to check out the files that are available in the repository. After having access to the manifest files and after Kubeval is set up in the runner, this job runs a set of commands to validate the manifest files, by comparing them with the schemas available in the repository [33]. The second job is named "*deployment*" and it is responsible for the deployment of the components. This job runs in a VM previously configured to be a Self-Hosted Runner so that the components could be created in the VM. As Figure 6.4 shows, this job needs to wait for the "*tests*" job to successfully finish before starting. If the "*tests*" job fails, the "*deployment*" job will not run. The action "*actions/checkout@v3*" is also used so that the "*deployment*" job can access the manifest files, and then the commands needed to apply the manifest files to create the components are run.

```yaml
# Events: Controls when the workflow will run
on:
  # Triggers the workflow on push request events but only for the "main" branch
  push:
    branches: [ "main" ]

  workflow_dispatch:
```

Figure 6.2: main.yml events

```yaml
# Jobs: A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow's first job is called "tests"
  tests:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      #
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - name: Check out code
        uses: actions/checkout@v3

      - name: Setup kubeval
        uses: lra/setup-kubeval@v1

      - name: Validate the Namespace scenario-2
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ scenario-2-namesapace.yaml

      - name: Validate MongoDB Secret
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ mongo-secret.yaml

      - name: Validate MongoDB Deployment and Service
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ mongo.yaml

      - name: Validate ConfigMap
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ mongo-configmap.yaml

      - name: Validate MongoExpress Deployment and Service
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ mongo-express.yaml

      - name: Validate MongoExpress Ingress
        run: kubeval --strict --schema-location https://raw.githubusercontent.com/joaommgoncalves/kubernetes-json-schema/master/ mongo-express-ingress.yaml
```

Figure 6.3: main.yml *test* job

```
#This workflow's second job is called "deployment"
deployment:
  # The type of runner that the job will run on
  runs-on: self-hosted

  needs: tests

  # Steps represent a sequence of tasks that will be executed as part of the job
  steps:
    # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
    - name: Check out code
      uses: actions/checkout@v3


    # Runs these commands using the runners shell

    - name: Create the Namespace scenario-2
      run: kubectl apply -f scenario-2-namesapace.yaml

    - name: Create MongoDB Secret
      run: kubectl apply -f mongo-secret.yaml

    - name: Create MongoDB Deployment and Service
      run: kubectl apply -f mongo.yaml

    - name: Create the ConfigMap, to store the path to the database (database_url)
      run: kubectl apply -f mongo-configmap.yaml

    - name: Create MongoExpress Deployment and Service
      run: kubectl apply -f mongo-express.yaml

    - name: Create MongoExpress Ingress
      run: kubectl apply -f mongo-express-ingress.yaml
```

Figure 6.4: main.yml *deployment* job

The delete_scenario.yml workflow has the function of deleting all the components without deleting the cluster. The configuration is represented in Figure 6.5. This workflow is manually triggered through GitHub and this option was configured by using "*workflow_dispatch*" in the events section. This workflow only has one job, since there is no need to test the manifest files. It runs in a VM previously configured to be a Self-Hosted Runner so that the components could be deleted from the VM. This job uses the "*actions/checkout@v3*" action to access the manifest files and then runs the set of commands to delete the components that were created.

```yaml
name: Delete Scenario 2

# Events: Controls when the workflow will run
on:
    workflow_dispatch:

# Jobs: A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
    # This workflow contains a single job called "delete-scenario-2"
    delete-scenario-2:
      # The type of runner that the job will run on
      runs-on: self-hosted

      # Steps represent a sequence of tasks that will be executed as part of the job
      steps:
        # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
        - name: Check out code
          uses: actions/checkout@v3

        # Runs these commands using the runners shell
        - name: Delete MongoExpress Ingress
          run: kubectl delete -f mongo-express-ingress.yaml

        - name: Delete MongoExpress Deployment and Service
          run: kubectl delete -f mongo-express.yaml

        - name: Delete the ConfigMap, to store the path to the database (database_url)
          run: kubectl delete -f mongo-configmap.yaml

        - name: Delete MongoDB Deployment and Service
          run: kubectl delete -f mongo.yaml

        - name: Delete MongoDB Secret
          run: kubectl delete -f mongo-secret.yaml

        - name: Delete the Namespace scenario-2
          run: kubectl delete -f scenario-2-namesapace.yaml
```

Figure 6.5: delete_scenario.yml

The start_minikube_cluster.yml workflow has the function of starting the local Minikube cluster. This workflow is manually triggered through GitHub and this option was configured by using "*workflow_dispatch*" in the events section. As Figure 6.6 illustrates, this workflow contains a single job called "*start-minikube*", which runs in a Self-Hosted Runner VM, that runs the commands to set VirtualBox as the default driver for Minikube, to start the cluster and to enable the ingress addon. It is important to note that the "$> minikube start " command must be protected so that the process started by this command doesn't get killed by the Runner when the workflow finishes. If the command was not protected, the command would create the cluster but the cluster's state would be "Stopped" instead of "Running" after the workflow is finished. To protect this command is used "RUNNER_TRACKING_ID="" && " before specifying the command, as it can be seen in Figure 6.6.

```
name: Start Minikube Cluster

# Events: Controls when the workflow will run
on:
    workflow_dispatch:

# Jobs: A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
    # This workflow contains a single job called "start-minikube"
    start-minikube:
      # The type of runner that the job will run on
      runs-on: self-hosted

      # Steps represent a sequence of tasks that will be executed as part of the job
      steps:
        #
        - name: Set VirtualBox as driver
          run: minikube config set driver virtualbox

        - name: Start minikube
          run: RUNNER_TRACKING_ID="" && (minikube start)

        - name: Enable ingress addon
          run: minikube addons enable ingress
```

Figure 6.6: start_minikube_cluster.yml

The delete_minikube_cluster.yml workflow has the function of deleting the local Minikube cluster. This workflow is manually triggered through GitHub and this option was configured by using "*workflow_dispatch*" in the events section. As is represented in Figure 6.7, this workflow contains a single job called "*delete-minikube*", which runs in a Self-Hosted Runner VM, that runs the command to delete the cluster.

```
name: Delete Minikube Cluster

# Events: Controls when the workflow will run
on:
    workflow_dispatch:

# Jobs: A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
    # This workflow contains a single job called "delete-minikube"
    delete-minikube:
      # The type of runner that the job will run on
      runs-on: self-hosted

      # Steps represent a sequence of tasks that will be executed as part of the job
      steps:
        #
        - name: Delete minikube
          run: minikube delete
```

Figure 6.7: delete_minikube_cluster.yml

The stop_minikube_cluster.yml workflow has the function of stopping the local Minikube cluster. This workflow is manually triggered through GitHub and this option was configured by using "*workflow_dispatch*" in the events section. As Figure 6.8 illustrates, this workflow contains a single job called "*stop-minikube*", which runs in a Self-Hosted Runner VM, that runs the command to stop the cluster.

```yaml
name: Stop Minikube Cluster

# Events: Controls when the workflow will run
on:
    workflow_dispatch:

# Jobs: A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
    # This workflow contains a single job called "stop-minikube"
    stop-minikube:
      # The type of runner that the job will run on
      runs-on: self-hosted

      # Steps represent a sequence of tasks that will be executed as part of the job
      steps:
        #
        - name: Stop minikube
          run: minikube stop
```

Figure 6.8: stop_minikube_cluster.yml

6.3 Scenarios

This section describes the implementation scenarios provided by ALB. These scenarios were designed and have been provided to be used as a basis for an automatic deployment solution in Cloud and Multi-Cloud environments. There are four scenarios in total, and the main difference between them is where and how they are deployed. The first scenario is to be deployed locally, while the second, third and fourth scenarios are to be deployed in the Cloud. The scenarios were designed and arranged so that the degree of complexity would gradually increase from one to the other, in order to have a step-by-step process to deploy the most complex scenario. The implementation process that was followed to deploy each scenario is reported after their description. The report consists of presenting the steps taken in each scenario, including the tools downloaded and why they were used.

The scenarios are composed of some Kubernetes objects. The objects used are listed in Table 6.1. The table presents the icon, name and description of each object used [46] [40] [45] [41].

Icon	Object	Description
node	Node	Virtual or physical machine that contains the services necessary to run containerized applications
control plane	Control Plane	Manages the worker nodes and the pods in the clusters
ns	Namespace	Used to isolate groups of resources within a single cluster
ing	Ingress	Exposes Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPs) routes from outside the cluster to services within the cluster
svc	Service	Routes the traffic to or between the pods and assigns a permanent Internet Protocol (IP) to a pod
pod	Pod	Used to run containers
rs	ReplicaSet	Maintains the number of stable sets of replica pods running at any given time
deploy	Deployment	Defines the desired state of the pods in the cluster

secret	Secret	Stores and manages sensitive information, such as passwords and Secure Shell (SSH) keys
cm	ConfigMap	Stores some external configuration of an application, such as Uniform Resource Locators (URLs) of a database or service

Table 6.1: Kubernetes objects

6.3.1 Scenario 1

This subsection presents the description, implementation and validation of Scenario 1.

6.3.1.1 Description

The first scenario, named *OnPrem TST environment*, is illustrated in Figure 6.9. The goal of this scenario is to be used as an introduction to the technology that will be used in the subsequent scenarios. This scenario is to be deployed locally and the tool used to deploy it is Minikube, therefore this scenario is seen as a test-minded scenario.

The scenario is composed of the following components: a single node that adopts the control plane role, a namespace, a deployment, a replica set, two pods, a service and an ingress.

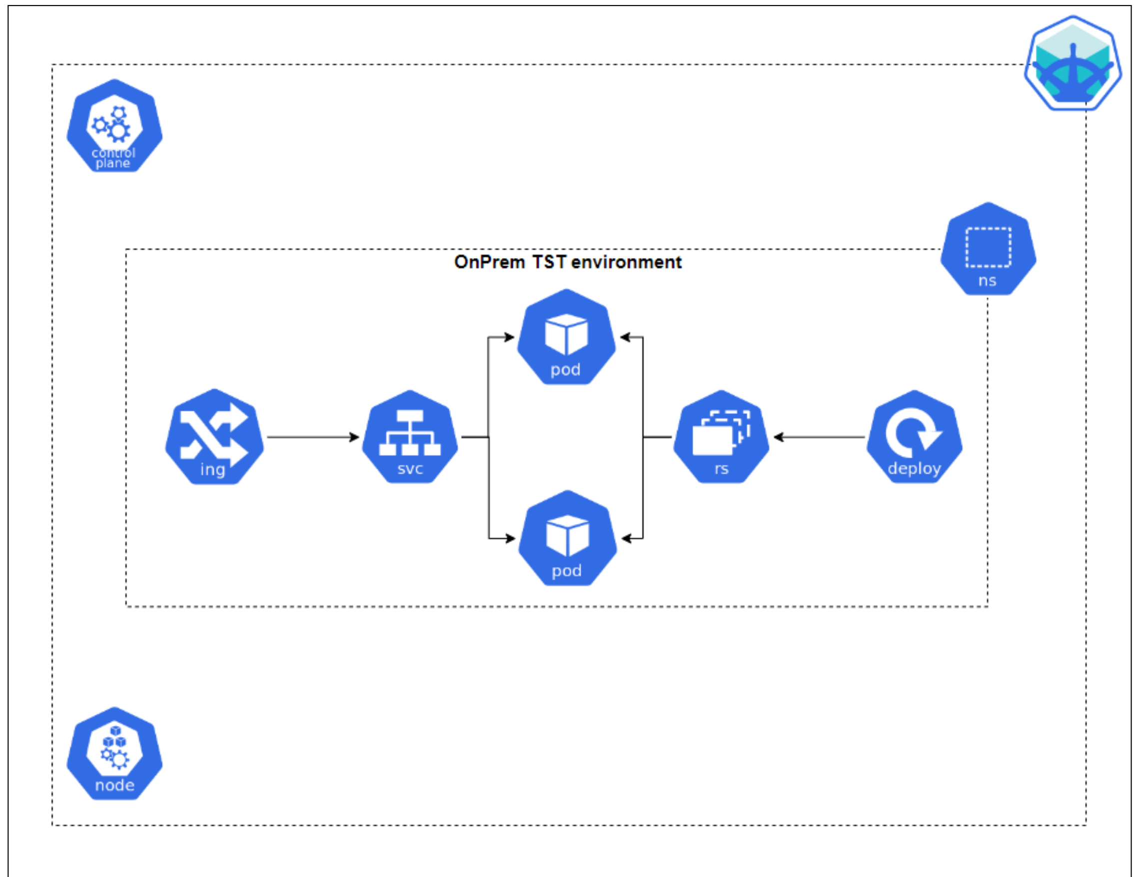


Figure 6.9: Scenario 1

6.3.1.2 Implementation and validation

To begin the deployment of this first scenario, a VM was created in order to have a dedicated machine to run the local cluster. The settings settled for the VM are the following: 2 Central Processing Units (CPUs), 6 Gigabytes (GB) of memory and 30 GB of disk space. An important point is that the VM needs to support virtualization, therefore that option needs to be checked in the VM settings. Also, the Operating System (OS) used was Ubuntu Desktop 22.04.1. After the VM was created, the next step consisted of updating and upgrading it.

The following step regards the installation of the necessary software and tools to deploy the scenario. Firstly, VirtualBox was installed because a hypervisor was needed to virtualize the cluster. Then followed the installation of Minikube, needed to create and start the cluster, and finally the Kubectl tool was installed in order to communicate with the cluster. It was also needed to enable an addon in Minikube in order for the ingress object could work.

After the initial setup was configured, it was previously defined that what was going to be deployed in the pods was an application and a database, being that the application deployed was Mongo Express and the database deployed was MongoDB. Taking that into consideration, there was the need to add two more components to the scenario in addition to those presented in 6.9, namely a secret and a config map. After determining what was going to be deployed in the

pods, it was initiated the creation and configuration of the manifest files needed to create the objects that compose the scenario. The manifest files configured were:

- scenario-1-namespace.yaml;
- mongo-secret.yaml;
- mongo.yaml;
- mongo-configmap.yaml;
- mongo-express.yaml;
- mongo-express-ingress.yaml.

A set of commands was then applied in a certain order to create the scenario. It is important to note that the order of the creation of the components matters because some components need to be created to be referenced later in other components' manifest files. The followed procedure was the following:

1. Start Minikube
   $> minikube start
2. Deploy the Namespace "scenario-1"
   $> kubectl apply -f scenario-1-namespace.yaml
3. Create MongoDB Secret
   $> kubectl apply -f mongo-secret.yaml
4. Create MongoDB Deployment and Service
   $> kubectl apply -f mongo.yaml
5. Create the ConfigMap, to store the path to the database
   $> kubectl apply -f mongo-configmap.yaml
6. Create Mongo Express Deployment and Service
   $> kubectl apply -f mongo-express.yaml
7. Enable the ingress addon on Minikube, to be possible to use this component on a local cluster
   $> minikube addons enable ingress
8. Create Mongo Express Ingress
   $> kubectl apply -f mongo-express-ingress.yaml
9. Run the following command, and then take note of the ingress' IP
   $> kubectl get ingress -n scenario-1
10. Open the "/etc/hosts" file, and add both the IP noted in the previous step and the URL defined in the "mongo-express-ingress.yaml" file
    $> sudo vim /etc/hosts
    - Example of a the line to add: 192.168.100.1 mongoexpress.com

11. Open a browser and then access the *mongoexpress.com* URL defined in the "mongo-express-ingress.yaml" file

Figure 6.10 illustrates the browser page of the Mongo Express application thus validating its deployment.

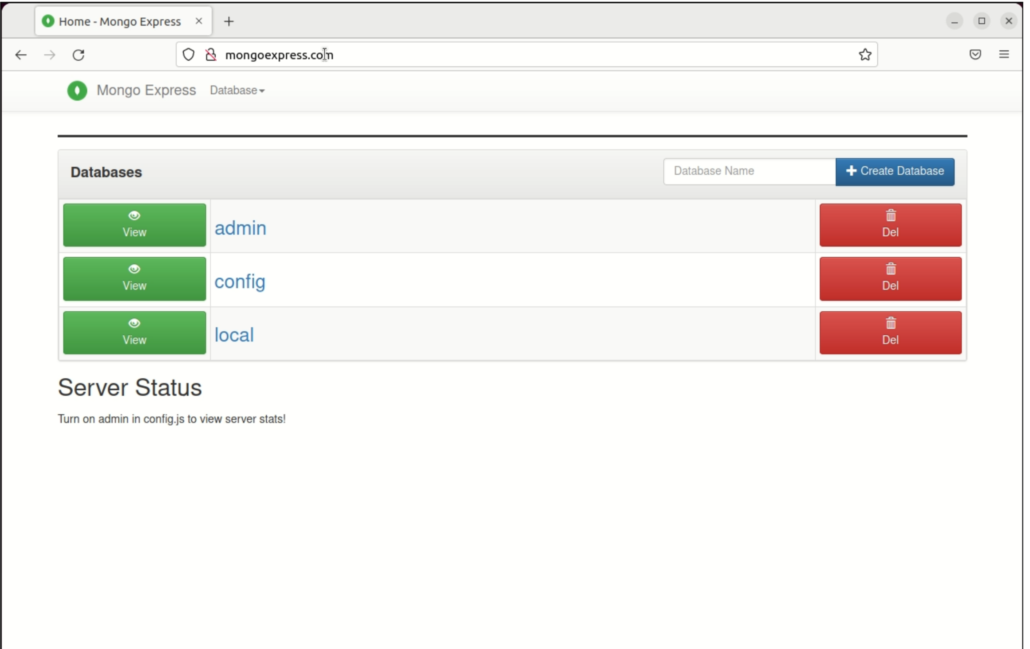


Figure 6.10: Mongo Express browser page

6.3.2 Scenario 2

This subsection presents the description, implementation and validation of Scenario 2.

6.3.2.1 Description

The second scenario, named *OnCloud DEV environment*, is illustrated in Figure 6.11. This scenario has two main goals, one being to add a degree of complexity by introducing a Continuous Integration/Continuous Delivery (CI/CD) tool to automate processes that were done manually in the previous scenario, and the other is to initiate the process of deployment into a Cloud. This scenario is to be deployed locally first and then in the Cloud. The tool used to perform the automation of processes is GitHub Actions. In there it will be configured a set of workflows responsible for the automatic deployment of the scenario. Minikube will also be used to create the cluster where the scenario will be deployed.

The scenario is composed of the following components: two nodes where one has the control plane role, a namespace, a deployment, a replica set, two pods, a service and an ingress.

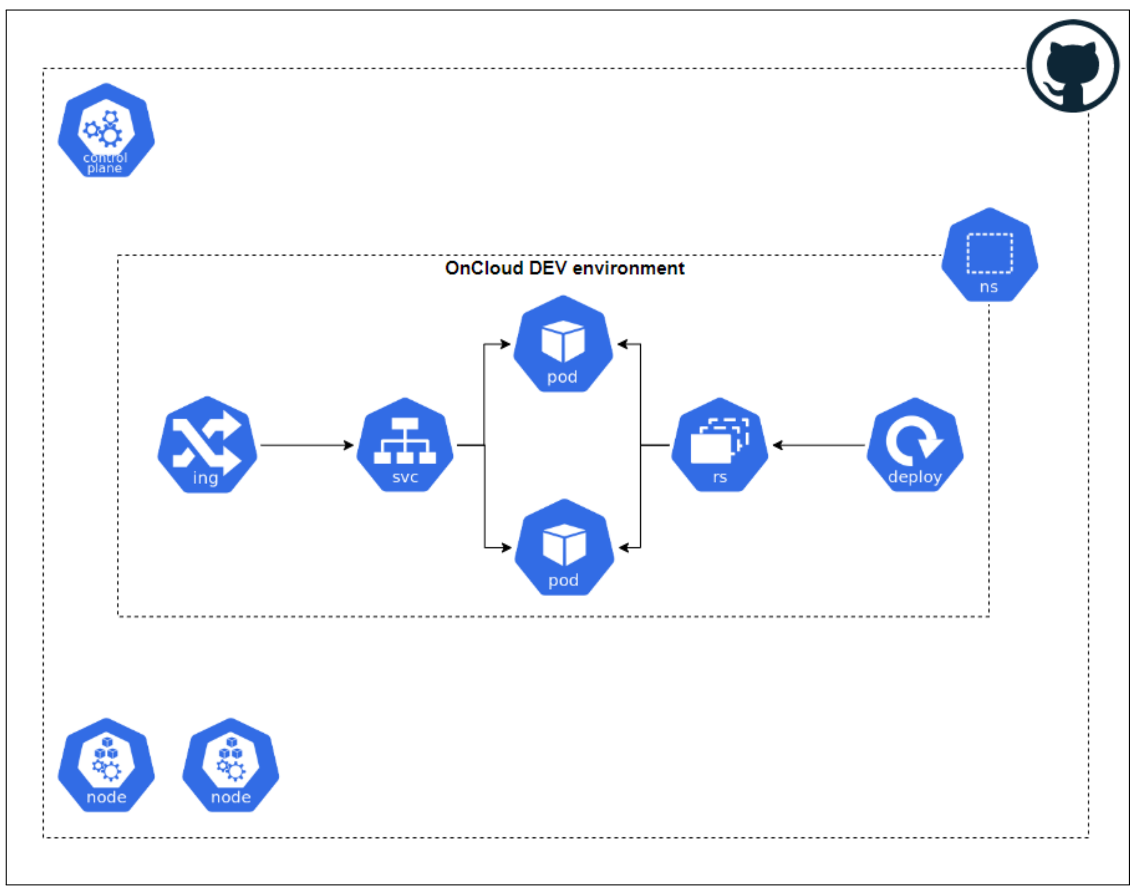


Figure 6.11: Scenario 2

6.3.2.2 Implementation and validation

The deployment of this scenario was divided into two parts. The first part is deploying the scenario via GitHub Actions into a Minikube cluster running in a local VM. The second part is deploying the scenario via GitHub Actions into a Minikube cluster running in a VM located in a Cloud.

To begin the deployment of the **first part**, a VM identical to the one used in the previous scenario was created in order to have a dedicated machine to deploy the scenario locally. The software and tools installed in this VM were the same as those installed in the previous VM. After the machine was ready, it was created a repository in GitHub with all the manifest files needed to create Scenario 2. The manifest files configured were:

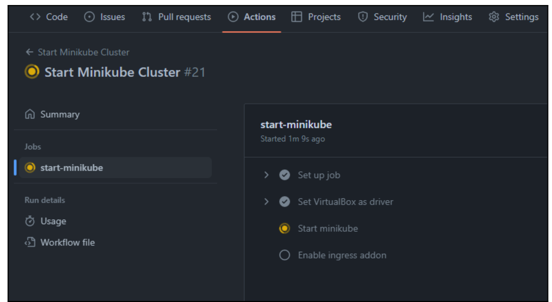
- scenario-2-namespace.yaml;
- mongo-secret.yaml;
- mongo.yaml;
- mongo-configmap.yaml;
- mongo-express.yaml;
- mongo-express-ingress.yaml.

As it was intended to deploy the scenario in a local VM, it was necessary to configure the VM to become a Self-Hosted Runner so that GitHub could communicate directly with it and in order to be possible to run locally the repository's workflows that would apply the manifest files to create the scenario. The GitHub's official documentation [23] was followed to configure the VM as a Self-Hosted Runner.

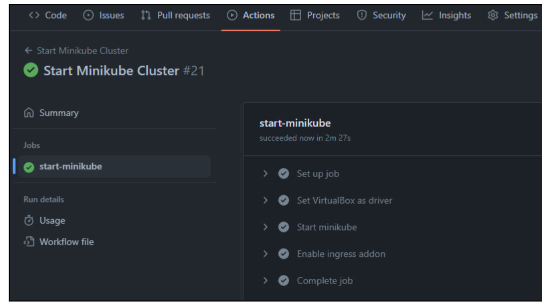
Afterwards, a set of workflows was used with the aim of replacing some manual steps that were taken to create the first scenario. The workflows used are the following:

- main.yml;
- delete_scenario_2.yml;
- start_minikube_cluster.yml;
- delete_minikube_cluster.yml;
- stop_minikube_cluster.yml.

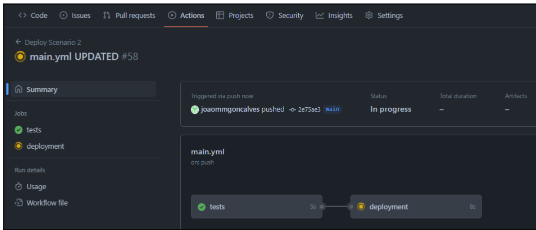
To deploy Scenario 2 via GitHub, start_minikube_cluster.yml and main.yml were executed in this specific order, the first to create the local cluster and the second to deploy the components. Figure 6.12 shows what GitHub presents when the workflows used to deploy Scenario 2 are running and also when they successfully finish running.



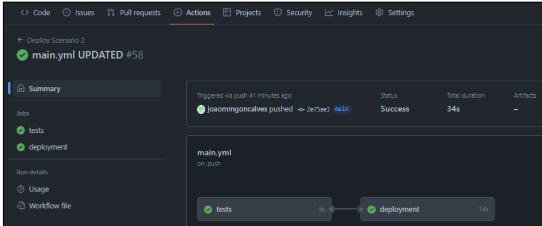
(a) Scenario 2 start_minikube_cluster.yml running



(b) Scenario 2 start_minikube_cluster.yml done



(c) Scenario 2 main.yml running



(d) Scenario 2 main.yml done

Figure 6.12: Scenario 2 workflows in GitHub

After Scenario 2 was deployed via GitHub, the 9th, 10th and 11th steps that were done in Scenario 1 were replicated in order to be possible to open the Mongo Express application page in the browser. Figure 6.11 illustrates the browser page of the Mongo Express application, which validates this deployment.

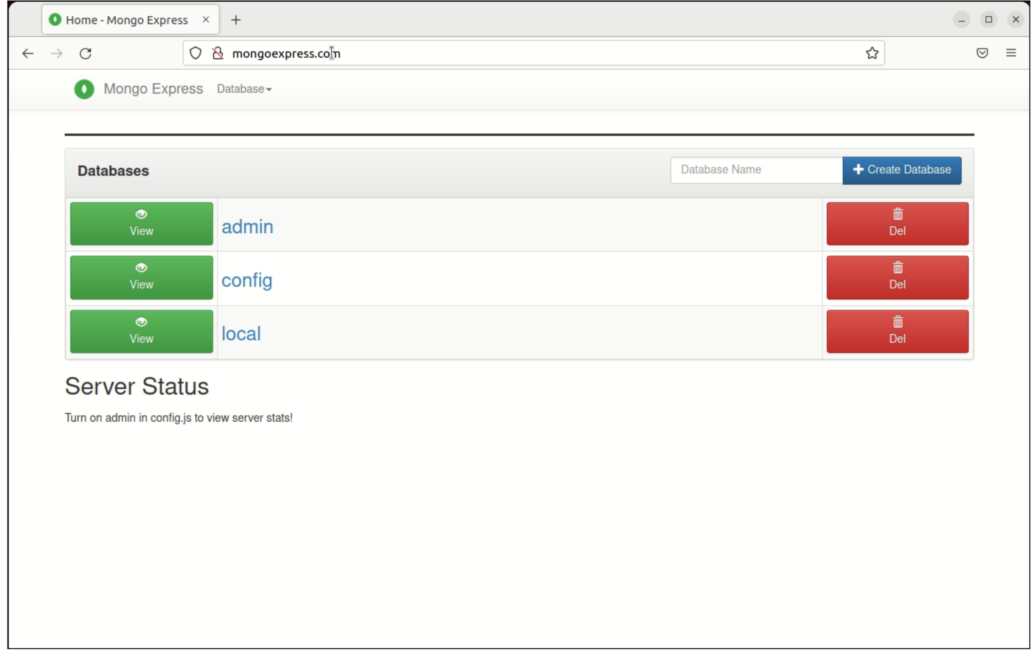


Figure 6.13: Mongo Express browser page

To deploy the **second part**, it was created a VM in Department of Informatics Engineering (DEI)'s Cloud, using the Xen Orchestra hypervisor. Since DEI could provide more resources, this VM was created with different settings than the ones used previously. The settings settled for this VM are the following: 6 CPUs, 16 GB of memory and 100 GB of disk space. Taking into consideration the available templates provided by DEI to create VMs, the OS used was Ubuntu Desktop 20.04. After the VM was created, the nested virtualization was enabled and then the VM was updated and upgraded.

Regarding the software and tools needed to create the scenario, its was installed Docker, Minikube and Kubectl. In this case, Docker was used instead of VirtualBox because the nested virtualization feature in the Xen Orchestra hypervisor is very unstable and has problems supporting other hypervisors.

Afterwards, the VM was configured to become a Self-Hosted Runner so that it could communicate with GitHub. The GitHub's official documentation [23] was followed to configure the VM as a Self-Hosted Runner. Since the VM used in the first part was already listed as a Runner in the repository, this new VM was configured with a new label, in order to distinguish the two VMs in the workflow configurations. The label configured was "vm-dei-cloud" and in Figure 6.14 can be seen, after the VMs names, which are the labels used to identify each Self-Hosted Runner.

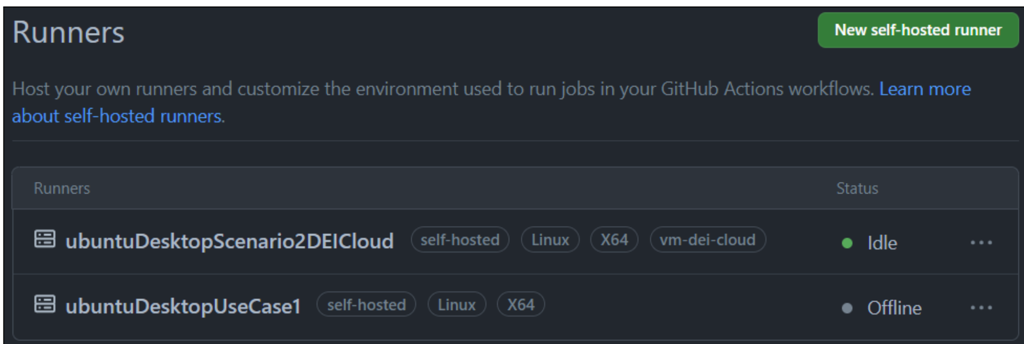


Figure 6.14: Self-Hosted Runners of Scenario 2

The workflows used in this second part were the same as the ones used in the first part, but in this case, the "runs-on" option was modified in all of them so that the workflows would run in the VM located in DEI's Cloud. The start_minikube_cluster.yml workflow was also modified so that it would set Docker as the default driver for Minikube (`$> minikube config set driver docker`) and that it creates the cluster with two nodes (`$> RUNNER_TRACKING_ID="" && (minikube start –nodes 2)`).

To deploy Scenario 2 via GitHub, start_minikube_cluster.yml and main.yml were executed in this specific order, the first to create the local cluster and the second to deploy the components. After the scenario was deployed via GitHub, the 9th, 10th and 11th steps that were done in Scenario 1 were replicated in order to be possible to open the Mongo Express application page in the browser. The

browser page of the deployed Mongo Express application is identical to the one shown in Figure 6.13, completing the validation of this scenario.

6.3.3 Scenario 3

This subsection presents the description, implementation and validation of Scenario 3.

6.3.3.1 Description

The third scenario, named *OnCloud QA environment*, is illustrated in Figure 6.15. The goal of this scenario is to introduce the deployment in a Kubernetes cluster, therefore it is to be deployed in a Kubernetes cluster, located whether in a Private or Public Cloud. GitHub Actions is also used in this scenario in order to automate processes. This scenario is indicated to be a production environment-minded scenario, thus Minikube is not used in this case.

The scenario is composed of the following components: two nodes where one has the control plane role, a namespace, a deployment, a replica set, two pods, a service, and an ingress.

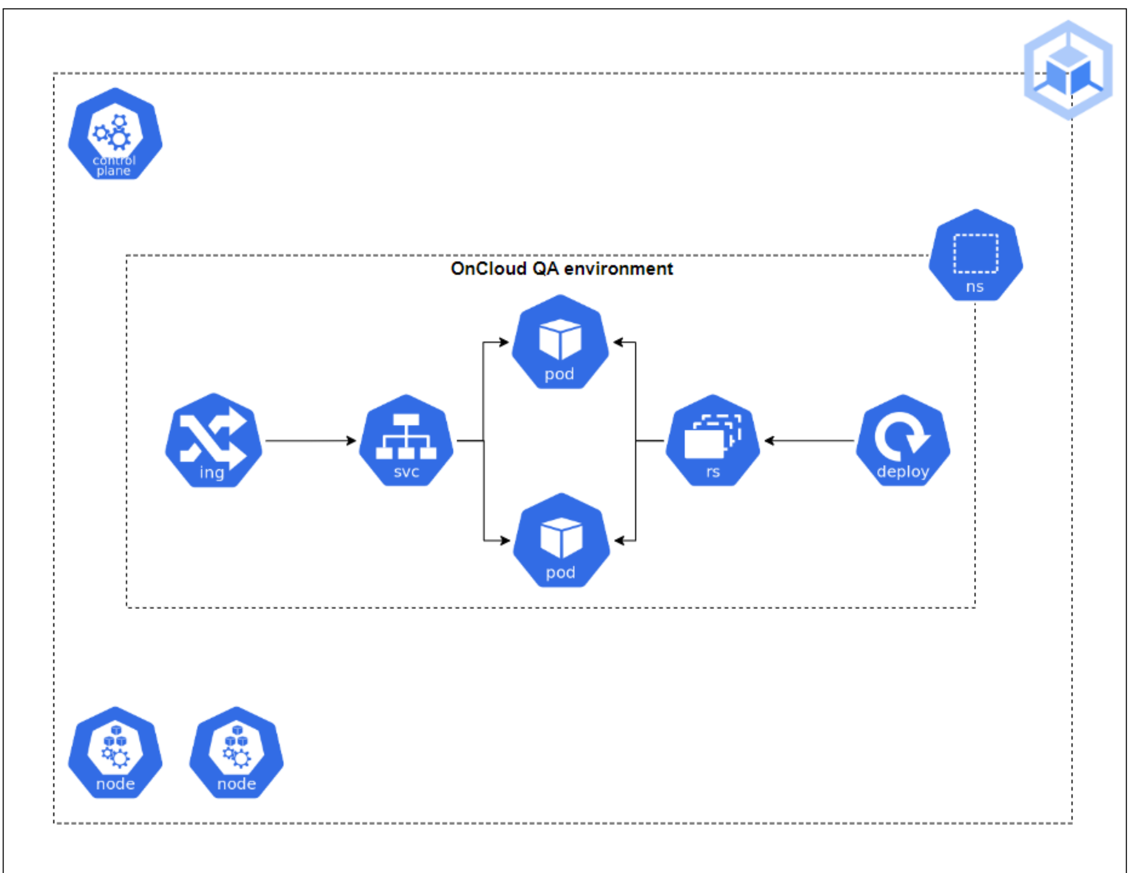


Figure 6.15: Scenario 3

6.3.3.2 Implementation and validation

To begin the deployment of this scenario, it was needed to access a Kubernetes cluster. The cluster was provided by DEI, thus is located in DEI's Cloud. The VMs where the cluster was running had the following settings: 4 CPUs, 16 GB of memory, 60 GB of disk space and Ubuntu Server 20.04.3 Long Term Support (LTS) as the OS. Also, a GitHub repository was created with all the manifest files needed to create Scenario 3. The manifest files used were:

- scenario-3-namespace.yaml;
- mongo-secret.yaml;
- mongo.yaml;
- mongo-configmap.yaml;
- mongo-express.yaml;
- mongo-express-ingress.yaml.

The VM that has the cluster's control plane role was configured to become a Self-Hosted Runner so that GitHub could communicate directly with it and in order to be possible to run locally the repository's workflows that would apply the manifest files to create the scenario. The GitHub's official documentation [23] was followed to configure the VM as a Self-Hosted Runner. This VM was configured with a label called "control-plane", in order to specify its function. Figure 6.16 shows, after the VMs names, which are the labels used to identify the Self-Hosted Runners.

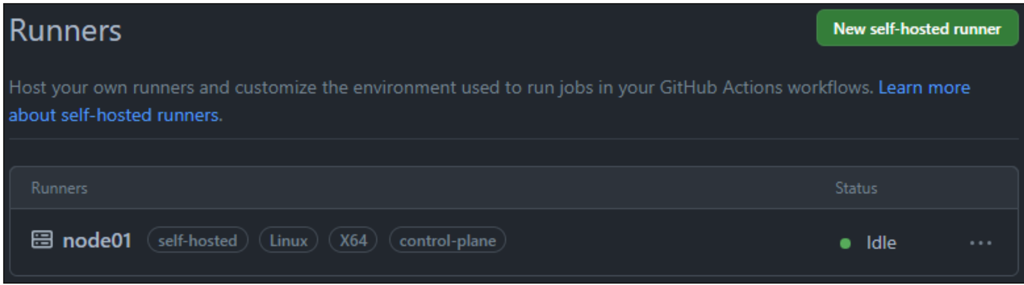
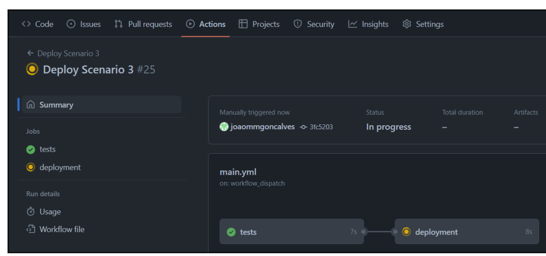


Figure 6.16: Self-Hosted Runners of Scenario 3

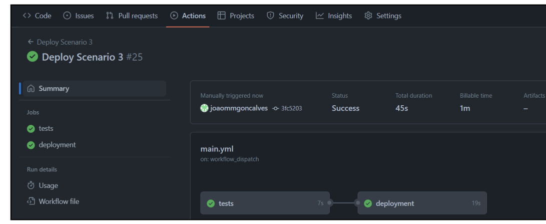
Afterwards, the workflows needed for this scenario were created in the repository. Since this scenario does not use a Minikube cluster, there was no need to create workflows for starting, stopping or deleting a Minikube cluster. Also, since the Kubernetes cluster is supposed to be already operational, there was also no need to create a workflow to start the cluster. For the workflows used in this scenario, the "runs-on" option was modified so that the workflows would run in the control plane VM located in DEI's Cloud. The workflows used are the following:

- main.yml;
- delete_scenario_3.yml;

To deploy Scenario 3 via GitHub, the main.yml workflow was executed to deploy the components. In Figure 6.17 it is possible to see an illustration of what GitHub presents when the workflows used to deploy Scenario 3 are running and also when they successfully finish running.



(a) Scenario 3 main.yml running



(b) Scenario 3 main.yml done

Figure 6.17: Scenario 3 workflows in GitHub

After Scenario 3 was deployed via GitHub, the "hosts" file of the host computer was edited with the control plane's IP and the URL defined in the "mongo-express-ingress.yaml" in order to be possible to open the Mongo Express application page in the browser. Figure 6.18 shows the browser page of the Mongo Express application deployed, completing the validation of this scenario.

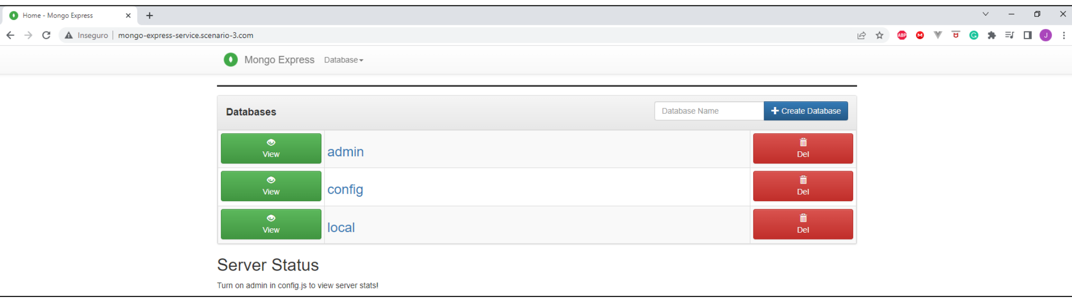


Figure 6.18: Scenario 3 Mongo Express browser page

6.3.4 Scenario 4

This subsection presents the description, implementation and validation of Scenario 4.

6.3.4.1 Description

The fourth scenario, named *OnCloud Tenant PROD environment*, is illustrated in Figure 6.19. This scenario has two main goals, one being to introduce the deployment in a Multi-Cloud environment and the other is to introduce the use of

a Multi-Cloud management tool. This scenario is to be deployed in a Kubernetes cluster located in a Multi-Cloud and is indicated to be a production environment-minded scenario. Similar to the previous scenario, GitHub Actions is also meant to be used to automate processes.

The scenario is composed of the following components: two nodes where one has the control plane role, a namespace, a deployment, a replica set, two pods, a service, and an ingress.

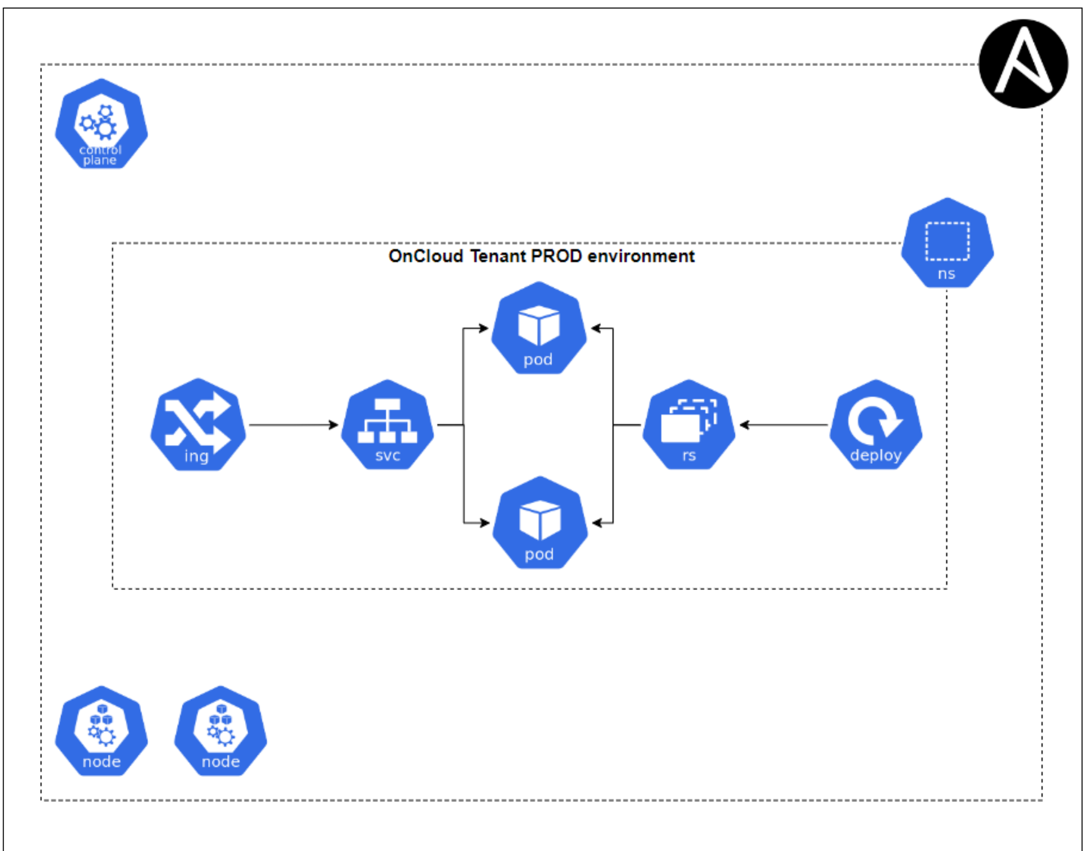


Figure 6.19: Scenario 4

6.3.4.2 Implementation and validation

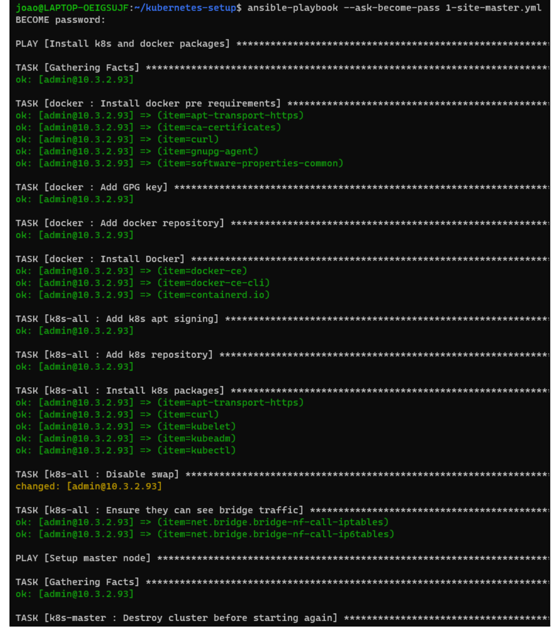
Since there was no budget provided to subscribe to Cloud Service Providers (CSPs), this scenario had to be deployed in DEI's Cloud and not in a Multi-Cloud environment, thus following the mitigation plan defined for this risk. Also, since it was not possible to pay for a Multi-Cloud tool subscription, the tool used in this scenario had to be a free tool, which in this case was Ansible. Ansible was used with the goal of automating the process of installing and creating a Kubernetes cluster in the VMs running in the Cloud. The VMs used for this scenario had the following settings: 4 CPUs, 8 GB of memory, 50 GB of disk space and Ubuntu Server 20.04.6 LTS as the OS.

Before starting the deployment of the scenario, Python and Ansible were installed on a computer with Windows under a Windows Subsystem for Linux (WSL) distribution, thus adopting the control node role. To be possible to install WSL, the "VM Platform" and "WSL" features need to be checked in the "Turn Windows features on or off" menu. After that, the "Ubuntu" and "WSL" programs were installed via the Microsoft Store. Then, Python was also installed in the VMs that were going to be managed. After the requirements for each node were fulfilled, it was generated a key pair in the control node for the SSH connections and then the pub key was copied to the managed nodes. To finalize the setup is was created the working directory in the control node and inside it is was created the inventory and ansible.cfg files.

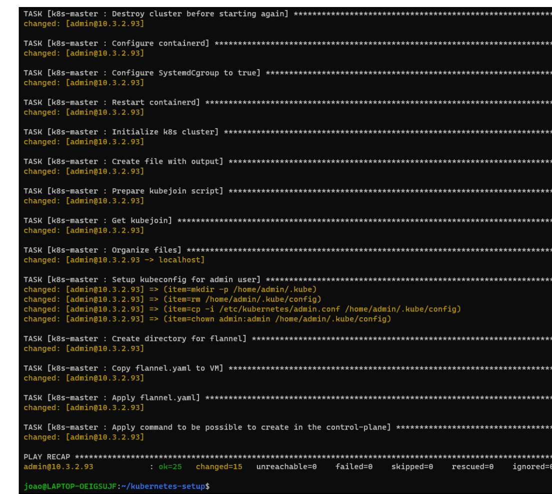
After the setup was finalized, it was configured the Ansible Playbooks needed to install a Kubernetes cluster in the managed nodes. The Playbooks were configured by using the Ansible Roles concept, thus separating the configurations into files organized in a directory structure. The roles created perform the following main tasks:

- Install Docker;
- Install Kubernetes packages;
- Destroy a Kubernetes cluster before creating a new one;
- Initialize a Kubernetes cluster;
- Apply the command to enable the deployment in the control plane.

Figure 6.20 illustrates what is shown in the control node console when the Playbook is running. The green lines mean that the changes performed by that task were already done in the managed node, so there is no need to re-apply those tasks. The yellow lines mean that those tasks performed a configuration change in the managed node.



(a) Ansible Playbook execution (Part 1)



(b) Ansible Playbook execution (Part 2)

Figure 6.20: Scenario 4 Ansible Playbook execution

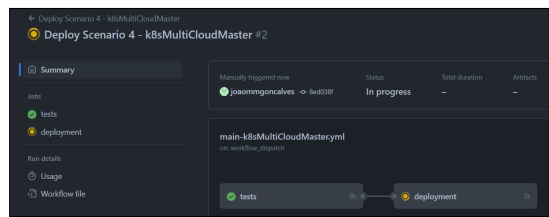
After Ansible finished running the Roles and the Kubernetes cluster was running, a GitHub repository was created with all the manifest files needed to create Scenario 4. Then the control plane VM was configured to become a Self-Hosted Runner. The GitHub's official documentation [23] was followed to configure the VM as a Self-Hosted Runner. This VM was configured with the labels "control-plane" and "master", in order to specify its function. The manifest files available in the repository were:

- scenario-4-namespace.yaml;
- mongo-secret.yaml;
- mongo.yaml;
- mongo-configmap.yaml;
- mongo-express.yaml;
- mongo-express-ingress.yaml.

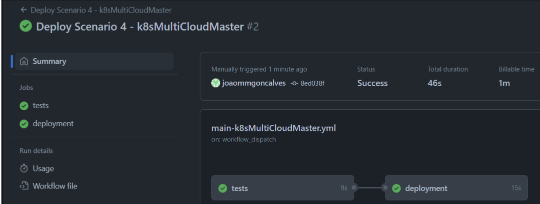
Afterwards, the workflows needed for this scenario were created in the repository. The "runs-on" option was modified in all the workflows so that they would run in the control plane VM. The workflows used are the following:

- main.yml;
- delete_scenario_4.yml;

To deploy Scenario 4 via GitHub, the main.yml workflow was executed to deploy the components. In Figure 6.21 it is possible to see an illustration of what GitHub presents when the workflows used to deploy Scenario 4 are running and also when they successfully finish running.



(a) Scenario 4 main.yml running



(b) Scenario 4 main.yml done

Figure 6.21: Scenario 4 workflows in GitHub

To verify the final state of the scenario, the VMs were accessed and in them was run the "$> kubectl get all -n scenario-4 " command to check if the components were created. In Figure 6.22, it is possible to see the components deployed and running in the cluster, thus validating the framework for automatic deployment. Even though the practical work to validate the framework was only performed in a Private Cloud, in theory, the framework can also be applied in a Multi-Cloud environment because the tools that were used (GitHub Actions and Kubernetes) are Cloud agnostic, so they do the same tasks whatever the type of Cloud that is being used.

```
admin@templateubuntu20:~$ kubectl get all -n scenario-4
NAME                                   READY   STATUS    RESTARTS   AGE
pod/mongo-express-859f75dd4f-hxw7h     1/1     Running   0          3m27s
pod/mongodb-deployment-699744c7d-dc27f 1/1     Running   0          4m10s

NAME                            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
service/mongo-express-service   ClusterIP   10.105.210.2    <none>        8081/TCP    3m27s
service/mongodb-service         ClusterIP   10.111.151.213  <none>        27017/TCP   4m10s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mongo-express        1/1     1            1           3m27s
deployment.apps/mongodb-deployment   1/1     1            1           4m10s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/mongo-express-859f75dd4f       1         1         1       3m27s
replicaset.apps/mongodb-deployment-699744c7d   1         1         1       4m10s
admin@templateubuntu20:~$
```

Figure 6.22: Scenario 4 deployed in a Kubernetes cluster running in DEI's Cloud

6.4 Summary

This chapter began by presenting the general architecture of the framework, by explaining its components and also the process of performing the automatic deployment of an application in the Cloud. The following section presented the GitHub Actions workflows that were configured to automate processes. Those

workflows can deploy and delete a scenario and can start, stop and delete a Minikube cluster. Then, the presentation of the scenarios that were used to perform the framework validation was made. The first scenario, which was deployed in a local VM, had the goal of introducing the Kubernetes technology. The second scenario, which was deployed first in a local VM but then in a VM located in DEI's Cloud, had the goal of introducing the CI/CD tool used to automate the deployment of the scenario. The CI/CD tool used was GitHub Actions. The third scenario, which was deployed in DEI's Cloud, had the goal of introducing the deployment in a Kubernetes cluster. Finally, the last scenario, which was also deployed in DEI's Cloud, had the goal of introducing the Multi-Cloud tool used to automate the configuration of the VMs that compose the scenario infrastructure. In this scenario, the tool used was Ansible.

Chapter 7

Conclusion

The framework presented in this internship report has the goal of automating the deployment of applications into a Multi-Cloud environment. The development process of this framework was divided into two main parts, which are the literature review and the practical segment. The literature review was made with the goal of understanding the theoretical concepts that cover the theme of this internship's work. The practical segment consisted of the design, implementation and testing of the framework.

The first topic addressed in the literature review was the Cloud Computing Paradigm. The main information provided by this chapter is the comparison between Hybrid Clouds and Multi-Clouds. The comparison led to the conclusion that the principal difference between them is that while Multi-Clouds are composed of at least two Public Clouds from different vendors, Hybrid Clouds need to have at least a Private Cloud integrated in its combination.

After concluding the Cloud Computing review to understand what is a Multi-Cloud environment, it followed the review on management tools for Multi and Hybrid Clouds. The tools reviewed were Terraform, Anthos, Elastic Kubernetes Service (EKS) and Ansible, leading to performing a comparison between them. The comparison induces the conclusion that Ansible is a complete and usable tool characterized by being Cloud agnostic and free to use without requiring any kind of subscription.

The last section reviewed was the tools used for automatic deployment. Firstly, it was introduced the Continuous Integration/Continuous Delivery (CI/CD) concept. After, it was made the presentation of tools used CI/CD workflows. The first tool introduced was Kubernetes and the second tool presented was GitHub Actions. After concluding the review of these tools, it was concluded that both could be used together in an automatic deployment solution, by using GitHub Actions to automate the deployment process of an application and by using Kubernetes to create the containerized environment where the application was going to be deployed.

After the literature review was finished, started the practical segment of the work. The process of reaching the final state of the framework was divided into

incremental steps. The first incremental step consisted of having the first contact with the Kubernetes technology. To do this was deployed an application in a Minikube cluster running in a local Virtual Machine (VM). The following step introduced automation and deployment in the Cloud. This was accomplished by automatically deploying an application, using GitHub Actions, in a Minikube cluster running in a VM located in Department of Informatics Engineering (DEI)'s Cloud. The third step consisted of performing the automatic deployment into a Kubernetes cluster, thus upgrading the type and complexity of the cluster that was being used. In this step, an application was automatically deployed into a Kubernetes cluster provided by DEI that was running in their Cloud. The last step introduced the use of a Multi-Cloud management tool. Here, the purpose of using the Multi-Cloud management tool was to automate the installation and creation of the Kubernetes cluster in the VMs provided by DEI. The Multi-Cloud management tool selected for this step was Ansible. To perform the Kubernetes cluster creation, Ansible Playbooks were configured by using the Ansible Roles concept. After the cluster was created, GitHub Actions was used once again to perform the automatic deployment of an application in the cluster created. As the application was automatically deployed to the created Kubernetes cluster with success, it can be concluded that not only Kubernetes and GitHub Actions can indeed work together, but also the framework was successfully validated.

During the development process of this framework, it appeared a set of restrictions that put into question the development and validation of the framework. These limitations were not having a budget provided to subscribe to Cloud Service Providers (CSPs) and to pay for a Multi-Cloud tool subscription. To overcome the first difficulty that appeared, which did not allow to deploy into a Public Cloud, the deployment of the last step had to be performed in DEI's Cloud. Also, to overcome the second limitation that occurred it was used a free Multi-Cloud management tool.

Regarding the objectives defined at the beginning of this work, it can be stated that the framework developed performs its main objective, which is performing the automatic deployment of an application into Cloud and Multi-Cloud environments by using a CI/CD tool to automate processes and a Multi-Cloud management tool to configure the environment used. This is supported by the accomplishment of the goals defined to achieve the main objective. The comparison to distinguish a Multi-Cloud from a Hybrid Cloud is presented in Table 2.1, thus completing the first goal. The identification and analysis of management tools used to manage the environment used was accomplished in Chapter 3. Finally, the last goal was achieved by performing and validating the automatic deployment framework by using the scenarios and Clouds provided by the POWER project's members.

The advantages that this framework brings to its users are not just to reduce the effort to perform the deployment of applications but also to provide a simple procedure to configure the environments where the applications' deployment is going to be performed. The usage of this framework will reduce the amount of human errors that could appear while performing the deployment of an application and the configuration of its environment, by using automation processes.

Furthermore, this framework can improve Altice Labs (ALB)'s workflows by easing the integration of new development team members because they do not need to have previous knowledge of the tool used.

Considering the literature review, it can be declared that the work presented can be a useful starting point for other researches related not just to Cloud computing, but also to management tools for Multi and Hybrid Clouds. Regarding the practical segment, despite the limitations that did not allow testing the framework in a Public Cloud, it can still be affirmed that the framework developed is a valid automatic deployment solution for Multi-Cloud environments. This statement is supported by the fact that the tools used in the framework (GitHub Actions, Kubernetes and Ansible) are all Cloud agnostic tools, so the practical validation that was done in DEI's Cloud can theoretically be done in a Multi-Cloud environment.

Taking the initial internship proposal into consideration, the next step regarding the usage of this framework consists of arranging the means to have access to a Public Cloud, in order to be possible to use the framework in a proper Multi-Cloud environment. Also, another step to be done in the future is to integrate this framework in other POWER subprojects, by using it to automate the deployment of applications to be used as testbeds for those subprojects. Furthermore, other future work that can be done with this solution is testing and integrating other tools that could allow the improvement of the Kubernetes deployment in Multi-Cloud environments by using a federated Kubernetes approach, thus permitting the management of multiple clusters from a single control point.

References

[1] https://www.alticelabs.com.

[2] https://www.it.pt/ITSites/Index/3,.

[3] https://www.ipn.pt,.

[4] https://www.uc.pt.

[5] Bashair Abdullah M Algarni. Managing deployed containerized web application on aws using eks on aws fargate. pages 1–39, 2021.

[6] Ansible. Ansible. https://www.ansible.com,. [Online; accessed 9-January-2023].

[7] Ansible. https://docs.ansible.com/ansible/latest/getting_started/index.html,. [Online; accessed 20-August-2023].

[8] Ansible. https://www.ansible.com/overview/it-automation,. [Online; accessed 22-August-2023].

[9] AWS. Amazon eks features. https://aws.amazon.com/eks/features,. [Online; accessed 28-December-2022].

[10] AWS. Amazon elastic kubernetes service (eks). https://aws.amazon.com/eks/?nc1=h_ls,. [Online; accessed 21-December-2022].

[11] AWS. Aws outposts family. https://aws.amazon.com/outposts,. [Online; accessed 28-December-2022].

[12] AWS. Amazon elastin kubernetes service (eks). https://aws.amazon.com/eks,. [Online; accessed 8-January-2023].

[13] AWS. What is amazon eks? https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html,. [Online; accessed 14-December-2022].

[14] Bitbucket. Bitbucket. https://bitbucket.org. [Online; accessed 22-August-2023].

[15] Yevgeniy Brikman. Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation. pages 1–11, 2016.

[16] Kate Brush. Moscow method. https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method. [Online; accessed 24-August-2023].

[17] Leonardo Carvalho and Aleteia Araujo. Performance comparison of terraform and cloudify as multi-cloud orchestrators. pages 1–10, 2020.

[18] Chef. Chef. https://www.chef.io. [Online; accessed 9-January-2023].

[19] Cloudify. Cloudify. https://cloudify.co. [Online; accessed 9-January-2023].

[20] dinCloud. How does cloud bursting work and its pros n cons? https://www.dincloud.com/blog/how-does-cloud-bursting-work-and-its-pros-n-cons. [Online; accessed 21-July-2023].

[21] Docker. Docker. https://www.docker.com. [Online; accessed 22-August-2023].

[22] Charles Ferrari, Benedek Kovács, Melinda Tóth, Zoltán Horváth, and Anna Reale. Edge computing for communication service providers: A review on the architecture, ownership and governing models. pages 1–6, 2021.

[23] GitHub. Adding self-hosted runners. https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/adding-self-hosted-runners, . [Online; accessed 11-April-2023].

[24] GitHub. Github. https://github.com/features/actions, . [Online; accessed 22-August-2023].

[25] GitHub. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions, . [Online; accessed 23-August-2023].

[26] GitLab. Gitlab. https://about.gitlab.com. [Online; accessed 22-August-2023].

[27] Google. Anthos. https://cloud.google.com/anthos. [Online; accessed 8-January-2023].

[28] Google Cloud. Anthos. https://cloud.google.com/anthos, . [Online; accessed 30-November-2022].

[29] Google Cloud. Evaluate vms with fit assessment. https://cloud.google.com/anthos/a4vm/docs/how-to/fit_assessment, . [Online; accessed 26-December-2022].

[30] Google Cloud. Google anthos: Overview. https://cloud.google.com/anthos/clusters/docs/multi-cloud, . [Online; accessed 30-November-2022].

[31] Google Cloud. What is a hybrid cloud? https://cloud.google.com/learn/what-is-hybrid-cloud, . [Online; accessed 30-October-2022].

[32] Sumit Goyal. Public vs private vs hybrid vs community - cloud computing: A critical review. pages 5–6, 2014.

[33] Yann Hamon. kubernetes-json-schema. https://github.com/joaommgoncalves/kubernetes-json-schema. [Online; accessed 11-May-2023].

[34] HashiCorp. Unlocking the cloud operating model: Cloud compliance and management. pages 1–13, .

[35] HashiCorp. What is terraform. https://developer.hashicorp.com/terraform/intro, . [Online; accessed 23-November-2022].

[36] Jiangshui Hong, Thomas Dreibholz, Joseph Adam Schenkel, and Jiaxi Alessia Hu. An overview of multi-cloud computing. page 7, 2019.

[37] Sajid Iqubal. Google anthos with hybrid and multicloud. https://www.linkedin.com/pulse/google-anthos-hybrid-multicloud-sajid-iqubal?trk=pulse-article_more-articles_related-content-card. [Online; accessed 26-December-2022].

[38] Nana Janashia. Github actions tutorial - basic concepts and ci/cd pipeline with docker. https://www.youtube.com/watch?v=R8_veQiYBjI. [Online; accessed 2-March-2023].

[39] Vijay Kanade. Multi-cloud vs. hybrid cloud: 10 key comparisons. https://www.spiceworks.com/tech/cloud/articles/multi-cloud-vs-hybrid-cloud. [Online; accessed 08-November-2022].

[40] Kubernetes. Configure service accounts for pods. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account, . [Online; accessed 27-May-2023].

[41] Kubernetes. https://github.com/kubernetes/community/tree/master/icons/png, . [Online; accessed 30-July-2023].

[42] Kubernetes. Kubernetes. https://kubernetes.io, . [Online; accessed 22-August-2023].

[43] Kubernetes. Kubernetes. https://minikube.sigs.k8s.io/docs/start, . [Online; accessed 23-August-2023].

[44] Kubernetes. Kubernetes. https://minikube.sigs.k8s.io/docs, . [Online; accessed 23-August-2023].

[45] Kubernetes. Statefulsets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset, . [Online; accessed 27-May-2023].

[46] Bruce D Kyle. Understanding kubernetes workload node objects. https://azuredays.com/2020/12/09/understanding-kubernetes-workload-objects. [Online; accessed 21-April-2023].

[47] David S. Linthicum. Emerging hybrid cloud patterns. page 2, 2016.

[48] Mika Majakorpi. Theory and practice of rapid elasticity in cloud applications. page 18, 2013.

[49] Peter Mell and Timothy Grance. The nist definition of cloud computing. pages 2–3, 2011.

[50] Microsoft. What is a private cloud? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-private-cloud,. [Online; accessed 07-September-2022].

[51] Microsoft. What is a public cloud? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-public-cloud,. [Online; accessed 22-September-2022].

[52] Yifat Perry. Google anthos: The first true multi cloud platform? https://bluexp.netapp.com/blog/gcp-cvo-blg-google-anthos-the-first-true-multi-cloud-platform. [Online; accessed 30-November-2022].

[53] Dana Petcu. Multi-cloud: Expectations and current approaches. page 1, 2013.

[54] Puppet. Puppet. https://www.puppet.com. [Online; accessed 9-January-2023].

[55] RedHat. Iaas vs. paas vs. saas. https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas,. [Online; accessed 06-November-2022].

[56] RedHat. https://www.redhat.com/en/topics/automation/learning-ansible-tutorial,. [Online; accessed 20-August-2023].

[57] RedHat. Types of cloud computing. https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud#common-questions, . [Online; accessed 20-August-2022].

[58] RedHat. https://www.redhat.com/en/topics/devops/what-is-ci-cd?cicd=32h281b,. [Online; accessed 22-August-2023].

[59] Syed R Rizvi, Andrew Lubawy, John Rattz, Andrew Cherry, Brian Killough, and Sanjay Gowda. A novel architecture of jupyterhub on amazon elastic kubernetes service for open data cube sandbox. pages 3387–3390, 2020.

[60] SimpliLearn. https://www.simplilearn.com/tutorials/ansible-tutorial/what-is-ansible. [Online; accessed 21-August-2023].

[61] Paulo Simões. An introduction to cloud computing, 2022. Presentation for a subject lectured at Faculdade de Ciências e Tecnologia da Universidade de Coimbra entitled "Serviços e Infraestruturas de Alto Desempenho".

[62] Stackscale. Cloud computing service delivery models. https://www.stackscale.com/blog/cloud-service-models/#Cloud_computing_service_delivery_models, https://www.stackscale.com/wp-content/uploads/2020/04/cloud-service-models-iaas-paas-saas-stackscale.jpg. [Online; accessed 06-November-2022].

[63] Techopedia. On-demand self service. https://www.techopedia.com/definition/27915/on-demand-self-service. [Online; accessed 07-October-2022].

[64] Terraform. Terraform. https://www.terraform.io. [Online; accessed 8-January-2023].

[65] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. pages 1–24, 2020.

[66] Udesh Udayakumar. Introduction to gke on-prem. https://medium.com/google-cloud/introduction-to-gke-on-prem-78a42d630eb9. [Online; accessed 1-December-2022].

[67] Veritis. Google's anthos: The hybrid and multi-cloud platform that you need. https://www.veritis.com/blog/googles-anthos-the-hybrid-and-multi-cloud-platform-that-you-need. [Online; accessed 26-December-2022].

[68] VMware. Hybrid cloud vs. multi-cloud: What is the difference? https://www.vmware.com/topics/glossary/content/hybrid-cloud-vs-multi-cloud.html, . [Online; accessed 08-November-2022].

[69] VMware. https://www.vmware.com/topics/glossary/content/kubernetes-cluster.html, . [Online; accessed 23-August-2023].

[70] VMware. What is a public cloud? https://www.vmware.com/topics/glossary/content/public-cloud.html, . [Online; accessed 22-September-2022].

[71] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. The resource pooling principle. page 2, 2008.

[72] Yucong Duan and Guohua Fu and Nianjun Zhou and Xiaobing Sun and Nanjangud C. Narendra and Bo Hu. Everything as a service(xaas) on the cloud: Origins, current and future trends. page 3, 2015.

[73] Yucong Duan and Yuan Cao and Xiaobing Sun. Various "aas" of everything as a service. pages 1–3, 2015.