INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

Agnostic Cloud Services with Kubernetes

João Gonçalo Martins Bonacho

(Licenciado em Engenharia Informática e de Computadores)

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor Carlos Jorge de Sousa Gonçalves
Professor Doutor António Luís Freixo Guedes Osório

Júri:

Presidente: Professor Doutor José Manuel de Campos Lages Garcia Simão

Vogais: Professor Doutor Pedro Medeiros
Professor Doutor Carlos Jorge de Sousa Gonçalves

NOVEMBRO, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

Agnostic Cloud Services with Kubernetes

João Gonçalo Martins Bonacho

(Licenciado em Engenharia Informática e de Computadores)

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor Carlos Jorge de Sousa Gonçalves
Professor Doutor António Luís Freixo Guedes Osório

Júri:

Presidente: Professor Doutor José Manuel de Campos Lages Garcia Simão

Vogais: Professor Doutor Pedro Medeiros
Professor Doutor Carlos Jorge de Sousa Gonçalves

NOVEMBRO, 2023

"In the middle of every difficulty lies opportunity."
Albert Einstein

Acknowledgments

This thesis has only been accomplished with the support, help, patience, guidance, and presence of the people mentioned here. Without you, none of this would be possible.

First of all, I would like to express my deep gratitude to the supervisors of this thesis, Professors Carlos Gonçalves and Luís Osório. To Professor Carlos Gonçalves, for accompanying me during this year in valuable meetings, for listening to all my concerns and doubts, for giving valuable advice, and for proposing challenges that helped me achieve greater goals. To Professor Luís Osório, for proposing this work and for all the knowledge transmitted, always enthusiastically and with a strong vision of the future.

Thank you Luís, for the many afternoons and evenings we spent working on our theses. Your remarkable sense of friendship, companionship, and humor made my days much lighter and joyful. Thank you also for encouraging me to take on this challenge.

Thank you, José and Ana, for making these two years of Master's degree much more worthwhile. You are some of the best people I have met during my studies course. A special mention also goes to my colleagues Dinis, Bernardo, Pedro, and in particular, André and João, for all the moments we shared during our graduation journey.

My most special thanks goes to my family, especially my brother Guilherme and my mother, for all the support they gave me, for the daily sacrifices, and for always being there for me, in good times and especially in bad. I hope one day I will be able to repay everything they have done for me.

Finally, I would like to thank all the people who make Instituto Superior de Engenharia de Lisboa our second home, sometimes our first, that we all appreciate and respect. Thank you also to the teachers who have been part of my journey, all of whom have directly or indirectly contributed not only to this work, but to who I am today.

Thank you all. Thank you very much.

Abstract

The vendor lock-in concept represents a customer's dependency on a particular supplier or vendor, eventually becoming unable to easily migrate to a different provider. Cloud computing is frequently associated with vendor lock-in restrictions, motivated by the proprietary technological arrangements of each provider.

This work proposes an agnostic cloud provider model that addresses such challenges, focusing on the establishment of a model for deploying and managing computational services in cloud environments. Concretely, it aims to enable informatics systems to be executed agnostically on multiple cloud platforms and infrastructures, thereby decoupling them from any cloud provider. Moreover, this model intends to automate service deployment by defining and generating the running configurations for the services.

Within this context, container technology is deemed as an efficient and standard strategy for deploying computational services across cloud providers, promoting the migration of informatics systems between vendors. Additionally, container orchestration platforms, which are becoming increasingly adopted by organizations, are essential to effectively manage the life-cycle of multi-container informatics systems by monitoring their performance, and dynamically controlling their behavior. In particular, the Kubernetes platform, an emerging open standard for cloud services, is proving to be a valuable contribution on achieving service agnostic deployment, namely with its Cloud Controller Manager mechanism, helping abstracting specific cloud providers.

As validation for the proposed approach, it is intended to prove the model's adaptability to different services and technologies supplied by heterogeneous organizations through the deployment of containerized applications (informatics systems) in multiple cloud service providers, public or on-premises. For this purpose, the Informatics System of Systems framework is adopted as a validator for structuring and organize heterogeneous technology artifacts from different suppliers.

Keywords: Agnostic Cloud; Containers; Kubernetes; Cloud Computing; Software Deployment; Distributed Systems; Vendor Lock-In.

Resumo

A computação na nuvem é frequentemente associada a restrições de dependência de fornecedor (*Vendor Lock-In*), motivado pelas diferentes tecnologias e implementações proprietárias que cada fornecedor de serviços em nuvem estabelece. Estas restrições consistem na dependência de um cliente relativamente a determinado fornecedor, o que dificulta a transição para outro fornecedor.

Num contributo para uma Nuvem Agnóstica, o desafio descrito neste trabalho consiste na definição de um modelo de implantação e gestão do ciclo de vida de elementos computacionais em contexto de Nuvem. Por conseguinte, o objetivo do trabalho centra-se no desenvolvimento de um modelo que desacople a implantação e a gestão de sistemas informáticos do fornecedor de Nuvem, permitindo que sejam executados de forma agnóstica em diferentes plataformas de Nuvem. Neste âmbito, recorrer-se-á a contentores, enquanto solução eficiente e padronizada de implantação de serviços computacionais em diferentes infraestruturas. Adicionalmente, pretende-se que o modelo automatize a geração de ficheiros de implantação, definindo as condições de execução do(s) serviço(s).

Atualmente, as plataformas de orquestração de contentores são importantes aliados das organizações, sendo responsáveis pela gestão da implantação e configuração dos sistemas informáticos formados por múltiplos contentores. Existem diversas plataformas que surgem neste contexto, capazes de monitorizar o desempenho e controlar dinamicamente as configurações dos sistemas. Um exemplo paradigmático é a plataforma Kubernetes, que emerge como um standard aberto para serviços de Nuvem, cujo componente *Cloud Controller Manager* contribui para a abstração de fornecedores de Nuvem. Neste sentido, é considerada uma contribuição valiosa para atingir um modelo agnóstico de Nuvem.

O sistema desenvolvido é validado através da implantação de aplicações (sistemas

informáticos) contentorizadas, em múltiplos fornecedores de serviços em Nuvem, públicos ou *on-premises* (locais). Para este efeito, o quadro Informatics System of Systems é adotado, enquanto validador, como o modelo apropriado para estruturar e organizar os artefactos tecnológicos heterogéneos que podem ser considerados.

Palavras-chave: Nuvem Agnóstica; Contentores; Kubernetes; Computação na Nuvem; Implantação de *Software*; Sistemas Distribuídos; *Vendor Lock-In*.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AI Artificial Intelligence. 27
AKS Azure Kubernetes Service. 52
ANSR National Road Safety Authority. 70, 72
API Application Programming Interface. 14, 15, 16, 20, 22, 25, 32, 35, 36, 42, 44, 45, 46, 48, 49, 51, 54, 55, 64, 75, 92
AWS Amazon Web Services. 26, 27, 32

CA Certificate Authorities. 21
CCM Cloud Controller Manager. 4, 48, 91
CES Cooperation Enable Services. 29, 79
CI/CD Continuous Integration and Continuous Delivery. xv, 4, 5, 34, 35, 70, 71, 73, 78, 86
CLI Command-Line Interface. xviii, xix, 6, 15, 36, 44, 56, 58, 73, 74
CNCF Cloud-Native Computing Foundation. 16, 18
CPU Central Processing Unit. 13, 14, 19
CRI Container Runtime Interface. 22
CSP Cloud Service Provider. 10

DNS Domain Name System. 18, 22, 23, 24, 32, 83
DSL Domain-Specific Language. 64
DTO Data Transfer Object. 53

EKS Amazon Elastic Kubernetes Service. 52
ER Entity-Relationship. xix, 56, 57, x

FQDN Fully Qualified Domain Name. 66, 83

GCP Google Cloud Platform. 26, 27, 74, 75, 76
GKE Google Kubernetes Engine. 52
GUI Graphical User Interface. 36

HTTP Hypertext Transfer Protocol. 24, 25, 55

IaaS Infrastructure as a Service. 10, 26, 28
IaC Infrastructure as Code. 22, 32, 92
IDE Integrated Development Environment. 92
IoT Internet of Things. 27
IP Internet Protocol. 13, 18, 22, 57, 86
ISoS Informatics System of Systems. xviii, 1, 7, 28, 38, 44, 45, 73, 79, 80, 81, 84, 85
IT Information Technology. xviii, 3, 27, 34, 41, 44, 48, 49, 70, 73, 90

JAR Java ARchive. 52, 71, 85
JSON JavaScript Object Notation. xxi, 33, 59, 66

LXC Linux Container. 14, 15

MIB Master Information Base. 72

OCI Open Container Initiative. 4, 15, 16, 45
OS Operating System. 10, 11, 12

PaaS Platform as a Service. 10, 26, 28

QA Quality Assurance. 71

RAM Random-Access Memory. 19
REST Representational State Transfer. 15, 22, 25, 30, 55, 64

SaaS Software as a Service. 9, 26, 27, 28, 32

SDK Software Development Kit. 92

SDMAC Services Deployment and Management Agnostic-Cloud. xv, xviii, 2, 31, 36, 41, 48, 49, 51, 52, 53, 67, 86, 89, 90, 91

SIGET Traffic Events Management System. 72

SINCRO National Speed Enforcement Network. xv, xviii, 70, 72, 73, 79, 80, 81, 82, 83, 84, 85, 86, 87, 90

SNMP Simple Network Management Protocol. 72, 86

SOA Service-Oriented Architecture. 7, 27, 29

SoS System of Systems. 44

SPI Service Provider Interface. xxi, 62

SSH Secure Shell Protocol. 19

TCP Transmission Control Protocol. 24, x

TOSCA Topology and Orchestration Specification for Cloud Applications. 35

UI User Interface. xviii, 53, 59, 60, 81, 84, 85

UML Unified Modeling Language. xvi, xviii, 53, 54, 67, iii, iv

URL Uniform Resource Locator. xviii, 21, 46, 60, 75, 78, 81

VM Virtual Machine. xix, 3, 11, 12

YAML Yet Another Markup Language. 20, 34, 35

1

Introduction

Deploying informatics systems in multiple cloud providers can be very useful since organizations can leverage the unique capabilities and features of each provider, in order to optimize their operations and to benefit from the currently available offerings. In addition, by being able to distribute workloads (i.e., a unit of processing) across multiple providers, organizations can also improve the performance and reliability of their systems. Nevertheless, deploying informatics systems on different cloud platforms can be complex and demanding, as it can require significant technical expertise, and be very time-consuming and costly. The management and maintenance of services (i.e., a fundamental building block or component of an informatics system) deployed across multiple providers is also an arduous task, especially when it involves coordinating heterogeneous services and systems, and ensuring their seamless integration. The vendor lock-in concept, characterized in cloud computing by expensive and time-consuming migration of applications and data to alternative providers [99], is closely associated with these challenges and concerns.

In this field of research, the terminology surrounding terms such as "informatics system", "application", and "computational service" can lead to some misconceptions, given their apparent similarities in definition. Therefore, it is proposed that an "informatics system" corresponds to a more comprehensive and clearer term for an "application", encompassing a set of connected components. Additionally, based on a simplified version of the definitions established by the Informatics System of Systems (ISoS) framework [49], an informatics system comprises one or more "computational

services", which refers to an independent computing entity with a computational responsibility. By establishing these definitions, it is intended to provide a unified understanding of these terms, promoting improved communication and clarity of concepts throughout this document.

This work aims to mitigate vendor lock-in in cloud computing by decoupling informatics systems from any specific provider. On this basis, the research question focuses on establishing a vendor-agnostic cloud provider model, streamlining the deployment and management of computational elements (services), mainly in a cloud context. Therefore, it is proposed the development of a framework designated Services Deployment and Management Agnostic-Cloud (SDMAC). The approach to address this challenge considers the use of container technology, as an efficient strategy of deploying computational services to cloud providers, and container orchestration platforms, namely Kubernetes [53], for effective management of multi-container informatics systems in the cloud.

1.1 Context

Vendor lock-in phenomenon occurs when a customer becomes dependent on a specific supplier or vendor, making it challenging to transition to an alternative one. This situation can lead to a reliance on the products or services offered by a single supplier, which often promotes a lack of competition and potentially higher costs for the customer, as discussed in [85]. The same source claims that vendor lock-in is mainly caused by contracts or intellectual property rights with specific restrictions and commitments that create a sense of captivity for the customer with the supplier, and inhibit organizations' ability to explore other options or even negotiate more advantageous terms with their current provider.

Cloud computing is a paradigmatic example of vendor lock-in restrictions, which results from the proprietary technologies employed by each cloud provider. Accordingly, [69] describes the vendor lock-in problem in cloud computing as the difficulty and complexity that customers face when attempting to switch from one cloud provider to another without facing technological barriers, and possibly significant costs and legal limitations. This scenario becomes more evident when a developer or an organization is heavily reliant on a particular supplier and its technological strategies and implementations, thus becoming unable to straightforwardly migrate to a different vendor.

The proposed framework consists on the development and implementation of a model

for deploying and managing computational services in cloud environments. This approach aims to follow an agnostic cloud provider model, in order to reduce vendor lock-in restrictions, predominantly in cloud computing. The goal of this model relates with the Agnostic Cloud concept, that refers to the design and conception of informatics systems that can be executed on any cloud platform, whether public, private, or hybrid, and regardless of the specific provider, without requiring any changes. Containers are a valuable contribution to address this task, as they provide an efficient and standardized method for deploying service elements across multiple cloud providers.

The framework's model is evaluated through the deployment of containerized applications on multiple Information Technology (IT) infrastructures, either public cloud providers or on-premises platforms. It is intended to prove that the model is able to flexibly accommodate different technologies and services, enabling deployments across a variety of cloud environments in distinct scenarios, reducing any type of vendor lock-in. Furthermore, to ensure the proposed model's compatibility and reliability, a combination of multiple services, containers, and cloud providers is utilized.

1.2 Approach Overview

The research question refers to creating a cloud provider model with agnostic characteristics, allowing the deployment and management of informatics systems on any cloud platform, without being tied to a particular provider. By being platform-agnostic, the expected model strives to be capable of supporting diverse technologies and specifications, so that it can incorporate multiple cloud providers, whether public or on-premises platforms. Ultimately, this type of strategy allows organizations to effortlessly switch between different technologies and vendors as needed, avoiding vendor lock-in restrictions. Figure 1.1 illustrates a global view for the deployment of services according to the proposed model, highlighting the gray box with straight corners in the figure, which represents the focus of this study and development.

The container concept offers a standard and portable method for packaging and deploying applications along with their dependencies, facilitating the migration of applications between different cloud providers. Considering the scope of this work the proposed model uses containers instead of Virtual Machines (VMs) as containers offer faster deployment and startup, cost effectiveness, and ease of migration, as disclosed in [22]. Furthermore, [23] asserts that containers can help managing and replicating services across multiple cloud providers, as they provision a consistent and isolated

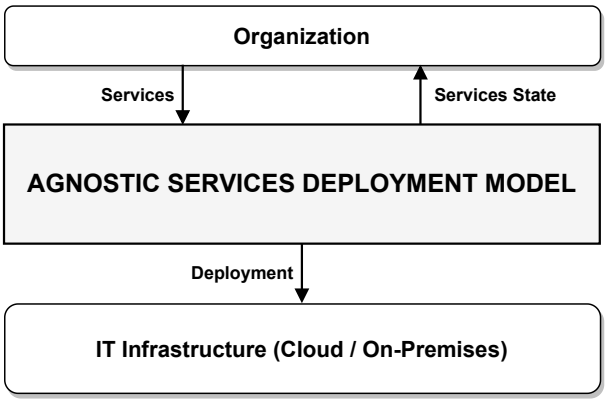


Figure 1.1: Adopted strategy overview

environment for each service. Finally, containers are able to be executed on any platform that supports the container technology, namely through standard container runtimes and Open Container Initiative (OCI) images, as later discussed in Section 2.2.5, which can be very helpful in overcoming vendor lock-in constraints.

Containerization can also be extremely helpful in Continuous Integration and Continuous Delivery (CI/CD) [18] processes by ensuring consistent execution environments, as explored in [17]. Container technology enables applications to be executed under the same software versions and environment requirements, such as external dependencies or libraries, regardless of the deployment target, which supports and facilitates their migration from development environments to production environments.

Among container orchestrators' landscape, the Kubernetes platform (K8s) is identified as the ideal platform for the purpose of this work, as it has become the *de facto* standard for deploying and operating containerized applications (informatics systems) [27], particularly because of its the control plane. Furthermore, Kubernetes's Cloud Controller Manager (CCM), a component of the control plane, enables the abstraction of cloud providers, allowing for integration across different cloud platforms. This mechanism also enables location transparency, meaning that multiple Service elements of a single informatics system can be deployed across different cloud infrastructures while operating seamlessly. Ultimately, these technologies can significantly simplify the migration of applications between different cloud providers, by providing standard execution environments and interfaces for programmatic interaction, thereby helping organizations to overcome cloud deployment efforts, and mitigating the constraints imposed by vendor lock-in.

1.3 Work Objectives

The work's objective is to develop a vendor-agnostic provider model, thereby unifying the deployment and management of computational elements (i.e., services) in the Cloud. Specifically, it is intended to develop an infrastructure element that enables informatics systems to be deployed, executed and monitored on multiple cloud provider platforms without being tied to a particular provider. The automation of deployment scripts for Kubernetes, which are reportedly complex and time-consuming for organizations, is also considered pertinent in this work, serving as a cue to its simplification.

The developed framework helps organizations overcome the challenges associated with cloud deployment, allowing software development teams to focus more on developing their core services and products. Ultimately, the framework empowers organizations and developers to effortlessly switch between different technologies and vendors as needed, thereby contributing to the reduction of vendor lock-in restrictions in organizations. To achieve these goals, the following requirements are considered:

- Develop an interface to receive services from organizations' development environments to be deployed and managed on cloud platforms;
- Build a software module responsible for defining the execution configurations for the services to be deployed, as well as creating the required container images;
- Deploy and dynamically manage services agnostically on a Kubernetes cluster, ensuring that cluster's components are properly created and maintained according to services' requirements;
- Enable effortless and transparent vendor switching for organizations, facilitating the migration of services across different cloud providers;
- Dynamic services updates triggered by CI/CD pipelines in GitLab repositories;
- Test and validate the framework's model using multiple services, containers, and cloud providers to ensure compatibility and reliability.

1.4 Contributions

This work contributes to the field of multi-cloud service deployment and management through the following outcomes:

- A dissertation that thoroughly explores the deployment and management of services across multiple cloud providers, covering theoretical and practical foundations, while addressing vendor lock-in challenges;
- A functional prototype platform with a Service Management API for back-end service exposure;
- The project includes a web interface and a Command-Line Interface (CLI) to support and simplify service configuration and management;
- All artifacts' source code is made publicly available at [81], promoting further development and collaboration;
- A peer-reviewed publication in Springer LNCS paper format, published and presented at INForum 2023 (Portuguese National Conference on Informatics), summarizing main research findings and results.

1.5 Document Organization

This document is divided into six chapters. Apart from this introductory chapter, the rest of the document is structured as follows. Chapter 2 summarizes related research work and industry contributions in the area under investigation, where the most relevant content found and existing work are presented. It also provides an overview of similar approaches to the proposed work, highlighting the main differences and added value of the presented solution. Chapter 3 presents the proposed architecture for the framework, describing its components and their responsibilities within the system. In Chapter 4, key development aspects and technologies employed in system's reference implementation are described. Chapter 5 aims to demonstrate how the developed platform's prototype was tested and validated. Finally, Chapter 6 focuses on the conclusions, which include a summary of the work accomplished and a discussion of future work.

2

Background and Related Work

This chapter aims to expose the background and related work and organizes as follow: Section 2.1 introduces the Cloud Computing paradigm, defining it and outlining its essential characteristics. Section 2.2 exposes the main concepts of the container technology and its evolution and standardization. Section 2.3 discusses containers' orchestration, focusing on Kubernetes and other platforms, e.g., Docker Swarm and Marathon. Section 2.4 depicts an overview of major cloud providers and how to deploy services on cloud environments. Section 2.5 provides a description of the Informatics System of Systems (ISoS) framework, as an approach to structure technology artifacts, namely in a Service-Oriented Architecture (SOA), where promoting independence from any specific technological solution is of paramount importance. Finally, Section 2.6 provides a description of similar solutions to the proposed work.

2.1 Cloud Computing

Cloud Computing represents a paradigm shift in computing landscape that changed the way data is stored and accessed, and applications are deployed and executed. It is being embraced by multiple organizations, from different industries and with distinct sizes, as a cost-effective and efficient strategy compared to traditional on-premises computing. Notwithstanding, it currently faces significant challenges, including concerns about data privacy, security, vendor lock-in, and technical limitations (e.g., latency, bandwidth constraints, or even debugging difficulties). As stated in Section 3.1,

the objective of this work is to solve some of these challenges, particularly in vendor lock-in problem. But first, it is of vital importance to provide an overall understanding of fundamental cloud computing characteristics. Therefore, this section aims to provide an overview of cloud computing, focusing on the key concepts and terminologies of service and deployment models.

2.1.1 Defining Cloud Computing

In 2011, NIST (US National Institute of Standards and Technology) defined **Cloud Computing** as model that enables convenient and effortless access to a shared pool of configurable computing resources, including networks, servers, storage, applications, and services, through an on-demand network. These resources can be provisioned and made available with minimal service provider interaction and management endeavor [66].

This governmental agency also presented and clarified the essential characteristics of a Cloud model [66]:

1. **On-demand self-service**: consumers can access resources from the pool without requiring a human administrator involvement;
2. **Broad network access**: all resources can be obtained through a network without the need for direct physical access;
3. **Resource pooling**: the service provider abstracts and collects resources into a pool, and distributes portions of it to different consumers;
4. **Elasticity**: consumers are able to expand or contract the resources they use from the pool (by provisioning or deprovisioning), frequently done automatically. This feature helps consumers aligning resource usage with demand faster and more accurately;
5. **Measured service**: the provision of resources is metered to guarantee that consumers only utilize what they are allotted, i.e., clients only pay for what they consume. This is usually done transparently for both the provider and consumer of the utilized service.

2.1.2 Deployment Models

A **Cloud Deployment Model** is the type of architecture in which a cloud system is deployed, differing in terms of administration, ownership, access control, and security

protocols. The four main types of deployment models are Private Cloud, Public Cloud, Hybrid Cloud, and Community Cloud. These models are now described, according to [74].

A **Private Cloud** is owned by an organization and managed centrally. It can be hosted by a third party (e.g., public cloud provider) or be kept on-premises within the organization, if organizations prefer to manage it from their local data center.

In a **Public Cloud** the service provider owns and operates all the infrastructure to run a cloud, keeping the hardware in large data centers. It is widely used for development and testing purposes due to typically being inexpensive, and easily configured and deployed. Hence, it is a common option for web applications, file sharing, and non-sensitive data storage.

A **Hybrid Cloud** combines public and private clouds, allowing data and applications to move and interact seamlessly between them. Typically, organizations start with a private cloud and then expands it in order to include public cloud services. This type of model is very useful for companies with regulations that require data protection and storage or sensitive data, not appropriate to be stored on public cloud infrastructure.

The **Community Cloud** shares cloud service for organizations with common "interests, governance, security requirements, and policies". This cloud can be hosted on the organization's premises, a peer organization's premises, at a single provider, or a combination of these. It is noteworthy that the term "community cloud" is used for marketing purposes, considering that the technology behind can be private, or hybrid cloud.

2.1.3 Service Models

Nowadays, data centers are designed to make available the three most common Service Models: Software as a Service, Platform as a Service, and Infrastructure as a Service. These services are offered as utilities, allowing consumers to only pay for what they use on a pay-per-use model. In [28] authors describe these service models as follows:

- **Software as a Service** (**SaaS**): a model that allows multiple users to access applications, e.g., a web-mail server through a web browser client application. It is becoming popular due to its accessibility and the rise of service-oriented architectures and web services. It is typically accessed over the Internet, and purchased on a "pay-as-you-go" subscription model.

- **Platform as a Service** (**PaaS**): a service model that provides a platform for customers to build and deploy their applications using provider's infrastructure, libraries and tools. This "middleware solution" simplifies the deployment process and reduces costs by do not requiring to manage hardware and software. Therefore, it constitutes a cost-effective and streamlined approach to deploy applications.

- **Infrastructure as a Service** (**IaaS**): a model that supplies customers computing resources such as processing, storage, and other necessary components to run an application. In this scenario customers have full control over the operating systems, applications, and networking components, but do not manage the underlying cloud computing infrastructure, as it is cloud provider's responsibility.

Figure 2.1 depicts the differences between the existing service models through a Shared Responsibility Model. This model highlights the security and compliance responsibilities of both Cloud Service Providers (CSPs) and customers, regarding all aspects of the cloud environment, including hardware, infrastructure, endpoints, data, configurations, settings, Operating System (OS), network controls, and access rights.

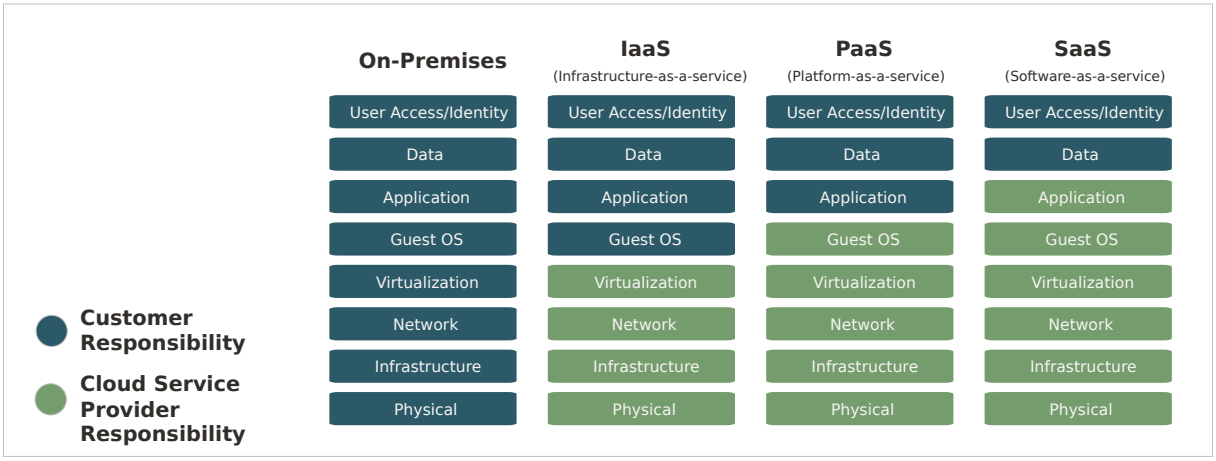


Figure 2.1: Cloud Shared Responsibility Model, extracted from [15]

In subsequent discussions of orchestration tools and their alignment with cloud environments, this shared responsibility model will serve as an important framework for assessing the security and compliance implications inherent in the various orchestration solutions.

2.2 Containers

This section introduces the main concepts of container technology and how it evolved over the last decades. It also differentiates containers from virtual machines, depicting the benefits on using containers in the intended model. Containers' standardization is also discussed, as an emerging topic.

2.2.1 Virtualization

IBM defines **Virtualization** [45] as the process that allows a more efficient usage of a computer's hardware. This is accomplished by using a hypervisor, responsible for creating an abstraction layer between the physical resources and the virtual environment, dividing the hardware components into multiple virtual machines, on top of an operating system, or directly onto the hardware. Each virtual machine is able to run its own OS behaving independently, as a computer consuming the resources that are allocated by the hypervisor.

The evolution of computing paradigms is illustrated in Figure 2.2.

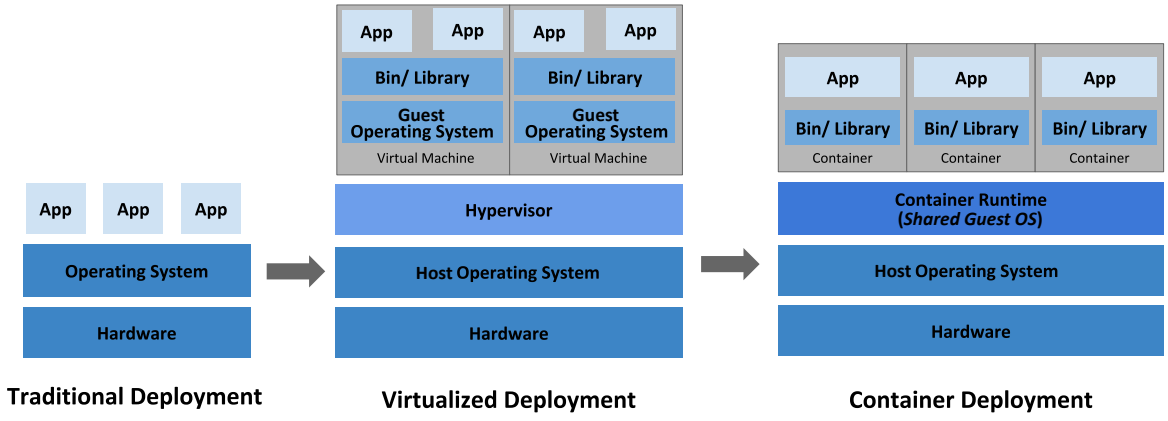


Figure 2.2: Evolution of computing paradigms, adapted from [53]

In a **Traditional Deployment**, organizations had to run applications directly on the OS of physical servers, which was a problem concerning resource allocation. **Virtualized Deployment** was introduced as a solution that allowed to run multiple virtual machines on a single physical server, providing better resource management and scalability. Finally, containers emerged as a technology that resembles VMs, but offers enhanced portability across multiple clouds and operating systems, due to its lightweight nature, making **Container Deployment** the preferred method in current landscape [53].

2.2.2 Containers vs. Virtual Machines

Regarding virtualization technologies, both **Virtual Machines** and **Containers** must be considered due to their encapsulation and isolation capabilities. However, these two technologies are significantly different.

The fundamental difference between a Virtual Machine (VM) and a Container is that when using a VM to execute an application, it requires the installation of an entire operating system, whereas with a container, which consists of the application itself and its dependencies, that is not necessary. This difference is also represented in Figure 2.2. Moreover, containers are executed in the host operating system and are isolated from other processes. Hence, a container has much smaller size than a virtual machine. Additionally, booting a container does not require the entire operating system to be initiated, making it highly efficient concerning deployment and startup.

A significant disadvantage of containers concerns security vulnerabilities, as [100] asserts that "containers may be less secure than VMs since the underlying OS is shared". I.e., if an attacker compromises a container, he can potentially gain access to the OS where other containers are running and compromise them as well.

Table 2.1 illustrates the differences outlined between Containers and Virtual Machines.

Table 2.1: Differences between VMs and containers, adapted from [19]

Containers	Virtual Machines
Lightweight	Heavyweight
Native performance	Limited performance
All containers share the host OS	Each VM runs in its own OS
OS virtualization	Hardware-level virtualization
Startup time in milliseconds	Startup time in minutes
Requires less memory space	Allocates required memory
Process-level isolation, less secure	Fully isolated, more secure

The proposed model uses containers instead of virtual machines as the benefits of containers outweigh those of virtual machines in the context of this work, primarily due to faster deployment and startup, cost effectiveness, and ease of migration.

2.2.3 Main Concepts

A primary objective of **Containers** is to provide isolation to multiple processes running on the same host. Based on [1], a container can be defined as an isolated group of

processes that are restricted to a private root filesystem and process namespace. Those processes share the kernel and other services of the host operating system, but with some restrictions, e.g., by default they cannot access files or system resources outside their container.

The authors of [1] also assert that containers are a "fusion of numerous existing kernel features, filesystem tricks, and networking hacks" and that **container engine** is the management software that facilitates the cooperation of all elements. Another fundamental concept is the **container image**, that packages an application and its dependencies into a standard and portable file, allowing any host with a compatible container runtime engine to create a container using that image as a template.

The more significant UNIX and Linux features that containers rely on can be briefly described as follows [1]:

1. **Kernel**: specific kernel features enable processes isolation, such as:
   (a) `Namespaces`: isolated processes for filesystem mounts, process management, and networking resources;
   (b) `Control groups`: managing Central Processing Unit (CPU) and memory resources between isolated containers;
   (c) `Capabilities`: mechanisms for access control of processes to kernel operations and system calls;
   (d) `Secure computing mode`: similar to `Capabilities`, but with a more precise and detailed control.
2. **File System**: container images use union filesystem, i.e., a filesystem that combines multiple filesystems into one, to improve performance and portability. Moreover, the usage of this filesystem allows the creation of a single and unified filesystem from multiple sources. The directory structure and file locations in a container image are similar to those found in a standard Linux distribution.
3. **Networking**: by default, containers are connected to the network through a network namespace and a bridge within the host, which gives them a private Internet Protocol (IP) address. Then, it is the administrator's responsibility to make the container's ports accessible from the outside world. However, containers can be configured in "host mode networking", which means that the all of its ports are exposed directly to the network and have full access to host's network stack.

By leveraging these characteristics, containers improve and simplify the process of developing, testing, and deploying informatics systems by distributing responsibilities among system components. Consequently, they provide isolated and transparent computational resources, and automated deployment operations.

2.2.4 Container Evolution

The origin of operating system virtualization dates back to the creation of an UNIX command called `chroot` in 1979, which allowed to change the root directory of a process and its children to a new location in the filesystem, providing isolation of file system resources for each process. According to [43], the main contributions to the development and evolution of containers as we see them today can be described as follows.

In 2000, **FreeBSD Jails** was released, adding isolation of user and network resources to the `chroot` command.

Based on the idea of FreeBSD Jails, in 2001 **Linux VServer** was created, isolating resources such as file system, CPU usage, network address and memory on an operating system.

In 2004, Sun released **Solaris Containers** as a feature in Solaris 10, containing system resource control and binary isolation provided by Zones, i.e., a fully isolated virtual server within the operating system instance.

Similar to Solaris Containers, in 2005, SWsoft issued **OpenVZ**, that provided virtualization, isolation, resource management and checkpoints through patched Linux cores.

On a project developed by Google in 2006, **Process Containers** was released, which could record and isolate each process's resources such as CPU, memory, hard disk I/O and network. In 2007, it was renamed to **Control Groups (Cgroups)**.

In 2008, **Linux Container (LXC)** was first available based on Cgroups and Linux Namespaces implementations added to the kernel. This became known as the first more complete container technology.

Another big milestone took place in 2011, when CloudFoundry created **Warden** that was able to operate on any operating system, run as a daemon and provide an Application Programming Interface (API) to manage containers.

In 2013, Google created an open source container technology stack called **lmctfy**. Later, in 2015, donated the core technology of lmctfy to **libcontainer**.

In 2013, **Docker** [30] was created, initially an internal project of DotCloud. At first Docker used LXC but later replaced them with **libcontainer**. The main different between Docker and the others is that Docker builds a complete ecosystem around containers, including container imaging standards, container Registry, RESTful APIs, CLI, a container cluster management tool called Docker Swarm, etc..

In 2014, CoreOS released **rkt** to improve a container engine for Docker's security defect rewrites, including service discovery tools **etcd** and Web tool **Frankel**.

Finally, in 2016 Microsoft released **Hyper-V Container**, a Windows-based container technology that works like container technology under Linux to ensure that processes running in a container are isolated from the outside world.

2.2.5 Container Standardization

Nowadays, standardization in container technology is of fundamental importance to promote interoperability between different types of container technologies, establishing standards on their core components. This section outlines the main projects created in this context.

Prior to container platforms like Docker, deploying applications in containers was a complex and time-consuming process that required further knowledge and experience from the developer. However, Docker has revolutionized this process. Docker containers offer a easier way of building and managing applications compared to previous methods, which provides smoother assembly and maintenance duties for developers [12]. Also according to [12], Docker became mainstream because it simplified the "transfer of an application's code and dependencies from a developer's laptop to a server", which made "Docker quickly became close to a *de facto* industry standard for containers".

The same authors also exposed a growing topic concerning the management and coordination of large groups of containers, that proved to be challenging and could demand new platforms and tools. This subject will be discussed in Section 2.3.

2.2.5.1 Open Container Initiative

The **Open Container Initiative (OCI)** [5] is a Linux Foundation project started in 2015, whose main compromise is to establish open standards for virtualization at the operating system level. The OCI defines specifications for containers to ensure compatibility and interoperability across different container platforms. It defines three specifications:

the Image Specification (`image-spec`), the Runtime Specification (`runtime-spec`), and the Distribution Specification (`distribution-spec`). The **Image Specification** is intended to enable the creation of interoperable tools for building, transporting and preparing a container image to run. Therefore, it defines an OCI Image, composed by image manifest, an image index (optionally), a group of filesystem layers, and a configuration [48]. The **Runtime Specification**'s objective is to define the configuration, execution environment, and lifecycle of a container. The execution environment is specified to ensure that the applications operating inside the container maintain a uniform environment across runtimes [82]. Lastly, the **Distribution Specification** establishes an API protocol that simplify and standardizes content distribution [29].

It is of paramount importance to demystify the relationship between Docker and OCI, as it has lead to some misconceptions over the last years. As mentioned above, OCI enables compatibility and interoperability between different container platforms, establishing a set of specifications for containerization, including runtime and image format, while Docker is a platform that provides a complete solution for building, deploying, and managing containers. Docker contributors claim that "standards are important, but they are far from a complete production platform" [36]. Due to the inherent incompleteness of standards, vendors commonly introduce novel proprietary features to give their products a competitive edge. This is where Docker, along with alternatives such as the open-source Podman [76], comes into operation as they strive to differentiate their products. According to [36], Docker platform uses OCI specifications, but also offers other features such as development, distribution, security, and orchestration. Same source asserts Docker has contributed significantly to the OCI, by donating code that formed the basis of the OCI runtime and image format specifications.

2.2.5.2 Cloud-Native Computing Foundation

The **Cloud-Native Computing Foundation (CNCF)**, also established in 2015 in the context of a Linux Foundation project, is depicted in [101] as a foundation that provides a neutral hub for cloud native computing collaboration, to connect developers, end users and vendors, including public cloud providers. In addition, it hosts cloud native projects, such as Kubernetes and Prometheus.

2.3 Container Orchestration

The widespread adoption of the container concept for instantiating services is evident, as it offers a powerful method for isolating and running computer systems (applications). As the number of containers in computer systems increases, it is essential to coordinate them effectively in order to scale with minimal risk of failure. Therefore, tools for composing and orchestrating containers are becoming crucial for organizations, as they provide the mechanisms and automation demanded for this arduous task.

2.3.1 Motivation and Definition

Orchestration refers to the processes and actions held by cloud providers or application owners in order to manage the configuration, deployment, and dynamic maintenance of resources, aiming to ensure a certain level of quality of service, performed either manually or automatically, as described in [14]. Furthermore, orchestration intends to manage the resources required to support an application or service in a efficient and effective manner.

The same paper ([14]) describes **Container Orchestration** as a concept that empowers cloud and application providers to manage the deployment and configuration of multi-container computer systems in a cloud context, including selecting and deploying the containers, monitoring their performance, and dynamically managing their configuration. Typically, this involves adopting tools or platforms available for container orchestration, e.g., Kubernetes, Docker Swarm, Marathon, and Red Hat OpenShift. These platforms, beyond from providing the basic features for container orchestration, also intend to provide other services, such as discovery, load balancing, and other health monitoring utilities.

Organizations are widely adopting these platforms as they are becoming increasingly valuable to their businesses in order to increase the reliability and performance of their informatics systems, in a heterogeneous environment.

2.3.2 Orchestration Tools

This section presents and describes the main orchestration tools, as the most popular and representative market players according to [60], with a particular focus on Kubernetes concepts.

2.3.2.1 Kubernetes

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem, and Kubernetes services, support, and tools are widely available [11].

Kubernetes was initiated by Google in 2014 as an open-source project. The name *Kubernetes* originates from Greek, and denotes "helmsman" or "pilot". It is often abbreviated as **K8s**, which represents the eight letters located between the "K" and "s". Kubernetes "combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community" [57].

The lineage of Kubernetes can be directly traced back to Borg, an internal project used by Google to run applications for many years. Kubernetes has improved upon its predecessor by using the best ideas from Borg and addressing user pain points over the years [47]. In 2015, the project was donated to CNCF, as the first project to be contributed to this foundation [47]. The Kubernetes project is now being developed under open specifications by the Kubernetes community, hosted by CNCF.

As depicted in Section 2.2.3, containers offer a convenient way to package and run applications. However, in a production environment, it is imperative to manage the containers to avoid downtime, which typically involves ensuring that failed containers are replaced by new ones to maintain availability. To simplify this process, Kubernetes offers a framework for running distributed systems in a resilient manner. Kubernetes official documentation outlines the following topics as platform's main key characteristics when addressing this challenge [57].

- **Service discovery**: a container can be exposed by Kubernetes via its Domain Name System (DNS) name or IP address.
- **Load balancing**: the platform offers load balancing and network traffic distribution when containers are experiencing heavy traffic, to ensure that the deployment remains stable.
- **Storage orchestration**: with Kubernetes, it is possible to automatically mount a storage system, including local storages, or public cloud providers, to store data that the container generates or needs to access.
- **Automated rollouts and rollbacks**: Kubernetes allows developers to define the desired state of the deployed containers, and modify them to match the specified state, in a controlled manner.

- **Automatic bin packing**: the amount of CPU and Random-Access Memory (RAM) usage that each container needs can be configured. Thereafter, Kubernetes is capable of placing the containers onto nodes and optimize the available resources.
- **Self-healing**: K8s is designed to automatically perform several actions in response to container failures, including restarting or replacing them, besides from terminating containers based on user-defined health check failures. Additionally, Kubernetes ensures that a container is fully operational before advertising it to clients.
- **Secret and configuration management**: Kubernetes allows users to store, administer, deploy and update secrets (e.g., passwords, OAuth tokens, and Secure Shell Protocol (SSH) keys) and application configuration without exposing them in configuration stack, and without having to rebuild the container images.

Deploying Kubernetes results in the creation of a cluster [54]. A cluster refers to a Kubernetes network mounted on the control plane, operating with zero or more worker nodes. A **worker node** hosts Pods (i.e., the application workload) and the **control plane** is responsible for managing the worker nodes and the Pods within the cluster. The cluster's architecture is represent in Figure 2.3.

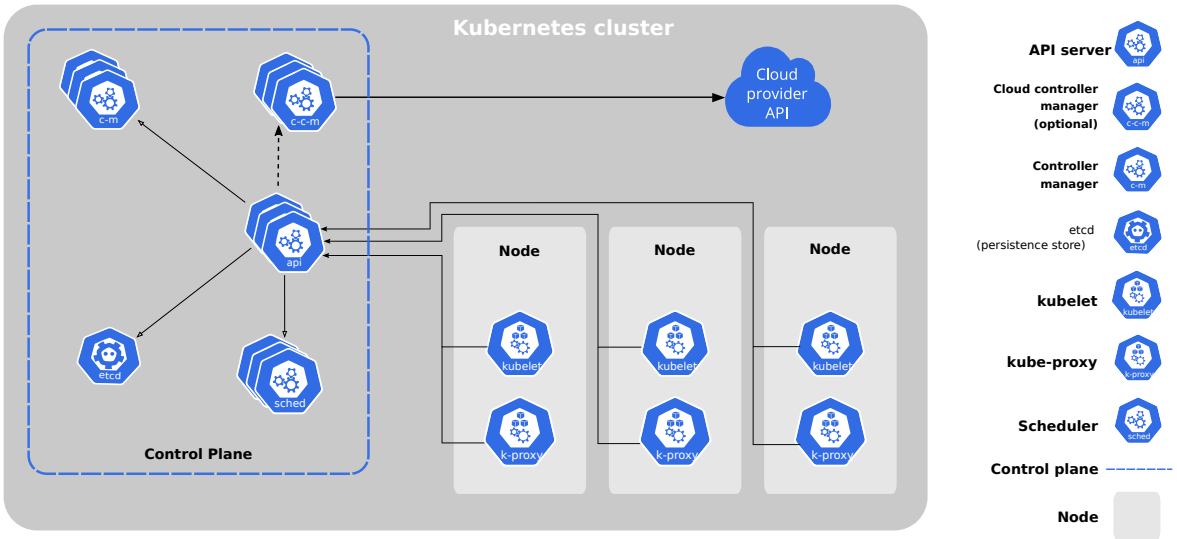


Figure 2.3: Kubernetes cluster, extracted from [53]

The **Control Plane** components "make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied)" [11]. These components include a **Kubernetes API Server**, a **Scheduler**, a **Controller Manager**, an

etcd and optionally a **Cloud Controller Manager**. According to [54], the API Server (`kube-apiserver`) is the component with which the user or developer and other control plane components communicate; the Scheduler (`kube-scheduler`) is responsible for assigning a worker node to each deployable component in order to schedule the applications; the Controller Manager (`kube-controller-manager`) handles cluster-level tasks such as component replication, worker node monitoring and node failure management; the `etcd` represents a "reliable" data store that persistently stores the configuration of the cluster; and, finally, the `cloud-controller-manager` is a component of the Kubernetes control plane that links the cluster to a cloud provider's API, separating out components that interact with the cloud platform from those that only interact with the cluster.

Node components, `kubelet` and `kube-proxy`, are responsible for managing the execution of pods and creating the runtime environment for Kubernetes on each individual node [47], respectively. The `Kubelet` manages worker nodes in Kubernetes. It is responsible for registering the node, starting and monitoring pod containers, and reporting their status and resource usage to the API server. It also handles liveness probes and container termination. The `kube-proxy` process on every node is responsible for managing all aspects of Services in Kubernetes.

In order to create a fully functional Kubernetes cluster, it is essential to have a comprehensive understanding of other key concepts, including **Objects**, **Pods**, **Services**, and **Volumes**.

A Kubernetes **Object** corresponds to a persistent entity in the Kubernetes system, that indicates which "containerized" applications are running on each node, which resources are available for them, and which policies define the way these applications act (either restart, upgrade or fault tolerance policies). They are declared in a Yet Another Markup Language (YAML) file (`.yaml`) and managed through Kubernetes API. The status of an object describes the current state of the object and the Kubernetes control plane is what allows to continuously and actively manage the state of each object so that it always corresponds to the desired state. Furthermore, they can be created, edited or removed via the Kubernetes API, either through the `kubectl` command line interface or using client libraries such as Quarkus (in Java). The **Quarkus framework**, an open-source project, provides the feature of automatically generating Kubernetes resources based on user-supplied configuration, being able to deploy an application to specific Kubernetes cluster by "applying the generated manifests to the target cluster's API Server" [79]. In addition, Quarkus is capable of creating a container image and pushing it to a registry.

In this context, emerges a configuration file, commonly named **kubeconfig**, used to configure access to Kubernetes clusters. This file contains information about clusters, users, namespaces, authentication mechanisms, and other settings. It is used by tools like `kubectl` that interacts with Kubernetes clusters, enabling them to select the appropriate cluster and corresponding API Server [52]. Kubeconfig files are fundamentally composed by the following sections:

- **Clusters** - section that establishes the Kubernetes clusters possible to connect to, incorporating information about cluster's name, server Uniform Resource Locator (URL) (i.e., API Server endpoint) and Certificate Authorities (CA).
- **Users** - defines users and client authentication credentials (e.g., client certificates, client keys, and tokens).
- **Contexts** - a mechanism that associates a cluster, a user, and a namespace. They are used to specify which cluster and user is required to be used in a particular interaction.
- **Current Context** - a section used that defines a Context that is used by default when using the specified configuration file.

A **Pod** is the smallest deployable object that can be created and managed in Kubernetes. It is a group of one or more containers that share storage, network resources, and a description of how to run the containers. A pod models an application-specific "logical host" that contains one or more application containers that are tightly coupled. They can be of two types: running in a single container (the most common), or running in multiple containers. A representation of the Pod concept is shown in Figure 2.4.

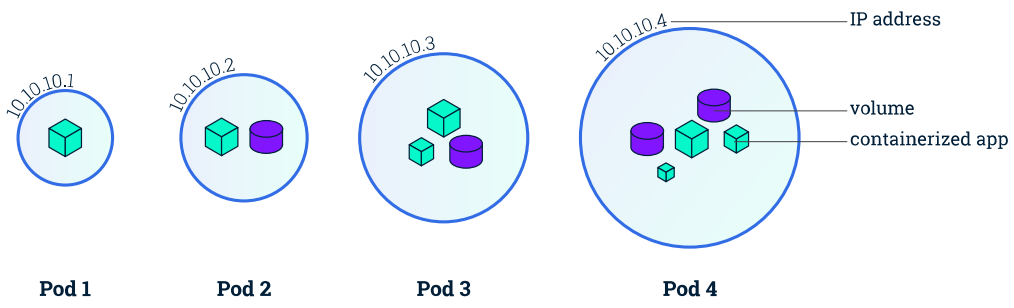


Figure 2.4: Kubernetes pods overview, extracted from [53]

In Kubernetes, a **Service** is defined as an infrastructure abstraction consisting of a logical set of Pods and a policy for accessing them. Specifically, it is a REST object that is managed by a Kubernetes Server API. It is also worth noting that Kubernetes provide each Pod with their own IP addresses and a (unique) DNS name for a set of pods, for load balancing purposes.

Finally, a **Volume** is basically a component required for file sharing between multiple containers.

Each node in a Kubernetes cluster requires a container runtime for `kubelet` to launch pods and their containers. For this purpose and as a complement to Section 2.2.5, another effort in container standardization is the **Container Runtime Interface** (CRI), which provides a communication protocol between `kubelet` and the container runtime, allowing `kubelet` to be compatible with different container runtimes and without having to recompile the cluster components to keep its operation [55]. Moreover, the CRI only provides support for container runtimes that are compliant with the Open Container Initiative, as disclosed in [24].

Ultimately, it is of imperative nature to depict the "*declarativeness*" of Kubernetes. As discussed earlier and as the authors of [11] claim "Everything in Kubernetes is a declarative configuration object that represents the desired state of the system", which brings the concept of Infrastructure as Code (IaC), in conjunction with declarative vs. imperative approaches. IBM defines IaC as "using code to manage and provision infrastructure (networks, virtual machines, load-balancers, clusters, services and connection topology) in a descriptive model instead of manual processes." [46]. Accordingly, Kubernetes "describes the state of the world, declarative configuration does not have to be executed to be understood. Its impact is concretely declared." [11], which makes this process less error-prone and with the possibility of having a version control system over the state of the system.

2.3.2.2 Docker Swarm

Docker's cluster management and orchestration capabilities embedded in the Docker Engine are built using `swarmkit`, an independent project that implements an orchestration layer, natively integrated with Docker platform. A swarm is a group of Docker hosts running in swarm *mode* that can act as managers, workers, or both. A **manager** is responsible for managing membership and delegation, whereas a **worker** runs swarm services. **Swarm service** stacks can be defined and executed via Docker Compose, also used to specify and execute containers. When creating a service, it is possible to

define its desired state, e.g., number of replicas, network and storage resources, and exposed ports. The platform is responsible for maintaining the desired state, namely by scheduling tasks on other nodes if a worker becomes unavailable. Unlike standalone containers, Swarm services can change their configuration (e.g., update the networks and volumes they use) without requiring to restart the services. Docker will update the configuration by stopping "obsolete" service tasks and creating new ones with updated settings, per [32].

Docker's documentation emphasizes the importance of the concepts of Nodes, Services, Tasks, and Load Balancing when working with Docker swarm services. These concepts can be summarized as:

- **Nodes** [34]: a **node** is an instance of the Docker engine that participates in the swarm, and that can be distributed across multiple machines. To deploy an application to a swarm, the service definition is sent to a manager node, which dispatches **tasks** to worker nodes. The **manager nodes** handle orchestration and cluster management to maintain the swarm's desired state, while worker nodes execute tasks. Furthermore, a **worker node** runs an agent that reports on the assigned tasks and notifies the manager node of its current state.

- **Services** [35]: a **service** defines the tasks to be executed on manager or worker nodes. To create a service, a container image must be specified, as well as the commands to execute it.

- **Tasks** [35]: the swarm manager distributes replica tasks among nodes based on the desired scale, known as **replicated services** model. While **global services** run one task for the service on each available node.

- **Load Balancing** [33]: in a swarm, the swarm manager exposes services to outside the swarm by using **ingress load balancing**. For that purpose, the port (`PublishedPort`) can be chosen by the user or use a predefined one. External components can access the service on any node's port, and all swarm nodes redirect the incoming connections to the specified service. Moreover, the swarm has a DNS feature to help distribute requests among services, in a process called **internal load balancing**.

Figure 2.5 presents an example of these concepts in practice.

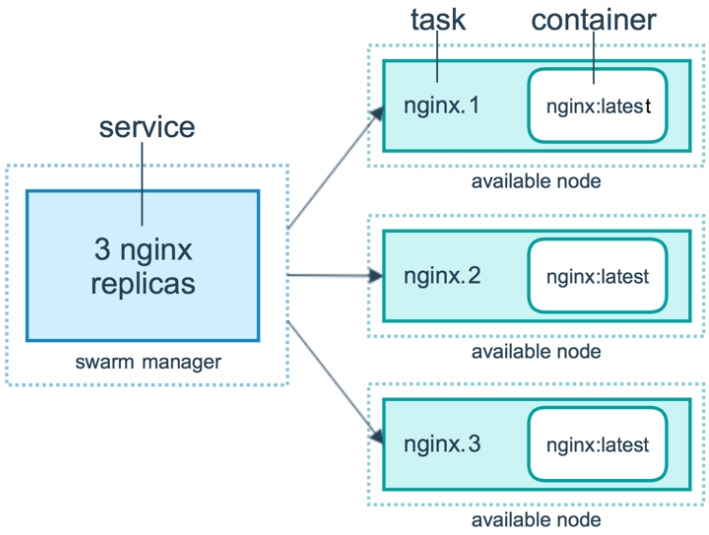


Figure 2.5: Docker Swarm services diagram, extracted from [31]

2.3.2.3 Marathon

Marathon is a "production-grade container orchestration platform" for Mesosphere Datacenter Operating System (DC/OS) [25] and Apache Mesos [7], according to [62].

Based on [64], these are the key features of Marathon container orchestration platform:

- **High Availability**: multiple running Marathon instances are pointed to the same ZooKeeper quorum for leader election, enabling continuous application operation and performance even if an instance fails.
- **Multiple container runtimes**: Marathon offers "first-class support" for Mesos containers (via `cgroups`) and Docker.
- **Stateful apps**: to create a stateful application, a local persistent volume can be specified to prevent data loss when applicational tasks are restarted.
- **Constraints**: constraints help optimize where an application runs for fault tolerance or locality purposes. In fault tolerance it spreads a task out on multiple nodes, whereas for locality it runs all of an applications tasks on the same node.
- **Service Discovery** and **Load Balancing**: traffic can be routed to an application from internal and external applications of the cluster, which can be done through `Mesos-DNS`, for DNS-based service discovery, or port-based service discovery with HAProxy.
- **Health Checks** - the application's health can be measured via Hypertext Transfer Protocol (HTTP) or Transmission Control Protocol (TCP) checks.

- **Event Subscription** - it is possible to define an HTTP endpoint to receive notifications upon application events.
- **REST API**: it offers a REST API for integration and scriptability.

Marathon's documentation exemplifies Marathon orchestration platform in action for various applications and services in Apache Mesos [63]. According to the example of Figure 2.6, Marathon is launched alongside Mesos and uses a scheduler to manage processes. It can run other Mesos frameworks like **Chronos** and can manage other application containers such as JBoss servers, Jetty, Sinatra, and Rails. Ultimately, it shows that Marathon helps maintain 100% uptime for other frameworks and applications, working seamlessly with them to manage workloads in Mesos.

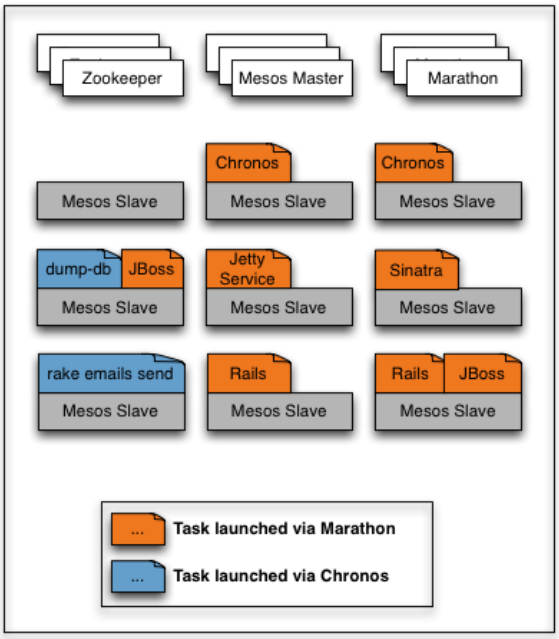


Figure 2.6: Example of Marathon architecture, extracted from [63]

2.4 Deployment on Cloud Providers

This section defines and describes the process of deploying services in cloud providers. Typically this includes choosing the cloud provider, selecting the desired deployment model and service model, configuring the environment, deploying the application, and, finally, testing and monitoring its operation and performance. The current section starts by exposing the major cloud providers in the market, depicting some of their more relevant characteristics.

2.4.1 Cloud Providers

This section presents and describes the cloud providers with largest market share in 2020, according to a study conducted by ACM (Authority for Consumers and Markets). For simplicity, we will only focus on the European market.

2.4.1.1 Market Share

The Authority for Consumers and Markets (ACM) is a dutch regulatory authority responsible for ensuring competition and protecting the consumer against abuses of market power in multiple sectors. According to this organization, cloud market shares are difficult to obtain due to different definitions of cloud layers used by the providers. Only market shares on IaaS layer are public and reliable. Data on PaaS and SaaS has to be calculated based on total cloud market. Therefore, the sources of data were generally market operators and public sources [65].

The results produced by this research are presented in Figure 2.7. This chart shows that Amazon Web Services (AWS) and Microsoft Azure both have market shares of between 35% and 40%, followed by Google Cloud Platform (GCP) and Oracle Cloud that hold market shares ranging from 5-10%. IBM Cloud trail behind with a share between 0% and 5%. The others operators (e.g. OVHcloud) have a market share of a few percent.

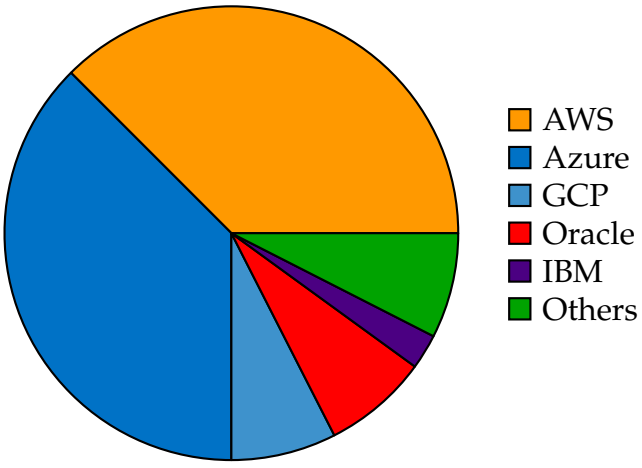


Figure 2.7: Cloud providers European market share in 2020, adapted from [65]

The same source asserts that the described concentration in cloud services market appears to be on the rise, rather than experiencing a decline.

2.4.1.2 Major Operators

According to Section 2.4.1.1, the cloud providers with largest market share are the hyperscalers Amazon Web Services, Microsoft Azure and Google Cloud Platform. This section will briefly describe them.

Amazon Web Services (AWS) is a cloud services provider founded in 2006 by the Amazon group. It was created to overcome the problem of underutilized IT capacity during non-peak seasons at Amazon's e-commerce business, and nowadays is the most profitable sector of Amazon, generating a profit margin of nearly 30%. AWS offers a wide variety of services, including computing, storage, databases, analytics, machine learning, and a "marketplace" for third-party cloud services [65].

Microsoft Azure started providing cloud services in 2010, by the name of Windows Azure. Microsoft Azure initially provided application development services and later expanded their business model to infrastructure services and MS Office software products. Currently, it offers over 130 services across 21 categories such as computing, containers, databases, Internet of Things (IoT), mixed reality, storage, etc., operating globally with its own data centers. Financially, their cloud services generate high profit margins, accounting for over 40% of Microsoft's total revenue [65].

Google started offering public cloud services under the name **Google Cloud Platform** (GCP) in 2011, with an App Engine for businesses to easily develop application. Later, expanded its services to include storage and SaaS in Google Workspace. GCP operates across all three layers of the cloud market and provides over 100 services in 18 categories, with data centers in more than 200 countries. The services supplied include Compute, Storage, Artificial Intelligence (AI), IoT, among many others. Google Cloud (GCP and Google Workspace) increased by 47% in revenue in 2021, mainly due to GCP's infrastructure and platform services. Nevertheless, Google Cloud is still operating at a loss, which fell to $3.099 billion in 2021, a 45% decrease from the previous year [65].

2.4.2 Deploying Service Elements

Deploying services (from the SOA terminology) in cloud providers involves migrating a service or an application from a local environment to a remote cloud, where the infrastructure required to execute the computer system is handled by the cloud provider.

After configuring a user account in the cloud provider, introducing personal and payment information, and defining required settings, the next stage comprises selecting

the service model. The cloud user can choose from Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), or Software-as-a-Service (SaaS), whose main characteristics are described in Section 2.1.2.

The subsequent stage considers the deployment of the application or service, which is accomplished through code uploading, establishing the necessary resources, and adjusting the environment. The cloud provider plays a crucial role by offering tools that streamline the deployment process.

Once the service is successfully deployed and running, it is essential to test and continuously monitor its performance to guarantee its proper functioning and optimize its efficiency. Towards that end, tools developed by the vendor can be used to monitor the performance and health of the workload.

In conclusion, deploying services in cloud providers is a straightforward process that involves choosing the cloud provider, selecting a service, setting up the environment, and deploying the workload. Nevertheless, it is noteworthy that while the basic and general steps remain consistent with the described process, the specific procedure will vary depending on the cloud provider and the required service. Additionally, the majority of the tools and mechanisms involved in the deployment process are proprietary, which may require specialized technical knowledge and are prove to be a time-consuming endeavor for the cloud user. This presents a significant challenge for organizations, as vendor lock-in can become an onerous obstacle when considering the need to migrate services or workloads to a different cloud provider, potentially increasing the cost and complexity of switching providers.

2.5 Informatics System of Systems Framework

This section briefly describes the Informatics System of Systems (ISoS) framework [49]. First, the ISoS framework motivation and definition will be introduced. Then, it will be presented an overview of the ISoS Service concept and their instances accessibility. Finally, several benefits of ISoS architecture will be depicted.

2.5.1 Motivation

The development of this framework was driven by the need for a systematic and efficient method for managing and organizing the technological artifacts provided by

different suppliers. This framework aims to provide a standardized and unified approach for organizing and structuring the heterogeneous artifacts, thereby ensuring consistency and reliability of computer systems.

2.5.2 Definition

The **Informatics System of Systems (ISoS)** framework proposes a non-intrusive integration model to establish a multi-supplier technology landscape to reduce vendor lock-in risks, according to [40]. This technology and modeling structuring approach promotes independence from any specific technological solution, creating competitiveness in the technology landscape of organizations, as depicted in [39].

The ISoS framework is based on three core modeling elements: `ISystem`, `CES`, and `Service`. As described in [40], an **ISystem** establishes a "coarse" computational and cooperation responsibility border, and comprises one or more **CES** (**Cooperation Enable Services**), whereas a CES consists of one or more **Services**. The Service concept refers to an independent and (possibly) autonomous computing entity, representing some computational responsibility. This model architecture is illustrated in Figure 2.8. Furthermore, ISoS elements model the technology artifacts through a set of properties, such as name, version, supplier or description.

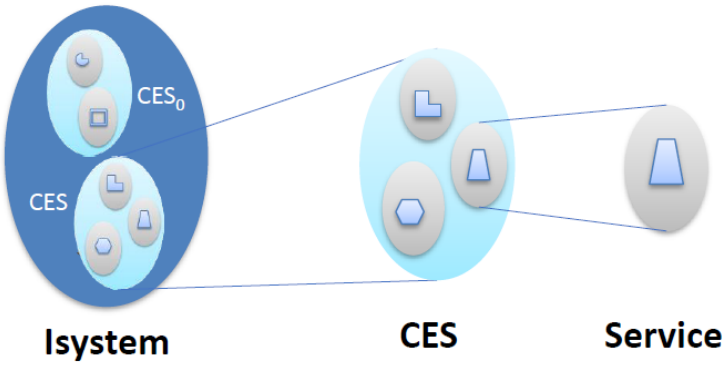


Figure 2.8: Informatics System model, extracted from [49]

2.5.3 The Service-Oriented Concept

The **ISoS Service** concept is based on the traditional SOA pattern and Microservice terminology, capable of abstracting independent computing entities. In [72], authors differentiate ISoS Service from microservice since ISoS Service does not imply any size

or complexity restriction nor imposes an interaction protocol. Additionally, the microservice concept often tends to be associated with the cloud, and the Service instance can be running anywhere from on-premises to a cloud provider.

2.5.4 ISoS Service Instance Accessibility

To be ISoS-enabled, an organization needs to instantiate the Meta-ISystem, which is an instance of the $ISystem_0$, i.e., an ISystem with the role of managing and coordinate the ISoS landscape. An illustration of this ISystem is represented in Figure 2.9.

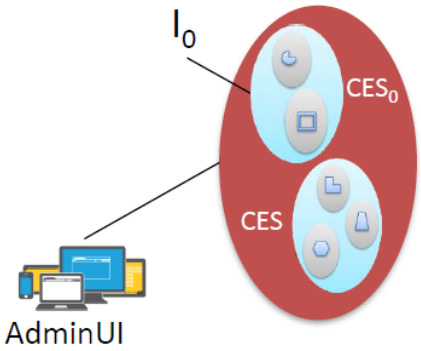


Figure 2.9: The Meta-ISystem, $ISystem_0$, extracted from [49]

According to [71] and [39], with ISoS framework, any allowed Service instance is accessible from both inside and outside organization, through the interface I_0 of $System_0$. $ISystem_0$ works as a Service registry, making clients lookups possible. This ISystem has a REST interface accessible at `isos.<organization domain>:2058` endpoint, so that any authorized peer computing service can access the I_0 REST interface and any implemented service, through its `ISystem/CES/Service` path.

2.5.5 Benefits of ISoS Architecture

According to the ISoS architecture, any integrated system becomes interoperable, allowing to cooperate with other systems following the same architecture, under pre-established conditions and preferably based on open standards (idea disclosed in [73]). Furthermore, it is safe to say that the technology behind the data and its associated models are irrelevant.

In [72], an ISoS reference implementation was presented allowing to establishing collaboration among two different organizations. Each organization with their well identified responsibilities, and more importantly without knowing the internal details of

each other. This demonstration can be considered highly valuable since it can be extended to more complex scenarios involving several organizations, where the only requirement is that the involved organizations can access the ISoS landscape of each other.

2.5.6 The Usage of ZooKeeper System

As discussed in Section 2.5.4, ISoS Service instances aim to be consistent and reliable, specially ISystem_0's due to its criticality, as the other ISystems depend on its availability. To overcome this challenge, [39] explores the usage of the open-source ZooKeeper system in the ISoS framework. The open-source ZooKeeper [107] can be configured in redundancy mode, known as *quorum mode*, so that it maintains a consistent replica in N independent servers, i.e., the *ensemble*. Still according to this paper, beyond the fault-tolerance and distributed coordination strategies, the ISystem_0 implementation is prepared to scale multiple Service instances through the Observer nodes concept, in order to speed up read-only service lookup operations.

2.6 State of the Art

Related work in this research area revolves around platforms that provide infrastructure automation and management capabilities, primarily for the deployment of applications and services in cloud environments. This section introduces and briefly describes existing similar solutions, highlighting their main functionalities and objectives, in Section 2.6.1, and presents an overview of related research in terms of scientific and academic contributions, in Section 2.6.2.

The aim of this study is to conduct a comparative analysis of these platforms and works, in order to verify the added value of the proposed approach, and to learn certain features and details for the design and implementation of the SDMAC framework.

2.6.1 Similar Platforms

This section provides an overview of technologies and platforms related to infrastructure automation and management. Within this scope, several technologies and platforms can be identified and categorized as follows: Infrastructure as Code [44] tools, including platforms as Terraform [91] and CloudFormation [20], which streamline the deployment and management of applications; Multi-cloud orchestration frameworks,

including Cloudify [21] and Heat [41], that provide resource and application orchestrating capabilities across multiple cloud platforms; and Cloud Computing platforms, namely OpenStack [70] and Anthos [6], providing diverse services for efficient application deployment and management in the cloud.

Tersely, **AWS CloudFormation** is an Infrastructure as Code service, owned by Amazon, that enables customers to model, provision, and manage AWS and third-party resources. **OpenStack Heat** is a service that enables infrastructure and application lifecycle management in OpenStack clouds, accessible by both humans and machines.

In order to conduct a more nuanced and thorough analysis of these platforms, this section provides detailed descriptions of Terraform (Section 2.6.1.1), Cloudify (Section 2.6.1.2), and Anthos (Section 2.6.1.3). These platforms have been selected for this purpose due to their more significant similarities of concepts and objectives with the proposed work.

2.6.1.1 HashiCorp Terraform

Terraform by HashiCorp is "an infrastructure as code tool that lets you build, change, and version cloud and on-prem resources safely and efficiently", per [103].

More specifically, **HashiCorp Terraform** is an Infrastructure as Code tool that enables developers to define and manage cloud and on-premises resources using configuration files. These configuration files are human-readable and can be reused and shared across the community. It also provides a consistent workflow for provisioning and managing the infrastructure at multiple stages of its life cycle and can manage low-level components (e.g., compute, storage, and networking resources) and high-level components (e.g., DNS entries or SaaS features) [42].

This tool supports any cloud platform or service that exposes an API to work with, i.e., the Providers, which are "a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources." [92]. The idea behind Terraform is depicted in Figure 2.10.

Summarily, the core Terraform workflow comprises three stages: Write, Plan and Apply [42], as illustrated in Figure 2.11.

According to [42], these stages can be described as follows [90]:

- In **Write** stage, a developer can define an infrastructure (e.g, resources and services) through configuration files, using the IaC concept.

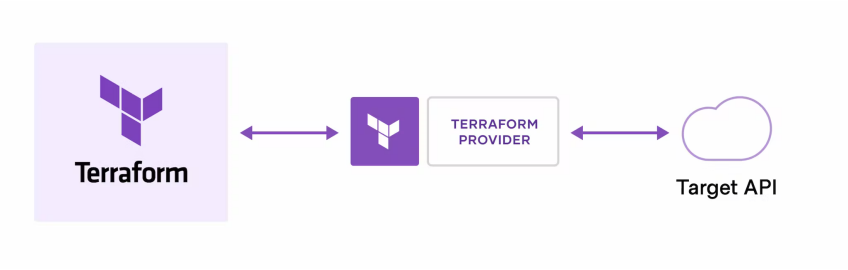


Figure 2.10: Terraform overview, extracted from [42]

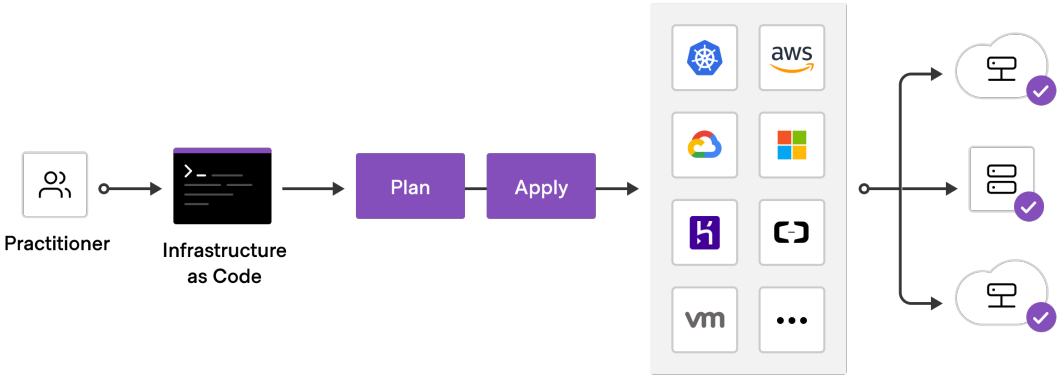


Figure 2.11: Terraform deployment workflow, extracted from [89]

- In the **Plan** stage, the platform creates an execution plan, to be applied to the infrastructure (existing or not) according to the configurations established in the previous phase.
- The **Apply** stage is where Terraform provisions the infrastructure, taking into account the operations, the order of the operations, and the corresponding resource dependencies, and updates the state file (i.e., a JavaScript Object Notation (JSON) file that tracks the current state of the infrastructure resources and helps Terraform to manage the infrastructure [90]).

According to HashiCorp founders, Terraform "solves infrastructure challenges", specifically due to the following characteristics. Terraform adopts an **immutable infrastructure approach** that simplifies the process of modifying or upgrading services or the infrastructure [61]. This platform relies on a **State File** to identify the changes required to align the infrastructure with the given configuration. The state files also act as a "source of truth" to keep **track of infrastructure state** [97]. Moreover, the above mentioned configuration files are declarative, which means they describe **the desired state of the infrastructure**. Besides that, to achieve an **efficient resource management**, Terraform creates and modifies non-dependent resources in parallel, based on a dependency resource graph [8]. In addition, Terraform embraces **standardize configurations**

through modular and reusable components called **Modules** to define an infrastructure [88], while collaborating with the community.

2.6.1.2 Cloudify

Cloudify is an "open source intent based orchestration service that helps DevOps teams turn their existing application and infrastructure resources (Terraform, Kubernetes, Ansible, Docker, Scripts, etc.) into self service development and production environments on any cloud." [102]. Figure 2.12 represents Cloudify's main building blocks.

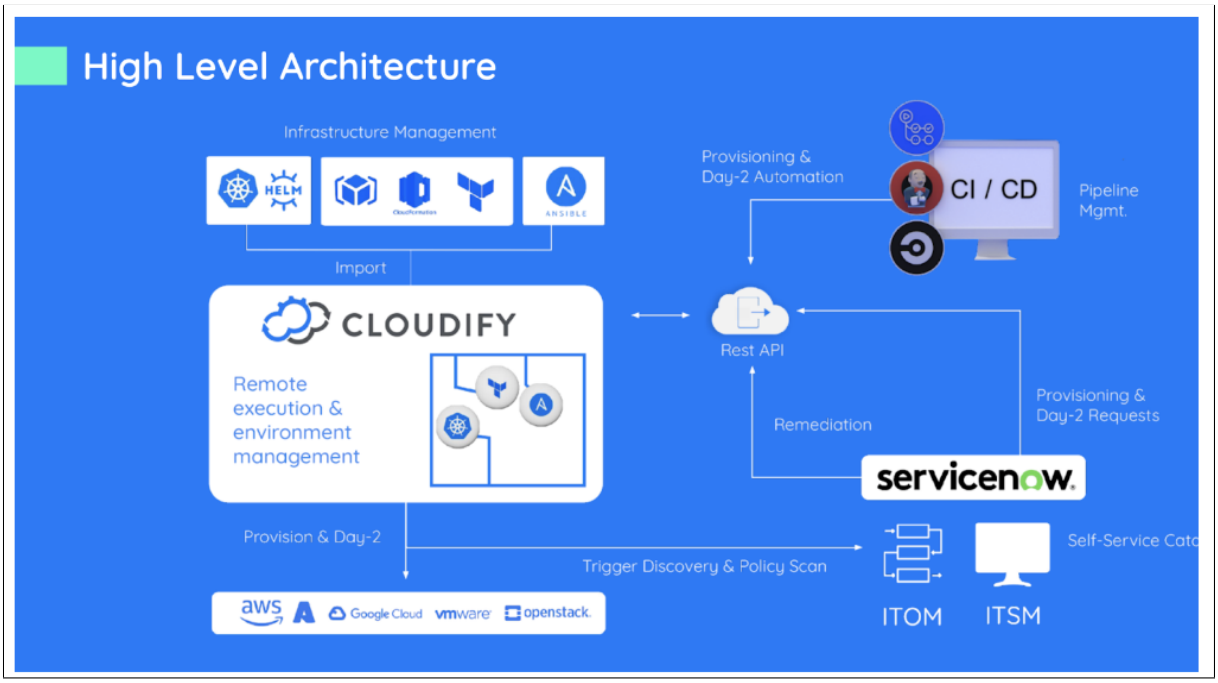


Figure 2.12: Cloudify high-level architecture, extracted from [84]

As illustrated in Figure 2.12, Cloudify offers services for remote execution and environment management. The Cloudify's benefit that stands out is its ability to seamlessly integrate with existing DevOps toolchains and IT environments by tightly integrating with CI/CD tools and ServiceNow [83] (a cloud-based platform for managing IT services, operations, and businesses for enterprises). In addition, this tool allows users to automate their existing development and production environments in a easier manner, namely through a "rich set" of built-in plugins and "out of the box" environments.

In order to better understand the purpose and functionality of the Cloudify platform, there are a number of key concepts to refer to, including [9]:

- **Blueprint** - a YAML file based on the Topology and Orchestration Specification

for Cloud Applications (TOSCA) specification [96], used to create an automation plan.

- **Environment** - a "special instance of a deployment" used as a container for other deployments, storing configuration information that is common to all the contained deployments.
- **Input/Output** - a configuration created by the user, and supplied to a deployment.
- **Deployment** - a blueprint instance generated according to input configuration.
- **Nested Deployments** (`ServiceComponent`) - a composition of different deployments managed under a single blueprint.
- **Workflow** - a group of tasks created based on the state information from a deployment.
- **Plugin** - a resource library that links a resource API with the corresponding YAML representation, in the context of a blueprint.
- **Secret** - a storage system that maintains its data in an encrypted format using key/value pairs.

2.6.1.3 Google Cloud Anthos

Google Cloud Anthos [6] is a Kubernetes-focused container platform launched by Google in 2019, that provides enterprise-grade management and orchestration for consistent application deployment across multiple infrastructures.

Key features of the platform include a managed service mesh with monitoring and troubleshooting capabilities, policy enforcement and security measures automation, and enhanced security for hybrid and multi-cloud environments.

Anthos excels for allowing organizations to modernize their applications by ensuring Kubernetes governance and streamlining CI/CD processes on diverse cloud environments where Kubernetes can run, including Google Cloud, on-premises infrastructures, or other public clouds.

2.6.1.4 Comparison

Table 2.2 provides a summary comparison between the described similar solutions as multi-cloud orchestrators, including their integration capabilities with cloud providers,

open source availability, supported artifacts, and available interfaces (command line interface, API, and Graphical User Interface (GUI) support). This table also includes the developed framework in this work, the SDMAC framework.

Table 2.2: Multi-Cloud orchestrators comparison, adapted from [26]

Orchestrators	**Integrable with**	**Open-Source**	**Artifact**	**CLI**	**API**	**GUI**
Cloud Formation	AWS	No	YAML, JSON	No	Yes	Yes
Heat	AWS, OpenStack	Yes	HOT, YAML	Yes	Yes	No
Terraform	AWS, GCP, Azure, OpenStack, Oracle, vSphere, and more than 30	Yes	HCL, JSON	Yes	online functions	No
Cloudify	AWS, GCP, Azure, OpenStack, vCloud, vSphere	Yes	TOSCA, YAML	Yes	Yes	Yes
SDMAC	Azure, GCP, AWS	Yes	N/A	Yes	Yes	Yes

The solutions analyzed are all multifaceted platforms capable of handling multiple features related to infrastructure automation and management, mainly in cloud environments. However, organizations still need to rely on these platforms specifications and tools to deploy their applications, which can difficult the migration from one platform to another. For example, if an organization has deployed its services in the cloud using a platform like Terraform, it may not be a straightforward process to switch to Cloudify, and vice-versa.

2.6.2 Related Research

Related research examination is of paramount importance when developing novel solutions for such a rapidly evolving area as the deployment and management of computational services in cloud environments. Therefore, the following scientific and academic articles provide a portrait of some related literature, key advances, existing gaps, and technological comparative analysis on this research topic.

The paper presented in [3] provides a similar approach to address this issue, discussing a cloud-agnostic container orchestrator that aims to improve interoperability by enabling live migration of containers between different cloud platforms. To this end, it

introduces Adapters, a centralized orchestrator, and an interface for managing container deployments and associated migrations. This system demonstrates promising potential to efficiently address cloud interoperability challenges, representing a novel approach for the future of automation and autonomous container management.

The paper available at [4] compares and evaluates a set of container orchestration tools, including Kubernetes, Docker Swarm, Mesos, and OpenShift. By providing an overview of each tool's architecture and features evaluating factors such as security, deployment, stability, scalability, cluster installation, and learning curve, it aims to assist researchers in selecting the most suitable tool for their needs. The research conducted by the authors revealed the following key points: Kubernetes is recognized for its advanced scheduling capabilities and efficient resource management; Docker Swarm stands out for its ease of use and user-friendly interface; Mesos offers optimal scalability and resource management; OpenShift is known for its robust security capacities.

In another strand, the author of [59] discusses the concept of cloud-agnostic workloads and how Kubernetes platform strives to enable them. Although Kubernetes offers standardized deployments and management, the variations and proprietary features of each cloud provider pose challenges to achieving true cloud agnosticism. As a result, it is suggested that organizations should adopt a multi-cloud strategy and focus on the development of advanced management tools.

The article in [58] discusses whether to build or to buy Kubernetes platform for deploying organizations' services, it and suggests to buy managed Kubernetes services. The main reason behind this decision is related to choosing a provider that is not tied to a specific cloud platform, thus providing a portable solution that aligns with cloud-native principles. In this sense, the risk of being locked into a single vendor's ecosystem (vendor lock-in) is reduced, and organizations can still benefit from community-driven software.

Finally, papers in [13] and [2] reveal the lack of effective solutions concerning this endeavor. In [13] the author outlines challenges in autonomic container orchestration, specifically the need for improved mechanisms that incorporate "runtime self-adaptation". In particular, it describes the lack of container orchestration frameworks with effective features of monitoring, profiling, resource management, and optimization capabilities. Complementarily, [2] exposes that although cloud container technologies are becoming a fundamental strategy for organizations, further efforts are required to face existing challenges, namely in enhancing orchestration support, and validating this technology through practical use cases. It also highlights the necessity

for advancements in resource monitoring, effective failure management, and seamless integration with continuous development techniques.

2.7 Summary

This chapter presented a comprehensive overview of the state of the art in cloud computing, containers, container orchestration, and application deployment on cloud platforms. It started by describing the cloud computing paradigm, its basic principles and the challenges it faces today. It then revealed how containers have emerged as a lightweight and portable alternative to traditional virtual machines, providing more flexibility and scalability for services deployment, and how container orchestration tools, such as Kubernetes and Docker Swarm, have become essential for managing containerized applications at scale. An overview of the challenges of deploying and managing containers in the cloud, especially in terms of vendor-specific configurations, was also discussed. Furthermore, similar solutions and research to the proposed work were described, focused on their automation and management capabilities for cloud infrastructure, simplifying the deployment process, and reducing the risk of configuration errors. Finally, the ISoS framework was discussed to demonstrate that it offers a valuable contribution to the testing and validation of the proposed model to prove the model's adaptability to different services and technologies supplied by heterogeneous organizations.

Beyond the reasons for using containers and the need for container orchestration, preferably via Kubernetes, and simultaneously understanding cloud computing and its deployment paradigm, the fundamental result from this chapter is that all similar solutions have some variances and partially different purposes from the proposed framework. By adopting the aforementioned platforms, organizations continue to rely on proprietary services and specifications, as each platform requires the writing of configuration files with specific syntax and commands. Ultimately, this dependency can make it difficult to migrate applications between platforms. This work is designed to address this difficulty by establishing an open standard for an agnostic cloud that offers a straightforward method for deploying services, through containers on diverse IT infrastructures, decoupling organizations from any vendor-specific configuration or implementation.

This chapter also provided an overview of the rapidly evolving field of cloud services. It explored multiple scholarly works covering advances, gaps, and technological assessments. The discussion delved into the concept of cloud-agnostic workloads,

emphasizing their importance and challenges, further unveiling a cloud-agnostic container orchestrator. Additionally, it revealed the strategic value of managed Kubernetes services to mitigate vendor lock-in. As a result of this research, the ongoing challenges in container orchestration and the need for advancements in cloud container technologies were identified. These findings recognize the critical role of research and development in cloud service deployment and management panorama.

3

System Architecture

This chapter presents the proposed architecture for the SDMAC framework. For this purpose, a set of diagrams are presented to visually represent the model, focusing on its key components, the responsibilities of the components within the system, and how they interact and communicate with each other. Section 3.1 provides an overview of the model, while Section 3.2 explores model's architecture, examining its specific components and interactions in greater detail. At last, Section 3.3 details and illustrates the interaction between the model and a Kubernetes cluster and its Cloud Controller Manager.

3.1 Model Overview

As discussed in Section 1.2, the SDMAC framework aims to establish an interface between organization's IT applications and IT infrastructures (on-premises or cloud-based platforms), to support the deployment of service elements agnostically across different cloud platforms. The simplified proposal of model's architecture for this solution is illustrated in Figure 3.1.

The model proposal comprises four main components: **Model Interface**, **Services Configuration**, **Agnostic Cloud Interface**, and **Kubernetes Manager**. From a general perspective, the **Model Interface** component is responsible for interacting and communicating with organizations that need to deploy and manage their services on a

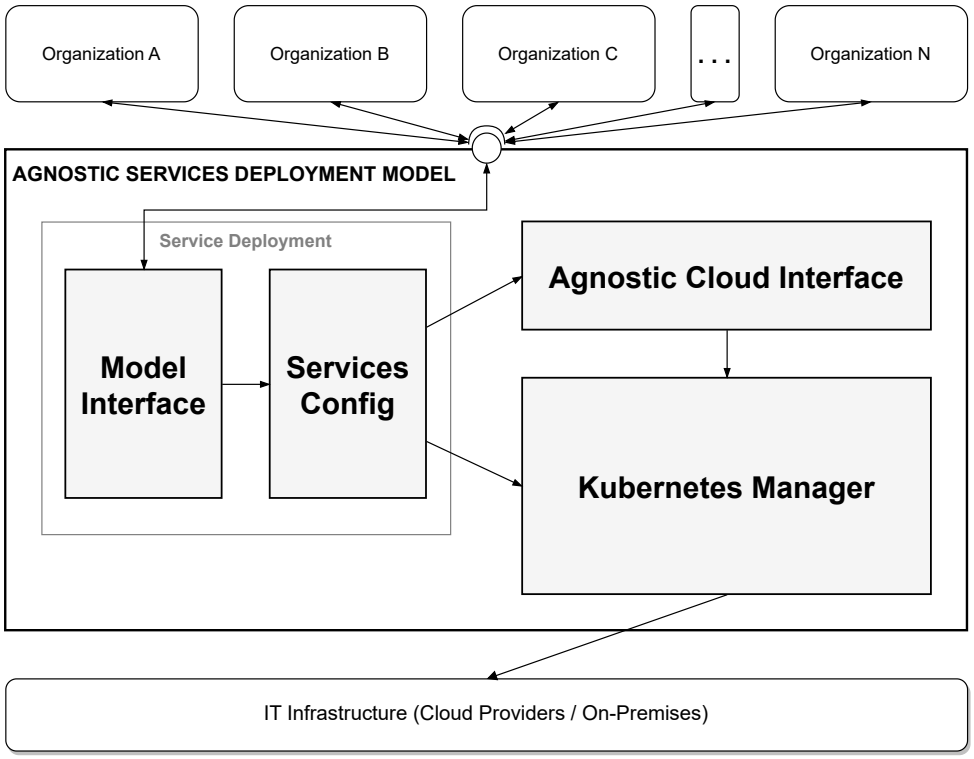


Figure 3.1: Model architecture of the proposed approach

remote infrastructure; **Services Configuration** component defines and creates the running configurations for the service to be deployed, and creates the required container images (if not provided). It also exposes an API to manage the deployed services. The **Agnostic Cloud Interface** component is responsible for abstracting different cloud providers, by configuring and adapting the services to conform to the selected cloud provider's guidelines, and it provides a configuration object for connecting to the Kubernetes cluster. Finally, the **Kubernetes Manager** component creates the required Kubernetes objects to fulfill the requirements of organization's services, and instantiates and manages them on a Kubernetes cluster (using a K8s client), hosted on the cloud provider's infrastructure or on-premises.

3.2 Detailed Architecture

Figure 3.2 provides a detailed view of the model's architecture as a cohesive whole, to show the interconnected nature of the entire system through a single visual representation. It captures the relationships and interactions between the various components, offering a high-level perspective. Ultimately, this diagram aims to illustrate how these components collaborate to achieve the model's objectives.

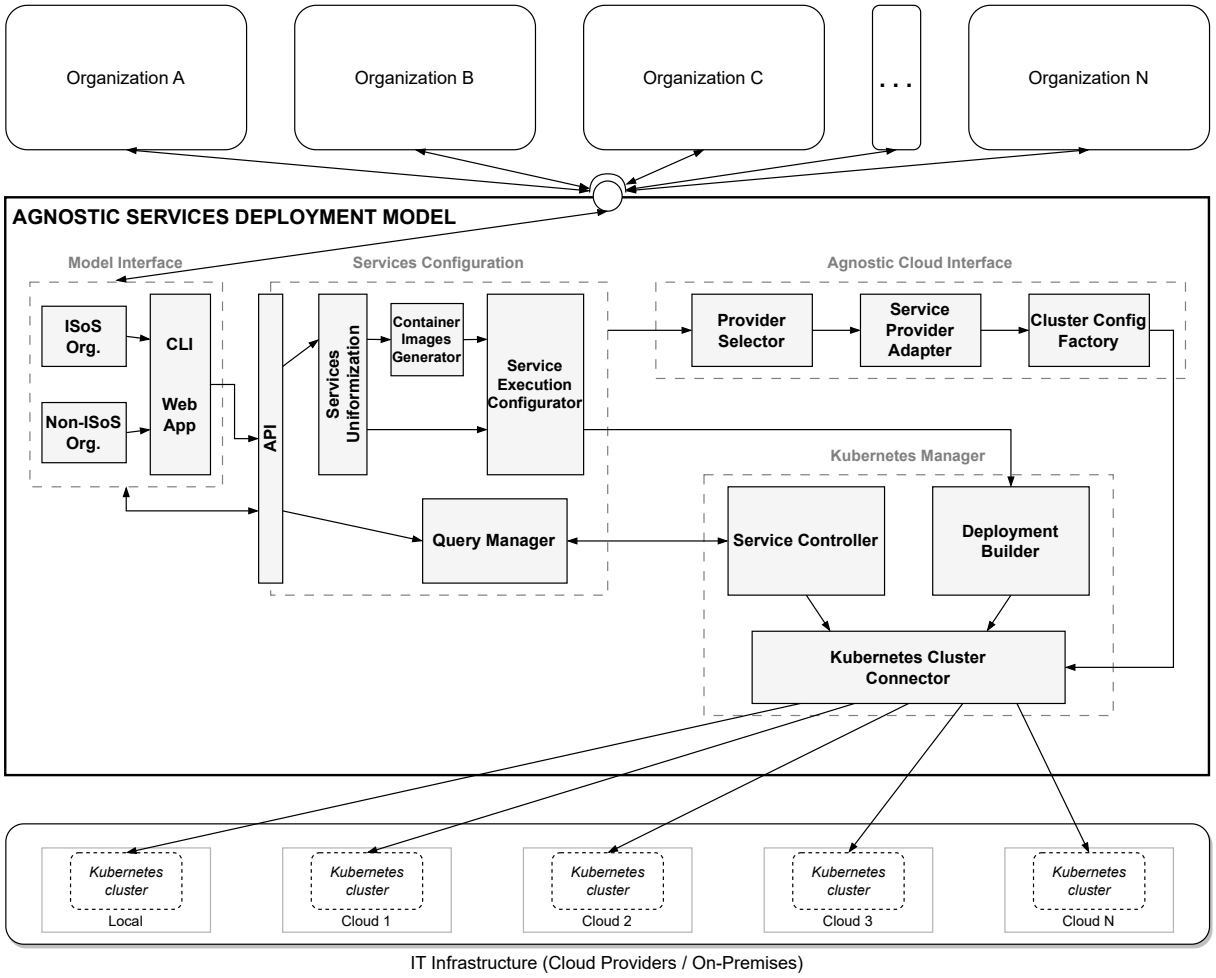


Figure 3.2: Detailed system architecture

Furthermore, the following diagrams and corresponding descriptions exhibit a more comprehensive view of the model's architecture, emphasizing the specific functions of each interconnected component, in order to provide a better understanding and specification of their capabilities. It is also noteworthy that all of these diagrams display a smaller representation of the model of Figure 3.1 in the top left corner, highlighting a black box that represents the component currently under discussion.

To provide a structured and logical organization for describing each diagram and its parts, each component is discussed individually. The **Model Interface** component is explored in Section 3.2.1, the **Services Configuration** component in Section 3.2.2, the **Agnostic Cloud Interface** in Section 3.2.3, and the **Kubernetes Manager** component in Section 3.2.4.

3.2.1 Model Interface

According to the proposed model, any organization can deploy services on multiple IT infrastructures (including public cloud or on-premises platforms), regardless of whether their technological artifacts are compliant or non-compliant to the structure and organization of the Informatics System of Systems (ISoS) [49] framework, represent by labels ① and ②, respectively. The model offers an interface for organizations to submit and manage the services they plan to deploy, through **Model Interface** component (Figure 3.3). This interaction is made possible through a unified interface that is intended to support any type of service or system, whether from ISoS-enabled organizations (represented by label ③) or non-ISoS-enabled organizations (label ④). At the time of writing this document, a CLI and a web interface are implemented and available for use, with the possibility of adding more in the future. Ultimately, this interface communicates with an API that exposes the system's functionalities (illustrated by labels ⑤ and ⑤′), as described in the Services Configuration component (Section 3.2.2).

Although beyond the immediate scope of this work, the model could also provide direct support and compatibility with the ISoS framework, or any other framework based on a System of Systems (SoS) [86] approach that formally structures and models technology artifacts from different suppliers. The motivation behind this potential support is to have the ability to build and manage the appropriate container images for the services to be deployed, regardless of how services are organized and modeled. In the case of ISoS model, this could involve the development of an `ISystem0 Metadata Converter`, requiring a more thorough exploration and association of concepts from

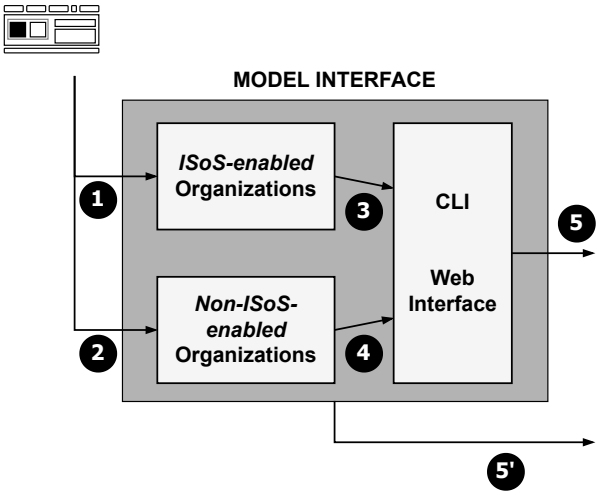


Figure 3.3: Model Interface component

both ISoS and Kubernetes technological frameworks, eventually resulting in a "concept mapping" between these two frameworks. Currently, assuming that the container images are either received or the necessary instructions for its construction are provided, the container image concept itself serves to standardize the various artifact structuring models.

3.2.2 Services Configuration

The subsequent stage involves the **Services Configuration** component (Figure 3.4), responsible for defining the running configurations for the service to be deployed, and generating the required container images (label ⑧ in Figure 3.4). This process requires specifying service's metadata and specifications, namely the container image (label ⑥), and other configuration settings (e.g, arguments or volumes), so that the Kubernetes platform can effectively manage and maintain the desired state for the service (label ⑦). If the container images already exist in the organization and adhere to OCI standards [5], they can be used. Containers' execution configurations are received using the interfaces exposed to organizations, and subsequently materialized in API requests (label ⑤). If not specified, default settings are employed. Moreover, this component exposes a service management API (label ⑤′), so that organizations can update or remove the previously deployed services (label ⑨).

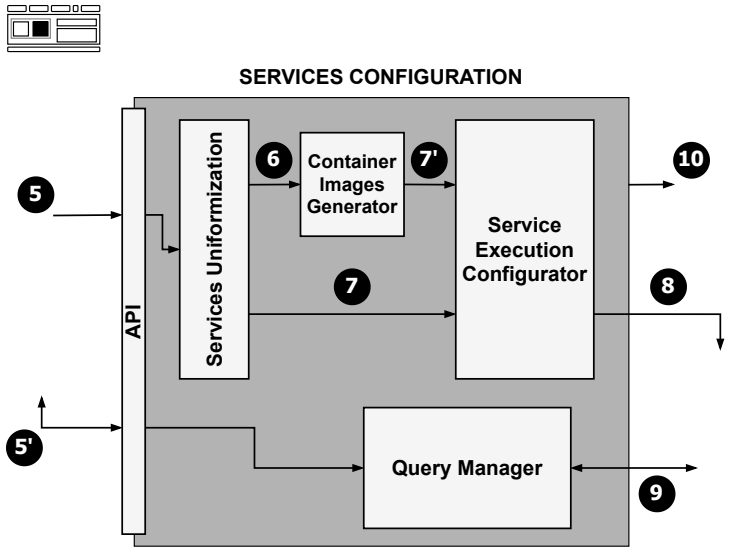


Figure 3.4: Services Configuration component

3.2.3 Agnostic Cloud Interface

The **Agnostic Cloud Interface** component (Figure 3.5) is responsible for abstracting different cloud providers, facilitating unified service deployment across multiple platforms. This is achieved through a three-step process. Firstly, the appropriate cloud provider is selected via `Provider Selector` module (label ⑩ in Figure 3.5), based on organization's requirements. Subsequently, the services (materialized in containers) are configured to conform with service provider's policies and specifications (e.g., to guarantee that the provider can access the specified container registry, or that the communication protocols used conform with security standards of the provider), via `Service Provider Adapter` module (label ⑪), creating an abstraction layer. Finally, `Cluster Config Factory` module (label ⑫) generates and provides a configuration object for connecting to the appropriate Kubernetes cluster. These client configuration objects enable the connection to a Kubernetes cluster, by incorporating the cluster's master URL and the necessary credentials to access the provider's platform. Such parameters are present in a kubeconfig file, which must be previously exported from the cluster (via provider's APIs or web interface) and made available to the model. These `kubeconfig` files are a valuable asset to the functioning of this mechanism for managing Kubernetes clusters, providing a simple and effective way to switch between different clusters and users when interacting with multiple clusters, namely through the definition of the desired *Context*, as described in 2.3.2.1.

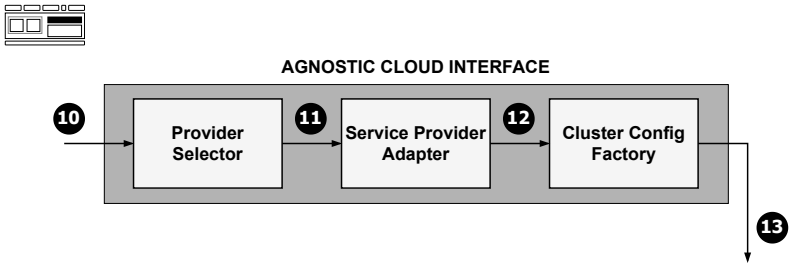


Figure 3.5: Agnostic Cloud Interface component

3.2.4 Kubernetes Manager

Finally, the **Kubernetes Manager** component (Figure 3.6) creates the required Kubernetes objects according to the characteristics of organization's services, via `Deployment Builder` module (label ⑧ in Figure 3.6). Additionally, the component includes the `Service Controller` module, responsible for building the queries necessary to retrieve the current state of the services from the cluster or to update them (label ⑨). To fulfill these requests (labels ⑭ and ⑮), a Kubernetes client is utilized to communicate with cluster's API Server, represented through label ⑯. For this purpose, the Kubernetes cluster must be first deployed on a public cloud provider or in organization's data centers. It is also worth noting that the decision to deploy the services in a Kubernetes cluster was strongly motivated by the capabilities it offers, namely automatic scaling, self-healing, and service discovery. These are key features for ensuring high availability and load balancing for the services, which ultimately contributes to improve the overall performance and reliability of the informatics systems.

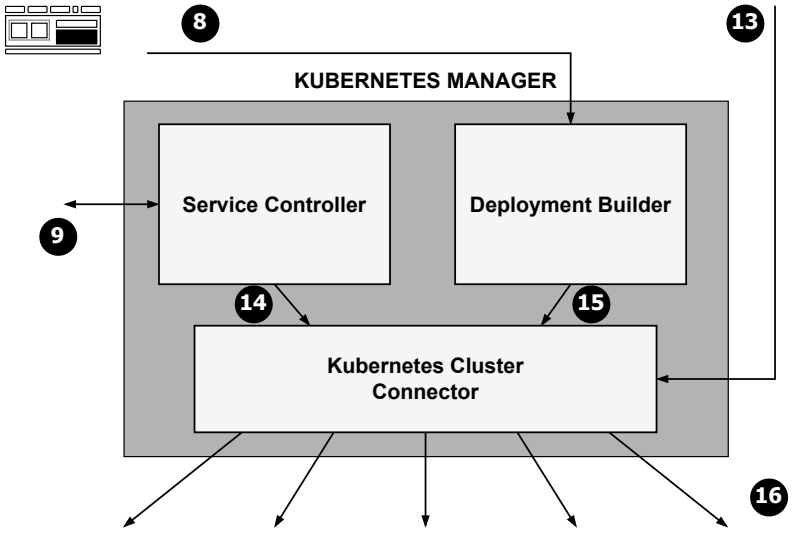


Figure 3.6: Kubernetes Manager component

3.3 Remote Kubernetes Cluster Architecture

Figure 3.7 illustrates the multiple types of IT infrastructures supported by the SDMAC framework, including both on-premises and public cloud platforms. Each provider is required to have at least one Kubernetes cluster, on which services are deployed. Notably, remote clusters, hosted on cloud providers, incorporate the Cloud Controller Manager component, responsible for managing provider's resources within the cluster, through API calls.

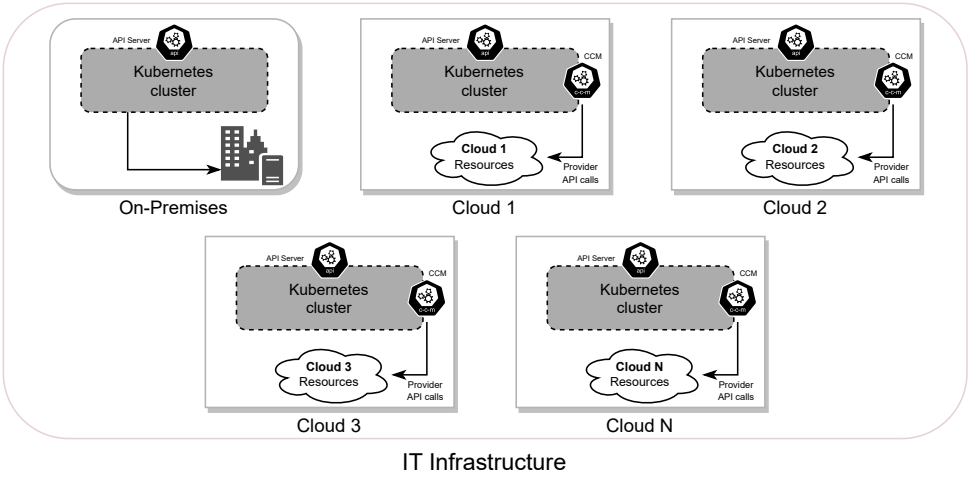


Figure 3.7: Supported IT Infrastructure

As depicted in Section 2.3.2.1, the **Cloud Controller Manager** (CCM) is a Kubernetes cluster component that manages the provider resources required for the cluster, such as load balancers, volumes and instances, as portrayed in Figure 3.8. Therefore, each cloud provider has its own implementation of the CCM, tailored to the cloud provider's specific APIs and services. Building a CCM means implementing an interface with functions designed to execute fundamental operations to manage resources inside the cluster, which are cloud-specific resources. In summary, the Cloud Controller Manager assumes a critical role in the proposed architecture and was a key factor in choosing the Kubernetes platform. By separating the components that interact with the cloud platform from those that communicate exclusively with the cluster, CCM enables the decoupling of service deployment from the underlying provider, which is fully aligned with the proposed framework's objectives and goals on achieving the desired architectural flexibility.

Figure 3.8 also illustrates the interaction between the proposed framework's model and a Kubernetes cluster, which is hosted on a cloud provider infrastructure, such as

Microsoft Azure. As elaborated in Section 3.1, users or developers from an organization interact with the system through Model Interface component. This action initiates the sequence of processes outlined in that section (3.1), culminating in the Kubernetes Manager component establishing communication with the cluster's Control Plane via its API Server (facilitated by a K8s client).

Additionally, the diagram of Figure 3.8 exhibits the Control Plane components responsible for making overarching decisions about the cluster, as well as detecting and responding to cluster events. Within this context, it's pertinent to emphasize the Cloud Controller Manager component connecting the cluster to the cloud provider's API. This is what further decouples components that interface exclusively with the cluster from those that interact with the specific cloud platform. In this example, the cloud platform's components represent several Microsoft Azure products and services.

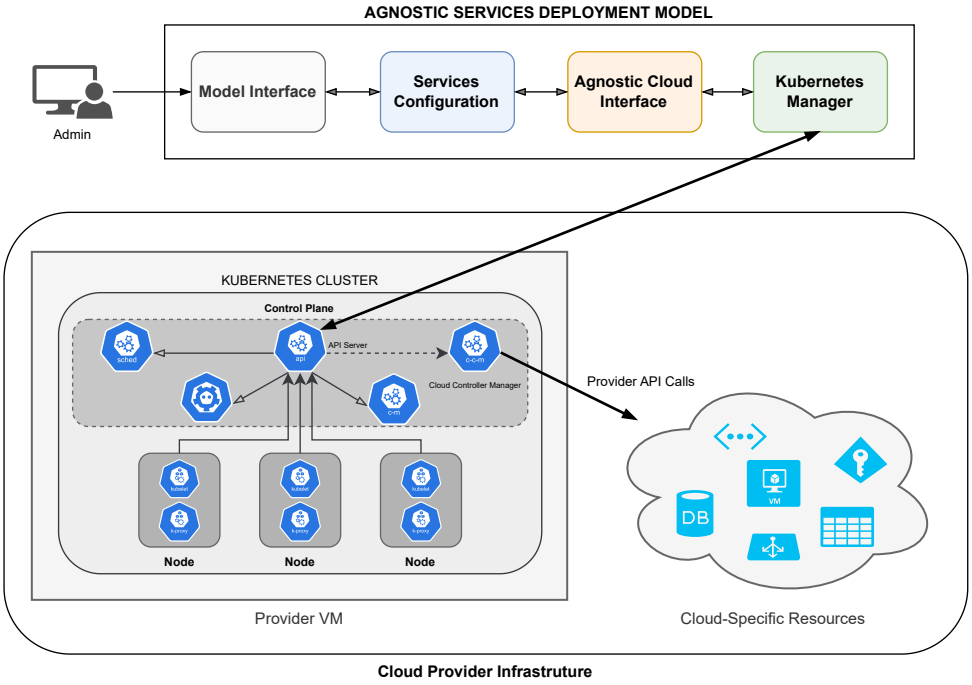


Figure 3.8: Interaction between the SDMAC framework and a Kubernetes cluster

At last, Appendix A shows the complete architecture of the model, focusing on the technologies and components to be used in model's implementation and validation.

3.4 Summary

This chapter introduces the proposed architecture for an approach that enables the deployment of services across different cloud platforms. The architecture allows organizations to deploy their services on various IT infrastructures. It consists of several

components that work together to provide a unified interface for service deployment. The architecture abstracts different cloud providers, selects the designated provider, and configures services to conform to the provider's policies. It also leverages Kubernetes for managing and scaling services. Overall, the proposed architecture offers organizations the flexibility and scalability to deploy services across multiple cloud platforms, aiming for performance and reliability.

4

Platform Implementation

The framework described in this document has been materialized and implemented as a platform. It is noteworthy that, in the context of this work, the term "platform" refers to a set of software components designed to allow applications and services to work together seamlessly [75]. The developed platform includes a web server, that exposes an API, and two user interfaces. More precisely, this platform must be characterized as a "prototype", or "platform prototype", as it is primarily intended to demonstrate and validate the essential features, concepts, and principles of the architecture proposed in Chapter 3. Furthermore, it is also important to clarify that the focus of the prototype is to demonstrate its cloud-agnostic capabilities and streamline service management, rather than to provide a complete solution.

The present chapter unveils the implementation of the referenced platform's prototype. Therefore, Section 4.1 describes the technologies employed in its development, encompassing those utilized in the prototype, as well as those inherent to the Kubernetes platform; the software architecture created for implementing the prototype is presented in Section 4.2; and Section 4.3 depicts the main implementation considerations of each framework's component.

4.1 Technologies and Services

The implementation of SDMAC framework involved two levels of technologies. The first level is based on the technologies used to directly develop the backend (Web

Server) and frontend (User Interface) of the prototype, described in Section 4.1.1, whereas the second one is related to Kubernetes technology supplied by the services providers, presented in Section 4.1.2.

4.1.1 Prototype Development

The reference implementation of the SDMAC framework's platform was supported by the Java ecosystem. Apache Maven was also used to manage the dependencies, structure the projects, and generate the Java ARchive (JAR) artifacts. Nevertheless, this implementation can be accomplished using other technological ecosystem, adhering to the same fundamental principles. The platform provides a Java-based command-line interface and a Web User Interface, supported by Eclipse Theia [93], for organizations to deploy and manage their services. In a first approach, both interfaces provided operations to manage services, receiving as input parameters: service name, container image, port, provider (cloud providers or on-premises), and K8s namespace. Users can deploy, remove, update, and migrate a service between providers, as well as list running services and get a specific service information.

4.1.2 Kubernetes Services

When exploring strategies for deploying web services across diverse cloud providers on Kubernetes clusters, the approach hinges on harnessing Managed Kubernetes Services. These services, offered by the majority of cloud providers, facilitate the deployment procedure and handle complex infrastructure operations, leading to a more efficient process.

Summarily, a Managed Kubernetes Service is a cloud-based service that simplifies and helps on the deployment, management, and scaling of Kubernetes clusters, making Kubernetes easier and more efficient to use. It handles the tasks of setup, upgrade, scaling, monitoring, and securing the clusters, allowing developers to focus on the deployment and management of applications, abstracting the underlying infrastructure. Popular cloud providers including Google Cloud Platform, Amazon Web Services, and Microsoft Azure offer their Managed Kubernetes Services, named Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), and Azure Kubernetes Service (AKS), respectively. It is worth noting that these cloud providers have implemented the Cloud Controller Manager within their clusters, which makes it feasible to offer the aforementioned services, as elaborated in Section 3.2.

4.2 Software Architecture of the SDMAC-based platform

The software architecture for the solution is represented through a simplified Unified Modeling Language (UML) Class Diagram, illustrated in Figure 4.1. This diagram serves as a "mapping tool" of the proposed model's architecture illustrated in Figure 3.1, to visually capture corresponding key classes and associations. In the diagram, the gray-colored classes represent Model Interface components, the blue-colored classes represent Services Configuration components, the green classes represent Kubernetes Manager components, and lastly, the classes in orange represent Agnostic Cloud Interface components. This representation aims to help describe the implementation considerations depicted in this chapter.

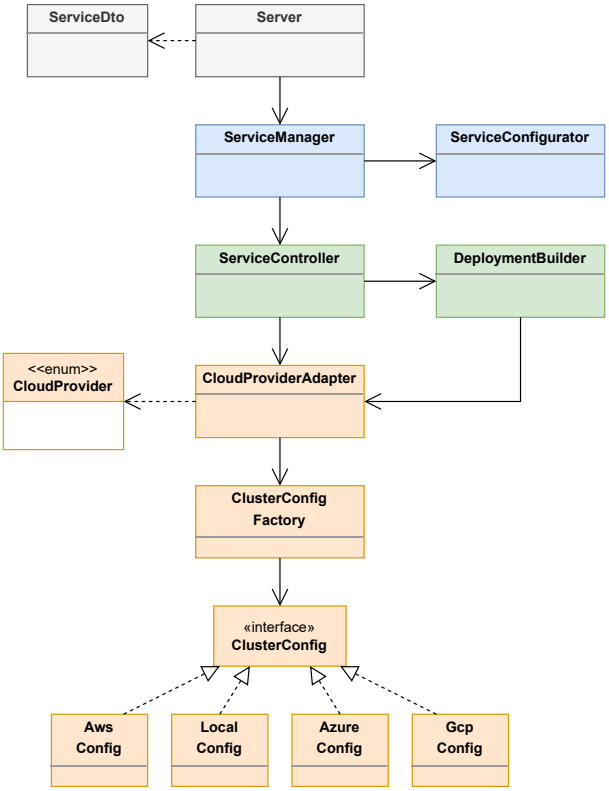


Figure 4.1: Simplified UML Class Diagram

The classes and interfaces represented in Figure 4.1 can be described as follows:

- `Server` is a class that acts as a server to receive and handle requests from the User Interfaces (UIs).
- `ServiceDto` class serves as Data Transfer Object (DTO) to aggregate services' data from the UI request.

- `ServiceManager` class exposes an API with multiple endpoints to manage and deploy services.
- `ServiceConfigurator` is a class whose instances define the running configurations and container images for the service to be deployed.
- `DeploymentBuilder` class is responsible for building the required Kubernetes resources and objects within a cluster, according to the operation to be executed.
- `ServiceController` class builds the queries necessary to retrieve the current state and to update services from the cluster, by interacting with Kubernetes clusters' API Server.
- `CloudProviderAdapter` class selects the designated cloud provider and validates the services to conform to provider's guidelines.
- `ClusterConfigFactory` class is a factory of configuration objects for connecting to Kubernetes clusters.
- Each cloud provider, project-represented, supported by the platform implements the `ClusterConfig` interface methods to generate the configuration objects for connecting to the proper Kubernetes cluster. For demonstration purposes in the scope of this work, support was provided for cloud providers Microsoft Azure, Google Cloud Platform and Amazon Web Services, materialized by `AzureConfig`, `GcpConfig` and `AwsConfig` classes, respectively. A local environment is also support through `LocalConfig` instances.

Appendix B includes the complete UML Class Diagram, playing a pivotal role in conveying the conceptual framework underpinning the developed software solution. This diagram aims to encapsulate the fundamental classes, their associated attributes, methods, and interconnections. Built upon the simplified version presented in Figure 4.1, this diagram elaborates on the diagram's components, providing a more insightful overview of the system's design.

4.3 Reference Implementation

This section delves into the core implementation decisions and fundamental principles underlying to each component (Services Configuration component in Section 4.3.1,

Model Interface in Section 4.3.2, Agnostic Cloud Interface in Section 4.3.3, and Kubernetes Manager in Section 4.3.4). The described aspects are elucidated through components' corresponding classes and interfaces, collectively forming the entirety of the prototype.

4.3.1 Services Configuration

This section focuses on the **Services Configuration** component, which plays a crucial role in defining the parameters and configurations for services. Specifically, the section explores the development of the Service Management API (Section 4.3.1.1) and details data representation, through an underlying data structure, used to facilitate service management (Section 4.3.1.2).

4.3.1.1 Service Management API

The developed platform exposes an API designed to address the inherent challenges of managing multiple web services across cloud providers and Kubernetes clusters. This API assumes a central role, serving as central interface through which backend services of the platform are made accessible, receiving requests from both the Command Line Interface and the Web User Interface. Built on RESTful principles, the API facilitates interactions between the interfaces and underlying backend services.

Leveraging API's functionalities, users can deploy, remove, update and migrate a service between providers, list running services and get a specific service information. The management operations of these services are executed by taking input parameters related to the configuration of the service, container specifications, and provider infrastructure settings.

A summary of the API's documentation, listing each functionality endpoint, corresponding HTTP method, endpoint description, and required parameters, is available in Appendix C.

4.3.1.2 Data Representation

To facilitate the instantiation of the input services within a K8s cluster while adhering to the requirements imposed by Kubernetes principles and concepts, a specific data structure was formulated. This structure purports to map and aggregate all pertinent information of the services into a singular entity, serving as a unified repository, regardless of the service under consideration. Figure 4.2 presents the simplified developed

Entity-Relationship (ER) model to achieve this objective, whose entities are described in Table 4.1.

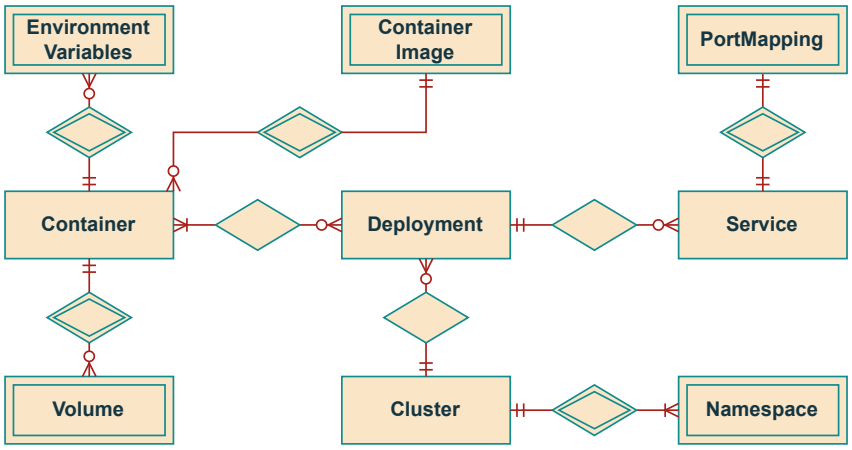


Figure 4.2: Simplified Entity–Relationship Model

Expressing the textual interpretation of this model, a **Deployment** comprises one or more **Containers**, each instantiated using a corresponding **Container Image**. These Containers can be configured with multiple **Environment Variables**, and mounted on a set of **Volumes**. Moreover, a Deployment is exposed to external access by a **Service**, which involves specifying **PortMappings**. Finally, the Deployment is instantiated within a Kubernetes **Cluster**, typically organized into **Namespaces**.

For a more comprehensive specification of this data structure, the ER model present in Appendix D was developed. Furthermore, Appendix E describes the entities' attributes selected for this model.

When developing the prototype of the platform, it was established that the operations exposed by the CLI would be based on a simplified version of the aforementioned data structure, mainly to expedite the development process. As a result, it is possible to focus on the essential parameters for deploying a service in a Kubernetes cluster, including service name, container image, port, provider, and K8s namespace. This simplification strategy provides efficient prototyping for development environments, to clarify core concepts and to help proving the viability of fundamental functionalities. On the other hand, the Web Interface provides more features and capabilities that can be applied to a real-world deployment and management platform for web services, where a more comprehensive and scalable data structure, such as the one presented in this section, offers the possibility of more extensive customization for a service, as well as support for less common types of services. Further details into the development and operation of both the CLI and Web Interface will be presented in Section 4.3.2.

Moreover, the Volume entity was intentionally excluded from the prototype (from

Table 4.1: ER model entities description, adapted from [38]

Entity	Description
Deployment	A configuration which defines how Kubernetes deploys, manages and scales container images. Kubernetes will ensure that the system matches this configuration.
Service	A Kubernetes service lets a deployment receive traffic and defines how a deployment is exposed.
Port Mapping	Exposes a deployment as a new service, providing networking and IP support to deployment's pods.
Container	Lightweight packages of an application code together with dependencies such as specific versions of programming language runtimes and libraries required to run software services.
Container Image	A portable machine image that bundles together an application and its dependencies, identified by the endpoint to its location.
Environment Variable	Variables can be set for containers to see certain information about their environment or inject specific data, including access to secrets for security credentials.
Volume	A mechanism for persisting and sharing data generated and utilized by containers, managed independently of container instances.
Cluster	A deployment will use compute instances managed in a logical grouping called a 'cluster', in order to share compute resources and logical groupings of jobs and applications.
Namespace	Groups inside clusters intended for use in environments with many users spread across multiple teams, or projects.

both CLI and Web Interface operations), as it was not considered essential for the intended validation and could introduce unnecessary entropy into the system, delaying the proof of concept. It is pertinent to recall the focus of the prototype is to demonstrate its cloud-agnostic nature, not to build a complete platform.

4.3.2 Model Interface

The **Model Interface** component is the part of the platform that serves as user interface for interacting with the system. This section introduces the Command-Line Interface (Section 4.3.2.1) and the Web User Interface (Section 4.3.2.2) as the front-end solutions for the platform, highlighting their functionalities and usage. Insights into their development are also included.

4.3.2.1 Command-Line Interface CLI

The functionalities provided by the platform through the command-line interface are described in Table 4.2, which outlines the available operations, along with their corresponding command option and required parameters for each option. These operations are executed using a specific endpoint of `ServiceManagementAPI`.

Table 4.2: Operations exposed by the platform, via CLI

Operation	Option	Parameters
Deploy Service	`deploy`	{Service name}, {Container image}, {Port}, {Provider}, {K8s namespace}
Deploy Service (builds container image)	`deploy -build Image`	{Service name}, {Dockerfile path}, {Context directory}, {Registry username}, {Port}, {Provider}, {K8s namespace}
Remove Service	`remove`	{Service name}, {Provider}, {K8s namespace}
Update Service	`update`	{Service name}, {Provider}, {K8s namespace}, {Container image}
Service Migration	`migrate`	{Service name}, {Source provider}, {Source K8s namespace}, {Target provider}, {Target K8s namespace}
List Running Services	`list`	{Provider}, {K8s namespace}
Get Service State	`state`	{Service name}, {Provider}, {K8s namespace}

In summary, to deploy a service, users can use the **deploy** operation with parameters including the service name, container image, port, provider (cloud provider or on-premises), and Kubernetes namespace. If the container image for the service does not exist, users can use the `-buildImage` flag in `deploy` option, passing as arguments the Dockerfile path, the context directory and registry username, so that the platform can build and register the image accordingly, and use it to deploy the service. Services can be removed using the **remove** operation, which takes the service name, provider, and K8s namespace as input. Users can also **update** a service, namely its container image version, receiving as parameters related to the deployed service and the new container image location. Additionally, there is a **migrate** operation to move a service from one provider and K8s namespace to another, requiring parameters for the service name, source/target provider, and source/target namespace. To view running services, the **list** operation can be utilized with options for the provider and namespace. At last, to retrieve the state of a specific service, the **state** operation requires the service name, provider, and namespace as parameters. The simplicity inherent to these operations and corresponding usage aim to provide an efficient and straightforward service management on the platform.

4.3.2.2 Web User Interface

The platform's Web User Interface was built and developed using an Eclipse Theia extension [94]. The **tree-editor** extension [95] creates a tree editor to display the content of JSON files. This Theia-based application allows users to create and load `.tree` files, which are opened with the tree editor. The editor is composed by two panels: the left side on the editor displays the hierarchy of the JSON data and allows to create or remove nodes and corresponding children; the right side shows the properties of a selected node and allows its modification. As a side note, the project was developed using TypeScript programming language [98], and the project's structure was built with a yeoman generator [104], provided by Theia platform.

In the scope of the proposed solution, it was established that a Node represents a **Deployment** and a Node's children corresponds to a **Service**. This approach allows users and developers to use the platform to create a Deployment and to have multiple Services associated to it, while allowing to have multiple Deployments loaded on the workspace (this relationship and further details are described on Section 3.2.2). The developed UI presents a third panel that offers a service's management menu, composed by a set buttons that trigger the operations of creation, removal, update, migration of the Service opened on the editor. An option to list the services in the selected

cluster/namespace pair is also included. Leveraging UI's functionalities, services can be easily created, removed on update within a Deployment.

Figure 4.2 displays a Deployment object, composed by a basic "Hello World" service, opened on the developer editor. This object is also represented and persistently stored on the file system through a `.tree`, as shown in Appendix F.

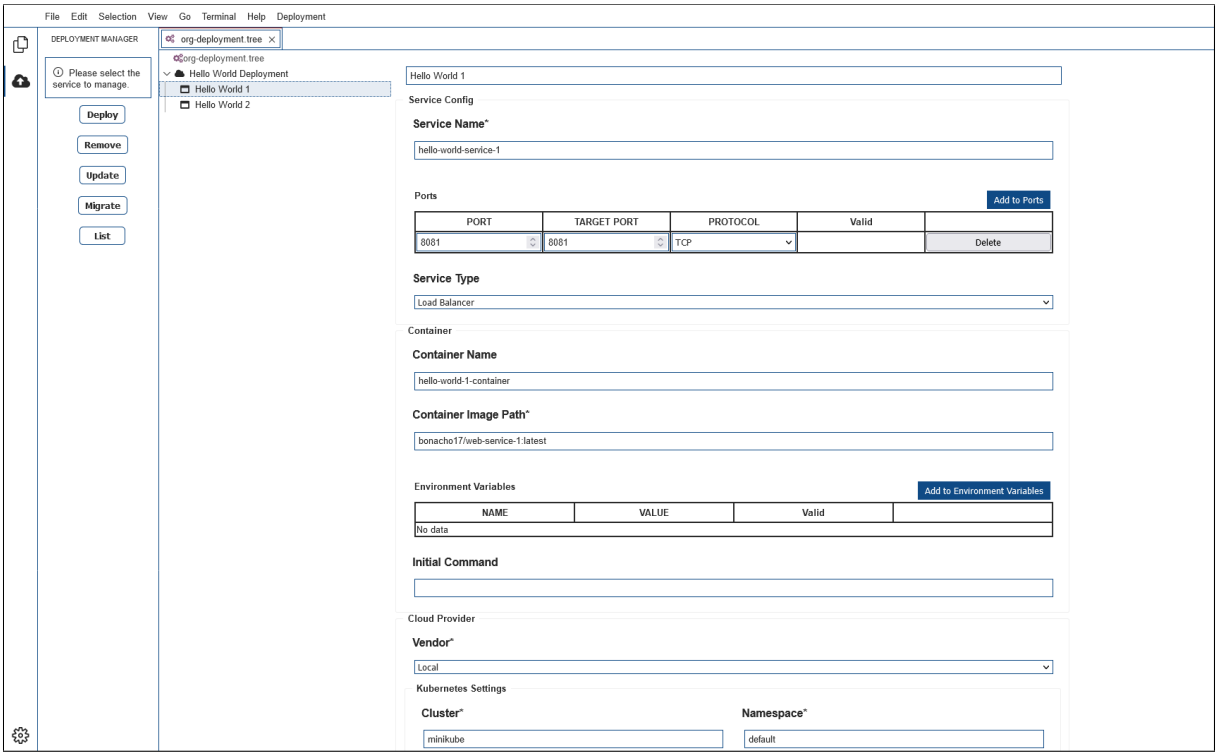
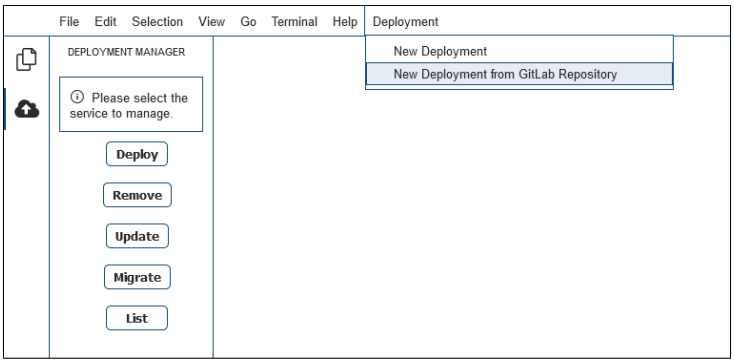


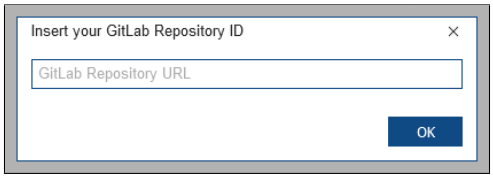
Figure 4.3: Web Interface with "Hello World" web service configuration

Furthermore, one essential feature added to the platform's web interface was the integration of GitLab repositories support. GitLab repositories inherently incorporate a container registry, simplifying the storage of container images for projects and applications hosted within the repository. Based on this mechanism, the primary objective behind this feature is to automate the process of generating and populating Deployment configuration files and corresponding Services objects based on the container images stored in the repository. By leveraging this feature, users or developers from an organization can navigate to the `Deployment` tab and select the `Deployment from GitLab repository` option (illustrated in Figure 4.4a). Then, inputting the repository URL (as shown in Figure 4.4b) is all that's required. Subsequently, the front-end application accesses the repository's container registry and generates the necessary configuration files and objects (i.e., Deployments and its Services) based on the container images found. To mention that handling the challenge of accessing private container images and making them accessible to the cluster is further discussed in

Section 4.3.4.2. In summary, this functionality stands as a pivotal asset in simplifying deployment efforts for developers and reducing the possibility of errors, such as misspelling the container image's name or version. Moreover, future enhancements to this application, particularly to this feature, could include advanced filtering capabilities for container images based on their versions or names, providing organizations greater control over this mechanism.



(a) *New Deployment from GitLab Repository* menu option



(b) *Insert your GitLab repository ID* dialog

Figure 4.4: GitLab-based features of the Web Interface

4.3.3 Agnostic Cloud Interface

This section describes the implementation of the **Agnostic Cloud Interface** component and its role in platform's design. Thus, it examines the modular architecture (Section 4.3.3.1) and configuration object factory (Section 4.3.3.2) that allow the platform to remain cloud provider agnostic, by seamlessly integrating new cloud providers.

4.3.3.1 Modular Architecture

The core implementation principle of this work dwells on its modular architecture. Each cloud provider is represented through an independent Java/Maven project that implements certain required interfaces. This creates an abstraction layer allowing the main project to remain agnostic from the underlying cloud provider implementations, thereby reducing dependencies and promoting flexibility.

In pursuit of a more modular architecture, the primary goal was to seamlessly integrate new cloud providers into the platform without having to modify and recompile the main project. This was achieved using Java's Service Provider Interface (SPI), a dynamic mechanism that allows to define and load implementations at runtime, eliminating the need for recompilation. The implementation of this pattern began with the creation of the `ClusterConfig` interface, which contains the necessary methods for generating configuration objects. Each cloud provider's configuration class was then required to implement this interface with provider-specific logic. In the main project, the `ClusterConfigFactory` class uses the Java's `ServiceLoader` class to dynamically load provider's implementations at runtime, as shown in Listing 4.1. Notably, each provider-specific project includes a file in the `META-INF/services` directory, specifying the fully qualified name of its `ClusterConfig` implementation (e.g., `agnosticcloudinterface.azure.AzureConfig`, for Microsoft Azure). In essence, this approach adheres to the Open-Closed Principle, enabling system extensibility without altering existing code. This means that the platform's main project remains unchanged, facilitating the seamless integration of new providers, as intended.

Listing 4.1: Cluster Config Factory implementation supported by SPI mechanism

```
1  public class ClusterConfigFactory {
2
3    public static ClusterConfig createConfig(String provider) {
4          ServiceLoader<ClusterConfig> loader = ServiceLoader.load(
     ClusterConfig.class);
5          for (ClusterConfig config : loader) {
6            if (config.supports(provider)) {
7                  return config;
8               }
9          }
10         // If provider is not found
11         return new LocalConfig();
12     }
13
14 }
```

4.3.3.2 Configuration Factory

The modular approach described in Section 4.3.3.2 is further complemented by the use of the programming Factory pattern. Using this pattern, the configuration objects can

be created without exposing their creation logic to the client application and can be referenced using a common interface, which reduces coupling and facilitates extensibility, as intended.

The interface that the projects representing a provider implement is closely related to the `Cluster Config Factory` component (Figure 3.5), which acts as a factory of configuration objects for connecting to Kubernetes platform. Listing 4.2 presents a simplified Java code for implementing this interface, which generates and returns a client configuration object for connection to a cluster/namespace pair of a cloud provider or local infrastructure. It is noteworthy that this code is invoked after selecting the required cloud provider by `Provider Selector` module of `Agnostic Cloud Interface` component.

Listing 4.2: Generation of a Kubernetes configuration object

```
1  public Config getConfig(String kubeconfigPath, String namespace) {
2    Config config = null;
3    try (InputStream inputStream = new FileInputStream(kubeconfigPath
     )) {
4      Yaml yaml = new Yaml();
5      Object kubeconfigObject = yaml.load(inputStream);
6      String kubeconfigYaml = Serialization.yamlMapper().
     writeValueAsString(kubeconfigObject);
7      config = Config.fromKubeconfig(kubeconfigYaml);
8      config.setNamespace(namespace);
9    } catch (IOException e) {
10     e.printStackTrace();
11   }
12   return config;
13 }
```

4.3.4 Kubernetes Manager

In this section, the Kubernetes Manager component is discussed as the element responsible for the platform's interaction with Kubernetes clusters. Specifically, it provides code examples and detailed explanations on creating and managing Kubernetes resources on remote clusters (Section 4.3.4.1), and how to handle authentication in private container registries (Section 4.3.4.2).

4.3.4.1 Kubernetes Client

The interaction between the developed platform and the clusters' Control Plane (namely API Server) is accomplished using the Fabric8 platform [37], specifically through its Kubernetes Client library [56]. This open-source Java client library enables access to Kubernetes REST APIs via a fluent Domain-Specific Language (DSL), providing a convenient and straightforward way to programmatically interact with Kubernetes clusters. Consequently, it facilitates the creation and management of Kubernetes resources and objects within a cluster, while also enabling the automation of deployment workflows.

Listing 4.3 presents a code snippet for creating a Kubernetes Deployment object in a Kubernetes cluster using the DSL provided by Kubernetes Client library, in Java.

Listing 4.3: Creation of a Kubernetes deployment object

```
Deployment deployment = new DeploymentBuilder()
    .withNewMetadata()
      .withName(deploymentName)
        .withLabels(labelsMap).addToLabels(labelsMap)
    .endMetadata()
    .withNewSpec()
      .withNewSelector()
        .addToMatchLabels(labelsMap)
      .endSelector()
      .withReplicas(NUMBER_OF_REPLICAS)
      .withNewTemplate()
        .withNewMetadata()
          .withLabels(labelsMap).addToLabels(labelsMap)
        .endMetadata()
        .withNewSpec()
          .addNewContainer()
            .withName(containerName)
            .withImage(containerImage)
            .withArgs(initialCommands)
            .withPorts(ports.stream()
                .map(port -> new ContainerPortBuilder()
                    .withContainerPort(port.getTargetPort())
                    .withHostPort(port.getPort())
                    .withProtocol(port.getProtocol().name())
                    .build())
                .collect(Collectors.toList()))
```

```
27                .withEnv(envvars)
28              .endContainer()
29            .endSpec()
30          .endTemplate()
31        .endSpec()
32      .build();
```

The service used to expose the Deployment of Listing 4.3 can be created as shown in Listing 4.4, using the same library and programming model.

Listing 4.4: Building of a Kubernetes Service

```
1
2   ServiceSpecBuilder spec = new ServiceSpecBuilder();
3   spec.withSelector(labelsMap);
4   spec.withType(serviceType);
5   List<ServicePort> servicePorts = new ArrayList<>();
6   for (PortMapping port: ports) {
7     ServicePort servicePort = new ServicePort();
8     servicePort.setName("port" + servicePorts.size());
9     servicePort.setPort(port.getPort());
10    servicePort.setTargetPort(new IntOrString(port.getTargetPort()));
11    servicePort.setProtocol(port.getProtocol().name());
12    servicePorts.add(servicePort);
13  }
14  spec.addAllToPorts(servicePorts);
15  Service service = new ServiceBuilder()
16      .withNewMetadata()
17        .withName(serviceName)
18          .withLabels(labelsMap)
19      .endMetadata()
20      .withSpec(spec.build())
21    .build();
22  service = client.services().inNamespace(namespace).createOrReplace(
     service);
```

4.3.4.2 Registry Authentication

Another relevant and challenging task consists on pulling container images from private container image registries or repositories. Based on [77] and [78], pulling images

from private registries involves a two-step configuration process in order to provide Kubernetes the needed credentials. As a reference for demonstrations purposes, GitLab's Container Registry will be employed. Parenthetical, the authentication in a GitLab's repository can be accomplished using user data and password or by using user data and personal access token, this one usually when using GitLab's two-factor authentication. The first step requires to create a Secret based on credentials to access the container registry. The authentication data can be set using a JSON file, identical to the one in Listing 4.5, where it is defined the username, password and email to access the repository. In this example, `registry.gitlab.com` represents the Fully Qualified Domain Name (FQDN) for the private docker registry server used in this demonstration. Afterwards, the Secret can be created as represent in Listing 4.6, line 11.

Listing 4.5: GitLab repository credentials example in JSON format

```
{
    "auths": {
        "registry.gitlab.com": {
            "username": "Bonacho17",
            "password": "xxxxxxxxxxxxxxxxxxxxxxxxx",
            "email": "goncalobonacho@hotmail.com"
        }
    }
}
```

Listing 4.6: Docker registry secret

```
    // Create the Docker registry secret
    Secret gitlabRegistrySecret = new SecretBuilder()
        .withNewMetadata()
          .withName(REGISTRY_SECRET_NAME)
        .endMetadata()
        .withType("kubernetes.io/dockerconfigjson")
        .addToData(".dockerconfigjson", base64EncodedConfigJson)
        .build();

    // Create or replace the secret
    client.secrets().createOrReplace(gitlabRegistrySecret);
```

The second step involves configuring the deployment definition to use this Secret. Therefore, when defining or updating the Kubernetes object to be manipulated on the cluster, namely deployment's container specification, the `imagePullSecrets` must

be used as the field that specifies where Kubernetes should get the credentials from. Listing 4.7 shows an updated version of the container specification of the Deployment created on Listing 4.2 in order to use the required GitLab's credentials when pulling the container image from the repository. With this configuration set, any container image from a specified GitLab repository can be deployed on the Kubernetes Cluster.

Listing 4.7: Specifying how the cluster can access the container image

```
1      .addNewContainer()
2        // Container specific configuration...
3      .endContainer()
4      .withImagePullSecrets(
5        new LocalObjectReference(REGISTRY_SECRET_NAME)
6      )
```

4.4 Summary

This chapter provided a practical understanding into the implementation of the SD-MAC's platform. It started by detailing the technology stack employed, with Java and Maven as the core for back-end development and Eclipse Theia for the web-based user interface. The software architecture for the solution was examined through a simplified and a complete UML class diagram. Regarding reference implementation, the chapter emphasizes the modular architecture approach, supported by Java's Factory Design and Service Provider Interface patterns, allowing seamless integration of cloud providers. In essence, by reviewing implementation considerations, this chapter fills the gap between the theoretical concepts discussed in the previous chapters and the actual platform functionality, while enhancing the perception of the system.

5

System Validation

This chapter focuses on the validation of the developed platform's prototype, which involves the deployment and management of services in multiple cloud platforms. For this purpose, Section 5.1 introduces and outlines chapter's main objectives. Then, Section 5.2 describes the test environment, in terms of cloud and local providers, test services, validation using industry projects, and other relevant elements. Finally, Section 5.3 delves into practical use cases to explore and analyze platform's capabilities.

5.1 Validation Objectives

The validation of the developed deployment and management platform is guided by a set of key objectives that intend to assess the platform's main functionalities through the available operations. Accordingly, it aims to prove that it can seamlessly deploy services across multiple cloud providers while maintaining availability, through a simplified user interface. In particular, the focus of this effort must be on mitigating vendor lock-in. Ultimately, it must demonstrate readiness for real-world deployment scenarios, while remaining aware of its prototypical nature.

5.2 Test Environment and Scenarios

For the purpose of conducting rigorous validation tests, a controlled yet dynamic test environment, aiming to replicate real-world conditions, was created. This section depicts key elements of this test environment, namely the selection of cloud providers and local environment technologies (Section 5.2.1), Kubernetes clusters configuration (Section 5.2.2), GitLab repository for CI/CD processes (Section 5.2.3), and the validation with web services (Section 5.2.4). Furthermore, it describes the Portuguese Vehicle Speed Enforcement Network (SINCRO) project, part of the National Road Safety Authority (ANSR) IT infrastructure, used to validate the developed platform (Section 5.2.5).

5.2.1 Cloud Providers and Local Environments

The cloud providers selected for this study were Microsoft Azure, Google Cloud Platform, and Amazon Web Services due to their significant market share and status as industry leaders, as detailed in Section 2.4.1.1. This choice is intended to make the results more relevant and applicable.

In addition to the major cloud providers, the validation was also performed on a local Kubernetes cluster using Minikube [68]. Minikube allows developers to run a single-node Kubernetes cluster on their local machines, providing a controlled environment for testing and troubleshooting. This local testing complements the cloud-based validation efforts, ensuring that the platform provides support across diverse environments.

5.2.2 Kubernetes Clusters

When setting up the test environment, a critical step was to configure Kubernetes clusters as deployment target providers for the services. The primary objective was to establish a straightforward and consistent deployment scenario. Therefore, default settings provided by the selected cloud providers' web interfaces were employed to instantiate these clusters.

5.2.3 GitLab Repository

A GitLab repository specifically created to host the source code of two "Hello World" services and corresponding container images, stored in the GitLab Container Registry.

This repository also served a crucial role in configuring CI/CD workflows, enabling the simulation of updates to the container images for the web services.

According to [16], a proper CI/CD setup is the one that "allows software development teams to iterate quickly by automating some of the code review process: checking for proper code formatting, running all tests, and ensuring it builds without errors (and, hopefully, without warnings)". This means that upon the integration of new code into the master branch, a mechanism that facilitates the automatic deployment of the code to a staging or Quality Assurance (QA) environment, is activated. This automation enables thorough testing before advancing to deployment in a production environment. Same article asserts that, in the past, independent platforms like Jenkins were employed as CI/CD solutions. However, nowadays, both GitHub and GitLab have incorporated CI/CD functionalities directly into their products and services, which facilitate developers' tasks in this context.

Within this frame of reference, building CI/CD processes to automatically generate new container images for each commit/push in a repository is a common practice and it's becoming a straightforward process, particularly due to version control systems' built-in support. For some of the use cases described in this section, using a GitLab repository, that has built-in support for CI/CD pipelines, the setup creation of this mechanism starts by creating a `.gitlab-ci.yml` file at the root repository to define the CI/CD process. This file is divided into multiple stages, e.g., "build" and "deploy". The "build" stages uses the existing Dockerfile to create the Docker image, whereas "deploy" stage specifies that the newly created image should be to registered in GitLab's Container Registry. Finally, "Triggers" action section must be configured to establish that this CI/CD should be triggered upon commits, pushes, or tags.

5.2.4 Web Services

For this demonstration purpose, simple "Hello World" web services were implemented in Java. These services were built into a JAR file using Maven, and Docker was subsequently used to generate the container images and to register them in a container registry, namely Docker Hub. For reference throughout this chapter, **Web Service 1** represents the service that prints "Hello World 1" in an HTML page, and **Web Service 2** follows the same logic but outputs "Hello World 2".

5.2.5 SINCRO Project

The **National Speed Enforcement Network** (**SINCRO**) case, started in 2010 by ANSR, was created to develop a Portuguese national vehicle speed enforcement network [49]. This network is composed by multiple enforcement positions, which are cabinets for road traffic control. Each cabinet requires a cinemometer to be plugged into the cabinet.

To gain a clearer understanding of this project, [49] describes the SINCRO open specification consisting of three main parts:

- The **cabinet** (or **cabin**), with specific physical dimensions, electrical interfaces, etc.; with a removable case to hold the radar system, composed by the cinemometer and its parts; and with a monitoring interface based on the Simple Network Management Protocol (SNMP) and a specialized Master Information Base (MIB) data model;
- The **cinemometer**, i.e., the radar subsystem, including a computational programmatic interface for the exchanging traffic events (namely, the evidence of speeding) with another subsystems; a monitoring interface based on the SNMP protocol; and a standard fix-structure that plugs the cinemometer into the cabinet;
- The **Traffic Events Management System** (SIGET) subsystem, which is responsible for the centralized collection and validation of speed events, configuration procedures (e.g, speed limit), and report generation.

According to [73], a fundamental concept underlying these systems is that each network equipment is associated with a MIB, serving as a modeling element that represents the attributes of technology components within a computer network. The SNMP, which monitoring systems use to retrieve the status and properties of the elements they oversee, is also associated with this concept. Furthermore, open source monitoring solutions, such as Zabbix [106], can be used to monitor these informatics systems, ensuring tasks such as verifying system responsiveness and checking if specific system metrics align with predefined thresholds.

The initial approach to this system involved creating a dedicated cyber-physical system for each cabin and subsystem. However, this approach proved to be both expensive and challenging in terms of development. Therefore, to streamline development and reduce complexity, these cyber-physical systems were replaced by informatics systems that emulate the same behavior. Notably, these simulations adhere to the same principles and functionalities, implementing the same protocols, with the only difference

being the absence of physical components. Subsequently, in a third phase, these informatics systems were encapsulated in Docker containers for faster deployment and startup, and improved cost-effectiveness. As a result, the current setup for the purpose of this validation consists of cabins represented by containers, built from a container image template.

Finally, the existing informatics systems are evolving to adopt the ISoS framework, which involves structuring the operating computational entities as `Service` abstractions and these as abstractions of informatics systems (`ISystem`) [73]. Furthermore, in order to build and participate in a collaborative network, the aforementioned systems must implement the $\texttt{ISystem}_0$ meta-informatics system, which is responsible for maintaining metadata about the remaining informatics systems, thus adopting an ISoS IT landscape.

5.3 Use Cases

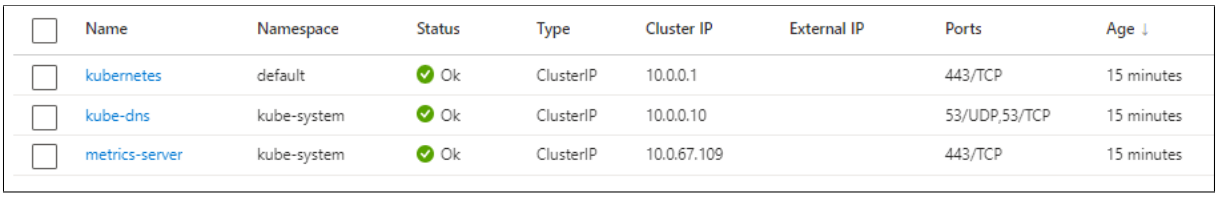
This section explores practical scenarios essential for the validation of this system, reflecting real-world challenges in deploying and managing services (mostly Web services) across multiple cloud providers. In this regard, the selected use cases include: deploying the basic "Hello World" web services on various cloud providers (Section 5.3.1), the migration of services between cloud providers, ensuring seamless transitions (Section 5.3.2), Git-based container image retrieval for Kubernetes deployment (Section 5.3.3), dynamic services updates triggered by CI/CD pipelines in GitLab repositories (Section 5.3.4), and, at last, the deployment of real systems from the SINCRO project (Section 5.3.5).

5.3.1 Deployment of Web Services on Multiple Cloud Providers

The first use case involved the deployment of the "Hello World" web services mentioned in Section 5.2.4. The cloud providers selected for this study were Microsoft Azure and Google Cloud Platform. The setup of the test environment involved activating the Kubernetes services and creating a cluster within each provider, using default settings.

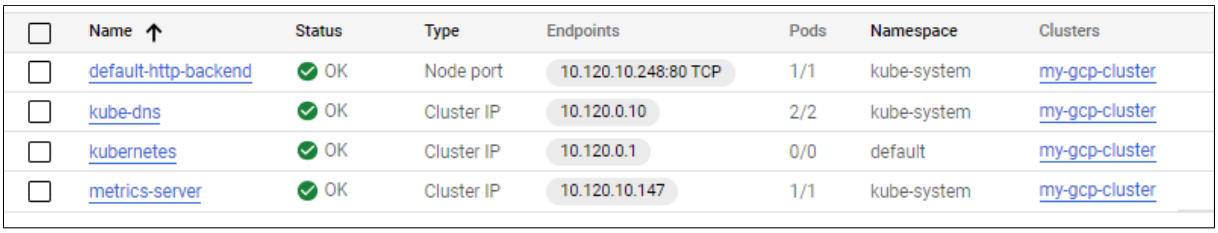
Figure 5.1a and Figure 5.1b display the web interfaces of Microsoft Azure and Google Cloud Platform, respectively, before deploying the service using the implemented prototype. No services, other than system objects, had been deployed at that point. The execution of the CLI application for deploying "Hello World" service in the Kubernetes

cluster hosted by Microsoft Azure is represented in Figure 5.2. The command for this operation aligns with the instructions provided in the first row of Table 4.2, and it generated output content related to deployment metadata. Following the execution, the operation was successfully completed, as suggested by the displayed message. This operation was completed in 4 seconds, given the low complexity and size of the deployed container, and default cluster configurations.



Name	Namespace	Status	Type	Cluster IP	External IP	Ports	Age
kubernetes	default	Ok	ClusterIP	10.0.0.1		443/TCP	15 minutes
kube-dns	kube-system	Ok	ClusterIP	10.0.0.10		53/UDP,53/TCP	15 minutes
metrics-server	kube-system	Ok	ClusterIP	10.0.67.109		443/TCP	15 minutes

(a) Services in Kubernetes cluster hosted on Azure



Name	Status	Type	Endpoints	Pods	Namespace	Clusters
default-http-backend	OK	Node port	10.120.10.248:80 TCP	1/1	kube-system	my-gcp-cluster
kube-dns	OK	Cluster IP	10.120.0.10	2/2	kube-system	my-gcp-cluster
kubernetes	OK	Cluster IP	10.120.0.1	0/0	default	my-gcp-cluster
metrics-server	OK	Cluster IP	10.120.10.147	1/1	kube-system	my-gcp-cluster

(b) Services in Kubernetes cluster hosted on GCP

Figure 5.1: Cloud providers web interface before service deployment

```
C:\Users\Bonacho\eclipse-workspace\tfm\CLI>java -jar target/CLI-0.0.1-jar-with-dependencies.jar deploy
 hello-world-service-1 bonacho17/web-service-1:latest 8081 azure default

Service name: hello-world-service-1
Container image: bonacho17/web-service-1:latest
Port: 8081
Provider: azure
K8s namespace: default
Service successfully deployed! Available at: 20.82.212.80:8081
```

Figure 5.2: Service deployment on Azure, via CLI execution

Finally, Figure 5.3a and Figure 5.3b show the web interfaces of the respective providers after the successful deployment. These figures illustrate the listing of the deployed service through the corresponding Kubernetes Deployment object, which is exposed by the service.

The equivalent test was performed through the developed User Interface. Figure 5.4a displays the web service opened in the editor with Azure selected as the target provider, whereas Figure 5.4b illustrates the equivalent service configured for the GCP provider, with a red frame highlighting this difference. Upon selecting the "Deploy" button located in the left-panel menu, the web services are effectively deployed onto the respective clusters, mirroring the result achieved through the CLI. This outcome aligns

Name	Namespace	Status	Type	Cluster IP	External IP	Ports	Age
kubernetes	default	Ok	ClusterIP	10.0.0.1		443/TCP	36 minutes
kube-dns	kube-system	Ok	ClusterIP	10.0.0.10		53/UDP,53/TCP	36 minutes
metrics-server	kube-system	Ok	ClusterIP	10.0.67.109		443/TCP	36 minutes
hello-world-service-1	default	Ok	LoadBalancer	10.0.1.222	20.82.212.80	8081:30297/TCP	4 minutes

(a) Services in Kubernetes cluster hosted on Azure

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
hello-world-service-1	OK	External load balancer	34.72.210.139:8081	1/1	default	my-gcp-cluster

(b) Services in Kubernetes cluster hosted on GCP

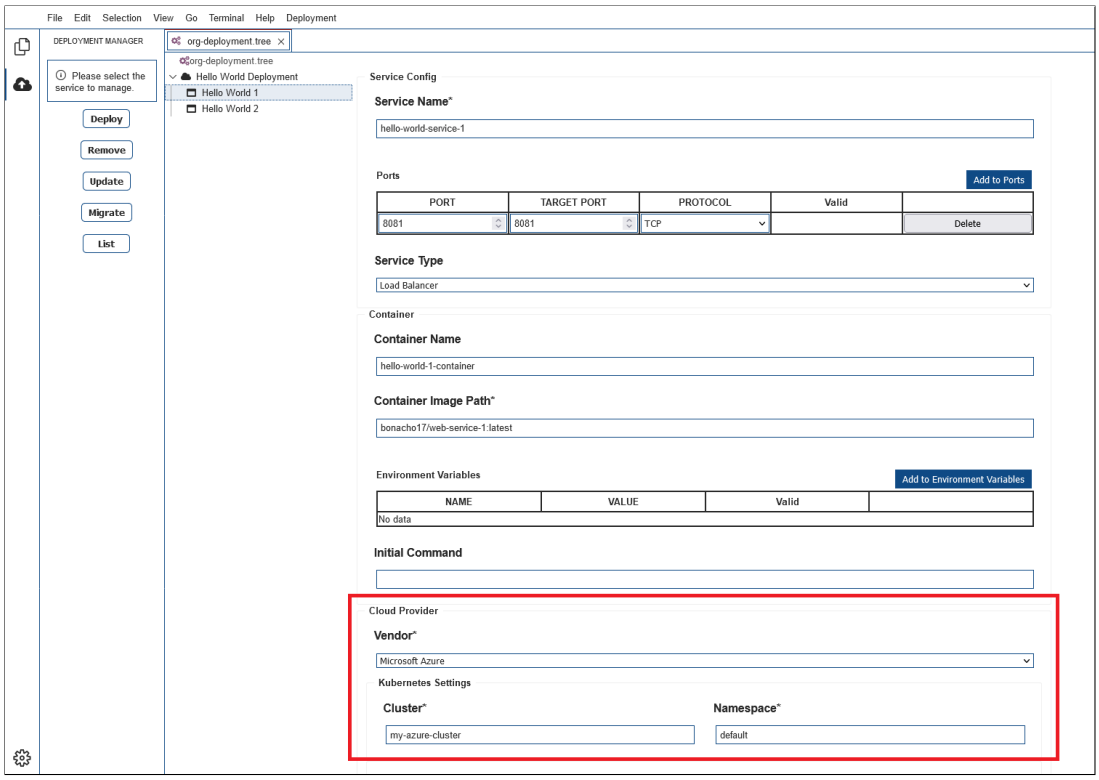
Figure 5.3: Cloud providers web interface after service deployment

with expectations, given that both interfaces employ identical API calls to interact with the platform.

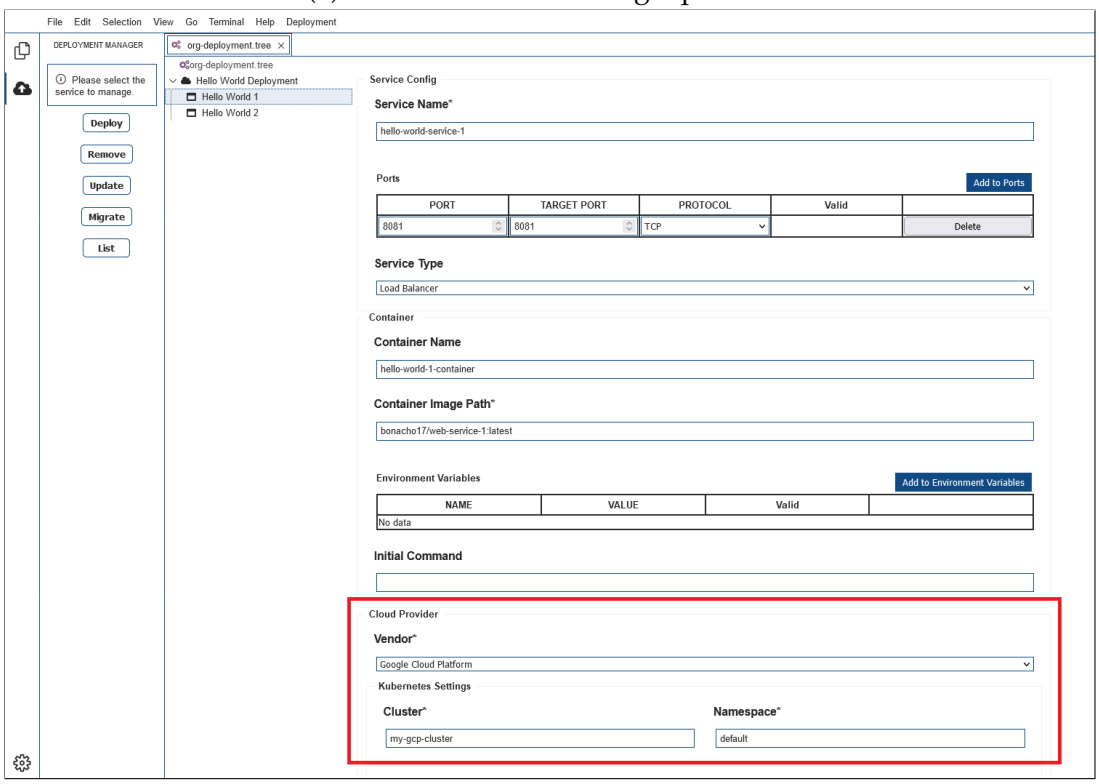
It bears mentioning that this validation was also successfully performed on a local Kubernetes cluster using Minikube [68]. Figure 5.5a figure shows, using the `kubectl` command line tool, that there were no deployments or services deployed in the local cluster before the deployment operation was performed. On the other hand, Figure 5.5b shows the Web Service 1 deployment and the corresponding service after the deployment.

5.3.2 Migration between Cloud Providers

The goal of this use case is to validate the migration of a web service from one provider to another. For demonstration purposes, a local infrastructure via Minikube is used as the source provider, and a cloud platform (e.g., Microsoft Azure) as the target provider. Users or developers using the web application initiate the operation by selecting "Service Migration?" option in the editor, followed by specifying "Target Provider", and corresponding "Target Cluster" and "Target Namespace", according to their needs. The migration process is triggered by clicking on the "Migrate" button, which removes the service from the source provider (Minikube) and instantiates it in the target provider (Azure) within the specified cluster and namespace. The success of the operation is confirmed by the absence of the service in Minikube, and its presence in the Azure's cluster. The ability to access the URL of the service from Azure's cluster, instead of its local access, is also a proof of the use case's accomplishment. As a visual reference, Figure 5.6 illustrates the Deployment file with the Service object used for this scenario, properly populated with the details of the migration target.

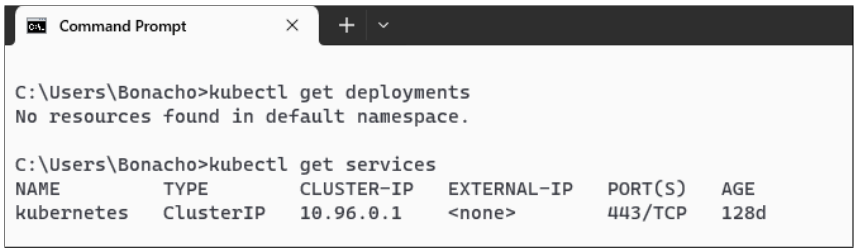


(a) Microsoft Azure as target provider



(b) GCP as target provider

Figure 5.4: Web Interface with services configuration



(a) Deployments and Services before "Web Service 1" deployment

(b) Deployments and Services after "Web Service 1" deployment

Figure 5.5: Command line with Minikube local cluster

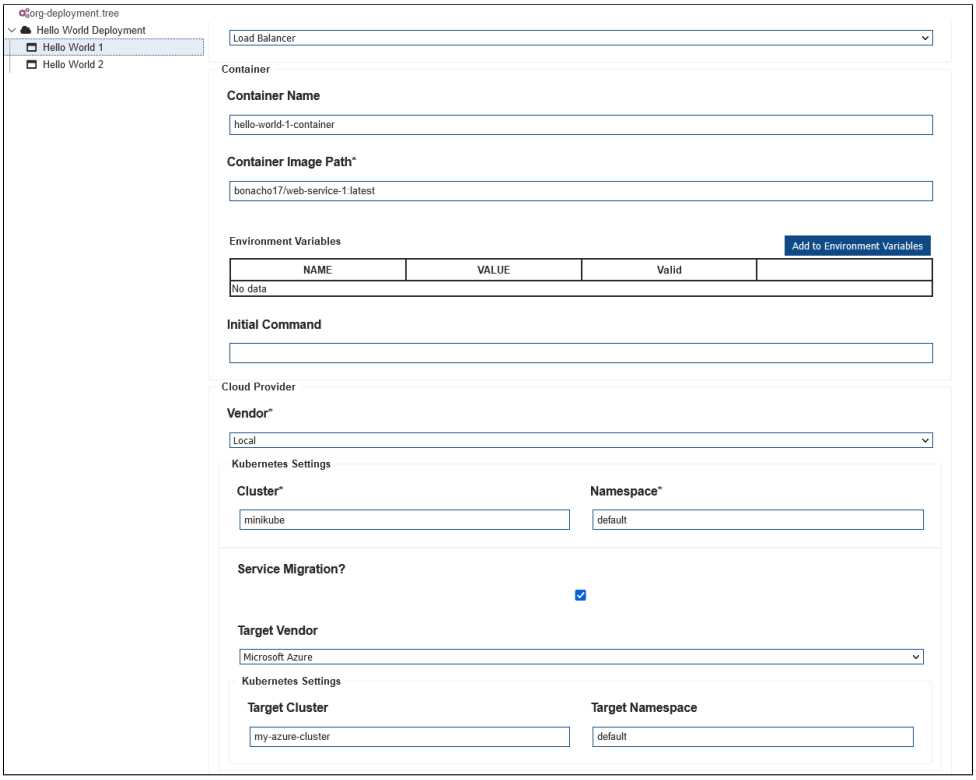


Figure 5.6: Web Interface's editor with service migration details

Overall, this test demonstrates the effectiveness and straightforwardness in web services' migration between different providers, while preserving service functionality and with a reduced downtime, mainly to ensure minimal service disruption.

5.3.3 Git-Based Image Retrieval for Kubernetes

Figure 5.7 shows the GitLab web interface with the container images resulting from the two web services hosted in the repository and registered in GitLab's container registry. Following the procedures detailed in Section 4.3.2.2, namely by inputting the repository's URL (Figure 5.8), its execution results in the automatic creation of the Deployment file and corresponding Service objects, tailored to the images found in the specified repository, populating the form with the container image and name for each service, as displayed in Figure 5.9.

Figure 5.7: Container registry from GitLab repository

Figure 5.8: Input of GitLab repository URL

Subsequently, the user then gains control over the process, allowing him to modify the pre-populated information (if necessary) and complete the rest of the form with the parameters required to successfully deploy the services.

5.3.4 Updating Running Services with CI/CD

This evaluation involves modifying the source code of Web Service 1, e.g., updating the greeting message from "Hello World 1" to "Hello World 1 - ISEL", and committing the changes to the GitLab repository, which triggers the CI/CD pipeline described in

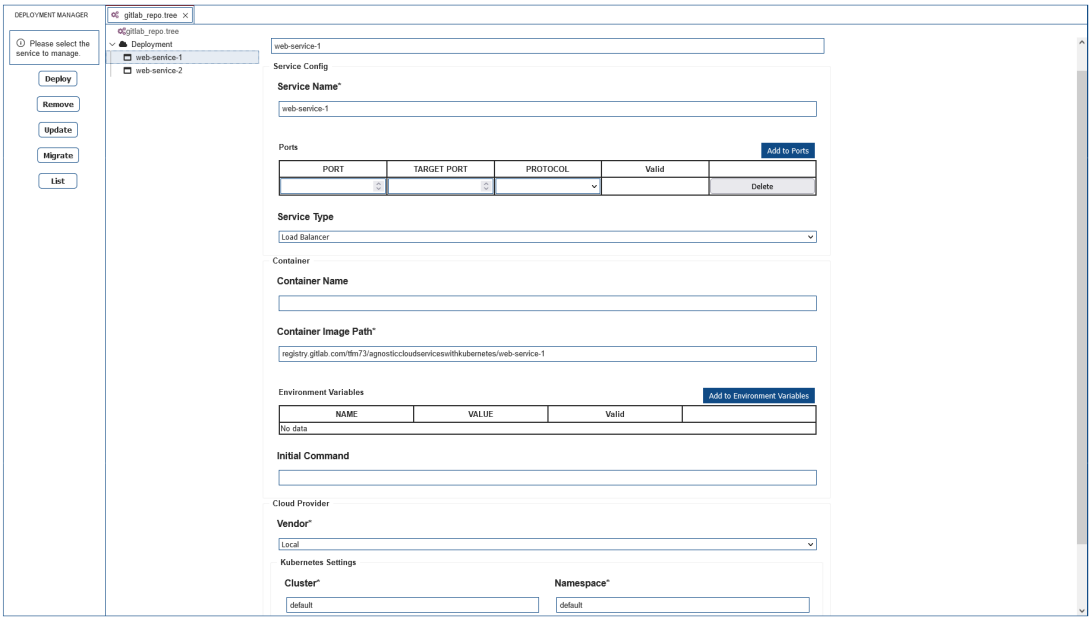
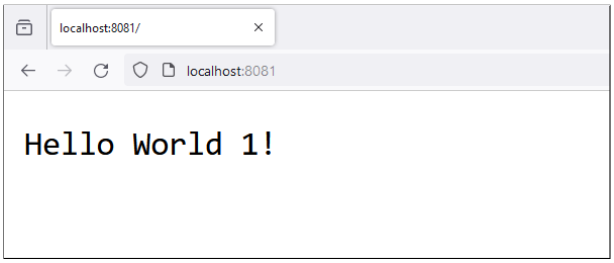


Figure 5.9: Service editor populated with container image details from GitLab repository

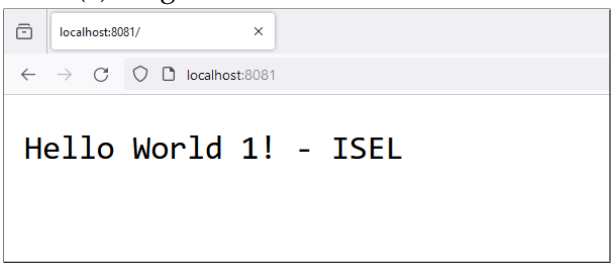
Section 5.2.3. This process results in the container image for the web service being updated. Once the container image has been properly updated, the test consists of opening the corresponding service in the editor and clicking the "Update" button in the service's management menu. Within about 3 seconds, the operation is completed and the web service is successfully updated, as shown in Figure 5.10b, where "Hello World 1 - ISEL" is the new greeting message for web service 1, instead of "Hello World 1", in Figure 5.10a.

5.3.5 Deployment validation for SINCRO Project

This validation use case follows the approach of monitoring roadside traffic equipment, which adopts the ISoS framework. Therefore, the cyber-physical elements of the SINCRO project, namely cabins and cinemometers, are each modeled as Service elements, which compose instances of CES, as briefly described in Section 5.2.5. From a simplified standpoint, it can be assumed that $\texttt{ISystem}_0$ is where the other SINCRO systems and services are registered, and the `ISystemGUI` is responsible for displaying all registered SINCRO elements and related details in a web interface. Moreover, this demonstration considers three cabins to be registered in $\texttt{ISystem}_0$ from two different suppliers, Micotec [67] and Yunex [105], composing `ISystemMicotec` and `ISystemYunex` ISystems, respectively. Micotec has two cabins located on the IC19



(a) Original "Hello World 1" service



(b) Updated "Hello World 1" service

Figure 5.10: Web Services execution in a Web Browser

road, and Yunex has one located on the A1 highway. Figure 5.11 represents these elements using the modeling strategy based on the ISoS model.

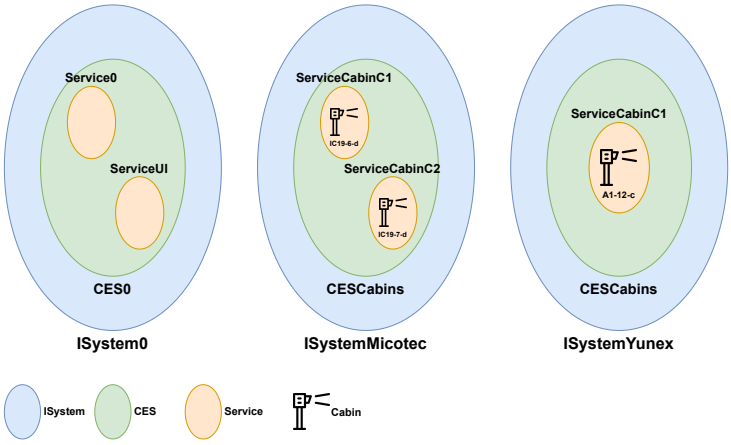


Figure 5.11: SINCRO elements modeled using the ISoS framework

In the scope of this work, it is noteworthy that each Service element is represented by a container image, which allows its instantiation in a Kubernetes cluster using the developed platform. Specifically, in this use case, three ISoS container images are considered: one for $\texttt{ISystem}_0$, another for `ISystemGUI`, and a third for the cabins.

Before delving into the deployment validation process, it should be noted that this validation was performed on a local cluster using Minikube, i.e., the SINCRO environment was all deployed locally, mainly for ease of testing and development.

The first step in this validation is to retrieve the required container images from SINCRO's GitLab repository. This process is accomplished by entering the URL of the repository in the developed platform, shown in Figure 5.12, which results in the generation of three deployment files, one for each service found, as proven in Figure 5.13.

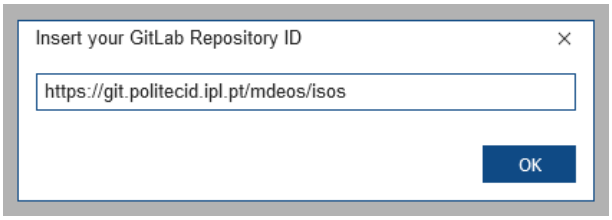


Figure 5.12: Input of SINCRO GitLab URL

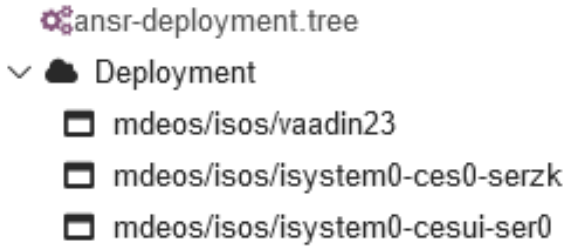


Figure 5.13: Services discovered in SINCRO GitLab

The first service to deploy is the ISoS $\texttt{ISystem}_0$. Using the developed platform, this service can be configured as shown in Figure 5.14, where the container image name is automatically retrieved from the GitLab repository with the SINCRO source code. Then, the developer, using the UI, completes the configuration for the service. It introduces parameters associated with service metadata, including a name for the service, a Kubernetes service type, and a set of port mappings (each consisting of a port, a target port, and a protocol). The container, in addition to specifying the container image, it is configured by providing a name (which can be derived from the container image name), the necessary environment variables (none for this service), and an initial command for the container. As for the provider settings, in this use case, the local cluster "minikube" and the corresponding namespace "default" were selected.

Figure 5.14 ensures that the $\texttt{ISystem}_0$ service is active.

Subsequently, ISoS `ISystemGUI` deployment configuration was performed. The appropriate container image for `ISystemGUI` was also obtained from the specified repository, and the parameters related to specific service configurations, other container specifications, and provider infrastructure settings were manually entered. Notably,

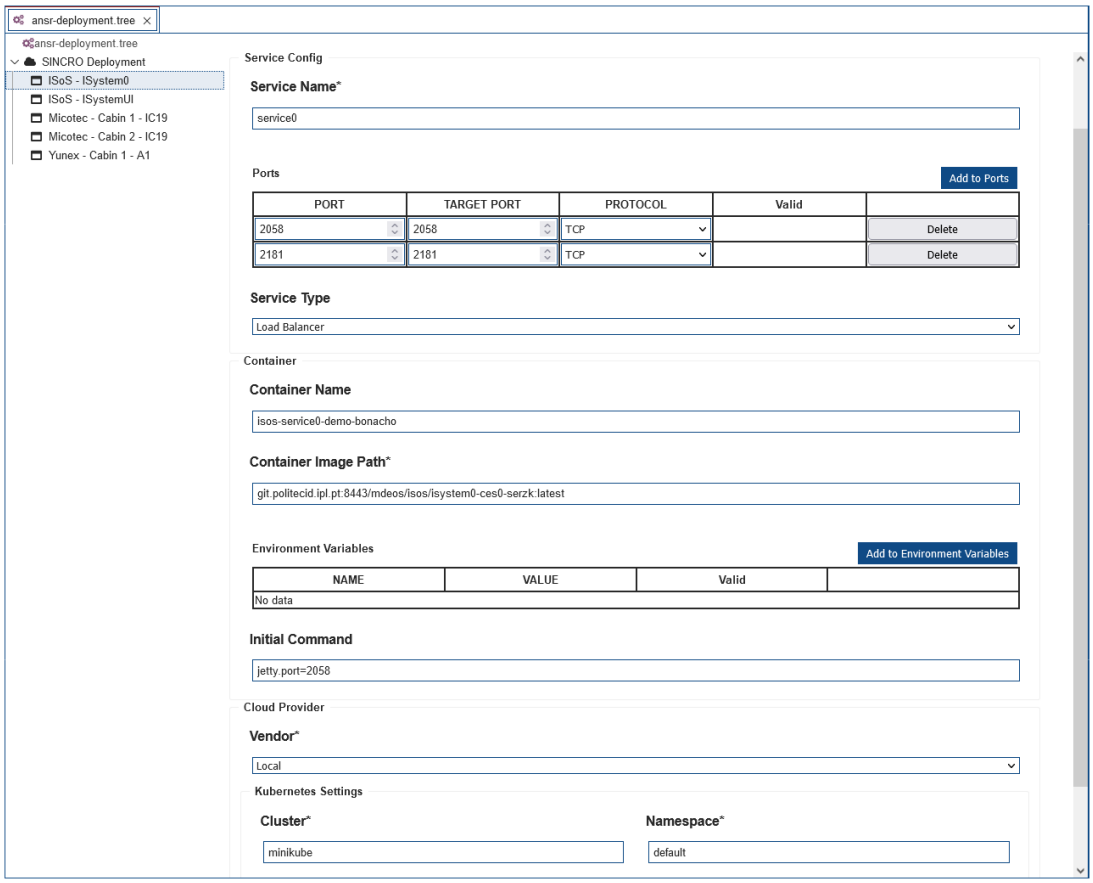


Figure 5.14: SINCRO $\texttt{ISystem}_0$ deployment configuration

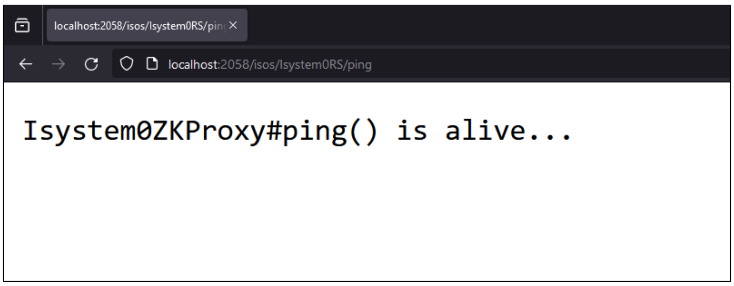


Figure 5.15: $\texttt{ISystem}_0$ liveness probe

this service requires the input of an environment variable to define `ISystem0`'s address, which can simply be pointed to `"service0"`. In Kubernetes networking, by default, the FQDN of any service follows the pattern `<service-name>.<namespace>.svc.cluster.local`. For services in the same namespace, the service name is typically sufficient for access, as Kubernetes is able to set up DNS lookup domains. This configuration is illustrated in Figure 5.16.

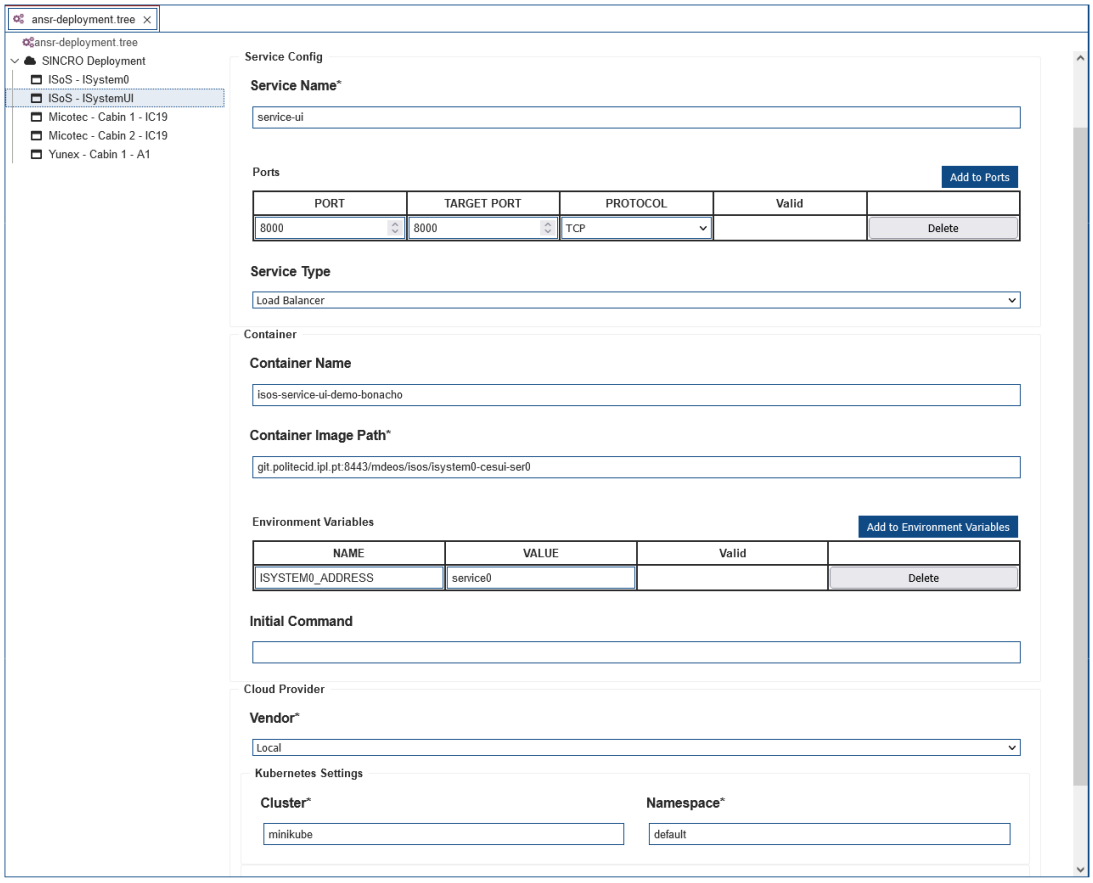


Figure 5.16: SINCRO `ISystemUI` deployment configuration

Figure 5.17 illustrates the success of the deployment operation, as evidenced by the display of the `ISystem0` administration interface.

The identical approach was then employed for deploying the cabins. All three cabins slated for network registration were created from a common container image. The remaining configuration settings required for the cabins vary in terms of service and container names, the port number to be exposed, and the `CABIN_NAME` environment variable. For reference, the deployment configuration for Cabin 1 of IC19, provided by Micotec supplier, can be observed in Figure 5.18.

Figure 5.21 displays all the SINCRO elements considered in this use case deployed locally in a Kubernetes cluster, properly built as Deployment and Service objects.

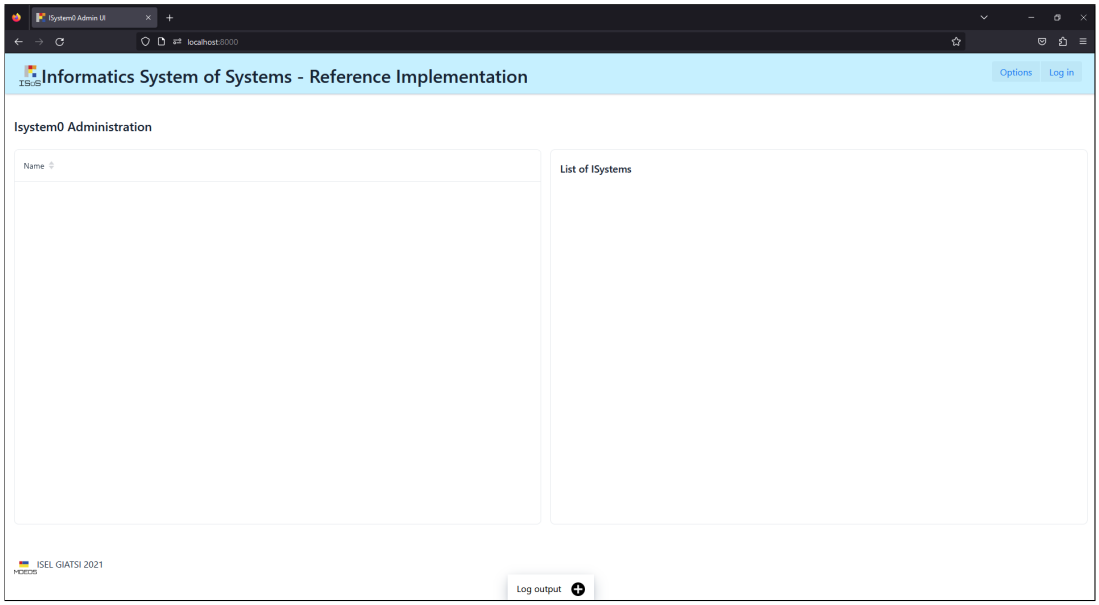


Figure 5.17: SINCRO ISoS UI web page

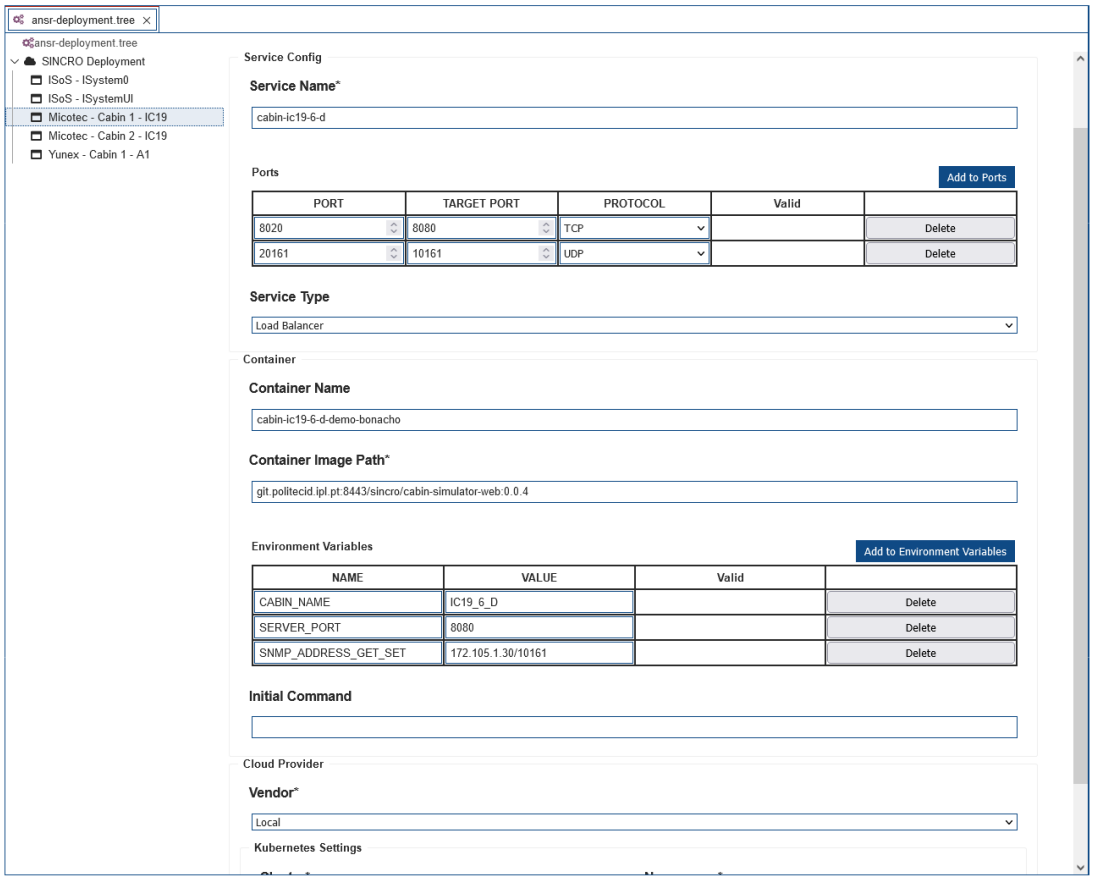


Figure 5.18: Cabin 1 (Micotec IC19) deployment configuration

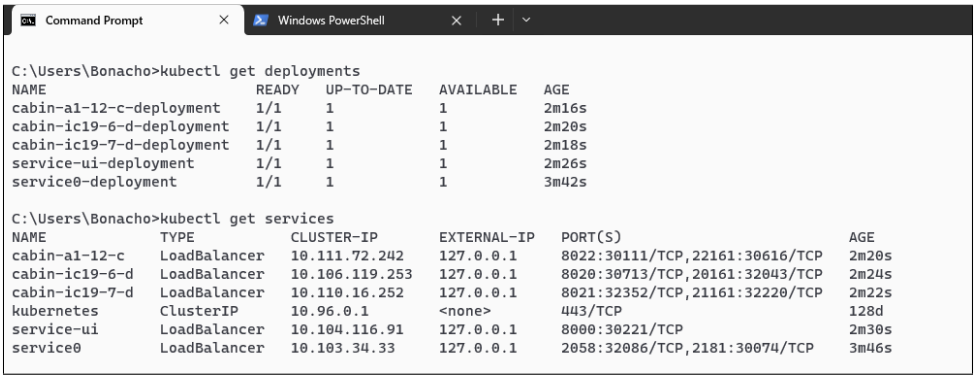


Figure 5.19: SINCRO elements instantiated in the local cluster

In the current development stage of SINCRO, the registration of a new cabin has to be done manually by instantiating a JAR file. Future approaches will allow the cabinet to register the corresponding containers on the network at boot time. Therefore, upon JARs' execution for the cabin of Yunex and Micotec suppliers, the UI presents the corresponding `ISystems`, namely `ISystemYunex` and `ISystemMicotec`, as illustrated in Figure 5.20.

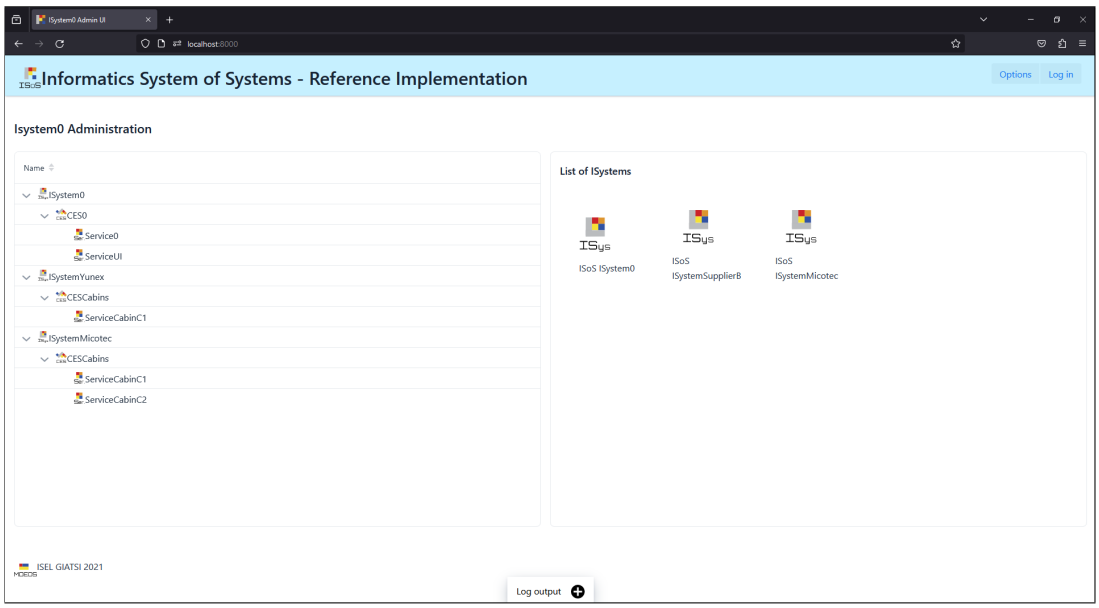


Figure 5.20: SINCRO ISoS UI with the registered cabins from the suppliers (tree view)

Finally, Figure 5.21 presents the synoptic monitoring interface for Cabin 1. In [87], a synoptic interface is defined as a view of technological elements associated with processes that allow an operator to interact with the elements being monitored. In this context, this interface is accessible via the "View Synoptic" button within the $\texttt{ISystem}_0$ administration interface under cabin's details, namely through ISoS `ISystemMicotec/CESCabins/ServiceCabinC1` path.

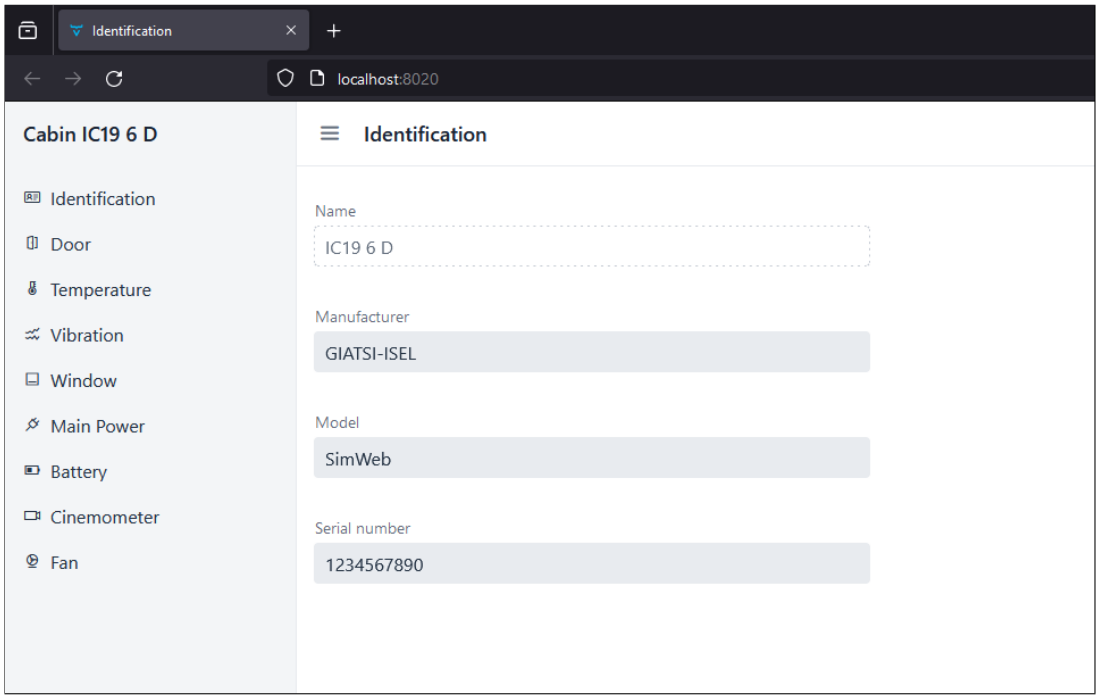


Figure 5.21: Synoptic interface of Cabin 1 (Micotec IC19)

Furthermore, although being beyond the scope of this work, once the services are registered in $\texttt{ISystem}_0$, a "SoT framework can discover the required endpoints to access the services' interfaces, namely the host IP and SNMP ports to obtain monitoring information", as disclosed in [87].

5.4 Summary

In this chapter, the primary objective was to comprehensively validate the developed vendor-agnostic service deployment platform. Guided by the premise of being vendor-agnostic, multiple capabilities and features were assessed through testing across various environments, including leading cloud providers and a local Kubernetes cluster, via Minikube. From this foundation, validation efforts focused on critical use cases, such as seamless web service deployment on distinct infrastructures, successful cloud provider migrations, the retrieval of private container images for Kubernetes deployments, CI/CD-enabled service updates, and the deployment of real systems from the SINCRO project.

The deployment of the SINCRO through the using the SDMAC framework-based platform was a successful demonstration of its ability to deploy more complex systems, while simplifying configuration for both users and developers. By abstracting technical details, it reduces the need for users to understand complex configurations or rely

on proprietary tools (vendor lock-in problem). The study conducted also confirmed the flexibility in choosing service providers, with an option for public cloud providers. In SINCRO's case, the automatic population of the service editor with container image details from Git repositories, also was also a proved of its value in terms of streamlining and accelerating service configuration.

Regarding cluster's basic configurations, it is duly noted that the intention was not to focus on fine-grained cluster customization or performance optimization. This decision was driven by the platform's validation objectives, which prioritize functionality and proof of concept, with a focus on minimizing vendor lock-in that organizations often encounter when migrating applications between providers.

Overall, the results of these validations confirm the platform's robustness and versatility, while recognizing that they are not fully complete and accurate due to the prototyping nature of this contribution. However, they did underscore the platform's usefulness in cloud computing and container orchestration practices by providing organizations with a vendor-agnostic solution for their multi-cloud requirements and needs.

6

Conclusions and Future Work

In this chapter, Section 6.1 presents the main conclusions in terms of findings and accomplishments derived from this work. Additionally, Section 6.2 provides an overview of future work related to this thesis.

6.1 Conclusions

This document presented and discussed the development of the work along this thesis, the Services Deployment and Management Agnostic-Cloud (SDMAC) framework. It included an introduction to the problem, a review of the state of the art and related work, the formulation of a model architecture to address the problem, and the implementation and validation of a corresponding platform's prototype.

The discussion outlined how containers provide flexibility and scalability for service deployment, and how container orchestrators, namely Kubernetes, are pivotal for managing containerized applications. An overview of the challenges of deploying and managing containers, especially in terms of vendor-specific configurations, was also disclosed. Furthermore, similar work was described, focused on automation and management capabilities in cloud infrastructures.

It is argued that by adopting the similar platforms mentioned in Section 2.6.1, organizations remain dependent on their services and specifications, which can difficult the migration of applications between platforms. Each platform requires the writing

of configuration files with its proprietary syntax and commands, reinforcing this dependency as organizations strive to integrate and utilize its features. This work was designed to face this problem, as it aims to establish an open standard for an agnostic cloud. The objective of deploying services, through containers, on different IT infrastructures was successfully achieved, freeing organizations from worrying about configurations and specific details of cloud platforms. Hence, this minimizes and mitigates the dependency on any vendor, infrastructure or platform, which effectively addresses vendor lock-in challenges. Therefore, the research conducted to date indicates that this offering adds value to the cloud deployment landscape and differentiates itself from existing platforms and tools.

The deployment of the SINCRO network, using the platform based on the SDMAC framework, allowed the validation of the work in a more comprehensive and concrete way. This was achieved by demonstrating the ability to deploy a real system of considerable complexity, confirming its correct functioning and operability, while complying with all the requirements it imposes.

The main conclusions of this study indicate that the proposed framework simplifies the deployment of systems and services for users and developers. The framework makes it easier and more practical to configure services, by hiding specific technical details from both the service provider and the underlying platform, namely Kubernetes. This transparency eliminates the need for users to understand complex configurations or rely on proprietary tools to leverage third-party services and offerings. Developers can focus on their informatics systems, their intricacies and business rules, rather than getting entangled in the complexities of underlying solutions, whether proprietary or open source (be it cloud service providers, or container orchestration platforms). Moreover, it reduces the ongoing challenges of adapting to different vendors, mitigating the vendor lock-in problem.

Users are expected to possess fundamental knowledge and concepts of Kubernetes clusters and networking concepts. While this knowledge is considered a prerequisite and a Kubernetes dependency, it can be justified by the fact that it can serve as a foundation to fully exploit the potential of Kubernetes clustering capabilities for services. Regardless, the platform strives to maximize the abstraction layer over the Kubernetes network, thus avoiding additional dependencies for the user as much as possible.

A pivotal observation from the validation conducted concerns the flexibility in choosing the service provider for the instantiation and execution of services. In SINCRO's case, a local infrastructure was utilized, but other use cases presented in Chapter 5 demonstrate that a public cloud provider could easily have been employed instead.

Such requirement only demands setting up a cluster on the chosen provider and supplying the platform with the appropriate `kubeconfig` file and credentials. Additionally, another efficient feature to highlight is the automatic population of the platform's service editor with container image details, when the image stored in a Git repository (currently with GitLab support only), which expedites service configuration and minimizes potential human errors (e.g., typographical errors). Future work includes expanding the auto-population of data in the editor.

Although Kubernetes clusters may pose initial challenges and require a significant learning curve, their use within the developed prototype revealed their significantly compelling advantages. In summary, these clusters are able to empower organizations with a scalable, self-healing, and portable infrastructure for their containerized services, while simplifying service discovery and load balancing. Fundamentally, their capabilities alongside with providing the freedom to deploy services across diverse environments, makes Kubernetes an essential tool in the dynamic landscape of cloud environments and container orchestration.

Finally, the Kubernetes Cloud Controller Manager emerges as a key element in the quest for cloud agnosticism. Beyond from abstracting cloud-specific complexities, this mechanism enables unified management and seamless workload migration. Ultimately, its capabilities and characteristics empower the framework (and organizations) to leverage the capabilities of multiple cloud providers without incurring in vendor lock-in. Therefore, it is thought that Kubernetes and CCM represent an invaluable asset in current cloud service deployment and management strategies.

6.2 Future Work

The developed prototype for the SDMAC-based platform was designed and built to demonstrate the viability of an approach that addresses the challenge of vendor lock-in within cloud environments. Nevertheless, it is imperative to acknowledge the developmental nature of this prototype, as it aims to serve as a proof of concept rather than a finalized product. While successfully demonstrating the potential for mitigating vendor lock-in through Kubernetes and its Cloud Controller Manager component, the prototype prioritizes functionality over user interface and comprehensiveness.

Moving forward, future work involves platform's refinement and expansion. The upcoming stages of development include, but are not limited to, the following areas of focus:

- **Login mechanism for the platform.** Enabling platform's login is primarily for security and user convenience. This mechanism could include multiple authentication methods, such as OAuth 2.0, traditional username/password options, etc., for flexibility purposes. By securely managing user credentials, it would allow users to connect their cloud accounts and access resources with the developed platform, mostly for accessing private image repositories and integration with cloud providers. This strategy eliminates manual input of resources (e.g., kubernetes clusters/namespaces), by allowing the selection of these existing resources from a given provider associated with the logged-in user, thus improving user experience and efficiency. Ultimately, this platform could work as a plugin to be embedded into an Integrated Development Environment (IDE), which facilitates the access to developers' resources and settings;

- **Build Kubernetes clusters programmatically on cloud providers.** The automated instantiation of clusters allows users to define cluster's requirements and preferences through the platform's interfaces. Clusters' provisioning involves exploring cloud provider-specific APIs and Software Development Kits (SDKs), and evaluating IaC tools, such as Terraform and CloudFormation for declarative provisioning approaches. The goal of this feature is to create an abstraction layer that simplifies the cloud-specific process of provisioning K8s clusters, further reducing vendor lock-in, while completing platform's capabilities;

- **Generate container images programmatically, without Dockerfiles.** This functionality allows users to define service requirements and dependencies, from which the platform can generate and register container images to be used on future deployments. Research involves tools and libraries for building container images, including BuildKit [10] or Kaniko [51], that can generate container images without relying on Dockerfiles. Alternatively, Quarkus extension `quarkus-container-image-jib`, powered by Jib [50], can be also used to easily and efficiently build container images, while eliminating the need for Docker or daemon processes [80]. This effort aims to simplify and accelerate, once more, as much as possible, the deployment process for organizations;

- **Extended Service Management API.** The expansion of Services Management API capabilities supports dynamic updates of service running settings. The goal is to have a broader control over the dynamic management of service configurations, which could include scaling policies, resource allocation based on workload requirements, user-defined policies, etc.. In particular, this will require an in-depth study of the Kubernetes API and its extensions;

- **Enable the deployment of other types of services.** Analyze other types of services, e.g., serverless functions, or databases, to determine if and how they can be effectively deployed and managed by the platform. Enabling this feature could involve developing deployment templates and configuration options tailored to different service types, and possibly integration with other frameworks or even databases;

- **Error control in clusters' connection.** Implementation of mechanisms to handle common errors and challenges typically found in the communication and management of K8s clusters. This would include reviewing Kubernetes' native error handling mechanisms, and studying automated error recovery strategies (e.g., rolling back to a stable cluster state in the event case of failures). In this regard, it is dully noted that refined error control in clusters' connection and communication is a critical element in ensuring the reliability and widespread adoption of the platform;

- **Input validation and error handling.** Ensuring platform's reliability, security and stability is of paramount importance. Consequently, the development of enhanced input validation and error handling mechanisms must be considered. For that purpose, it must prioritize thorough input validation practices across the platform to prevent security issues, such as injection attacks and other known attacks, and the integration of security tools into the development pipeline to improve security.

These tasks represent the evolution of the prototype, expanding its functionalities and improving its performance and security, in order to strengthen the competitiveness among similar approaches. Such efforts aspire to transform the prototype into a robust and versatile platform, capable of meeting the demands of industrial cloud deployments and further diminishing vendor dependency for organizations operating in this ever-evolving landscape.

References

[1] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, and Dan Mackin, *UNIX and Linux System Administration Handbook (5th Edition)*, 5th. Addison-Wesley Professional, 2017, ISBN: 0134277554.

[2] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi, "Cloud container technologies: A state-of-the-art review", *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pages 677–692, 2019. DOI: 10.1109/TCC.2017.2702586.

[3] David Elliott, Carlos Otero, Matthew Ridley, and Xavier Merino, "A cloud-agnostic container orchestrator for improving interoperability", in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pages 958–961. DOI: 10.1109/CLOUD.2018.00145.

[4] Anshita Malviya and Rajendra Kumar Dwivedi, "A comparative analysis of container orchestration tools in cloud computing", in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2022, pages 698–703. DOI: 10.23919/INDIACom54597.2022.9763171.

[5] *About the open container initiative*. [Online]. Available: https://opencontainers.org/about/overview/.

[6] *Hybrid cloud management with anthos | google cloud*. [Online]. Available: https://cloud.google.com/anthos.

[7] *Apache mesos*. [Online]. Available: https://mesos.apache.org/.

[8] *What is terraform: Automate changes*. [Online]. Available: https://developer.hashicorp.com/terraform/intro#automate-changes.

[9] *Blueprint files and packages*. [Online]. Available: https://docs.cloudify.co/latest/developer/blueprints.

[10] *Buildkit | docker docs*. [Online]. Available: https://docs.docker.com/build/buildkit/.

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

[12] Scott Carey, *What is docker? the spark for the container revolution*, 2021. [Online]. Available: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html.

[13] Emiliano Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges", in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS'16, Taormina, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017, 287–290, ISBN: 9781631901416. DOI: 10.4108/eai.25-10-2016.2266649. [Online]. Available: https://doi.org/10.4108/eai.25-10-2016.2266649.

[14] Emiliano Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges.", in *VALUETOOLS*, 2016.

[15] *Cloud computing layers*. [Online]. Available: https://docs.oracle.com/en-us/iaas/Content/cloud-adoption-framework/cloud-computing-layers.htm.

[16] *Github & gitlab ci/cd*. [Online]. Available: https://opencoursehub.cs.sfu.ca/bfraser/grav-cms/cmpt373/links/files/GitLab_CICD_Guide.pdf.

[17] *Ci/cd with containers - dzone refcardz*. [Online]. Available: https://dzone.com/refcardz/cicd-with-containers.

[18] *Technology topics: Devops: What is ci/cd?* [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[19] Molly Clancy, *Docker containers vs. vms: A look at the pros and cons*, 2022. [Online]. Available: https://www.backblaze.com/blog/vm-vs-containers/.

[20] *Provision infrastructure as code - aws cloudformation - aws*. [Online]. Available: https://aws.amazon.com/cloudformation.

[21] *What is cloudify? | cloudify documentation center*. [Online]. Available: https://docs.cloudify.co/4.3.0/about/introduction/what-is-cloudify.

[22] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou, "A comparative study of containers and virtual machines in big data environment", Jul. 2018, pages 178–185. DOI: 10.1109/CLOUD.2018.00030.

[23] Anuj Yadav, Lavleesh Garg, and Ritika Mehra, "Docker containers versus virtual machine-based virtualization: Proceedings of iemis 2018, volume 3", in Jan. 2019, pages 141–150, ISBN: 978-981-13-1500-8. DOI: 10.1007/978-981-13-1501-5_12.

[24] *Container runtime interface (cri): Past, present, and future*. [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-security/container-runtime-interface/.

[25] *The definitive platform for modern apps*. [Online]. Available: https://dcos.io/.

[26] Leonardo Rebouças De Carvalho and Aleteia Patricia Favacho de Araujo, "Performance comparison of terraform and cloudify as multicloud orchestrators", in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, IEEE, 2020, pages 380–389.

[27] Inc. Razorops, *Kubernetes-the de facto standard to deploy and operate containerized applications*. [Online]. Available: https://www.linkedin.com/pulse/kubernetes-the-de-facto-standard-deploy-operate-containerized-.

[28] Tinankoria Diaby and Babak Bashari Rad, "Cloud computing: A review of the concepts and deployment models", *International Journal of Information Technology and Computer Science*, vol. 9, no. 6, pages 50–58, 2017.

[29] Opencontainers, *Oci distribution specification*. [Online]. Available: https://github.com/opencontainers/distribution-spec.

[30] *Docker: Accelerated container application development*. [Online]. Available: https://www.docker.com/.

[31] *How services work*, 2023. [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/.

[32] *Swarm mode key concepts*, 2023. [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/#what-is-a-swarm.

[33] *Swarm mode key concepts - load-balancing*, 2023. [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/#load-balancing.

[34] *Swarm mode key concepts - nodes*, 2023. [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/#nodes.

[35] *Swarm mode key concepts - services and tasks*, 2023. [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/#services-and-tasks.

[36] 2023 Ajeet Singh Raina February 1, 2023 Guillaume Lours January 31, and 2023 Amy Bass January 25, *Demystifying the open container initiative (oci) specifications*, 2022. [Online]. Available: https://www.docker.com/blog/demystifying-open-container-initiative-oci-specifications/.

[37] *Fabric8: Open source integrated development platform for kubernetes*. [Online]. Available: https://fabric8.io.

[38] *Google kubernetes engine (gke) | google cloud*. [Online]. Available: https://cloud.google.com/kubernetes-engine.

[39] Carlos Gonçalves, A Luís Osório, Luís M Camarinha-Matos, Tiago Dias, and José Tavares, "A collaborative cyber-physical microservices platform–the sitl-iot case", in *Working Conference on Virtual Enterprises*, Springer, 2021, pages 411–420.

[40] Carlos Gonçalves, Tiago Dias, A Luís Osório, and Luis M Camarinha-Matos, "A multi-supplier collaborative monitoring framework for informatics system of systems", in *Working Conference on Virtual Enterprises*, Springer, 2022, pages 44–53.

[41] *Heat - openstack*. [Online]. Available: https://wiki.openstack.org/wiki/Heat.

[42] *What is terraform: Terraform: Hashicorp developer: How does terraform work*. [Online]. Available: https://developer.hashicorp.com/terraform/intro#how-does-terraform-work.

[43] "Container technology", in *Cloud Computing Technology*. Singapore: Springer Nature Singapore, 2023, pages 295–342, ISBN: 978-981-19-3026-3. DOI: 10.1007/978-981-19-3026-3_7. [Online]. Available: https://doi.org/10.1007/978-981-19-3026-3_7.

[44] *What is infrastructure as code (iac)?* [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

[45] *What is virtualization?* [Online]. Available: https://www.ibm.com/topics/virtualization.

[46] *Infrastructure as code*. [Online]. Available: https://www.ibm.com/topics/infrastructure-as-code.

[47] Bilgin Ibryam and Roland Huß, *Kubernetes patterns*, 2019.

[48] Opencontainers, *Oci image format*. [Online]. Available: https://github.com/opencontainers/image-spec.

[49] A.L. Freixo Guedes Osório, "Collaborative networks as open informatics system of systems (isos)", Available at https://hdl.handle.net/11245.1/233a4628-282e-4650-9ffb-b3d7b3187e2a, PhD Thesis, University of Amsterdam, Amsterdam, NL, 2020.

[50] GoogleContainerTools, *Googlecontainertools/jib: Build container images for your java applications.* [Online]. Available: https://github.com/GoogleContainerTools/jib.

[51] *Googlecontainertools/kaniko: Build container images in kubernetes*. [Online]. Available: https://github.com/GoogleContainerTools/kaniko.

[52] *Organizing cluster access using kubeconfig files | kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/.

[53] *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/overview.

[54] *Kubernetes components*, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[55] *Container runtime interface (cri)*, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cri/.

[56] *Github - fabric8io/kubernetes-client: Java client for kubernetes & openshift*. [Online]. Available: https://github.com/fabric8io/kubernetes-client.

[57] *Overview*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/.

[58] Lars Larsson, *The unexploited opportunity: Cloud agnostic managed kubernetes*, 2021. [Online]. Available: https://elastisys.com/cloud-agnostic-third-party-managed-kubernetes-services/.

[59] Clive Longbottom, *Containers bring cloud-agnostic workloads closer to reality: Techtarget*, 2020. [Online]. Available: https://www.techtarget.com/searchitoperations/feature/Why-cloud-agnostic-is-a-lofty-goal-for-multi-cloud-users.

[60] Isam al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, and Amedeo Palopoli, "Container orchestration engines: A thorough functional and performance comparison", May 2019, pages 1–6. DOI: 10.1109/ICC.2019.8762053.

[61] *What is terraform: Manage any infrastructure*. [Online]. Available: https://developer.hashicorp.com/terraform/intro#manage-any-infrastructure.

[62] *Marathon overview*. [Online]. Available: https://mesosphere.github.io/marathon/#overview.

[63] *Marathon orchestrates both apps and frameworks*. [Online]. Available: https://mesosphere.github.io/marathon/#marathon-orchestrates-both-apps-and-frameworks.

[64] *Marathon features*. [Online]. Available: https://mesosphere.github.io/marathon/#dcos-features.

[65] *Market study on cloud services*, 2022.

[66] Peter Mell, Tim Grance, *et al.*, "The nist definition of cloud computing", 2011.

[67] *Home - micotec*. [Online]. Available: https://micotec.pt/.

[68] *Minikube start | minikube*. [Online]. Available: https://minikube.sigs.k8s.io/docs/start/.

[69] Justice Opara-Martins, Reza Sahandi, and Feng Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective", *Journal of Cloud Computing*, vol. 5, no. 1, pages 1–18, 2016.

[70] *Open source cloud computing infrastructure - openstack*. [Online]. Available: https://www.openstack.org.

[71] A Luis Osório, Luis M Camarinha-Matos, Adam Belloum, and Hamideh Afsarmanesh, "Collaborative trusted digital services for citizens", in *Working Conference on Virtual Enterprises*, Springer, 2021, pages 212–223.

[72] A Luis Osório, Luis M Camarinha-Matos, Tiago Dias, Carlos Gonçalves, and José Tavares, "Open and collaborative micro services in digital transformation", in *Working Conference on Virtual Enterprises*, Springer, 2021, pages 393–402.

[73] A Luís Osório, Cláudia Antunes, Luis M Camarinha-Matos, and Carlos Gonçalves, "Collaborative management of traffic accidents data for social impact analytics", in *Working Conference on Virtual Enterprises*, Springer, 2022, pages 230–241.

[74] Hiral B Patel and Nirali Kansara, "Cloud computing deployment models: A comparative study", *International Journal of Innovative Research in Computer Science & Technology (IJIRCST)*, 2021.

[75] Jenna Phipps, *What is a software platform?: Webopedia definition*, 2021. [Online]. Available: https://www.webopedia.com/definitions/software-platform/.

[76] *The best free and open source container tools*. [Online]. Available: https://podman.io/.

[77] *Pull an image from a private registry | kubernetes*, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/.

[78] Łukasz Staśko, *How to pull docker images from gitlab container registry in kubernetes cluster*, 2020. [Online]. Available: https://medium.com/@stasko.lukasz/how-to-pull-docker-images-from-gitlab-container-registry-in-kubernetes-cluster-aae52787074a.

[79] *Quarkus - kubernetes extension*. [Online]. Available: https://quarkus.io/guides/deploying-to-kubernetes.

[80] *Container image extensions*. [Online]. Available: https://quarkus.io/guides/container-image#jib.

[81] *Bonacho17/agnosticcloudservices: Isel - trabalho final de mestrado (meic) 2022/23*. [Online]. Available: https://github.com/Bonacho17/AgnosticCloudServices.

[82] Opencontainers, *Oci runtime specification*. [Online]. Available: https://github.com/opencontainers/runtime-spec.

[83] Recommended Links, *The world works with servicenow™*. [Online]. Available: https://www.servicenow.com/.

[84] Nati Shalom, *Servicenow for devops engineering automation*, 2022. [Online]. Available: https://cloudify.co/blog/service-now-for-devops-engineering/.

[85] Bianca Sjoerdstra, "Dealing with vendor lock-in", B.S. thesis, University of Twente, 2016.

[86] *Systems of systems*. [Online]. Available: https://rs.ieee.org/technical-activities/technical-committees/systems-of-systems.html.

[87] Bruno Serras, Carlos Gonçalves, Tiago Dias, and A. Luís Osório, "Monitoring roadside traffic enforcement equipment within sot and isos frameworks", in *2023 7th International Young Engineers Forum (YEF-ECE)*, 2023, pages 14–19. DOI: 10.1109/YEF-ECE58420.2023.10209305.

[88] *What is terraform: Standardize configurations*. [Online]. Available: https://developer.hashicorp.com/terraform/intro#standardize-configurations.

[89] *What is infrastructure as code with terraform?: Standardize your deployment workflow*. [Online]. Available: https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code#standardize-your-deployment-workflow.

[90] *State: Terraform: Hashicorp developer*. [Online]. Available: https://developer.hashicorp.com/terraform/language/state.

[91] *Terraform | hashicorp developer*. [Online]. Available: https://developer.hashicorp.com/terraform.

[92] *Terraform registry*. [Online]. Available: https://registry.terraform.io/browse/providers.

[93] *Theia - cloud and desktop ide platform*. [Online]. Available: https://theia-ide.org/.

[94] *Eclipse-theia/generator-theia-extension: A yeoman generator for extensions to the theia ide*. [Online]. Available: https://github.com/eclipse-theia/generator-theia-extension.

[95] *Generator-theia-extension/templates/tree-editor at master · eclipse-theia/generator-theia-extension*. [Online]. Available: https://github.com/eclipse-theia/generator-theia-extension/tree/master/templates/tree-editor.

[96] *Oasis open*. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[97] *What is terraform: Track your infrastructure*. [Online]. Available: https://developer.hashicorp.com/terraform/intro#track-your-infrastructure.

[98] *Typescript: Javascript with syntax for types*. [Online]. Available: https://www.typescriptlang.org/.

[99] Justice Opara-Martins, R. Sahandi, and Feng Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing", Nov. 2014. DOI: 10.1109/i-Society.2014.7009018.

[100] *Why use containers vs. vms?: Vmware glossary*, 2023. [Online]. Available: https://www.vmware.com/topics/glossary/content/vms-vs-containers.html.

[101] CNCF Serverless WG, *Cncf wg-serverless whitepaper v1. 0*, 2018.

[102] *What is cloudify?* [Online]. Available: https://docs.cloudify.co/4.3.0/about/introduction/what-is-cloudify/.

[103] *What is terraform: Terraform: Hashicorp developer*. [Online]. Available: https://developer.hashicorp.com/terraform/intro.

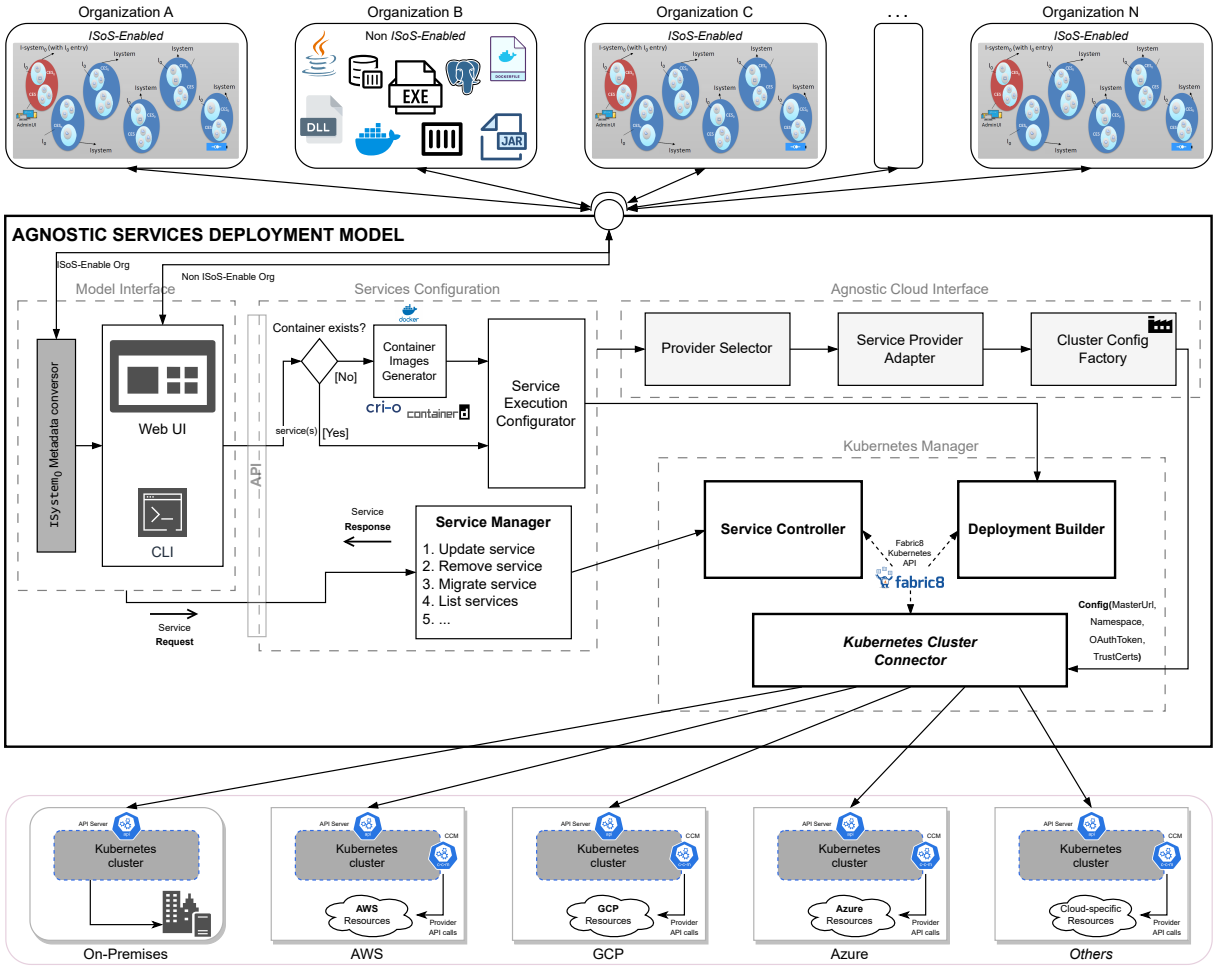
[104] *The web's scaffolding tool for modern webapps | yeoman*. [Online]. Available: https://yeoman.io/.

[105] *Yunex traffic - uniting what's next in traffic.* [Online]. Available: https://www.yunextraffic.com/.

[106] *Zabbix :: The enterprise-class open source network monitoring solution*. [Online]. Available: https://www.zabbix.com/.

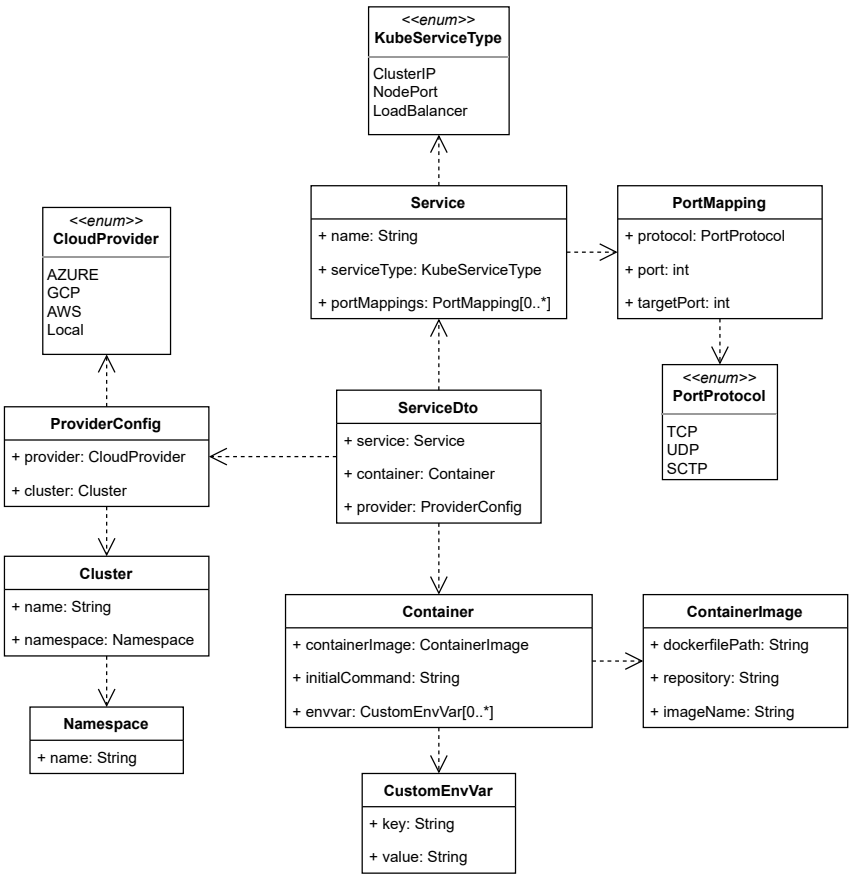
[107] *Welcome to apache zookeeper™*. [Online]. Available: https://zookeeper.apache.org/.

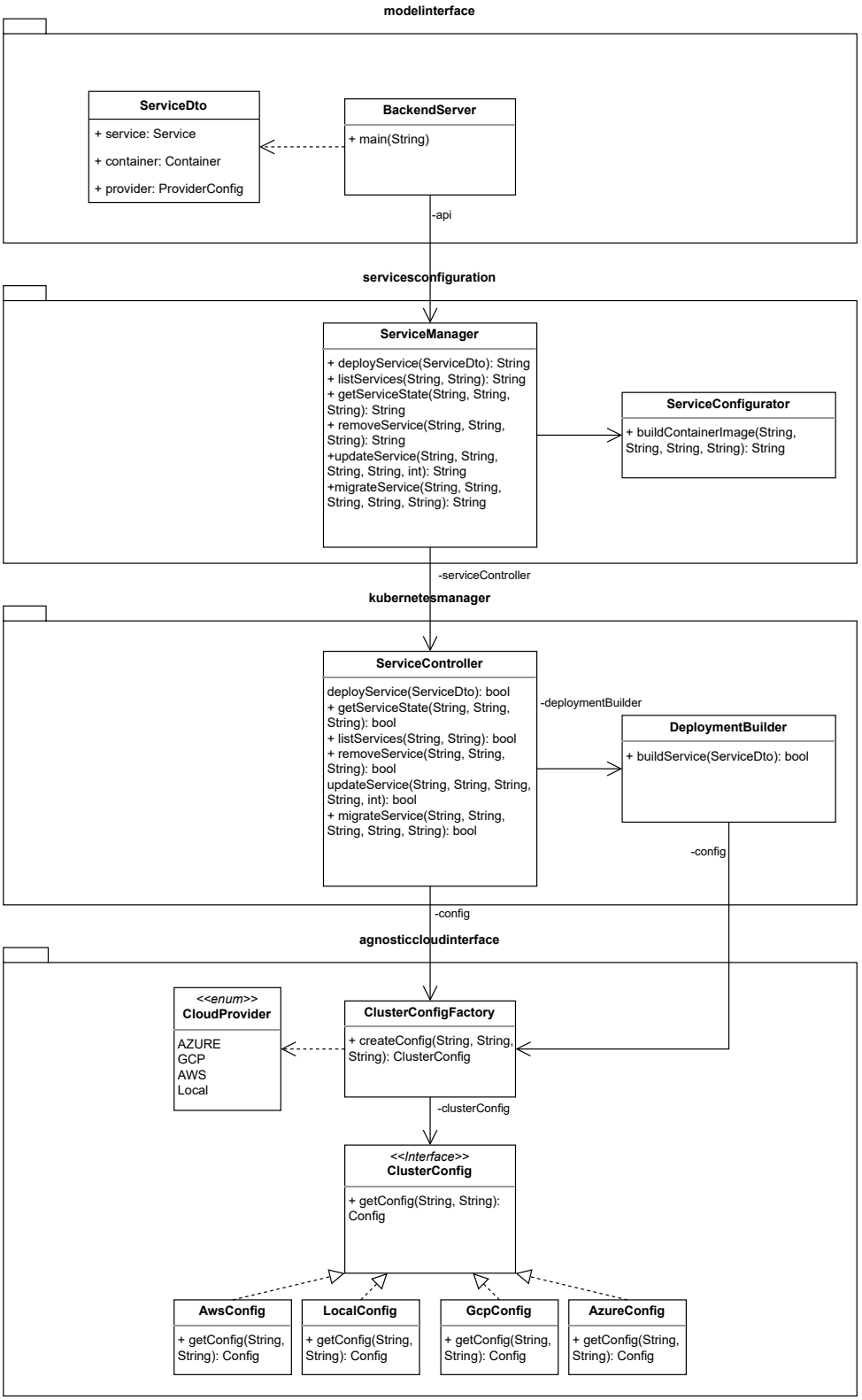
System Architecture


Organization A
ISoS-Enabled
Organization B
Non ISoS-Enabled
EXE
DLL
JAR
Organization C
ISoS-Enabled
...
Organization N
ISoS-Enabled
AGNOSTIC SERVICES DEPLOYMENT MODEL
ISoS-Enable Org
Non ISoS-Enable Org
Model Interface
Services Configuration
Agnostic Cloud Interface
ISystem0 Metadata conversor
Web UI
CLI
API
Container exists?
[No]
[Yes]
service(s)
Container Images Generator
cri-o
container
Service Execution Configurator
Provider Selector
Service Provider Adapter
Cluster Config Factory
Kubernetes Manager
Service Response
Service Manager
1. Update service
2. Remove service
3. Migrate service
4. List services
5. ...
Service Request
Service Controller
Fabric8 Kubernetes API
fabric8
Deployment Builder
Kubernetes Cluster Connector
Config(MasterUrl, Namespace, OAuthToken, TrustCerts)
Kubernetes cluster
API Server
CCM
AWS Resources
GCP Resources
Azure Resources
Cloud-specific Resources
Provider API calls
On-Premises
AWS
GCP
Azure
Others
IT Infrastructure

B

UML Class Diagram




modelinterface
ServiceDto
+ service: Service
+ container: Container
+ provider: ProviderConfig
BackendServer
+ main(String)
-api
servicesconfiguration
ServiceManager
+ deployService(ServiceDto): String
+ listServices(String, String): String
+ getServiceState(String, String, String): String
+ removeService(String, String, String): String
+updateService(String, String, String, String, int): String
+migrateService(String, String, String, String, String): String
ServiceConfigurator
+ buildContainerImage(String, String, String, String): String
-serviceController
kubernetesmanager
ServiceController
deployService(ServiceDto): bool
+ getServiceState(String, String, String): bool
+ listServices(String, String): bool
+ removeService(String, String, String): bool
updateService(String, String, String, String, int): bool
+ migrateService(String, String, String, String, String): bool
-deploymentBuilder
DeploymentBuilder
+ buildService(ServiceDto): bool
-config
-config
agnosticcloudinterface
<<enum>>
CloudProvider
AZURE
GCP
AWS
Local
ClusterConfigFactory
+ createConfig(String, String, String): ClusterConfig
-clusterConfig
<<Interface>>
ClusterConfig
+ getConfig(String, String): Config
AwsConfig
+ getConfig(String, String): Config
LocalConfig
+ getConfig(String, String): Config
GcpConfig
+ getConfig(String, String): Config
AzureConfig
+ getConfig(String, String): Config

Service Management API Documentation

DEPLOY SERVICE

Deploys a service on a cloud or local platform.

`POST /api/services/deploy`

```
    Body:
    {
        "service": {...},
        "container": {...},
        "providerConfig": {...}
    }
```

REMOVE SERVICE

Removes a service from a Kubernetes cluster/namespace.

`DELETE /api/services/{service_name}?`
`provider={provider}&cluster={cluster}&namespace={namespace}`

UPDATE SERVICE

Updates the container image of a service.

```
PUT /api/services/{service_name}?
provider={provider}&cluster={cluster}&namespace={namespace}
```

```
Body:
{
    "containerImage": "",
    "envVars": [],
    "initialCommand": ""
}
```

SERVICE MIGRATION

Migrates a service from one provider to another.

```
PUT /api/services/{service_name}?
provider={src_provider}&cluster={src_cluster}&namespace={src_namespace}
```

```
Body:
{
    "tgt_provider": "",
    "tgt_cluster": "",
    "tgt_namespace": ""
}
```

GET SERVICE STATE

Retrieves the state of a specific service.

```
GET /api/services/{service_name}?
provider={provider}&cluster={cluster}&namespace={namespace}
```

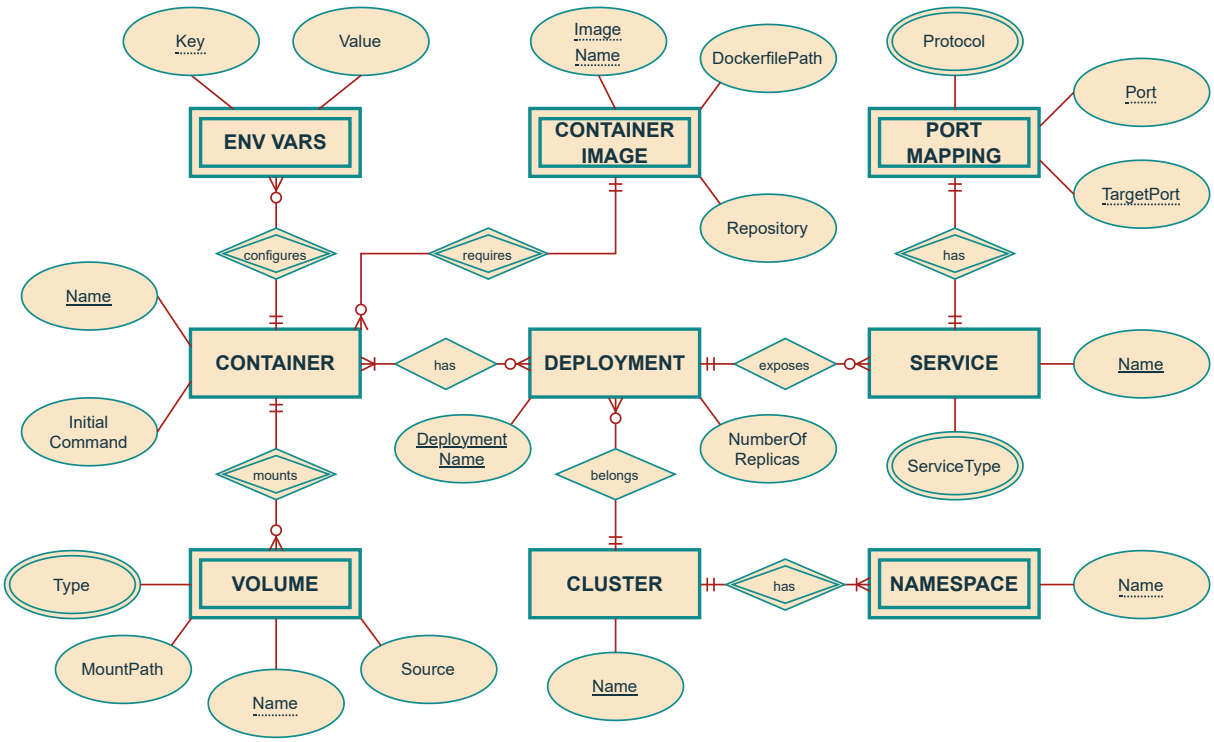
LIST RUNNING SERVICES

Lists running services.

```
GET /api/services?
provider={provider}&cluster={cluster}&namespace={namespace}
```

D

Entity-Relationship Model



Entity-Relationship Model Attributes

Table E.1: ER model attributes description, adapted from [38]

Attribute	Description
Name (from Deployment)	The name of the deployment (must be unique to the namespace of the cluster, and can be up to 253 characters long and consist of lower-case alphanumeric characters, '.' and '-').
NumberOfReplicas (from Deployment)	Field indicates the desired number of identical pod replicas that the Deployment should maintain in a Kubernetes Cluster.
InitialCommand (from Container)	Command to execute upon container boot-up (optionally).
ImagePath (from Container)	Container image location or endpoint.
Repository (from ContainerImage)	The entity or organization responsible for creating and maintaining the container image.
ImageName (from ContainerImage)	The name of the container image.
DockerfilePath (from ContainerImage)	Path to the Dockerfile from the root of the repository (defaults to 'Dockerfile').
Name (from Namespace)	The namespace name or identifier.
Name (from Cluster)	The cluster's name.
Name (from Service)	The name to be assigned to the service.
ServiceType (from Service)	A service type defines how the service is exposed and how it behaves. (`ClusterIP`, `LoadBalancer`, or `NodePort`)
Port (from PortMapping)	The port on which the service will accept traffic.
TargetPort (from PortMapping)	The port that the containers accept traffic on and where the service should route traffic to. To be used when the containers accept traffic on a different port from the service.
Protocol (from PortMapping)	The network protocol used for the port mapping (TCP is used by default). (`TCP`, `UDP`)
Name (from Volume)	A unique name for the volume, used to reference the volume in both Containers and Volumes sections in a pod specification.
Type (from Volume)	The type of the volume supported by Kubernetes to be created, including `EmptyDir`, `HostPath`, `PersistentVolumeClaim`, `ConfigMap`, and `Secret`.
Source (from Volume)	Path to the host's filesystem directory used as the volume source (optional, as it depends on the volume type).
MountPath (from Volume)	The path within the container where the volume is mounted.

Deployment Object (TREE file)

```
{
    "typeId": "Deployment",
    "name": "Hello World Deployment",
    "children": [
        {
            "typeId": "Service",
            "name": "Hello World 1",
            "container": {
                "containerImagePath":
↪ "bonacho17/web-service-1:latest",
                "containerName": "hello-world-1-container",
                "initialCommand": "",
                "envs": []
            },
            "serviceConfig": {
                "serviceName": "hello-world-service-1",
                "ports": [
                    {
                        "port": 8081,
                        "targetPort": 8081,
```

```
                            "protocol": "TCP"
                        }
                    ],
                    "serviceType": "Load Balancer"
                },
                "cloudProvider": {
                    "vendor": "Local",
                    "cluster": "minikube",
                    "namespace": "default",
                    "migration": false
                }
            }
        ]
    }
```