1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Diogo Duarte Moutinho Fevereiro

SMART ORCHESTRATION ON CLOUD-NATIVE ENVIRONMENTS

Dissertation in the context of the Master in Informatics Engineering, specialization in Communications, Services and Infrastructures, advised by Professor Doctor Bruno Miguel Sousa and Luis Filipe Vieira Cordeiro and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Diogo Duarte Moutinho Fevereiro

Smart Orchestration on Cloud-Native Environments

Dissertation in the context of the Master in Informatics Engineering, specialization in Communications, Services and Infrastructures, advised by Professor Doctor Bruno Miguel Sousa and Luis Filipe Vieira Cordeiro and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Diogo Duarte Moutinho Fevereiro

Orquestração Inteligente em ambientes Cloud-Native

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Comunicações, Serviços e Infraestruturas, orientada pelo Professor Doutor Bruno Miguel Sousa e Luis Filipe Vieira Cordeiro e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2023

This work falls in the communications, services, and infrastructure specialization area and it was supported by OneSource, Consultoria Informática Lda.

This work is partially supported by the project Cloud for Holography and Augmented RealITY (CHARITY) and has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 101016509.

This work has been supervised by Professor Bruno Sousa, who lectures at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra and OneSource's Chief Technical Officer Luís Cordeiro.

Acknowledgements

Firstly, I would like to thank Professor Bruno Sousa from University of Coimbra, and Luís Rosa and Vítor Fonseca, which were my supervisors at OneSource. Their knowledge and patience during the course of the internship, proved to be indispensable.

In addition, I would like to thank Luís Cordeiro and the rest of OneSource staff for the opportunity and receiving me with open arms, providing everything I needed for my success.

I also would like to thank my parents Fernando and Palmira, both who supported throughout my life every way they could, my older brothers Ricardo and Luís for being a goal to works toward to. They supported me and provided me everything I needed to be the best I could be, always advising me and teaching me to be a better person.

I would like to thank my closest friends Pedro Durão, Valter Pino and Diogo Muxagata, which I grew with and have known for almost 20 years. For them, I always strived to be a goal that they could work towards too, like my brothers were to me.

Lastly, I would like to thank my grandparents Fernando and Eva who helped raise me and wanted to see me graduate and lead a stable and peaceful life. They helped me throughout my whole life, always prioritizing me over them.

I dedicate this thesis to my grandfather Fernando José Fevereiro who passed away. His greatest dream was to see his family leading a stable and peaceful life. Being the youngest grandson, he wanted to see me graduate and live my future life the way he envisioned.

Abstract

Nowadays, managing Cloud-Native applications is becoming increasingly complex. These applications are progressively expanding into various domains and cloud environments, including multi-cluster scenarios. Orchestrating such a highly heterogeneous infrastructure becomes a challenge. In this work, conducted in the scope of the CHARITY project, we focused on researching solutions for automating the management of these environments and applications. This included the research of Cloud-Native approaches for consistent infrastructure bootstrapping and multi-cluster interconnectivity.

In that sense, we proposed a Cloud-Native orchestration architecture tailored for Kubernetes clusters and container-based applications. This architecture, aligned with the principles of the Zero touch network & Service Management (ZSM) framework from ETSI and closed loops, is rooted in the concept of automating the creation and management of different kinds of services and resources (i.e., clusters, applications, etc.). The proposed approach intends to allow ongoing monitoring and reaction to changes in resources and infrastructure (e.g., creating a cluster on demand, scale-in/-out a cluster on demand when needed). We implemented and validated the conceived solution using different scenarios and use cases of cross-cluster applications.

Our findings highlight the usefulness and feasibility of the proposed solution for supporting a more efficient (i.e., automated) lifecycle management of a multi-domain Cloud-Native infrastructure and applications. The performed implementation and obtained results were reflected in several scientific publications and CHARITY project demonstrations.

Keywords

Orchestration, Automation, Kubernetes, Cloud-Native, Resource Management.

Resumo

Atualmente, a gestão de aplicações Cloud-Native está a tornar-se cada vez mais complexa. Estas aplicações estão a expandir-se progressivamente para vários domínios e ambientes de cloud, incluindo cenários multi-cluster. Orquestrar uma infraestrutura altamente heterogénea torna-se um desafio. Neste trabalho, realizado no âmbito do projeto CHARITY, focámo-nos na pesquisa de soluções para automatizar a gestão destes ambientes e aplicações. Isso incluiu a pesquisa de abordagens Cloud-Native para a configuração consistente da infraestrutura e interconetividade multi-cluster.

Nesse sentido, propusemos uma arquitetura de orquestração Cloud-Native adaptada para clusters Kubernetes e aplicações baseadas em *containers*. Esta arquitetura, alinhada com os princípios do framework *Zero Touch Network & Service Management (ZSM)* da ETSI e com *closed loops*, assenta no conceito de automatizar a criação e gestão de diferentes tipos de serviços e recursos (ou seja, clusters, aplicações, etc.). A abordagem proposta pretende permitir uma monitorização contínua e reação a alterações em recursos e infraestrutura (por exemplo, criar ou redimensionar um cluster quando necessário). Implementámos e validámos a solução concebida em diferentes cenários e casos de uso de aplicações inter-cluster.

As nossas conclusões destacam a utilidade e viabilidade da solução proposta para apoiar uma gestão mais eficiente (ou seja, automatizada) do ciclo de vida de uma infraestrutura Cloud-Native multi-domínio e de aplicações. A implementação realizada e os resultados obtidos foram refletidos em várias publicações científicas e demonstrações do projeto CHARITY.

Palavras-Chave

Orquestração, Automação, Kubernetes, Cloud-Native, Gestão de Recursos.

Contents

Acronyms

AI	Artificial Intelligence.
AMF	Application Management Framework.
API	Application Programming Interface.
BGP	Border Gateway Protocol.
BYOH	Bring Your Own Host.
CD	Continuous Deployment.
CI	Continuous Integration.
CIDR	Classless Inter-Domain Routing.
CLI	Command Line Interface.
CNI	Container Network Interface.
CPU	Central Processing Unit.
CR	Custom Resource.
CRD	Custom Resource Definition.
CRI	Container Runtime Interface.
DEI	Department of Informatics Engineering.
DNS	Domain Name System.
ETSI	European Telecommunications Standards Institute.
FQDN	Fully Qualified Domain Name.
HTTP	Hypertext Transfer Protocol.
IaaS	Infrastructure as a Service.
IP	Internet Protocol.
IPAM	IP Address Manager.
IT	Information Technology.

K8S Kubernetes.

Kube Kubernetes.

MANO Management and Orchestration.

NFV Network Functions Virtualization.

NIST National Institute of Standards and Technology.

OODA Observe, Oriente, Decide, Act.

OS Operating System.

OSS Open Source Software.

OVN Open Virtual Network.

PaaS Platform as a Service.

PoC Proof of Concept.

RAM Random Access Memory.

SaaS Software as a Service.

SDN Software-Defined Networking.

SSH Secure Shell.

SSL Secure Sockets Layer.

TLS Transport Layer Security.

TOSCA Topology and Orchestration Specification for Cloud Applications.

VM Virtual Machine.

VNF Virtualized Network Functions.

VPC Virtual Private Cloud.

VPN Virtual Personal Network.

VR Virtual Reality.

ZSM Zero-touch network & Service Management.

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, Cloud-Native applications are increasing in size, complexity and number. The cloud-native approach aims to design, build and run virtual functions and services exploiting the Cloud model. Such applications are developed using tools that allow them to maximise the benefits of a Cloud-based environment. These benefits include greater agility in development, integration and installation. Tools and concepts such as Continuous Integration (CI) / Continuous Deployment (CD), container engines and container orchestrators such as Kubernetes are among the pillars and drivers of this Cloud-Native transformation.

On the other hand, these applications, which progressively take advantage of the edge-to-cloud continuum, require a revamped orchestration approach. Large distributed applications require intelligent and automated management instead of manual, complex, and error-prone configuration. Today, the management of clusters and applications mainly relies on the intervention of human operators for their configuration and oversight. Managing such applications can include orchestrating a complex infrastructure composed of heterogeneous environments and numerous application components spanning such infrastructure.

Hence, approaches such as European Telecommunications Standards Institute (ETSI) Zero-touch network & Service Management (ZSM) are considered today, focusing on bringing intelligence as well as service automation for multi-domain environments. Such intelligent and automated orchestration is a step towards the aim of minimising the need for the human factor.

Considering that this work, conducted in the scope of a master's internship at OneSource [1] and integrated within the context of a European research project, H2020 CHARITY [2], where OneSource is involved, focuses on researching solutions for automating the lifecycle management of these cloud environments and applications.

The major goal of this work was to develop a Cloud-Native orchestration system capable of managing clusters and applications in multi-domain environments. This included the research of Cloud-Native approaches for consistent infrastructure bootstrapping and multi-cluster interconnectivity.

We started by researching the main concepts of orchestration, management, and automation focused on clusters and *Kubernetes*. We researched concepts of the *Kubernetes* itself, management-related frameworks and interconnectivity approach targeting multi-cluster environments.

Such research resulted in the first proof of concepts regarding cluster interconnectivity and orchestration. For cluster interconnectivity, two scenarios were created, one for *Liqo* [3][4] and the other for *Submariner* [5][6]. These scenarios were used to compare those frameworks and decide which would best fit the proposed architecture. Concerning cluster orchestration, a scenario was created for *Cluster API* [7] to assess its capabilities and provider support.

Later, we defined the requirements based on internal project discussions and CHARITY objectives, and we proposed an orchestrator architecture to answer them.

The final steps included developing the orchestrator and its functional testing, integration and validation. We discuss the various architecture components and their roles in several use cases of distributed applications. We started by deploying the orchestrator in the CHARITY testbed. We performed a series of tests to validate the orchestrator's capabilities, and later, we conducted a set of integration and validation to further evaluate the correct behaviour and suitability in supporting the various project use cases.

1.1 Thesis Objective

In brief, this work aimed to develop a smart orchestration and management platform focused on distributed Cloud-Native environments. This means designing, implementing and evaluating a Cloud-Native orchestration solution focused on *Kubernetes* clusters and application provisioning on distributed and edge-cloud domains.

1.2 Contributions

The contributions of this work were:

1. Implementation of an orchestration proof of concept using Cluster API and OpenStack for enabling the creation of clusters on demand.

2. Implementation of two proof of concepts for cluster interconnectivity using different technologies, Liqo and Submariner.

3. Proposal of an orchestration solution based on the early proof of concepts.

4. Support of CHARITY integration and testing activities in the deployment and configuration of various components and applications.

5. Contribution to the CHARITY project documentation, and deliverables.

   This work also resulted on the following scientific contributions:

   - Video Streaming use-case discussed and presented in a journal publication entitled "Cross Kubernetes Cluster Networking to Support XR Services: Challenges, Solutions and Performance Evaluation" submitted to the IEEE Network Magazine. Despite not being credited as an author, tests and results from this work contributed to the referred paper.
   - Writing of a conference paper entitled ("Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case"), which was submitted and accepted to the "IEEE Symposium on Intelligent Edge Computing and Communications (iEDGE)" conference.
   - Showcasing the orchestrator capabilities at the EUCnC & 6G Summit 2023.

1.3 Document Structure

The structure of the document is organized as follows:

- **Chapter 2** provides an overview of the concepts covered in this work, such as container orchestration, cloud deployment models and multi-cluster architecture. It also provides an overview of the technologies used to implement the proof of concepts.
- **Chapter 3** presents this work's research objectives and explains the approaches taken for each objective.
- **Chapter 4** presents the planning and risks of this work.
- **Chapter 5** describes the first proof of concepts developed.
- **Chapter 6** provides a deep view of the development of the orchestration system.
- **Chapter 7** presents the orchestration's system testing, integration and validation with external frameworks and CHARITY use-cases.
- **Chapter 8** provides a conclusion and critical view of the work done during the internship.

Chapter 2

Background and Related Work

This chapter introduces the concepts of orchestration, automation and cloud deployment models, among other related topics needed to understand this work. The multi-cluster architecture is also introduced, along with its usefulness and challenges.

Section 2.1 briefly introduces the orchestration concept, the standard for current state of the art orchestrators and the different types of resources that can be orchestrated. Section 2.2 exposes concepts about the different cloud deployment models and their different characteristics. Section 2.3 is dedicated to multi-cluster architecture, where we introduce its concept, explain different approaches, the advantages it provides, as well as its challenges. Section 2.4, different tools and frameworks, that can be used to create possible solutions to the multi-cluster challenges, are also presented and explained.

2.1 Containers, Virtual Machines and Clusters Orchestration

Orchestration consists of the automated configuration and coordination of infrastructure servers, applications and services converging in a seamless workflow [8]. The automation reduces or entirely replaces human interaction with IT systems and instead uses software to perform the same tasks in order to reduce cost, complexity, and errors.

European Telecommunications Standards Institute (ETSI) Zero-touch network & Service Management (ZSM) is a framework and set of standards focused on automating and simplifying the management of telecommunication networks and services. It aims to reduce manual intervention in network operations and make networks more agile and efficient. ETSI ZSM is closely related to orchestration, especially in the context of network functions and service [9].

This framework focuses on automating various aspects of network and service management, as orchestrators play a pivotal role in this endeavor by defining and executing workflows for provisioning, scaling, and managing network resources and services. Orchestrators are typically built on technologies like Network Functions Virtualization (NFV) and Software-Defined Networking (SDN), which facilitate the automation of intricate network operations.

Several key features of the ETSI ZSM framework are detailed below:

- **Service Orchestration:** This framework introduces the concept of service orchestration, which involves the automated provisioning and management of end-to-end services, even when they span multiple network domains and technologies. Orchestrators enable the creation and management of such services by efficiently coordinating the deployment of Virtualized Network Functions (VNF) and physical resources.
- **NFV Integration:** NFV is pivotal for ETSI ZSM framework. This entails virtualizing network functions like routers, firewalls, and load balancers and deploying them as software instances on standard hardware. Orchestrators take charge of placing and connecting these virtualized functions as required to fulfill service requirements.
- **Resource Coordination:** The framework encompasses the orchestration of both physical and virtual resources. Orchestrators can dynamically allocate compute, storage, and network resources based on the demands of services. This ensures efficient resource utilization and the delivery of services with the necessary performance and scalability.
- **Multi-Domain Management:** ETSI ZSM recognizes the need to manage services that traverse multiple network domains and are offered by different operators. Orchestrators facilitate the coordination of activities across various administrative domains, enabling end-to-end service provisioning and management.
- **Complete Lifecycle Oversight:** As a component of the framework, orchestration assumes responsibility for the entire lifecycle management of network services. This encompasses service instantiation, scaling, self-healing, updating, and decommissioning. Orchestrators guarantee that services remain operational and adaptable to changing conditions.
- **Policy-Driven Approach:** The framework promotes a policy-driven orchestration model, where policies and rules dictate how services should be managed and resources allocated. Orchestrators enforce these policies to ensure compliance and the efficient use of resources [10].

ETSI ZSM establishes the overarching framework and standards for automating the management of network and service operations, while orchestration platforms put these principles into practice by automating and coordinating network functions and resources to deliver services efficiently and at scale.

Considering the different resources that we can orchestrate (e.g., containers, virtual machines, clusters), one does not replace the other. Each of the resources has its own advantages and disadvantages, so rather than completely replace what is orchestrated, it is added to the pool of orchestration appliances. Automating the process then becomes a necessity, as the difficulty increases as we increase the number of orchestration resources in the system, and managing different types of orchestration resources requires different configurations. Orchestrating containers is different from orchestrating virtual machines and even more when we orchestrate them together.

With the orchestration of a combination of resources, we acquire a more robust and refined scaling, as some applications can't run on a container due to its small *footprint*, and other applications require a complete system to run, manage and maintain them, which are designed as *clusters*. An application may run on the *frontend* on a virtual machine, and use a cluster as the *backend*.

A **Cluster** is defined as a collection of physical and/or virtual machines, called *nodes*, deployed as a single system running distributed workloads. Figure 2.1 represents the architecture of a *Kubernetes* cluster.

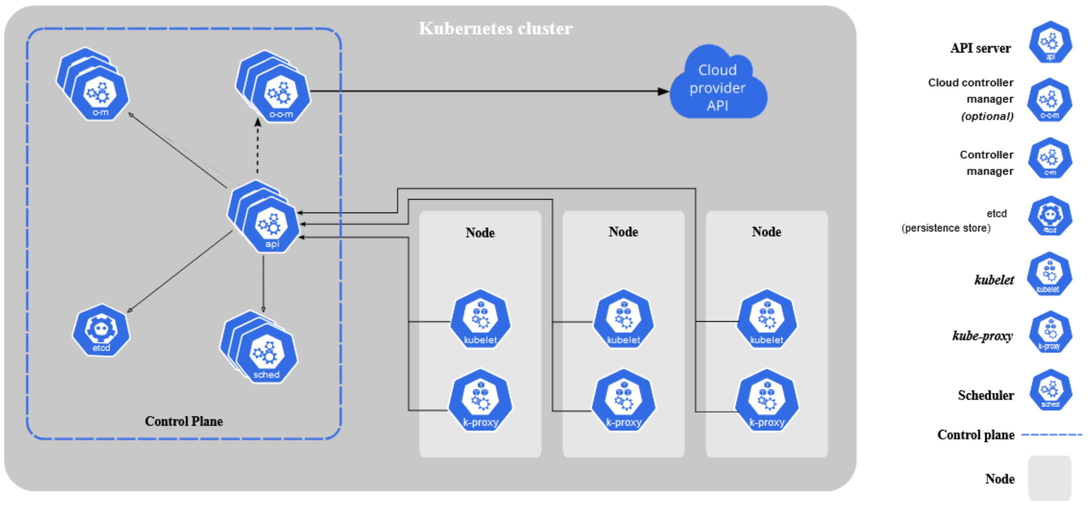


Figure 2.1: Kubernetes Single Cluster Architecture [11]

The nodes belonging to the cluster are viewed as computational resources, able to run workloads that a single node wouldn't be able to fulfil the performance needed. With this in mind, we can easily scale the cluster vertically by adding more and more nodes to the cluster. We can also scale horizontally, firstly by replicating the resources inside the cluster. For example, if a node was running a containerized application, we could instantiate replicas so we could load balance the workload between replicas. These replicas could be distributed across nodes delivering a form of disaster recovery to the system.

To ease and standardize the orchestration process of *Cloud-Native* applications, we use Topology and Orchestration Specification for Cloud Applications (TOSCA) blueprints.

TOSCA blueprints describe the architecture of cloud-based applications in a standardized format. TOSCA blueprints are written in *.yaml* and are used to define the components that make up an application, their relationships, and the policies that govern their deployment and operation, and are composed of two main sections:

- The service template, which describes the topology of the application and the components that make it up. It also defines the relationships between these components, such as which component depends on which other component.
- The policy templates describe the rules and constraints that govern the deployment and operation of the application. These can include things like scaling policies, security policies, and performance policies.

TOSCA blueprints allow for the automation of the deployment, scaling and management of cloud-based applications. This is achieved by **using TOSCA-compliant orchestration engines** that can interpret the blueprint and use it to create and manage the application on a cloud platform. These blueprints are **platform-agnostic** and can be used to deploy applications on any cloud infrastructure that supports TOSCA, allowing portability of the applications across different cloud environments. They are also widely adopted by cloud providers, network functions vendors and other IT vendors to provide standardization and automation of cloud application deployments.

2.2 Cloud Deployment Models

Cloud Computing, as defined by National Institute of Standards and Technology (NIST), "is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [12].

Cloud provides different types of services, the most common models being Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), and are described briefly below [13]:

IaaS provides on-demand access to computational resources such as servers, networking and storage, allowing users to scale resources as they are needed, eliminating the planning of physical infrastructure. Compared to the other services, *IaaS* is the lowest-level of computing resources.

PaaS provides developers with computational resources, development tools and infrastructure, where they can manage and develop their applications. The whole infrastructure is hosted and managed by the cloud provider at their data centre. Nowadays, *PaaS* is built around containers, being an easier and more lightweight way of deploying applications.

SaaS is application software hosted by the cloud provider and users can access it, as long as they are connected to the service.

The resources and services demanded of cloud environments are dependent on the cloud provider, and as such, to accommodate different software, different **cloud deployment models** are needed.

There are 5 types of cloud deployment models [12] as described below:

- **Public Cloud:** The cloud infrastructure is provisioned for use by the general public and it runs on the premises of the cloud provider. "It may be owned and managed by a business, academic, or government organization, or some combination of them".

- **Private Cloud:** The cloud infrastructure is provisioned for a single organization and it may exist on or off premises. The infrastructure can be owned and managed by the said organization, a third party, or both.

- **Community Cloud:** The cloud infrastructure is provisioned for exclusive use by a specific community consisting of member organizations that have shared concerns.

- **Hybrid Cloud:** "The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, public and community) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability".

- **Multi-Cloud:** The cloud infrastructure is provisioned by two or more different cloud providers. Similar to the hybrid cloud deployment approach, instead of combining private and public clouds, it combines multiple public clouds. The *multi-cloud* deployment model allows users to take advantage of specific services of each provider and lessens the chances of *vendor-lock in*.

We can deploy entire clusters to the cloud, and distribute them across different cloud types and even different cloud providers, taking advantage of all the benefits of each cloud type, and also of the services and tools supplied by specific cloud providers. Most of the cloud deployment models rely on the usage of multi-cluster architectures, hence the relevance of multi-cluster orchestration as referred in this work.

2.3 Multi-Cluster Architecture

This section provides an overview of multi-cluster architecture along with its concepts and challenges, as well as tools and frameworks used to research and solve the multi-cluster topology challenges.

A multi-cluster infrastructure is a collection of clusters that can work together to fulfil a set of business requirements. This allows new setups to be designed around this type of architecture [14]. As an example, you can have cluster redundancy architecture in which clusters are replicas of each other, ensuring *high availability* through a *load balancer*, as shown in Figure 2.2.

The main advantages provided by multi-cluster architectures are the following [15]:

- **Increased scalability and availability:** In multi-cluster topologies, clusters can be spread across different regions, increasing the availability for the

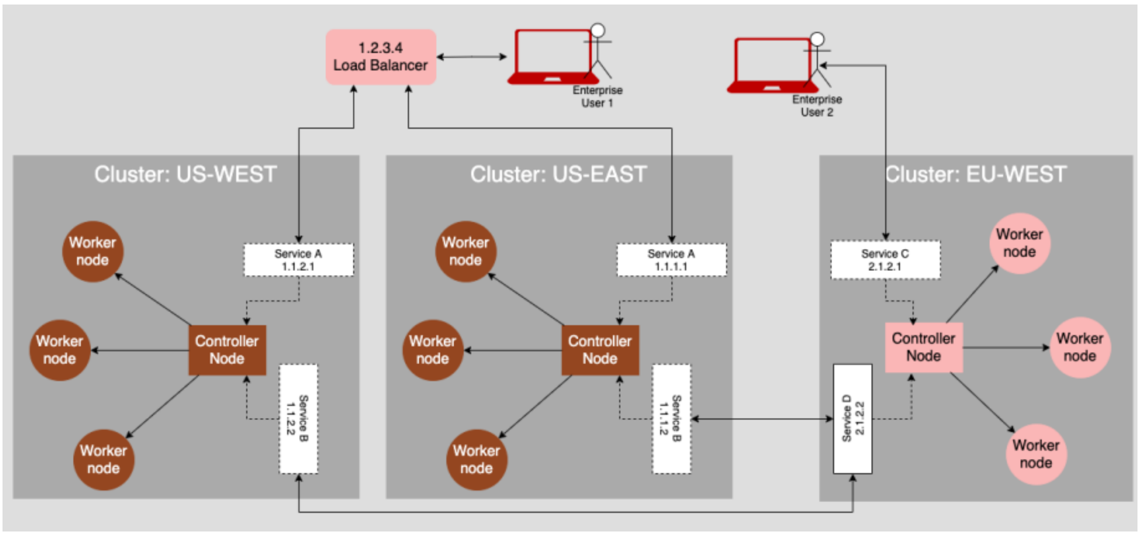


Figure 2.2: Multi-Cluster Architecture based on Kubernetes Clusters [14]

end-users, giving them a better experience, because of the reduced latency and the faster communication between the end-user and the application. Regarding the scaling, application deployments can be spread across clusters, and scaled accordingly to the load and resource consumption of each cluster.

- **Application isolation:** True application isolation can be achieved, by using different clusters for development and production, or by deploying distinct applications to different clusters. Issues also become easier to diagnose, as we are able to easily identify the problem, and optimizations on a cluster don't affect other deployments.
- **Regulatory compliance:** Regarding data privacy, different countries have different regulations and policies dictating how, what and which data we can exchange with users. Also, most regions only allow user data to reside within geographical limits. With multi-cluster topologies, we can deploy clusters in different geographical regions, each complying with different data policies, limiting the scope and meeting the demanded requirements.
- **Vendor lock-in:** Cloud resources are most of the time outside of our control. Cloud providers can lock consumers out, denying the services. With clusters deployed across different providers, we can mitigate this issue, so even if one provider fails, we still have access to cloud resources.
- **Distributed applications:** Most applications are now divided in multiple components, which can be containerized. Considering the increasing on *edge computing* requirements, the containerized components can be distributed across *edge clusters* and *core clusters*, enabling better performance and experience overall.

Multi-cluster topologies can be divided in **two main types** regarding its *design*, the types being **segmentation** and **replication**.

In a **segmentation** architecture, an application is separated into independent components, which are then deployed to the different clusters. The components can be represented as *Kubernetes* services and they can interact with each other across clusters if the application architecture demands it. Figure 2.3 presents an example of a multi-cluster segmentation architecture.

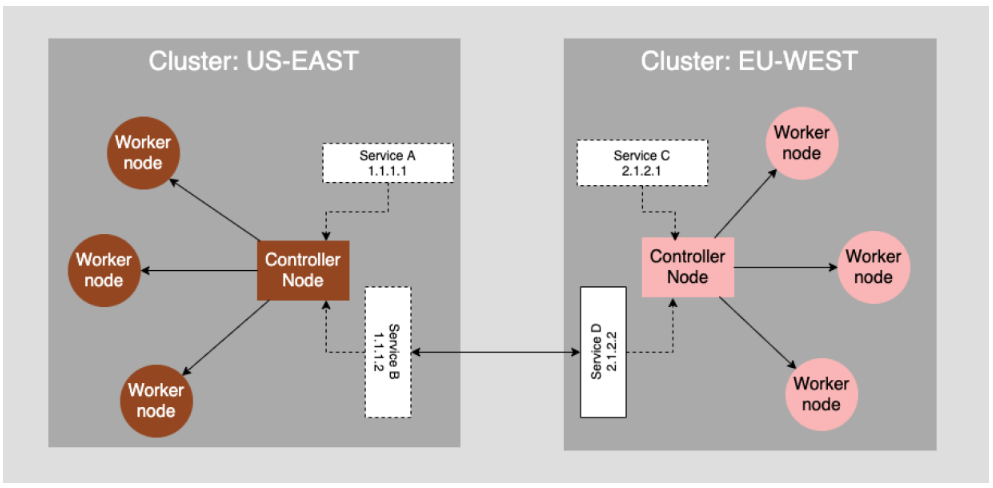


Figure 2.3: Multi-Cluster Segmentation Architecture [14]

In the **replication** approach, exact replicas of the cluster are created and are usually deployed to different regions, although it is not a requirement. Figure 2.4 presents an example of a multi-cluster replication architecture with clusters deployed in two different regions.

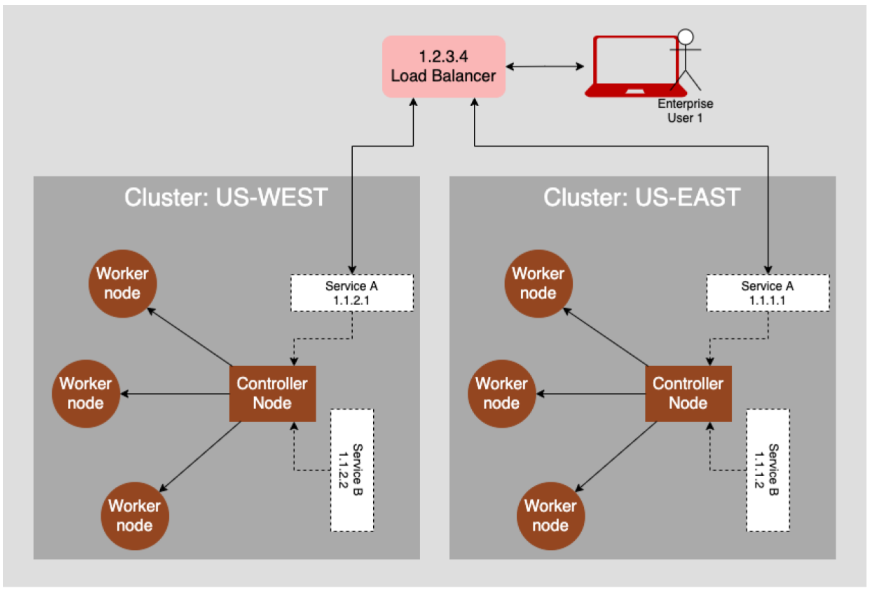


Figure 2.4: Multi-Cluster Replication Architecture [14]

As the clusters are deployed in different regions, you can route the services based on the proximity to the *end-user*, ensuring the best performance is achieved. The replication architecture also ensures *disaster relief*. If a cluster shuts down, a replica can act as a *backup*.

Overall, the specific design of a multi-cluster architecture will depend on the needs and goals of the system or application being deployed. However, despite the great advantages, multi-cluster topologies increase system complexity, making it hard to manage and maintain. This complexity bears some challenges we need to overcome, as detailed in the following subsections.

2.3.1 Multi-Cluster Connectivity

Cluster connectivity is a requirement in this type of topology, as pods and services need to be able to communicate with pods and services on other clusters seamlessly. This can prove to be quite bothersome, especially if the clusters are deployed across different regions. Some cloud providers offer services that allow connectivity between clusters, however, we are dependent on the cloud provider and we cannot implement an architecture where we distribute clusters across different cloud providers.

In *Kubernetes* clusters we can expose a service through an ingress using only native *Kubernetes* resources, but we would have to do it manually. This is not feasible at all, as every time we needed to expose a service, we needed to set up routing rules and make sure that those rules do not interfere with the previously defined rules. As the cluster complexity increases, this becomes an impossible task.

Liqo [3] [4] and **Submariner** [5] [6] are two of the technologies that can help mitigate the cluster connectivity issues, automating the process and providing some useful features, as with **Liqo** we have **workload offloading** and with **Submariner** we have **service discovery**. A more detailed explanation of **Liqo** and **Submariner** is given in Sections 2.4.2 and 2.4.3 respectively.

Two proof of concept scenarios were implemented during the internship which describes in detail both implementations and helps better understand this challenge (refer to Sections 5.3 and 5.4).

2.3.2 Multi-Cluster Orchestration

Multi-cluster orchestration refers to the process of coordinating and managing multiple clusters in order to achieve a specific goal. This involves managing and coordinating the activities of these clusters in order to ensure that they are working together efficiently and effectively. Multi-cluster orchestration can include tasks such as scheduling workloads, balancing resources, and monitoring the performance of the clusters.

As we add clusters to the infrastructure, the complexity increases and management becomes more of an issue. We can rely on tools like *Kubernetes* (detailed in Section 2.4.1) as it is already the most common tool for orchestration, but *Kubernetes* by itself does not suffice. The clusters need a machine to be deployed to, which needs to be already created before the cluster deployment. **Cluster API** is a tool that takes *Kubernetes* manifest logic to a higher level, being able to deploy custom virtual machines with clusters pre-installed, to different cloud providers.

A proof of concept scenario was implemented during the internship which solves this challenge (refer to Section 5.4).

2.3.3 Multi-Cluster Automation

Multi-cluster automation refers to automating the multi-cluster orchestration process detailed in the previous subsection.

Despite the fact that all of the orchestration can be done step by step, it is simply not feasible, as the complexity of the architecture is too great for a human to handle, as it is prone to mistakes and takes too much time to manage such complex system. There are several advantages to automate the orchestration process, as automation can help:

- **Improved efficiency:** Streamline processes and eliminate the need for manual intervention, resulting in improved efficiency.
- **Increased accuracy:** Reduce the risk of errors and improve the accuracy of tasks.
- **Reduced costs:** Reduce the cost of operations by eliminating the need for manual labor and improving the efficiency of processes.
- **Better resource utilization:** Optimize resource utilization, ensuring that resources are used effectively and efficiently.

The creation of a **custom orchestrator** that can use both *Kubernetes* and *Cluster API* (refer to Section 2.4.4) to its full potential, is the chosen approach for solving the automation challenge.

The orchestrator will be based on the Observe, Oriente, Decide, Act (OODA) approach [16] consisting in four steps, shown in Figure 2.5:

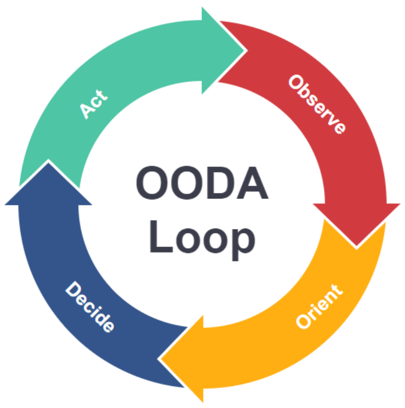


Figure 2.5: OODA Loop approach diagram [17]

- **Observe:** The first step is to identify the problem and gain an overall understanding of the environment. This can be interpreted as data gathering, where all of the information regarding the current state of the system is collected. The key point about the observe step is recognizing that the system is complex. All data is a snapshot in time and must be treated as such. Therefore, entities must gather whatever information is available, as quickly as possible in order to be prepared to make decisions based on the collected data.

- **Orient:** The orientation phase involves thinking about the information gathered through observations and deciding on the next steps to take. This requires a high level of situational awareness and understanding in order to make well-informed decisions. This phase involves consciously considering the reasons behind the decisions made before choosing a course of action. Machine learning tools can be used to create situational models that identify potential outcomes [10] [18] [19].
- **Decide:** The decision phase makes suggestions towards an action or response plan, taking into consideration all of the potential outcomes.
- **Act:** Action pertains to carrying out the decision and related changes that need to be made in response to the decision. This step may also include any testing that is required before carrying out an action.

2.4 Tools and Frameworks

This section focuses on giving a more detailed view of the technologies regarding the multi-cluster architecture.

2.4.1 Kubernetes

Kubernetes [20] is an open-source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications. Kubernetes focuses on automation and provides users with service discovery and load balancing, storage orchestration, automated rollouts and rollbacks, automatic bin packing, self-healing and secret and configuration management.

A description of the components is given to help understand how Kubernetes works.

Kubernetes Pod

Pods [21] are the smallest deployable unit that one can create or deploy in Kubernetes. A pod is a group of one or more containers tightly coupled with shared storage and network resources. For comparison, the same can be achieved with Docker by grouping containers with shared namespaces and shared filesystems.

Figure 2.6 presents pods within nodes, constituting a Kubernetes cluster.

Kubernetes Node

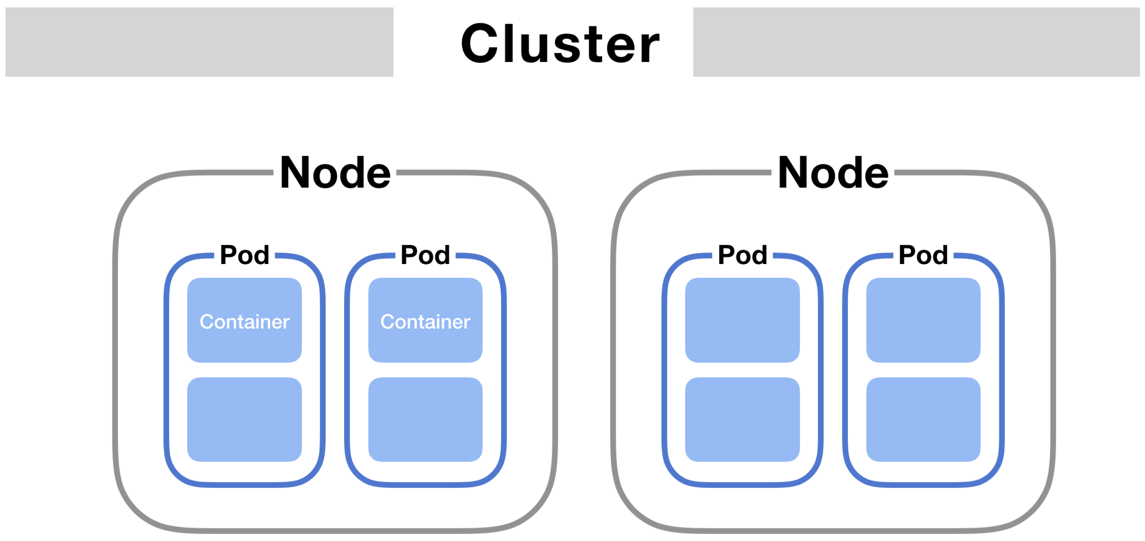


Figure 2.6: Kubernetes Pods within a cluster [22]

A *Kubernetes Node* [23] is a machine, virtual or physical, that houses pods running containerized workloads. In a cluster, one can have multiple nodes, up to 5,000. A node is composed of a *kubelet*, a *container runtime* and a *kube-proxy*.

- **kubelet** - *kubelet* is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The *kubelet* takes a set of *PodSpecs* that are provided through various mechanisms and ensures that the containers described in those *PodSpecs* are running and healthy. The *kubelet* doesn't manage containers which were not created by Kubernetes.
- **container runtime** - The container runtime is the software that is responsible for running containers. *Kubernetes* supports container runtimes such as *containerd*, *CRI-O*, and any other implementation of the *Kubernetes CRI* (Container Runtime Interface).
- **kube-proxy** - *kube-proxy* is a network proxy that runs on each node in the cluster, implementing part of the *Kubernetes Service* concept. It maintains network rules on nodes, which allow network communication to the pods from network sessions inside or outside of the cluster. *kube-proxy* uses the operating system packet filtering layer if there is one and it's available, otherwise *kube-proxy* forwards the traffic itself.

Kubernetes Deployment

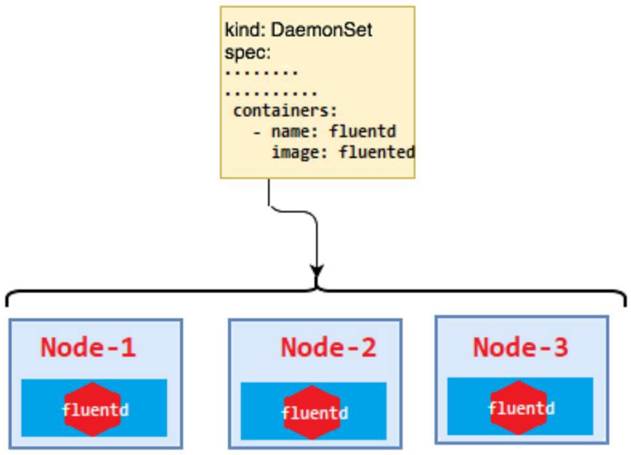
A *Kubernetes Deployment* [24] provides declarative updates for *Pods* and *ReplicaSets*.

Deployments tell Kubernetes how to create or modify instances of the pods that hold a containerized application. Deployments can help to efficiently scale the number of replica pods, enable the rollout of updated code in a controlled manner, or roll back to an earlier deployment version if necessary [25]. In a deployment, the pods are scaled across all the working nodes of a cluster.

The main benefit of using deployments in Kubernetes is error mitigation and time-saving which is achieved by automating the process involved in deploying, scaling, and updating production applications. More automation translates to faster deployments with fewer errors. *Kubernetes deployment controller* is continuously monitoring the health of pods and nodes, allowing real-time changes like replacing a crashed pod or if a node is down, scaling the deployment to other working nodes, and ensuring the operation of critical applications.

Kubernetes DaemonSet

A *Kubernetes DaemonSet* [26] applies the same logic as the *Kubernetes Deployment* [24], the difference being that ensures that all nodes are running a replica of a pod. These pods are created on demand, as nodes are added to the cluster. Figure 2.7 illustrates an example of how a daemonset is deployed.



```
kind: DaemonSet
spec:
........
..........
 containers:
   - name: fluentd
     image: fluented
```

Figure 2.7: DaemonSet deployment example [27]

Container Network Interface (CNI)

Container Network Interface is a framework that handles the configuration of network resources dynamically. *CNI* defines an interface for configuring the network, provisions IP addresses and maintains connectivity with several hosts [28]. A *CNI* when used together with *Kubernetes*, integrates with the *kubelet* to enable the use of an **underlay network** or **overlay network** to configure the network within the cluster automatically. **Overlay networks** encapsulate network traffic by using a virtual interface. **Underlay networks** work at the physical level and include switches and routers. [28]

Following the choice of the network configuration type, the *container runtime* defines the network that containers join. The *container runtime* adds the interface to the container namespace via a call to the CNI plugin and allocates the subnetwork routes via calls to the IP Address Management plugin.[28]

Kubernetes Service

A *Kubernetes Service* [29] is an abstraction of an application running on a set of

Pods as a **network service**. They are mainly used to expose applications or *micro-services*. As Kubernetes Pods [21] can be created or destroyed at any given time to accommodate the desired needs of the cluster, their **IP addresses are subject to change**. This can cause applications to become **unaccessible**, making the **services** extremely useful.

Kubernetes Services breakdown into four types: *ClusterIP*, *NodePort*, *LoadBalancer* and *ExternalName* [30]. Figure 2.8 presents the architecture of the different types of services in Kubernetes.

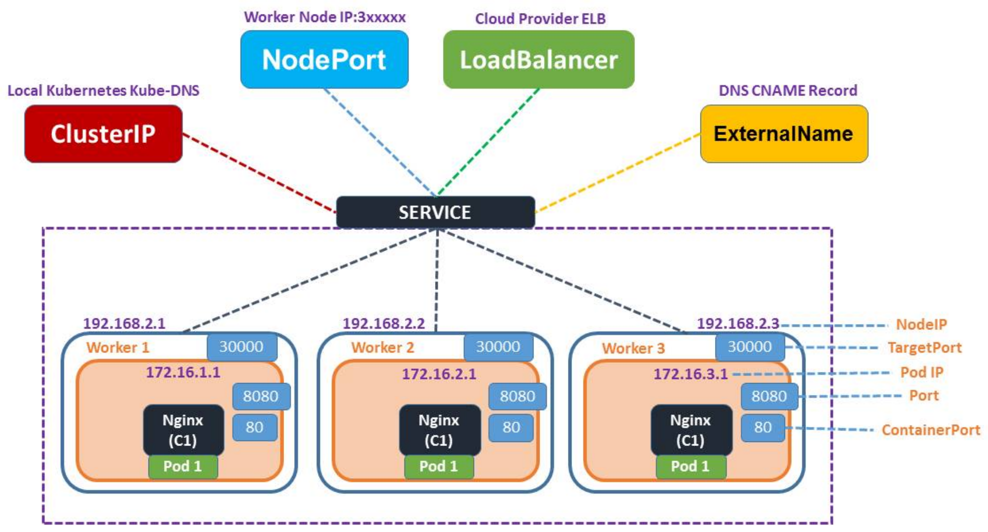


Figure 2.8: Kubernetes Service Architecture [30]

Kubernetes Ingress

A *Kubernetes Ingress* is an "API object that manages external access to the services in a cluster" [31]. Figure 2.9 is an example of an Ingress routing the traffic to a single service.

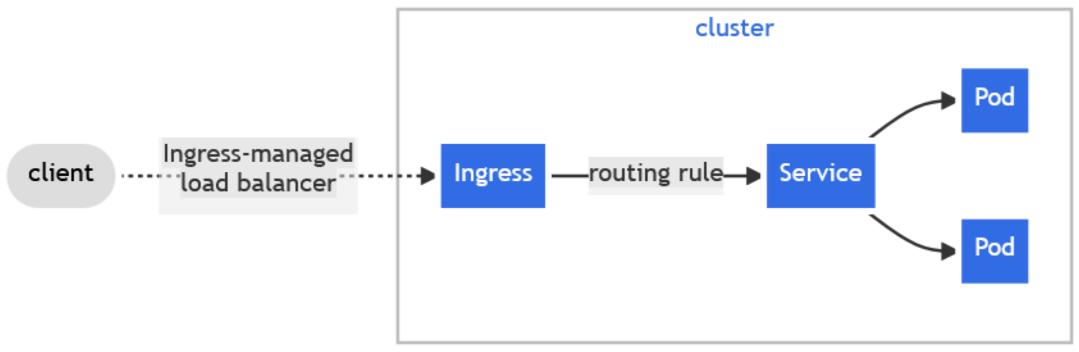


Figure 2.9: Kubernetes Ingress [31]

The ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster, similar to a network gateway. Traffic routing rules are defined on the Ingress resource.

Kubernetes Namespace

A *Kubernetes Namespace* [32] are used to isolate groups of resources within a cluster. Namespaces are useful in scenarios with many users, teams or even projects, ensuring that no one can interfere with each other.

Kubernetes Scheduler

Kube-scheduler is the default *Kubernetes* scheduler and runs as part of the cluster control plane [33]. The scheduler decides in which node unscheduled pods will be deployed, based on the pod requirements. "The scheduler finds feasible *nodes* for a *pod* and then runs a set of functions to score the nodes and picks the node with the highest score among the feasible ones to run the pod. The scheduler then notifies the API server about this decision in a process called binding".

Scheduling decisions take into account factors such as individual and collective resource requirements, affinity and anti-affinity specifications, data locality, inter-workload interference, and some other factors and constraints.

The node selection process consists of two steps, known as **filtering and scoring**. The filtering step consists of finding the set of nodes where a pod can be scheduled to run. After the filtering step, the scheduler ranks each of the filtered nodes and assigns them a score. The pod will be scheduled to the node with the highest score.

The scheduler allows customization of the scheduling policies and profiles, so we can adjust them to our system needs.

Kubernetes Operator

Operators in *Kubernetes* [34] are software extensions that use custom resources to manage applications and components. Operators are **designed for automating workloads** beyond what *Kubernetes* already provides, expanding the cluster's behaviour and acting as controllers to custom resources.

An operator can be used to automate different tasks, such as deploying applications on demand, handling application updates, taking and restoring backups of an application's state, and publishing a service to applications that don't support Kubernetes APIs, among others. To match the needs of different systems, operators can be coded and tailored to implement the wanted behaviour. A list of Kubernetes operators is presented below:

- **Prometheus Operator:** Manages Prometheus monitoring and alerting systems. It simplifies the deployment and management of Prometheus instances, as well as the configuration of monitoring targets.
- **Grafana Operator:** Simplifies the management of Grafana dashboards and configurations on Kubernetes.
- **MySQL Operator:** Automates the deployment and management of MySQL

databases on Kubernetes. It handles tasks like backup, scaling, and high availability.

- **Kafka Operator:** Automates the deployment and scaling of Apache Kafka clusters. It simplifies the configuration of Kafka topics, brokers, and *ZooKeeper*.

- **NGINX Ingress Controller Operator:** Automates the deployment and management of NGINX Ingress controllers for routing external traffic into the Kubernetes cluster.

2.4.2 Liqo

"**Liqo** is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premise, cloud and edge infrastructures" [3, 4].

Liqo provides interconnectivity between clusters using a peer-to-peer approach, allowing **workload offloading**, **service distribution across clusters**, and **multi-cluster applications traffic routing**. The simplicity of Liqo approach, when compared to similar tools and frameworks like *Skupper* and *Submariner*, the latter also studied in this work (refer to Section 2.4.3), is the main advantage, as the configuration is minimal and offloading is very straightforward. Liqo is also **CNI-agnostic**, allowing connectivity between clusters discarding the requirements of clusters need of having the same CNI installed, despite still having some caveats with the *Calico* plugin (refer to Chapter 5, Section 5.2 for more information on **Calico**), as it is common with this type of frameworks.

Peering Clusters

Establishing a peer-to-peer relationship [35] between clusters using Liqo consists in four steps:

- **Authentication:** each cluster, after being authenticated through tokens shared previously, obtains an identity used to interact with other clusters. This identity is then used to negotiate parameters and policies.

- **Parameter Negotiation:** the two clusters exchange parameters needed to complete the peering relationship, such as the amount of resources shared and the information about the setup of the network VPN tunnel. This process is automatic and doesn't require user intervention.

- **Virtual Node Setup:** the consumer cluster abstracts the resources shared with the provider cluster by creating a virtual node in the cluster. This enables task offloading [36] and is compliant with the standard Kubernetes practice, dismissing API modifications.

- **Network Fabric Setup:** the two clusters configure their network and establish a secure VPN tunnel between them, using the parameters negotiated in a previous step. This enables pods hosted by the local cluster to seamlessly interact with the pods offloaded to the remote cluster, regardless of the CNI plugin and configuration.

Liqo supports two types of peering: *in-band control plane* and *out-band control plane*.

The network traffic in the *out-band control plane* is separated, where the traffic between the pods of the two clusters flows in the VPN tunnel created during the peering process and the *Liqo* control plane traffic flows outside the VPN tunnel. This requires exposure of three different endpoints: Liqo VPN endpoint, Liqo authentication endpoint and Kubernetes API endpoint.

Using the *in-band control plane* approach, all the traffic flows through the VPN tunnel (pod traffic and Liqo control plane traffic), requiring only the Liqo VPN endpoint to be exposed. Figure 2.10 presents Liqo *in-band* and *out-band* control plane topologies.

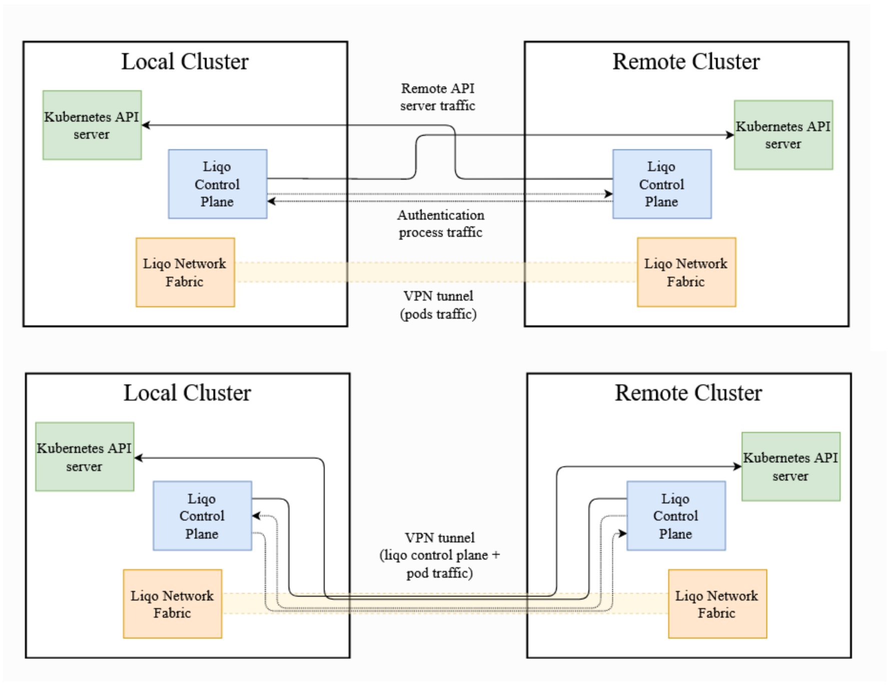


Figure 2.10: Liqo Out-Band Control Plane (top) and In-Band Control Plane (bottom) approaches [35]

Offloading Workloads

In the context of bidirectional cluster peering, Liqo introduces the concept of virtual nodes within each cluster. These virtual nodes represent the resources available in the remote cluster. Liqo goes further by introducing the concept of offloading [36], which allows for the reflection and execution of workloads on these virtual nodes. Offloading enables the exposure of services and the execution of workloads in remote clusters. Liqo allows to offload namespaces, services, and pods.

For example, when offloading a namespace, Liqo extends it by creating an identical twin namespace in the remote cluster. This enables pods and services to operate within this shared cross-cluster namespace. Figure 2.11 compares pod offloading and service offloading. Both modes start with establishing cluster peering, involving creating a dynamic VPN tunnel and configuring a shared namespace.

However, the pod offloading strategy involves transferring the actual execution of pods and services to the peered cluster. For instance, in Figure 2.11, application components are initially deployed in the Green Cluster but are later executed in the Rose cluster. This approach is particularly useful for resource-intensive computing tasks, like video processing or managing peak traffic loads, which can be seamlessly moved to a more suitable cloud cluster. By offloading certain application workloads to a cloud cluster, resource utilization across the edge-cloud continuum can be optimized, resulting in cost savings and improved overall efficiency.

In contrast, service offloading focuses on exposing only the Kubernetes services in a remote cluster. In this scenario, pod execution remains in the original cluster, and the deployment of pods must be initiated anew in the target cluster. Additionally, other components need to be aware of the service names in the remote cluster [37].

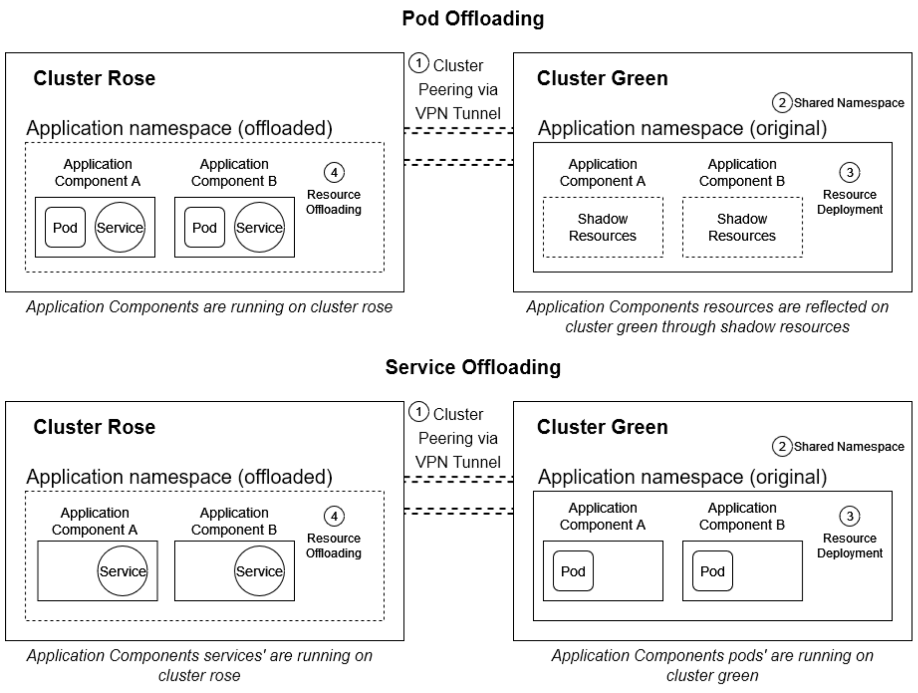

Pod Offloading
Cluster Rose
Application namespace (offloaded)
Application Component A
Application Component B
Pod
Service
4 Resource Offloading
1 Cluster Peering via VPN Tunnel
Cluster Green
2 Shared Namespace
Application namespace (original)
Shadow Resources
3 Resource Deployment
Application Components are running on cluster rose
Application Components resources are reflected on cluster green through shadow resources
Service Offloading
Application Components services' are running on cluster rose
Application Components pods' are running on cluster green

Figure 2.11: Pod offloading and Service Offloading [37]

2.4.3 Submariner

Submariner [5] is an open-source tool that allows connection across different clusters and is built to be compatible with any network plugin. Submariner's architecture relies on several components to achieve cluster connectivity as shown in Figure 2.12, highlighting the most relevant components:

- **Gateway Engines** responsible to manage the tunnels between clusters

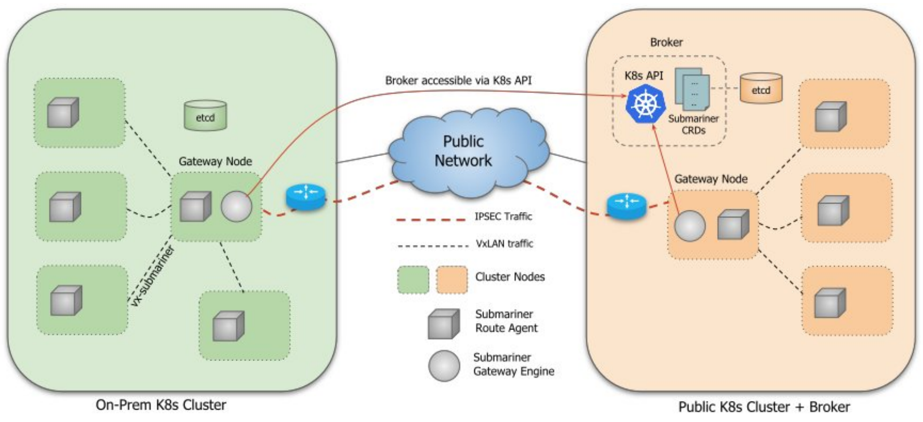


Figure 2.12: Submariner architecture [38]

- A **Broker**, which can be running on a dedicated cluster or on one of the connected clusters, enables Gateway Engines to discover one another

- **Route Agents** injected in each node for routing the traffic from the nodes to the gateway engine so the information can cross to the connected clusters

- **Service Discovery** providing DNS discovery of services across the connected clusters

The Lighthouse project is accountable for the DNS service discovery in the Submariner by deploying an agent in each of the connected clusters. DNS discovery is done by **Lighthouse** deploying an agent in every connected cluster, which then communicates with the broker to exchange information about the deployed services in all the clusters.

For every service in the local cluster for which a *ServiceExport* has been created, the Agent creates a corresponding *ServiceImport* resource and exports it to the Broker to be consumed by other clusters and for every *ServiceImport* resource in the Broker exported from another cluster, it creates a copy of it in the local cluster. This ends up having an overhead on the cluster because the amount of resources created is proportional to the number of existing services. Kubernetes *CoreDNS* is configured to forward requests to the Lighthouse external DNS server, which then uses the *ServiceImport* to resolve the DNS request.

An existing issue with Submariner is the fact it cannot handle overlapping CIDRs between clusters. In that case, we need to add another component to the architecture which is the **Globalnet Controller**.

This component consists of a global private virtual network (global CIDR) designed to support Submariner. Each cluster is assigned a subnet of the global private virtual network, which is used to provide virtual IPs to the pods and services of the cluster. New IP routing rules are created by the **IP Address Manager (IPAM)** to accommodate the new IPs. Globalnet is also supported by Lighthouse meaning the overlapping addresses will not affect the DNS resolving requests.

2.4.4 Cluster API

Cluster API is a Kubernetes project focused on providing declarative APIs and tooling to simplify provisioning, upgrading, and operating multiple Kubernetes clusters [39].

Cluster API uses Kubernetes-style APIs and patterns to automate cluster lifecycle management for platform operators. The supporting infrastructure, like virtual machines, networks, load balancers, VPC, and the Kubernetes cluster configuration are all defined in the same way that application developers operate deploying and managing their workloads (*.yaml* blueprints). This enables consistent and **repeatable cluster deployments** across a wide variety of infrastructure environments [39]. Figure 2.13 represents **Cluster API** topology.

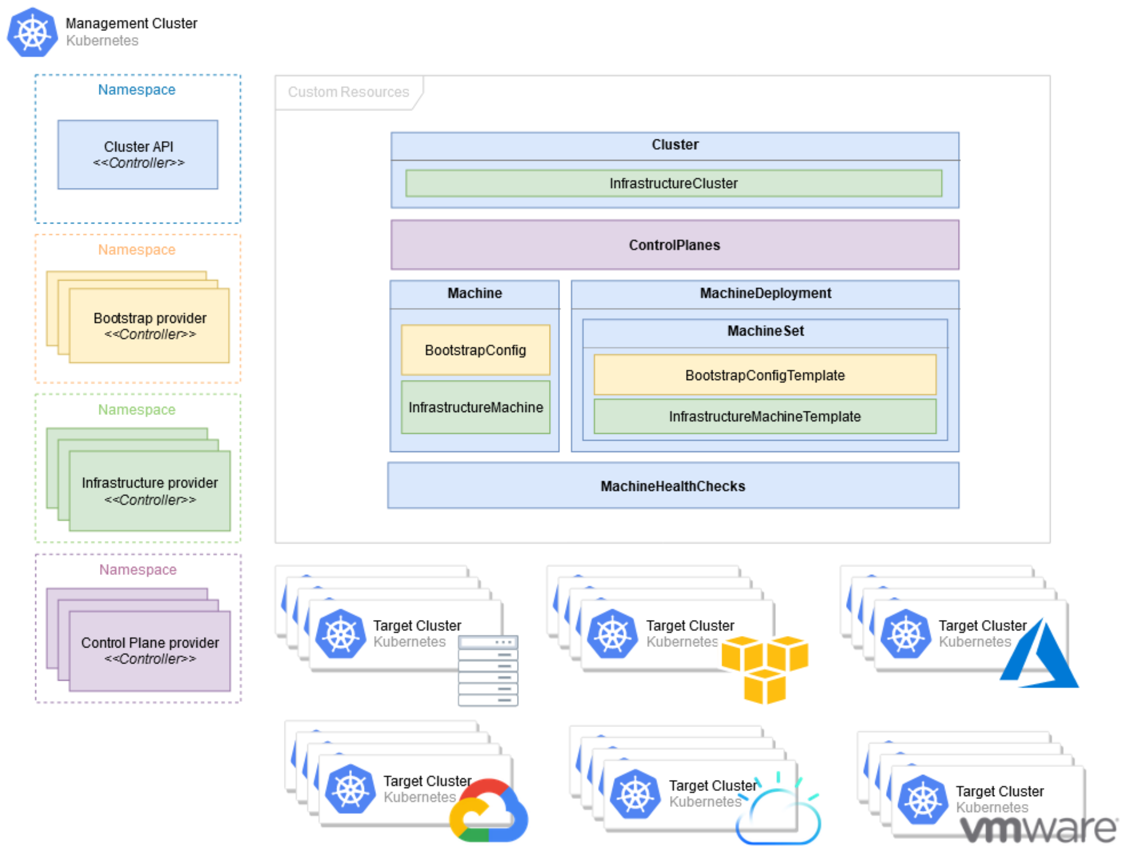


Figure 2.13: Cluster API topology [39]

Cluster API main components are briefly described below [40]:

- **Cluster API Provider** (*CAP*) is responsible for creating and managing the underlying infrastructure for a Kubernetes cluster. The *CAP* communicates with cloud providers or infrastructure providers, such as AWS, Azure, or vSphere, to provision and configure the necessary resources for deploying a cluster.
- **Cluster API Management** (*CAM*) is responsible for creating and managing the Kubernetes objects that make up a cluster. This includes objects such

as pods, services, and deployments. The *CAM* communicates with the Kubernetes API server to create and manage these objects.

- **Cluster API Bootstrap** (*CAB*) is responsible for bootstrapping new control plane nodes, which are the nodes that run the Kubernetes control plane components. This includes objects such as the API server, *etcd*, and the *controller-manager*.

Cluster API provides a set of Kubernetes CRDs that can be used to declaratively define a cluster and its components. These CRDs can be used to create, update, and delete clusters, as well as their underlying infrastructure and Kubernetes objects. Some CRDs are described below:

- **Cluster** CRD represents a Kubernetes cluster and contains information such as the number of nodes, the version of Kubernetes to use, and the cloud provider or infrastructure provider to use.
- **Machine** CRD represents a single node in a Kubernetes cluster and contains information such as the instance type, the image to use, and the network configuration.
- **MachineSet** CRD represents a group of machines that are created together and have the same configuration.
- **MachineDeployment** CRD represents a deployment of a set of machines.
- **ClusterDeployment** CRD represents a deployment of a cluster, it's a parent object of **MachineDeployment**.
- **ClusterConfig** CRD represents the configuration of a cluster and contains information such as the Kubernetes version and the network configuration.
- **ClusterStatus** CRD represents the status of a cluster and contains information such as the number of nodes that are up and running.

Cluster API is widely adopted to provide automation, standardization and portability of Kubernetes clusters across different cloud infrastructures, it works well with other popular kubernetes tools like *Kubernetes Operator*, *GitOps*, and *Helm*.

2.4.5 OpenStack

"**OpenStack** is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms" [41]. Being an IaaS, it also provides orchestration, fault management and service management along with other functionalities. Figure 2.14 shows an overview of the OpenStack architecture.

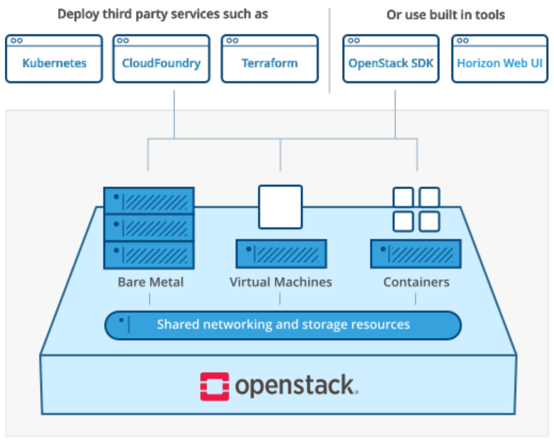


Figure 2.14: Overview of OpenStack architecture [41]

MicroStack is a single-machine, snap-deployed OpenStack cloud. Being a lighter version of OpenStack, it was built for testing workloads. It can also be used for IoT and appliances and for the Edge Clouds. The caveats of this build are the lack of services, as it only provides the main components: Nova, Keystone, Glance, Horizon and Neutron (with *OVN*) [42]. MicroStack was used in the third proof of concept (5.5) for its less demanding requirements and faster setup.

2.5 Summary

In this chapter, the key concepts about orchestration and automation of containers, virtual machines and clusters are introduced, as well as the cloud deployment topologies, to understand the multi-cluster architecture and its implementation better.

In the first section, the concept of orchestration and ETSI ZSM standard for orchestrators, as well as the resources that can be orchestrated. We explain the notion of TOSCA blueprints, which can be used in the orchestration process.

The second section is the notion of cloud computing and the services it can provide. The five different types of cloud deployment models are presented, and the concept of edge cloud computing is introduced as well as its relation with the multi-cluster architecture.

The third section introduces the multi-cluster architecture concept, which is a common architecture for cloud deployment models, and the different implementations are discussed. Multi-cluster advantages and challenges that led to the proof of concept scenarios (refer to Chapter 5) are also presented and discussed in this section.

In the last section, several tools such as Kubernetes, Submariner, Liqo and Cluster API are introduced, and their architecture and components are explained. The tools described in this section are all fundamental to the resolution of the multi-cluster challenges, also explained in this chapter. Furthermore, some of the tools in this section are used in the development of the proof of concepts, explained in

Chapter 5.

Chapter 3

Research Objectives and Approach

This chapter discusses the objectives and the proposed approach followed in this work. Section 3.1 provides a breakdown of the several enumerated objectives, explaining and contextualizing each one. Section 3.2 explains the outlined approach to accomplish the objectives.

3.1 Research Objectives

The main goal of this work was to develop an intelligent orchestration and management platform focused on distributed Cloud-Native environments. This means designing, implementing and evaluating a solution for dynamic Kubernetes cluster and application provisioning on distributed and edge-cloud domains. Such objective was divided into various objectives as described below:

1. **Objective 1 - Develop and evaluate an orchestration system**

   This objective is within the scope of the multi-cluster architecture challenges. The orchestration solution should comply with a list of requirements and meet the CHARITY goals. Moreover, the capabilities of such a solution should be tested and evaluated. This solution should be in line with the overall idea of bringing automation to the process of orchestrating a multi-domain infrastructure and applications. Namely, we aim to incorporate the concept of closed loops as discussed in the ZSM specification.

2. **Objective 2 - Integrate and validate cluster inter-connectivity tools**

   This specific objective derives from the multi-domain nature of the targeted environments. We aim to support cross-cluster connectivity that could allow a seamless interconnection of various application components spread over multiple clusters. We aim to research different cluster connectivity technologies, evaluate their capabilities and compare them to decide which one to use in the final architecture.

3. **Objective 3 - Integrate and validate multi-cluster orchestration tools**

   Similar to the previous one, with this objective, we aim to investigate the use of declarative APIs and tooling for bootstrapping and orchestration of Kubernetes-based clusters. For objectives 2 and 3, we also aim to validate both technologies within the scope of CHARITY use cases and demonstration activities.

3.2 Proposed Approach

Following the state-of-the-art review about orchestration, automation focused on clusters and *Kubernetes*, we started by defining the work plan (c.f. Chapter 4). This included the list of risks and countermeasures. Later, we conducted a series of tests and developed early proof-of-concepts to familiarise ourselves with and better assess the technologies involved. Such exploratory work is reflected in Chapter 5. Then, we elicited a list of requirements based on CHARITY project discussions and needs. Afterwards, we defined a reference architecture (c.f. Chapter 6) to answer the defined goals and requirements. Similarly to the requirements, this architecture considers the overall CHARITY architecture and external components the proposed solution should communicate. Later, we proceed to implement such architecture. Last but not least, we tested and evaluated the capabilities of the implemented solution and conducted additional validation taking into consideration CHARITY project use cases (c.f. Chapter 7).

The mapping between each individual objective and the proposed approach is overviewed below.

3.2.1 Objective 1 - Develop and evaluate an orchestration system

The first objective consisted of developing an orchestrator to automate the lifecycle management of Kubernetes-based environments and applications. To achieve that, we designed a reference architecture for a solution capable of bootstrapping clusters automatically. Such a solution follows the OODA loop concept and will use a **Kubernetes Operator** as an auxiliary mechanism to help the orchestrator's automation process.

3.2.2 Objective 2 - Integrate and validate cluster inter-connectivity tools

After researching solutions regarding the multi-cluster architecture challenges, we considered two candidate tools: Submariner and Liqo. These tools were deemed the most promising tools to help resolve the cross-cluster connectivity challenge. Both needed to be tested and compared to decide which tool was to be integrated into the orchestrator. We develop a proof of concept for each of the technologies, better detailed in **Chapter 5**, in Sections 5.3 for **Submariner** and 5.4 for **Liqo**.

3.2.3 Objective 3 - Integrate and validate multi-cluster orchestration tools

Likewise, Cluster API was considered to be a promising solution to resolve the orchestration challenge. Cluster API is free and natively supported by Kubernetes. Moreover, it allows us to declarative define the infrastructure. We started to assess Cluster API capabilities as an orchestration tool and also evaluate the OpenStack compatibility as a provider, a proof of concept was developed, better detailed in Chapter 5, in Section 5.5, fulfilling the proposed objective.

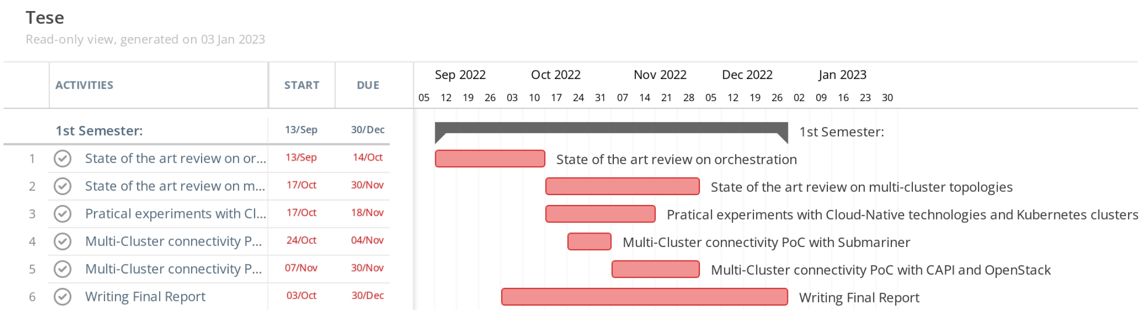
Chapter 4

Work Plan

This chapter overviews the internship work plan and compares the executed and planned work plans. The tasks of the plan are briefly explained as well as the risks.

4.1 First Semester

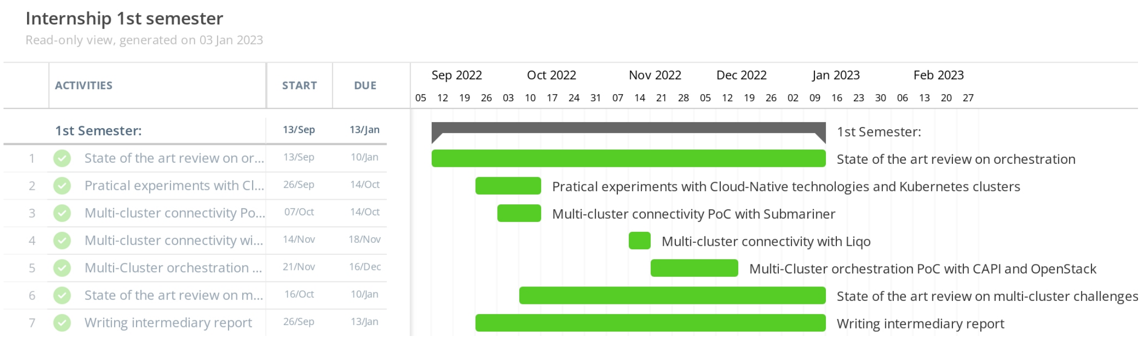
This section describes the work plan for the first semester of the internship. Figure 4.1 presents the expected workplan to be done in the *first semester* and Figure 4.2 presents the concluded work plan in the first semester.


Tese

Read-only view, generated on 03 Jan 2023

	ACTIVITIES	START	DUE
	1st Semester:	13/Sep	30/Dec
1	✓ State of the art review on or...	13/Sep	14/Oct
2	✓ State of the art review on m...	17/Oct	30/Nov
3	✓ Pratical experiments with Cl...	17/Oct	18/Nov
4	✓ Multi-Cluster connectivity P...	24/Oct	04/Nov
5	✓ Multi-Cluster connectivity P...	07/Nov	30/Nov
6	✓ Writing Final Report	03/Oct	30/Dec

Figure 4.1: 1st Semester Expected Work Plan


Internship 1st semester

Read-only view, generated on 03 Jan 2023

	ACTIVITIES	START	DUE
	1st Semester:	13/Sep	13/Jan
1	✓ State of the art review on or...	13/Sep	10/Jan
2	✓ Pratical experiments with Cl...	26/Sep	14/Oct
3	✓ Multi-cluster connectivity Po...	07/Oct	14/Oct
4	✓ Multi-cluster connectivity wi...	14/Nov	18/Nov
5	✓ Multi-Cluster orchestration ...	21/Nov	16/Dec
6	✓ State of the art review on m...	16/Oct	10/Jan
7	✓ Writing intermediary report	26/Sep	13/Jan

Sep 2022
Oct 2022
Nov 2022
Dec 2022
Jan 2023
Feb 2023
05 12 19 26 03 10 17 24 31 07 14 21 28 05 12 19 26 02 09 16 23 30 06 13 20 27
1st Semester:
State of the art review on orchestration
Pratical experiments with Cloud-Native technologies and Kubernetes clusters
Multi-cluster connectivity PoC with Submariner
Multi-cluster connectivity with Liqo
Multi-Cluster orchestration PoC with CAPI and OpenStack
State of the art review on multi-cluster challenges
Writing intermediary report

Figure 4.2: 1st Semester Executed Work Plan

During the research phase of the internship, a **new promising technology** regarding the cluster connectivity challenge was found. **Liqo** promised to be more powerful and more versatile than **Submariner**, which was proven in the proof of concept scenarios 5.

A brief description of the tasks concluded in first semester is given below:

- **Task 1 - State of the art review on orchestration:** Research about the concepts of orchestration and related technologies.
- **Task 2 - Practical experiments with Cloud-Native technologies and Kubernetes clusters:** Experimentation with the technologies to gain experience and develop more insightful thoughts on the current research subjects.
- **Task 3 - Multi-Cluster connectivity PoC with Submariner:** Proof of concept on multi-cluster connectivity using the **Submariner** technology.
- **Task 4 - Multi-Cluster connectivity PoC with Liqo:** Proof of concept on multi-cluster connectivity using the **Liqo** technology.
- **Task 5 - Multi-Cluster connectivity PoC with CAPI and OpenStack:** Proof of concept on multi-cluster orchestration and management using the **Cluster API** framework with **OpenStack** cloud provider.
- **Task 6 - State of the art review on multi-cluster challenges:** Research about multi-cluster topologies, its challenges and related technologies to solve them.
- **Task 7 - Writing intermediary report:** Writing the report on the research and development of the subjects conducted during the first semester of the internship.

The schedule was extended and tasks delayed, as during the writing of this report and development of the proof of concepts, other courses and projects of the masters were being worked on in parallel.

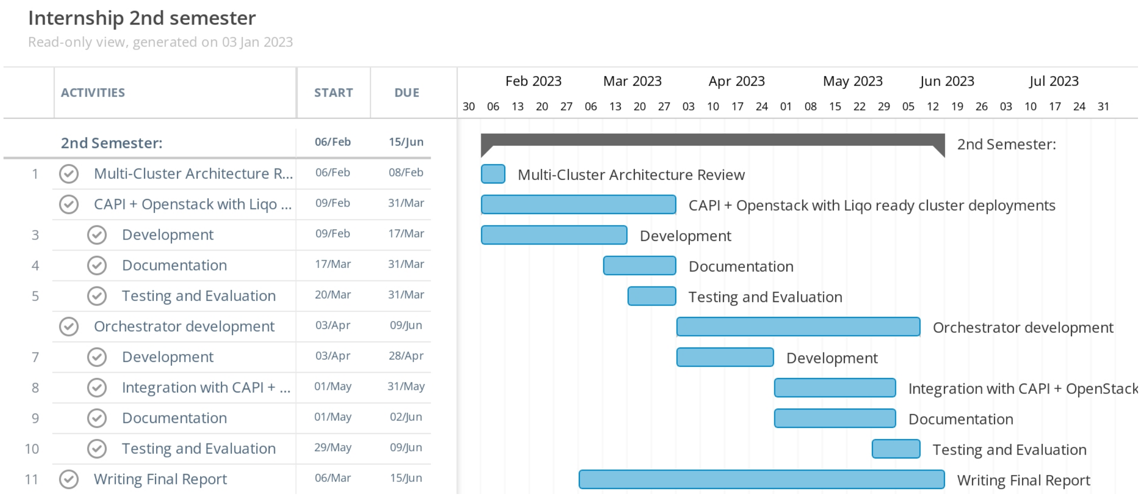
4.2 Second Semester

This section describes the work plan for the second semester of the internship.

Figure 4.3 presents the second semester expected work plan and Figure 4.4 presents the second semester executed work plan.

A brief description of the tasks concluded in the second semester is given below:

- **Task 1 - Multi-Cluster architecture review:** Review small details of the final architecture.

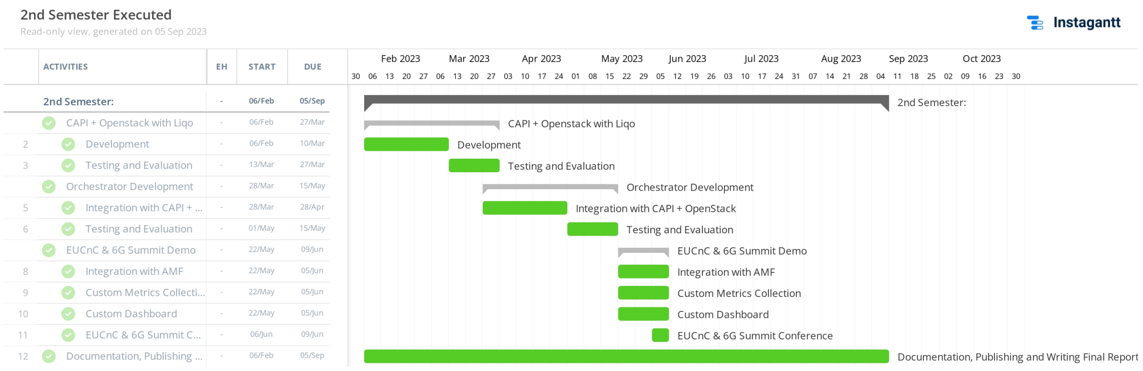


Internship 2nd semester

Read-only view, generated on 03 Jan 2023

	ACTIVITIES	START	DUE
	2nd Semester:	06/Feb	15/Jun
1	✓ Multi-Cluster Architecture R...	06/Feb	08/Feb
	✓ CAPI + Openstack with Liqo ...	09/Feb	31/Mar
3	✓ Development	09/Feb	17/Mar
4	✓ Documentation	17/Mar	31/Mar
5	✓ Testing and Evaluation	20/Mar	31/Mar
	✓ Orchestrator development	03/Apr	09/Jun
7	✓ Development	03/Apr	28/Apr
8	✓ Integration with CAPI + ...	01/May	31/May
9	✓ Documentation	01/May	02/Jun
10	✓ Testing and Evaluation	29/May	09/Jun
11	✓ Writing Final Report	06/Mar	15/Jun

Figure 4.3: 2nd Semester Expected Work Plan



2nd Semester Executed

Read-only view, generated on 05 Sep 2023

	ACTIVITIES	EH	START	DUE
	2nd Semester:	-	06/Feb	05/Sep
	✓ CAPI + Openstack with Liqo	-	06/Feb	27/Mar
2	✓ Development	-	06/Feb	10/Mar
3	✓ Testing and Evaluation	-	13/Mar	27/Mar
	✓ Orchestrator Development	-	28/Mar	15/May
5	✓ Integration with CAPI + ...	-	28/Mar	28/Apr
6	✓ Testing and Evaluation	-	01/May	15/May
	✓ EUCnC & 6G Summit Demo	-	22/May	09/Jun
8	✓ Integration with AMF	-	22/May	05/Jun
9	✓ Custom Metrics Collecti...	-	22/May	05/Jun
10	✓ Custom Dashboard	-	22/May	05/Jun
11	✓ EUCnC & 6G Summit C...	-	06/Jun	09/Jun
12	✓ Documentation, Publishing ...	-	06/Feb	05/Sep

Figure 4.4: 2nd Semester Executed Work Plan

- **Task 2 - CAPI + OpenStack with Liqo ready cluster deployments:** Prepare Cluster API to deploy clusters with *Liqo*, document, test and validate the architecture.

- **Task 3 - Orchestrator development:** Development of the orchestrator that will be integrated with CAPI and OpenStack, document, test and validate the orchestrator, and integrate it with the Cluster API + OpenStack architecture.

- **Task 4 - Writing final report:** Writing the report on the research and development of the subjects conducted during the internship, which will contribute to CHARITY documentation.

As seen in the executed plan, new tasks were added or updated. A list of the new and changed tasks is presented below:

- **Task 1 (Deleted) - Multi-Cluster architecture review:** This task is included in the task 2 - CAPI + OpenStack with Liqo.

- **Task 4 (Updated) - Documentation, Publishing and Writing Final Report:** Writing the report on the research and development of the subjects conducted during the internship, contributing to CHARITY documentation (deliverables) and publishing of a conference paper and a journal paper.

- **Task 5 (New) - EUCnC & 6G Summit Demo** - Integration of the orchestrator with AMF, as well as adding a custom metrics system and a custom dashboard, readying the orchestration system for a live demo at the EUCnC booth exhibition.

The testing and validation of the orchestrator, which included a live demonstration at EUCnC & 6G Summit Conference, took longer than expected, leading to a small delay. In turn, we also took the opportunity to improve the quality of documentation and final report. Additionally, we also devoted significant efforts to presenting the achievements of this work in the form of scientific publications.

4.3 Risk Assessment

This section describes the risks that may hinder the progress of this work. The list of risks and their mitigation plans for each one are described below:

- **Risk 1:** Implementation challenges due to lack of experience with the tools and frameworks.

  Mitigation Plan: Research documentation and tutorials, practising with the tools and frameworks. Develop and test initial prototypes to better assess the time and complexity of each development task.

- **Risk 2:** Being too ambitious with the work, leading to features not being implemented.

  Mitigation Plan: Outline the envisioned approach, including the goals, scope, work plan, and risk assessment. Moreover, define a limited list of requirements and establish priorities for each.

- **Risk 3:** Changes in tasks and priorities derived from the CHARITY and OneSource's goals.

  Mitigation Plan: Follow an agile-based approach with small iterations, regular discussions and more frequent deliveries.

We were faced with Risk 1. To overcome that, we devoted quite a significant amount of effort to early testing, experiments and PoC as presented in Chapter 5. Similarly, the underlying concept behind this work was concluded to be quite ambitious. For the sake of this work, we had to define small, more concrete and manageable goals and requirements that were pivotal to successfully achieving the overall idea of having an orchestration solution. Moreover, at the halfway point of this work, we revisited and better detailed the planned tasks and respective times.

Chapter 5

First Testing and Proof of Concepts

This chapter covers the proof of concept scenarios regarding the multi-cluster architecture challenges. The proof of concepts described in this chapter were developed with the goal of better understanding the technologies and validating their capabilities.

Scenario I and **Scenario II** are possible solutions for the same challenge, using **Submariner** and **Liqo** respectively, comparing both to decide what technology better fits the purposes of this work. **Scenario III** is related to the multi-cluster architecture orchestration challenge, and was developed to assess the extent of **Cluster API** capabilities and functionalities, as well as evaluate the **OpenStack** cloud provider compatibility with **Cluster API**.

5.1 Methodology

In this section, we explain the methodology and common approaches taken for the scenarios in general, also during this section and the following sections, where the scenarios are explained in detail, **a cluster is considered to be a Kubernetes cluster**.

Every cluster represented in each scenario is a single node cluster, in which the only node running is the **control plane node**. This is relevant to the diagrams of each scenario.

The first two scenarios regarding cluster connectivity were developed with the same goal in mind, connecting more than one cluster. To validate this goal, we resort to a simple and common approach used to test single Kubernetes clusters setups, and adapt to the multi-cluster setup, which is using a *sleep* pod to *curl* a *httpbin* pod.

The *httpbin* pod runs a container with a simple HTTP request and response service and the *sleep* pod runs a container that allows us to execute *bash* commands like the *curl* command needed to test the connection. For the cluster connectivity scenarios, the *sleep* pod is deployed in one cluster and the *httpbin* pod is deployed in the opposite cluster.

All the clusters in scenario I and II except one, were deployed using **Kubeadm** bootstrap (for more details refer to Subsection 5.2). The exception cluster was deployed with **KinD** which is also a tool to build Kubernetes clusters. This exception was due to the fact that this cluster was **running in an existing setup of a member of the CHARITY team**.

In the **Submariner** scenario, we opted for using **Calico** and for the **Liqo** scenario we opted for **Flannel**. The change in CNI plugins was due to the fact that both of them had caveats with Calico and some additional attention is needed regarding the setup. As we were not using the additional features provided by **Calico**, we decided to change to Flannel as it is a simpler and lighter solution.

For a functional setup, we also need a load balancing solution, where the chosen was the installation of **MetalLB** [43], which is solution for *Kubernetes* that monitors for services with the type *LoadBalancer* and assigns them an IP address from a virtual pool. For more detail regarding MetalLB, refer to Subsection 5.2.

For the third scenario, we only need to prepare one cluster that will assume the role as the **management cluster**. The management cluster is deployed with **Kubeadm** and we use the **Flannel** CNI plugin, both detailed in Subsection 5.2.

All the scenarios each have a section dedicated to them in this chapter, to better explain them.

5.2 Deployment Tools and Solutions Explained

This section lists some of the chosen deployment tools and solutions that help build the scenarios' setup, better detailing each one.

- **Kubeadm:** Kubeadm [44] performs the actions necessary to get a minimum viable cluster up and running. It is a commonly known tool to deploy a *vanilla* Kubernetes cluster, as it does not install any addons, like monitoring solutions, load balancer and even a CNI plugin. Kubeadm is usually used by more experient users, being more easily customized as it does not install nothing besides the necessary components. Less experient users often opt for **K3S** [45], **Minikube** [46] and **KinD** [47] as it already installs plugins and add-ons by default, which provides the user with a deployment-ready cluster.

- **KinD:** KinD [47] is a tool for running local Kubernetes clusters using Docker container "nodes" and it was designed primarily for testing Kubernetes itself. Despite bootstrapping the cluster with **Kubeadm**, it also installs addons like the CNI plugin and it only supports the Docker CRI. Regarding the CNI, it uses **Kindnetd**.

  Kindnetd is a simple CNI solution that fulfills the two main CNI requirements, **reachability and connectivity**. As it is a simple plugin, it only works on simple network environments, where all the cluster nodes belong to the same subnet.

- **Calico:** Calico [48] is a CNI plugin that provides a highly-scalable networking and network security solution that supports fine-grained network segmentation, traffic shaping, and network policy enforcement. **Calico** is built on top of the BGP routing protocol and uses IP-in-IP encapsulation to provide network connectivity between pods. When a pod is created, Calico assigns an IP address to the pod and programs the host's networking stack to route traffic to the pod. It allows for inter-node communication by creating a virtual network that spans across all nodes in the cluster. This virtual network is created by using BGP to advertise the IP addresses of pods running on each node to all other nodes in the cluster. This allows pods running on different nodes to communicate with each other, even if they are not in the same subnet. **Calico** CNI supports a rich set of network policies that can be used to control network traffic between pods, services, and namespaces. It also has built-in support for network segmentation and isolation, which can be used to restrict access to specific pods or services.

- **Flannel:** Flannel [48] is the most popular CNI plugin for Kubernetes. When a pod is created, the Flannel CNI plugin assigns an IP address to the pod from the host's subnet. This IP address is then used for communication between pods and services within the Kubernetes cluster. **Flannel** CNI also allows for inter-node communication by using an overlay network. This overlay network allows pods running on different nodes to communicate with each other, even if they are not in the same subnet. This is done by using a Software-Defined Networking (SDN) technology called *VXLAN* to create a virtual network that spans across all nodes in the cluster. **Flannel is focused on networking, unlike Calico which is focused on network policies.**

- **MetalLB:** MetalLB [43] is installed on top of the cluster and provides a load-balancer implementation to the network. It allows the creation of *LoadBalancer* type services in clusters that do not have an alternative solution, like a **cloud provider load balancer**. The two main features it provides are **address allocation**, which allows IP address allocation to services using *IP Pools* previously defined and **external announcement**, which announces the allocated IPs beyond the cluster.

- **NGINX Controller:** NGINX [49] is a popular open-source web server that is often used as a reverse proxy and load balancer. In the context of *Kubernetes*, NGINX is commonly used as an ingress controller, which means that it acts as the entry point for incoming traffic to your *Kubernetes* cluster.

  When NGINX is deployed as an ingress controller in a Kubernetes cluster, it is responsible for routing incoming requests to the appropriate backend services based on the URL paths and hostnames defined in the ingress rules and can also perform load balancing across multiple backend instances of a service. In addition to its role as an ingress controller, NGINX can also be used as a sidecar container alongside a web application container in a *Kubernetes* pod. The NGINX container can handle tasks such as caching, SSL termination, and serving static content, while the application container handles dynamic content.

- **FastAPI:** FastAPI [50] is a modern, high-performance web framework for building APIs (Application Programming Interfaces) with Python. It is designed to be fast, efficient, and easy to use. FastAPI leverages the asynchronous capabilities of Python through the use of the asyncio library, making it well-suited for high-performance applications that require concurrent and scalable processing.

- **TOSCA:** TOSCA [51] provides a standardized and portable approach for managing cloud applications throughout their lifecycle, from design and deployment to scaling and monitoring. By using TOSCA, organizations can achieve greater interoperability, flexibility, and automation in their cloud application deployments, facilitating seamless integration and management across different cloud environments and technologies.

  TOSCA is typically written in *.yaml* format. The *.yaml* file contains the specifications for the application's topology, including its components, relationships, properties, and their configurations. It describes how the various application components, such as software, containers, virtual machines, and networking resources, are structured and interconnected. It may also include definitions of policies, interfaces, workflows, and other metadata associated with the application. These elements define the desired behaviour and operations of the application during its lifecycle.

- **WireGuard:** WireGuard [52] is a modern and secure open-source virtual private network (VPN) protocol. It was designed with simplicity and efficiency in mind, aiming to provide a fast and reliable solution for creating secure network connections. Unlike traditional VPN protocols, such as *IPsec* and *OpenVPN*, **WireGuard** is known for its simplicity, ease of use, and minimalistic code base.

5.3 Scenario I - Cluster connectivity using Submariner

This section describes the scenario created to test cluster connectivity using *Submariner*, its goals and results.

The **goals of the scenario** are listed as the following:

- Evaluate **Submariner** architecture and functionalities
- Assess **Submariner** connectivity capabilities

Scenario Description and Procedure

Each cluster is running on a virtual machine deployed within **CloudSigma** cloud provider infrastructure with Kubernetes *v1.24* deployed with **Kubeadm** and *Calico* CNI and using *MetalLB v0.12.7*. Figure 5.1 represents the schema for *scenario I*.

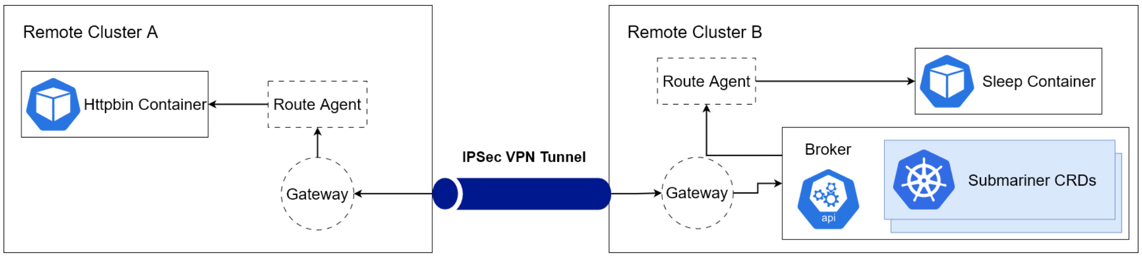


Figure 5.1: Scenario I - Cluster connectivity with Submariner

After setting up the clusters, we proceed with the installation of **Submariner**, following the official documentation [5].

As the clusters CIDR overlapped, we needed to enable *Globalnet*, so that **Submariner** would use a previously defined global IP pool to assign pods and services in all the clusters a unique IP address.

The broker was deployed to **Remote Cluster B** and then we proceeded with the cluster connection, which creates an **IPSec** VPN tunnel between the clusters. As we were using *Calico* CNI, we needed to use a different IP range for *Globalnet* because Calico uses the default values internally. We also needed to define *IPPools* resources in both clusters, each pool representing the joined clusters CIDRs.

Then, we deployed a pod containing *httpbin* to **Remote Cluster A** and a pod containing *sleep* to **Remote Cluster B**, exposed *httpbin* to the connected clusters and we were able to *curl* the service using the *sleep* pod successfully.

Results and Discussion

We were able to fulfil the goals of the scenario, as we were able to connect the two clusters, expose a service and access it through the opposing joined cluster. After the defined goals were achieved, was deployed a service to a node where the gateway service does not reside. The connection to it was not successful, demonstrating that in the current version of Submariner, traffic rerouting is not good enough.

5.4 Scenario II - Cluster Connectivity using Liqo

This section describes the scenario created to test cluster connectivity using *Liqo*, its goals and results.

The **goals of the scenario** are listed as the following:

- Evaluate **Liqo** architecture and functionalities
- Assess **Liqo** connectivity capabilities to connect to more than one cluster
- Assess **Liqo** offloading capabilities for workload reflection

Scenario Description and Procedure

The remote cluster is running on a virtual machine deployed within **CloudSigma** cloud provider infrastructure with Kubernetes *v1.24* and was deployed using **Kubeadm** with *Kube-Flannel* CNI and *MetalLB v0.13.5*.

The local cluster is running on a virtual machine deployed on a local workstation with the Kubernetes *v1.25* and was deployed using **KinD** with the default configuration using *kindnetd* as CNI. Figure 5.2 represents the schema for *scenario II*.

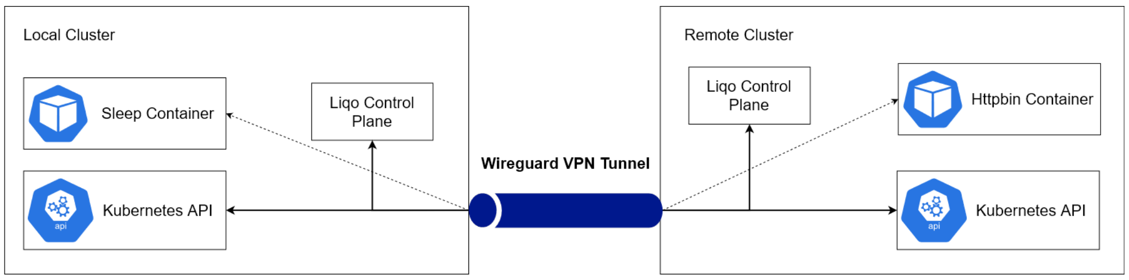


Figure 5.2: Scenario II - Liqo Connectivity

As **KinD** already provides a load balancing solution for cluster services using *NodePort* type resources, we do not need to install any additional software. **Kubeadm** however, needs a proper solution. We opted for **MetalLB** as it is a common solution for services load balancing and is briefly explained below.

After building the base setup, we proceeded with the installation of *Liqo* in each cluster using the official documentation as a guide[3]. As we were limited by the *IP* addresses in the remote cluster, we opted for the **in-band control plane** approach as it only needs one public *IP* address for the peering process. We peered the clusters successfully and connected them bidirectionally, allowing both inbound and outbound traffic from each cluster.

To test the connectivity between the clusters we deployed a pod in the local cluster with a **sleep container**, which will be used to send *curl* commands to another pod, deployed in the remote cluster, containing an **httpbin container** which is a simple *webpage*. After we **successfully verified the connectivity**, we also tested *Liqo* pod **offloading capabilities**, by offloading the pod containing *httpbin*, to the local cluster and then *curling* the *httpbin* pod.

Results and Discussion

With this scenario, we were able to validate both Liqo's connectivity and offloading capabilities. While not being a goal of this scenario, we also concluded that we could use Liqo's features independently of how we deployed the cluster and the version of Kubernetes used in each one.

We also proved that Liqo is CNI agnostic as using different CNIs had no impact at any point during the process (installation, peering and offloading). One caveat of the *in-band control plane* approach is the need to have access to both clusters to peer them, as we need access to both clusters' configuration files.

By comparing the **Liqo** scenario to the **Submariner** scenario, we decided to use **Liqo** in the planned architecture for the second semester, because **Liqo** offers the offloading feature which is a determining feature for distributed topologies and it is lighter than **Submariner**, being better prepared for clusters deployed on the edge of the network.

5.5 Scenario III - Cluster Provisioning using Cluster API with OpenStack

This section describes the scenario created to test cluster deployment with *Cluster API* with *Openstack*, its goals and results.

The **goals of the scenario** are listed as the following:

- Evaluate **Cluster API** architecture and functionalities
- Assess **Cluster API** deployment capabilities
- Assess compatibility with **OpenStack** provider
- Deploy a fully functional *Kubernetes* cluster

Scenario Description and Procedure

The **management cluster** is running on a virtual machine, deployed within **CloudSigma** cloud provider infrastructure with Kubernetes *v1.25* and is deployed using **Kubeadm** and *Kube-Flannel* CNI. Figure 5.3 represents the schema for *scenario II*.

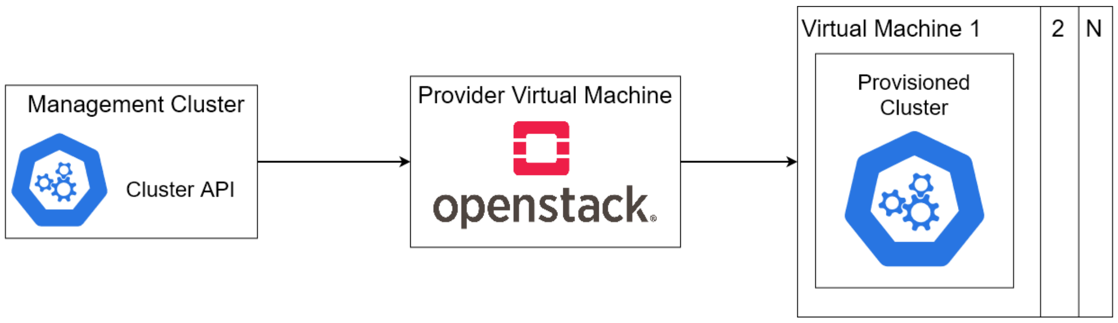


Figure 5.3: Scenario III - Cluster Provisioning

In the **management cluster**, we install **Cluster API** following the official documentation [7]. In the **provider virtual machine** we install **MicroStack**, a *lightweight* distribution of **OpenStack**, following the official documentation [42].

After the basic setup is complete, we deploy a simple virtual machine in **OpenStack**, using a *cirrOS* image, to check if everything is correctly configured. We also needed to define some specific configurations regarding OpenStack in Cluster API, such as the *access credentials* and the *environment variables*. We generated a cluster blueprint specific to the **OpenStack**, which is an *.yaml* file we can modify to our needs, but already comes with predefined parameters, using the previously mentioned *environment variables*. Then, we upload to the **OpenStack** provider a specific virtual machine image, which supports **Cluster API** deployments needed by **OpenStack** in order to create compatible virtual machine instances.

Proceeding, we are ready to deploy a virtual machine with the provided cluster, by applying the cluster blueprint as a normal resource of *Kubernetes*. Using the dashboard of **OpenStack**, we verify that the machine containing our cluster is running. To access the cluster, we retrieve the cluster configuration file, using the **Cluster API** command line, and use the configuration file to interact with the cluster. With the configuration file, we deploy a **CNI** plugin (Kube-Flannel), and the provisioned cluster is completely functional.

Results and Discussion

As expected, we were able to deploy virtual machines running *Kubernetes* clusters with **Cluster API** with the **OpenStack** provider. This was confirmed by checking the **Openstack** dashboard and by using the cluster configuration file (kubeconfig) to interact with the deployed clusters. We were also able to deploy a cluster and command it to download and install *Kube-Flannel* automatically using only the cluster blueprint. This was achieved by adding the needed *kubectl* commands to the cluster blueprint previously generated by **Cluster API**. Regarding the implementation, it took longer to implement than expected, mostly due to a lack of documentation.

5.6 Summary

The tests conducted during this work were pivotal for understanding which and how existing Open Source frameworks and technologies could be integrated into the envisioned orchestrator. Cluster API will be used to strengthen the orchestration features as it proved capable of using Kubernetes primitives for cluster management and provides support for different providers, while Liqo will be used for the cluster interconnectivity, as it proved superior to Submariner, as the latter lacked workload distribution features. Such an early prototype was also decisive for understanding the most relevant features to be automated and integrated within an orchestration system.

Chapter 6

Proposed Solution

This chapter documents the development of the orchestrator integrating the tools and frameworks related to automation and interconnectivity presented in the previous chapter. Section 6.1 gives a brief overview of the high-level requirements of the orchestrator. Section 6.2 presents the high-level architecture of the orchestrator and explains briefly each element composing the architecture. Section 6.3 explains each component composing the orchestrator individually, presenting the role and specific requirements of the component while also detailing the component functionalities. All the work presented in this chapter was showcased during the EUCnC & 6G Summit 2023 booth exhibition [53].

6.1 Requirements

This section gives an overview of the general requirements of this work. Based on the CHARITY project discussions and needs, these requirements were set to answer the previously defined objectives. We divided them into general requirements, which the orchestrator needs to fulfil and individual component requirements. Both requirements follow the *MoSCoW* scale.

Table 6.1 summarizes the requirements related to the main features of the orchestrator.

ID	Name	Description	MoSCoW	Achieved
CHA1	Create clusters	Create Kubernetes clusters	Must have	Y
CHA2	Delete clusters	Delete Kubernetes clusters	Must have	Y
CHA3	Update clusters	Update Kubernetes clusters	Should have	Y
CHA4	Deploy applications	Deploy applications in the Kubernetes clusters	Must have	Y
CHA5	Delete applications	Delete applications in the Kubernetes clusters	Must have	Y
CHA6	Connect clusters	Establish a network connection between clusters created via the orchestrator	Must have	Y
CHA7	Orchestrator components monitoring	Obtain metrics from the orchestrator components	Should have	Y
CHA8	Orchestrator deployed clusters monitoring	Obtain metrics from the Kubernetes clusters deployed by the orchestrator	Could have	Y
CHA9	Support for AMF	Add support for the AMF external component	Could have	Y
CHA10	Integration with Kafka	Integrate Kafka with the orchestrator, for true asynchronous communication	Could have	N

Table 6.1: General Requirements - Yes (Y), No (N)

These general requirements represent the key operations the orchestrator must be able to do. This includes create/scale/delete clusters and the deployment and deletion of applications and different resources, including container-based applications. Cluster connectivity is another key objective behind this work. The orchestrator should be able to distribute workloads across the different clusters. Additional capabilities, such as monitoring, were also identified but assigned as less priority considering the complexity of the key features and, at the same time, the time frame of this work.

6.2 Reference Architecture

This section presents the high-level proposed orchestration architecture, including some of the external components directly interfacing the main orchestrator system, which is the focus of this research work. This sections also introduces the orchestrator components, their role and functions.

The architecture is represented in Figure 6.1.

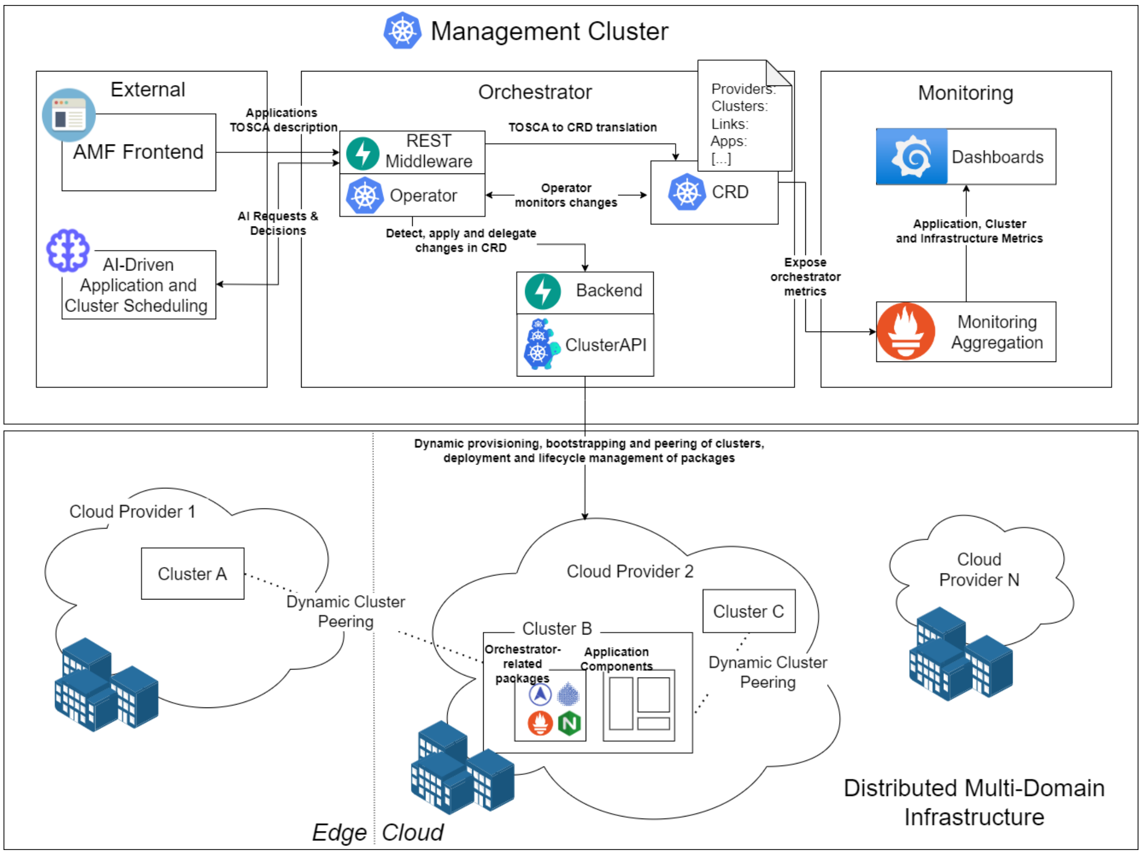

Management Cluster
External
AMF Frontend
AI-Driven Application and Cluster Scheduling
Applications TOSCA description
AI Requests & Decisions
Orchestrator
REST Middleware
Operator
TOSCA to CRD translation
Operator monitors changes
Detect, apply and delegate changes in CRD
CRD
Providers: Clusters: Links: Apps: [...]
Backend
ClusterAPI
Expose orchestrator metrics
Monitoring
Dashboards
Application, Cluster and Infrastructure Metrics
Monitoring Aggregation
Dynamic provisioning, bootstrapping and peering of clusters, deployment and lifecycle management of packages
Cloud Provider 1
Cluster A
Dynamic Cluster Peering
Cloud Provider 2
Cluster B
Orchestrator-related packages
Application Components
Cluster C
Dynamic Cluster Peering
Cloud Provider N
Distributed Multi-Domain Infrastructure
Edge
Cloud

Figure 6.1: Reference Architecture

A brief description of the components composing the architecture is given below:

- **AMF Frontend:** Represents the user interface for CICD admins that will be used to interact with the system. Through the AMF, the user can design their application that is translated into a TOSCA blueprint (refer to Section 5.2 for more details on the TOSCA blueprint). This is an external component that interacts with the orchestrator via a REST API.

- **AI-Driven Application and Cluster Scheduling:** The AI component, based on the information of the infrastructure and the requirements of the application to be deployed, decides what is to be demanded of the orchestrator, such as cluster creation, scaling and linking. This is then translated to the CRD monitored by the operator. This is an external component that interacts with the orchestrator via a REST API.

- **Custom Resource Definition:** The CRD contains the most up-to-date information regarding the status of the infrastructure.

- **Operator:** The operator is responsible for monitoring the CRD and handling its changes (i.e., insert, delete and update). When a change to the CRD is detected, the operator acts accordingly. For instance, adding a cluster definition to the CRD will trigger a new cluster provisioning as detailed in Section 6.3.2. The operator was inspired by the ETSI ZSM standard and closed-loop concept as a way to orchestrate and automate a Kubernetes-based environment.

- **Backend:** The backend is responsible for executing the changes to the infrastructure detected by the *operator*. The backend is complemented by a REST API that the *operator* uses to request the changes detected, and the backend uses the data received to translate the changes (e.g., the operator requests the creation of a cluster, sending the data collected from the CRD, the backend uses the data to create the cluster in the infrastructure and reports the feedback). More details can be found in Section 6.3.3.

- **Monitoring Aggregation:** It is responsible for collecting metrics exposed by the different components composing the *orchestrator*, as well as infrastructure metrics (e.g., number of clusters running, number of providers, number of applications).

- **Dashboards:** The dashboard allows visual feedback of the metrics collected by the *monitoring aggregation* component.

- **Distributed Multi-Domain Infrastructure:** The infrastructure relies on different cloud providers that will host the clusters for the applications. An application can be distributed or replicated across different clusters and cloud providers. The clusters can be connected across different providers, if the distribution of the application demands it, as to guarantee communication between the components comprising the application.

The orchestrator is prepared to be distributed across clusters, as each of the components composing the orchestrator is independent. For example, the backend could be in a different cluster than the operator and it will not affect the orchestration system's operation, as long as the clusters have access to each other. This independence is given through the developed middleware, acting as a bridge between the components. The orchestrator is also prepared to support different cloud providers and as such, the environments created by the orchestrator can be distributed across the supported providers.

A key idea behind the orchestrator and the CHARITY project consists of combining an external AI scheduler for deciding and predicting the resource allocation. This closes the gap in bringing intelligence to the orchestration process. The orchestrator can accommodate such components by providing a REST interface. Nevertheless, such integration and component is out of the scope of this thesis.

This work focus on the development of the orchestrator's main components (refer to Section 6.3): Custom Resource Definition (CRD), operator and backend, as well as integrating the monitoring components and the AMF external component.

6.3 Orchestrator Development

This section describes in detail each of the components that compose the orchestrator, where the focus of this work lies, including their requirements, roles and functionalities.

6.3.1 Custom Resource Definition

The proposed orchestrator system leverages the concept of a CRD to maintain the distributed infrastructure and keep a record of its status. Table 6.2 presents the requirements for the operator, for fulfilling the general requirements of the architecture(refer to Table 6.1).

ID	Name	Description	MoSCoW	Achieved
CR1	Syntax validation	Basic validation for CR fields	Must have	Y
CR2	Support for cluster data	Definition for cluster information fields	Must have	Y
CR3	Support for cluster links data	Definition for cluster interconnectivity links fields	Must have	Y
CR4	Support for application data	Definition for application information fields	Must have	Y

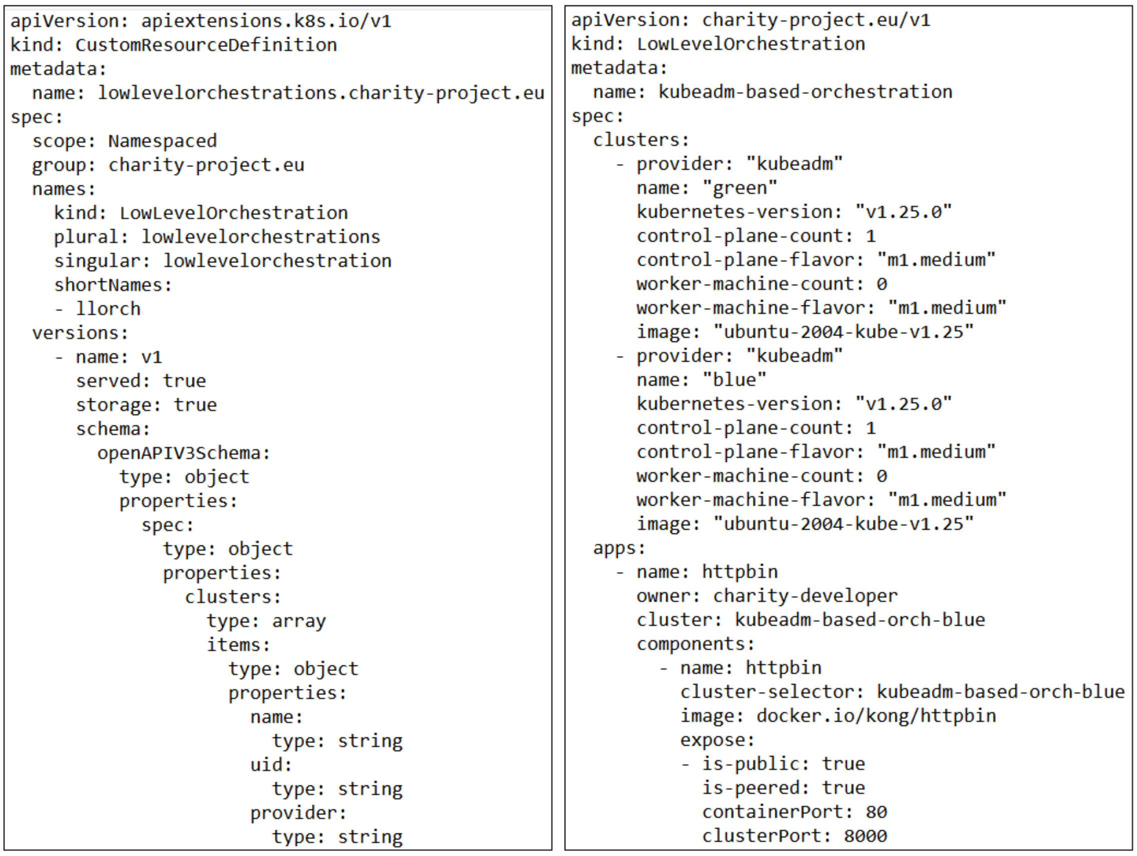
Table 6.2: CRD Requirements - Yes (Y), No (N)

The Custom Resource Definition specifies the structure and some validation rules the Custom Resource should follow, as a native *Kubernetes* resource. These resources are used together with a *Kubernetes operator*, which acts as a controller for the CRD and its CRs, detecting changes made to the resources. The orchestrator leverages a single Custom Resource is used to keep the registry of all the data related to the infrastructure.

The CR always has the up-to-date status of the infrastructure. If this is not verified, the operator, as the CRD controller, is always trying to translate the CR to the infrastructure, until the status is the same in both the infrastructure and CR. The CRD in the architecture 6.1 represents both the CRD and the CRs associated with the CRD represented in Figure 6.2.

The CRD is mainly divided in the following fields:

- **Clusters:** All the information regarding the clusters is registered within this section of the CRD. It is represented as a list of clusters and each cluster is treated as an individual, each containing its own information (e.g., each entry of a cluster in the CRD contains additional fields such as name, kubernetes-version, control-plane-count, control-plane-flavor, image)
- **Links:** This section contains the information regarding the clusters that are peered and connected. It is represented as a list with pairs of the names of connected clusters.
- **Apps:** This section contains the information regarding the applications deployed. It is represented of a list of applications and each application is treated as an individual, containing the following information (e.g., each entry of an application contains additional fields such as name, owner, cluster, components). As an application is considered a set of components, the components also have their own fields (e.g., name, cluster-selector, image, expose) as observed in Figure 6.2.



```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: lowlevelorchestrations.charity-project.eu
spec:
  scope: Namespaced
  group: charity-project.eu
  names:
    kind: LowLevelOrchestration
    plural: lowlevelorchestrations
    singular: lowlevelorchestration
    shortNames:
    - llorch
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                clusters:
                  type: array
                  items:
                    type: object
                    properties:
                      name:
                        type: string
                      uid:
                        type: string
                      provider:
                        type: string
```

```
apiVersion: charity-project.eu/v1
kind: LowLevelOrchestration
metadata:
  name: kubeadm-based-orchestration
spec:
  clusters:
    - provider: "kubeadm"
      name: "green"
      kubernetes-version: "v1.25.0"
      control-plane-count: 1
      control-plane-flavor: "m1.medium"
      worker-machine-count: 0
      worker-machine-flavor: "m1.medium"
      image: "ubuntu-2004-kube-v1.25"
    - provider: "kubeadm"
      name: "blue"
      kubernetes-version: "v1.25.0"
      control-plane-count: 1
      control-plane-flavor: "m1.medium"
      worker-machine-count: 0
      worker-machine-flavor: "m1.medium"
      image: "ubuntu-2004-kube-v1.25"
  apps:
    - name: httpbin
      owner: charity-developer
      cluster: kubeadm-based-orch-blue
      components:
        - name: httpbin
          cluster-selector: kubeadm-based-orch-blue
          image: docker.io/kong/httpbin
          expose:
          - is-public: true
            is-peered: true
            containerPort: 80
            clusterPort: 8000
```

Figure 6.2: Orchestrator Custom Resource Definition and Custom Resource Instance

The CRD is persistent across the *Kubernetes* cluster and the resources (CRs) created from it are also persistent but they are only persistent across namespaces as the CRs are a namespaced resource. This data persistency is inherited from *Kubernetes* as the all the native Kubernetes resources remain on the cluster/namespace as long as they are not explicitly deleted as shown in Figure 6.3.

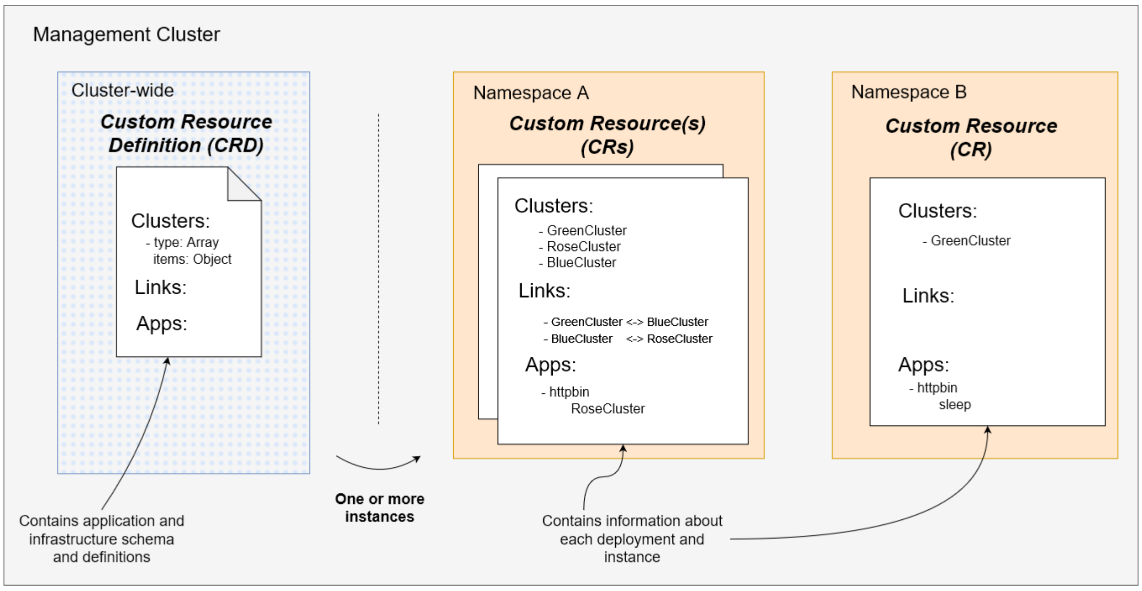


Figure 6.3: Orchestrator CRD and CR overview

This makes the CRs true data registries as they are sustained across node/cluster shutdowns, pod failures and recoveries, implementing a fault-tolerance mechanism native to *Kubernetes*. As the CRD has its own *Kubernetes operator* (described in the Section 6.3.2) managing the CRD and its custom resources lifecycle, if the cluster restarts for whatever reason, the cluster shuts down and restarts, when the operator resumes its normal behavior, it will read the CRD and CRs still remaining in the cluster as they are persistent (considering nothing was deleted manually), check for any requests to be sent to the backend still pending before the cluster restart, and if so, proceeds to handle the pending requests.

6.3.2 Operator

ID	Name	Description	MoSCoW	Achieved
OP1	Webhook syntax validation	Custom validation for CR fields	Could have	P
OP2	Detection of CRD and CR changes	Detect changes done to any of the orchestrator CRD and CR, ensuring constant monitoring	Must have	Y
OP3	Communication with the backend	Request backend endpoints to execute detected changes	Must have	Y
OP4	Create new CRs	Create CRs from scratch	Must have	Y
OP5	Update existing CRs	Update existing CR fields (e.g., count of worker/control-plane machines of clusters)	Must have	Y
OP6	Delete existing CRs	Delete the namespaced CRs	Must have	Y
OP7	Asynchronous communication with the backend	Communication with the backend is done asynchronously	Could have	N

Table 6.3: Operator Requirements - Yes (Y), No (N), Partial (P)

The operator is a containerized component running on the management cluster leveraging the concept of a *Kubernetes Operator*, with the role of monitoring changes to the Custom Resource created based on the CRD of the orchestrator. and request action from the *backend* component based on the detected changes, **acting as the controller of the CRD** The operator exposes a *REST* interface for handling outside requests(i.e., Application Management Framework, more details on the AMF in Chapter 7, Section 7.3). The *REST* interface is implemented with *FastAPI* [50].

Table 6.3 presents the requirements for the operator, for fulfilling the general requirements of the architecture.

The operator is constantly monitoring the CRD and its CRs, respecting the Observe, Oriente, Decide, Act (OODA) loop, represented in Figure 6.4. First, the operator is waiting for the monitored resources to change (**observe phase**). If any change is detected, the operator is responsible for admitting and validating the change, which consists of checking if the change is allowed and syntactically valid (**orient phase**). Based on the changes made to the resources, it detects what type of operation is needed (i.e. create, update, delete) and decides how to act based on the type of operation, outlining the execution plan (**decide phase**). Finally, based on the decision from the previous step, instead of executing the decided action himself, it requests action from the backend component, responsible for changing the infrastructure according to the data and demands received from the operator (**act phase**). This describes the recurring working process of the operator.

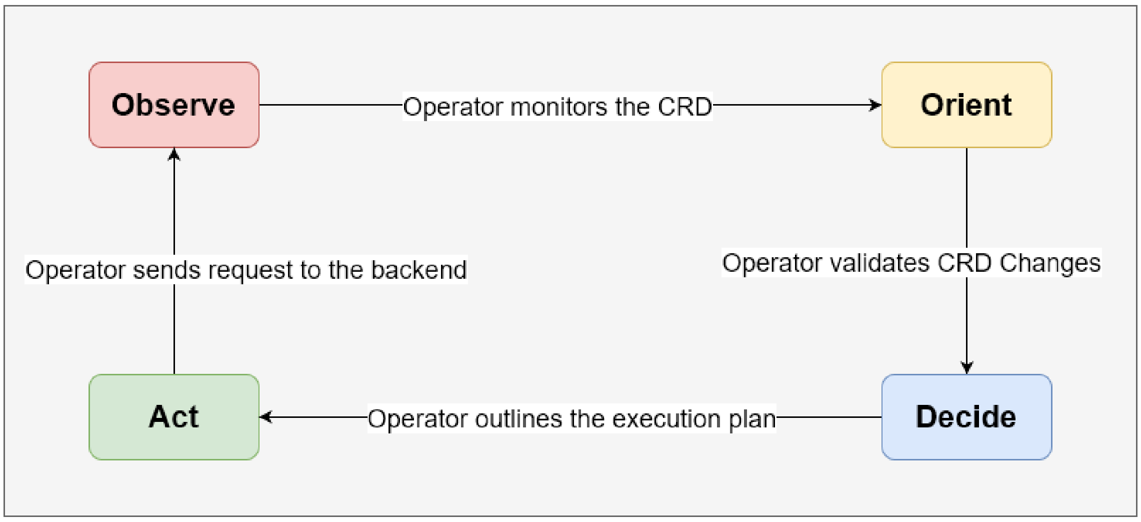

Observe
Operator monitors the CRD
Orient
Operator sends request to the backend
Operator validates CRD Changes
Act
Operator outlines the execution plan
Decide

Figure 6.4: Operator's OODA Loop

6.3.3 Backend

The backend is a containerized component running on the management cluster, which handles all the heavy-duty operations of the orchestration system involved in updating the infrastructure (i.e., when requested to create a cluster, the backend communicates with Cluster API and OpenStack to create said cluster). The backend exposes a *REST* interface used by the operator to request changes to the infrastructure. The *REST* interface is implemented with *FastAPI* [50]. To deliver most of the operations, the backend integrates with **Cluster API** [7], **Openstack** [54] and **Liqo** [3], researched and documented in the previous chapter (5).

Table 6.4 presents the requirements for the backend, for fulfilling the general requirements of the architecture.

The backend working cycle starts when the operator detects a change in the CRD, which also demands a change in the infrastructure. This triggers a request to

Table 6.4: Backend Requirements

ID	Name	Description	MoSCoW	Achieved
BE1	Load Providers List	Obtain the list of installed providers in CAPI	Must have	Y
BE2	Generate cluster manifest	Get the Kubernetes cluster CR from CAPI and adapt it, so it is ready for scheduling	Must have	Y
BE3	Schedule cluster creation	Schedule the Kubernetes cluster creation on Openstack	Must have	Y
BE4	Generate kubeconfig of deployed clusters	Get the cluster access files for the deployed Kubernetes clusters	Must have	Y
BE5	Install add-on packages	Install add-ons (NGINX, MetalLB, Prometheus, etc) for additional features and use-case support	Must have	Y
BE6	Peer clusters	Link created clusters using Liqo	Must have	Y
BE7	Install deployments	Transform CRD data into Kubernetes deployments	Must have	Y
BE8	Install services	Transform CRD data into Kubernetes services	Must have	Y
BE9	Install ingresses	Transform CRD data into Kubernetes ingresses	Could have	Y
BE10	Install docker secrets	Transform CRD data into Kubernetes docker secrets	Must have	Y
BE11	Install TLS secrets	Transform CRD data into Kubernetes TLS secrets	Could have	Y
BE12	Offload components	Offload individual components of applications to achieve a distributed multi-cluster environment	Must have	Y
BE13	Unoffload components	Unoffload individual components of applications	Could have	N
BE14	Unoffload applications	Unoffload application as a whole	Must have	Y
BE15	Delete components	Delete individual components of applications running on scheduled clusters	Should have	N
BE16	Delete applications	Delete applications running on scheduled clusters	Must have	Y
BE17	Scale scheduled clusters	Scale up/down nodes of scheduled clusters	Could have	Y
BE18	Associate floating IP to Virtual Machine	Assign a floating IP to the VM, so the cluster is accessible from the outside	Must have	Y
BE19	TOSCA Conversion to CRD	Convert TOSCA input to the orchestrator's CRD syntax	Should have	Y
BE20	List deployed applications	List deployed applications running on the scheduled clusters	Must have	Y
BE21	Schedule Cluster Deletion	Schedule and delete the Kubernetes cluster from Openstack	Must have	Y

the backend *REST* interface, making then the backend act accordingly (activating a functionality) to the request itself and data received within the request. This should be assumed for all the functionalities explained in more detail, further in this **Section**.

All the cluster operations made by the backend (i.e., create, update, delete, interact) integrate the *Cluster API* [7] framework, enabling more powerful and robust functionalities, and a better-managed infrastructure as the clusters and its resources (i.e., worker machines, control-plane machines) are stored as CRs and are independent per cloud provider (i.e., Openstack, AWS, BYOH) and per bootstrap provider (i.e., Kubeadm, MicroK8S, K3S). As default bootstrap provider and cloud provider for the functionalities explanation, found further in this **Section**, it should be assumed Kubeadm and Openstack, respectively.

Cluster Operations

The cluster creation process includes the creation of the VM, which will host the cluster and the bootstrapping of the cluster itself. The creation of a cluster starts by loading the correct bootstrap and cloud provider (e.g., Kubeadm and Openstack, respectively) from **Cluster API** based on the data received from the operator, which is also stored in the CRD. Following this, the backend runs a script that generates the cluster CR based on a CRD from **ClusterAPI**, and filled using the data (i.e., name, image, machine count) from the request. During the generation of the manifest, it is also added to **PostKubeadmCommands field** of the generated cluster CR, bash commands to download, install and configure the CNI of the cluster automatically. Although this makes the creation of the cluster slightly slower, it is rewarded with the creation of a cluster ready to use. After the generation of the cluster manifest, it is then applied using the kubernetes CLI. The management cluster is able to interpret the manifest as it is based on the **Cluster API** CRD. During the cluster deployment, after the host VM is created, it is also created a *floating IP*.

The floating IP is used by **Cluster API** to generate the cluster access file (*kubeconfig*). Although the creation of the floating IP is done automatically by **Cluster API** and **Openstack**, the assignment of this IP to the VM is not. The automation of this step is central for seamless cluster creation, so a solution was implemented. This assignment is done by checking if the VM is created, by checking the **Cluster API** CRs, and when it is created, it assigns the floating IP previously created to the host VM, **automating the floating IP assignment step**. Finally, the backend waits for the cluster to be ready which is when all the nodes of the cluster achieve a *READY* status.

The orchestrator supports the installation of add-on packages (i.e. Liqo, NGINX, MetalLB, Prometheus, Kafka) and these packages can be installed after the cluster is ready. Due to infrastructure limitations, these packages are all installed by default with every cluster with the exception of *Kafka* as it is a more resource-intensive add-on. A more detailed explanation of the behind-the-scenes process of the installation and configuration of each package can be found in Section 6.3.5. The cluster creation is truly finished when all the selected packages are installed.

After the cluster is created and ready, it is possible to scale the cluster. The orchestrator supports scaling the control plane nodes and the worker nodes, which can be scaled individually. The operator sends the cluster's name, the control plane node count and the worker node count saved within the CRD. As the control plane and worker machines are stored in individual CRs, the backend is capable of distinguish between them and checks for the resources individually. If the resources exist, the backend changes their replica count to the desired value, *Cluster API* detects changes to the CRs and acts accordingly, scaling the cluster nodes, up or down.

For cluster deletion, the operator sends the cluster's name registered in the CRD. As the cluster resources are uniquely named, the backend checks for the CR residing in the management cluster, and proceeds to delete the Kubernetes resource. Subsequently, this triggers the deletion of the cluster in the infrastructure.

Cluster Interconnectivity

Regarding cluster interconnectivity, the backend integrates *Liqo* [3][4] framework, which in addition to providing a solution to cluster interconnectivity, provides support to workload distribution across the connected clusters. The features implemented by the orchestrator that integrate with the *Liqo* framework, are also exposed by the backend to the operator via HTTP endpoints.

The starting point of the interconnectivity is the cluster peering. The backend receives the names of the clusters that should be peered with *Liqo* from the operator which are contained within the CRD. For the peering process, the backend does a series of checks, to make sure the peering can begin. The checks are done individually by cluster and only when both clusters pass all the checks, the peering process begins. First, the backend checks if the clusters are already deployed and ready. If this first check is verified, the backend generates the *.kubeconfig* files to access the clusters via the *Cluster API* CLI and perform the remaining checks. Accessing the cluster, the backend checks for the list of components composing the *Liqo* framework, verifying if they are ready and available.

After all the checks for both clusters are validated, the clusters are peered, using the *Liqo* CLI (installed in the backend). As it functions similarly to the Kubernetes CLI, the *.kubeconfig* files used to access the clusters can be inserted as a parameter and the clusters are peered. If the cluster checks are not valid, the peering process stops.

With peered clusters, it is possible to distribute components of an application, using Liqo's offloading capabilities (refer to Section 2.4.2).

Application Deployment

The orchestrator is prepared to deploy containerized applications in clusters hosted within the providers integrated with *Cluster API*. If the applications are composed of more than one component, the whole application can be deployed in a single cluster or distributed across different clusters.

The application deployment process begins with a request sent from the operator to the backend with the information regarding the application registered within the CRD. The backend translates the information received into native Kubernetes resources (i.e., deployment, services, ingresses) and deploys each component individually, as not all of the components need to be exposed through an ingress or even a service. It is also during this step that the backend distributes the components across different clusters using Liqo's offloading feature according to the information registered within the CRD. The clusters where the components of an application should be deployed are included in the information gathered from the CRD, as each component may have a different cluster associated. The backend uses *Kubernetes* labels and adds the name of the cluster as a label to the *Kubernetes* deployment resources of each component (the *Kubernetes* deployments are created from custom templates) with the name of the cluster associated to the component. As labels are native to Kubernetes, *Liqo* already knows how to leverage them during the offloading phase done by the backend using a bash command via the *Liqo* CLI, installed in the backend component.

As every application and its components are namespaced *Kubernetes* resources, when the deletion process is requested by the operator, the backend receives only the name of the application, which is used to identify the namespace where the application is deployed. Even if the application is distributed across clusters, as the identification of the namespace is kept across clusters, the backend unoffloads the namespace regardless of the application being distributed or not, as it does not affect the outcome or the performance of the process. After the unoffloading, the backend deletes the namespace and all the *Kubernetes* resources within.

6.3.4 Cluster and Application Monitoring

The orchestrator supports a monitoring system via a *Prometheus* server running in the management cluster, which collects the metrics exposed by the orchestrator, as well as resources deployed by it (i.e., clusters, applications, links) if Prometheus orchestration package (refer to Section 6.3.5) is installed within the clusters deployed via the orchestrator. All the metrics collected by the *Prometheus* server can be checked through a custom *Grafana* dashboard, as represented in Figure 6.5.

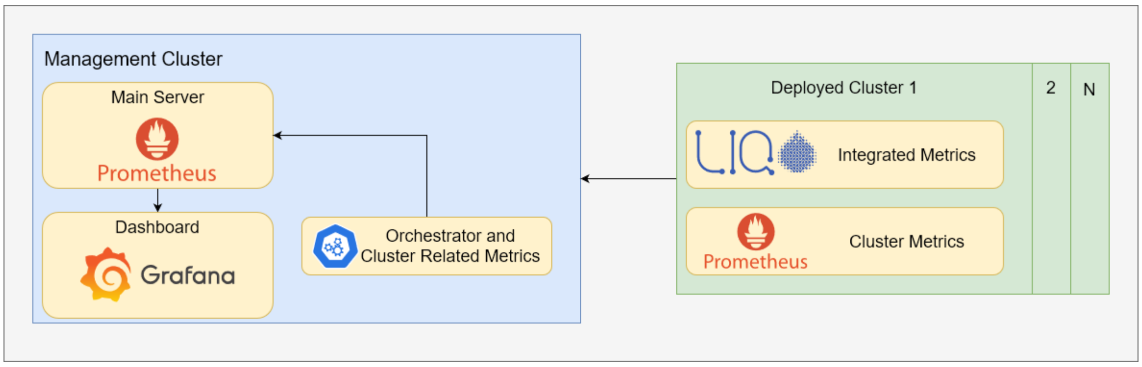


Figure 6.5: Orchestration Monitoring

The operator runs a routine as soon as it starts and every time a change is made to the CRD. The routine consists of checking the CRD and calculating the wanted metrics (i.e., number of clusters, number of applications, number of providers, number of application components). As the CRD contains the most up-to-date information regarding the infrastructure, the metrics are the most accurate. The operator exposes an endpoint, which is registered in *Prometheus* server of the management cluster, and the metrics can be visualized through a dashboard, as the *Prometheus* server is registered in *Grafana* as a data source.

The orchestrator also integrates the metrics exposed through *Liqo* (for more details on this process, refer to Section 6.3.5) and also leverages a *Prometheus* deployment if the monitoring orchestration package is deployed with the clusters.

The metrics collected by the monitoring package include most of the cluster-related metrics (e.g., CPU usage, RAM usage, disk usage, number of pods, number of nodes). Many of the collected metrics can be filtered by node and namespace. The metrics collected by *Liqo* include more specific information regarding the connectivity between the clusters (e.g., latency between the clusters, cross-cluster throughput, and number of connected clusters). All the metrics can be checked via a custom dashboard in *Grafana*, represented in figure 6.6.

6.3.5 Orchestrator Related Packages

This section details the integration of the orchestration-related packages with the clusters deployed with the orchestrator. The integration also involves the overlapping of said packages, as they need to work together and share limited resources (i.e., IPs, network ports).

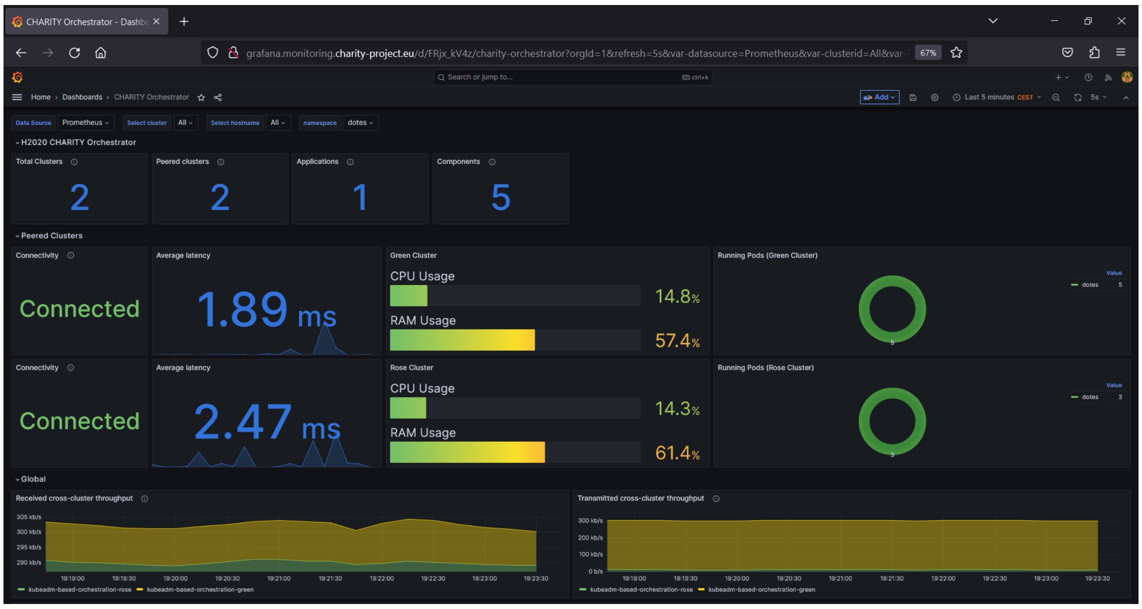

CHARITY Orchestrator - Dashb
grafana.monitoring.charity-project.eu/d/FRjx_kV4z/charity-orchestrator?orgId=1&refresh=5s&var-datasource=Prometheus&var-clusterid=All&var
Home › Dashboards › CHARITY Orchestrator
Last 5 minutes CEST
Data Source Prometheus
Select cluster All
Select hostname All
namespace dotes
H2020 CHARITY Orchestrator
Total Clusters 2
Peered clusters 2
Applications 1
Components 5
Peered Clusters
Connectivity
Connected
Average latency
1.89 ms
Green Cluster
CPU Usage 14.8%
RAM Usage 57.4%
Running Pods (Green Cluster)
Connectivity
Connected
Average latency
2.47 ms
Rose Cluster
CPU Usage 14.3%
RAM Usage 61.4%
Running Pods (Rose Cluster)
Global
Received cross-cluster throughput
Transmitted cross-cluster throughput
kubeadm-based-orchestration-rose
kubeadm-based-orchestration-green

Figure 6.6: Custom Monitoring Dashboard

The orchestration-related packages are **optional** software packages which bring additional functionalities to the deployed clusters. Depending on the type of infrastructure, the orchestration-related packages may deem unnecessary as they consume more resources of the cluster (i.e., if the cluster is located on the Edge, which usually is more constrained in terms of resources, some orchestrator-related packages may be discarded). Each add-on can be individually selected, installed, and is ready to work simultaneously with the other supported orchestrator-related packages.

The following list presents the currently supported packages by the orchestrator, as well as their role/function:

- **LoadBalancer:** MetalLB
- **Ingress controller:** NGINX
- **Cluster interconnectivity:** Liqo
- **Monitoring:** Prometheus
- **Asynchronous communication:** Kafka

For a brief overview on some of the packages, please refer to Chapter 5 Section 5.2.

The orchestrator not only installs but also configures the orchestration-related packages, fully automating the process. Each package has its own details and requires a configuration tailored to it. The orchestrator contains **bash scripts** dedicated for each package, with additional configuration files if needed (e.g., .yaml pre-configured templates of Kubernetes resources). The backend component of the orchestrator is responsible for executing the **bash scripts** and passing needed environment variables to the correct clusters.

As the orchestration-related packages need to work together, some added care was needed and applied to ease the overlapping of the packages, which is explained below:

IP/Port Sharing

As we are limited regarding public IPs to only one public IP, all the orchestration-related packages and applications deployed within the orchestration system need to leverage this single IP and provide working services. This is IP sharing is achieved by creating an *IPPool* Kubernetes resource supported by the *MetalLB* package. The *IPPool* contains the single public IP and it allows us to share the IP between *Kubernetes* services natively by recurring to *Kubernetes* annotations. In the orchestration-related packages, this is taken into account and the *Kubernetes* services of the packages are annotated automatically as needed. Although the IP can be shared, two services cannot be exposed through the same IP and the same network port. During the installation of the orchestration-related packages, the default values of network ports of some services are overridden to avoid conflicts.

Cluster DNS

To expose a service in a particular namespace through an ingress using the *NGINX* package, the clusters' DNS needs to be configured properly. The configuration allows the translation of *hostnames* into FQDN service names.

This is achieved by editing the *Kubernetes CoreDNS* configuration, which controls the DNS of the *Kubernetes* cluster. All the configuration of both the *NGINX* and the DNS is automated, adding the needed new rules of *hostnames* translation to the cluster's *CoreDNS* configuration. This configuration is done through a script using *regex* to respect the syntax of the configuration files which are in *.yaml* format.

Liqo Metrics Exposition

Liqo's metric collection is completely independent of the *Prometheus* orchestration package. This allows better management of resources, as we do not need a whole Prometheus deployment in each cluster to collect metrics from *Liqo*.

During *Liqo's* installation, the orchestrator checks for the *Prometheus* deployment of the *management cluster*. If it is running, the endpoint of *Liqo's* metrics exposition service is added to the management cluster's *Prometheus* server via *regex* in the *Liqo's* installation script. As there are scenarios where multiple clusters with Liqo may run together, the overlapping of endpoints is avoided by using the cluster's name, which is unique when adding the endpoint to the *Prometheus* configuration.

6.4 Summary

This chapter presented the elicited requirements for all the components of the proposed orchestrator. Derived from the project discussions and needs, they intended to characterize the key functionalities of the envisioned orchestrator. Later, this chapter discussed the architecture and functionalities provided by the orchestrator and the role of each component composing it. Such an orchestrator, inspired by the

ETSI ZSM, aims to automate the lifecycle management of distributed Kubernetes-based environments and containerized applications, the main goal behind this thesis.

This orchestrator makes use of Kubernetes CRD and Operator to realize the idea of closed loops for continuous infrastructure and application management. The CRD keeps an updated registry of the applications and infrastructure, while the operator monitors the CRD and reacts to its changes. We discuss how different orchestration capabilities can be implemented using such an idea of closed loops (e.g., OODA loops). The orchestrator also leverages the Cluster API and Liqo OSS to fulfil two remarkable operations, the declarative infrastructure configuration and bootstrapping, as well as the dynamic interconnection between several Kubernetes clusters, following the intention of allowing a seamless distribution of microservice-based applications across multiple clusters. Additionally, this work also includes the monitoring and package installation as complementary components to support the overall orchestration system.

Chapter 7

Testing, Integration and Validation

This chapter covers the integration and validation steps to ensure both the individual and the overall system behave as expected. This included the integration within the CHARITY testbed and components and the validation through the deployment of applications. The work presented in this chapter contributed to scientific publications. In the same way, this work also contributed to and was demonstrated within the CHARITY project showcasing activities. Section 7.1 explains how the orchestration system was deployed and integrated within the CloudSigma provider testbed. Section 7.2 details the tests used to evaluate and validate the orchestrator's functions. Section 7.3 details the integration of the Application Management Framework (AMF) with the orchestrator. Section 7.4 details the integration of an asynchronous communication mechanism to the CHARITY use cases and the orchestrator. Section 7.5 details the integration and validation of different bootstrap providers with the orchestrator. Section 7.6 details the orchestration system handling of real use cases.

7.1 Testbed Integration

This section details the testbed used to test and validate the orchestration system together with the CHARITY use cases and the required adaptations regarding the infrastructure environment, involving the CHARITY project cloud provider CloudSigma.

CloudSigma [55] is a partner in the CHARITY project, and is the main provider of cloud computing resources (e.g., Virtual Machines). The testbed for the orchestration system resided in a CloudSigma hosted environment. Despite being a cloud provider, the resources were limited, specially regarding the public IPs, as they are a recurrent needed resource throughout the several experiments and the orchestrator's features. The environment was composed of a virtual machine hosting a **single node Kubernetes cluster** (the management cluster), and a virtual machine hosting **an OpenStack provider lightweight distribution, MicroStack**.

As Cluster API does not support CloudSigma as an official cloud provider, the orchestrator's deployed clusters via Cluster API use the OpenStack provider hosted within CloudSigma premise. This means a cloud provider is hosting another cloud provider, which consequently is hosting the virtual machines hosting the Kubernetes clusters deployed via the orchestrator. Figure 7.1 represents the infrastructural setup.

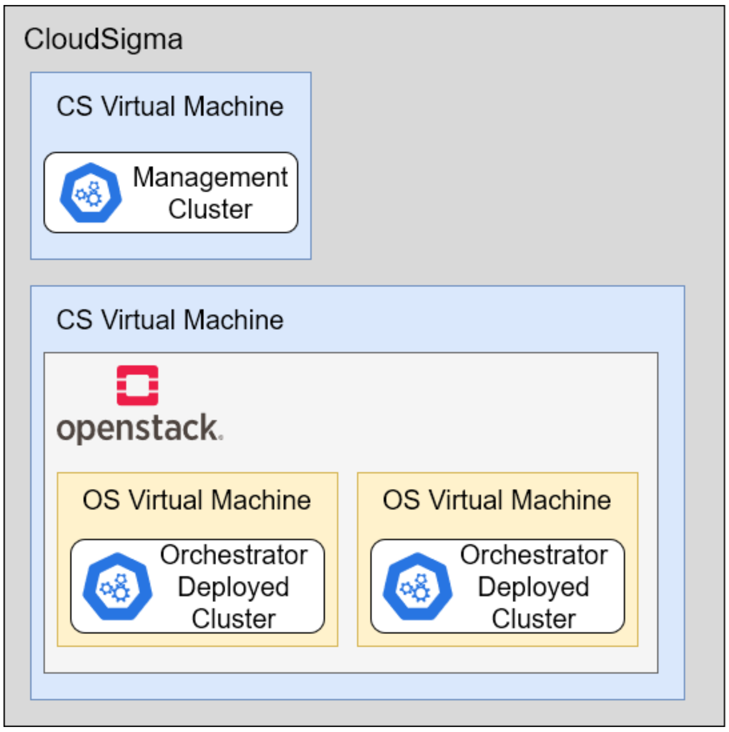


Figure 7.1: Testbed Infrastructure Setup

Aside from the provider itself, a VPN for accessing the virtual machines and the Kubernetes clusters hosted within them from outside the network is deployed using *WireGuard* [52] (for more details on WireGuard, refer to Section 5.2).

The testbed was used throughout the testing of the orchestration system, including scenarios reported in a published paper [56] and live demonstrations such as at CHARITY partner meetings and at EUCnC & 6G Summit 2023 [53], where the orchestrator was showcased.

7.2 Functional Testing

This section describes the functional tests made to test the different functions of the proposed solution. These tests aimed to assess the correct behaviour and capabilities of the orchestrator. They were all performed in the CloudSigma testbed (refer to Section 7.1).

The following list details such tests and their respective results mapped by each requirement. The requirements lists can be found in the previous Chapter 6: General Requirements (CHA-, Table 6.1), Custom Resource Definition (CR-, Table 6.2), Operator (OP-, Table 6.3) and Backend (BE-, Table 6.4). Each table entry contains the result as follows: (Y)es, (N)o, and (P)artially achieved.

- **Create clusters** - Deploying a Kubernetes cluster with/-out orchestration-related packages.

  We added a new cluster definition in the CRD and verified the operator detected the change and that clusters were successfully created and could be accessed through the *.kubeconfig*. Using the Openstack provider also means the creation of a Virtual Machine. This test maps with the requirements (IDs): *CHA1, CR1-2, OP1-6, BE1-5, BE18.*

- **Scale a cluster** - Scale an already existing Kubernetes cluster.

  We updated the node count in the cluster definition and verified the clusters were successfully scaled-in/-out, showing the different node counts. VMs were also created and removed accordingly. This test maps with the requirements (IDs): *CHA3, CR1,CR2, OP1-6, BE1, BE4, BE17.*

- **Delete a cluster** - Delete a Kubernetes cluster environment.

  We deleted the cluster from the CRD and verified the clusters were successfully deleted, as they were not listed in the management cluster, and the corresponding VM was also removed. This test maps with the requirements (IDs): *CHA2, CR1-2, OP1-6, BE21*.

- **Connect two clusters** - Connect two existing Kubernetes clusters using Liqo.

  We added a cluster link definition in the CRD and verified the clusters were successfully connected (paired in Liqo terminology) by observing the connection status using *Liqoctl*, the Liqo CLI. This test maps with the requirements (IDs): *CHA6, CR1, CR3, OP1-6, BE6.*

- **Deploy a distributed application** - Deploy an application across two Kubernetes clusters using Liqo's offloading capabilities.

  First, we added the application definition in the CRD and then verified the application components were correctly installed in each cluster. Later, we verified the network communication between components. For that, we used different reference applications, including the VR Tour Creator and Video Streaming use cases as later detailed in Sections 7.6.1 and 7.6.2) respectively. This test maps with the requirements (IDs): *CHA4, CR1, CR4, OP2-6, BE7-12*.

- **Delete a distributed application** - Delete an application running in a Kubernetes environment.

  We deleted the application definition from the CRD and verified the application was deleted successfully by checking the content of the clusters using the respective *.kubeconfig*. This test maps with the requirements (IDs): *CHA5, CR1, CR4, OP2-6, BE14, BE16.*

- **Metric collection** - Collecting metrics using Prometheus and checking them using a custom Grafana dashboard.

  We verified the metrics and dashboard were working correctly by observing the Prometheus/Grafana dashboards. This included metrics exposed by Kubernetes itself through *kube-state-metrics*, and metrics exposed via orchestrator. This test maps with the requirements (IDs): *CHA7-8.*

Overall, the tests and requirements were successfully achieved, highlighting the capabilities of the orchestrator in fulfilling the originally planned features. The proposed architecture proved to be a valuable fit for bringing automation to the orchestration process. Indeed, the idea of having different closed loops and routines allowed for the automation of different aspects of orchestration based on a declarative definition. Such definition, leveraging the concept of Kubernetes CRDs, also plays a relevant role in future integration with additional components.

7.3 Application Management Framework Integration

This section describes the workflow between the orchestrator and Application Management Framework (AMF), including the workflow and the process of translating the output of the Application Management Framework (AMF) into the orchestrator CRD. This integration fullfills the requirements with the following IDs: *CHA9, CR1, CR4, OP2-6, BE14, BE16, BE19, BE20*.

AMF acts as a frontend for the developers to design and deploy their applications to a distributed cloud infrastructure without worrying about infrastructural details. The infrastructural transparency is delivered through the orchestration system which takes the input received by the AMF, and converts it to the infrastructure.

Figure 7.2 represents the internal process running of the orchestrator integration with the AMF.

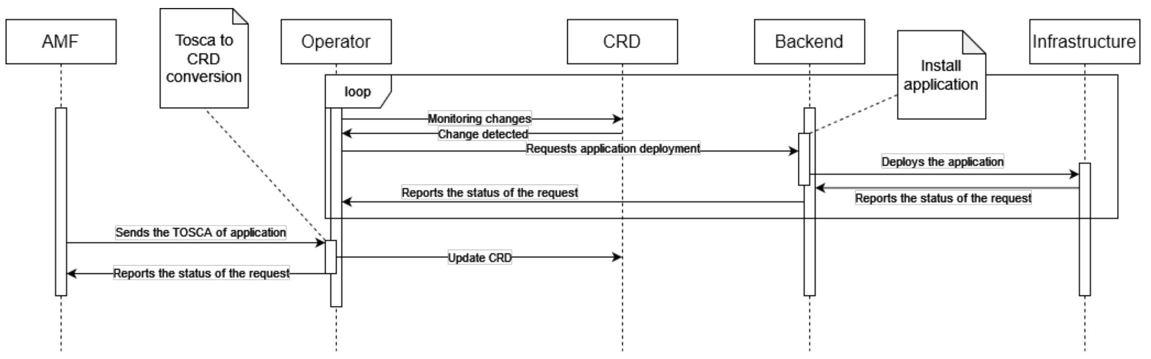

AMF
Tosca to CRD conversion
Operator
CRD
Backend
Install application
Infrastructure
loop
Monitoring changes
Change detected
Requests application deployment
Deploys the application
Reports the status of the request
Reports the status of the request
Sends the TOSCA of application
Update CRD
Reports the status of the request

Figure 7.2: Orchestration system handling the AMF Input

For a better understanding of the orchestrator's behind-the-scenes process, figure 7.3 showcases an example of a TOSCA application blueprint and figure 7.4 represents a **part of the custom resource** resulting from the conversion done by the orchestrator.

The process begins as soon as the user deploys the application through the AMF frontend. Deploying the application via the AMF triggers an *HTTP* request to the orchestrator with the TOSCA data of the application blueprint.

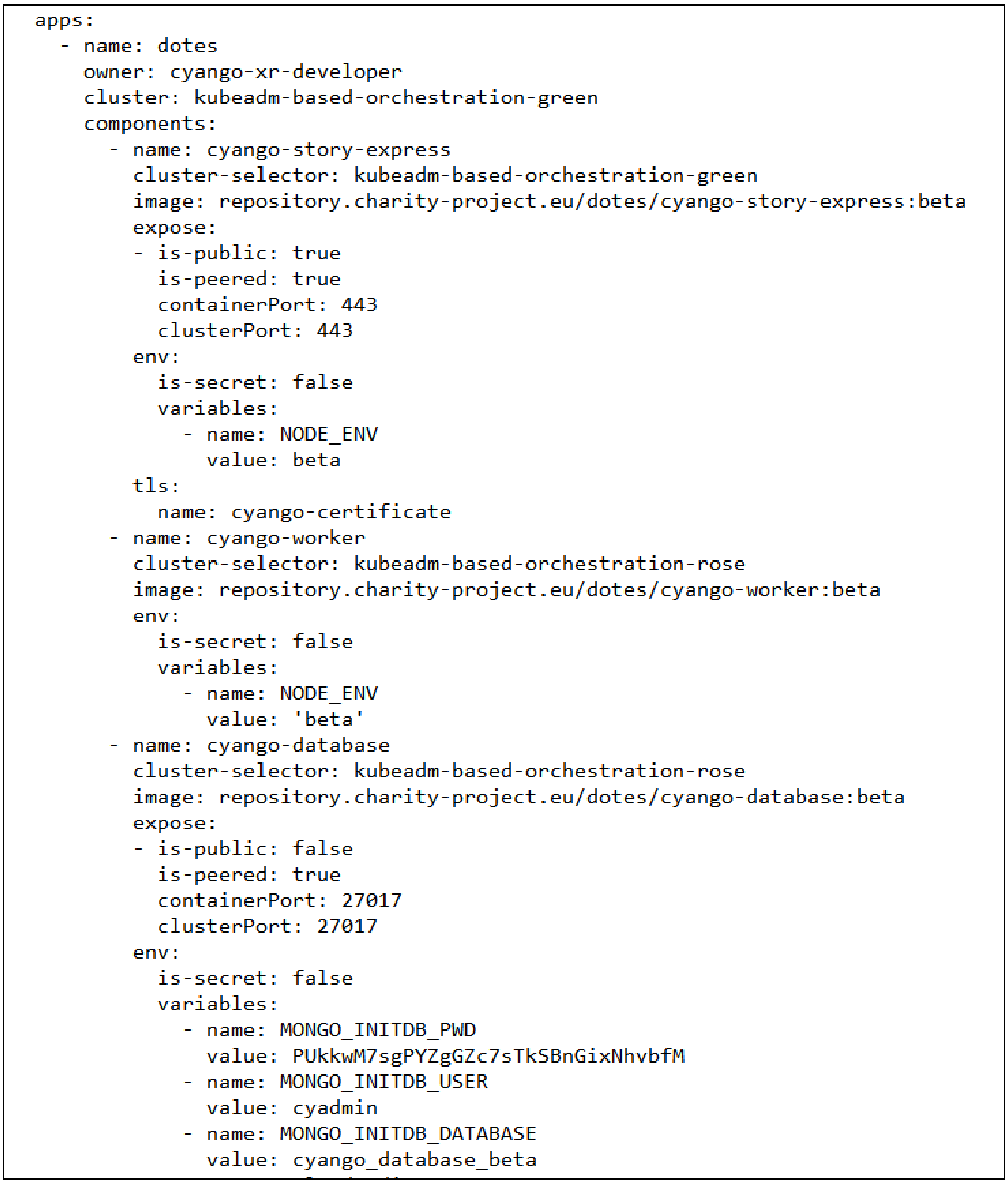
The operator's middleware interprets the information regarding the components of the application (i.e., name, image repository, component type), as well as information about the network connections between the components, included

```
description: Cloud Studio Service

metadata:
  # The following fields are "normative" and expected in TOSCA
  template_name: Cyango Cloud Studio - reimported
  template_author:  cyango-xr-developer
  template_version: ''

imports:
  - charity_custom_types_v08.yaml

topology_template:
  inputs:
  node_templates:
    charity-kafka:
      type: Charity.Component
      properties:
        name: charity-kafka
        deployment_unit: EXTERNAL
    AmazonS3:
      type: Charity.Component
      properties:
        name: amazons3
        deployment_unit: EXTERNAL
    cyango-backend:
      type: Charity.Component
      properties:
        name: cyango-backend
        deployment_unit: K8S_POD
        placement_hint: CLOUD
        image: repository.charity-project.eu/dotes/cyango-backend:beta
        environment:
          NODE_ENV: { get_input: NODE_ENV }
      requirements:
        - host: cyango-backendNode
    cyango-backendNode:
      type: Charity.Node
      node_filter:
        capabilities:
          - host:
              properties:
                num_cpus:
                  - equal: 0
                mem_size:
                  - greater_than: 0 MB
```

Figure 7.3: TOSCA Application Blueprint Example

in the TOSCA blueprint. The blueprint may also include information regarding external sources.

The conversion runs in the **middleware part of the operator** and starts by taking the basic TOSCA data fields and assigns those fields to the corresponding fields of an empty template based on the CR. These fields (i.e., name, image repository, environment variables) are used to create the Kubernetes deployment. During this first step, the components are also assigned additional information agnostic to the AMF and the user, as is the **cluster scheduling**. The dynamic cluster scheduling should be handled by an AI algorithm, but for **testing purposes**, there are default clusters defined. The next step during the conversion is translating the component network connections, known as virtual links in the TOSCA. This part is rather



```yaml
apps:
  - name: dotes
    owner: cyango-xr-developer
    cluster: kubeadm-based-orchestration-green
    components:
      - name: cyango-story-express
        cluster-selector: kubeadm-based-orchestration-green
        image: repository.charity-project.eu/dotes/cyango-story-express:beta
        expose:
        - is-public: true
          is-peered: true
          containerPort: 443
          clusterPort: 443
        env:
          is-secret: false
          variables:
            - name: NODE_ENV
              value: beta
        tls:
          name: cyango-certificate
      - name: cyango-worker
        cluster-selector: kubeadm-based-orchestration-rose
        image: repository.charity-project.eu/dotes/cyango-worker:beta
        env:
          is-secret: false
          variables:
            - name: NODE_ENV
              value: 'beta'
      - name: cyango-database
        cluster-selector: kubeadm-based-orchestration-rose
        image: repository.charity-project.eu/dotes/cyango-database:beta
        expose:
        - is-public: false
          is-peered: true
          containerPort: 27017
          clusterPort: 27017
        env:
          is-secret: false
          variables:
            - name: MONGO_INITDB_PWD
              value: PUkkwM7sgPYZgGZc7sTkSBnGixNhvbfM
            - name: MONGO_INITDB_USER
              value: cyadmin
            - name: MONGO_INITDB_DATABASE
              value: cyango_database_beta
```

Figure 7.4: Custom Resource Converted From TOSCA

complex as the data received from the *AMF* is not designed with *Kubernetes* in mind. As such, we need to analyse thoroughly the information given by the TOSCA virtual links and decide if the components should be exposed internally and/or externally. The analysis consists firstly on verifying the components which are external to the application and save the links to which they are associated on a list. This list will be used to compare to the application components virtual links, so we know which components should be publicly exposed and create the corresponding **Kubernetes ingress**. For the internal network exposition, we check if the virtual links of the application components have a common virtual link. If so, the component needs to be exposed as a **Kubernetes service**.

The final step of the conversion is joining all of the converted data with the correct *.yaml* syntax and format. The output of the TOSCA conversion is then applied to the existing Custom Resource through the operator's middleware. The operator detects the changes in the CR and requests the backend to apply the changes to the infrastructure. Feedback from the changes is sent from the infrastructure to the AMF frontend, while going through the middleware of the orchestrator. This results in a successfully deployed application with the AMF and the orchestrator.

It is also possible to delete applications through the AMF. Assuming the user is logged in through the AMF, the user can request the orchestrator for a list of his deployed applications. The request goes through the middleware of the operator, and checks the CRD for entries of deployed applications, using the user who requested the list as a filter. This is possible as the definition of an application in the CRD requires it to be associated with a user. The list of applications is then returned to the AMF with a list of the applications or, in case there is any application associated with that user, an empty list.

For the deletion of an application through the AMF, assuming the user has at least one deployed application, the user chooses from the list of applications obtained and explained in the previous step. The user selects from the list, the application to be deleted, and the request is sent to the orchestrator. The orchestrator's middleware uses the content of the request, which is the name of the application and deletes the whole application entry from the CR. This causes the operator to act, as the CR changed and sends a request to the backend with data of the application entry removed from CR. The backend then unoffloads the application, in case it was distributed across clusters and deletes the namespace which the application was deployed in, as all resources related to applications are namespaced.

7.4 Asynchronous Communication Mechanism Integration

This section details the integration of an asynchronous communication mechanism using Kafka within a Kubernetes cluster residing in CloudSigma, offering the CHARITY Use-Cases a new way to handle communications if they choose to integrate Kafka.

Kafka [57] is an open-source distributed streaming platform, excels at handling real-time data streams in modern data-driven applications. Its core design centers around asynchronous communication between producers and consumers, enabling seamless data flow without immediate processing constraints. This unique architecture empowers asynchronous data processing, making it exceptionally effective for managing vast data volumes and real-time streams.

To deploy a *Kafka* cluster in Kubernetes, **Strimzi** [58] and **Koperator** [59] were tested as potential solutions. Both solutions feature a Kubernetes operator, managing the lifecycle of the Kafka cluster and its resources. As *Koperator* proved to be difficult to install, and Strimzi offered an optional integrated dashboard (CruiseControlUI [60]), we opted for the latter. For testing the Kafka cluster, a number of functional (i.e., creating, deleting topics, sending messages, message history) and performance tests (i.e., number of Kafka nodes, number of topics, message sizes, flow of traffic) were executed.

A first version of Kafka using *Strimzi* was deployed in the *CloudSigma* use-cases testbed, and it was tested and integrated with the *VR Tour Creator* use-case (more details in Section 7.6.1). The VR Tour Creator use-case uses Kafka topics to assign media conversion tasks and handle them asynchronously.

Kafka with the *Strimzi* operator can be deployed to clusters managed by the orchestrator, as an orchestration package 6.3.5, thus the same setup as in the *CloudSigma* testbed can be created using the orchestrator. The orchestrator should also integrate Kafka, using it as an asynchronous monitoring mechanism, enabling more precise decisions regarding the actions between the components composing the whole orchestration system. Stopping the deployment of a cluster or application if an error is encountered and revert the changes made in the CRD, can be held as examples of actions influenced by the asynchronous communication.

As the requests processed by the orchestrator are already handled as tasks that change status according to the tasks' steps and progress, each task should be included in a *Kafka* topic, updated according to the progress of the task, resulting in asynchronous monitoring and communication, as the the tasks' feedback can be exchanged between intervening components and with the exchanged information, act accordingly. Kafka also provides persistency, as Kafka topics can remain stored and be used for logs and debugging, allowing for further improvements.

7.5 Bootstrap Provider Integration and Validation

This section presents the bootstrap providers (i.e., Kubeadm, K3s, MicroK8S) tested with Cluster API integrated with orchestrator and assess their compatibility and performance.

Despite not being an author, the results presented in this section are reflected in the publication "*Cross Kubernetes Cluster Networking to Support XR Services: Challenges, Solutions and Performance Evaluation*", submitted to the *IEEE Network Magazine* [56].

Kubeadm is the default provider and is officially supported by *Cluster API*, and as such it is the most compatible. Although *MicroK8S* is officially supported by *Cluster API* as a bootstrap provider, it only works with single node cluster deployments. When deploying a cluster with more than one node with *MicroK8S*, the worker nodes do not connect to the control plane node as the control plane deems the worker nodes as foreign entities. A workaround was developed by injecting a modified *cloud-init* script into each virtual machine hosting a node created by *Cluster API* consisting on using SSH connections adding every node's hostname and IP to each node hosts list. Even with the custom script, the setup would often fail when deploying with more than 1 node and it was deemed unfeasible in a real environment. As such, we do not include tests for *MicroK8S*.

K3s is not officially supported by *Cluster API*, but by analysing the templates available for the other bootstrap providers, a new usable custom template was created. This is relevant as there maybe situations where computing resources are limited and we need a lighter Kubernetes cluster and as such, the K3s provider is a lighter Kubernetes distribution designed for the edge of the network. Through consequence, as the orchestrator integrates Cluster API, we can support the deployment of lighter Kubernetes environments.

The evaluation involved measuring several key steps, including the time required to generate the cluster resource definitions (.yamls), apply these resources to the management cluster, create the relevant Cloud resources (i.e., Virtual Machines in OpenStack), and finally, set up the Kubernetes cluster itself. To generate the cluster definitions, we utilized *Cluster API* CLI, along with default templates provided by each provider (*K3s* a custom template was used, as explained above).

For each *Kubernetes* distribution, we assessed the provisioning time for different node configurations: one node (comprising a control plane), three nodes (one control plane and two workers), and five nodes (one control plane and four workers). All nodes, including Virtual Machines, were set up using *Ubuntu* images and a *m1.medium* flavor with 2 *vCPUs*, 4GB RAM, and *20GB* of disk allocated for both the control plane and worker nodes.

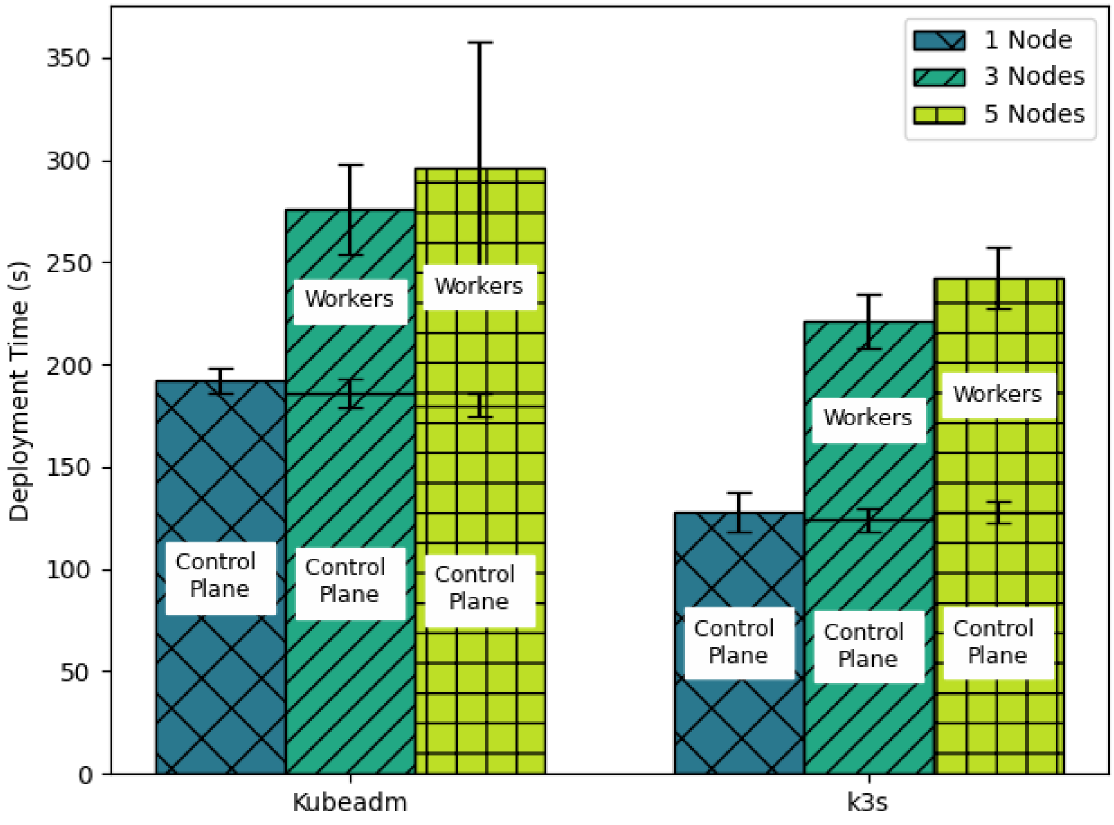


Figure 7.5: Cluster deployment time by bootstrap provider and cluster size [56]

In Figure 7.5, we showcase the experiment results, encompassing the deployment time for the control plane and the overall deployment time for all cluster nodes. The deployment time is measured as the duration until the control plane is marked as "Ready" for single-node clusters or until all nodes are marked as "Ready" for multi-node clusters. To ensure reliability, each test was repeated 30 times, and the error bars on the graph indicate the standard deviation.

As expected, the deployment time increases as we increase the number of nodes of the cluster, in both Kubeadm and K3s. When comparing both bootstrap providers, K3s has a faster deployment time than Kubeadm in all of the tests (i.e., 1 node, 3 nodes, 5 nodes). As K3s is designed as an edge Kubernetes distribution, is lighter and consequently faster than Kubeadm which is considered to be the "vanilla" Kubernetes distribution.

More detailed results are presented in the referred paper [56]

7.6 Use-Case Integration, Validation and Support

This section describes use-cases integrated with the orchestrator (specified in Section 6.3). Initially, the use-cases were manually deployed with the manual process being explained in detail in the respective sub-section for each use-case. Later on, in the development phase, they were deployed using the orchestrator, showcasing its versatility and capability for deploying different kinds of applications.

7.6.1 Use-Case I - VR Tour Creator Use-Case Multi-Cluster Deployment

This section describes the scenario created to test the deployment of real use-case application of **CHARITY** (already working on a single-cluster architecture), on a multi-cluster architecture using the *Liqo* and *Cluster API*, tools researched in the previous scenarios described in this chapter (Liqo in section 5.4 and Cluster API in section 5.5).

The **VR Tour Creator** use-case [61] is a "video editing software crafted for anyone who wants to create 360° content and immersive digital experiences", which is deployed in the cloud using Kubernetes.

The **goals of the scenario** are listed as the following:

- Compare the behaviour of the application between the single-cluster and multi-cluster architectures.
- Compare the performance of the application between the single-cluster and multi-cluster architectures.

Scenario Description and Procedure

The **rose** cluster is running on a virtual machine deployed within **CloudSigma** cloud provider infrastructure with Kubernetes *v1.25* and was deployed using **Kubeadm** with *Kube-Flannel* CNI and MetalLB *v0.13.7*.

The **green** cluster is running on a machine provisioned by **ClusterAPI + OpenStack** with Kubernetes *v1.25* and was deployed using **KubeAdm** with *Kube-Flannel* CNI and MetalLB *v0.13.5*. Figure 7.6 represents the schema for *scenario IV*.

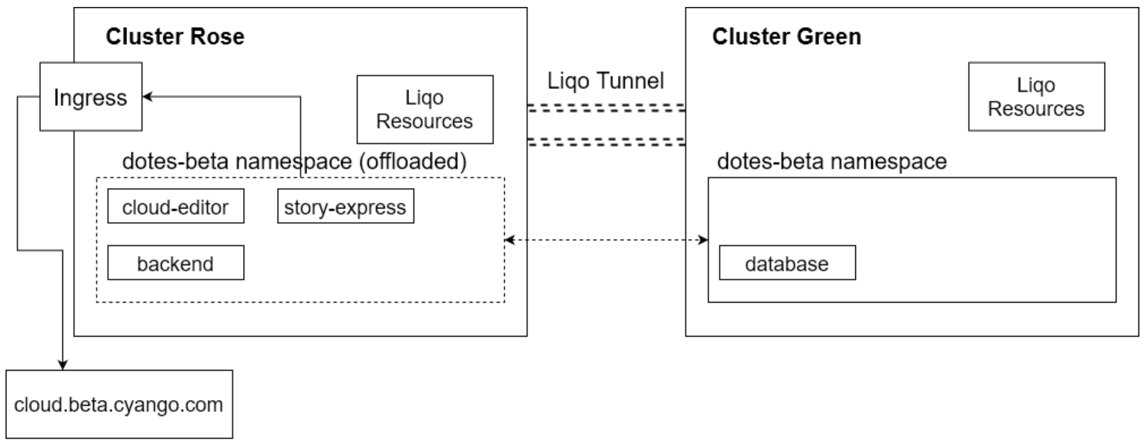


Figure 7.6: Use Case I - VR Tour Creator Use-Case Multi-Cluster Deployment

In the **rose** cluster, we proceed with the installation of *NGINX* ingress controller (refer to Section 5.1), which is **responsible for the management of the ingress resources** needed for **exposing services** to network traffic external to the cluster. After the installation of *NGINX*, we need to force it to use a fixed and public IP, previously defined as a *IPPool* in *MetalLB*. With the ingress controller properly installed and configured, we proceed with the installation of *Liqo*. Due to the fact that *Liqo* gateway and authentication services need to be exposed to the outside, in order to the **green** cluster to reach the **rose** cluster, and as we have only one public IP, it needs to be shared between the *NGINX* controller and *Liqo* services. By using **Kubernetes annotations** on both *Liqo* and *NGINX* services, we are able to share the IP.

In the **green** cluster, we install *MetalLB* and define the *IPPools* to be used by the services created by *Liqo*. **Green** cluster's setup is simpler as it doesn't need the an ingress controller.

We proceed with the cluster peering using the **in-band control-plane approach**, as we have access to the *kubeconfig* files of both green and rose clusters. The peering is also **bidirectional** meaning that a virtual node representing each cluster will be created in the opposing cluster. The **green** cluster will have a virtual node representing the **rose** cluster and the **rose** cluster will have a virtual node representing the **green** cluster. With the *Liqo* VPN tunnel established between clusters, we start the offloading process and distribute the workload between the clusters.

In the **green** cluster, we create a new namespace called "dotes-beta". In this specific scenario, the nomination is extremely important as the **endpoints defined in the services of the application depend on the namespace's identification**. We proceed with the offloading of the namespace using *Liqo*. During this process, we override the *Liqo's* default values and use the "**–EnforceTheSameName flag**" so that the services can use the **DNS** of Kubernetes, **maintaining the same operation** as the single-cluster architecture, as the components need this to communicate with each other.

As we have the *.yaml* files used to deploy the different components of the application, in each of the them we define the affinity regarding the local or virtual nodes so that *Liqo* and *Kubernetes* knows in which cluster to deploy the **DOTES'** components. As illustrated in Figure 7.6, the **database** component is running on the **green** cluster (ClusterAPI + OpenStack) and the **cloud-editor**, **story-express** and **backend** components are running on the **rose** cluster. To verify that the components were deployed in the correct cluster, the "describe" command of **Kubernetes CLI** is used to ensure the correct deployment. With every component deployed and running **we can now test the access to the application** by adding the ingress hostname to the hosts file of our system.

Results and Discussion

After the deployment of the use-case, its features could be accessed via the web browser as it is intended. Comparing the single-cluster against the multi-cluster approach, we don't notice any difference regarding the usability and the performance of the application, although there is a minimal overhead in the traffic flowing through the VPN tunnel. Liqo uses *wireguard* behind the scenes (refer to Section 5.2) as the distributed components need to exchange data. With these results, we assure that there is no compromise on the user side and the application maintains its transparency regarding the implementation.

Orchestrator Integration

To support the *VR Tour Creator* use-case, new features were added to the orchestrator, such as support for TLS certificates for outside communication through ingress. The use-case can be deployed through the orchestrator in both single-cluster and multi-cluster architectures.

This use-case is fully integrated with the orchestrator, and was showcased during the EUCnC & 6G Summit 2023 [53] booth exhibition.

7.6.2 Use-Case II - Video Streaming Service

This section describes the scenario created to test the deployment of real use-case application on a multi-cluster architecture using the *Liqo* and *Cluster API*, tools researched in the previous scenarios described in this chapter (Liqo in Section 5.4 and Cluster API in Section 5.5).

Despite not being an author, the results of this use-case integration are reflected in the publication "*Cross Kubernetes Cluster Networking to Support XR Services: Challenges, Solutions and Performance Evaluation*", submitted to the *IEEE Network Magazine* [56].

The **goals of the scenario** are listed as the following:

- Compare the behaviour of the application between the single-cluster and multi-cluster architectures.

- Compare the performance of the application between the single-cluster and multi-cluster architectures.

Scenario Description and Procedure

The **rose** cluster is running on a virtual machine deployed within **CloudSigma** cloud provider infrastructure with Kubernetes *v1.25* and was deployed using **Kubeadm** with *Kube-Flannel* CNI and MetalLB *v0.13.7*.

The **green** cluster is running on a machine provisioned by **ClusterAPI + OpenStack** with Kubernetes *v1.25* and was deployed using **KubeAdm** with *Kube-Flannel* CNI and MetalLB *v0.13.7*.

In this experiment, as there was not a single-cluster version of the application already deployed, we start by deploying the video-streaming service in a single-cluster architecture so we can compare to the multi-cluster approach. Figure 7.7 represents the single-cluster version of the video streaming service in **Rose** cluster.

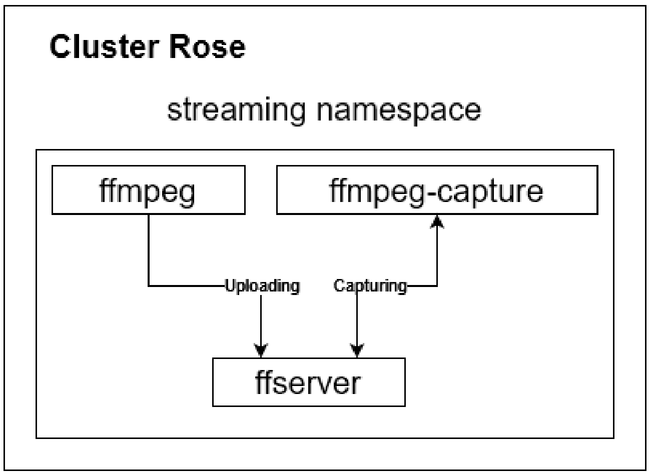


Figure 7.7: Use Case II - Single-Cluster Deployment

For this first deployment, we chose the **Rose** cluster. We start by creating the "streaming" namespace, where the video streaming components will be deployed on. The **ffserver** component is responsible for receiving video data from a *provider* and broadcasting it to *clients*. In this particular case, the **ffmpeg** component produces a stream from a previously downloaded video file to the **ffserver** and the **ffmpeg-capture** component consumes the video being broadcasted by the **ffserver**. To make **ffserver** accessible to the both the providers and the clients (ffmpeg and ffmpeg-capture), a **Kubernetes service** is added to the deployment, exposing the component.

After deploying the components to the namespace, we start uploading the video to the **ffserver** using **ffmpeg** and we start the capture using **ffmpeg-capture**. Ffmpeg-capture saves the captured video to the pod and when the capture stops, the video needs to be copied from the pod to the virtual machine, so we can observe the capture results.

Figure 7.8 represents the multi-cluster version of the video streaming service.

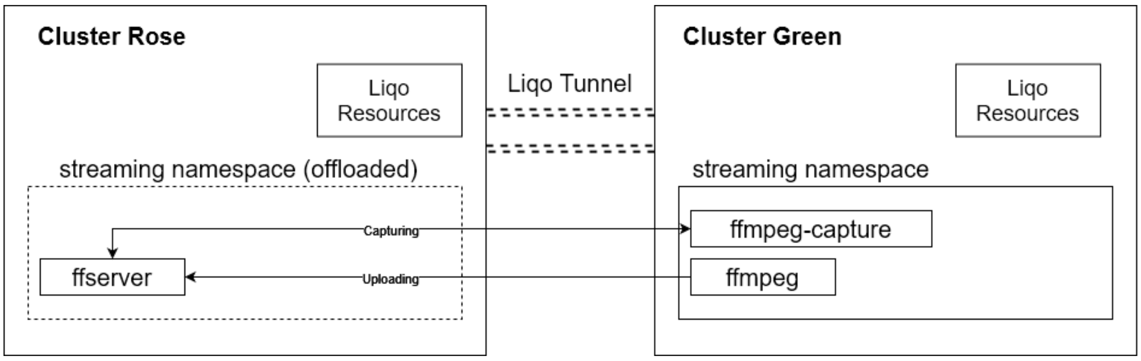


Figure 7.8: Use Case II - Multi-Cluster Deployment

Different from the single-cluster architecture, the first step is to peer the clusters using *Liqo*, as explained in the *VR Tour Creator* use-case scenario (section 7.6.1). With the green and rose clusters peered successfully, we create the *streaming* namespace in the **green** cluster and offload it to the **rose** cluster. With the namespace now extended across both clusters, we deploy the video streaming components to **green** cluster which will be distributed across both clusters. As the cluster affinity was previously defined in the *.yaml* configuration files of each component, we just apply them in the **green** cluster. The whole setup is now ready for testing. We execute same test as in the single-cluster approach.

We start uploading the video to the **ffserver** using **ffmpeg** and we start the capture using **ffmpeg-capture**. Ffmpeg-capture saves the captured video to the pod and when the capture stops, the video needs to be copied from the pod to the virtual machine, so we can observe the captured results.

Results and discussion

The multi-cluster deployment of the streaming service proved to be on par with the single-cluster deployment, despite the network overhead of having two clusters connected, as registered in the paper [56] and shown in the results presented in Figure 7.9. This may occur as in the distributed scenario, neither cluster is burdened with all the components leading to a reduction in latency on both clusters. As the previous use-case (Section 7.6.1), the application works the same in both approaches, maintaining the transparency needed for users. Despite not being an author, this use case and its results contributed to the referred paper [56].

Orchestrator Integration

No additional orchestrator changes were needed. The video streaming use case consisted of a regular containerized-based application, already supported by the orchestrator.

7.7 Summary

Overall, the orchestrator proved to be a viable solution to automate and orchestrate Cloud-Native applications. The orchestrator's functionalities were thoroughly

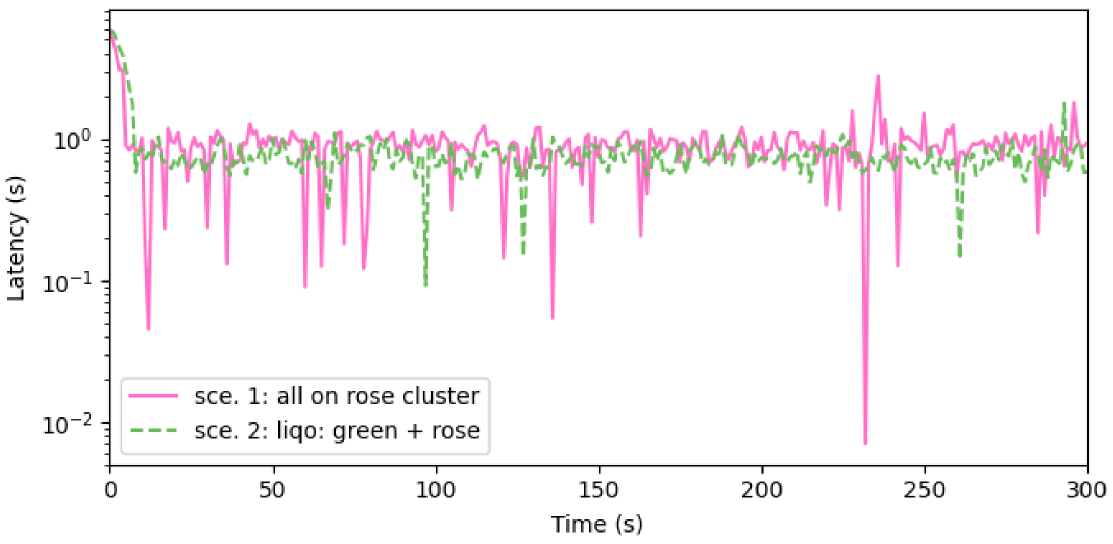


Figure 7.9: Latency between streaming and view times of the two scenarios [56]

tested using different scenarios and reference use cases.

Indeed, **The VR-Tour use-case and the AMF integration with the orchestrator were showcased during the EUCnC & 6G Summit 2023 [53] booth exhibition.** The attendee feedback, in general, was positive, stressing the relevance of such solution.

Chapter 8

Conclusion

The next generation of cloud-based applications can be distributed across domains, providers, and heterogeneous environments. Considering that, this work focused on investigating approaches to address their lifecycle management, including the underlying infrastructure of such multi-domain environments. We conceived a Cloud-Native orchestration solution for automating the management of these environments focusing on multi-cluster Kubernetes environments.

The overall thesis objective of researching an orchestration system was achieved. We designed, implemented and tested a full-featured proof-of-concept of such a solution. We investigated various open-source tools and frameworks as building blocks of the whole orchestrator, having integrated and tested them within the CHARITY testbed environment. We evaluated the various orchestrator's capabilities to assess their functional behaviour and performed the integration with AMF, an additional component of CHARITY. Liqo and Submariner were analysed as potential cluster interconnectivity solutions, while for cluster bootstrapping, the considered tool was Cluster API. According to the results, Liqo proved superior to Submariner due to its workload distribution features. Whereas Cluster API proved to be a robust framework capable of unifying the cluster management primitives across distinct providers. Moreover, Cluster API also allows the usage of different bootstrap providers, such as Kubeadm and K3s.

In brief, we highlight the following contributions provided by this research: (i) the proposed architecture and the AMF integration, which was part of a conference paper; (ii) the orchestrator implementation itself leveraging Liqo and Cluster API; (iii) the functional and performance evaluation of different Cluster API bootstrap providers reflected as part of a contribution to a journal publication; (iv) the validation using the VR Tour Creator use case of CHARITY; (v) and last but not least, the showcasing of orchestrator capabilities at the EUCnC & 6G Summit 2023 [53].

The developed orchestrator represents a significant step towards bringing automation to the management of Cloud-Native container-based applications and Kubernetes clusters. Nevertheless, these capabilities are planned to be further integrated into CHARITY's architecture by adding AI-based mechanisms for intelligently deciding, for instance, what clusters need to be created or where to deploy the application components. Likewise, we plan to have components to communicate asynchronously, which promotes a more modular approach to initiating, delegating, and monitoring individual operations. This means that time-consuming tasks, like cluster creation or package installation, can be initiated, run in the background, and their status checked at any moment, irrespective of the initial request.

References

[1] OneSource. Onesource. [Online]. Available: https://onesource.pt/

[2] CHARITY. What is charity? [Online]. Available: https://www.charity-project.eu/

[3] Liqo. What is liqo? [Online]. Available: https://docs.liqo.io/en/v0.6.1/

[4] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini, "Computing without borders: The way towards liquid computing," pp. 1–18, 2022.

[5] Submariner. Submariner: project documentation. [Online]. Available: https://submariner.io/

[6] L. Osmani, T. Kauppinen, M. Komu, and S. Tarkoma, "Multi-cloud connectivity for kubernetes in 5g networks," pp. 1–6, 2021.

[7] C. API. Kubernetes cluster api. https://cluster-api.sigs.k8s.io/. (accessed: 06.12.2022).

[8] Redhat. What is orchestration? [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-orchestration

[9] ETSI. Zero touch network & service management (zsm). [Online]. Available: https://www.etsi.org/technologies/zero-touch-network-service-management

[10] ——, "Experiential networked intelligence (eni); overview of prominent control loop architectures," pp. 1–17, 2021.

[11] Kubernetes. Kubernetes: Kubernetes components. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/#node-components

[12] T. G. Peter Mell, "The nist definition of cloud computing," pp. 1–7, 2011.

[13] IBM. What is cloud computing? [Online]. Available: https://www.ibm.com/topics/cloud-computing

[14] B. Reselman. 3 questions to answer when considering a multi-cluster kubernetes architecture. [Online]. Available: https://www.redhat.com/architect/multi-cluster-kubernetes-architecture

[15] Kubernetes multi-clusters: How & why to use them. [Online]. Available: https://www.bmc.com/blogs/kubernetes-multi-clusters/

[16] S. Lewis. Ooda loop. [Online]. Available: https://www.techtarget.com/searchcio/definition/OODA-loop

[17] V. Paradigm. What is ooda loop? [Online]. Available: https://online.visual-paradigm.com/knowledge/decision-analysis/what-is-ooda-loop/

[18] P. H. Pierre Imai and T. Amin, "Towards a truly autonomous network," pp. 1–61, 2020.

[19] R. Zager and J. Zager, "Ooda loops in cyberspace: A new cyber- defense model," pp. 1–20, 2017.

[20] RedHat. What is kubernetes? [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-kubernetes

[21] Kubernetes. Kubernetes: Pods. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[22] M. Palmer. Kubernetes networking guide for beginners. [Online]. Available: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html

[23] Kubernetes. Kubernetes: Nodes. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/

[24] ——. Kubernetes: Deployments. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[25] VMWare. What is a kubernetes deployment. [Online]. Available: https://www.vmware.com/topics/glossary/content/kubernetes-deployment.html

[26] Kubernetes. Kubernetes: Daemonset. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[27] K. Advocate. How to deploy daemonsets service in kubernetes (k8s)? [Online]. Available: https://medium.com/avmconsulting-blog/deploying-daemonsets-service-in-kubernetes-k8s-37d642dcd66f

[28] Tigera. Kubernetes cni explained. [Online]. Available: https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/

[29] Kubernetes. Kubernetes: Service. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[30] LearnITGuide. Kubernetes services explained with examples. [Online]. Available: https://www.learnitguide.net/2020/05/kubernetes-services-explained-examples.html

[31] Kubernetes. Kubernetes: Ingress. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/

[32] ——. Kubernetes: Namespaces. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[33] ——. Kubernetes: Scheduler. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[34] ——. Operator pattern. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[35] Liqo. Peering - liqo. [Online]. Available: https://docs.liqo.io/en/v0.6.1/features/peering.html

[36] ——. Offloading - liqo. [Online]. Available: https://docs.liqo.io/en/v0.6.1/features/offloading.html

[37] T. Z. Benmerar, T. Theodoropoulos, D. Fevereiro, L. Rosa, J. Rodrigues, T. Taleb, P. Barone, K. Tserpes, and L. Cordeiro, "Intelligent multi-domain edge orchestration for highly distributed immersive services: An immersive virtual touring use case," in *Proc. IEEE Symposium on Intelligent Edge Computing and Communications (iEDGE)*, Chicago, USA, Jul. 2023.

[38] Submariner. Submariner architecture. [Online]. Available: https://submariner.io/getting-started/architecture/

[39] Cluster API, "Kubernetes cluster api," https://cluster-api.sigs.k8s.io/, [Online; accessed 12-December-2022].

[40] ——, "Concepts," https://cluster-api.sigs.k8s.io/user/concepts.html, [Online; accessed 12-December-2022].

[41] OpenStack, "Software overview," https://www.openstack.org/software/, [Online; accessed 12-December-2022].

[42] Canonical. What is edge computing? everything you need to know. [Online]. Available: https://microstack.run/docs/single-node

[43] MetalLB. Metallb - concepts. [Online]. Available: https://metallb.universe.tf/concepts/

[44] Kubernetes. Kubernetes: Kubeadm. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/

[45] K3S. K3s - lightweight kubernetes. [Online]. Available: https://docs.k3s.io/

[46] Minikube. Minikube - documentation. [Online]. Available: https://minikube.sigs.k8s.io/docs/

[47] KinD. Kind - user guide. [Online]. Available: https://kind.sigs.k8s.io/

[48] R. Admin. Comparing kubernetes cni providers: Flannel, calico, canal, and weave. [Online]. Available: https://www.suse.com/c/rancher_blog/comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

[49] NGINX. Nginx ingress controller. [Online]. Available: https://docs.nginx.com/nginx-ingress-controller/

[50] FastAPI. Fastapi. [Online]. Available: https://fastapi.tiangolo.com/

[51] S. J. B. Paul Kirvan. Tosca (topology and orchestration specification for cloud applications). [Online]. Available: https://www.techtarget.com/searchcloudcomputing/definition/TOSCA-Topology-and-Orchestration-Specification-for-Cloud-Applications

[52] WireGuard. Wireguard. [Online]. Available: https://www.wireguard.com/

[53] E. C. on Networks and Communications. 2023 eucnc and 6g summit. [Online]. Available: https://www.eucnc.eu/

[54] Opensource, "What is openstack?" https://opensource.com/resources/what-is-openstack, [Online; accessed 12-December-2022].

[55] CloudSigma. Cloudsigma. [Online]. Available: https://www.cloudsigma.com/

[56] T. Theodoropoulos, L. Rosa, A. Boudi, T. Z. Benmerar, A. Makris, T. Taleb, L. Cordeiro, and K. Tserpes, "Cross kubernetes cluster networking to support xr services: Challenges, solutions and performance evaluation," in *IEEE Network Magazine*, 2023.

[57] Kafka. Apache kafka. [Online]. Available: https://kafka.apache.org/

[58] CNCF. Strimzi. [Online]. Available: https://strimzi.io/

[59] B. Cloud. Koperator. [Online]. Available: https://banzaicloud.com/products/kafka-operator/

[60] K. Liberti. Hacking strimzi for cruise control ui. [Online]. Available: https://strimzi.io/blog/2023/01/11/hacking-for-cruise-control-ui/

[61] Cyango. Cyango cloud studio - 360º video software. [Online]. Available: https://www.cyango.com/cloudstudio