MÁSTER UNIVERSITARIO EN TELECOMUNICACIONES

TRABAJO FIN DE MASTER

AUTONOMOUS INFRASTRUCTURE MANAGEMENT FOR MULTI-CLOUD KUBERNETES CLUSTERS

9 INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURA

Estudiante: Ruiz García, Daniel

Directora: Ibarrola Armendariz, Eva

Curso: 2022-2023

Fecha: 17, Septiembre, 2023

RESUMEN:

Este proyecto desarrolla una gestión de infraestructura autónoma para clústeres de Kubernetes de Multi-Cloud, donde el enfoque radica en aprovechar las API de Kubernetes para construir y gobernar una infraestructura de clúster autónoma en un entorno multi-cloud, donde se activa el autocontrol en AWS, GCP y Azure. Para el desarrollo del proyecto, y en concreto, para la gestión del ecosistema del cluster, se han utilizado diversas herramientas y proyectos de código abierto como Kubernetes, Kubebuilder, Kubeadm, Linux Ubuntu o WireGuard entre los más relevantes. Para controlar los recursos de los proveedores, este proyecto se basa en los kits de desarrollo de software (SDKs) de programación en AWS, GCP y Azure. El proyecto abarca una combinación de lenguajes de programación, pero se utilizó predominantemente Go Language y Shell Script.

ABSTRACT:

This project develops an Autonomous Infrastructure Management for Multi-Cloud Kubernetes Clusters, where the focus lies in harnessing Kubernetes' APIs to construct and govern an autonomous cluster infrastructure in a multi-cloud environment, where self-control across AWS, GCP, and Azure is enabled. For the development of the project, and specifically, to manage the cluster ecosystem, several open-source tools and projects have been used like Kubernetes, Kubebuilder, Kubeadm, Linux Ubuntu, or WireGuard among the most relevant. To control providers' resources this project relies on the go programming Software Development Kits (SDKs) of AWS, GCP, and Azure. The development journey encompasses a blend of programming languages, but predominantly Go Language and Shell Script was used.

LABURPENA:

Hodei anitzeko Kubernetes-en klusteretarako azpiegitura autonomoaren kudeaketa garatzen duen proiektua, non kuberneten APIak aprobetxatzen ditu kuberneten APIa eraikitzeko eta kluster autonomoaren azpiegitura bat gobernatzeko multi-cloud-en ingurune batean, non aktibatu da autokontrola AWS-n. , GCP eta Azure. Proiektuaren garapenerako, eta konkretuki, cluster-aren ecosistemaren kudeaketarako, kode irekiko hainbat tresna eta proiektu erabiltzen ditu Kubernetes, Kubebuilder, Kubeadm, Linux Ubuntu edo WireGuard garrantzitsuenen artean. Hornitzaileen baliabideak kontrolatzeko, proiektu hau AWS, GCP eta Azure programaziorako software-ko kitetan (SDK) oinarritzen da. Programazio-lengoaien konbinazio bat ireki zuen proiektua, baina nagusiki Go Language eta Shell Script erabili zituen.

PALABRAS CLAVE:

Computación en la Nube, Kubernetes, Plano de Control Universal, Kubebuilder, Multi-Cloud, Infraestructura como código, Autogestión

KEY WORDS:

Cloud Computing, Kubernetes, Universal Control Plane, Kubebuilder, Multi-Cloud, Infrastructure as Code, Self-management

GAKO-HITZAK:

Cloud Computing, Kubernetes, Kontrol Hegazkin Unibertsala, Kubebuilder, Multi-Cloud, Azpiegitura Kode gisa, Autokudeaketa

List of acronyms

AWS	Amazon Web Services
API	Application Programming Interface
VM	Virtual Machine
K8s	Kubernetes
GCP	Cloud Computing Services
VPN	Virtual Private Network
DNS	Domain Name System
DDNS	Dynamic Domain Name System
UCP	Universal Control Plane
CNCF	Cloud Native Computing Foundation
CR	Custom Resource
CRD	Custom Resourc Definition
SDK	Software Development Kit
SSH	Secure Shell
LB	Load Balancer
NAT	Network Address Translation
IP	Internet Protocol
IaC	Infrastructure as Code
PoC	Proof of Concept

Index of content

Figures index

Table index

1. Introduction

In the rapidly evolving scenario of modern technology, the concept of Cloud Computing has risen to the forefront, redesigning the way organizations deploy and manage their applications and infrastructure. Kubernetes in parallel is the most famous and adapted Cloud Computing orchestrator whose objective is to manage Cloud applications. Both of them have become indispensable tools for many businesses seeking scalability, agility, and efficiency in their solutions.

Cloud Computing, with its promise of on-demand access to a virtually limitless pool of computing resources, has revolutionized the way we think about IT infrastructure. The Cloud is not just a trend, it's a fundamental design pattern that makes organizations move away from traditional solutions to achieve a more flexible and scalable model for running their applications.

Kubernetes, on the other hand, has emerged as the de facto orchestration platform for containerized applications. It provides a powerful framework for automating the deployment, scalability, and management of containerized applications across multiple nodes independently of their location. With Kubernetes, businesses can grant scalability, high availability, and easy management to their applications, regardless of whether these applications run on-premises on local servers, in cloud providers such as AWS, or in hybrid environments.

In this era, where resilience and cost-effectiveness are a priority, understanding and adopting the core principles of Cloud Computing or Kubernetes is not merely an advantage; it's a necessity. This project will delve into these two concepts and will propose a solution where, taking advantage of the adoption of the cloud and Kubernetes, extends Kubernetes functionality for making the Cloud control its underlying infrastructure.

This project tries to achieve the idea of having a single API (Kubernetes API) to rule everything. Having a polyvalent tool that can control an entire cloud solution, from the microservices it is composed to the hardware that processes them. This concept could be referred as an “Universal Control Plane”.

2. Context

2.1. History of servers' management

At the beginning of the internet's development servers were significantly different from the system we know and use today. An engineer needed to buy a computer, configure it manually, install an operating system, connect it to the internet, and configure it to serve the desired purpose. This configuration did not adapt to the conditions of the moment, the computers were the same regardless of the amount of traffic it hosted. Hence, the engineer needed to configure the number of computers that were going to be able to handle peak moments of traffic, while in the downtime the computer would be using only part of their power. What is more, if the popularity of the service grew the engineering team needed to configure additional computers — which meant buying them, shipping them, configuring them… taking days or weeks at a time.

Later on, with the adoption of virtualization, this process evolved. Provider companies with high computing resources started to rent virtual computers also known as Virtual Machines (VMs). As a VM can be created in a matter of seconds, and it is possible to create several virtualized versions of varying sizes inside of a physical computer, it provides higher speed flexibility and optimizes the use of resources. The virtualization layer slightly reduces the total amount of computing power, however, it allows for much more efficient use of resources, saving in costs and providing better speed. For renting these VMs to the provider clients used graphical tools like web pages or computer applications.

Afterward, these providers developed Application Programmatic Interfaces (APIs). To understand better what the APIs are, saving the distances, languages are to humans what APIs are to computers, they allow computers and their programs to interact with each other. Provider APIs allowed to interact with them programmatically instead of doing it through graphical tools, which allowed users to control provider resources programmatically. Control the provider programmatically drastically increases the possibilities in terms of functionality, since one can create programs that contain a specific logic and make them interact with the provider. For example, it is possible to create sets of instructions (also called scripts) that can simultaneously automate the creation, deletion, or configuration of hundreds of VMs.

2.2. Imperative vs Declarative approach

This brings us to this project. The next logical step in this evolution of computer services management is to overcome the imperative approach and advance toward a declarative one. Imperative solutions consist in giving a set of instructions that will be executed, for example, “create,” “deploy,” “update,” “delete,” etc. On the other hand, a declarative solution defines the desired state of a specific service. To clarify the purpose, let us consider the following example:

If the necessity was to create 10 VMs spread across Europe, within an imperative approach, several steps will need to be done: choose a provider like AWS, configure it, pay it, install it, and execute the commands to create the servers in different locations of Europe like: "aws create vm size medium location Frankfurt ...", "aws create vm size medium location madrid...", ... and so on until the 10 VMs are created. In comparison, in a declarative approach, only one step is necessary, one that could look like a single command similar to this "vm create amount 10 location europe". The engineers have to take care of all the complexity and only allow the command to specify the most relevant information, like in this case, the number of VMs (10) and the continent (Europe). The program will take care of all the complexity of managing the providers and other decisions such as select what will be the specific location of the servers, like Madrid or Frankfurt, or the size that these servers will have, among other needed decisions.

The crucial difference is that in the declarative approach, a background service needs to be running continuously waiting for the orders, and more importantly, it must be able to understand them. Due to this, the complexity of this approach is significantly higher, as well as the time it requires to be programmed. Yet, since it runs without interruption, it ensures that the outcome desired, like having 10 VMs running across Europe, is always true. If it detects that the desired state is not true, it will work towards fixing it. To follow up on the example, if one of the VMs dies leaving only 9 VMs running, the program will detect the discrepancy and try to create another VM, with the objective of matching the real state to the desired state.

Therefore, while its programming is initially more time-consuming, the declarative approach certainly triumphs with its long-term benefits, not only by saving time but also by subtracting complexity, allowing people to use the tool without needing high expertise in the programming field. In our example, the instruction "vm create amount 10 location europe" is easier to execute by moderate users with less expertise in the field.

2.3. Kubernetes

The most successful tool at the time of writing that automates processes in a declarative approach is the open-source project Kubernetes [1], which was originally created for “container orchestration” a technology used in Kubernetes to implement microservices.

These microservices are an architectural approach directly linked with the concept of "The Cloud" more known in popular culture. Without entering into detail about microservices and "The Cloud" yet (described further), they are becoming nearly indispensable for well-designed applications, particularly at the "application layer." This layer is built over the "infrastructure layer," which until this point we had focused on. For example, in a computer that has Microsoft Windows installed, Windows would be the application and the computer itself would be the infrastructure. Both layers are necessary for providing internet services, like websites or applications. To automate and manage the application layer in a declarative approach, Kubernetes is the most adopted tool.

This project wants to get advantage of the fact that numerous companies are already using Kubernetes in their systems and expand Kubernetes functionality to automate the infrastructure layer in the same way they are already doing for their applications.

2.4. Multi-cloud solution

In addition to that objective, this project tries to break the dependency applications have on their infrastructure provider, and prove it by working over multiple cloud providers. To better understand the motivation behind this, these are the reasons.

On one hand, a multi-cloud solution offers numerous benefits. Among the most important ones are the following. Firstly, it reduces vendor dependency by providing flexibility to choose the most suitable services and pricing models from different providers. This helps mitigate the risk of becoming tied to a single provider's proprietary ecosystem, making it easier to adapt to changing business needs and cost considerations. And secondly, a multi-cloud strategy enhances resilience and redundancy. By distributing workloads across multiple providers, organizations can protect against failures or performance issues from any single provider. This redundancy ensures uninterrupted service availability and minimizes downtime, which is crucial for maintaining operational stability.

And, in there other, the most dominant infrastructure provider at the moment of writing is Amazon with their AWS solution, taking almost a 30 percent control of the market [2]. The biggest providers like AWS offer self-managed Kubernetes services that take care of the infrastructure for the client. The problem with this is that these self-managed solutions are designed to work only within the provider that delivers the service with few customization options which makes multi-cloud clusters' design very complicated. Sometimes these solutions even force the client to buy additional services from the provider due to the design of the Kubernetes service offered.

Because of the reasons just described, this project does not use these self-managed Kubernetes solutions, and it treats separately the new functionality the project aims to achieve from the providers themselves, making this project less dependent on any provider and giving it the possibility to work over multiple ones. This project will use the 3 leading infrastructure providers currently from Amazon, Google, and Microsoft; but because the provider's logic is separated from the rest of the project, it should be easy to migrate it to work over new different providers.

2.5. The Chicken and Egg Situation

In this context of making a cloud application control infraestructure there is the problem of the chicken and egg, which this project tries go give a solution as one of its main goals.

This problem refers to a situation where two interdependent components or processes rely on each other to function, but neither can start without the other already in place. In this scenario, one component is like the "chicken," and the other is like the "egg," and it's unclear which should come first.

For instance, consider a cloud application designed to manage and provision VMs in a data center. The chicken might be the application itself, which needs VMs to operate and provide services. The egg represents the VMs, which need the application to create and manage them. The problem arises because the cloud application needs VMs to be up and running to perform its tasks, but it also needs the application to exist in the first place to create those VMs. This circular dependency makes it challenging to get the system up and running smoothly.

3. Objectives

The main objectives of this project are to make a Kubernetes cluster able to control its underlying infrastructure and doing it building a reliable architecture avoiding a single point of failure across the whole infrastructure design. In addition to avoid a single point of failure at the provider level and to avoid dependency from any specific cloud provider, the architecture must be multi-cloud, working across different providers and not depending entirely on one of them. Having these objectives in mind the following specific objectives must be accomplished:

- Find and design the best way to control infrastructure natively from Kubernetes, with new K8s components.
- Implement infrastructure resource control for VMs of AWS, GCP, and Azure from these new K8s components to create.
- Find a way to solve the chicken-egg problem, for Kubernetes to be able to control its own infrastructure.
- Build a VPN for computers spread across the internet programmatically to build a cluster.
- Build K8s node architecture over this new cluster.
- Avoid a single point of failure for every component in the architecture.
- Make the architecture adaptable to changes and self-healable to errors that could happen in a real live environment.

4. Benefits provided

The present project proposes a solution that has inherent benefits because of the fact of using Kubernetes, controlling infrastructure from it, and having a multi-cloud solution.

4.1. Technical Benefits

Automated Infrastructure Management

Kubernetes excels at managing components automatically. Its core concept involves programming Kubernetes components to continuously monitor and self-heal. This means that the infrastructure adapts to changes and fixes errors on its own, saving human operators time. Instead of spending hours fixing issues, Kubernetes can resolve them within minutes. This results in a more reliable infrastructure compared to one controlled solely by humans.

Multi-Cloud Resilience

Using a multi-cloud cluster architecture prevents a single point of failure at the cloud provider level. If one cloud provider's services experience downtime, the infrastructure seamlessly adapts by shifting the workload to other providers. This redundancy enhances the overall resilience of the system.

Unified Control via Kubernetes API

In a cloud-based solution, Kubernetes controls the infrastructure through its API. This means that the entire application ecosystem has a single control API, making management more streamlined. All components can easily communicate with each other through this unified control API. Furthermore, integrating this control API with other systems becomes more straightforward. Many third-party tools already have built-in integrations with Kubernetes, allowing for seamless integration of infrastructure components.

Simplified Integration

When one Kubernetes cluster manages its own infrastructure, it leads to a more unified system. In contrast, if infrastructure management were in a separate environment, it would require additional effort and resources to create and manage. By keeping everything within a single cluster, all components are perfectly integrated, resulting in a more cohesive and efficient system.

4.2. Economic Benefits

The various technical advantages just discussed lead to several economic benefits:

Human Hours

Initially, setting up and configuring everything may require more human effort, leading to higher short-term costs in terms of human hours. However, the long-term benefits outweigh this initial investment due to the higher level of automation in the infrastructure. For instance, if the infrastructure experiences an issue on a weekend night, the company would need to pay engineers to work the extra hours to resolve it in the lowest time possible, incurring extra shift costs. Meanwhile, if this fix was automated it won't produce extra cost.

Infrastructure

This project's approach, which involves building and managing the infrastructure internally rather than relying solely on a service provider, results in substantial cost savings and greater flexibility. The hourly cost of VMs provided by any cloud service provider is typically lower than the cost of self-managed K8s clusters available in the providers. Additionally, the adaptable nature of this infrastructure allows for the automatic removal of unused resources, directly reducing operational costs. If these resources are needed again, they can easily be recreated. Moreover, the flexibility to build a hybrid cloud cluster means it can be integrated with cheaper infrastructure as could be, company local servers, potentially reducing costs.

Multi-cloud

Self-managed solutions offered by cloud providers are usually designed to function exclusively within their own ecosystems, making costs and downtime more dependent on the provider's offerings. In contrast, the solution presented in this project allows for the specification of the desired cloud distribution for the multi-cloud infrastructure. This means that if one cloud provider offers significantly lower prices than another, you can allocate a higher weight to the cheaper provider in the cloud distribution, while reducing or eliminating the weight assigned to the more expensive one. Additionally, a multi-cloud approach helps mitigate the consequences of downtime in a single provider's service. In situations where infrastructure is not multi-cloud, a service outage from the provider can result in substantial costs due to denied services for clients, damage to the company's reputation, extended downtime, and more.

4.3. Social Benefits

Open source

This project embraces open-source solutions, fostering a collaborative and inclusive community of developers and contributors. This collaborative ethos promotes knowledge sharing, skill development, and innovation, benefiting individuals and organizations alike. It encourages a culture of transparency and accessibility, making technology more equitable by ensuring that anyone can access, use, and modify the software.

Internet decentralization

The decentralization of infrastructure reduces the dependency on a few dominant cloud providers. This decentralization enhances the resilience of digital services against potential disruptions, whether caused by technical failures, cyberattacks, or other unforeseen circumstances. The way providers are integrated in the project, simplifies the process of including new ones. This democratizes access to computing resources, enabling smaller businesses and startups to compete on a more level playing field and reducing the risk of monopolistic practices in the industry.

5. State of the Art and Alternative Analysis

5.1. The cloud and microservices

The cloud and microservices are two fundamental pillars of modern computing that have revolutionized the way software applications are developed and deployed. The cloud refers to a network of remote servers hosted on the internet, which enables users to access and store data and applications over the web, eliminating the need for on-premises infrastructure. It offers scalability, flexibility, and cost-efficiency. On the other hand, microservices is an architectural approach where complex applications are broken down into smaller, and independent services that can be developed, deployed, and scaled independently. This design fosters agility, easier maintenance, and the ability to adapt to changing demands.

Both technologies complement each other and are deeply interconnected, with the cloud providing a flexible infrastructure necessary in the nature of microservices. Microservices, in turn, leverage the cloud's features to create a more efficient and flexible application architecture, and developers can build and deploy microservices independently with each service dynamically adjusting its resources according to the cloud's capabilities, optimizing the use of resources.

The Cloud Native Computing Foundation (CNCF) is a non-profit organization that was founded in 2015. It serves as a home for a variety of open-source projects specifically focused on cloud-native technologies. The foundation's primary objective is to support the development and adoption of cloud computing paradigms, and every year they launch reports on the state of cloud technologies. Looking at some reports they did in last years like on the one in 2022 [3], different trends indicate the adoption rise of cloud technologies every year. Like, the adoption of cloud technologies by companies or the increased use and knowledge by developers of these technologies.

5.2. Universal control plane APIs

The concept of a Universal Control Plane (or UCP) API is yet to be explored in previous papers, but it is occasionally mentioned in some mediums. In the opinion of the writer, the concept was born with the creation of Crossplane [4], which is an application that extends Kubernetes functionality, and that will be described in further detail ahead in this article.

A universal control plane API refers to a standardized API (or lenguaje) that allows interaction with all different components, services, and systems. This API serves as a central control point, enabling unified management, configuration, and coordination across all the diverse resources, regardless of their underlying technologies. Specifically for cloud solutions, there are two different systems that need to be controlled equally in other to have a UCP. On one hand, the containers, which are inherent to cloud applications, and on the other, the infrastructure where all these containers are running.

Only a few technologies seem to go in the direction of the concept of UCP. On one hand, there are several solutions highly focused on infrastructure management, and on the other, a few solutions focused on cloud application management.

The chosen technology needs to fit different criteria. First, it has to be widely used and have a promising future since the idea behind this project is to achieve a concept that will make the future of the cloud and automation easier and more versatile. Second, it has to be a flexible solution since this project aims to extend its functionality to become a UCP. And third, it will make the development of the project easier the fact that the technology is easy to use.

5.2.1. Infraestructure oriented solutions

There are many technologies that facilitate the management of infrastructure with very distinct approaches. In the book "Terraform: Up & Running" [5], a chapter contains a comparison of all the most relevant infrastructure management technologies. They are compared in different aspects such as community, simplicity, approach, etc. These technologies are Chef, Puppet, Ansible, Pulumi, CloudFormation, Heat, and Terraform.

From the comparison offered and having in mind the requirements described before, most of the solutions can be discarded, and only Ansible and Terraform seem to be the better candidates. Both of them share in common that they have the biggest communities which is crucial in terms of use and versatility. The community grants wider support across services offered by the providers and easier debugging when facing complications with their implementations.

The main differences between these solutions are the next ones. While Terraform approach is a declarative one similar to the one that is described in the introduction of this article, the Ansible approach is more imperative or procedural as it is described. The approach of Ansible makes it more complicated to use but its complexity gives it more flexibility in terms of possibilities and functions extensibility. Terraform on the other hand is simpler and easy to use, and its declarative approach makes the maintenance of the services easier.

5.2.2. Cloud oriented solutions

On the other side, there are solutions designed to manage the cloud. These solutions are also called container orchestration tools in a more technical name. This name is due to the fact that managing the cloud means coordinating and managing (orchestrating) all the containers, which saving the distances are the microservices, that form a cloud application.

The paper “Overview of Docker container orchestration tools” [6] goes through the main orchestration tools available at the moment and describes their benefits and disadvantages. It tasks about three solutions which are Kubernetes, OpenShift, and Docker Swarm. And having in mind the requirements seems that the better candidate is Kubernetes, due to the fact that is the one more used and the one with the biggest community. It is very important, especially for an open-source project to have a huge and active community behind otherwise the project will end up dying, as probably will end up happening for Docker Swarm. OpenShift is a privative solution based on Kubernetes itself with some additional functionality, but less used and less popular than Kubernetes. Also to remark that this project tries to exclude as many privative technologies while there is a good open-source alternative.

The community behind Kubernetes is very active and this makes Kubernetes a very versatile project with lots of different functionalities. For example, other tools for data visualization or GitOps tools, among many others, offer an integration with the API of Kubernetes. This means that if this project chooses Kubernetes many of these tools, like visualization or management ones, could be used alongside this universal control plane, making it easier or faster to work with. Also, the Kubernetes community already has worked on different sub-projects made to either control infrastructure or extend Kubernetes functionality, projects that will come in handy to achieve the objective of this project and that will be analyzed further in this paper.

5.2.3.Best solution

Even though there are a few different alternatives, and tools like Terraform or Ansible are compatible with a wide range of services and seem closer to the idea of a UCP, in the opinion of the writer it is very obvious that the best solution is Kubernetes. Here are described a few reasons why.

The first reason is the huge utilization by people and companies and the wide community this brings. Being constantly growing and adapting to new challenges and with great quality in the design. Kubernetes is one of the most prominent projects of the CNCF and every year the launch report about Kubernetes trends like the one of June 2023 [7], where it is visible how the project keeps growing year by year in contributions, people and companies participating in the project.

And the second reason is that the complexity inherent to a cloud orchestrator like Kubernetes is significantly higher than the complexity behind managing infrastructure resources or services. A tool like Kubernetes is composed of several internal microservices that will need to be reprogrammed in order to replace its functionality. Kubernetes is a more complex solution because it solves a more complex problem. In these two pictures, it is visible the internal difference between an infrastructure provider like Terraform and Kubernetes.

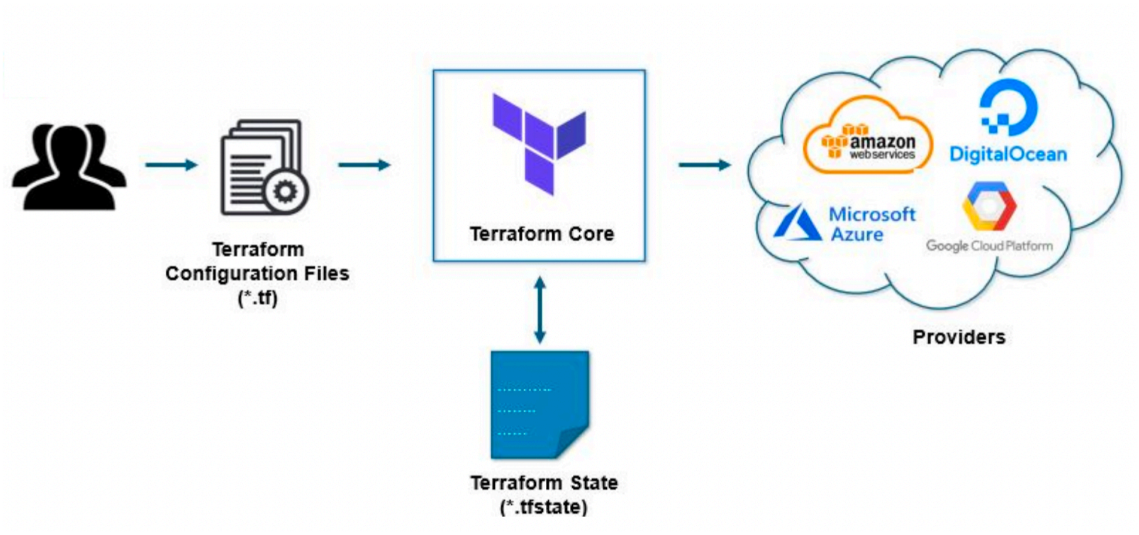


Figure 1. Terraform components. [14]

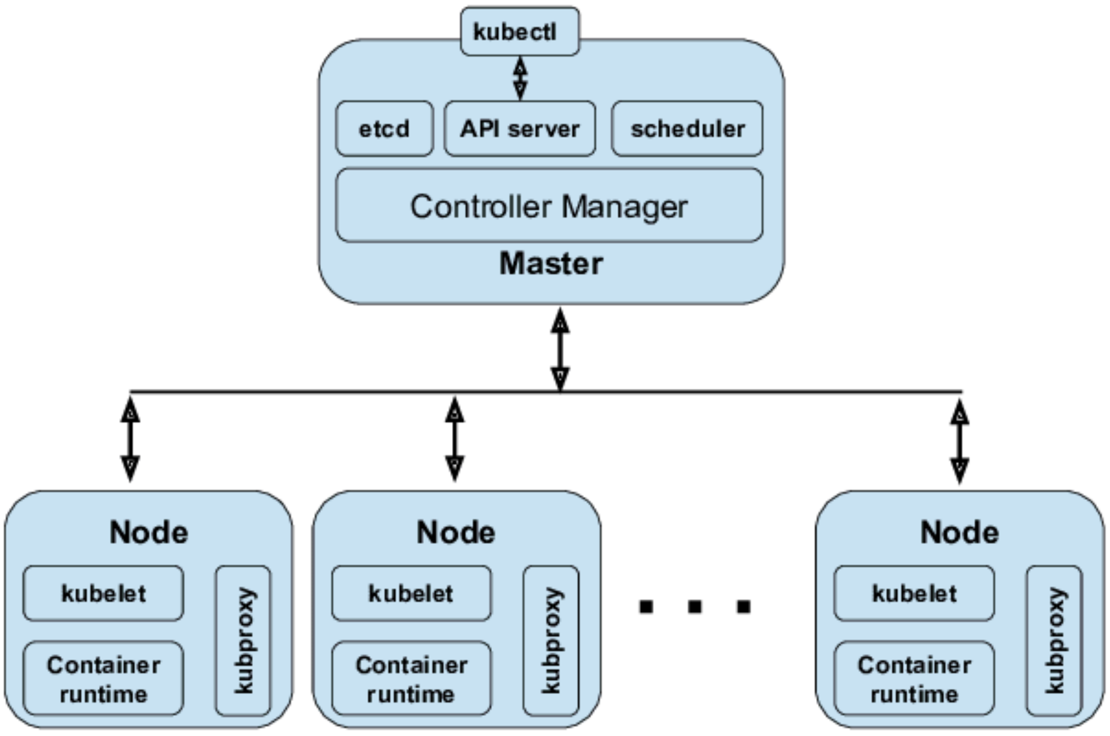


Figure 2. Kubernetes internal core components. [15]

So in other words, it seems quite easier to replace Terraform/Ansible functionality inside Kubernetes, than the other way around.

5.3. Kubernetes to control infraestructure

Once justified that Kubernetes is the best solution to achieve this project objective, a harder decision needs to be made. The idea to control infrastructure from Kubernetes itself isn't original from this project, in fact, several projects already exist built around Kubernetes to control infrastructure inside of Kubernetes. - The originality or innovation of this project relies on making a Kubernetes instance control its own infrastructure over multiple providers - The following projects make a Kubernetes instance control other infrastructure, but not its own. This way the control of the infrastructure over which Kubernetes runs needs to be done apart, breaking with the idea of a UCP that was described before in this paper and that this project tries to achieve.

The three main projects that already exist and that can be used to control infrastructure are Cluster API, Crossplane, and Kubernetes API.

5.3.1.Cluster API

Cluster API is a Kubernetes sub-project focused on providing declarative APIs and tooling to simplify provisioning, upgrading, and operating multiple Kubernetes clusters, hosted by the CNCF. [8] [9]

Even though it is a good solution in many use cases it has two main problems. On one hand, the project doesn't seem very mature at the time of writing compared with other solutions, that solve the problem of creating and managing infrastructure for Kubernetes clusters, but outside of Kubernetes itself. And on the other hand, it is a complex project that requires a huge understanding in order to use it in an effective way. [10]

Additionally, there is another solution that tries to solve a similar problem but it is more ambitious that also aims to bring a solution to many other new use cases. This solution is Crossplane which is described in further detail in the following section.

5.3.2.Crossplane

Crossplane is an open-source project, also acquired by the CNCF, that extends Kubernetes by enabling the management of cloud services and infrastructure through the Kubernetes API. It provides a declarative way to define and manage resources across multiple cloud providers, allowing developers to provision and manage services seamlessly, abstracting away the complexities of different cloud APIs.

Very similar to what Terraform does but with the difference that Crossplane is running inside Kubernetes. The major advantage of this is that this allows Crossplane to run all the time in the background, so it can constantly ensure that the state of the wanted service matches the desired state, and fix it if it doesn't match. The similarity with Terraform is such, that the Crossplane community built a project called Terrajet. This project consists of making Crossplane able to understand Terraform code and act as a translator between both technologies, which drastically increased Crossplane's provider’s services support, taking advantage of the huge community behind Terraform.

5.3.3. Kubernetes API

Kubernetes API extension capabilities refer to the ability to customize and enhance the Kubernetes API to support new functionalities beyond its core features. Kubernetes offers a powerful extension mechanism that enables developers to create Custom Resource Definitions (CRDs) and Custom Controllers. CRDs allow users to define their custom objects for their new resources. These custom resources can represent anything from applications, services, configurations, infrastructure, or any other entity.

CRDs work with Custom Controllers, to manage the lifecycle of these custom resources. Custom Controllers use the Kubernetes API to watch for changes to custom resources, and track the real state of the entity they represent. And if something isn't as it is supposed to be, they take action accordingly to fix it. This allows the automation and orchestration of complex workflows and empowers users to automate processes using Kubernetes.

To be able to program these custom resources and their controllers, Kubernetes Operators exist. There is a huge market of operators with different focuses and designs that will be discussed in the following section.

5.3.4. Best solution

From the options above seems like Crossplane was the better candidate to control infrastructure. Because it is a technology built to control any kind of service in Kubernetes, and its apparent simplicity, especially compared to Cluster API. Initially in this project, Crossplane was the chosen option to develop this project, but due to the following reasons it was discarded and this project was developed with Kubernetes API extensibility directly.

The first reason is the fact that Crossplane is a very young project and its native support of services is very limited. The main resource this project needs to control is VMs, but it wasn't supported by any of the 3 main providers. Since the project Terrajets exist, it was tried to use it for being able to control VMs, but due to the fact that Terrajet is even a younger project the lack of community and support made the development almost impossible.

And the second huge reason is that the customization level necessary to achieve what this project want to achieve wasn't enough with either Cluster API or Crossplane. To make a Kubernetes cluster control its own infrastructure, as is the goal of this project, it is necessary to manage and create the VMs over the ones Kubernetes will run. Both Cluster API and Crosspalne can create these machines but the customization over these ones isn't enough. To better understand this, creating VMs is a mandatory service required for this project, but while all these tools can create these VMs, it is not enough with that. Kubernetes needs to configure its networking, include it in a VPN, join them to the Kubernetes cluster, solve the chicken-egg problem, monitor status parameters, etc. And doing this with Crossplane seemed too difficult, and it was decided to work with the Kubernetes API directly to create custom resources. It is initially more complex to learn, but the inherent flexibility it provides will make this project easier to develop.

5.4. Other relevant decisions

Although there are other aspects of the project that require choosing a technology to work with, the impact on the overall outcome is relatively minor. Due to the fact that choosing alternative technologies would yield almost identical results. Still, a decision needs to be made, and here it is explained why in each of them.

5.4.1. Kubebuilder vs other operators

To create custom resources in Kubernetes there are over a hundred different Kubernetes Operators that can be used to create them. Even though all of them cover a similar functionality, only 2 options were considered, the 2 most popular ones: the Operator SDK developed by RedHat, a company with a huge reputation in the Linux ecosystem, and Kubebuilder, the operator framework developed by the Kubernetes community.

Due to the presence of sizable communities and the shared objective of addressing the same problem, both Operator SDK and Kubebuilder decided to collaborate and pool their efforts. As a result, the Operator SDK now builds upon the Kubebuilder project, effectively expanding its capabilities. While both tools are excellent open-source solutions, the choice was made to adopt Kubebuilder as it serves as the foundation for both projects and because is the project that belongs to the Kubernetes community. Nonetheless, it's worth noting that a similar outcome could likely have been achieved with the Operator SDK as well.

5.4.2. Kubeadm to build the cluster

Similarly, for managing a Kubernetes cluster, several options exist that could be used. But Kubeadm [11] was the chosen option due to its simplicity, robustness, and alignment with the Kubernetes project because it also belongs to Kubernetes. Kubeadm simplifies the process of bootstrapping and maintaining a cluster, making it accessible to both beginners and experienced users. It follows best practices and leverages the official Kubernetes components, ensuring compatibility and stability. While other alternatives may offer additional features and customizations, Kubeadm's focus on core functionality ensures a reliable foundation for a Kubernetes deployment, reducing complexity and minimizing the risk of compatibility issues. As a result, Kubeadm remains an optimal choice for organizations seeking a straightforward and well-supported approach to efficiently manage Kubernetes clusters.

5.4.3. Wireguard to build the VPN

Among all the different VPN protocols, several options exist for building a VPN infrastructure, with some of the most widely used being OpenVPN, IPsec, and Wireguard. However, after consideration, Wireguard was chosen as the better fit for this project. Wireguard is as a relatively new and increasingly popular choice, primarily due to its exceptional performance, and simplicity in both usage and management. These attributes make Wireguard’s future very promising and also make it a highly attractive solution for this project. [12]

5.4.4. SSH, and Shell Scripts to manage the nodes

For managing the nodes SSH and Shell Script offer a straightforward, and widely supported approach to interact and automate tasks on the nodes of the infrastructure. SSH provides secure remote access, enabling fluid communication and control over the nodes without exposing sensitive information. Additionally, Shell Scripts offers a lightweight and versatile means of automating repetitive tasks, such as deploying or configuring the nodes. This combination is a powerful toolset that requires minimal setup and provides ease of maintenance.

6. Deployment Description

6.1. Kubernetes Custom Components Design

6.1.1. Kubernetes API extension capabilities with Custom Resources

In recent years, Kubernetes has evolved beyond being a simple container orchestration platform to becoming an extensible platform for managing diverse workloads and applications. One of the key features that enables this extensibility is the ability to define Custom Resource Definitions (CRDs). CRDs allow users to extend the Kubernetes API by introducing their own custom resources and controllers. These controllers, created using tools like Kubebuilder, enable the automation and management of these custom resources.

6.1.2. Reconciliation Loop

When creating new custom resources, the most part of the programming relies on the reconciliation process or loop. This reconciliation loop is a crucial concept in Kubernetes controllers. It ensures that the actual state of resources matches the desired state specified in the custom resources. The following figure helps to understand better this process:

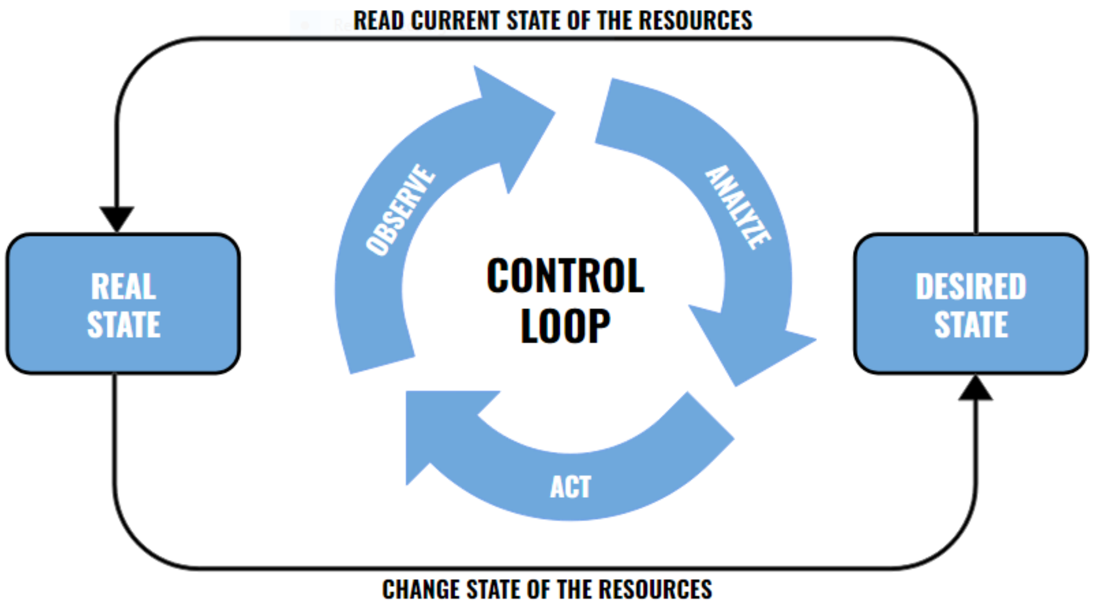

READ CURRENT STATE OF THE RESOURCES
REAL STATE
OBSERVE
ANALYZE
CONTROL LOOP
ACT
DESIRED STATE
CHANGE STATE OF THE RESOURCES

Figure 3. Kubernetes Reconciliation Loop.

This critical process is what enables Kubernetes to maintain the desired state, detect any discrepancies, and autonomously take corrective actions as needed, making it ideal for building robust and self-managing applications. As Kubernetes continues to evolve, the reconciliation loop in CRDs extensibility remains the central core concept.

6.1.3. Custom Resources Created for this Project

In this project, two custom resource definitions, namely "Vm" and "VmGroup," have been created using Kubebuilder. These CRDs extend the Kubernetes API to accommodate the specific requirements of the project.

— **Vm**: This custom resource definition represents a virtual machine instance, allocated in any of the available providers.

— **VmGroup**: This custom resource manages all the Vm that this project is using, building, managing, and adapting the cluster over them, where Kubernetes runs.

The following figure helps to understand the relation between these two CRDs and why both of them are necessary to manage the K8s cluster.

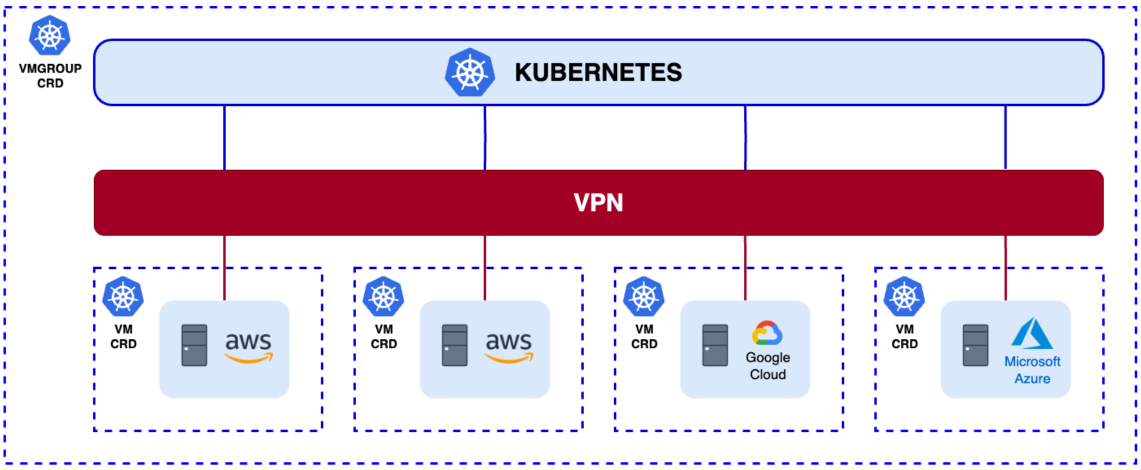


Figure 4. VmGroup and Vm Design.

The creation of a VmGroup involves the creation of several Vms which the VmGroup configures for them to be able to belong and communicate in the VPN and it builds the K8s cluster over them.

6.1.4.Vm CRD

For creating a Vm it is necessary to specify some Specs that act as input variables that are necessary to create the VM. The necessary Specs are the next ones:

— **Provider:** It will be the provider where the VM will be created, either AWS, GCP, or Azure

- **Role:** The role the VM will play. The options are either, Load Balancer, K8s Master, or K8s Worker
- **NetworkNumber:** The Network Number is the number in the VPN network that the VM will have, it depends on the role the VM is made for, and the Network Number already used by other VMs. The IP assigned will look like this "**10.0.0.NetworkNumber**".

The following diagram shows the reconciliation process of the Vm controller, in charge of the maintenance of the Vm CRDs. This reconciliation process starts when a Vm CRD is defined and ends when it is deleted. These two actions are done by the VmGroup reconciliation process explained later.

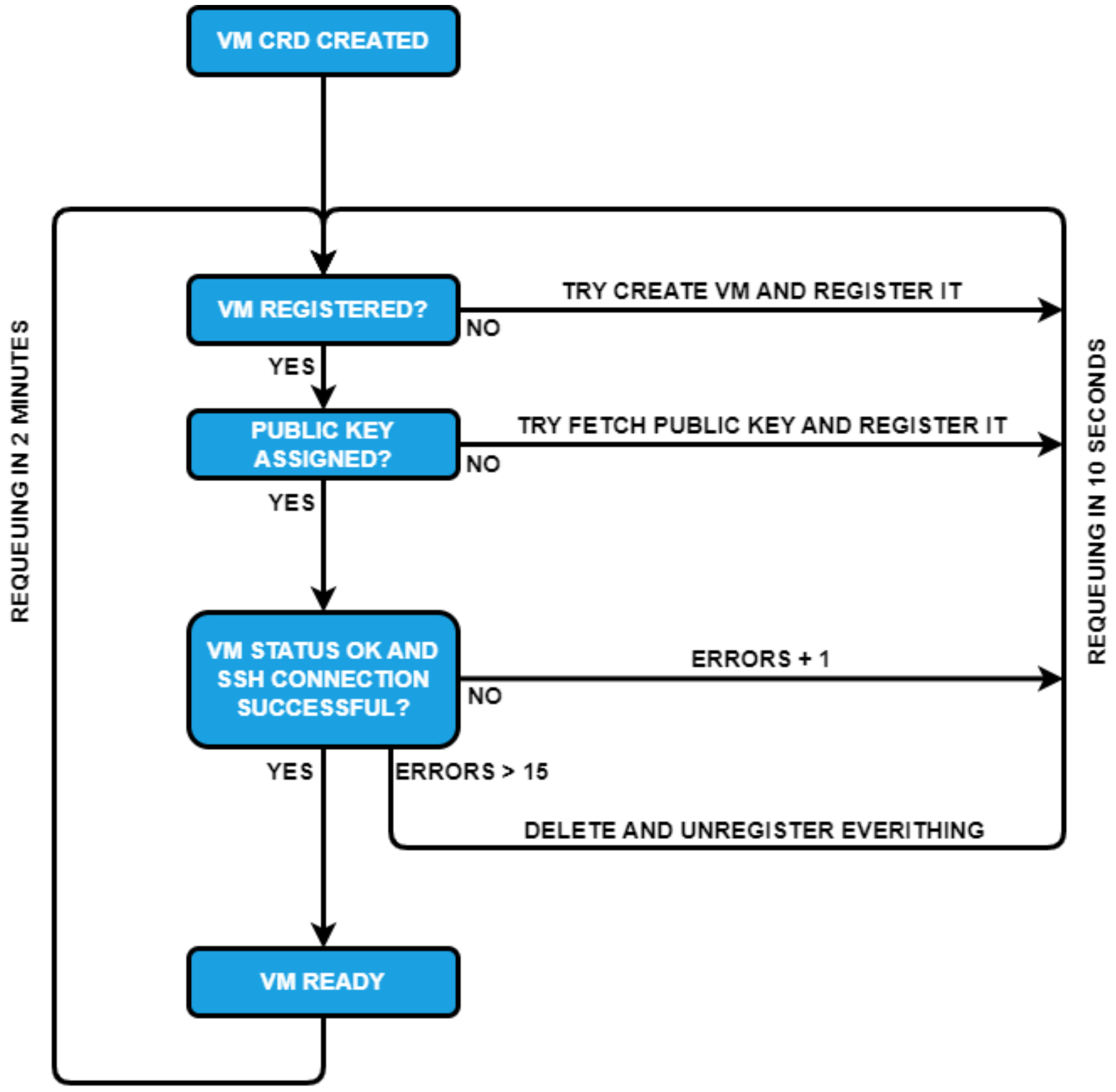

VM CRD CREATED
VM REGISTERED?
TRY CREATE VM AND REGISTER IT
NO
YES
PUBLIC KEY ASSIGNED?
TRY FETCH PUBLIC KEY AND REGISTER IT
NO
YES
VM STATUS OK AND SSH CONNECTION SUCCESSFUL?
ERRORS + 1
NO
YES
ERRORS > 15
DELETE AND UNREGISTER EVERITHING
VM READY
REQUEUING IN 2 MINUTES
REQUEUING IN 10 SECONDS

Figure 5. Vm CRD Reconciliation Process.

To encapsulate the essence of this diagram, the creation of a Vm CRD serves the purpose of generating a VM within the designated provider. Once this creation process is completed, the system attempts to establish a connection with the public key specified during VM creation. If this connection is successfully established, the VM is marked as "Ready." Subsequently, the system continuously monitors the VM's status, conducting checks at two-minute intervals.

In the event of any anomalies or issues, the error count is incremented, prompting the system to initiate corrective measures within a 10-second timeframe, trying to rectify the situation and restore normal operation.

6.1.4.1. *Vm Creation Scripts*

When VM custom resources initiate an order to a provider to create a virtual machine, they include an initial configuration script that ensures the VM is prepared for integration into the cluster. This script may vary from one provider to another due to differences in default configurations and also depending on the tipe of the node (LB, master, or worker), but it generally includes the following essential steps for each machine during its initial configuration:

- **Assignment of a Master Private Key**: A master private key is assigned to the VM. This key is used as a master key for SSH all the nodes in the cluster, but its use is limited to the K8s VMGroup instance. It provides a secure way to control for cluster.
- **Installation of VPN and Key Pair Generation**: A Virtual Private Network (VPN) is installed on the VM, and a unique set of private and public keys is generated. These keys play a crucial role in securing communication within the cluster.
- **Initial VPN Configuration**: The VPN is configured with the VPN IP assigned by the VMGroup. This step ensures that the VM can securely communicate within the cluster through the established VPN tunnel.
- **Enabling Forwarding in the Node**: Network packet forwarding is enabled on the VM node. This is a necessary step for proper Kubernetes functionality and the correct behavior of cluster load balancer nodes. It allows the VM to efficiently route network traffic as required by the Kubernetes cluster.
- **Installation of Kubernetes Dependencies**: Finally, the VM's dependencies are installed to enable it to join and function as a Kubernetes node. These dependencies typically include containerd (container runtime), kubelet (Kubernetes node agent), kubeadm (Kubernetes administration tool), and kubectl (Kubernetes command-line tool). These components are essential for the VM to participate in the Kubernetes cluster, manage containers, and execute Kubernetes commands. This step is skipped for the load balancer because they do not belong to the K8s cluster.

6.1.5. VmGroup CRD

For creating a VmGroup it is necessary to specify some Specs that act as input variables as in the Vm CRD. But in this case, the necessary Specs are the next ones:

- **MasterNodes:** The number of masters wanted in the cluster.
- **MinimumWorker:** The minimum number of workers wanted in the cluster.
- **MaximumWorkers:** The maximum number of workers wanted in the cluster.
- **CloudDistribution**: The distribution of VMs across the different providers. It is a struct composed of 3 integers AWS, GCP, and AZURE. The number specified in them represents the distribution, and if one is 0 that provider won't be used.
- **OriginNode**: An object that is composed of the id of the first VM used to initialize the cluster and the provider to which this VM belongs to. The use of this parameter will be explained later in the section "Solving the chicken and egg problem".

The diagram presented below illustrates the reconciliation process of the VmGroup controller, responsible for overseeing the maintenance of VmGroup CRDs. Similar to the VM controller, this reconciliation process begins when a VmGroup CRD is initially defined and concludes when it is deleted. However, in contrast to Vm CRDs, these two actions are initiated and managed by human operators interacting with the Kubernetes system.

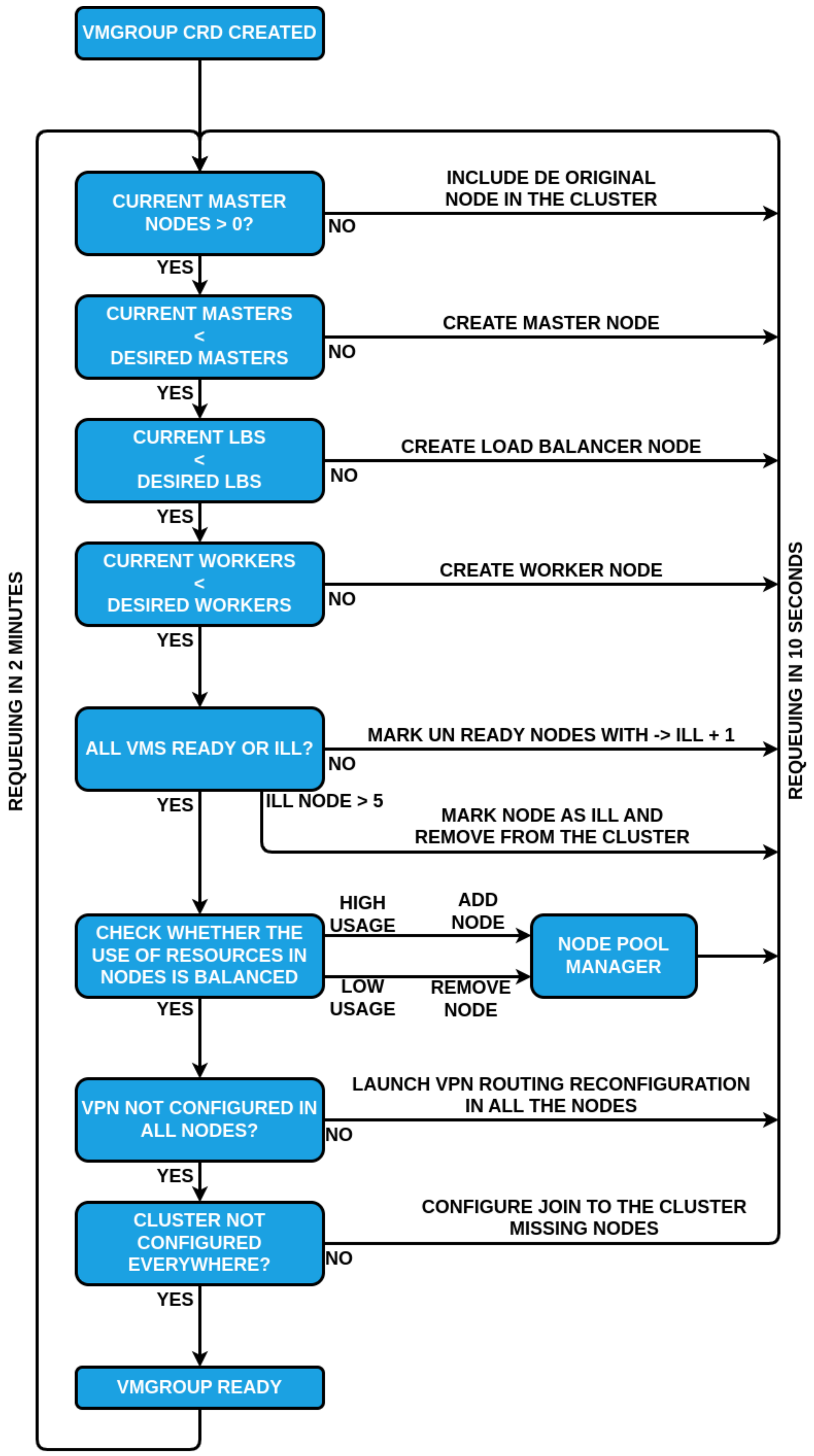

VMGROUP CRD CREATED
CURRENT MASTER NODES > 0?
NO
INCLUDE DE ORIGINAL NODE IN THE CLUSTER
YES
CURRENT MASTERS < DESIRED MASTERS
NO
CREATE MASTER NODE
YES
CURRENT LBS < DESIRED LBS
NO
CREATE LOAD BALANCER NODE
YES
CURRENT WORKERS < DESIRED WORKERS
NO
CREATE WORKER NODE
YES
ALL VMS READY OR ILL?
NO
MARK UN READY NODES WITH -> ILL + 1
ILL NODE > 5
MARK NODE AS ILL AND REMOVE FROM THE CLUSTER
YES
CHECK WHETHER THE USE OF RESOURCES IN NODES IS BALANCED
HIGH USAGE
ADD NODE
LOW USAGE
REMOVE NODE
NODE POOL MANAGER
YES
VPN NOT CONFIGURED IN ALL NODES?
NO
LAUNCH VPN ROUTING RECONFIGURATION IN ALL THE NODES
YES
CLUSTER NOT CONFIGURED EVERYWHERE?
NO
CONFIGURE JOIN TO THE CLUSTER MISSING NODES
YES
VMGROUP READY
REQUEUING IN 2 MINUTES
REQUEUING IN 10 SECONDS

Figure 6. VmGroup CRD Reconciliation Process.

In summary, the creation of a VmGroup CRD involves a multi-step process. First, it establishes a connection with the original node used to initialize the cluster, a process detailed in the "Solving the Chicken Egg Problem" section. Subsequently, the VmGroup CRD interprets the specifications provided during instance creation. This interpretation determines the required number of LBs, master nodes, and worker nodes. If any of these components are missing, the VmGroup CRD generates them by creating Vm CRD instances, as previously explained.

Once all the essential VMs are created and reach the 'ready' status, the VmGroup CRD conducts resource utilization checks. If there is an imbalance in resource allocation, it takes appropriate action, such as creating or deleting worker nodes to maintain balance. Following this, the VmGroup CRD verifies that all VMs are correctly configured within the VPN. If any misconfigurations are detected, it initiates the VPN Propagation Protocol method, as detailed in Figure 14.

After confirming that the VPN is correctly configured across all VMs, the VmGroup CRD checks whether all VMs are part of the cluster. If any VMs are not part of the cluster, they are added accordingly. Finally, the VmGroup CRD enters a loop to continuously monitor the status of the cluster at two-minute intervals. If any issues arise, it switches to a more frequent 30-second re-check cycle.

6.2. Cluster architecture

The main goals in designing the architecture of the cluster are the two following:

- **Eliminating Single Points of Failure:** This goal revolves around ensuring that the architecture remains resilient even in the face of component failures. Single point of failure refers to the situation when in an architecture the failure of a single component can break the complete architecture down.
- **Multi-cloud Architecture:** Another key objective of this project is to distribute the architecture across multiple service providers. This approach aligns with the first goal, as spreading the workload across various providers prevents a complete system failure in the event of a provider experiencing downtime.

6.2.1. Single Node Architecture

To understand the final architecture of the Kubernetes cluster and the factors that led to its design, let's begin by delving into the foundational concept of a single-node cluster. Where the control plane and the applications themselves run in the same node.

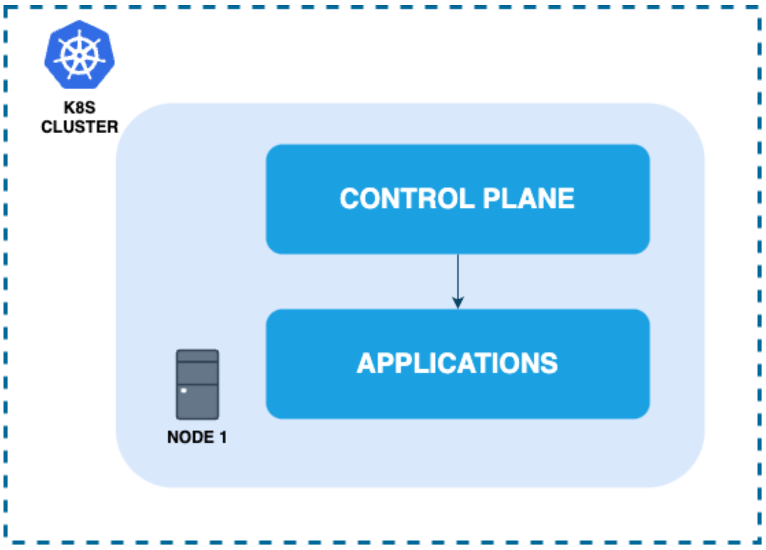


Figure 7. Single node K8s architecture.

However, this architecture is not flexible to changes and is neither robust since it supposes a single point of failure for all the components. If this node fails, everything falls with it.

6.2.2. 1 Master / 1 Worker Architecture

Kubernetes was conceived with the separation of control plane nodes and worker nodes in mind. The control plane nodes, known as Master nodes, are responsible for managing everything unrelated to the applications themselves, including provisioning, networking, security, and more. On the other hand, the Worker nodes host the applications and can dynamically scale the number of instances based on varying demands. Following this characteristic, the next step in this architecture is to use 2 different nodes 1 Master for the control plane and 1 Worker for the applications.

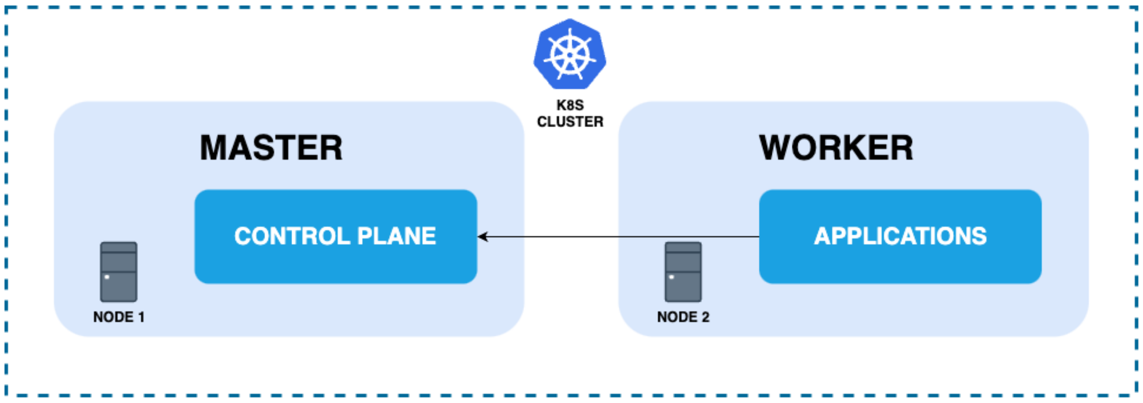


Figure 8. 1 Master / 1 Worker - K8s architecture.

The benefits of this architecture compared with the previous one is that the control plane can take care of and ensure the good behavior of the Worker node, if it fails or something doesn't work as expected the control plane node will try to fix it. However, the control plane won't be able to heal itself, so the single point of failure at this point is on the Control plane node.

6.2.3. Multiple Masters and a Load Balancer Architecture

The next step to avoid a single point of failure in the control plane is through several masters. Kubernetes allows to work with several control planes natively, only a Load Balancer (LB) is necessary between the workers and the masters so that the load balancer can choose to route the request to a working master. The architecture would look like this.

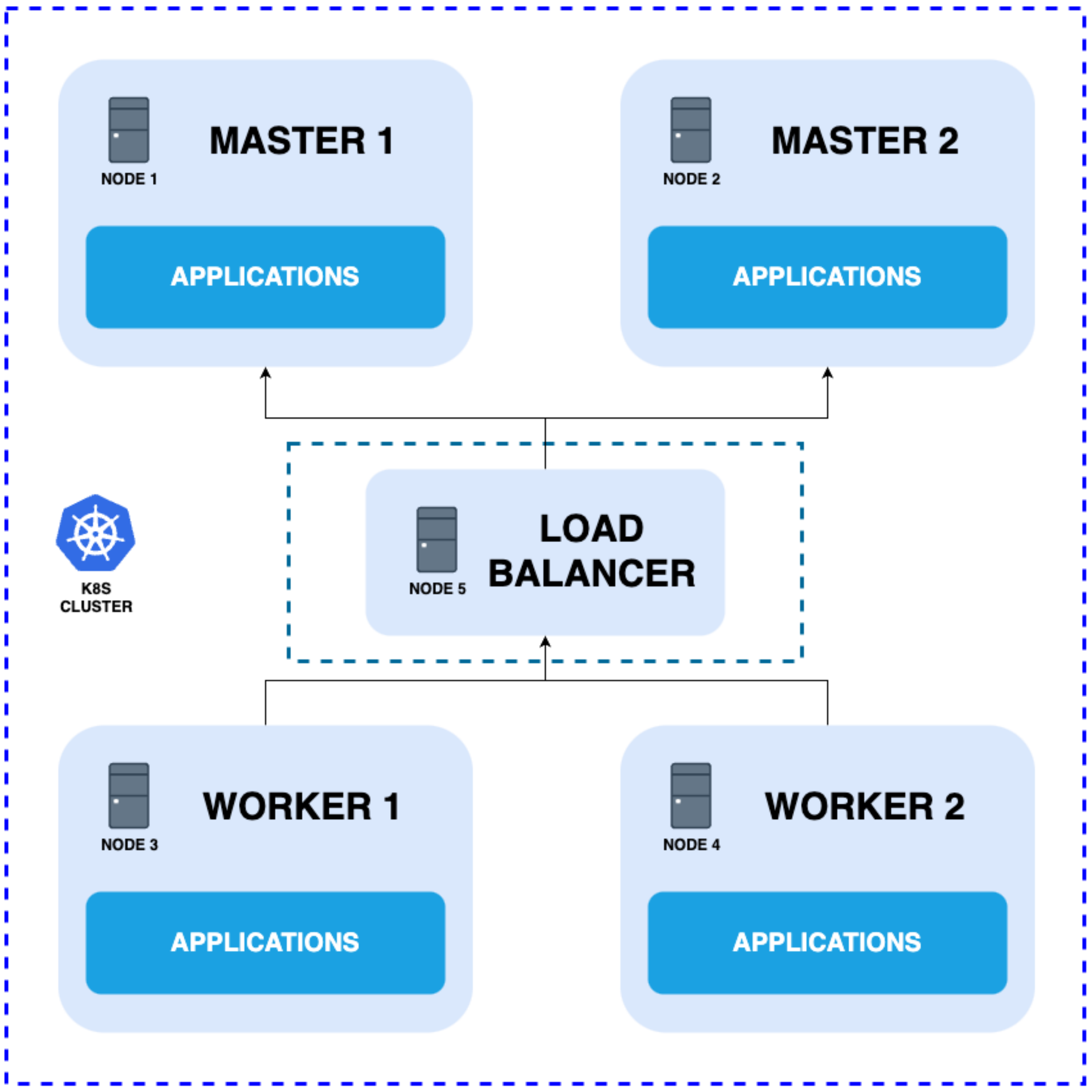


Figure 9. 2 Masters / 2 Workers / 1 LB - K8s architecture.

At this point, the architecture still has a single point of failure. But this time it is the load balancer, if it falls the workers won't be able to connect with the masters.

6.2.4. High Available Architecture with multiple LBs and multi-cloud Architecture

Because of this, the next step is to create several load balancers, so that if one fails another load balancer can replace it. There is a mechanism used to manage the load balancers with a "Floating IP" explained in a further section of the paper. The following figure, also shows how the architecture will look like in a multi-cloud environment with 3 Masters, 3 Load Balancers, and 3 Workers, one of each in each provider.

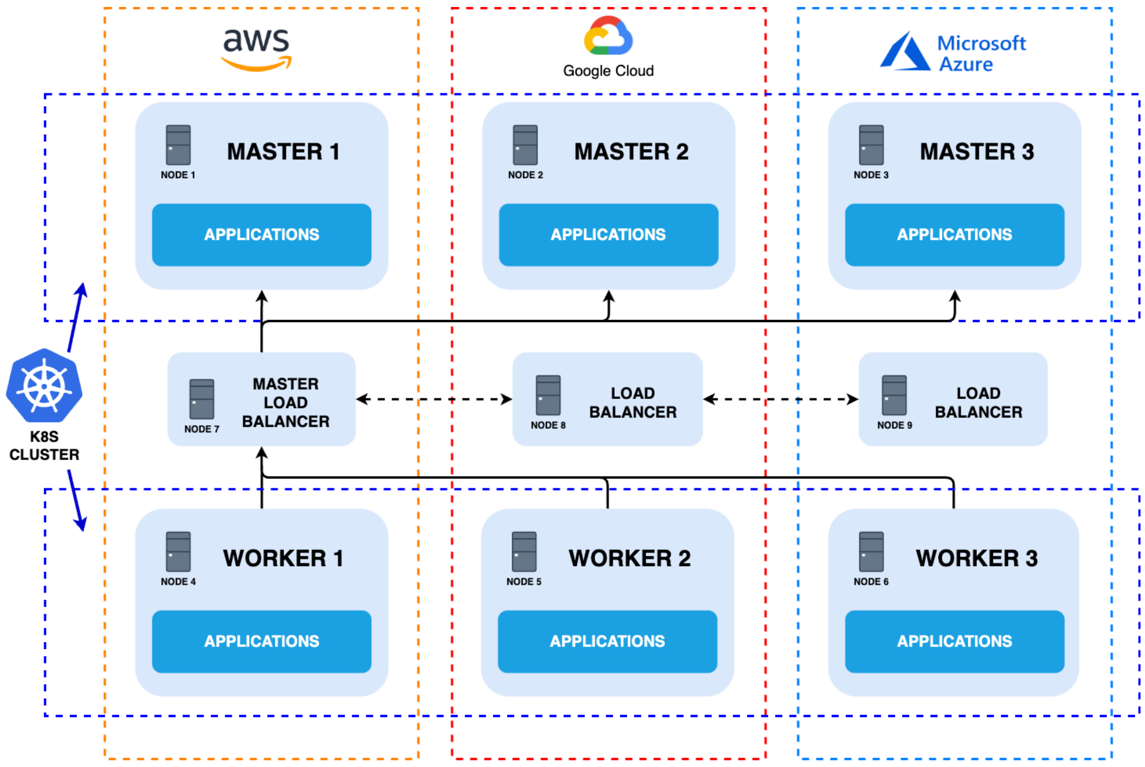


Figure 10. Final K8s High Availability Architecture

At this point, the architecture boasts a robust design free from any single points of failure. Multiple master nodes work in tandem to ensure high availability across the entire infrastructure. The Master's VmGroup controllers monitor the status of all nodes, including the masters themselves, and take corrective actions in any event of failure.

VmGroup will also add additional workers if the use of resources is very high and scale down when resource utilization is low. Moreover, in the unfortunate event of one provider's failure, VmGroup seamlessly deploys additional nodes from alternative providers to maintain uninterrupted service until the affected provider is back online.

6.3. VPN Management and Control Network

6.3.1. Wireguard basic behavior

WireGuard is a modern and efficient VPN (Virtual Private Network) protocol that allows two or more computers (referred to as peers) to securely communicate over an untrusted network, such as the Internet. It's designed for simplicity and performance while maintaining strong security. The following figure illustrates a basic communication between 2 computers A and B.

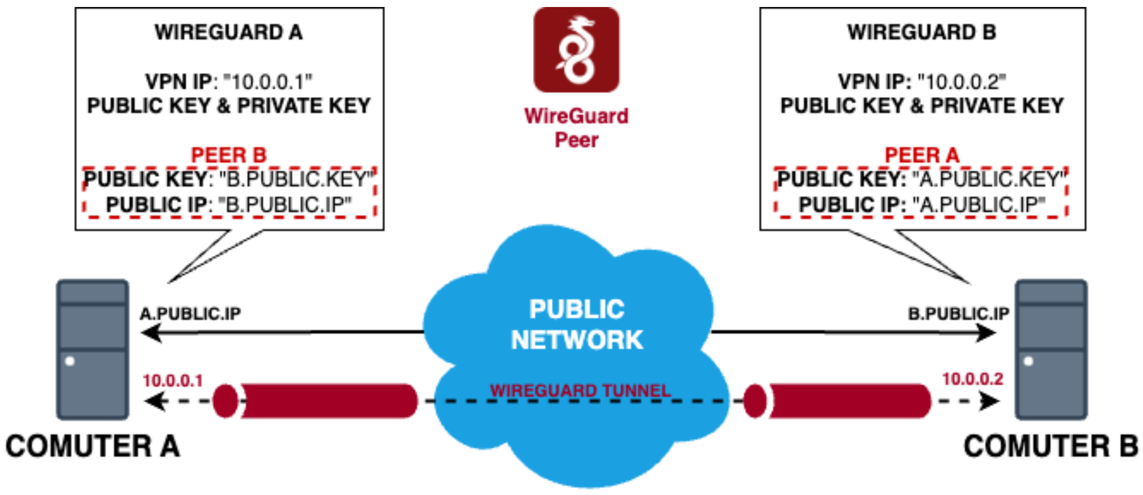


Figure 11. WireGuard Single Peers Communication

In order to establish a secure and private connection between two computers using a Virtual Private Network (VPN), several key pieces of information are required. These details facilitate the creation of a secure tunnel that allows data to flow between the two peers while maintaining confidentiality and integrity. The three essential pieces of information necessary for this connection are:

- **VPN IP Address:** Each computer involved in the VPN connection is assigned a unique VPN IP address. The VPN IP address acts as an identifier for the specific computer within the VPN network, enabling data to be directed to the correct peer.
- **Public IP Address:** The public IP address of the computer on the other end of the VPN tunnel is essential for routing data across the internet. It serves as the endpoint for the VPN tunnel, allowing data to be transmitted to and received from the correct location. The public IP address acts as a network address translation (NAT) reference, enabling data packets to traverse the public internet and reach the intended destination.

— **Public Key:** Public key cryptography is a fundamental element of VPN technology. Each computer generates a pair of cryptographic keys: a public key and a private key. The public key is shared openly with the other peer, while the private key remains confidential. The public key is used to encrypt data that is sent to the other computer, ensuring that only the corresponding private key holder can decrypt and access the information. This asymmetric encryption method enhances the security of the VPN connection.

When both computers possess these three pieces of information about each other, the VPN connection can be established.

6.3.2. VPN Configuration

Once we have a clear understanding of how WireGuard functions, we can delve into how Kubernetes VmGroup custom resource initially sets up the VPN on each of the computers. The initial landscape consists of multiple VMs spread across various cloud providers, each possessing distinct public IP addresses and geographical locations. In some cases, these machines may even be situated on different continents. The visual representation below illustrates better this scenario of VMs originating from three distinct cloud providers.

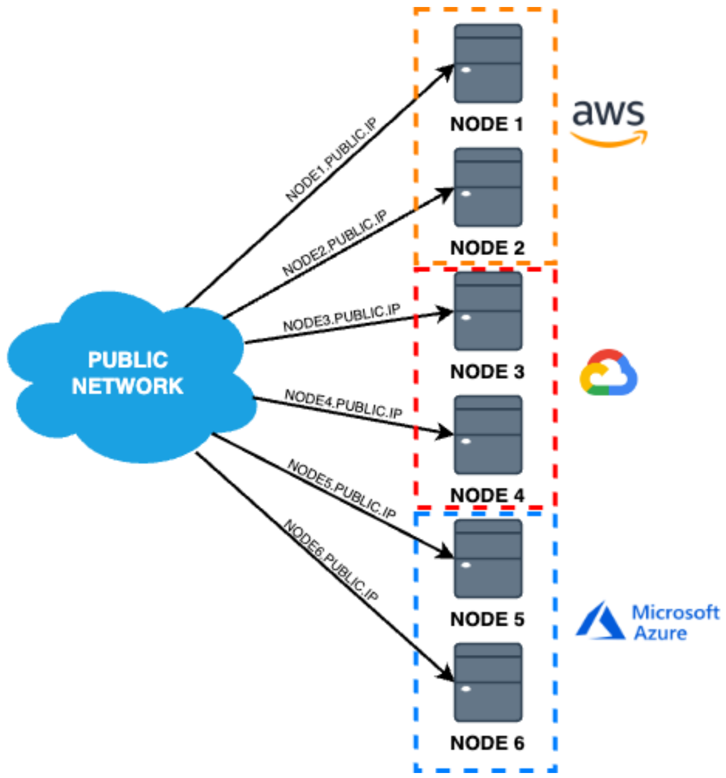


Figure 12. VMs spread across the internet from each provider.

Once the VMs have been successfully created, are up and running, and are reachable through ssh, the Kubernetes VMGroup will proceed with their initial configuration. This configuration process is accomplished by establishing SSH connections to the VMs using their respective public IP addresses. And once connected the configuration basically consists of the assignation of an IP depending on the type of node:

— **Load Balancers**: IP addresses range from 10.0.0.1 to 10.0.0.9.
— **Kubernetes Masters**: IP addresses range from 10.0.0.10 to 10.0.0.19.
— **Kubernetes Workers**: IP addresses range from 10.0.0.20 to 10.0.0.255.

The following figure illustrates this initial configuration.

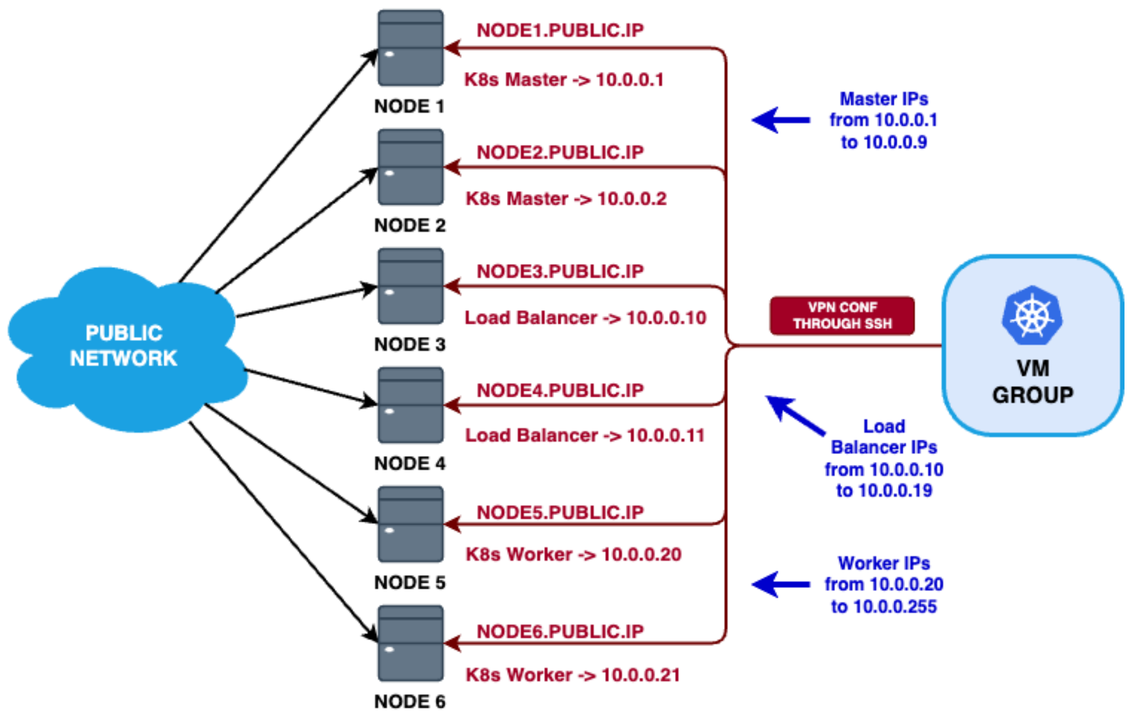


Figure 13. K8s VmGroup initial VPN configuration of the nodes.

6.3.3. VPN propagation

After completing the initial configuration, there is one crucial step remaining to ensure the VPN is fully operational. Although the VPN has been correctly configured on each node, these nodes lack awareness of one another's existence, resulting in a lack of connectivity. To establish comprehensive connectivity among all nodes, each node must disseminate its information to the others.

As illustrated in Figure 11, in order to establish a successful connection with a peer within the VPN, each node must share the following essential information with the other nodes: **VPN IP Address**, **Public IP Address**, and **Public Key**.

The propagation method involves a star communication network among all nodes, where each node connects with every other node to exchange information. The process is visually ilustrated in the following figure.

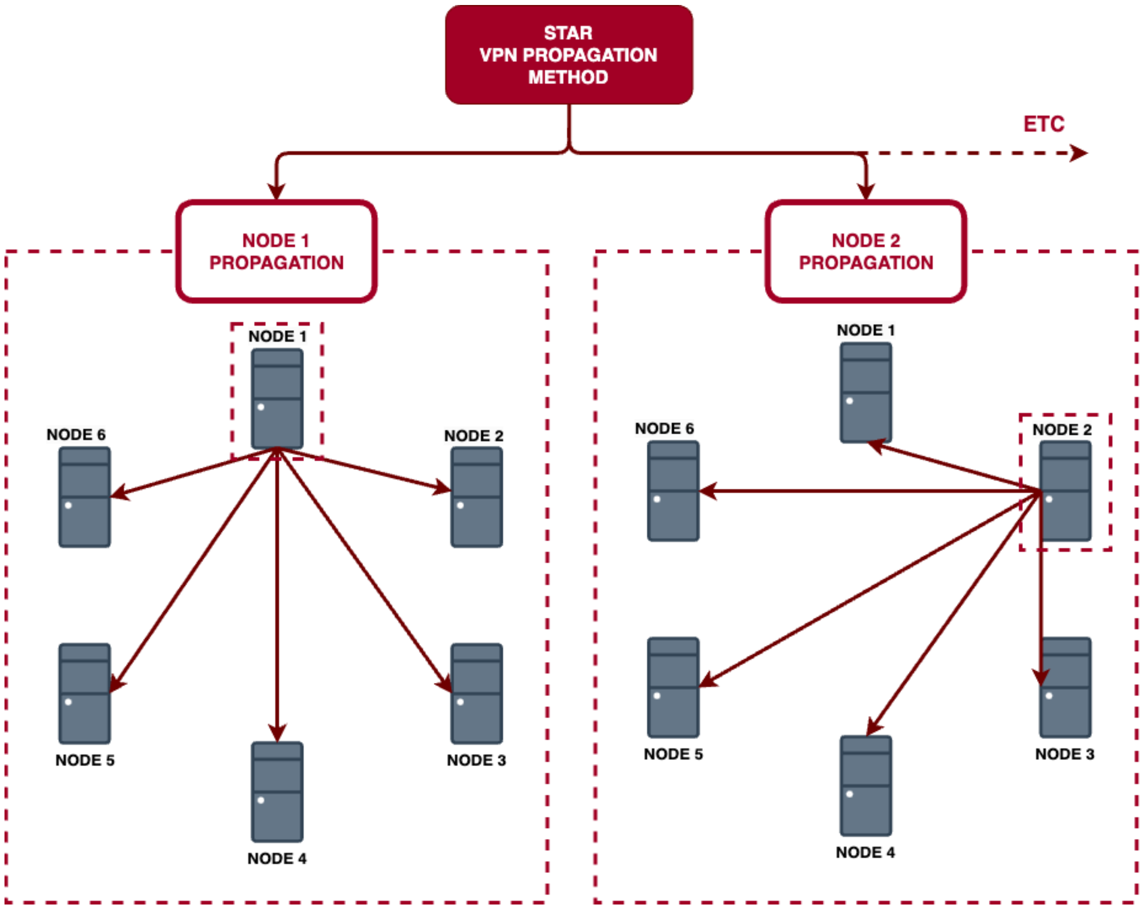


Figure 14. VPN star propagation method.

The method progresses from the initial node to the final one, disseminating information along the way. However, the program conducts parallel executions for each node's propagation, ensuring that configuration time does not experience exponential growth as the cluster's node count increases.

6.3.4. VPN final configuration

After completing the initial configuration and the VPN propagation of the information, the nodes transform their networking by adjusting their default routing settings within the VPN, effectively cutting the access to the broad internet. This strategy limits public network access to exclusively allow tunneling traffic among VPN members, while only a select few nodes retain the privilege of maintaining direct internet connectivity. Within the project's architectural framework, the design places the responsibility for internet access upon the load balancers, as illustrated in the following figure.

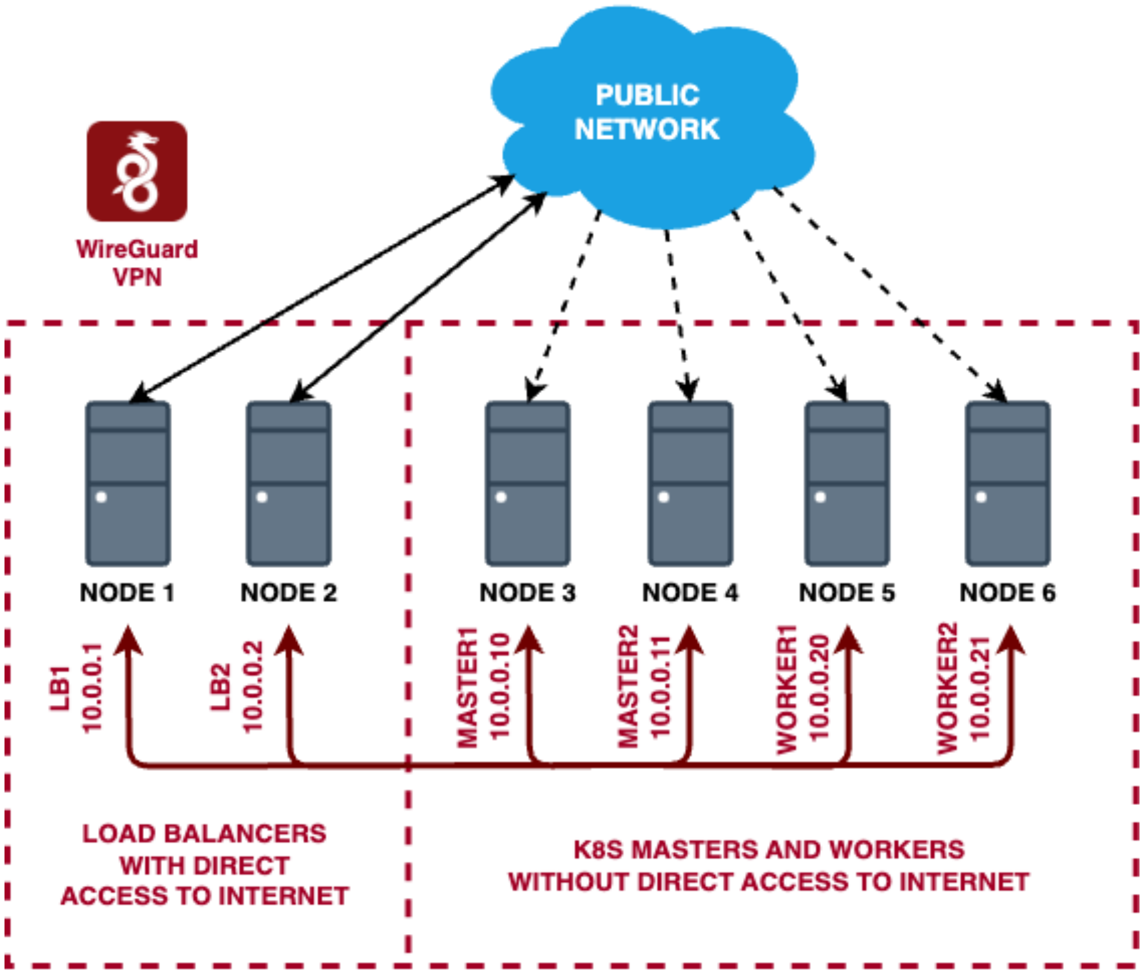


Figure 15. VPN Final Design.

6.3.5.VPN control network

Given the restricted connectivity between nodes within the VPN, direct connections from K8s to these nodes are not possible when changes are required. Therefore, a new method for accessing these nodes must be implemented. After the VPN configuration is complete, the control network involves a double SSH tunneling process through one of the load balancers, which serves as the entry point to the VPN network. This process is better illustrated through the following figure.

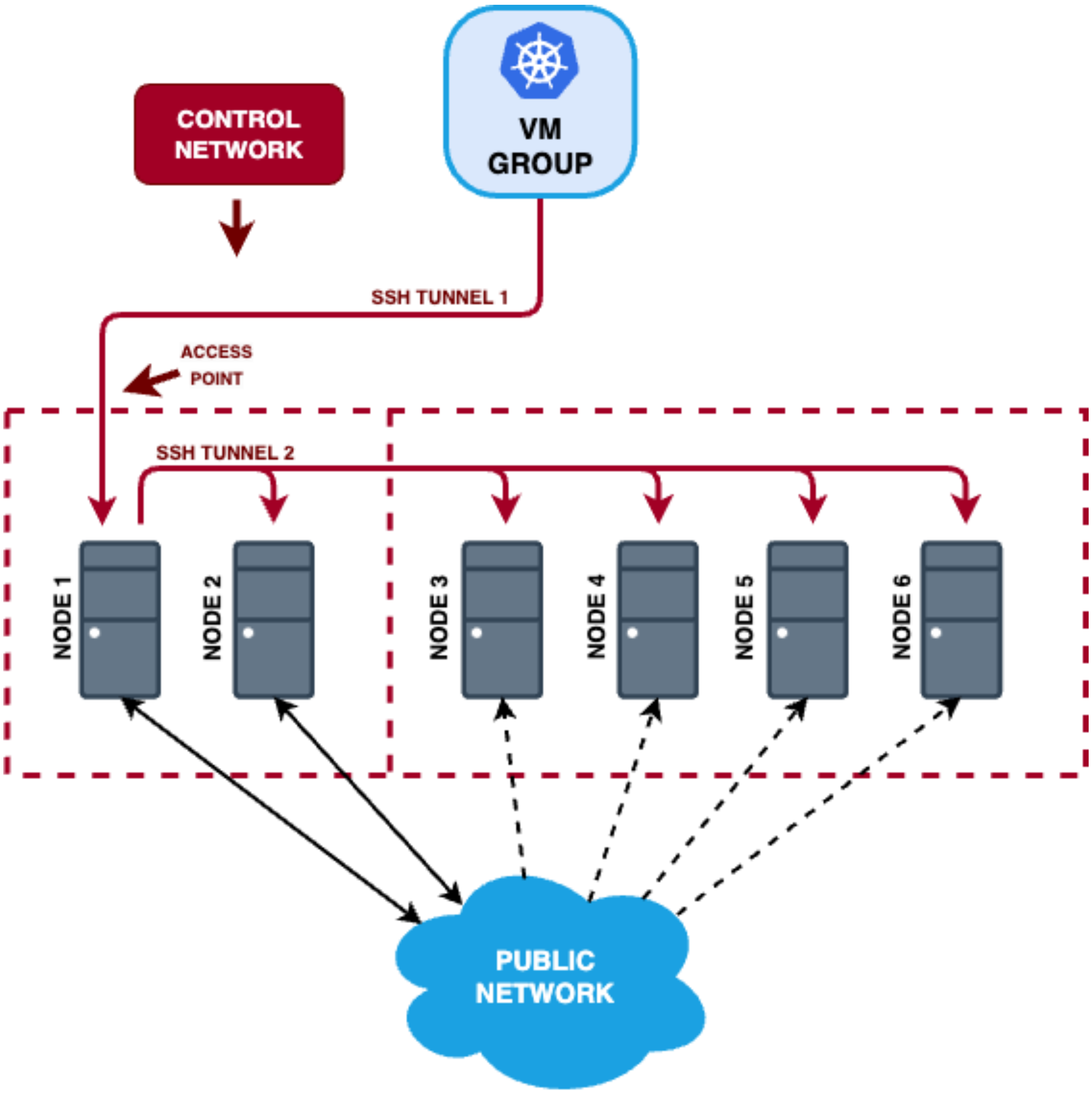


Figure 16. Control Network.

6.4. Load Balancers' Management

In the architectural context, it is important to address the reliance on multiple load balancers to mitigate the risks associated with a single point of failure, as it was already explained in the section of Cluster Architecture of this paper. The essence of load balancer management involves the complete replacement of one load balancer with another, transitioning all infrastructure to utilize the new one. However, this process is complex, primarily because the load balancer serves as both the central hub of the architecture and the point of entry from the public network. Achieving this transition is not straightforward, and while there exist protocols to solve similar challenges, none were found that fully meet the specific requirements of this project. Consequently, in this project, it was chosen to manually program the solution within the Kubernetes VMGroup resource.

The initial scenario is the one illustrated in the following figure, where there are several Load Balancers but only one is active and the rest remain inactive and unused.

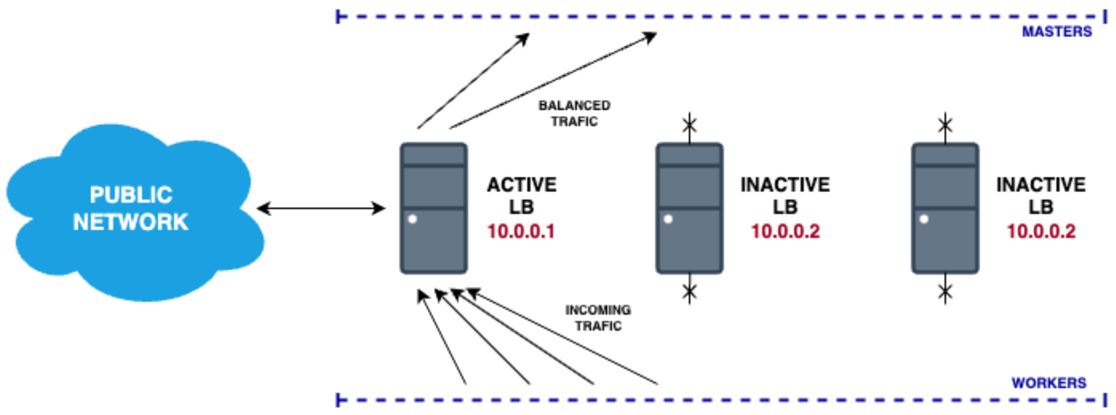


Figure 17. LBs Initial scenario.

All the traffic directed from the workers to the masters goes through the active LB and this LB is also used as the default route to reach the internet because LBs are the only ones with direct access as explained in the section of the VPN management of this paper.

If this LB fails it creates two challenges one within the public network and another within the private VPN network, and to fix them Floating IP and Dynamic DNS are used respectively.

6.4.1.Floating IP

To fix the problem within the VPN network it was chosen to use a Floating IP that will move depending on the active LB. As displayed in the following figure instead of using the initially assigned IP to balance the traffic the IP **10.0.0.100** is used.

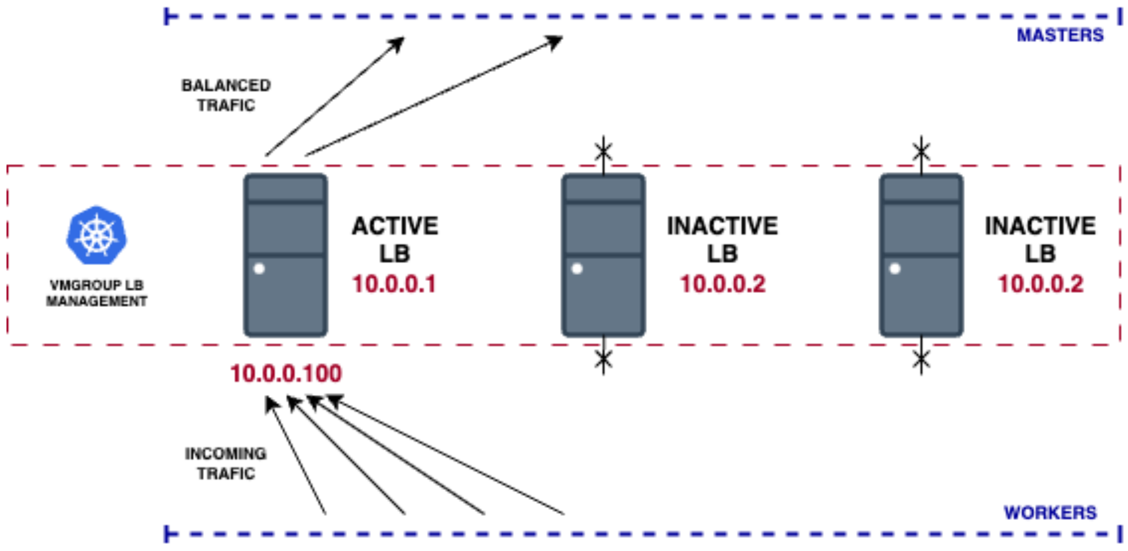


Figure 18. LBs with Floating IP.

Now if the active LB fails the VmGroup instance will notice and it will move the floating IP to a working LB as displayed here.

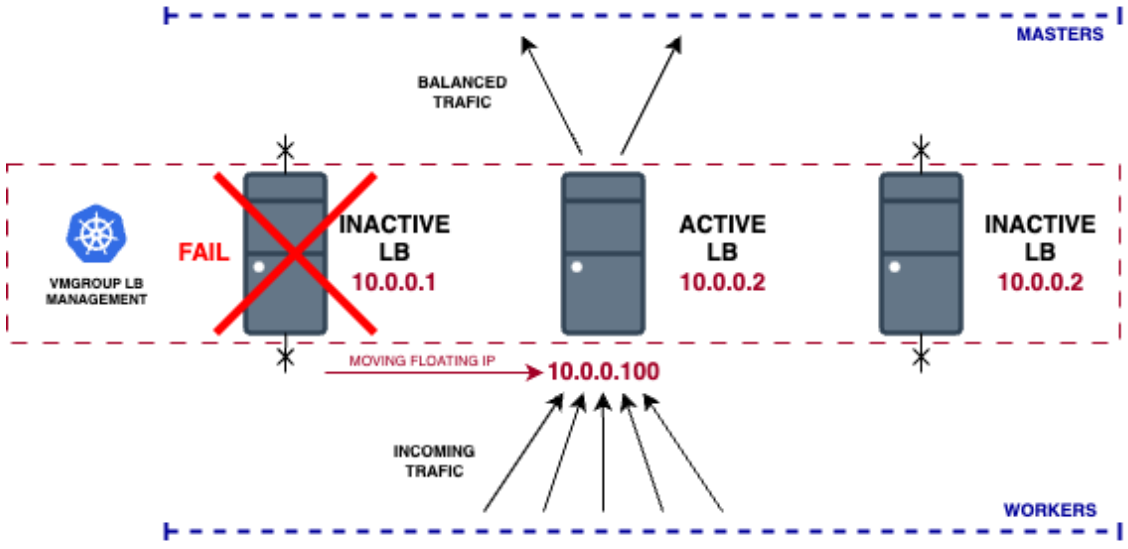


Figure 19. Migrating LBs' Floating IP.

The way to achieve this for the VmGroup is by these steps:

- First changing internally a variable that points the active LB, to point to another working LB.
- Accessing the new active LB by SSH and changing its IP to 10.0.0.100.

— Relaunching the VPN propagation information method that is explained and illustrated in Figure 14, to let all the nodes know that the IP 10.0.0.100 now is associated with a different public IP, corresponding to another LB.

6.4.2. Dynamic DNS

And to address the problem within the public side, regarding the access from the internet to the cluster, a Dynamic DNS (DDNS) solution is employed, with Duck DNS [13] being the specific service of choice.

In this scenario, the VmGroup does an additional step, involving the configuration of DDNS on the newly active LB. This step is crucial for ensuring that the DNS server being utilized is informed of the updated public IP address required to access the cluster.

— Access with SSH to the new active LB, and configure the DDNS

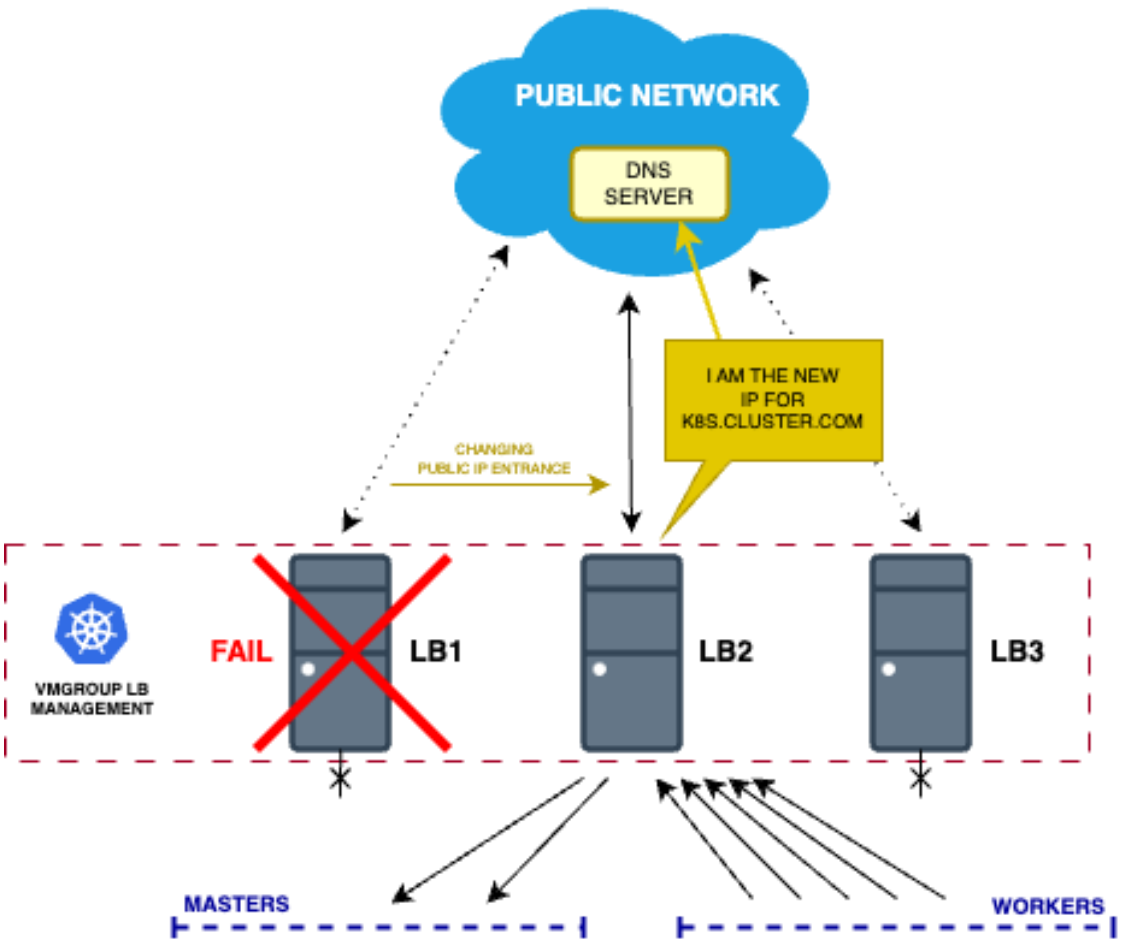


Figure 20. LBs with Dynamic DNS.

6.5. Solving The Chicken and Egg Problem

As explained within the context of this paper, achieving unified control of an entire application through the Kubernetes API necessitates the resolution of the classic "chicken and egg" scenario. This project aims to establish a Kubernetes instance capable of autonomously managing its underlying infrastructure, encompassing the creation, destruction, and adaptation of resources to various situations. However, a fundamental challenge arises: how can this Kubernetes instance initiate the creation of its infrastructure when the infrastructure itself does not yet exist, and as a consequence neither is the Kubernetes instance.

The most optimal solution found to address this situation is the one illustrated in this figure.

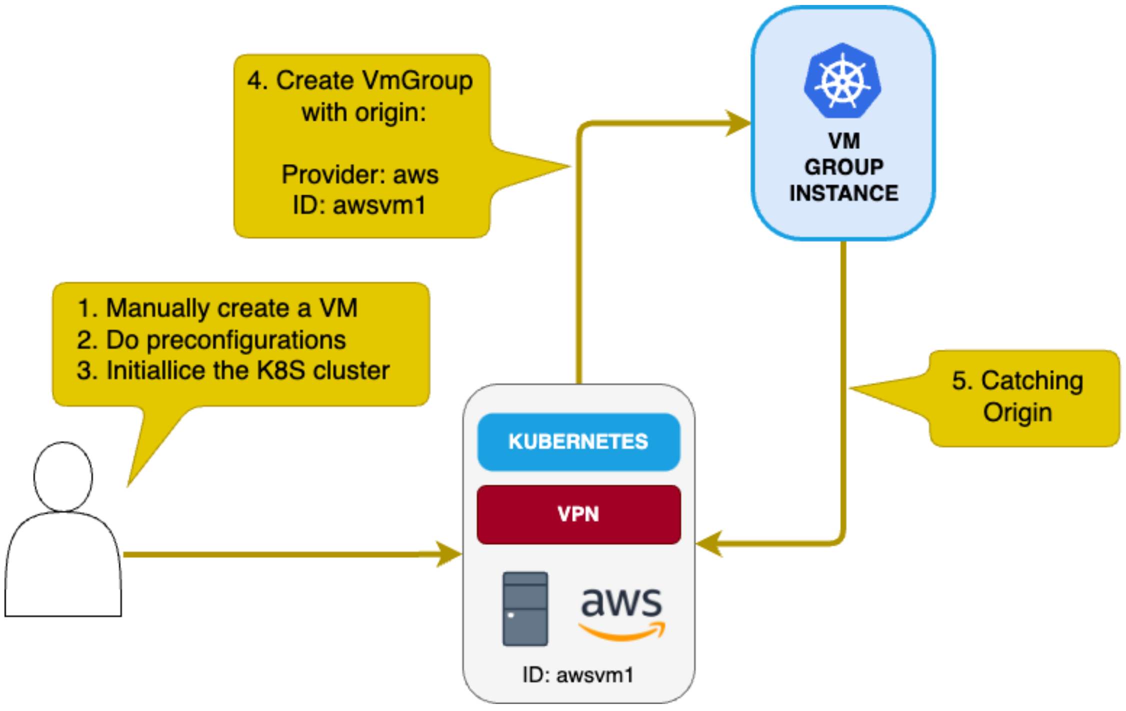


Figure 21. Chicken and Egg situation catch.

Explaining further each step:

- **Manual VM Creation:** A human operator manually creates a VM that serves as the genesis of the infrastructure. This VM can be provisioned on any of the available cloud providers.
- **Configuration Setup:** The operator configures this VM with all the necessary settings to integrate it into the forthcoming cluster. This includes tasks such as

configuring VPN and SSH settings, installing Kubernetes dependencies, and any other prerequisites.

- **Initialization of Kubernetes Cluster**: The Kubernetes cluster is initiated on the configured VM, commencing the process of establishing the infrastructure.
- **VmGroup Creation:** Create the CRD VmGroup described in the “VmGroup CRD” section of this paper. To do so it is necessary to specify a parameter called OriginNode that contains information about the VM created in step 1.
- **Catching Origin:** With the information of the original Vm passed as a parameter, the VmGroup can join and catch the already created VM instead of creating a new one, and this way being able to manage and access it.

Upon completing these steps, Kubernetes gains control over its initial node, effectively resolving the "chicken and egg" problem. Subsequently, the VmGroup continues with the creation and configuration of the remaining infrastructure components.

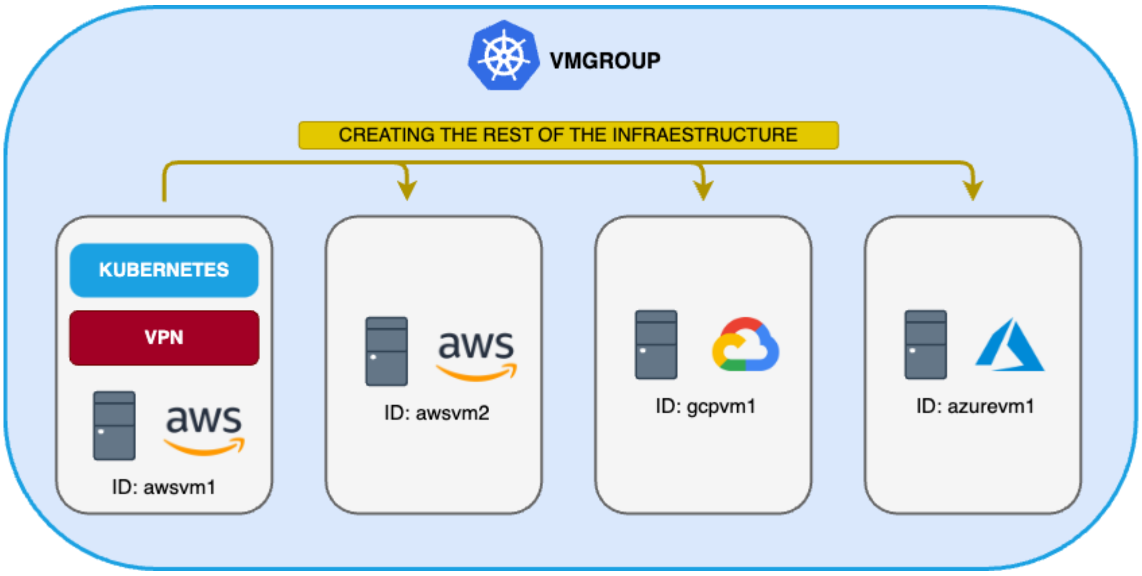


Figure 22. VmGroup creation of the cluster from the original node

6.6. Multi-cloud Solution

To establish a multi-cloud environment, the virtual machines (VMs) utilized in this project are treated independently, regardless of their hosting platform. It is assumed that each VM possesses the following prerequisites:

- A public IP address.
- Access to the internet.
- An open SSH port to facilitate incoming connections.
- Open ports required for Kubernetes.

Any VM meeting these requirements can be integrated into the cluster, regardless of its cloud provider.

In addition to these prerequisites, it is essential to program various methods specific to each cloud provider's SDK. To maintain independence from any particular provider, these methods are encapsulated within a common interface. In this case, the interface is programmed in Go and includes the following methods that must be implemented for integration into the cluster:

— `**CreateVm**`: This method should encompass the process of creating a VM.
— `**DescribeVm**`: In this method, a mechanism must be programmed to describe the VM with the ID provided as a parameter. This method is crucial for determining the VM's state, whether it is running, in the process of creation, or experiencing issues.
— `**DeleteVm**`: This method should include the logic for deleting the VM with the ID passed as a parameter.
— `**ListVms**`: In this method, a procedure must be programmed to list all the VMs that have been created.
— `**Configurations**`: This method is responsible for executing any necessary pre-configuration processes to ensure that the VMs created align with the before mentioned prerequisites. Each cloud provider has its unique processes, including profiles, projects, networks, and more, which are configured within this method.

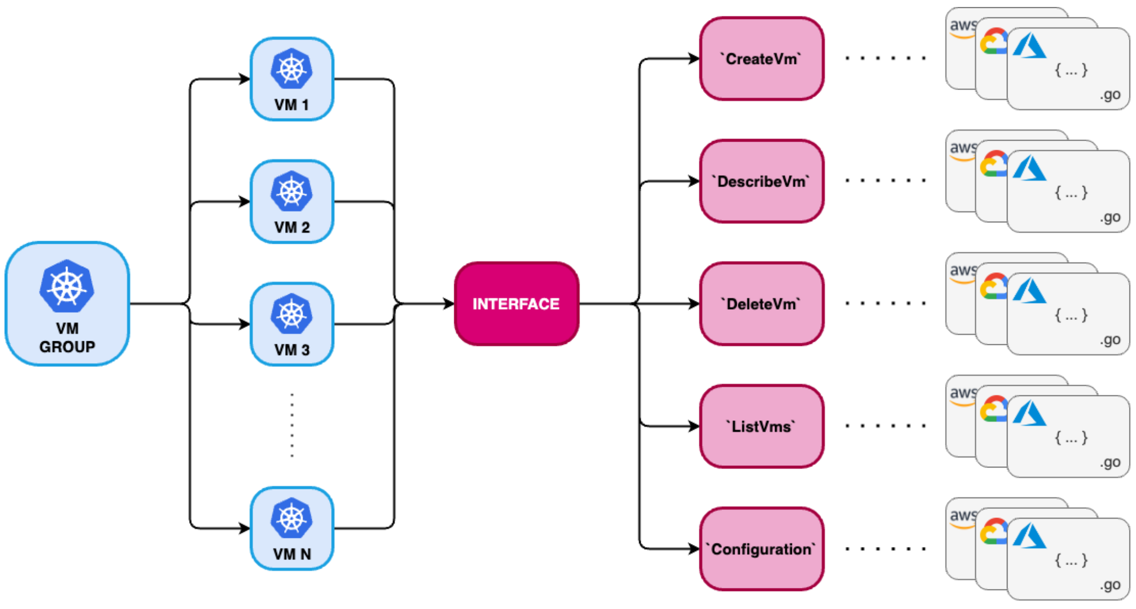


Figure 23. Multi-Cloud Providers Interface

7. Planning

The project planning has established the development timeline from February 1st to September 17th, totaling 156 working days. These days exclude weekends and official holidays in the Basque Country. The official holidays that fall on working days are as follows:

- April 6, 2023 - Jueves Santo (Holy Thursday)
- April 7, 2023 - Viernes Santo (Good Friday)
- May 1, 2023 - Día Internacional del Trabajador (International Workers' Day)
- July 25, 2023 - Santiago Apostol
- July 31, 2023 - San Ignacio
- August 15, 2023 - Asunción de la Virgen (Assumption of the Virgin)
- August 25, 2023 - Viernes de Semana Grande (Friday of Semana Grande)

In addition to these holidays, 5 working days of vacation have been scheduled from May 26th to 30th. Taking these holidays and vacation days into account, the project effectively contains 151 working days. Each of these days is considered as a half-day, meaning 4 hours of work per day, resulting in a total of 604 hours of work for the project.

7.1. Tasks' description

The project has been divided into 6 task packages:

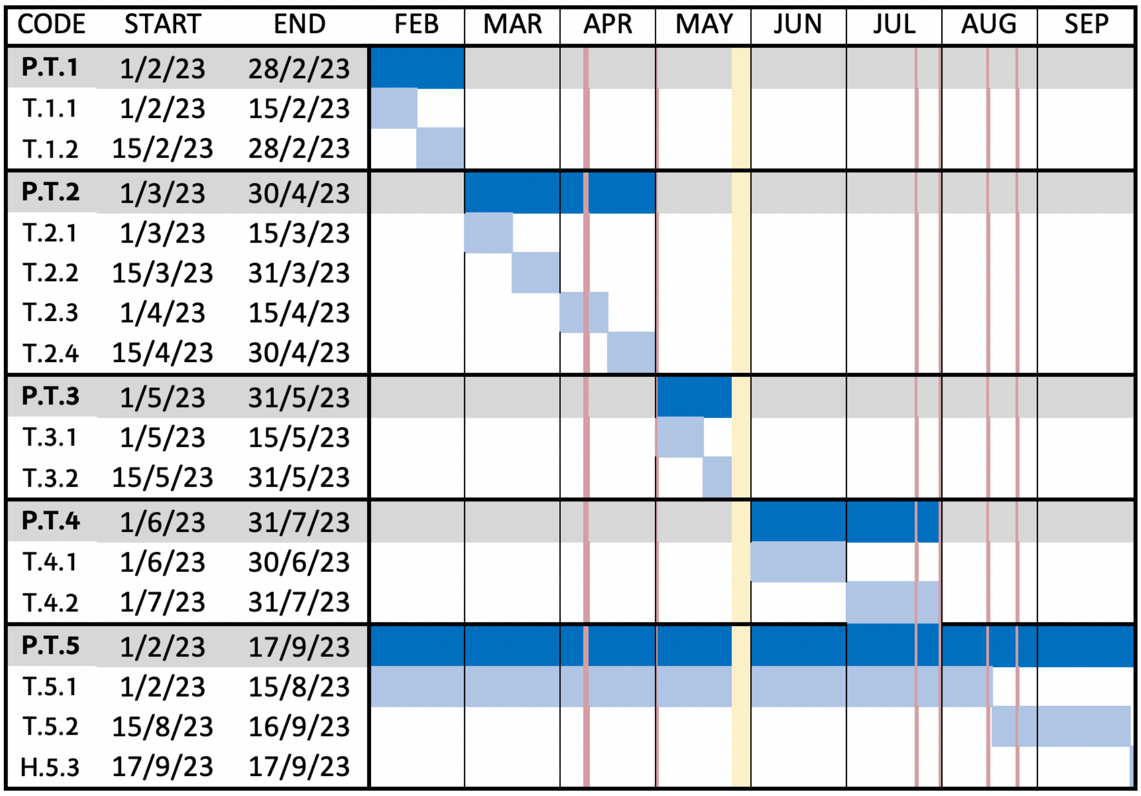
- P.T.1. Definition of the project
- P.T.2. Study and analysis of technologies
- P.T.3. Analysis of alternatives
- P.T.4. Proof of concept (PoC) design
- P.T.5. PoC Deployment
- P.T.6. Project management and documentation

These task packages, as well as the tasks contained in each one, are briefly described in Table 1. Likewise, the duration of each of them is shown in the form of a Gantt Chart in Figure 24.

Table 1. Description of the tasks that compose the project.

Code	Task name	Task description
P.T.1	**Definition of the project**	**Definition of the project bases**
T.1.1	Discussion and definition of the topic	Definición de las bases del proyecto
T.1.2	Definition of scope and objectives	Definition of the basis of the project
P.T.2	**Study and analysis of alternative technologies**	**Training, testing, and choosing the best technologies necessary for the development of the project**
T.2.1	Orchestration technologies	Testing a choosing the container orchestration technology.
T.2.2	IaC tools	Testing a choosing the best way to control services with project requirements
T.2.3	Kubernetes API extension frameworks	Chosing the best framework to to extend and create new K8s components
T.2.4	VPNs tecnologies	Trying and finding the easiest way to build and manage programatically a VPN
P.T.3	**Proof of concept (PoC) design**	**Basic and high-level design of the deployment architecture**
T.3.1	Requirements analysis	Analyze deployment requirements
T.3.2	Architecture design	Design the architecture, with its components, nodes and necessary applications
P.T.4	**PoC Deployment**	**PoC Implementation**
T.4.1	Development of the application	Development of all the components necesary for the deployment of the cluster
T.4.2	Deployment of the cluster	Execution of the application and deployment of all the components
P.T.5	**Project management and documentation**	**Preparation of project documentation**
T.5.1	Project documentation	Generation of documentation for the entire project
T.5.2	Final review of documentation	Review of final documentation
H.5.3	Final delivery	Final delivery milestone

7.2. Gantt Chart



CODE	START	END
P.T.1	1/2/23	28/2/23
T.1.1	1/2/23	15/2/23
T.1.2	15/2/23	28/2/23
P.T.2	1/3/23	30/4/23
T.2.1	1/3/23	15/3/23
T.2.2	15/3/23	31/3/23
T.2.3	1/4/23	15/4/23
T.2.4	15/4/23	30/4/23
P.T.3	1/5/23	31/5/23
T.3.1	1/5/23	15/5/23
T.3.2	15/5/23	31/5/23
P.T.4	1/6/23	31/7/23
T.4.1	1/6/23	30/6/23
T.4.2	1/7/23	31/7/23
P.T.5	1/2/23	17/9/23
T.5.1	1/2/23	15/8/23
T.5.2	15/8/23	16/9/23
H.5.3	17/9/23	17/9/23

Figure 24. Gantt Chart

8. Budget

This section provides an overview of the project's associated costs, encompassing both direct and indirect expenses, with an additional allocation of 10% reserved for unforeseen events.

The breakdown of direct costs typically comprises four components: internal hours, amortization, expenses, and subcontracting. However, it's worth noting that no funds have been earmarked for subcontracting in this particular project. The remaining three components are elaborated upon in the following tables: Table 2, Table 3, and Table 4.

Table 5 offers a comprehensive breakdown of the project's total cost, totaling 24.320,84€. Notably, the majority of the budget allocation (80%) is directed towards internal hours, primarily because the project's demands do not entail substantial material resources.

Table 2. Internal hours in the project budget.

INTERNAL HOURS			
Concept	**Unit cost (€/h)**	**Units Nº (h)**	**Total**
Telecommunication engineer	30	604	18.120,00 €
Project Manager	50	30	1.500,00 €
		SUBTOTAL	19.620,00 €

Table 3. Amortizations in the project budget.

AMORTIZATIONS				
Concept	**Acquisition cost**	**Useful life (years)**	**Use (months)**	**TOTAL**
Laptop Lenovo ThinkPad	1.199,00 €	5	8	159,87 €
Licence Microsoft 365	69,00 €	1	8	46,00 €
			SUBTOTAL	205,87 €

Table 4. Expenses in the project budget.

Expenses			
Concept	**Unit cost**	**Nº units**	**Total**
Electrical expense	0,117 €/kWh	2000 kWh	234,00 €
Office supplies	-	-	40,00 €
		SUBTOTAL	274,00 €

Table 5. Total cost of the project.

DIRECT COSTS	Internal Hours	19.620,00 €	20.099,87 €
	Amortizations	205,87 €	
	Expenses	274,00 €	
INDIRECT COSTS	10% of direct expenses		2.009,98 €
		SUBTOTAL 1	22.109,86 €
UNFORESEEN	10% of subtotal 1		2.210,98 €
		TOTAL (subtotal 1 + unforeseen)	24.320,84 €

9. Conclusions

In conclusion, this project successfully addressed the complex task of enabling a Kubernetes cluster to autonomously manage its own cluster infrastructure across multiple cloud providers. Leveraging the power of Kubernetes, this project developed CRDs that provided the necessary abstractions and control mechanisms. This innovative approach not only achieved the primary objective but also opened up new possibilities for enhanced automation, scalability, and resilience in multi-cloud environments. The work showcases the potential of Kubernetes as a versatile orchestration platform and underscores the significance of custom CRDs in extending its capabilities. As cloud-native technologies continue to evolve, the seamless integration of Kubernetes with diverse infrastructure resources becomes increasingly crucial, and this project represents a significant step in that direction.

Originally, the plan was to develop this project with Crossplane, a technology that aims to become a universal control plane with Kubernetes. However, as the project's customization needs grew, a more flexible framework was chosen, Kubebuilder. Nevertheless, the core idea inspired by Crossplane persisted, and the project retained its ability to oversee and manage all infrastructure within the Kubernetes ecosystem.

Speaking of infrastructure, the primary goal was to ensure self-healing and resilience in such a way that no single point of failure could break the entire infrastructure. Remarkably, this project accomplishes this by eliminating single points of failure in each node and making all the components self-healable and reset, including the LBs who are responsible for routing most of them within the cluster and gateways to the Internet.

To achieve this, the project establishes a VPN and implements a dynamic routing protocol that unifies all nodes as if they belonged to the same cluster, even though each node could be on different continents. Whenever changes occur in the infrastructure, this protocol ensures that all nodes promptly update their information, so everyone successfully communicates with each other. Additionally, a combination of a manually programmed Floating IP and a DDNS adds a layer of flexibility to the architecture, making it resilient to changes and adaptable to evolving needs.

In summary, this project and multi-cloud infrastructure management showcased the platform's potential. It stands as a testament to the innovative spirit driving the evolution of cloud-native technologies and their integration with Kubernetes, marking a significant leap towards a more automated, scalable, and resilient future.

This project shows how Kubernetes and multi-cloud management are coming together, demonstrating what the platform can do. It's proof of how technology is advancing to make cloud-based systems more automated, flexible, and strong for the future. And it also exemplifies the convergence and ambition of Kubernetes and its adaptability to overtake new challenges.

10. Bibliography

[1] Kubernetes. [Online] Available here: https://kubernetes.io

[2] Statista (2023). *Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 4th quarter 2022.* [Online] Available here: https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/#:~:text=Vendor%20market%20share%20in%20cloud%20infrastructure%20services%20market%20worldwide%202017-2022&text=In%20the%20fourth%20quarter%20of,percent

[3] CNCF (2022). *Annual Report*. [Online] Available here: https://www.cncf.io/reports/cncf-annual-report-2022/

[4] Crossplane. [Online] Available here: https://www.crossplane.io

[5] Yevgeniy Brikman.*Terraform: Up & Running.*

[6] Marek Moravcik, Martin Kontsek (2020). *Overview of Docker container orchestration tools.* IEEE

[7] CNCF (2023). *Project Journey Report*. [Online] Available here: https://www.cncf.io/reports/kubernetes-project-journey-report/

[8] CNCF. [Online] Available here: https://www.cncf.io

[9] Kubernetes. *The Cluster API Book* [Online] Available here: https://cluster-api.sigs.k8s.io

[10] Steve Francis (2023). *Is Cluster API Really the Future of Kubernetes Deployment?. The Nex Stack.* [Online] Available here: https://thenewstack.io/is-cluster-api-really-the-future-of-kubernetes-deployment/

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

[12] Jason A. Donenfeld (2017). *WireGuard: Next Generation Kernel Network Tunnel*. [Online] Available here: https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_04A-3_Donenfeld_paper.pdf

[13] Duck DNS. [Online] Available here: https://www.duckdns.org

[14] Geekflare. Terraform Architecture and Components through diagram. [Online] Available here: https://www.devopsschool.com/blog/terraform-architecture-and-components-through-diagram/

[15] Research Gate (2019). Kubernetes Components. [Online] Available here: https://www.researchgate.net/figure/Kubernetes-Components-The-Kubernetes-setup-has-at-least-three-components-kublet-daemon_fig1_336889240

[16] Kubernetes. *The Kubebuilder Book.* [Online] Available here: https://book.kubebuilder.io

[17] Jason Dobies, Joshua Wood (2020). *Kubernetes Operators*. O'Reilly Media