POLITECNICO DI TORINO

MASTER DEGREE COURSE IN COMPUTER ENGINEERING

Master Degree Thesis

Software Delivery in Multi-Cloud Architecture

Supervisors
prof. Fulvio Risso

Candidates
Amir Boroufar
matricola: 258015

Internship Tutor
Pasquale Lepera

Academic Year 2019-2020

Dedicate to
my family

Abstract

Two of the most popular trends in technical school nowadays are microservices and multi-cloud, and if you've attended a technical school conference recently, you've seen some sessions dedicated to them. The microservices adoption will be increased by the day as developers look to use smaller, modular services to increase application functionality. Another parameter accelerating the move to microservices by enterprises is that several organizations have returned to benefit from componentized software systems in development and preparation. For example, within the cloud, componentization will bring organizations many benefits, such as resiliency and support for horizontal scaling. However, the risk of service accessibility failure within a single cloud has decreased the popularity of single cloud suppliers amongst the users of the cloud. As a result, a new trend is beginning to emerge, which could be a movement towards multi-clouds. One of the most important challenges of a multi-cloud approach is that completely different cloud solutions run in various software system platforms. Nowadays, organizations need to make applications that may simply move across a large vary of those environments while not making integration difficulties. The ultimate goal of this thesis is to tackle the mentioned challenges by design a multi-layer architecture to employ DevOps techniques for software delivery in multi-cloud environments. To achieve the desired result we used different technologies and solutions ranging from OpenVPN to Istio to handle available challenges such as software delivery across a distributed infrastructure on clouds and traffic management between microservices. Finally, we measured the functionality of the proposed architecture by deploying a real microservice-based application on AWS and Azure infrastructure through CI/CD pipelines and control traffic distribution between microservices by Istio.

Contents

Chapter 1

Introduction

1.1 Microservices and Multi-Cloud

In tech, today microservices and multi-cloud are two of the hottest trends, and if you've attended a tech conference recently, you've likely seen at least a couple of sessions dedicated to them. The use of microservices increases by the day as developers look to use smaller, modular services that work in tandem to enable larger, application-wide functionality.

Microservices architecture 1.1 provide us with an alternative to developing a traditional, 'monolithic' application all in one go.

The main idea behind the microservices is breaking applications into smaller pieces make it easier for us to build and maintain them.

Compared to monoliths, microservices are easier to understand, test, and maintain over the life cycle the application. By helping us achieve greater agility in development, microservices significantly reduce the time needed to get working improvements to production [24].

Another factor accelerating the move to microservices by enterprises' reason is that many organizations have come to recognize the benefits of componentized software in development and deployment. For example, in the cloud, componentization can bring organizations many advantages, including resiliency and support for horizontal scaling. With microservices, these benefits can be magnified considerably [26].

When it comes to the cloud, we've all heard of public cloud, private cloud, and hybrid cloud, but a relatively new term for is multi-cloud: the

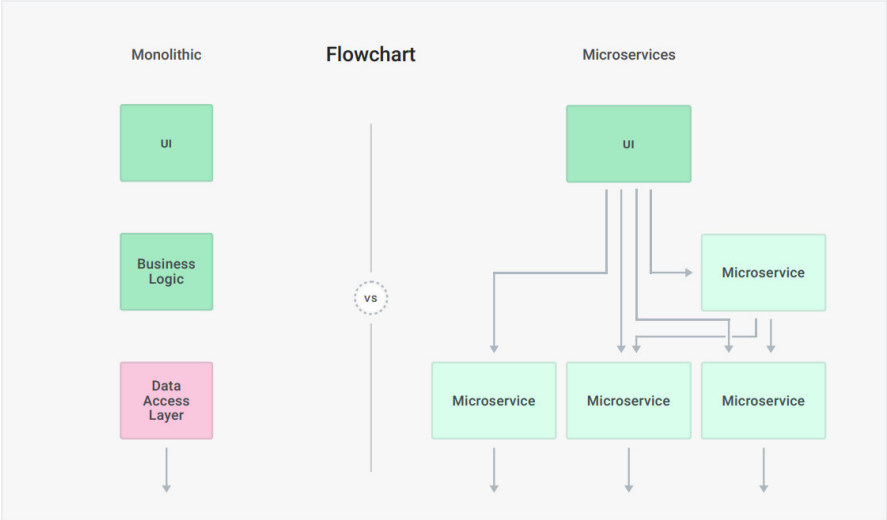

Figure 1.1: A microservices architecture.

use of more than a single public cloud. The multi-cloud usage pattern came about as a result of organizations wanting to avoid dependence on a single public cloud provider. With multi-cloud, organizations can choose specific services from each public cloud to get the best of available options.

Microservices can make it easier to deploy and use a multi-cloud environment but ensuring this needs a lot more technologies to set up and support this concept; The structures required to implement the multi-cloud infrastructure fit to a microservice architecture is discussed is this thesis.

1.2 The Rise of Multi-Cloud

The risk of service availability failure and the possibility of malicious insiders in the single cloud are predicted to decrease the popularity of single cloud providers amongst the users of the cloud. As a result, a new trend is starting to emerge, which is a movement towards multi-clouds [3].

Many say that multi-cloud is the future of information technology. While that is true, multi-cloud is also IT's present reality. In fact, over 80% of the organizations already have a multi-cloud strategy [2].

Following is a multi-cloud diagram 1.2 depicting an overview of a multi-cloud environment:

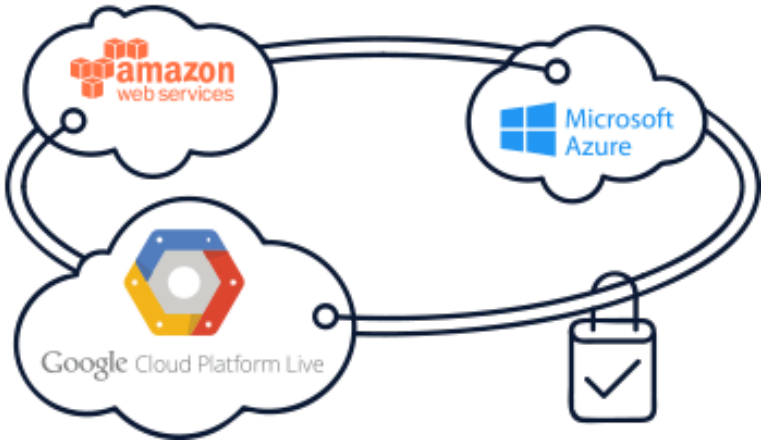

Figure 1.2: An example of a typical secure multi-cloud environment.

1.3 Multi-Cloud Enablers

1.3.1 Containers

Among the biggest challenges of a multi-cloud approach is that different cloud platforms run in various software environments. At the same time, organizations want to build applications that can easily move across a wide range of these environments without creating integration difficulties. Since they isolate software from the underlying environment, containers can be ideal here. This enables developers to build applications that can be deployed virtually anywhere.

There are many benefits to using containers. First, DevOps teams benefit greatly due to the more efficient approach to software development.

Containers bring agility for developers by reducing wasted resources and improving teams to build and share code more rapidly in the context of microservices.

On top of this, containerization improves scalability through a more lightweight and resource-efficient approach. This improved scalability, coupled with the improvements in development efficiency and velocity, results in greater time and cost efficiencies [1].

As we adopt a microservices architecture, it's also important to figure out how to monitor across the different services. This is where Kubernetes and its growing ecosystem of tools come into the picture.

As a result, the decision to use Kubernetes is not solely based on which container orchestration tool to use to further an organization's microservices strategy. What organizations are considering is also what the complementary ecosystem of tools looks like. The Kubernetes ecosystem offers all the building blocks for everything we need to leverage containers to build out a rock-solid microservices architecture [19].

1.3.2 Service Mesh

A service mesh is one of the hottest topics in technology right now good reason. Service mesh represents the next innovative leap in transitioning from centralized architectures to decentralized architectures.

These tools have made it easy to decouple services and have helped us to stop thinking in terms of monoliths. With them, we can separate out the execution of our services and keep their isolation consistent. In essence, Docker and Kubernetes provide the tooling needed to enable mainstream adoption. While some companies like Netflix and Amazon transitioned without these tools, their process of decoupling monoliths was more challenging.

Service mesh offers us the same subset of traditional use cases for north-south traffic deployed in a way that better handles the increased east-west traffic generated by a microservices architecture [4]. Our service mesh proxy can collect telemetry, handle routing and error handling, and limit access to our services in the same way that traditional gateways have handled north-south traffic for years. In this thesis, we're using the same technology and features to handle east-west traffic generated by a microservices architecture across a multi-cloud infrastructure.

Service mesh represents future innovative jump in transitioning from centralized architectures to decentralized architectures.

1.3.3 DevOps

Flexibility and shorter cycles are the main reasons to pursue a microservices architecture. However, without continuous integration and continuous delivery (CI/CD) procedure can keep your team from reaching the highest level of agility that's necessary to support microservices development and delivery. This thesis describes a variety of challenges and provides an architecture to mixing CI/CD with microservices application development.

1.4 Benefits of a Multi-Cloud Architecture

There are several factors that make a multi-cloud approach important for enterprises and the IT sector. The top reasons for enterprises and IT to enable a multi-cloud environment are as follows.

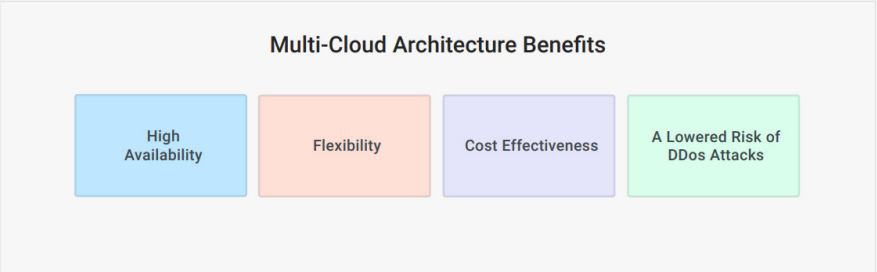

Figure 1.3: The benefits of a multi-cloud architecture [16].

- **High Availability** Redundancy for an organization's services against security and outages is supplied by a multi-cloud architecture. In a multi-cloud environment we have the lowest level of unavailability, even if one cloud is not accessible to run applications, other ones will be replaced..
- **Flexibility** In a multi-cloud environment, we are able to have the choice and flexibility of selecting the best of each cloud provider to meet our business goals. Typically, organizations can manage their data, infrastructure, and applications using several different clouds.

- **Cost Effectiveness** With a multi-cloud strategy, enterprises can control operational expenditures by taking advantage of the competitive market on price.
- **Lowered Risk** With the growth in cloud deployments, the likelihood of DDoS attacks is decreasing and multi-cloud architecture can provide a higher level of resiliency not possible with a single provider.

The main goal of this research is to integrate solutions and technologies in various layers for software delivery in multi-cloud environments through DevOps pipelines, but as a result of this approach many advantages will result in business layer in the long term that all will be very interesting for the market. We will discuss different aspects of each solution from infrastructure to the application layer in an incremental approach to finally reach a mature multi-cloud architecture for software delivery.

Chapter 2

Background

In order to give basic comprehension of the context of work of this thesis, this the chapter provides a brief description of technologies that were used in this thesis.

2.1 Multi-Cloud Networking

Companies are relying more and more on multiple public cloud providers. The reasons for multi-cloud environments range from different internal teams preferring to use specific services available from different providers, to reducing reliance or “lock-in” when using only one cloud provider. Setting up and maintaining the connectivity between cloud providers is difficult. Each provider offers different network services with varied constraints and performance. Also, the cloud providers are not highly incentivized to make it easy to connect different clouds together [6].

In this thesis, we employed OpenVPN solution to simplifies multi-cloud networking by providing an IaC solution for connectivity between the major cloud providers – including AWS, Azure, and Google cloud through IPSEC tunnel connections.

Figure 2.1 shows a graphic representation of secure multi-cloud peering betwwen different public cloud providers:

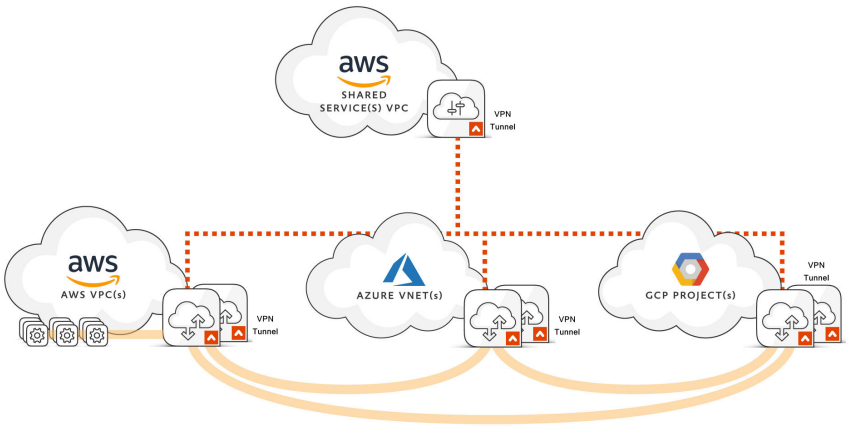

Figure 2.1: A secure multi-cloud peering diagram [6].

2.1.1 Site-to-Site VPN

A site-to-site VPN is to connect two or more isolated networks together through a VPN tunnel. Devices located in one network can reach devices in the other network, and vice versa. OpenVPN solution can create a bridge between two networks that works transparently for all devices in the two networks to be reachable across the networks.

2.2 Infrastructure as Code

Infrastructure as Code, or IaC for short, is an infrastructure management approach that allows you to save the entire infrastructure in a file, and deploy it in an automated fashion when needed. IaC also makes it possible to complete multiple repeatable deployments promptly. You get your code, deploy it, and that's that. If you wish to introduce changes for future deployments, it's also easy. Change whatever you need, save modifications, and you're done. In this thesis, we used Terafform to provision, deploy, and destroy infrastructure resources on the clouds and employed Ansible to configure our multi-cloud networking solution in a way to supports building our infrastructure from the ground up.

2.2.1 Terraform

Terraform is an open source tool to provision your infrastructure as code on various cloud providers (including AWS, Azure, Google Cloud) through a simple declarative programming language using a few commands. For instance, in this research we employed Terraform v0.12 to have an automatic solution for deploying AWS resources such as EC2, instead of manually clicking around a webpage, here is all we need to configure a server on AWS:

Listing 2.1: Example of EC2 Instance Creation using Terraform

```
resource "aws_EC2_instance" "example" {
ami = data.aws_ami.example.id
instance_type = "t3.micro"
}
```

2.2.2 Ansible

Ansible is the fastest growing open source IT automation tool that can be used to configure or manage systems, applications, and infrastructure on many cloud platforms. Ansible is not only limited to managing servers but it also manages whole cloud infrastructure. Ansible has a unit of codes named module to executes on the remote target node like AWS, GCP, VMWare, and Microsoft Azure, and collects return values.

In this thesis, we used Ansible v2.9.14 as a configuration management tool to have our OpenVPN solution as a code that prepares a full-mesh VPN topology across different cloud providers. Figure 2.2 shows a graphic representation of Ansible to manage AWS EC2 instances ready for OpenVPN servers:

2.3 Kubernetes

Kubernetes, or k8s for short, is an open-source container orchestrator. Originally developed by the engineers at Google, Kubernetes solves many problems involved with running a microservice architecture in production [18]. Kubernetes is responsible to automatically handle load-balancing, scaling, rolling updates, and other tasks that used to be

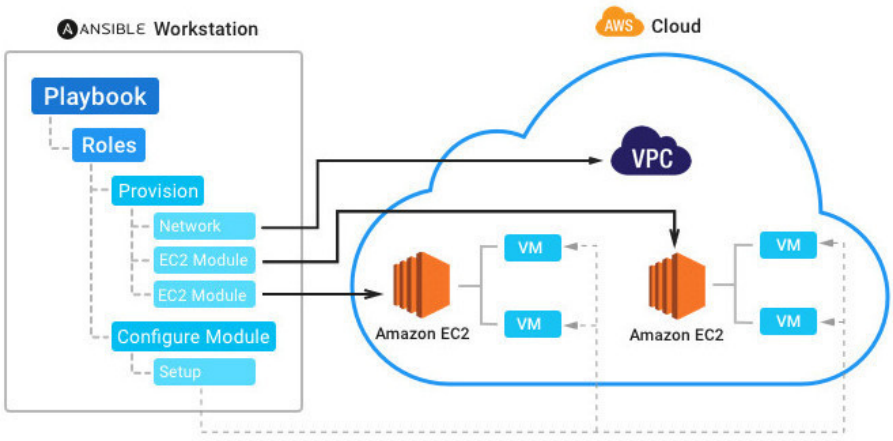

Figure 2.2: Use Ansible to manage AWS infrastructure.

done manually by DevOps engineers. Kubernetes cluster consists of one or more masters and multiple worker nodes that master nodes orchestrate the applications running on nodes, and monitor them constantly to ensure that they are in the desired state defined by programmer.

2.3.1 Kubernetes Components

Master Components

Kubernetes cluster will be monitored by master components to handle cluster events such as scheduling or restarting unhealthy pods. There are five components on the master node as follows:

- **Kube-apiserver** serves as the frontend for the Kubernetes control plane. The API server exposes an HTTP API that lets end-users, different parts of your cluster, and external components communicate with one another.
- **Etcd** is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.
- **Kube-scheduler** watches new workloads/pods and selects a node for them to run on.
- **Kube-controller-manager** central controller to reduce complexity by watching the Node controller, replication controller, services,

and service accounts.

- **Cloud-controller-manager** interacts with the remote cloud provider to check and manage resources.

Node Components

Responsible for cluster running environment, run on every worker node, and maintaining running pods.

- **Kubelet** an agent that runs on each node in the cluster to monitor the container health and report to the master, also handle the received commands from the kube-apiserver.
- **Kube-proxy** maintains the network rules to allow pod's communication from inside to outside of the cluster and vice versa.
- **Container runtime** is the software that is responsible for running containers.

Figure 2.3 shows a graphic representation of Kubernetes architecture and components.

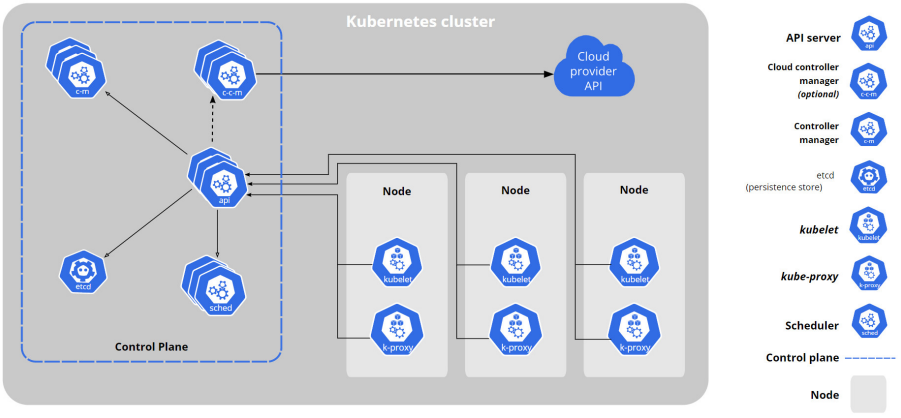

Figure 2.3: Kubernetes architecture and components [17].

2.3.2 Kubernetes Workloads

Kubernetes workloads are divided into two main components: pods and controllers.

- **Pods** The pod is the smallest and the simplest unit in Kubernetes architecture, it can be compared with what a container is for Docker, a single instance of an application. A Pod encapsulates one or more containers as well as storage resources, an IP address, and rules on how the container(s) should run, as can be seen from Figure 2.4.

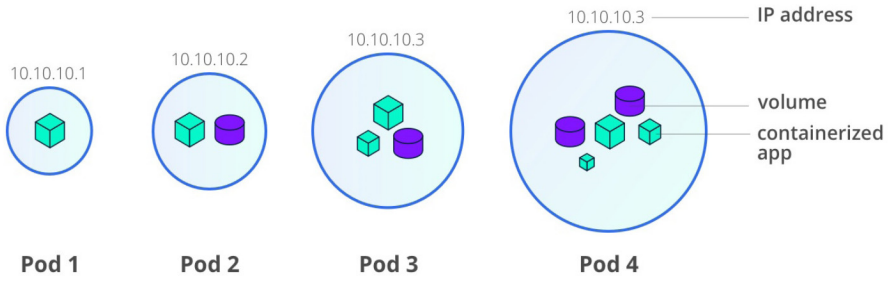

Figure 2.4: pods.

  Pods are the atomic unit on the Kubernetes platform to host only an application instance at the same time.

- **Controllers** As mentioned earlier, Pods are usually deployed indirectly via Controllers. There are different types of controllers as following:

  - **ReplicaSet** A ReplicaSet's purpose is to deploy the specified replicas of the Pods running to guarantee the existence of a specified number of Pods.
  - **Deployments** Deployments allow for rolling updates and easy rollbacks on top of ReplicaSets, you can define the desired state in the Deployment model, including scaling, rolling updates in canary or blue/ green fashion and Deployment will take care of it for you [18].

2.4 Rancher

Rancher is a container management platform built for organizations that deploy containers in production. Rancher makes it easy to run Kubernetes everywhere, meet IT requirements, and empower DevOps teams. Rancher supports centralized authentication, access control,

and monitoring for all Kubernetes clusters under its control. In this thesis, we employed Rancher as a complete container management platform for Kubernetes to successfully run Kubernetes on different public cloud providers [21].

In this research, we employed Rancher as a solution to deploy and maintain Kubernetes clusters distributed on different public cloud providers, here AWS and Azure.

The below diagram 2.5 shows how the cluster controllers, cluster agents, and node agents allow Rancher to control downstream clusters.

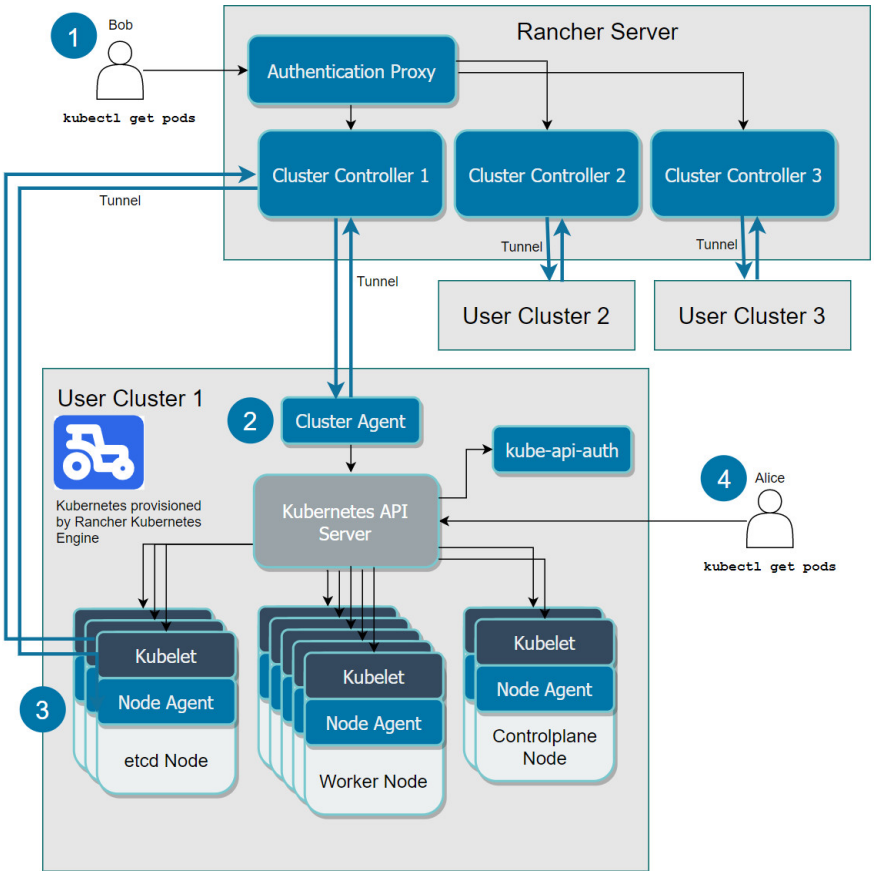

Figure 2.5: Rancher communication with Kubernetes clusters [21].

The following descriptions correspond to the numbers in the diagram 2.5:

1. **The Authentication Proxy** Forwards all Kubernetes API calls to downstream clusters.

2. **Cluster Controllers and Cluster Agents** Each downstream user cluster has a cluster agent, which opens a tunnel to the corresponding cluster controller within the Rancher server.

   (a) Watches for resource changes in the downstream cluster

   (b) Brings the current state of the downstream cluster to the desired state

   (c) Configures access control policies to clusters and projects

   (d) Provisions clusters by calling the required Docker machine drivers and Kubernetes engines, such as RKE and GKE

3. **Node Agents** If the cluster agent is not available, one of the node agents creates a tunnel to the cluster controller to communicate with Rancher.

4. **Authorized Cluster Endpoint** An authorized cluster endpoint allows users to connect to the Kubernetes API server of a downstream cluster without having to route their requests through the Rancher authentication proxy.

2.5 Helm chart

When deploying an application on Kubernetes, it is required to define and manage several Kubernetes resources such as pods, services, deployments, and replica sets in YAML format.

It becomes a difficult task to maintain several manifest files in the context of complex application deployment. If we could have a template-based approach able to separate the manifest files and configuration parameters, it will allow us to customize the deployments and have version control factors as well. This is where Helm plays a crucial role in automating the process of installing, configuring, and upgrading complex Kubernetes applications.

Helm is a package manager for Kubernetes. It allows developers to easily configure, package, and deploy applications on Kubernetes clusters. With Helm, configuration settings are isolated from manifest files. This allows to the template and customizes settings without changing

the entire manifest file [25] .

In this thesis, We benefit from Helm charts v2 for Kubernetes package management to deploy microservices and Istio configurations on destination clusters.

2.5.1 Helm Components and Terminology

Helm v2 has two elements, a client (helm) and a server (Tiller). The server element runs inside a Kubernetes cluster and manages the installation of charts. This diagram 2.6 shows how Helm components are related to each other:

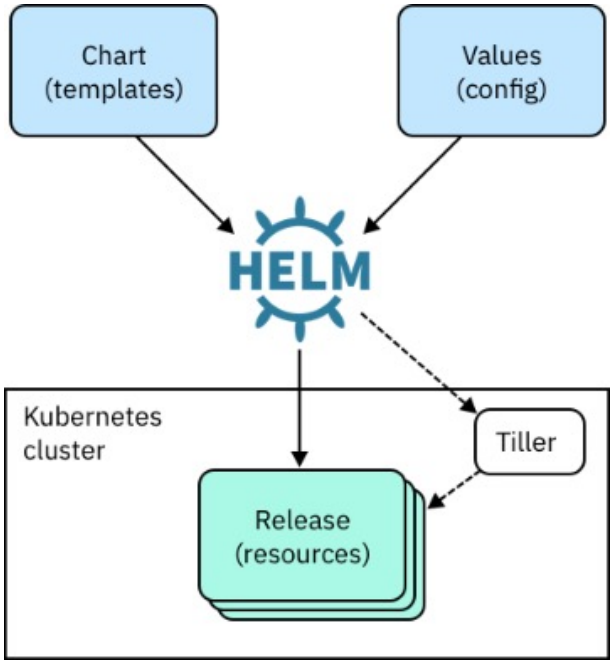

Figure 2.6: Helm v2 components [12].

Helm A command-line interface (CLI) that installs charts into Kubernetes, creating a release for each installation. To find new charts, you search Helm chart repositories.

- **Chart** A chart is an application package that describes Kubernetes resources by a collection of files necessary to run the application.

The chart includes a values file that describes how to configure the resources.

- **Repository** Storage for Helm charts.
- **Release** An instance of a chart that is running in a Kubernetes cluster. You can install the same chart multiple times to create many releases.
- **Tiller** The Helm server-side engine, which runs in a pod in a Kubernetes cluster. Tiller processes a chart to generate Kubernetes resource manifests, which are YAML-formatted files that describe a resource.

2.6 Service-Mesh

A service mesh is used to describe the network of microservices that shape applications and the communication between them. Service meshes architecture provides a policy-based, network service to manage the workloads between microservices through enforcing policies of the network.

As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements such as canary deployment. Istio provides behavioral insights and operational control over the service mesh as a whole, offering a complete solution to satisfy the diverse requirements of microservice applications. [13].

2.6.1 Istio

Istio is an open-source implementation of a service mesh, in the cloud-native ecosystem, it's second in the scope of objectives to Kubernetes.

Istio helps you add resiliency and observability to your services architecture in a transparent way to intercepts and handles network traffic on behalf of the application. So, an Istio-based service mesh can also be deployed across platforms like OpenShift, Mesos, as well as traditional deployment environments like VMs and bare-metal servers [7].

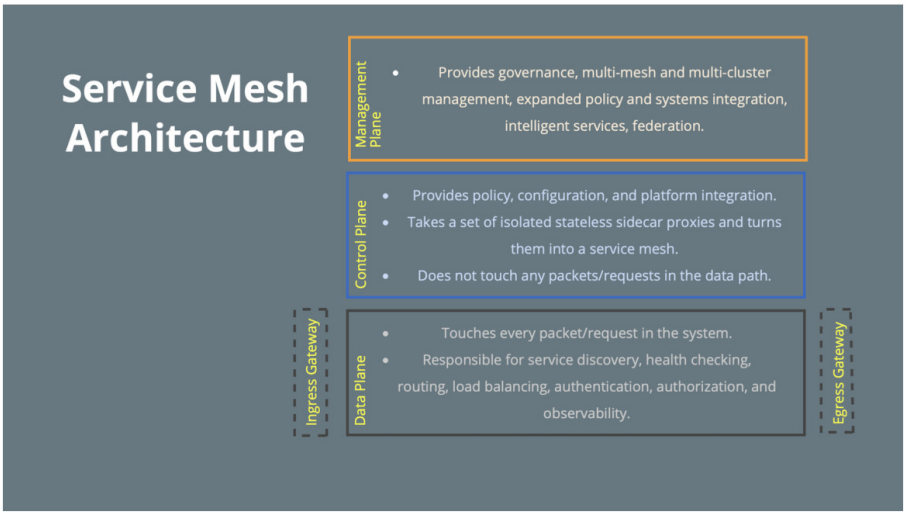

Figure 2.7: Service-mesh architecture [7].

In this research, we chose Istio v1.4.2 as the best available service mesh solution in the market to interconnect Kubernetes clusters across the Internet and also handle traffic flow between microservices on a multi-cloud architecture.

2.6.2 Istio at a Glance

Istio service mesh is logically divided into a control plane and a data plane. The data plane is composed of a set of intelligent proxies deployed as sidecars. These proxies control all network communication between microservices. They also collect and report telemetry on all mesh traffic. The control plane manages and configures the proxies to route traffic.

The following diagram 2.8 shows the different components that make up each plane:

Control Plane Components

In the Istio version 1.5 and above, the control plane is shipped as a single binary Istiod and comprises of three parts: Pilot, Citadel, and Galley.

- **Pilot** Is the central controller of the service mesh and is responsible

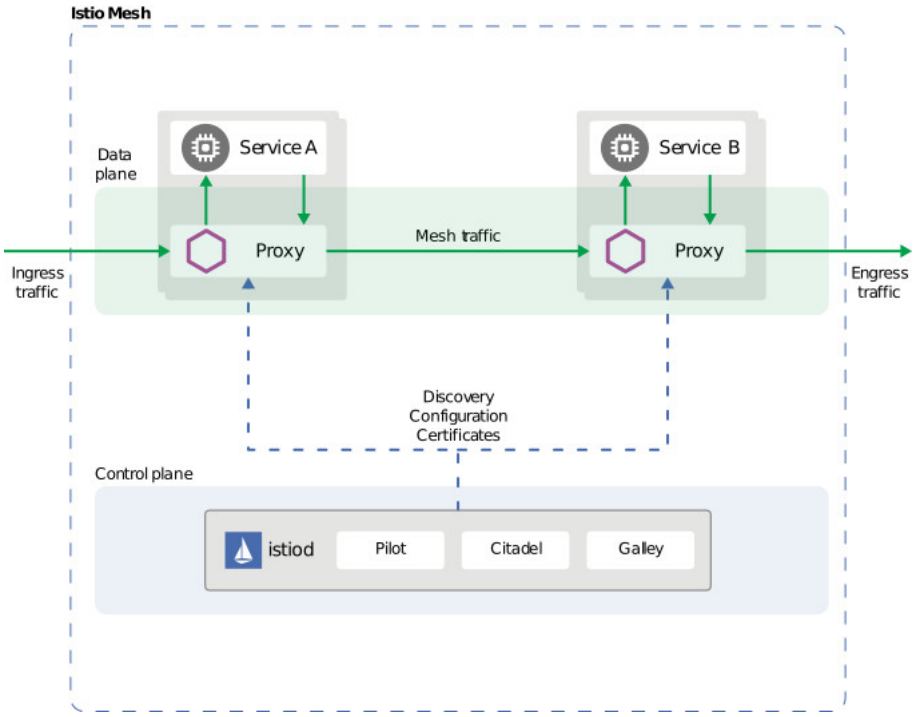

Figure 2.8: Istio architecture [13].

for communicating with the Envoy sidecars using the Envoy API. They parse the high-level rules defined in the Istio manifests and convert that to Envoy configuration.

- **Citadel** Identity and access management between your services is the central feature of Istio. It helps you allow secure communication between your Kubernetes pods. What that means is that while your developer has designed components with insecure TCP, the Envoy proxy would ensure communication between pods is encrypted.

- **Galley** Is responsible for providing configuration validation, ingestion, processing, and distribution for your service mesh. It's the interface for the underlying APIs with which the Istio control plane interacts.

Data Plane Components (Envoy proxy)

- **Traffic control** Helps in controlling how traffic moves through your service mesh, such as providing routing rules for HTTP, TCP, WebSocket, and gRPC traffic.
- **Security and authentication** Enforce identity and access management over the pods so that only the right pods can interact with another. They also implement mutual TLS and traffic encryption to prevent man-in-the-middle attacks. They provide rate limiting, which prevents runaway cost and denial of service attacks.
- **Network Resiliency** They help provide network resiliency features such as retries, failover, circuit breaking, and fault injection.

The following diagram 2.9 shows the relation between each component across the mesh:

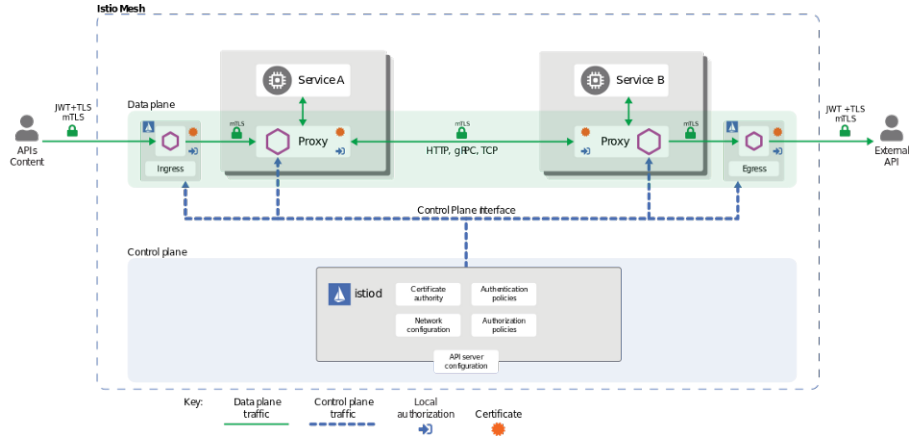

Figure 2.9: How components interact with each other [9].

2.7 DevOps

Organizations are primarily focussing on cloud transformation and in this journey trying to improve their operational efficiency and want to minimize their IT expenditure overall. DevOps is normally referred as a set of industry operational practices that mainly focus on collaboration and communication of both software developers and information

technology (IT) professionals in order to automate the process of software delivery and infrastructure changes [14]. It aims at providing a platform and establishing a culture in making the environment where building, testing, and releasing software can happen rapidly, frequently, and more reliably. DevOps brings improvement and standardization in both development and operational process brings these two together and making room for collaboration and communication between them.

2.7.1 Key focus area in DevOps

DevOps consists of key functional areas. We would like to introduce those key concepts in the following section:

- **Continuous Integration (CI)** It is the development practice that developers integrate the code into the shared repository continuously multiple times in a day. Each integration can then be verified by an automated build, allowing teams to detect problems early.

- **Continuous Deployment (CD)** It is a strategy closely related to Continuous Integration for software releases when any commit that passes the automated testing phase will automatically released to a production environment.

- **Orchestration** In any multiple environment management, lots of tasks are performed at different places and need to be coordinated, collaborated, communicated with different workflows and needs to be managed effectively. Hence a good orchestration the mechanism is essential to co-ordinate, manages, executed as a single task across the different environment by managing dependency, and does proper notification [14].

- **Configuration as a code** It is the discipline of thinking of configuration elements of a software system as you would think of code. It allows the entire configuration to store as a source code. It enables us to collaborate with operations on the application environments to ensure that they have correct configurations. It allows continuous deployment and prevents continuous drifts [14].

2.7.2 Devops for Multi-Cloud

Containers, microservices, and orchestration tools like Kubernetes play an important role in automation, organizations need to optimize cloud infrastructure. Kubernetes has helped companies turn the idea of multi-cloud into a reality.

Businesses want to choose cloud providers for their inherent value and use the services that best meet their needs. A multi-cloud future gives organizations the flexibility to deploy anywhere and run workloads across multiple clouds.

In this thesis we use Gitlab v13.2.4 as a complete DevOps platform, delivered as a single application, ensuring visibility across the entire SDLC 2.10.

Operations teams can work concurrently in a single application as the result of concurrent DevOps cycles and higher efficiency across all stages of the software development lifecycle. There's no need to integrate and synchronize tools. Everyone interacts to a single platform, instead of managing multiple disparate DevOps tools.

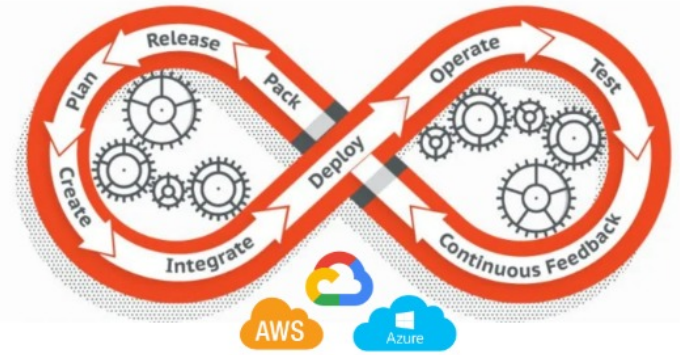

Figure 2.10: Systems development life cycle.

2.7.3 CI/CD pipeline

A CI/CD pipeline 2.11 is a series of steps that must be performed in order to deliver a new version of the software. Continuous integration/continuous delivery (CI/CD) pipelines are a practice focused on improving software delivery using either a DevOps or site reliability engineering (SRE) approach [22].

A CI/CD pipeline introduces monitoring and automation to improve the process of application development, particularly at the integration and testing phases, as well as during delivery and deployment. Although it is possible to manually execute each of the steps of a CI/CD pipeline, the true value of CI/CD pipelines is realized through automation.

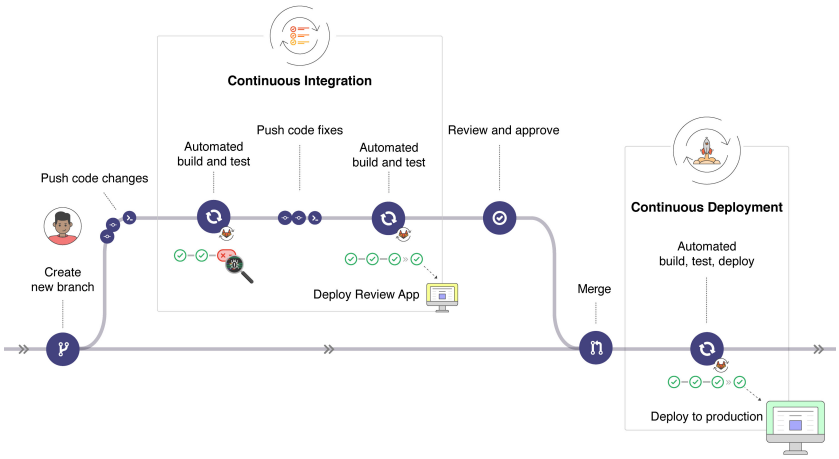

Figure 2.11: Sample CI/CD pipeline [11].

Chapter 3

Network Infrastructure Architecture

3.1 Introduction to OpenVPN

OpenVPN provides flexible VPN solutions to secure your data communications, whether it's for Internet privacy, remote access for employees, securing IoT, or for networking Cloud data centers. Our VPN Server software solution can be deployed on-premises using standard servers or virtual appliances, or on the cloud [20].

OpenVPN offers strong features such as SSL/TLS protocols which are not typical for other VPN solutions, it is an SSL-based VPN solution to secure the connections. It can be configured to use pre-shared keys as well as X.509 certificates. OpenVPN also uses a hashing algorithm for ensuring the integrity of the packets delivered.

OpenVPN is compatible with any operating system that supports a virtual network adapter. OpenVPN uses this virtual network adapter (a tun or tap device) as a communication channel between the user-level and the operating system.

OpenVPN has the notion of a control channel and a data channel, both of which are encrypted and secured differently. However, all traffic passes over a single UDP or TCP connection. The control channel is encrypted and secured using SSL/TLS, the data channel is encrypted using a custom encryption protocol [8].

3.2 Advantages and Disadvantages

OpenVPN is easy to deploy and configure even on restricted network topologies, including NAT'ted networks. Also, OpenVPN includes strong security features as IPSec-based solutions and support for different user authentication mechanisms. Lack of scalability and its dependence on the installation of client-side software can be considered as disadvantages of OpenVPN solution. Incompatibility between the tap interface driver and some version of Windows often caused deployment issues.

3.3 OpenVPN Packages

There are several OpenVPN packages available on the Internet:

- The open-source or community version of OpenVPN
- Commercial version offering by OpenVPN Inc

3.3.1 The Open Source Version

Open-source versions of OpenVPN are made available for multiple platforms, including both 32-bit and 64-bit Windows clients. The community version of OpenVPN can act both as a VPN server and as a VPN client. There is no separate client-only version.

In this thesis, we employed the open-source v2.4.8 to shape our architecture.

3.4 OpenVPN Components

3.4.1 The Tun/Tap Driver

One of the crucial elements of OpenVPN is the tun/tap driver. The concept comes from the Unix/Linux world and it is often embedded in the operating system. This virtual network adapter communicates by the OS as either a point-to-point adapter (tun-style) for IP-only traffic or as

a full virtual Ethernet adapter for all types of traffic (tap-style). OpenVPN as a backend application is responsible to process incoming and outgoing traffic.

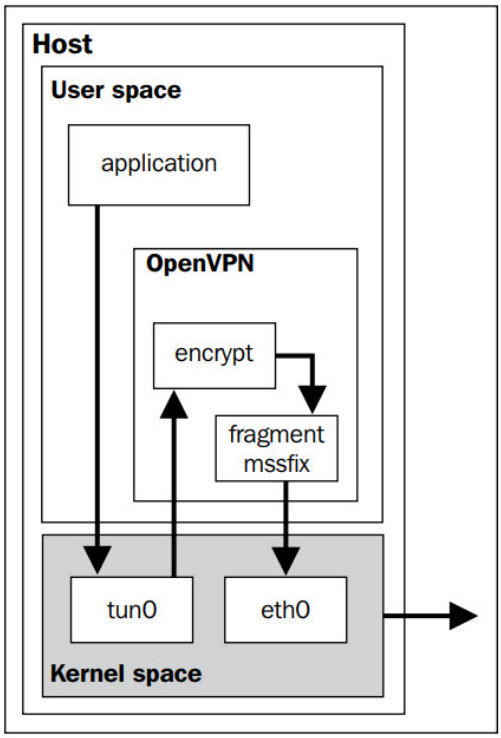

Figure 3.1: openvpn-tun-drive [8] .

The flow of traffic from a user application via OpenVPN is depicted in the diagram 3.1. In the diagram, the application is sending traffic to an address that is reachable via the OpenVPN tunnel. The steps are as follows:

1. The application delivers the packet to the OS.
2. If the traffic needs to be routed by VPN, the operating system forwards packets to the kernel tun device.
3. The kernel tun device forwards the packets to the OpenVPN process
4. The OpenVPN apply required actions like encryption and fragmentation, if necessary, and then deliver it to the kernel again.

5. The kernel picks up the encrypted packet and forwards it to the remote VPN endpoint.

It can also be seen in this diagram that the performance of OpenVPN will always be less than that of a regular network connection. For most applications, the performance loss is minimal and/or acceptable. However, for speeds greater than 1GBps, there is a performance bottleneck, both in terms of bandwidth and latency [8].

3.4.2 The Control and Data Channels

OpenVPN uses two virtual channels to communicate between the client and server:

- A control channel to exchange information and materials between the client and server such as configuration, cipher material and encrypted keys.
- A data channel is to exchange the encrypted payload

3.5 Deployment Models

3.5.1 Point-to-Point Mode

The pre-shared keys are the only available option in point-to-point mode when using OpenVPN, in this mode, only a single endpoint can connect to a server instance at a time. The client is responsible to initiate the connection, the other endpoint is considered as the server, here both endpoints are more or less equal when it comes to functionality.

3.5.2 Client-Server Mode

Point-to-point topology is an excellent way to connect a small number of sites. In large scale scenarios when there are many endpoints to interconnect, it is better to employe client/server mode.

The client/server mode was first introduced with OpenVPN 2.0. In this mode, the server is a single OpenVPN process to which multiple

clients can connect. Each authenticated and authorized client is assigned an IP address from a pool of IP addresses that the OpenVPN server manages. Clients cannot communicate directly with one another. All traffic flows via the server, which has both advantages and disadvantages [8].

Pros and cons of the Client-Server Mode

The advantages are as follows:

- By default, clients are not allowed to communicate directly, but the administrator is able to control the traffic by using the OpenVPN client-to-client option to allow clients to communicate with each other.

- Easier to ensure connectivity of many clients through one single server that can be reached by many different clients.

The disadvantages are as follows:

- As all traffic is passing from client to server and vice versa, the topology can be considered as a single point of failure in large scale scenarios.

- The performance of the network is lower compare to direct client-to-client connectivity because all traffic between two clients must be routed by the server as the midpoint of the communication.

3.6 Setting up the Public Key Infrastructure

Before we can set up a client-server VPN, we must set up a Public Key Infrastructure. In the client-server model, OpenVPN using a PKI with X.509 certificates and private keys.

Also, we need to generate a Diffie-Hellman parameter file that is required for VPN session keys, the session keys are temporary keys and are generated at the first set up.

3.7 Config files against the Command line

It is also possible to use configuration files for both server and client-side to apply options used to setup the OpenVPN.

3.7.1 Server Configuration

The server configuration file contains the following lines:

- **proto udp** This is the default protocol, TCP is another candidate.
- **port 1194** This is the default local port that OpenVPN will listen to, but any valid and available port number can be used.
- **dev tun** This specifies the name of the tun device that will be used for the server.
- **server <network>** The network specify the subnet and mask to use for the VPN server and clients.
- **keepalive** This is used to make sure that the VPN connection remains up, even if there is no traffic flowing over the tunnel.
- **dh <path to Diffie Hellman file>** This specifies the path to the DH file that is required for the OpenVPN server.
- **ca <path to CA file>** This specifies the path to the CA file.
- **cert <path to X.509 certificate file>** This specifies the path to the server X.509 public certificate file.
- **key <path to private key file>** This specifies the path to the server private key file.
- **client-config-dir** determine whether a CCD file is located.
- **route <network>** The route entries that sould be routed over the vpn, these routes also will be added on the clients, telling them to route those networks over the vpn.

- **client-to-client** OpenVPN also allows you to set up client-to-client traffic. By default, the VPN clients are not allowed to communicate directly with each other.

Listing 3.1: Example OpenVPN 2.0 config file for server

```
port 1194
proto udp
dev tun
ca    ca.crt
cert  server.crt
key   server.key
dh    dh2048.pem
server 10.8.0.0 255.255.255.0
client-config-dir /etc/openvpn/ccd
client-to-client
keepalive 10 120
```

3.7.2 Client Configuration

- **client** This puts OpenVPN into client mode.
- **proto udp** This specifies the protocol to use.
- **remote openvpnserver.example.com** This specifies the name of the VPN server to connect to.
- **port 1194** This is the port that the OpenVPN client will use to connect to the server.
- **dev tun** This specifies the name of the tun device that will be used for the server.
- **ca <path to CA file>** This specifies the path to the CA file.
- **cert <path to X.509 certificate file>** This specifies the path to the server X.509 public certificate file.
- **key <path to private key file>** This specifies the path to the server's private key file.

Listing 3.2: Example OpenVPN 2.0 config file for client

```
client
proto udp
dev tun
remote openvpn03 1193
ca     ca.crt
cert   server.crt
key    server.key
```

3.7.3 Client-specific Configuration – CCD files

If we need to define VPN options per client or may overwrite global server options, client-config-dir is very useful for this. This option is also vital if you want to route a subnet from the client-side to the server-side.

- **push** This is useful for pushing DNS and WINS servers, routes, and so on to the client.
- **iroute** This is useful for routing IPv4 client subnets to the server.
- **ifconfig-push** This is useful for assigning a specific IPv4 address to a client (tunnel-IP).

Listing 3.3: Example OpenVPN 2.0 config file for CCD

```
ifconfig-push 10.8.0.20 255.255.255.0
iroute 10.3.1.0 255.255.255.0
push "route 10.2.1.0 255.255.255.0"
```

3.8 Our Topology

In this thesis, we employed Client/Server Mode with tap Devices to establish a full mesh flat network able to guarantee end-to-end communication between distributed Kubernetes clusters on different public cloud providers, here AWS, Azure, and Google.

In order to achieve the goal, our topology consists of three server instances (Ubuntu 16.04) each one located on different cloud providers. To shape a redundant mesh network each instance plays both server and client roles at the same time. Meanwhile, one server is responsible for certificate management and play the role of the CA server.

Figure 3.2 shows a graphic representation of a full mesh OpenVPN site-to-site topology:

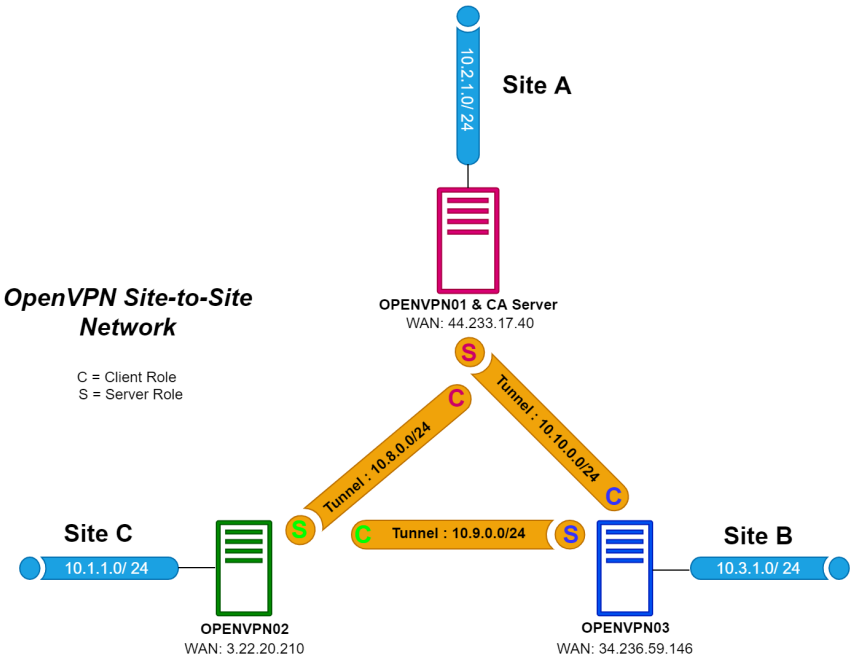

Figure 3.2: An OpenVPN site-to-site topology

3.8.1 How to set up our VPN topology on AWS

1. Create and configure three VPC on three AWS regions

   (a) configure subnets and route tables

      i. A VPC is a virtual network specific within AWS for you to hold all your AWS services. It is a logical data center in AWS and will have gateways, route tables, network access control lists (ACL), subnets, and security groups.

2. Launch three new EC2 Instances on each AWS region.

(a) An EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS) infrastructure. We use Ubuntu OS because that's easy to configure and its scripts are easily available.

(b) Amazon EC2 is hosted in multiple locations worldwide. These locations are composed of Regions, Availability Zones, Local Zones, and Wavelength Zones. Each Region is a separate geographic area. Each Amazon EC2 Region is designed to be isolated from the other Amazon EC2 Regions.

3. Configure the Instance's Security Groups to only allowed traffic has to access the VPN server.

4. Create an Elastic IP for each instance

   (a) Upon launching an EC2 instance, a Public IP address is assigned so that that instance is available. As soon as the instance is shut down, a new public IP gets assigned for the same instance. This means if we set up the VPN server with the default IP, we won't be able to access the VPN if the instance is shut down. Elastic IP solves this issue and assigns a permanent IP address.

5. Setup remote OpenVPN servers using Ansible playbook 3.3.

   Figure 3.3 shows a graphic representation of our Ansible playbook structure, employed to deploy OpenVPN servers..

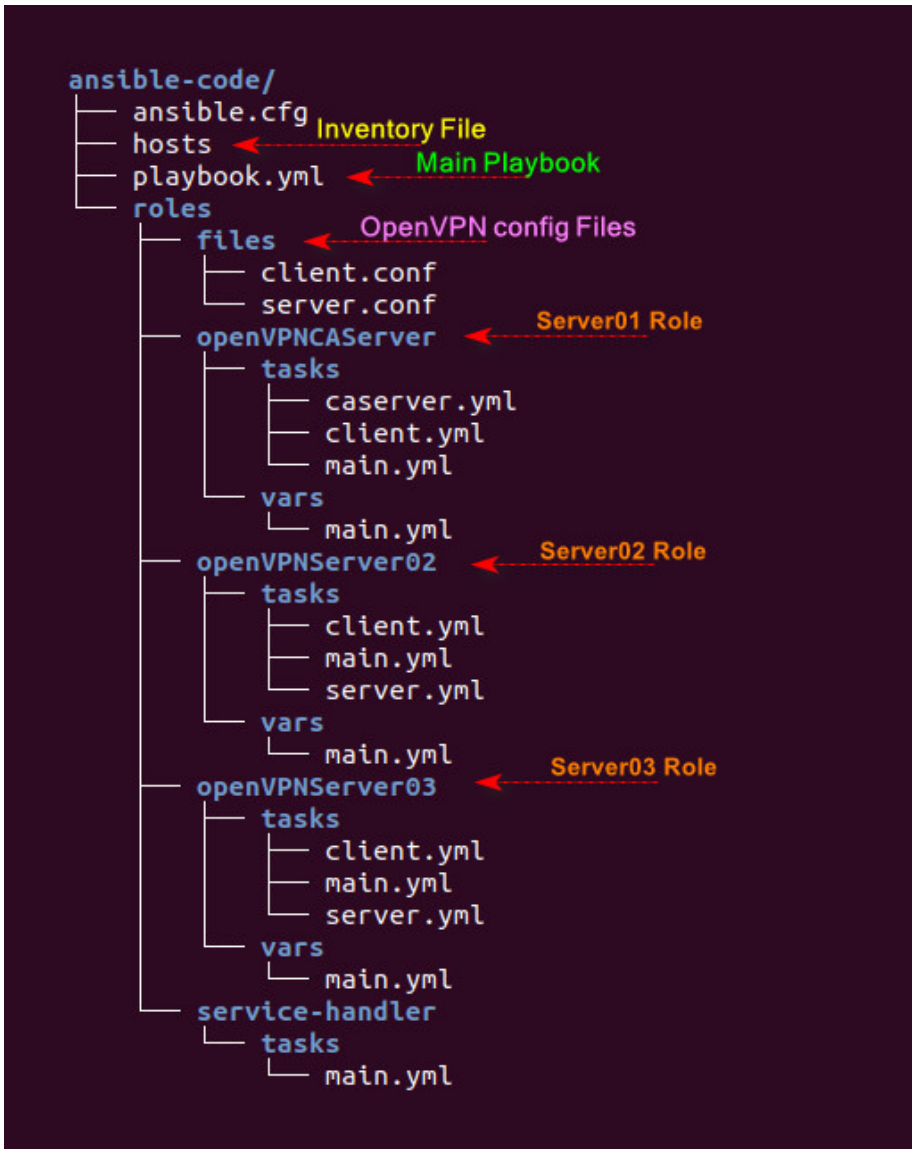

Figure 3.3: Ansible playbook structure

Figure 3.4 shows a graphic representation of our OpenVPN topology on AWS.

As can be seen from Figure 3.4 we tried to simulate the final OpenVPN topology by three isolated servers in different AWS regions, to have an end-to-end communication between three completely separated infrastructure. Here each server is responsible to advertise its local networks to the neighbors and also route external networks across a local AWS router toward the responsible destination server. All the

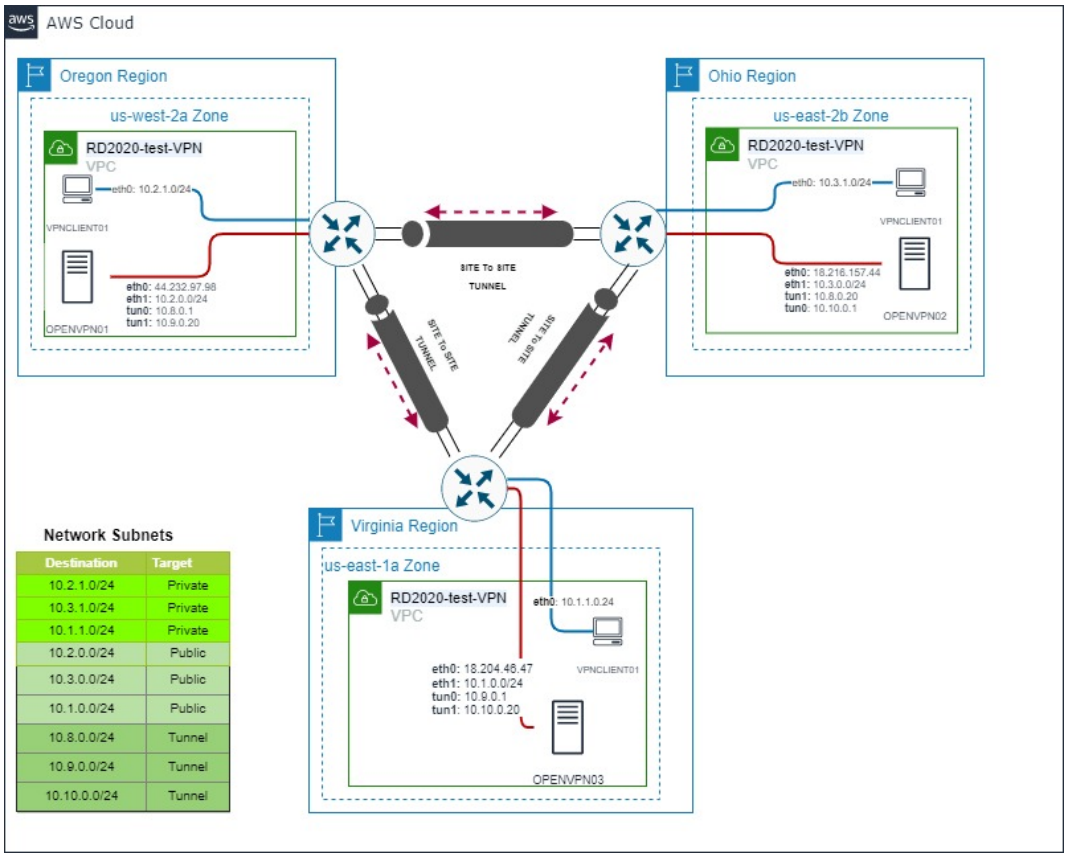

Figure 3.4: End-to-end connectivity between three AWS region through OpenVPN

traffic will be route inside secure transparent tunnels.

The code is consists of three major parts, certificate management, client-side configuration, and server-side configuration, we tried to keep the highest level of modularity for future developments. Here in order to reduce the overhead of certificate management among clients and servers in different regions, one of the OpenVPN servers plays the role of CA server to sign and dispatch all required certificates required for the implementation.

Chapter 4

Service Mesh Architecture

A service mesh is a dedicated infrastructure layer that adds features to a network between services. It allows to control traffic and gain insights throughout the system. A service mesh does not require code changes. Instead, it adds a layer of additional containers that implement the features reliably and agnostic to technology or programming language. [5]

Although service meshes have no impact on the code, but they change operations procedures and require knowledge of new concepts and technology. Choosing the most flexible service mesh with the most features seems logical at first.

There are many reasons to love Istio among all available solution in the market 4.1, but among all of them there are few highlighted features such as:

- Being a free solution to run
- Has a large community of following
- Excellent routing and networking features
- Support multi-cluster implementation

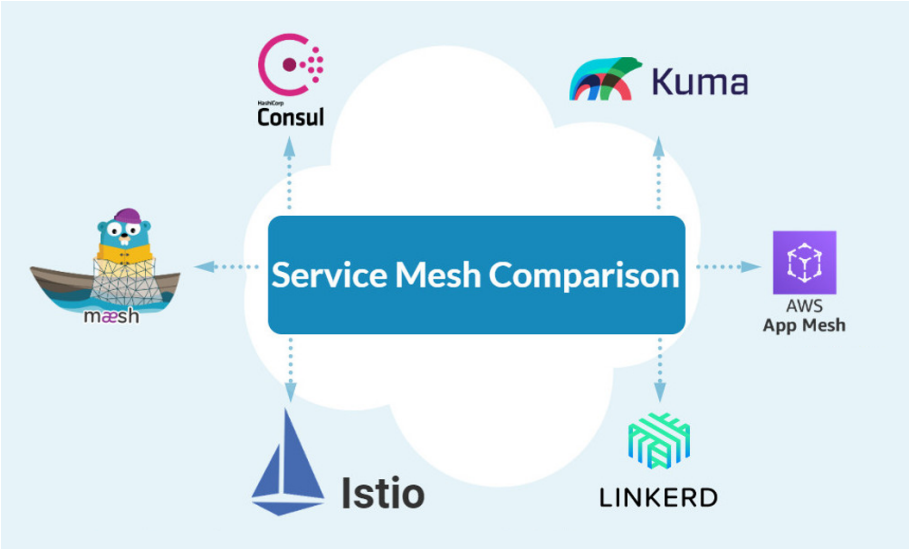

Figure 4.1: Available top service-mesh solutons in market.

4.1 Traffic Management in Istio

Traffic management is among the main capabilities of all service meshes. With Istio's networking APIs traffic management become easier and under control, enabling the ability to do things like canary new deployments.

4.1.1 Understanding How Traffic Flows in Istio

At the first step, it is crucial to know how Istio's network flows traffics across mesh through available components in Istio topology, Istio networking configurations, and the mesh's service proxies.

As the data-plane service proxy, Envoy intercepts all incoming and outgoing requests at runtime (as traffic flows through the service mesh). This interception is done transparently via iptables rules or a Berkeley Packet Filter (BPF) program that routes all network traffic, in and out through Envoy [7].

4.1.2 Understanding Istio's Networking APIs

Consequently, Istio's network configuration has adopted a name-centric model, in which:

- Gateways expose names
- VirtualServices configure and route names
- DestinationRules describes how to communicate with the workloads behind a name
- ServiceEntrys enable the creation of new names

Application requests initiate with the call to the service's name, as shown in Figure 4.2.

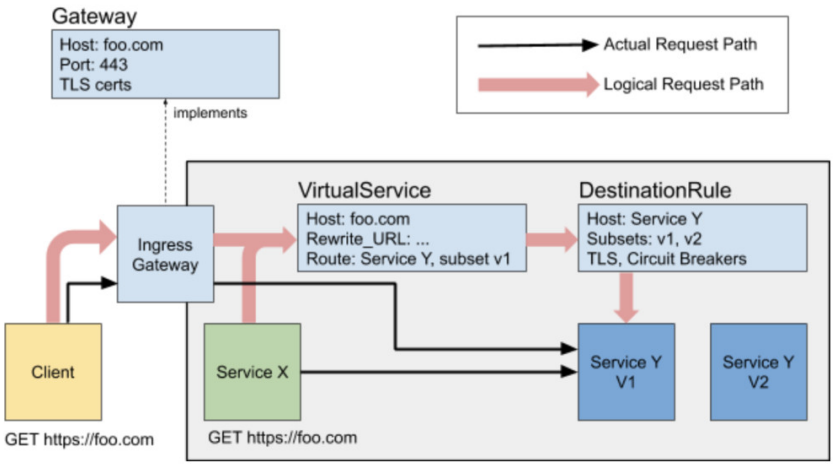

Figure 4.2: Istio core networking concepts implicated as traffic flows through the system [7].

- **ServiceEntry** enables adding additional entries into Istio's internal service registry so that auto-discovered services in the mesh can access/route to these manually specified services. Entries in the service registry can receive traffic by name and be targeted by other Istio configurations.

Listing 4.1: Example of an Istio ServiceEntry

```
1
2  apiVersion: networking.istio.io/v1alpha3
3  kind: ServiceEntry
4  metadata:
5    name: http-server
6  spec:
7    hosts:
8    - altran.domain.com
9    ports:
10   - number: 80
11     name: http
12     protocol: http
13   resolution: STATIC
14   endpoints:
15   - address: 4.4.4.4
```

Given the ServiceEntry in Example 4.1, service proxies in the mesh will forward requests to altran.domain.com to the IP address 4.4.4.4.

- **DestinationRule** They allow a service operator to describe how a client in the mesh should call their service. With DestinationRules, we can configure a low-level connection pool settings like the number of TCP connections allowed to each destination host, the maximum number of outstanding HTTP1, HTTP2, or gRPC requests allowed to each destination host, and the maximum number of retries that can be outstanding across all of the destination's endpoints [7].

  The Example 4.2 shows a DestinationRule that allows a maximum of ten TCP connections per destination endpoint and a maximum of 1,00 concurrent HTTP2 requests over those four TCP connections.

Listing 4.2: Example destinationRule configuring low-level connection poo

```
1
2 apiVersion: networking.istio.io/v1alpha3
3 kind: DestinationRule
4 metadata:
5   name: cloud.altran
6 spec:
7   host: "cloud.altran.svc.cluster.local"
8   trafficPolicy:
9     tcp:
10      maxConnections: 10
11    http:
12      http2MaxRequests: 100
```

- **VirtualService** A VirtualService defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol, as shown in Example 4.3.

Listing 4.3: Example of an Istio VirtualService

```
1
2 apiVersion: networking.istio.io/v1alpha3
3 kind: VirtualService
4 metadata:
5  name: cloud-department
6 spec:
7  hosts:
8  - cloud.altran.svc.cluster.local
9  http:
10 - route:
11   - destination:
12       host: cloud.altran.svc.cluster.local
```

The VirtualService in Example 4.3 forwards traffic addressed to cloud.altran to the destination cloud-altran.svc.cluster.local.

- **Gateway** describes a load balancer operating at the edge of the

mesh receiving incoming or outgoing HTTP/TCP connections. The specification describes a set of ports that should be exposed, the type of protocol to use, etc.

Suppose that you have a webserver.altran.svc.cluster.local service deployed in your mesh that serves your website, altran.com. You can expose that webserver to the public internet using a Gateway to map from your internal name, webserver.altran.svc.cluster.local, to your public name, altran.com. You also need to know on what port to expose the public name and the protocol with which to expose it, as shown in Example 4.4.

Listing 4.4: Example Gateway definition, exposing HTTP/80:

```
1
2  apiVersion: networking.istio.io/v1alpha3
3  kind: Gateway
4  metadata:
5    name: altran-gateway
6  spec:
7    selector:
8      app: gateway-workloads
9    servers:
10   - hosts:
11     - altran.com
12     port:
13       number: 80
14       name: http
15       protocol: HTTP
```

4.2 Canary Deployment

Canary deployment is the practice of sending a small portion of traffic to newly deployed workloads, gradually ramping up until all traffic flows the new workloads. The goal is to verify that a new workload is healthy (up, running, and not returning errors) before sending all traffic to it [7].

the simplest canary deployment is a percentage-based traffic split.

We can start by sending 10% of traffic to the new version, gradually pushing new VirtualService configurations, ramping traffic up to 100% to the new version, as shown in Example 4.5.

Listing 4.5: Example canary deployment using traffic shifting

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: cloud-canary-virtual-service
spec:
  hosts:
  - cloud.altran.svc.cluster.local
  tcp:
  - route:
    - destination:
        host: cloud.altran.svc.cluster.local
        subset: v2
      weight: 10
    - destination:
        host: cloud.altran.svc.cluster.local
        subset: v1
      weight: 90
```

In this thesis, we apply such describe methods and concepts as destinatioRule, virtualService, and Canary deployment to shape the traffic across the mesh.

4.3 Multiple-Cluster Meshes

Single-cluster service mesh deployments might be enough in some environments. But there are other scenarios that need multiple clusters service mesh deployments.

The communication between distributed Kubernetes clusters on separated service meshes can be provided through Istio multi-control plane deployment to unify Kubernetes clusters.

4.3.1 Istio Multicluster (single mesh)

Shared or single control planes 4.3 is a centralized approach for connecting service meshes into a single service mesh. Here we have a cluster that serves as the master cluster, and other clusters play the role of remote cluster. Here, there is one control plane entire of the mesh, to dispatch the routing policies to remote data planes. A single-control plane Istio deployment can span across the clusters as long as there is network connectivity between them and no IP address range overlap.

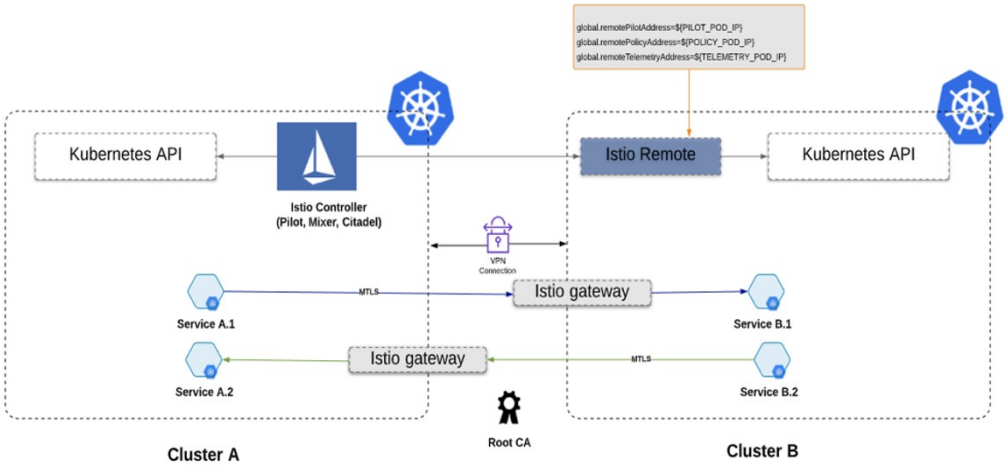

Figure 4.3: The Istio multicluster approach: a single-control-plane Istio deployment with direct connection (flat networking) across clusters [15].

In this scenario, as can be seen from Figure 4.3, multiple Kubernetes API servers and Istio components in each cluster need to see each other for integration. Once connected, Envoy communicates with a single control plane and forms a mesh network across multiple clusters.

To guarantee cross-cluster communications, here there are also two different approaches for implementation:

- In the first approach, the remote clusters are able to have direct communication with the master cluster as the result of having a flat network topology across the clusters. This is the method we employed in our implementation.

- Also in multi-network environments that we are not able to establish a flat network topology without overlap, here connectivity across the clusters can be handled and managed through ingress Istio gateways.

4.3.2 Istio Multicluster (mesh federation)

Replicated control plane 4.4 it is decentralized approach to unifying meshes. This approach is useful for the scenario that we have relatively different configurations per service mesh under different administrative domains running in different regions.

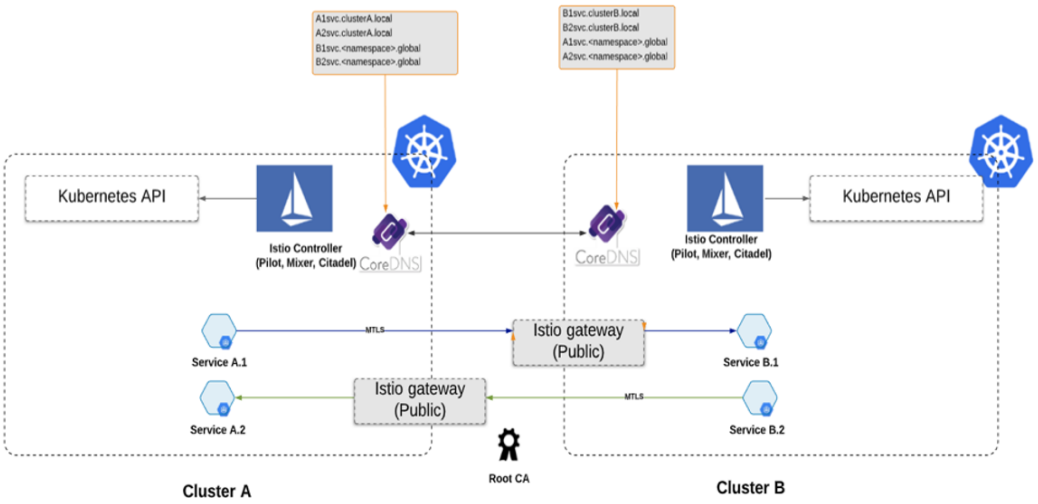

Figure 4.4: The Istio multicluster approach: a multi-control-plane Istio deployment [15].

There are three crucial elements in this architecture as following:

- For multi-cloud networks without VPN connectivity or with overlapping IP ranges, Istio replicated control planes can be used to connect services across the clusters. Instead of using a shared Istio control plane to manage the mesh, in this approach each cluster has its own Istio control plane installation, each managing its own endpoints. The IP address of the Istio ingress gateway service in each cluster must be accessible from every other cluster, ideally using L4 network load balancers (NLBs).

- Here for each service in a given cluster that needs to be accessed from a different remote cluster requires a ServiceEntry configuration in the remote cluster.
- Each cluster has its own DNS domain to be resolved through the CoreDNS component, installed with Istio.

4.3.3 Pros and cons of each approach

Through Table 4.1, we are trying to demonstrate the advantages and disadvantages of each approaches to manage a Multi-Cluster Mesh.

Pros and Cons	Mesh Federation	Single Mesh
Complexity in implementation	Higher	Lower
Monitoring, Trouble shooting	Per site	Centralized
Level of availability	Higher	Lower
Isolation in configuration	Per site	Master controller
Logical view	Per site	Single view
Administrative domain	Per site	Single domain
Single point of failure		Master controller

Table 4.1: Pros and cons of each approach in multi-cluster Istio management

4.4 Our Architecture

As can be seen from the depicted Figure 4.5 in this thesis We employed a combination of technologies and services such as OpenVPN, Rancher, Istio, AWS to implement a shared control plane service mesh scenario able to handle traffic management across the mesh between distributed microservices on various public cloud provider.

4.4.1 How to Set up our Service Mesh

1. Deploy Kubernetes clusters on different public cloud providers, here AWS and Azure by Rancher.

(a) Individual cluster Pod CIDR ranges and service CIDR ranges must be unique across the multi-cluster environment and may not overlap

(b) All pod CIDRs in every cluster must be routable to each other

(c) All Kubernetes control plane API servers must be routable to each other

2. Deploy master Istio Control Plane

(a) This installation guide uses the istioctl command-line tool to provide rich customization of the Istio control plane and of the sidecars for the Istio data plane.

(b) The simplest option is to install the default Istio configuration profile, default enables components according to the default settings of the IstioControlPlane API

Listing 4.6: Istio master controlplane config

```
1
2 istioctl --context master manifest apply \
3   --set values.prometheus.enabled=true \
4   --set values.grafana.enabled=true \
5   --set values.kiali.enabled=true \
6   --set values.kiali.createDemoSecret=true
```

3. Deploy remote Istio Control Plane

(a) Use the following command on the remote cluster to install the Istio control plane service endpoints.

(b) Before you install Istio on the remote cluster, you need to get the Pilot Pod IP address and the Policy and Telemetry Pod IP addresses from the master cluster. These IP addresses are configured in the remote cluster, which connects back to a shared Istio control plane.

Listing 4.7: Istio remote controlplane config

```
1
```

```
istioctl --context remote manifest apply \
  --set profile=remote \
  --set values.global.controlPlaneSecurityEnabled=false \
  --set values.global.createRemoteSvcEndpoints=true \
  --set values.global.remotePilotCreateSvcEndpoint=true \
  --set values.global.remotePilotAddress=${PILOT_POD_IP} \
  --set values.global.remotePolicyAddress=${POLICY_POD_IP} \
  --set values.global.remoteTelemetryAddress=${TELEMETRY_POD_IP} \
  --set gateways.enabled=false \
  --set autoInjection.enabled=true
```

4. Configure cross-cluster service registries

   (a) To enable cross-cluster load balancing, the Istio control plane requires access to all clusters in the mesh to discover services, endpoints, and pod attributes. To configure access, create a secret for each remote cluster with credentials to access the remote cluster's kube-apiserver and install it in the primary cluster. This secret uses the credentials of the istio-reader-service-account in the remote cluster

   (b) Set the environment variables needed to build the kubeconfig file for the istio-reader-service-account service account

   (c) Create a kubeconfig file in the working directory for the istio-reader-service-account service account with the following command

   (d) Create a secret and label it properly for each remote cluster on the the master

Listing 4.8: Set the environment variables to build the kubeconfig file

```

export WORK_DIR=$(pwd)
CLUSTER_NAME=$(kubectl config view --minify=true -o jsonpath='{.clusters[].name}')
export KUBECFG_FILE=${WORK_DIR}/${CLUSTER_NAME}
SERVER=$(kubectl config view --minify=true -o jsonpath='{.clusters[].cluster.server}')
NAMESPACE=istio-system
SERVICE_ACCOUNT=istio-reader-service-account
SECRET_NAME=$(kubectl get sa ${SERVICE_ACCOUNT} -n ${NAMESPACE} -o jsonpath='{.secrets[].name}')
CA_DATA=$(kubectl get secret ${SECRET_NAME} -n ${NAMESPACE} -o jsonpath="{.data['ca\.crt']}")
TOKEN=$(kubectl get secret ${SECRET_NAME} -n ${NAMESPACE} -o jsonpath="{.data['token']}" | base64 --decode)
```

Listing 4.9: Create a kubeconfig file

```
cat <<EOF > ${KUBECFG_FILE}
apiVersion: v1
clusters:
   - cluster:
       certificate-authority-data: ${CA_DATA}
       server: ${SERVER}
     name: ${CLUSTER_NAME}
contexts:
   - context:
       cluster: ${CLUSTER_NAME}
       user: ${CLUSTER_NAME}
     name: ${CLUSTER_NAME}
current-context: ${CLUSTER_NAME}
kind: Config
preferences: {}
users:
   - name: ${CLUSTER_NAME}
     user:
       token: ${TOKEN}
EOF
```

Listing 4.10: Create a secret and label for each remote cluster

```

kubectl create secret generic ${CLUSTER_NAME} --from-file ${KUBECFG_FILE} -n ${NAMESPACE}
kubectl label secret ${CLUSTER_NAME} istio/multiCluster=true -n ${NAMESPACE}
```

5. Finally, we span a shared control plane scenario across the clusters that is accessible through an AWS load balancer from outside.

Figure 4.5 shows a graphic representation of our multi-cloud service mesh architecture.

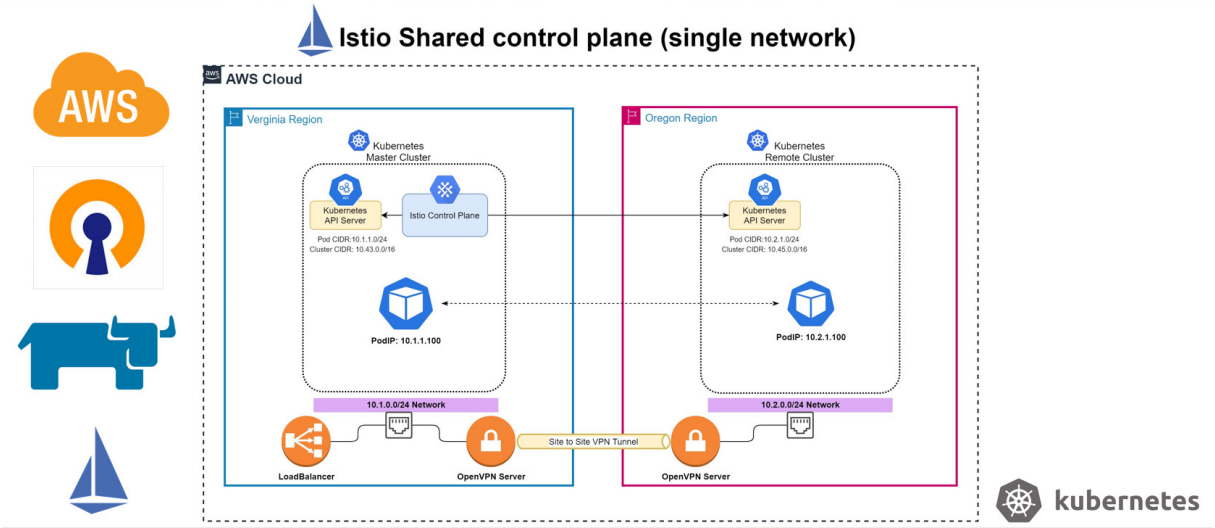

Figure 4.5: Multiple Kubernetes clusters running on AWS, integerated with a shared Istio control plane using OpenVPN.

Chapter 5

CI/CD Pipeline Architecture

As the result of previous phases and described technologies, we are now able to have access to a wonderful integrated multi-cloud infrastructure between AWS and Azure with the ability of traffic management, so its time to think about workflow portability.

Workflow portability is what makes deploying anywhere possible. Instead of having to tailor certain workflows to certain clouds, developers can have one workflow with cloud-independent DevOps processes and frameworks for making deployment decisions [10].

In this thesis, we employed GITLAB as a professional solution for code repository and DevOps pipelines. There are two other technologies that We used in this section, Helm chart and Docker hub to deliver Kubernetes resources to destination clusters and hosting docker images respectively.

5.1 Why Gitlab CI/CD?

In order to have a mature CI/CD with all the required fundamentals, many DevOps platforms are dependent on other tools to satisfy requirements. Many organizations have to maintain complicated and costly toolchains to take advantage of CI/CD capabilities. Undoubtedly, maintaining a separate source code management like GitHub connected to a separate testing tool, that connects to their CI tool, which connects

to a deployment tool like Puppet, that also connects to various security and monitoring tools is challengable and complex. Instead of focusing on building and development, organizations have to waste their time to maintain and manage a complicated toolchain.

GitLab is a single platform for the entire DevOps lifecycle, meaning it supplies all the fundamentals for CI/CD in one integrated environment.

5.1.1 How GitLab Enables Multi-Cloud

GitLab CI/CD is a very powerful platform to support CI/CD with a lot of various features. But still, we need to know what a pipeline is, and how to see a branch deployed to an environment. In this section I will try to cover necessary concepts and features as possible, highlighting how the end users can apply them:

Anytime developers apply a change in code they save their modifications in the format of a commit in Gitlab, then the other developers are able to review the code. If GitLab CI/CD has been configured, GitLab will also perform some tasks on that commit. This work is executed by a runner. We can consider a runner as a server that executes instructions listed in the .gitlab-ci.yml file and reports the result back to GitLab itself, which will show it in his graphical interface.

5.1.2 Gitlab Pipelines

Every commit that is pushed to GitLab generates a pipeline attached to that commit. A pipeline is a collection of jobs split into different stages. All the jobs in the same stage run concurrently (if there are enough runners) and the next stage begins only if all the jobs from the previous stage have finished with success [23]. A pipeline can be failed as soon as a job fails. There is an exception for this, if a job type is manual, then a failure will not cause the pipeline to fail.

Each job belongs to a single stage. All jobs in a single stage run in parallel. Execution of the next stage depends on the result of the previous stage if all jobs from the previous stage complete successfully.

Figure 5.1 shows a graphic representation of a Gitlab pipeline and its stages.

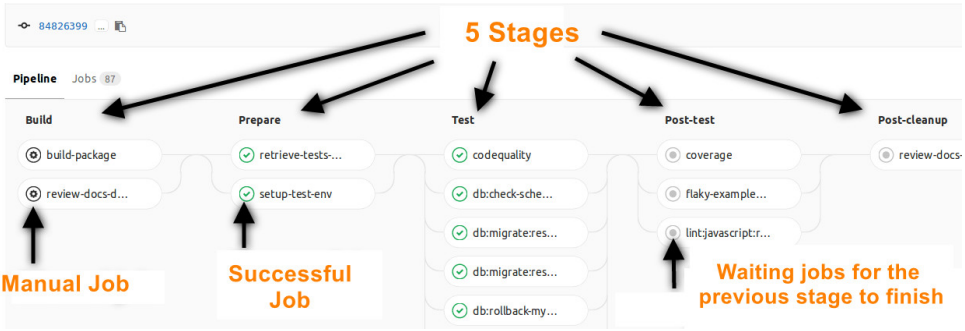

Figure 5.1: The Information about stages and stages' status [23]

Example 5.1 shows a sample Gitlab pipeline configuration script with three stages.

Listing 5.1: Sample Gitlab pipeline configuration script

```
1
2  stages:
3    - build
4    - test
5    - deploy
6
7  build website:
8    stage: build
9    script:
10     - npm install
11     - npm install -g gatsby-cli
12     - gatsby build
13   artifacts:
14     paths:
15       - ./public
16
17 test artifact:
18   image: alpine
19   stage: test
20   script:
21     - grep -q "Gatsby" ./public/index.html
22
23 test website:
24   stage: test
25   script:
26     - npm install
27     - npm install -g gatsby-cli
28     - gatsby serve &
29     - sleep 3
30     - curl "http://localhost:9000" | tac | tac | grep -q "Gatsby"
31
32 deploy to surge:
33   stage: deploy
34   script:
35     - npm install --global surge
36     - surge --project ./public --domain instazone.surge.sh
```

Types of pipelines

1. **Basic Pipelines** This is the simplest pipeline in GitLab that runs everything in each stage concurrently, followed by the next stage.

2. **Parent-child pipelines** Splitting complex pipelines into multiple pipelines with a parent-child relationship can improve performance by allowing child pipelines to run concurrently.

3. **Pipelines for Merge Requests** In a basic configuration, GitLab runs a pipeline each time changes are pushed to a branch (per commit). If you want the pipeline rather than for every commit, run jobs only on commits that are associated with a merge request, you can use pipelines for merge requests.

   (a) **Job** is the smallest unit to run in GitLab CI/CD, a collection of instructions that a runner has to execute. A single job can contain multiple commands (scripts) to run.

   A job can be automatic, so it starts automatically when a commit is pushed, or manual. A manual job has to be triggered by someone manually. This can be useful, for example, to automate a deploy, but still to deploy only when someone manually approves it. A job can also build artifacts that users can download them.

      i. **Artifacts** As we said, a job can create an artifact that users can download to test. It can be anything, like an application for Windows, an image generated by a PC. Every pipeline collects all the artifacts from all the jobs, and every job can have multiple artifacts.

5.2 Our Architecture

The ultimate goal of this thesis is to design an architecture to facilitate software delivery in a multi-cloud infrastructure with the highest level of availability, agility, and scalability. Until now, we could construct a stable infrastructure able to supply our requirements, but in order to complete this chain, we need to also have a professional DevOps solution in application layer to handle CI/CD stages. we chose Gitlab for this

step. Although it is equipped with the necessary features required for multi-cloud environments, we need to tailor them perfectly to support the desired goals.

5.3 How to set up our DevOps pipelines

1. Desing the high-level workflow 5.2 of DevOps pipeline, including the relation between pipelines, jobs, and stages

   (a) At can be seen from the depicted picture, we handle CI/CD through three different branches, Master, Development, and Deployment branch. In this project we used a combination of methods to achieve the desired result, we used different types of pipelines and jobs to create the workflow and handle CI/CD stages.

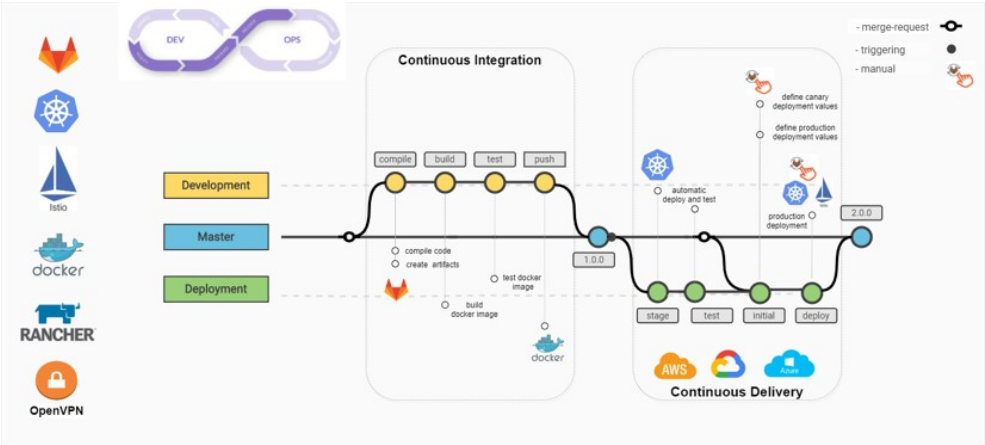

Figure 5.2: An overview of Gitlab pipeline architecture

2. Integrate Gilab with distributed Kubernetes clusters on cloud providers, here AWS and Azure

3. Create Gitlab project and required branches

   (a) Here Master is the main and default branch which is responsible to define the parent pipeline at the beginning to share with other branches.

(b) Development branch is responsible to maintain microservices code and also handle CI tasks.

(c) Deployment branch is responsible for software delivery in a multi-cloud environment, here AWS, Azure and also controlling traffic distribution through Istio canary deployment.

4. Prepare required resource for each defined stage per pipeline, including Helm charts and YAML files.

(a) Development branch includes microservices code and profiles, Docker file and certificates.

Figure 5.3 shows a graphic representation of Development branch.

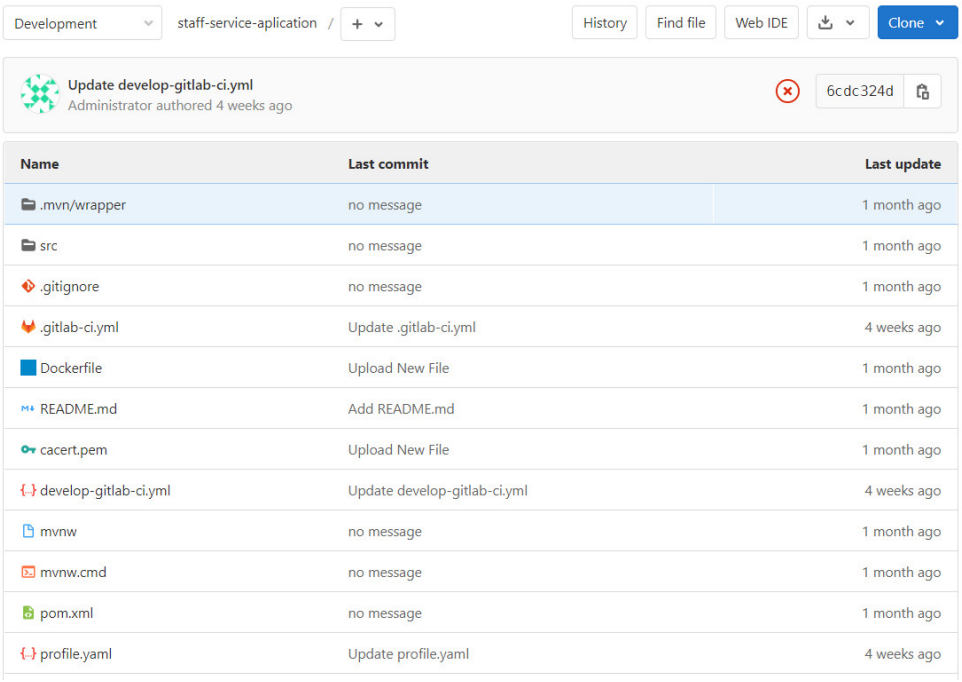

Figure 5.3: Development branch

(b) Deployment branch includes kubernetes manifest file, Istio virtualService, and destinationRule in Helm chart format.

Figure 5.4 shows a graphic representation of Deployment branch.

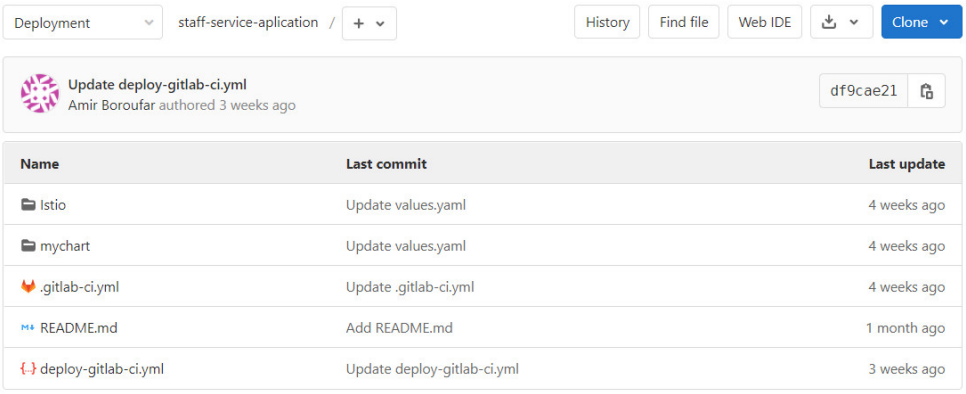

Figure 5.4: Development branch

5. Create pipelines for each branch according to pre-defined stages and the designed workflow.

   (a) Development pipeline 5.5 is responsible to compile the code and create artifacts, build docker images and tests, push docker images on Docker Hub. The main role of the second stage in the parent pipeline is to trigger the remote pipeline in the Deployment branch for continuous deployment.

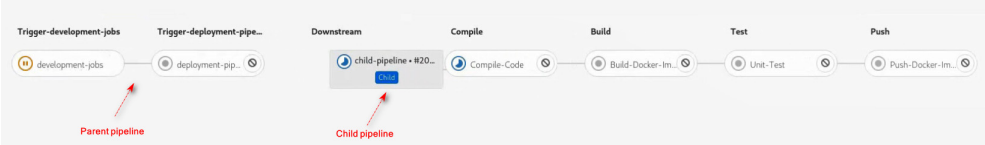

Figure 5.5: Development pipeline

   (b) Deployment pipeline 5.6 has four stages, the first two will be execute automatically to deploy Kubernetes applications on stage environment to confirm application functionalities, after that we have two other stages to deploy Kubernetes applications on production environments through Helm charts and control traffic distribution by canary deployment method, these stages will be executed manually by DevOps engineers.

   (c) Both Deployment and Development branches have their own child pipeline plus to the inherited parent pipeline from the

Figure 5.6: Deployment pipeline

Master branch. The Parent pipelines are triggered by merge request.

5.4 Real scenario on AWS and Azure

After a deep analysis of the proposed multi-cluster architecture to show the result of our job, we carried out a real scenario to deploy a microservice based application on different public cloud providers, here AWS and Azure through DevOps pipelines.

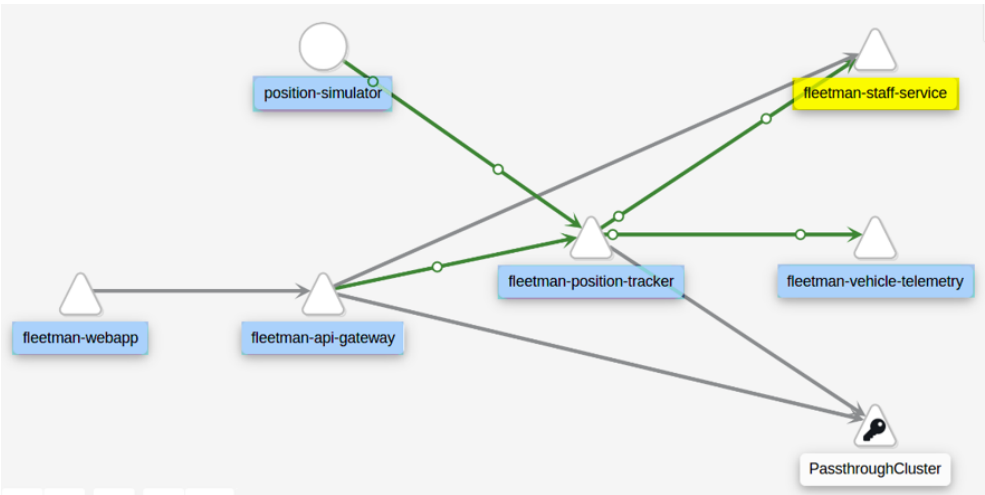

Figure 5.7: Service diagram of a microservice based application

As can be seen from Figure 5.7 this application consists of 6 microservices, each one has its own responsibility to track vehicles on the roads and show the vehicle's telemetries. For instance, the web app module is the front end of the application, and staff-service highlighted with yellow color shows driver's information such as picture and full name. We deployed all microservices except for staff-service on AWS

as the master cluster, next in order to check the performance and functionality of the Istio shared control plane, we created different versions of the staff-service module through CI pipeline and try to deliver on different remote Kubernetes clusters located on AWS and Azure by CD pipeline. Regardless of the cluster, as new microservices are added, they were automatically discovered and added to the mesh. At the next step, we also used Istio canary deployment to control traffic distribution between two different versions of the application (staff-service) located on AWS and Azure.

Chapter 6

Conclusion

This section is trying to show you the most important technologies and solutions we employed in different steps of this research to overcome challenges and finally accomplish each step.

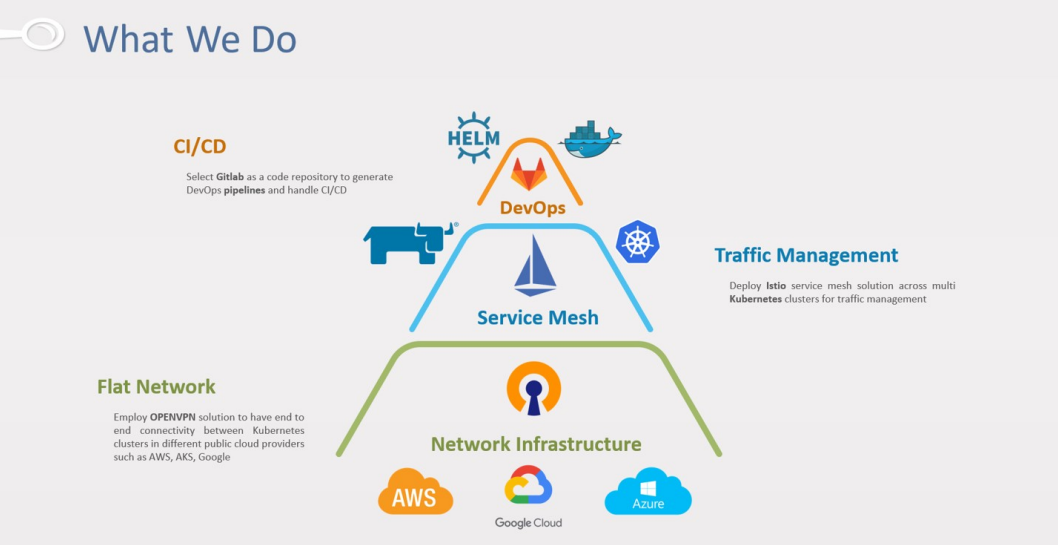

Figure 6.1: The technologies and solutions are employed in different levels.

As can be seen from Figure 6.1 our project is divided into three different layers that each one is responsible to supply the required requirements for the next step.

At the bottom as the first level, we had some challenges regarding network infrastructure, in order to have an end-to-end communication

across a distributed infrastructure among various public providers such as AWS, Azure, we use OpenVPN solution to have a flat network connectivity among different public cloud providers. As a result of this phase Kubernetes clusters are able to communicate with each other for service delivery and shaping the service mesh.

Next in the second phase we employed Rancher for provisioning and managing multiple Kubernetes clusters across the Internet. Then chose Istio service mesh solution for traffic management, for the sake of the excellent routing and networking features and supporting multi-cluster implementation. This architecture supports us to have Istio canary deployment method for software delivery in the last phase.

Finally, it turns to the application layer, as the result of previous phases and described technologies, we are now able to have access to a wonderful integrated multi-cloud infrastructure between AWS and Microsoft Azure with the ability of traffic management, so its time to think about software delivery. At this phase, we employed GITLAB as a professional solution for code repository and also handling DevOps pipelines. There are two other technologies that we employed in this level, Helm chart and docker hub, we used the Helm chart to deliver Kubernetes resources on the remote clusters and the Docker hub for hosting docker images.

In the end we measured the functionality of the proposed architecture by deploying a real microservice-based application on AWS and Azure infrastructure through CI/CD pipelines and control traffic distribution between microservices by Istio canary deployment. The demo application was fully functional and running across two Kubernetes clusters in two environments. The flat network connectivity allows for the flexibility of having multiple clusters while accommodating the ease of governing all microservices through a shared control plane.

Acknowledgements

Lascio le ultime righe di questa tesi in italiano per ringraziare chi mi ha sostenuto durante il lungo viaggio che mi ha condotto a questa meta.

Il ringraziamento principale va al mio supervisore, il professor Fulvio Risso che si è sempre fidato di me, lasciandomi la giusta autonomia per lavorare al meglio sotto la sua supervisione.

Ringrazio Tito Petronio e Pasquale Lepera che mi hanno accompagnato in questo percorso, per il loro costante supporto che mi ha permesso di ampliare e consolidare il mio bagaglio di conoscenze e completare questa tesi.

Infine, ringrazio l'azienda Altran per avermi dato l'opportunità di realizzare questo progetto.

Amir Boroufar

Fulvio Risso

Pasquale Lepera

Tito Petronio

Bibliography

[1] Ruslan Synytsky. Unleashing the Full Potential of Containerization for DevOps, and Avoiding First-Time Pitfalls. https://jelastic.com/blog/containerization-devops/. Online; accessed 17 September 2020.

[2] Alison DeNisco Rayome. How enterprises employ a multi cloud strategy? https://www.techrepublic.com/article/why-86-of-enterprisesemploy-a-multi-cloud-strategy-and-how-it-impact Online; accessed 17 September 2020.

[3] M. A. AlZain, E. Pardede, B. Soh, and J. A. Thom. Cloud computing security: From single to multi-clouds. In *2012 45th Hawaii International Conference on System Sciences*, pages 5490–5499, 2012.

[4] Aneel Kumar, Badri Narayan RD and Ram Ramalingam. Shifting your appliance to a cloud native architecture. https://www.accenture.com/_acnmedia/PDF-104/Accenture-Great-Migration-Shifting-appliance-to-cloud-native-archite pdf. Online; accessed 17 September 2020.

[5] Anja Kammer, Christine Koppelt. Service Mesh Comparison. https://servicemesh.es/. Online; accessed 18 September 2020.

[6] Aviatrix Systems. Networking is Complex in Multicloud Environments. https://a.aviatrix.com/solutions/multicloud-peering.php. Online; accessed 17 September 2020.

[7] L Calcote and Z Butcher. Istio: Up and running: Using a service mesh to connect, secure, control, and observe. page 254. O'Reilly Media, 2019.

[8] E.F. Crist and J.J. Keijser. *Mastering OpenVPN*. Community experience distilled. Packt Publishing, 2015.

[9] Gaurav Agarwal. How Istio Works Behind the Scenes on Kubernetes. https://medium.com/better-programming/how-istio-works-behind-the-scenes-on-kubernetes-aeb8003f2cb5. Online; accessed 17 September 2020.

[10] Gitlab. How to navigate the multi-cloud future. https://about.gitlab.com/resources/. Online; accessed 18 September 2020.

[11] GitLab. Introduction to CI/CD with GitLab. https://docs.gitlab.com/ee/. Online; accessed 18 September 2020.

[12] IBM. Get started with Helm to configure and manage Kubernetes charts. https://www.ibm.com/cloud/architecture/content/course/helm-fundamentals/helm-def/. Online; accessed 17 September 2020.

[13] Istio. Istio Architecture. https://istio.io/latest/docs/ops/deployment/architecture/. Online; accessed 17 September 2020.

[14] Baskaran Jambunathan and Dr Kalpana. Design of devops solution for managing multi cloud distributed environment. *International Journal of Engineering and Technology*, 7:637, June 2018.

[15] Karthik Ramamoorthy. Kubernetes and Multicloud. https://www.cloudtp.com/doppler/kubernetes-and-multicloud/. Online; accessed 18 September 2020.

[16] Kong. Microservices and Multi-Cloud: Building Cross-Cloud Harmony. https://konghq.com/ebooks/microservices-multi-cloud/. Online; accessed 17 September 2020.

[17] Kubernetes community. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/. Online; accessed 18 September 2020.

[18] Leverege. An Introduction to Kubernetes. https://www.leverege.com/ebooks/kubernetes-ebook. Online; accessed 17 September 2020.

[19] Marco Palladino. How Kubernetes Is Modernizing the Microservices Architecture. https://konghq.com/blog/kubernetes-where-are-we-going/. Online; accessed 17 September 2020.

[20] Openvpn. Site-To-Site VPN Routing Explained In Detail. https://openvpn.net/vpn-server-resources/

site-to-site-routing-explained-in-detail/. Online; accessed 17 September 2020.

[21] Rancher. Rancher Server Architecture. https://rancher.com/docs/rancher/v2.x/en/overview/architecture/. Online; accessed 17 September 2020.

[22] Redhat. What is CI/CD?. https://www.redhat.com/en/topics/devops/what-is-ci-cd. Online; accessed 18 September 2020.

[23] Riccardo Padovani. A beginner's guide to continuous integration. https://about.gitlab.com/blog/2018/01/22/a-beginners-guide-to-continuous-integration/. Online; accessed 18 September 2020.

[24] Romana Gnatyk. Microservices vs Monolith: which architecture is the best choice for your business? https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-busine Online; accessed 17 September 2020.

[25] Sathya Bandara. An Introduction to Helm Charts. https://medium.com/@technospace/an-introduction-to-helm-charts-41be1544370c. Online; accessed 19 September 2020.

[26] F. John Roberts Tell. The Modern Firm: Organizational Design for Performance and Growth. *J Manage Governance*, 10(4):455 – 458, 11 2006.