POLITECNICO DI TORINO

Faculty of Engineering

Master of Science in Computer Engineering

Master Thesis

Securing communication between microservices in a multi-cloud scenario using Istio service mesh

Advisors

Prof. Cataldo Basile

Candidate:

Francesco Nevola

Company tutors

Dott. Riccardo Cavazza

Spike Reply

A.Y. 2022/2023

To my family, my friends and me

Nosce te ipsum.

Socrate

Acknowledgements

I thank my family and friends for being very supportive in this journey. Without them, I would not have grown this way and I would not have been able to reach this milestone.
I will always be grateful to you.

Summary

The purpose of this work is to ensure, in a multicloud context, that services in a microservice application communicate securely.
To do this, we first created two clusters on two different cloud providers and then installed Istio service mesh.
After creating the environment, we deployed a microservices application, dividing the services between the two clusters.
Through the creation of ad-hoc metrics, we verified that the services communicate with each other by encrypting the connections. We also created additional metrics to monitor other security-related aspects such as scans of certificates associated with services, number of requests for individual services.
In addition, we test the environment to evaluate if communication take place encrypting data and application is resilient to Dos attack. In conclusion Istio service mesh proved to be a good solution in accomplishing the tasks we set out to do even though it does not guarantee encryption of all communications.
For future work, it is also possible to use this tool to create authentication policies and improve dashboard.

Contents

List of Figures IX

List of Tables XI

1 Introduction 1

1.1 Objectives . . . 3

1.2 Chapter organization . . . 3

2 Service mesh 4

2.1 Introduction . . . 4

2.2 Microservices architecture vs monolithic architecture . . . 4

2.2.1 Microservices architecture . . . 5

2.2.2 Monolithic architecture . . . 6

2.3 Service mesh definition . . . 9

2.4 Services and components . . . 9

2.5 Control plane and data plane . . . 10

3 Service mesh tools 13

3.1 Introduction . . . 13

3.2 Istio . . . 13

3.2.1 Definition . . . 13

3.2.2 Working mechanism . . . 14

3.2.3 Components . . . 15

3.2.4 Security features . . . 17
3.3 AWS app mesh . . . 20
3.3.1 Overview . . . 20
3.3.2 Security features . . . 22
3.3.3 Istio vs AWS service mesh . . . 22
3.4 Linkerd . . . 23
3.4.1 Overview . . . 23
3.4.2 Security features . . . 23
3.4.3 Istio vs Linkerd . . . 25
3.5 Kong mesh . . . 25
3.5.1 Overview . . . 25
3.5.2 Security features . . . 25
3.5.3 Istio vs Kong service mesh . . . 26

4 Multi-cloud cluster design 28
4.1 Introduction . . . 28
4.2 Multi-cloud environment . . . 28
4.2.1 Definition . . . 28
4.2.2 Multi-cloud vs Hybrid-cloud . . . 29
4.2.3 Benefits and Challenges . . . 29
4.3 Kubernetes cluster . . . 30
4.3.1 Definition . . . 30
4.4 Istio deployment models . . . 31
4.5 Environment setup . . . 32

5 Environment monitoring and testing 37
5.1 Intoduction . . . 37
5.2 Environment monitoring . . . 37
5.2.1 Prometheus . . . 37
5.2.2 Configuration . . . 38
5.2.3 Kiali . . . 39

5.2.4 Grafana 40
5.3 Environment testing 48
5.3.1 Istiod traffic analysis 48
5.3.2 Dos attack 48
5.3.3 Nmap test 49

6 Conclusions 51

A Clusters installation 53
A.1 AKS cluster 53
A.2 GCP cluster 54

B Istio configuration and installation 56
B.1 Plugin certificates to cluster 56
B.2 Multi-primary installation 59
B.3 Multi-primary verification 61

References 65

List of Figures

2.1 Microservices architecture [7] 5
2.2 Monolithic architecture [7] 7
2.3 The control plane in a service mesh distributes configuration across the sidecar proxies in the data plane [10] 12

3.1 Sidecar proxy along with every service deployed in Istio mesh [11] 15
3.2 Istio components architecture [12] 16
3.3 Istio security architecture [13] 18
3.4 Istio identity provisioning [13] 18
3.5 Peer authentication policy with "STRICT" mode enabled [13] 19
3.6 Request authentication policy [13] 19
3.7 Example of authorization policy [13] 20
3.8 AWS service mesh components [14] 21
3.9 Linkerd's architecture [21] 24
3.10 Kong's architecture [25] 26

4.1 AKS cluster after creation 33
4.2 GCP cluster after creation 33
4.3 Online boutique architecture [34] 34
4.4 Home page Online boutique 36

5.1 In-mesh Production Prometheus for monitoring multicluster Istio [36] 38
5.2 Prometheus federation configuration 40
5.3 Graph of the micorservices app on Kiali 41
5.4 Istio mesh dashboard 42
5.5 Frontend service on Istio service dashboard 43
5.6 Microservice app dashboard on Grafana 44
5.7 mTLS vs non mTLS connections from my-gcp cluster to my-aks-cluster 45
5.8 mTLS vs non mTLS connections from my-aks cluster to my-gcp-cluster 46
5.9 Hours remaining to the expiration of certificates in AKS cluster 47
5.10 Analysis of packets exchanged by the Istio controller with other services 48
5.11 DOS on istiod 49
5.12 nmap on ingress gateway 50
5.13 nmap on egress gateway 50

B.1 Example of config file in which there are informations about the clusters created 57
B.2 Helloworld version toggling between v1 and v2 after sending at least two requests from the Sleep pod on cluster1 64
B.3 Helloworld version toggling between v2 and v1 after sending at least two requests from the Sleep pod on cluster2 64

List of Tables

2.1 Microservices vs Monolithic advantages and disadvantages [7] 8

3.1 Istio vs AWS service mesh 23
3.2 Istio vs Linkderd 25
3.3 Istio vs Kong mesh 27

Chapter 1

Introduction

Before the advent of microservices-based approaches, software development was primarily driven by monolithic architectures, where an application is built as a single cohesive entity with all its components tightly interconnected. This made software challenging to scale and maintain. Moreover, with the advancement of technology and the evolution of the Internet, those applications moved towards the web. Initially, they followed the full-stack model, where the entire application was handled both on the client and server sides. For these reasons they tended to be complex to manage and difficult to update.
The introduction of "REST APIs" (Application Program Interfaces) led to the adoption of "Service-Oriented Architectures" but complexity remained high. Then, the shift to microservices was a game-changer. This approach allowed complex applications to be developed as a collection of independent services, each handling a specific functionality. Microservices are modular, scalable, and can be managed independently, enabling organizations to develop those applications in more flexible and efficient manners. So, their design and development changed into a technology-agnostic and highly scalable architecture [1].

In parallel, cloud computing model has become increasingly popular in application developement since it is efficient, helps reducing costs, and speeds up timeline [2].
Considering all these aspects, today, microservices-based approach is the most effective way to develop application in cloud [3]. Moreover, many companies decided to rely on multiple cloud providers to benefit from many advantages [4]. However, being able to secure such environment is very challenging for several reasons:

- Cloud computing provides loads of resources for users, such as storage and bandwdith capacities. Because of this it is difficult to have an overall control over them and malicious users could take advantage of this weakness by targetting a certain cloud resource and launching a DDoS (Distributed denial of service) [5].

- Those services and platforms must be continuously updated to be compliant with the latest regulatory polices, which includes the new GDPR (General Data Protection Regulation) [6].

- Malicious insiders could bypass security systems like firewalls and intrusion detection systems, gaining root privilege to network components and tampering with sensitive and confidential data.

- Since software and hardware maintenance is directly handled by cloud services, clients aren't involved in these tasks anymore, leading to threats related to security compliance, hardening, auditing and patching.

- During the transmission of data, if weak authentication and encryption schemes are used, data leakage becomes an issue, resulting in huge economic damages for companies [5].

These are the motives why securing service communication in a multicloud environment is crucial for every application.

1.1 Objectives

The purpose of this work is to use a service mesh tool called "Istio", in a multi-cloud scenario, to manage encryption and authentication of a microservices application, ensure resilience by limiting the damage of a possible DDoS attack, and provide constant monitoring of the entire application.
For the monitoring part, tools such as Prometheus, Kiali and Grafana will be discussed and some security metrics will be analyzed. In addition, tests will be conducted to evaluate whether the proposed security properties are met.

1.2 Chapter organization

The work is organized in this way:

- In the second chapter, we will analyze what is meant by microservices apps and the advantages compared to a monolithic approach. We will also study what a service mesh is and its main components.
- In the third chapter, we will focus on four service mesh tools, with a primary focus on Istio, comparing it to the other three.
- In the fourth chapter, we will create a multi-cloud scenario with Istio and deploy a microservices application.
- Finally, in the last chapter, we will install and use monitoring tools to showcase the potential of the service mesh in gathering data related to the security of our application. In addition, there will performed three security tests.

Chapter 2

Service mesh

2.1 Introduction

This chapter will take a general look at the concept of service mesh.
Before discussing the main topic, it will be defined what is meant by a microservice architecture and what the differences are with a monolithic one.
Then a formal definition of service mesh will be given, which will contain some useful terms for understanding how it works.
Finally the analysis will conclude by looking specifically at the various services and components offered by this paradigm, and by discussing two fundamental components: the control plane and the data plane.

2.2 Microservices architecture vs monolithic architecture

In this section, we will define microservices architecture analyzing its pros and cons and than we will do the same for the monolithic one. Finally we will make a comparison between the two architectures.

2.2.1 Microservices architecture

Microservices applications are characterized by having several independent components that can communicate with each other through proper APIs. Each microservice is managed independently of other services and can be written in different programming language [7].

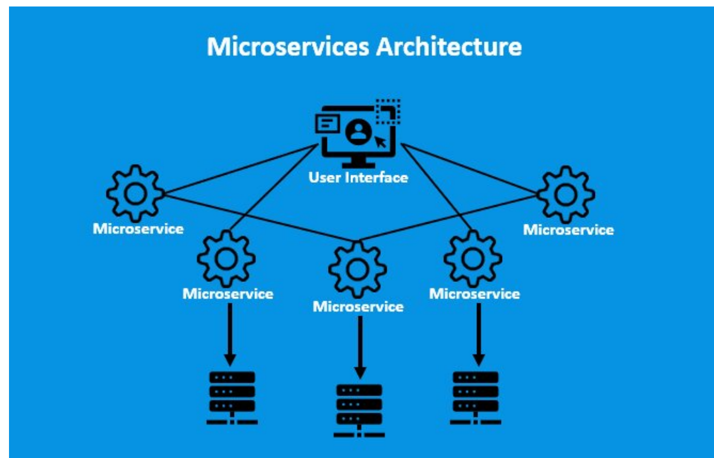


Figure 2.1: Microservices architecture [7]

The advantages of microservices are:

- Agility – Small teams can deploy frequently.
- Flexible scaling – If a microservice is overloaded, new instances of that service can be quickly distrbuted.
- Continuous deployment – Very frequent releases compared to the past.
- Highly maintainable and testable – New features can be experimented with, and it is easier to find and fix bugs.
- Independently deployable – Since microservices are individual units they allow for fast and easy independent deployment of individual features [8].

- Technology flexibility – Teams are free to choice the tools they want to use.
- High reliability – A service can be modified without bringing down the entire application.

The disadvantages of microservices can be summarized as follow:

- Development sprawl – If development sprawl isn't properly managed, it results in slower development speed and poor operational performance [8].
- Exponential infrastructure costs – Each microservice has its own cost.
- Added organizational overhead – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.
- Debugging challenges – Debugging is more complicated.
- Lack of standardization – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.
- Lack of clear ownership – As more services are introduced, so are the number of teams running those services [8].

2.2.2 Monolithic architecture

Monolithic applications are characterized by having all program components in a single block. In this case, if one component needs to be updated, the entire application requires recompiling and testing [8].

The advantages of a monolithic architecture include:

- Easy deployment – Deployment is simpler since there is only one executable file.

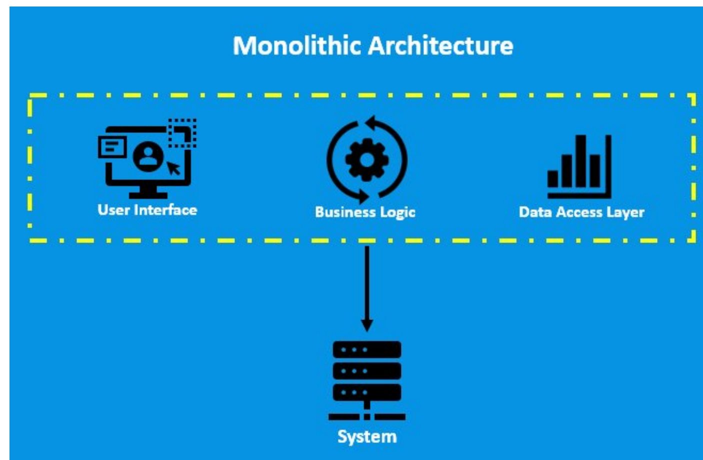


Figure 2.2: Monolithic architecture [7]

- Easy Development – The application is build with one code base so it is easier to develop.
- Performance – A single API can perform the same function of numerous APIs.
- Simplified testing – The phase of testing can be performed faster than with a distributed application.
- Easy debugging – Easier to find an issue since all code located in one place.

While, the disadvantages of a monolith include:

- Slower development speed – The development is more complex and slower.
- Scalability – Individual components cannot be scaled.
- Reliability – An error in any module could affect the entire application's availability.

- Barrier to technology adoption – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming [8].

- Lack of flexibility – The technologies used are constrained to those already present in the monolith.

- Deployment – If there is a small change to a monolithic application the entire monolith require to be redeployed [8].

In this table the two architectures are compared:

Microservices	Monolithic
Indipendent scaling of services.	Not scalable as microservices since all components int the same system.
Each service developed with different technologies.	Not flexible as microservices since all components use the same technology stack.
A single failure not affect the entire system since components are isolated.	All components are combined in a single system.
Continuous deployment.	Challenging to adopt new technology.
Network latency can impact performances.	Faster communication between components.
Testing can became more complicated as the number of services grows.	It takes longer to develop new features.
Service coordination more challenging.	Easier management and monitoring of all components.

Table 2.1: Microservices vs Monolithic advantages and disadvantages [7]

2.3 Service mesh definition

Once these concepts have been defined, we want to give a formal definition of a service mesh.
A service mesh is a dedicated infrastructure level that provides several features to microservices applications. It allows to improve application resilience, observability, and security by using sidecar proxy mechanism. In this approach, each service instance of the application is linked to a "sidecar" [9]. These proxies are components aimed at handling communications among services, manage the phase of monitoring, and deal with security issues [10].

2.4 Services and components

In this discussion it is important to give a panoramic about the various component services and functions used in the context of service mesh. So we can analyze the following:

- Container Orchestration Framework: a tool that deals with monitoring and managing the set of containers within an application. One of the most widely used frameworks is Kubernetes.
- Services and Instances: an instance is a single running copy of a microservice; in Kubernetes an instance consists of a small group of containers called pods.
- Sidecar Proxy: a sidecar proxy runs alongside a single instance or pod. It allows traffic to be routed between containers. Besides it is managed by the orchestration framework.
- Service discovery: instances that want to interact with each other perform a DNS lookup. The list of available instances are managed by the

container orchestration framework, which also provides the interface for DNS queries.

- Load balancing: load balancing is executed providing richer algorithms and more powerful traffic management.
- Encryption: request and responses are directly encrypted or decrypted by service mesh. The most common implementation for traffic encryption is mutual TLS (mTLS), in which a public key infrastructure (PKI) generates and distributes certificates and keys for use by sidecar proxies.
- Authentication and authorization: request coming from inside or outside the application can be authorized and authenticated by service mesh so that instances can receive validated requests.
- Support for the circuit breaker pattern: unhealthy instances are isolated and gradually returned to the pool of healthy instances [10].

2.5 Control plane and data plane

There are two fundamental components that work together to ensure network properties and observability to a microservice application: control plane and data plane.

The control plane is responsible for managing the configuration and controlling the overall operation of the service mesh [9].
In particular the control plane has the following features:

- Keeps track of all the services within the mesh and enables their communication.
- Manages traffic routing, load balancing, and traffic shaping across the mesh.

- Handles security and authentication, making sure that communication channels are encrypted.
- Reinforces the various policies related to traffic management, security and observability.

The data plane manages network traffic between services within the mesh network by serving sidecars proxy. Each service is associated with a sidecar proxy [9].
These are the various functions of data plane:

- Sidecar proxy intercepts the incoming and outgoing traffic of each microservice.
- Ensure efficient use of resources.
- Handles communication between the various services within the mesh applying some policies such as circuit breaking.
- Sidecar proxy collects and sends telemetry data.

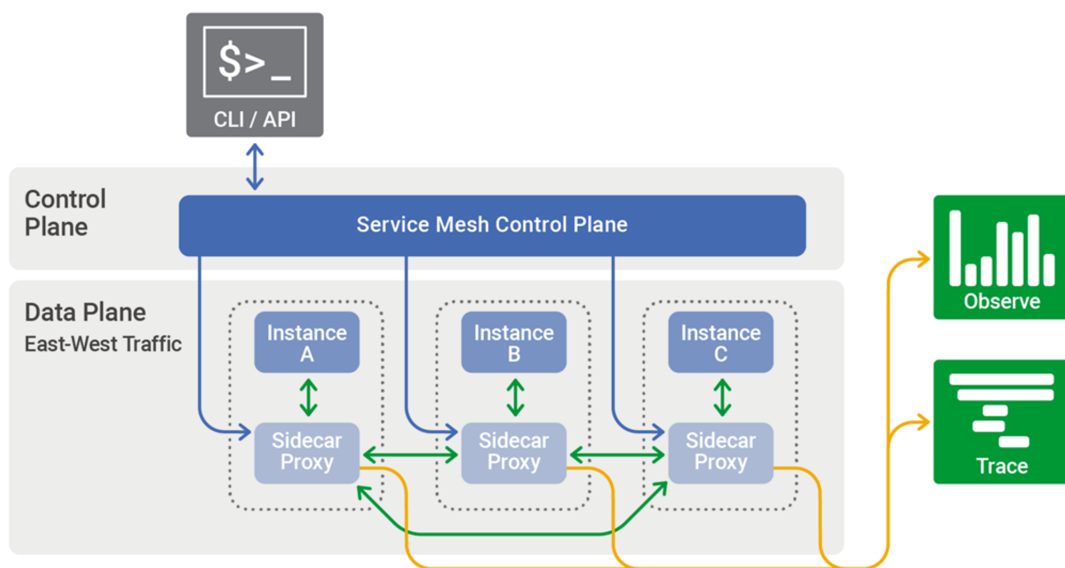


Figure 2.3: The control plane in a service mesh distributes configuration across the sidecar proxies in the data plane [10]

Chapter 3

Service mesh tools

3.1 Introduction

In the previous chapter, we defined what microservices and monolithic architectures are and provided a general overview of the concept of a service mesh: we gave a definition, described its properties, and analyzed its components.
In the following we will go on to analyze Istio and three others service mesh tools.
The first part of each section will focus on giving a general overview of the tool, while in the second we will analyse the security properties of each tool.
Finally we will compare each service mesh tool with Istio service mesh.
Let's dive into the discussion.

3.2 Istio

3.2.1 Definition

Istio is an open-source project that can seamlessly integrate with existing distributed applications, improving their functionality without significant

changes to the app code.
With Isio, users benefit of:

- Secure interservice communication with features such as TLS encryption and strong identity-based security measures.
- Built-in load balancing for different transport types, such as HTTP and gRPC.
- Advanced traffic control capabilities, including directional code and traffic offense management.
- A scalable system that can manage access, restrictions, and other quotas.
- Advanced analysis with metrics, logs, and trace tools for all cluster traffic.

Although Istio's control plane is based on Kubernetes, it is versatile enough to add applications from other clusters or those outside of Kubernetes, such as VMs. With its extensive community and partner support, users can manually install Istio or choose a vendor solution that integrates and manages Istio services for them [11].

3.2.2 Working mechanism

Istio consists of two main parts: the data plane and the control plane.
The data plane handles the actual communication between different services. Without a tool like Istio, the network is pretty much blind to the details of the data it's moving. But, with a service mesh like Istio, there's a helper called a proxy (specifically, the Envoy proxy) that checks all the data going through. This allows the network to understand and make decisions based on that data. Whenever you add a new service in your system, this proxy is automatically set up with it, whether it's in a cluster or on separate

computer systems (VMs).
The control plane, on the other hand, manages and updates these proxies. It uses the rules you provide and the information it knows about the services to keep the proxies in line with any changes [11].

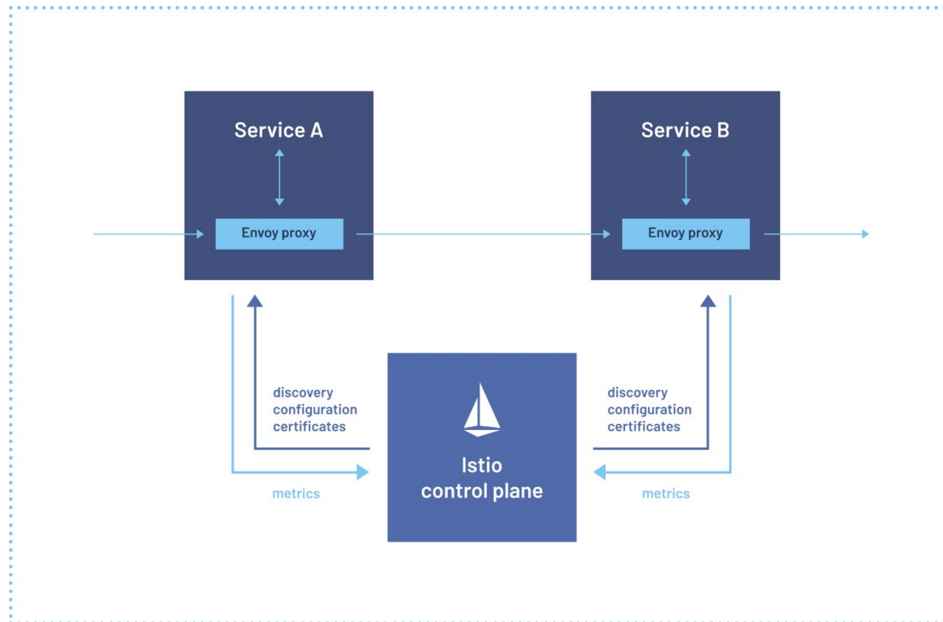


Figure 3.1: Sidecar proxy along with every service deployed in Istio mesh [11]

3.2.3 Components

Envoy

Istio uses the Envoy proxy, a high-efficiency tool made in C++, to handle all incoming and outgoing traffic within its service mesh. In Istio, only Envoy proxies deal with this traffic. These Envoy proxies are attached to services, enhancing them with numerous advanced features.
Some of these are:

- Finding services dynamically

- Balancing loads
- Ending TLS sessions
- Monitoring health and handling traffic disruptions

The proxy ensures that Istio can make and enforce rules while also gathering detailed data about the network's activity. Besides, this approach allows to add Istio's features to a system without changing the original code. With the help of Envoy proxies, Istio can direct traffic with detailed rules, ensure network stability with measures like retries and failovers, boost security and control access, and add custom features through a flexible extension system based on WebAssembly [12].

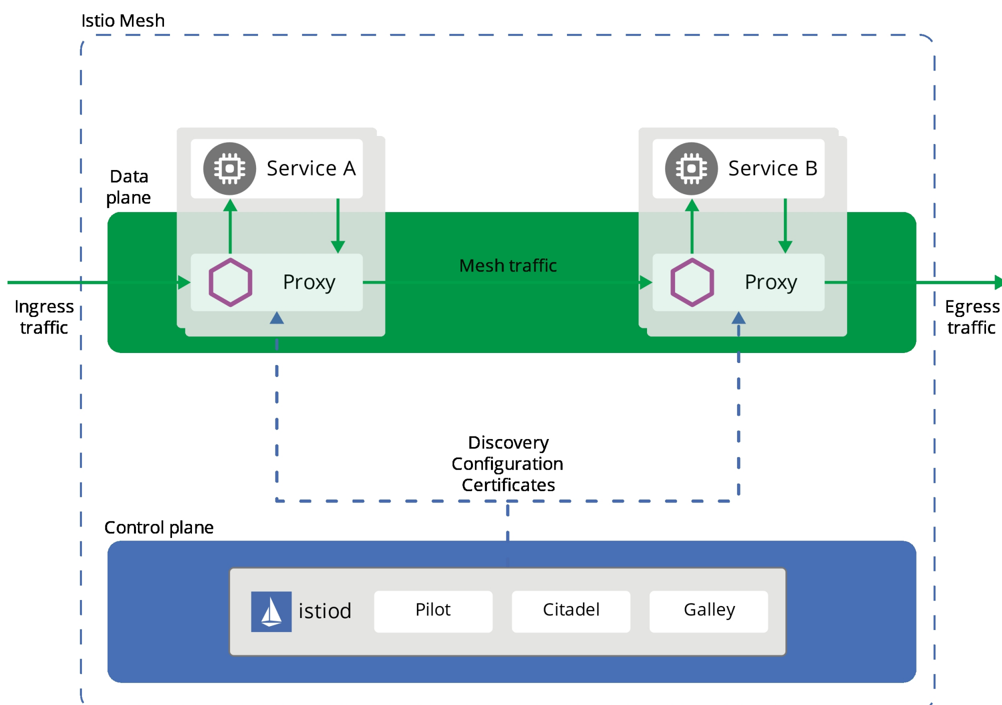


Figure 3.2: Istio components architecture [12]

Istiod

Istiod is responsible for discovering services, managing configurations, and handling certificates. It takes high-level traffic rules and translates them into specific settings for Envoy proxies, sending these settings to the sidecars when needed. While Istio can work with different environments like Kubernetes or VMs, Istiod makes sure all sidecars understand these settings in a consistent manner. Through Istio's Traffic Management API, users can guide Istiod to fine-tune how traffic flows within the service mesh. For security, Istiod ensures safe communication between services and users. Istio can enhance unsecure traffic within the mesh and make policy decisions based on service identities instead of less reliable network indicators. With Istio, there's also control over who can reach the services. Lastly, Istiod functions as a Certificate Authority, creating certificates to ensure encrypted and secure communication within the network [12].

3.2.4 Security features

Identity and certificate management

Istio uses a mechanism to understand if a message comes from a person, a single service, or a group of services. For platform that don't have a built-in way of identifying services, it can use things like the service name.
Besides, it gives every workload a special ID using X.509 certificates. There are Istio agents by each Envoy proxy that help manage these certificates and keep them updated [13].

Authentication

Istio provides two kind of service authentication: peer authentication and request authentication.

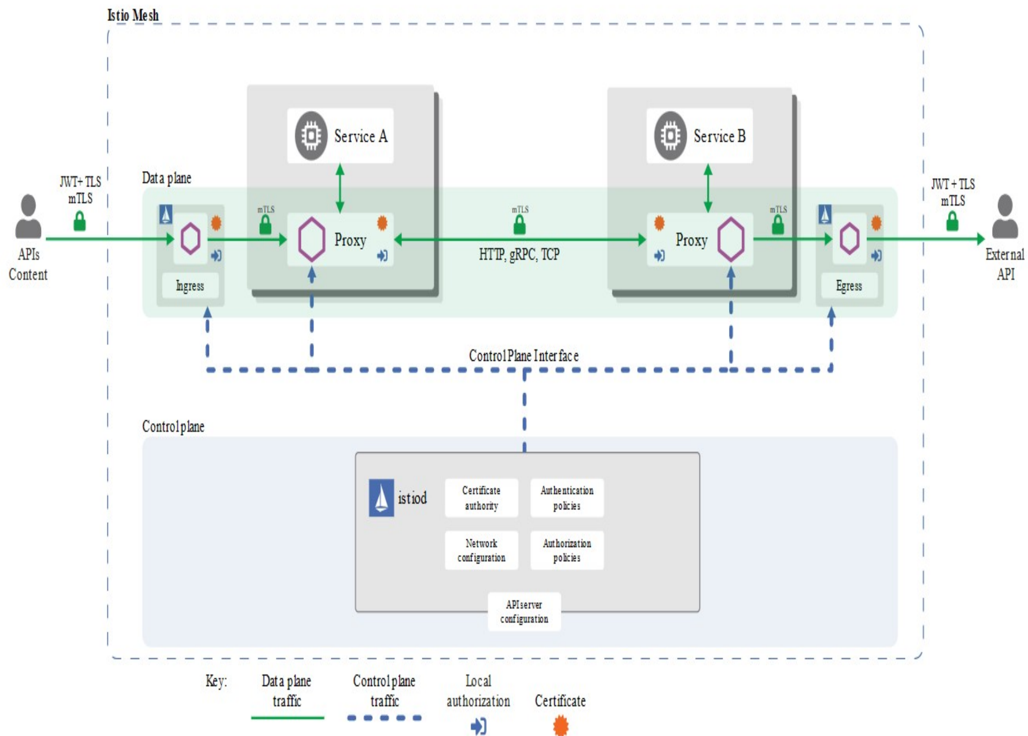


Figure 3.3: Istio security architecture [13]

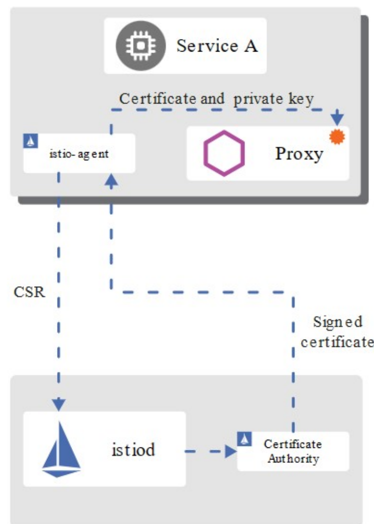


Figure 3.4: Istio identity provisioning [13]

- Peer authentication: Istio uses mTLS to provide service to service authenticatication. With this mechanism, every service gets a special ID so they can talk to each other, the communication between services is private, and "keys" or "passwords" are managed directly by Istio.
  For operators who need to migrate to Istio, "permissive" mode can be enabled, that allows services to accept both mTLS and plaintext traffic [13].

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-policy"
  namespace: "foo"
spec:
  mtls:
    mode: STRICT
```

Figure 3.5: Peer authentication policy with "STRICT" mode enabled [13]

- Request authentication: it is used to evaluate credential attached to the request. JSON Web Token and custom authentication provider such as ORY Hydra are used for enabling this authentication.
  Istio uses request authentication policies to check if JSON Web Token (JWT) is a valid token [13].

```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: ingress-jwt
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.19/security/tools/jwt/samples/jwks.json"
```

Figure 3.6: Request authentication policy [13]

All the authentication policies are stored in "Istio config store".

Authorization

Istio uses authorization policies which can be configured to allow or deny the service to access to a specific workload. Each Envoy proxy manages the authorization request according to the policies provided.
For instance, fig. 3.7 shows an authorization policy that denies requests if the source is not the foo namespace [13].

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: httpbin-deny
 namespace: foo
spec:
 selector:
   matchLabels:
     app: httpbin
     version: v1
 action: DENY
 rules:
 - from:
   - source:
       notNamespaces: ["foo"]
```

Figure 3.7: Example of authorization policy [13]

3.3 AWS app mesh

3.3.1 Overview

AWS App Mesh is a service mesh that facilitates monitoring and control of services. It standardizes the way your services communicate with each other, giving you end-to-end visibility and high availability for your applications. App Mesh gives you consistent visibility and control of network traffic for each service in an application.

The components of AWS app mesh are the following:

- Service mesh – A service mesh manages the network traffic between the services within it.

- Virtual services – A virtual service abstracts the actual service provided by a virtual node directly or indirectly through a virtual router.

- Virtual nodes – A virtual node acts as a logical pointer to a discoverable service, such as an Amazon ECS or Kubernetes service. For each virtual service, you will have at least one virtual node [14].

- Virtual routers and routes – Virtual routers handle traffic for one or more virtual services within the mesh. A virtual router has a route associated with it. The route is used to match requests for the virtual router and to distribute traffic to its associated virtual nodes.

- Proxy – After the mesh creation, proxies can be associated to services. The proxy manages the traffic in an appropriate way [14].

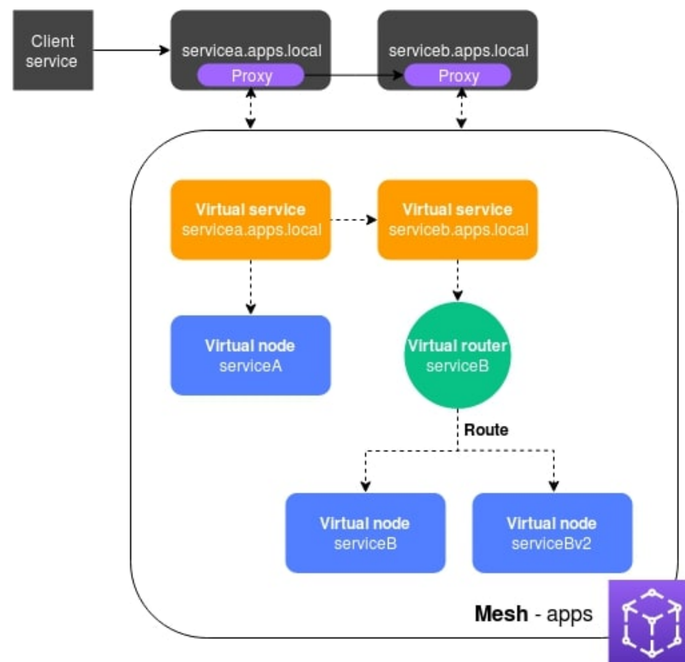


Figure 3.8: AWS service mesh components [14]

3.3.2 Security features

Now we want to describe the principal security features provided by AWS service mesh:

- Encryption: AWS App Mesh uses the Transport Layer Security (TLS) protocol to encrypt network traffic between services in the mesh, ensuring that sensitive data is protected from eavesdropping attacks;optionally is possible to have MTLS authentication [15].
- Authorization: AWS App Mesh supports role-based authorization, allowing teams to define and manage permissions for access to services in the mesh. In addition, AWS App Mesh supports integration with AWS Identity and Access Management (IAM) to provide granular access control [16].
- Credential management: AWS App Mesh enables centralized management of credentials for accessing services in the mesh, reducing the risk of exposure of sensitive credentials [17].
- Protection against DDoS attacks: AWS App Mesh includes protection features against DDoS attacks, such as AWS Shield and AWS WAF, to ensure the availability of services in the mesh [18].
- Integration with AWS CloudTrail: AWS App Mesh is integrated with AWS CloudTrail to provide comprehensive auditing of activity in the mesh, enabling teams to track user activity and identify security breaches [19].

3.3.3 Istio vs AWS service mesh

In this subsection we provide a comparison between Istio and AWS service mesh.

Istio	AWS service mesh
Open source platform.	Based on AWS platform.
Can be executed on any infrastructure.	Specifically suitable for AWS environment.
IT teams have to take care of installation, configuration, updating, and troubleshooting.	All the activities are simplified since are managed by AWS underlying environment.
More flexible.	Less flexible in terms of advanced customisation.
Cost depends on the infrastructure and personnel required for management.	Cost based on the consumption of resources and network requests made.

Table 3.1: Istio vs AWS service mesh

3.4 Linkerd

3.4.1 Overview

Linkerd is a service mesh for Kubernetes. It makes running services easier and safer by giving runtime debugging, observability, reliability, and security. Like others service mesh tools, Linkerd has two basic components: a control plane and a data plane. Linkerd works by installing a set of ultralight, transparent "micro-proxies" next to each service instance. These proxies automatically handle all traffic to and from the service [20].

3.4.2 Security features

In this section we analyze security features of Linkerd service mesh:

- Authentication and authorization: Linkerd uses the mTLS protocol

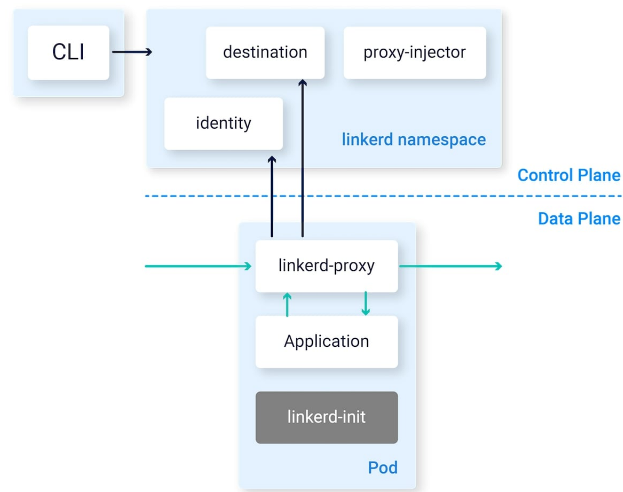


Figure 3.9: Linkerd's architecture [21]

to authenticate and authorize communications between microservices. This means that each microservice must authenticate itself and present a valid certificate before communicating with other microservices [22].

- Certificate management: Linkerd has a main control system with a certificate-making part named "identity." This part creates security certificates for each Linkerd proxy. These certificates are tied to the identity of the pods they are in and last only a day, being replaced automatically. Each proxy uses the certificates to safely communicate with one another [22].

- Access control: Linkerd allows to set authorization rules to control access to microservices ensuring that only authorized applications can access the microservices [23].

3.4.3 Istio vs Linkerd

In table 3.3 we can appreciate the differences between Istio and Linkerd.

Istio	Linkerd
It uses sidecar proxy for each service.	It uses Linkerd proxy written in Rust.
More complexity and higher learning curve.	It focuses on ease of use and user experience.
More overhead.	High performance and low overhead.
Wider community and more developed ecosystem.	Active community.
Most comprehensive set of advanced features.	Essential functionalities based on ease of use.
Broader interoperability due to its support for the Envoy application network interface (API).	Integration with other instruments and technologies.

Table 3.2: Istio vs Linkderd

3.5 Kong mesh

3.5.1 Overview

Kong Mesh is a service mesh built on top of the Kong Gateway API platform. It provides a set of features that help to manage and secure microservices running in a Kubernetes cluster or any other cloud environment. The features provided are traffic routing, load balancing, service discovery, observability, security, and policy enforcement [24].

3.5.2 Security features

Security features provided by Kong mesh are the following:

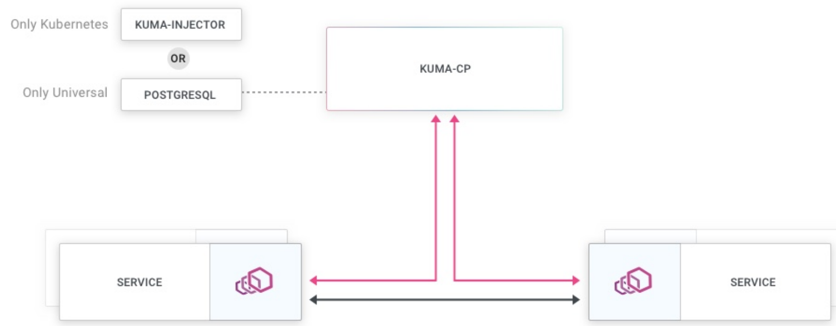


Figure 3.10: Kong's architecture [25]

- Mutual TLS (mTLS): Kong Mesh uses mutual TLS to provide secure communication between services. It ensures that only authenticated and authorized requests are allowed to access the service, and it encrypts the data transmitted between services [26].
- Certificate management: Kong Mesh gives an identity to every proxy within the data plane. It is flexible in terms of Certificate Authority (CA) backends and handles certificate renewals automatically.
  Kong Mesh provides two main CA backends: builtin and provided. With the first, CA root certificate and the corresponding keys are created and stored as secret. While, with provided CA backend, users give their own CA root certificate and key [26].
- Access Control Lists (ACLs): Kong Mesh allows administrators to define access control lists to restrict access to services based on IP addresses, HTTP methods, or other criteria [27].

3.5.3 Istio vs Kong service mesh

Istio	Kong mesh
It uses sidecar proxy for each service.	It integrates service mesh functionalities based on Envoy and includes Kong's API gateway functionality.
It provides granular control over traffic routing, advanced load balancing rules and flexible configuration options.	It offers traffic routing, load balancing, and advanced request control using configurable plugins and rules.
It supports service authentication, role-based authorisation (RBAC), data-in-transit encryption and other advanced security features.	It provides authentication, authorisation, access control and credential management.
Very active developers community.	It benefits the Kong community.
It offers broad compatibility with the Kubernetes ecosystem and has tight integration with monitoring and tracking services.	It offers integrations with monitoring tools and take advantage of Kong's extensions and plugins.

Table 3.3: Istio vs Kong mesh

Chapter 4

Multi-cloud cluster design

4.1 Introduction

In this chapter we will study how to create a multicloud cluster using Istio service mesh.
I will first go on to compare the concepts of multicloud and hybrid cloud.
Next I will discuss Azure Kubernetes Service and Google Cloud platform and list the steps followed to create clusters in these cloud providers.
Then I will discuss Istio's various solutions for creating the multicluster and will explain why I chose one of these.
Finally I will discuss how I deployed a microservices app in the cluster multi-cloud.

4.2 Multi-cloud environment

4.2.1 Definition

Multi-cloud is a strategy that is used by organizations in order to use multiple cloud services on which to deploy their applications. A multi-cloud environment may involve the use of two or more public clouds, two or more

private clouds, or a combination of both. The main goal is to provide management flexibility for the organization's various workloads [28].

4.2.2 Multi-cloud vs Hybrid-cloud

The concepts of multi-cloud and hybrid cloud differ in the type of cloud infrastructure. Speaking of multi-cloud, we mean the use of cloud services from different public cloud providers for different workloads.
Conversely, if we talk about hybrid cloud, we indicate the use of multiple computing environments, such as public cloud environments and private cloud environments, for common workloads.
In order to better understand the differences of the two concepts, this analogy can be used. The hybrid cloud can be associated with a hybrid car, which combines two different types of engine, an electric motor and a traditional combustion engine, to power the car.
While a multi-cloud infrastructure could be associated with using different types of transportation to get to different places. For example, one could drive to the mall because it is easier to take shopping bags home, but one could choose the train to save gasoline and avoid rush-hour traffic [28].

4.2.3 Benefits and Challenges

In the introduction to multi-cloud, we stated that flexibility is one of the main reasons why we choose this strategy. Now we are going to analyze in more detail what benefits lead to choosing this approach:

- Taking the best of each cloud: it allows you to choose among various cloud providers the one that best suits your needs such as speed, performance, reliability and others.

- No vendor lock-in: you are not tied to a single provider allowing you to reduce data, interoperability and cost issues.

- Cost efficiency: you can significantly reduce costs by taking advantage of the best combination of different providers.
- Innovative technology: you can take advantage of new technologies offered by various cloud providers as they invest heavily in the development of new products.
- Advanced security and regulatory compliance: regardless of the service, provider or environment, consistent security policies and compliance technologies are guaranteed across all workloads.
- Increased reliability and redundancy: in case one cloud stops working this does not affect the services of other clouds and computing needs can be handled by another cloud [28].

Having listed the benefits that a multi-cloud architecture can bring to an organization, we want to list the difficulties that such an implementation can generate. First, we can corroborate the difficulties of managing the environment since, having to manage various cloud providers, it is important to understand how the organization's requirements can be met. In addition, other thorny nodes are maintaining consistent security, integration of software environments, and difficulties in achieving consistent performance and reliability across clouds.

4.3 Kubernetes cluster

4.3.1 Definition

A Kubernetes cluster is a set of nodes running containerized applications managed by Kubernetes, an open-source system for automating, scaling, and managing containerized applications. The cluster is managed by the control plane. In particular, it deals with Nodes and the Pods. Nodes are machines where containers run. Each node communicate with the control

plane and contains the services necessary to run Pods. Pods are the smallest deployable units and hold one or more containers. Containers within a pod share the same network namespace, enabling them to communicate using localhost. Namespace divides a cluster into multiple virtual clusters, useful when multiple users or teams share a cluster [29].
Among the various cloud services that enable the creation of a kubernetes cluster are AKS (Azure kubernetes service) and GCP (Google cloud platform).

AKS cluster

AKS is a hosted Kubernetes service that helps to deploy in a simple way a Kubernetes cluster. It handles tasks like monitoring and maintenance. Along with the creation of AKS cluster, there is the creation and configuration of the control plane [30].

GCP cluster

GCP is a managed Kubernetes service used to deploy and operate containerized applications using Google's infrastructure. It allows to configure the infrastructure environment such as networking, scaling, hardware, and security. It provides control plane and nodes to manage the various components. The environment of GCP is composed by nodes that are grouped together to form a cluster. Apps are packaged into containers that are deployed as Pods in nodes. To interact with the workloads Kubernetes API are used [31].

4.4 Istio deployment models

Within an Istio mesh, the control plane orchestrates communication between workloads. These workloads fetch their settings from the control plane.
In its most basic form, a single control plane can manage a single cluster,

which is termed a primary cluster. However, when we look at multicluster setups, they can either have their own control planes or share them. Clusters with their own control planes are called primary clusters, while those without are termed remote clusters [32].
Istio provides four possibilities for install a mesh across multiple Kubernetes cluster[33]:

1. Multi-primary
2. Multi-primary on different networks
3. Primary-Remote
4. Primary-Remote on different networks

I decided to distribute multiple control planes across different clusters.This setup provides resilience to the application [32].
In particular, we have the following benefits:

- Failure of one control plane affects only the clusters it manages.
- Changes in one region or cluster don't impact others.
- Configurations can be introduced more carefully, possibly one cluster at a time.
- Service access can be limited, allowing for better control. For instance, a service available in Cluster A might be kept unavailable in Cluster B.

4.5 Environment setup

At the beginning, two clusters were created: one on "Azure Kubernetes Service" (AKS) as shown in Figure 4.1 and another on "Google Cloud Platform" (GCP) as shown in Figure 4.2.

Figure 4.1: AKS cluster after creation

Figure 4.2: GCP cluster after creation

After that, Istio service mesh was installed with the "multi-primary on different network" deployment model that allows one control plane on AKS

and another one on GCP. Once Istio has been installed, a namespace was created for each cluster and a proxy sidecar enabled. Then, an e-commerce application consisting of 11 microservices was deployed. Searching on github, I found an application that fits the bill: Online boutique.
It is an e-commerce application composed by 11 microservices where users can browse items, add them to the cart, and purchase them [34].
In fig.4.3 we can see how the application is organized. All services talk witch each other over gRPC.
For demonstration purposes, I decided to divide the microservices between

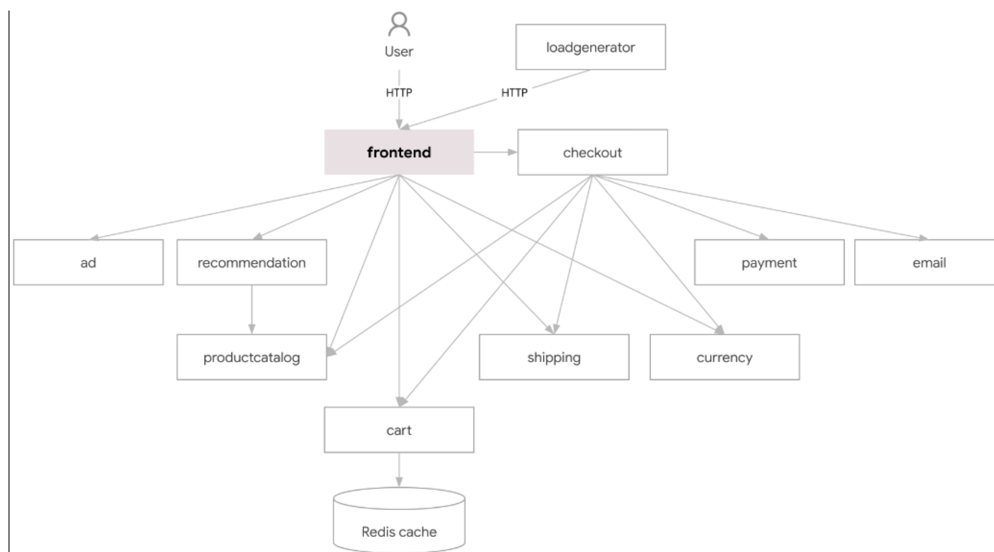


Figure 4.3: Online boutique architecture [34]

the clusters of AKS and GCP. In the app folder kubernetes manifests of services are provided.
In AKS, I applied manifests for the following services:

- frontend
- shippingservice
- productcatalogservice
- paymentservice

- emailservice
- currencyservice

Whereas in GCP for the following services:

- adservice
- cartservice
- checkoutservice
- recommendationservice

Before applying the manifests, I modified the "frontend," "checkout," and "productcatalog" manifests to allow all services to communicate properly. For "frontend" and "productcatalog", which are on the AKS cluster, I replaced the names of called services with the respective addresses of services on the GCP cluster; conversely, for "checkout," present on GCP, I replaced the names of called services with the respective addresses of services on the AKS cluster. Finally, I applied the service manifests by dividing them into clusters as seen above.
After the following steps, we can see 4.4 what the main page of the application looks like.

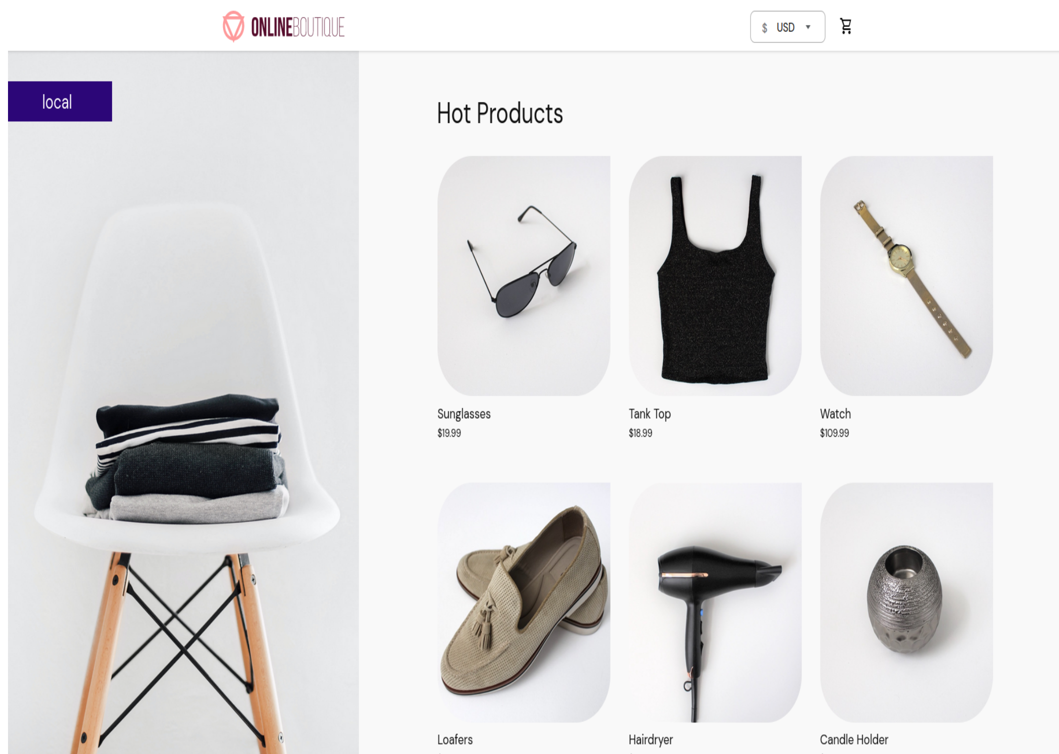


Figure 4.4: Home page Online boutique

Chapter 5

Environment monitoring and testing

5.1 Intoduction

In the last chapter, I showed all the steps followed to create a multi-cloud cluster with Istio on AKS and GCP and to deploy a microservice application. In this, some tools to monitor the whole cluster will be used. In particular we will see tools such as Prometheus, Kiali, and Grafana. Finally, some security tests done on the environment will be shown.

5.2 Environment monitoring

5.2.1 Prometheus

Prometheus is an open-source monitoring and alerting toolkit. It monitors by collecting time series data: metrics recorded with timestamps and associated key-value labels [35].
Prometheus comes with several components:

- The primary server that scrapes and stores data.

- Client libraries to integrate into applications.
- A push gateway for ephemeral jobs.
- Specific exporters for various services.
- An alert manager.
- Several auxiliary tools.

.

5.2.2 Configuration

To monitor our multicloud cluster on which Istio is installed, we decided to install the prometheus instance on both clusters that gather data and then consolidate this information to a production mesh-wide Prometheus instance. In our case, we applied this mesh-wide Prometheus instance in AKS cluster [36].

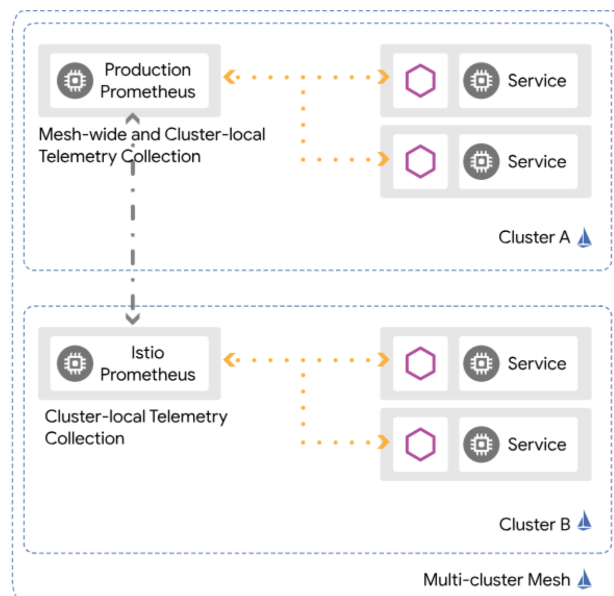


Figure 5.1: In-mesh Production Prometheus for monitoring multicluster Istio [36]

For this installation we followed the following steps:

- Applied the prometheus deployment in each cluster with this command:

```
kubectl apply
https://raw.githubusercontent.com/istio/istio
/release-1.19/samples/addons/prometheus.yaml
```

- Edited prometheus configuration present on AKS cluster with this:

```
KUBE_EDITOR="nano"
kubectl -n istio-system edit
cm prometheus -o yaml
```

  We configure the file as in Figure 5.2. For the remote cluster, we decided "gcp-cluster" so we provided also the address of Istio ingress gateway on GCP cluster. While, for the local cluster we used AKS.

5.2.3 Kiali

Kiali is a monitoring tool for Istio that offers insights into the configuration and health of a service mesh. By analyzing traffic patterns, it provides a clear view of the mesh's topology and highlights any issues [37].

Because I decided to install production mesh-wide Prometheus istance on AKS, I also applied the Kiali instance on this cluster.
We used a simple installation provided by Istio with the following command:

```
kubectl apply -f https://raw.githubusercontent.com/
istio/istio/release-1.19/samples/addons/kiali.yaml
```

```
scrape_configs:
- job_name: 'federate-{{gcp-cluster}}'
  scrape_interval: 15s

  honor_labels: true
  metrics_path: '/federate'

  params:
    'match[]':
      - '{job="kubernetes-pods"}'

  static_configs:
    - targets:
      - 'prometheus.{{34.74.68.96}}'
      labels:
        cluster: '{{gcp-cluster}}'

- job_name: 'federate-local'

  honor_labels: true
  metrics_path: '/federate'

  metric_relabel_configs:
  - replacement: '{{aks-cluster}}'
    target_label: cluster

  kubernetes_sd_configs:
  - role: pod
    namespaces:
      names: ['istio-system']
  params:
    'match[]':
    - '{__name__=~"istio_(.*)"}'
    - '{__name__=~"pilot(.*)"}'
```

Figure 5.2: Prometheus federation configuration

Microservices app on Kiali

In the Figure 5.3 we can appreciate the graph of our microservice application. We can see that on the left is the AKS cluster with its services while on the right is the GCP cluster.
Taking a closer look at the graph, we can appreciate the locks between the various services indicating that they communicate using mTLS.

5.2.4 Grafana

Grafana is a monitoring tool that allows to set up dashboards for Istio. It lets you track the well-being of Istio as well as the applications operating within its service mesh [38].
As in the case of Kiali, we applied Grafana to AKS with the following

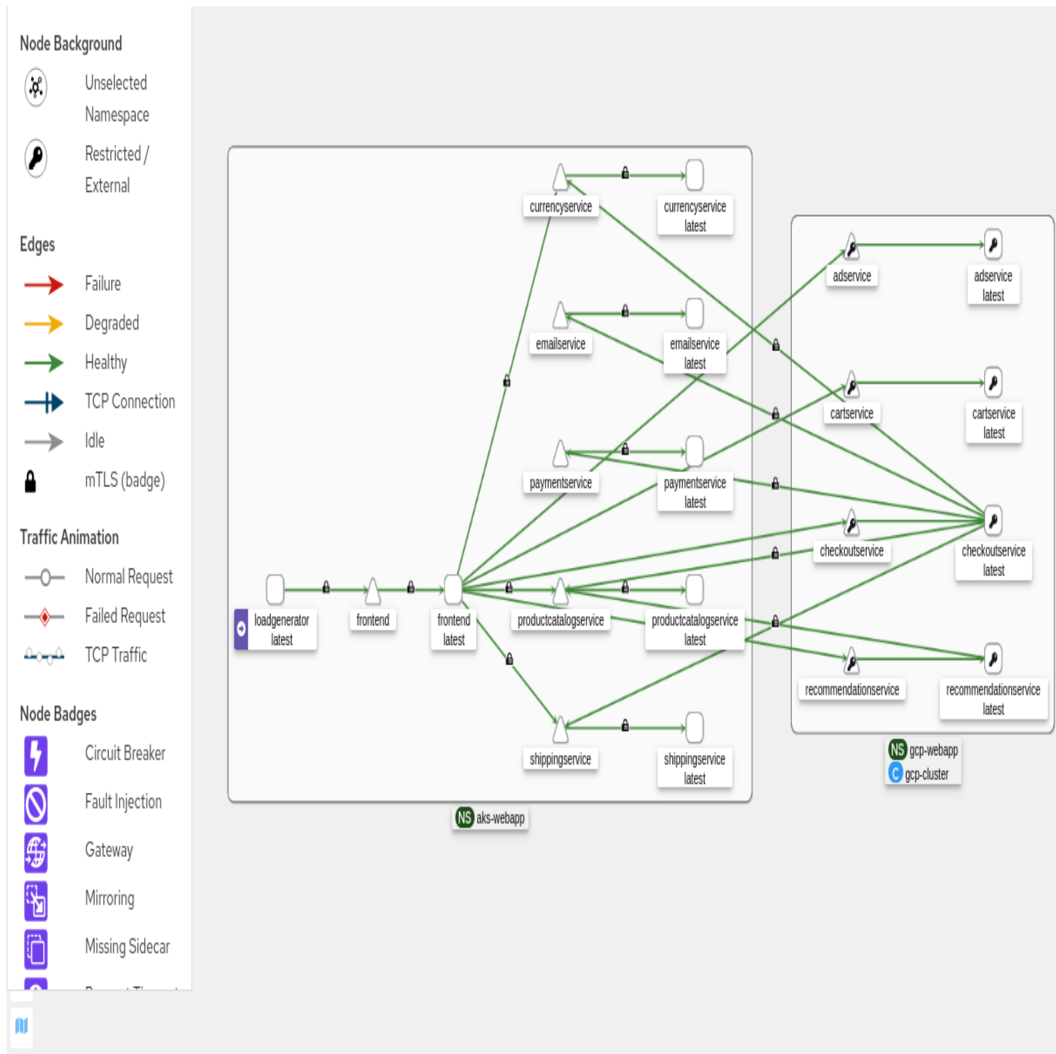


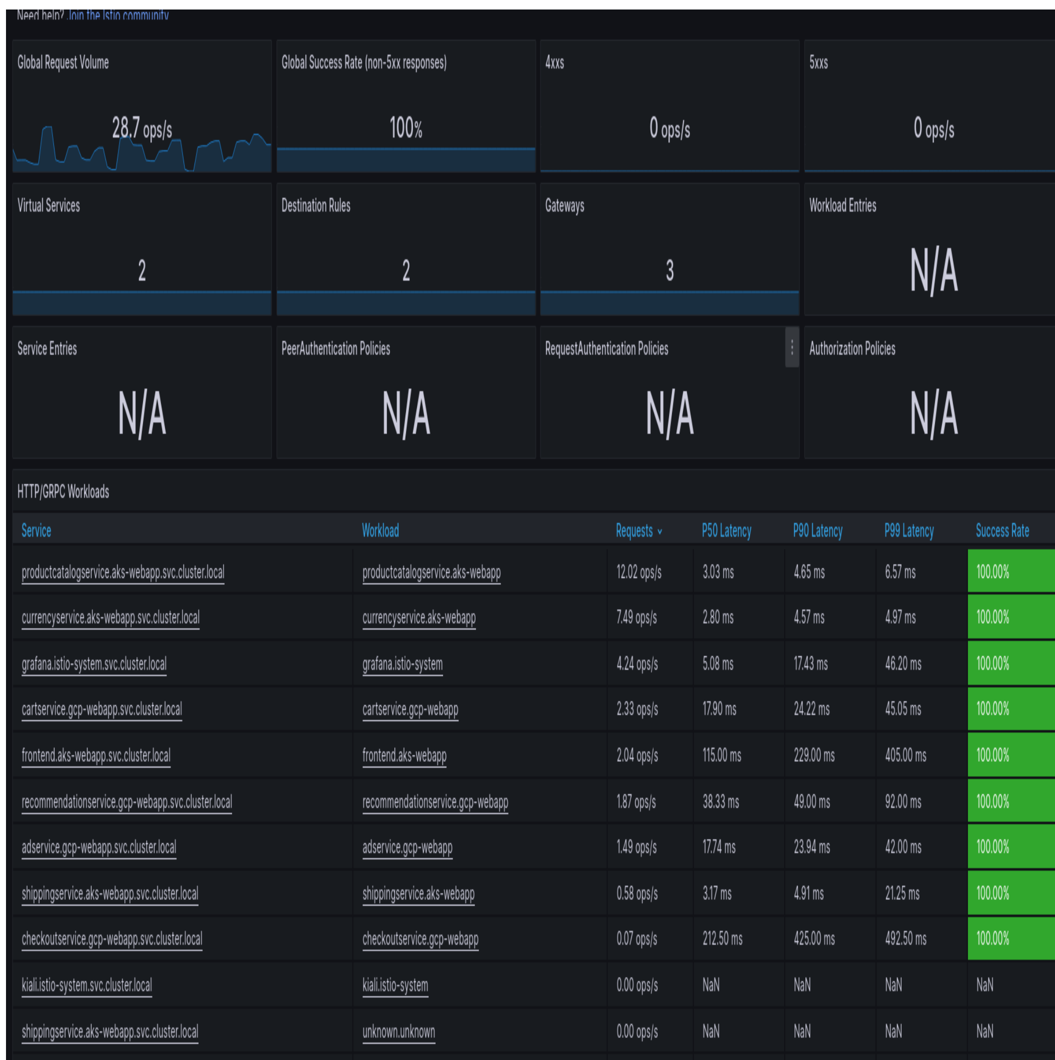
Figure 5.3: Graph of the micorservices app on Kiali

command:

```
kubectl apply -f https://raw.githubusercontent.com/
istio/istio/release-1.19/samples/addons/grafana.yaml
```

Microservices app on Grafana

Once you have logged into Grafana you can access the Istio dashboards. For me, was interesting to keep an eye on the mesh dashboard Figure 5.4, which provides a number of useful pointers for all the services of the mesh.
We can note that for each service there is the number of requests per second, latency, and success rate.



Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
productcatalogservice.aks-webapp.svc.cluster.local	productcatalogservice.aks-webapp	12.02 ops/s	3.03 ms	4.65 ms	6.57 ms	100.00%
currencyservice.aks-webapp.svc.cluster.local	currencyservice.aks-webapp	7.49 ops/s	2.80 ms	4.57 ms	4.97 ms	100.00%
grafana.istio-system.svc.cluster.local	grafana.istio-system	4.24 ops/s	5.08 ms	17.43 ms	46.20 ms	100.00%
cartservice.gcp-webapp.svc.cluster.local	cartservice.gcp-webapp	2.33 ops/s	17.90 ms	24.22 ms	45.05 ms	100.00%
frontend.aks-webapp.svc.cluster.local	frontend.aks-webapp	2.04 ops/s	115.00 ms	229.00 ms	405.00 ms	100.00%
recommendationservice.gcp-webapp.svc.cluster.local	recommendationservice.gcp-webapp	1.87 ops/s	38.33 ms	49.00 ms	92.00 ms	100.00%
adservice.gcp-webapp.svc.cluster.local	adservice.gcp-webapp	1.49 ops/s	17.74 ms	23.94 ms	42.00 ms	100.00%
shippingservice.aks-webapp.svc.cluster.local	shippingservice.aks-webapp	0.58 ops/s	3.17 ms	4.91 ms	21.25 ms	100.00%
checkoutservice.gcp-webapp.svc.cluster.local	checkoutservice.gcp-webapp	0.07 ops/s	212.50 ms	425.00 ms	492.50 ms	100.00%
kiali.istio-system.svc.cluster.local	kiali.istio-system	0.00 ops/s	NaN	NaN	NaN	NaN
shippingservice.aks-webapp.svc.cluster.local	unknown.unknown	0.00 ops/s	NaN	NaN	NaN	NaN

Figure 5.4: Istio mesh dashboard

I get more detailed information about a single service from the services

dashboard as shown in Figure 5.5 In this case, we can see the frontend service. Moreover, with grafana and prometheus I was able to to create new

Figure 5.5: Frontend service on Istio service dashboard

metrics. In this regard, I thought of creating ad-hoc metrics for our application.

In Figure 5.6 all the metrics included in the new dashboard can be seen.

Figure 5.6: Microservice app dashboard on Grafana

Mtls connections vs non mTLS connections

To verify that the entire application communicates using the mTLS protocol, I created several metrics that monitor the number of communications per second that use this security protocol versus those that use unknown protocols.
The graph in Figure 5.7, represents types connections per second between the GCP cluster and the AKS cluster. In particular, the green line represents

the number of mTLS connections while the yellow line should represents the number of requests that does not use mTLS. In this case the number of first connections fluctuate and are zero only at a brief juncture while unknown connection is equal to zero.

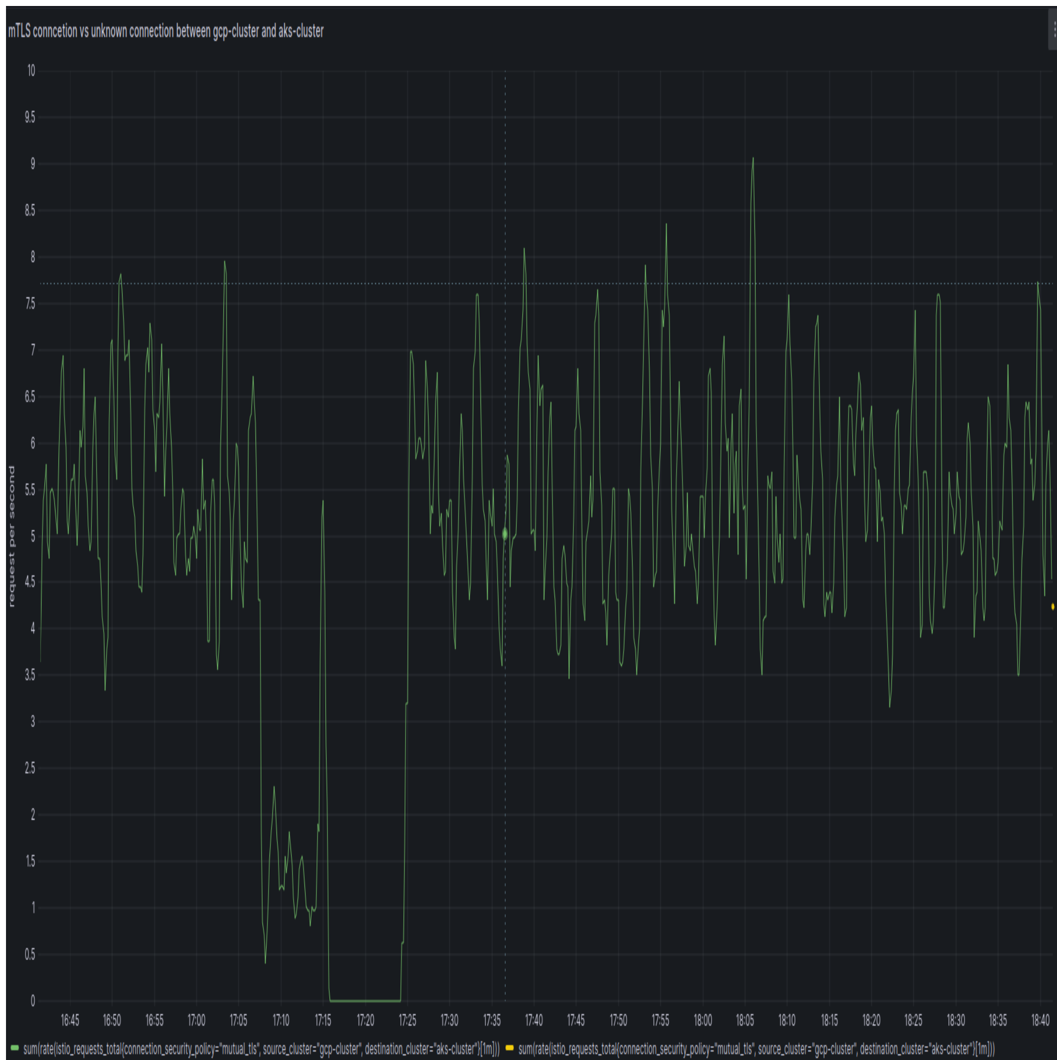


Figure 5.7: mTLS vs non mTLS connections from my-gcp cluster to my-aks-cluster

In the other hand, the graph in Figure 5.8 , deals with types connections per second between the AKS cluster and the GCP cluster. Again, the green

line represents the number of mTLS connections while the red line represents the number of unknown connections. In contrast to the previous case, we can observe that both types of connections are present in a similar way.

Figure 5.8: mTLS vs non mTLS connections from my-aks cluster to my-gcp-cluster

Certificate expirations

I created another metric that monitors how many hours remain until the expiration of certificates for all services in the mesh.
We expect that Istio rotates the certificates and assigns them to services frequently. In fact, in this Figure 5.9 we can see that there are less than 24 hours until the certificates of all services in the cluster expire.

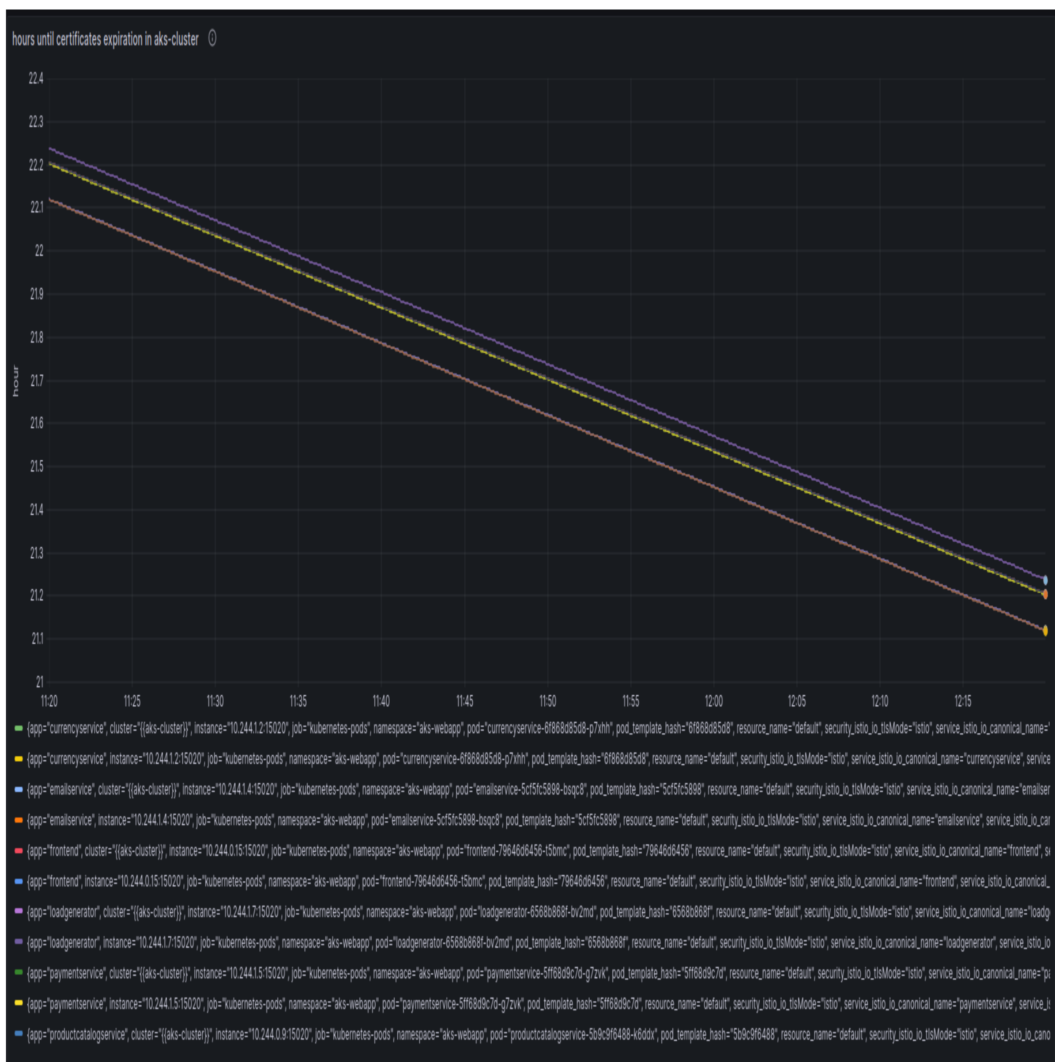


Figure 5.9: Hours remaining to the expiration of certificates in AKS cluster

5.3 Environment testing

5.3.1 Istiod traffic analysis

In the first test, I performed traffic analysis with "Wireshark" to evaluate how communications between the Istio controller and the other services take place. Considering the time frame ranging from 0 to 1.06 seconds of the Figure 5.10, we can observe that istio controller, which has the address "10.48.0.1", exchanges data with Istio egress gateway, which has the address "10.44.3.5", using tls protocol. We can see the same behaviour between Istio egress gateway ("10.44.3.5") and the service with address "10.44.1.10", in the time frame from 1,507 to 1,508 seconds.

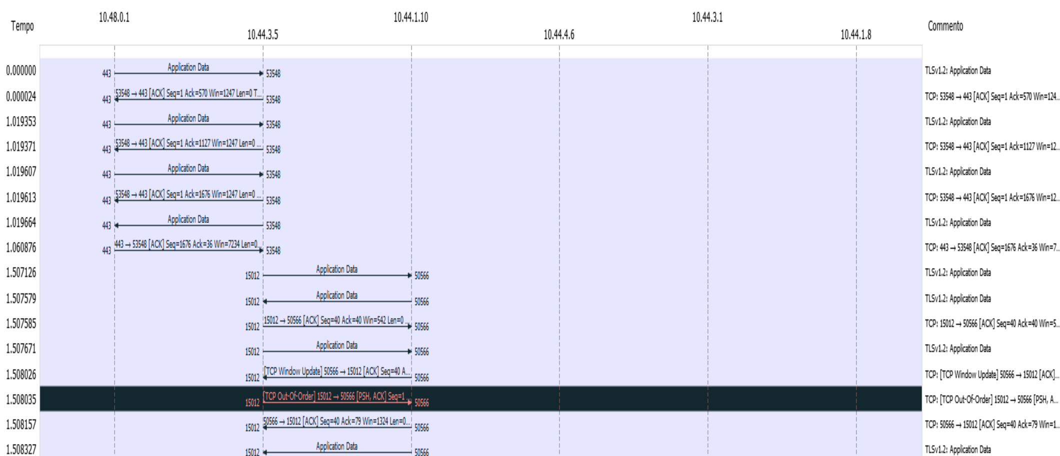


Figure 5.10: Analysis of packets exchanged by the Istio controller with other services

5.3.2 Dos attack

In this test, I attempted to do two DoS attacks on the Istio controller of GCP cluster. To do this, I installed three dedicated pods from which I launched the attack to "istiod". The graph in the Figure 5.11 shows the input and output traffic of all containers in the GCP cluster. The top part shows the

input traffic while the bottom part the output traffic. In the left part of the graph, we can see the first attack. In this case, the load of the incoming connections starts from 5 Mb/s, after a period of time it reaches around 10 MB/s, and slowly drops to zero. In the middle part of the graph we can observe the second attack. In this other case, the traffic starts from 0 MB/s, rises to 7.5 MB/s, and then drops to zero.

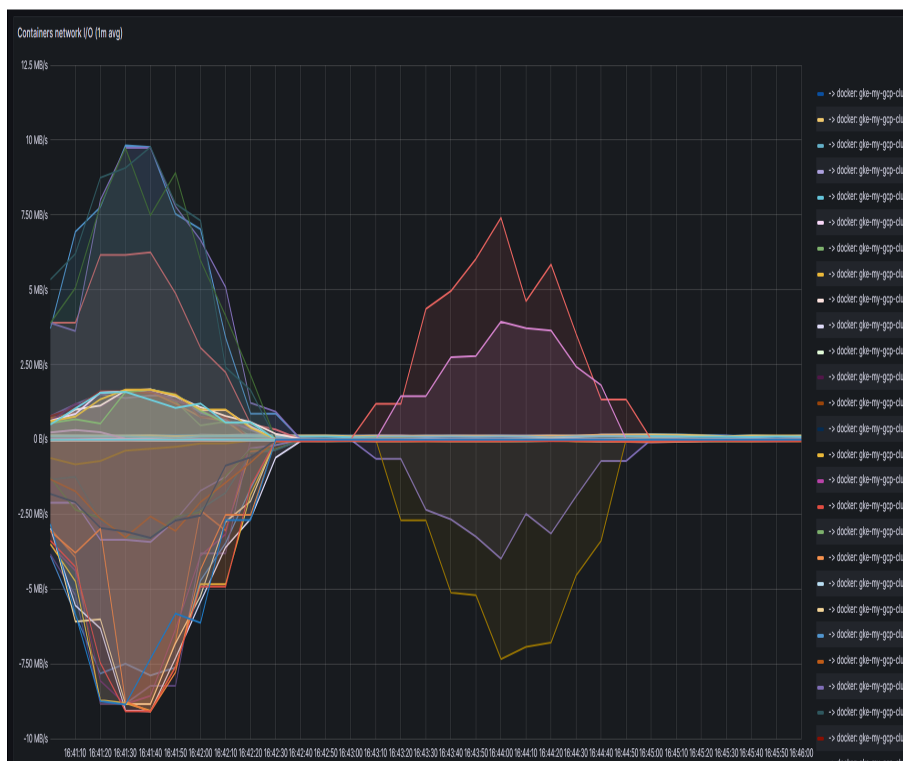


Figure 5.11: DOS on istiod

5.3.3 Nmap test

In the third test, I used nmap on the ingress gateway and egress gateway of Istio across both clusters. By running the following command "nmap -p- -vv –script firewall-bypass -o ingess.nmap", I identified two open ports on eastwest as shown in figure 5.12 and two open ports on the ingress gateway as seen in Figure 5.13.

```
Host is up, received echo-reply ttl 108 (0.12s latency).
Scanned at 2023-09-06 08:52:44 EDT for 202s
Not shown: 65533 filtered tcp ports (no-response)
PORT      STATE SERVICE REASON
443/tcp   open  https   syn-ack ttl 42
15021/tcp open  unknown syn-ack ttl 42
```

Figure 5.12: nmap on ingress gateway

```
Host is up, received echo-reply ttl 109 (0.12s latency).
Scanned at 2023-09-06 08:59:53 EDT for 5469s
Not shown: 65533 filtered tcp ports (no-response)
PORT      STATE SERVICE REASON
15021/tcp open  unknown syn-ack ttl 42
15443/tcp open  unknown syn-ack ttl 43
```

Figure 5.13: nmap on egress gateway

Chapter 6

Conclusions

In this work, we demonstrated how Istio service mesh constitute a good solution in a multicloud environment. It proves to be useful for mTLS communication between services by taking care of certificate management even though it does not guarantee encryption of all communications. In addition, thanks to monitoring tools such as Prometheus, Kiali, and Grafana, it is possible to keep track both globally and specifically of all the services in the mesh. In particular, using Grafana it is possible to create dashboards with custom metrics to monitor crucial aspects such as encrypted connections and certificate expiration. Finally, the security tests performed verified that Istio's controller communications take place using the tls protocol, the application is resilient to possible dos attacks because Istio limits the number of incoming requests, and both the input gateway and the inter-cluster communication gateway (egress gateway) have only two open ports.
For future work it would be interesting to create authentication policies to enable the mandatory use of mtls both at the whole mesh and the individual namespace levels. Also, it might be useful to create authentication policies with JWT Token to accept or deny connection requests. Finally, the collection of metrics with Prometheus, Kiali, and Grafana in a multicloud environment could be optimized by creating additional metrics that

can record the number of requests per individual service and for the entire mesh.

Appendix A

Clusters installation

A.1 AKS cluster

There are several methods to create a cluster on AKS:

- Azure command line interface
- Azure PowerShell
- Azure portal

The Azure command line will be used in the proposed solution.
First, to create a cluster, we need to create a resource group. A resource group is a logical container in which to deploy and manage Azure resources [39]. The following command can be used to create the resource group:

```
az group create --name myResourceGroup \
--location eastus
```

In this command you can replace "myResourceGroup" with the name of the desired resource group and "eastus" with the value for the chosen region. To get a complete list of regions you can type the following command:

```
az account list-locations
```

Once the creation of the resource group is complete, we need to create a container registry: An Azure container registry is a private registry for container images and is used to securely build and deploy our applications. Here is the command to enter on the Azure CLI:

```
az acr create --resource-group myResourceGroup \
--name <acrName> --sku Basic
```

In this case the resource group will be the one created earlier, <acrName> will be the name of the container registry to be created which must be unique within Azure and contain 5 to 50 alphanumeric characters. Basic sku is an optimized entry point that balances storage and throughput.

Now that we have both the resource group and the container registry we can create the cluster.
To create the cluster run the following command:

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--node-count 2 \
--generate-ssh-keys \
--attach-acr <acrName>
```

In this case we need to provide the name of the previously created resource group, enter a new name for the cluster instead of "myAKSCluster" enter the number of nodes (in this case we want to create a cluster with two nodes), and provide the name of the container registry already created.

A.2 GCP cluster

There are two options for creating a GKE cluster:

- Autopilot: mode with which to create a preconfigured cluster, with optimized configuration ready for production workloads.
- Standard: allows you to manage the configuration in a customized way, adapting to production workloads.

Here we will see the steps for installing a cluster with Autopilot mode using Google Cloud CLI [40]. To create the cluster, you must have a project. If it has not yet been created run the following command:

```
gcloud projects create PROJECT_ID
```

Where ProjectID is the ID of the Project you want to create, it must start with a lowercase letter and contain only ASCII letters, digits and dashes and must be between 6 and 30 characters long.
Now to create the cluster just use the following command:

```
gcloud container clusters create-auto CLUSTER_NAME \
--region REGION \
--project=PROJECT_ID
```

In ClusterName should be entered the name of the new cluster, in region should be entered the region for the cluster, instead of ProjectID should be entered ID of the project on which to create the cluster. To be able to see the available regions run the following command:

```
gcloud compute regions list
```

Appendix B

Istio configuration and installation

B.1 Plugin certificates to cluster

First, after receiving the various information from the clusters, we need to access the "config" file in ".kube" directory. In this file are the configurations of the clusters created.

Now, we need to list config file in our KUBECONFIG environment variable with this command:

```
export KUBECONFIG=
"${KUBECONFIG}:${HOME}/.kube/config"
```

After, we need to set two environment variable for the two clusters [41]:

```
export CTX_CLUSTER1=<your cluster1 context>
export CTX_CLUSTER2=<your cluster2 context>
```

We can replace "<your cluster context>" with the name of the context

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUU2VENDQXRHZ0F3SUJB>
    server: https://myakscluster-dns-oa90zx4z.hcp.eastus.azmk8s.io:443
  name: My_AKS_Cluster
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUVMRENDQXBTZ0F3SUJB>
    server: https://35.196.36.191
  name: gke_tesinevola_us-east1-b_my-gcp-cluster
contexts:
- context:
    cluster: My_AKS_Cluster
    user: clusterUser_TesiNevola_My_AKS_Cluster
  name: My_AKS_Cluster
- context:
    cluster: gke_tesinevola_us-east1-b_my-gcp-cluster
    user: gke_tesinevola_us-east1-b_my-gcp-cluster
  name: gke_tesinevola_us-east1-b_my-gcp-cluster
current-context: gke_tesinevola_us-east1-b_my-gcp-cluster
kind: Config
preferences: {}
users:
- name: clusterUser_TesiNevola_My_AKS_Cluster
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUZIVENDQXdXZ0F3SUJBZ0l>
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlKS0FJQkFBS0NBZ0VBc29SaG9>
    token: 72wneox6lm5roxllqxyb4ekyux6hcq18qe8ttnk4npu3lkubq07gfvc2epfwu9r0lavt1un5125ll0bk1>
- name: gke_tesinevola_us-east1-b_my-gcp-cluster
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      command: gke-gcloud-auth-plugin
      installHint: Install gke-gcloud-auth-plugin for use with kubectl by following
        https://cloud.google.com/blog/products/containers-kubernetes/kubectl-auth-changes-in>
      provideClusterInfo: true
```


Figure B.1: Example of config file in which there are informations about the clusters created

present under the section "contexts" in the "config” file.
As said before, before starting the installation, we need to plugin CA certificate. At the beginning we generate the root certificate and key [42]. So we create the directory in Istio installation package to hold certificates and keys:

```
mkdir -p certs
pushd certs
```

We continue with the generation of root certificate and key:

```
make -f ../tools/certs/Makefile.selfsigned.mk
root-ca
```

Then for each cluster we need to generate an intermediate certificate and key:

```
make -f ../tools/certs/Makefile.selfsigned.mk
cluster1-cacerts

make -f ../tools/certs/Makefile.selfsigned.mk
cluster2-cacerts
```

In each cluster we create a secret cacerts. For instance we can provide this command for the cluster1:

```
kubectl create --context="${CTX_CLUSTER1}"
namespace istio-system

kubectl create --context="${CTX_CLUSTER1}"
secret generic \
cacerts -n istio-system \
--from-file=cluster1/ca-cert.pem \
--from-file=cluster1/ca-key.pem \
--from-file=cluster1/root-cert.pem \
--from-file=cluster1/cert-chain.pem
```

Then we can do the same for the second cluster:

```
kubectl create --context="${CTX_CLUSTER2}"
namespace istio-system

kubectl create --context="${CTX_CLUSTER2}"
secret generic \
cacerts -n istio-system \
--from-file=cluster2/ca-cert.pem \
--from-file=cluster2/ca-key.pem \
--from-file=cluster2/root-cert.pem \
--from-file=cluster2/cert-chain.pem
```

B.2 Multi-primary installation

After configuring the certificates, comes the actual installation of the multi-primary cluster [43]. First set the cluster's network to istio-system namespace:

```
kubectl --context="${CTX_CLUSTER1}"
get namespace istio-system && \

kubectl --context="${CTX_CLUSTER1}"
label namespace istio-system
topology.istio.io/network=network1
```

Then the clusters must be configured as primary. You create the Istio configuration for the first cluster:

```
cat <<EOF > cluster1.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
 values:
  global:
   meshID: mesh1
   multiCluster:
    clusterName:cluster1
    network: network1
 EOF
```

Now you can apply the configuration with the following command:

```
istioctl install --context="${CTX_CLUSTER1}"
-f cluster1.yaml
```

After Istio is configured, the east-west gateway must be installed in the cluster:

```
samples/multicluster/gen-eastwest-gateway.sh \
--mesh mesh1 --cluster cluster1
--network network1 | \
istioctl --context="${CTX_CLUSTER1}"
install -y -f -
```

Now comes the stage where services are exposed in the cluster:

```
kubectl --context="${CTX_CLUSTER1}" apply
-n istio-system -f \
samples/multicluster/expose-services.yaml
```

The same steps should be followed for the installation of Istio in the second cluster. Once you have completed the installation of Istio on both clusters you need to enable endpoint discovery. To do this Install remote secret in cluster2 to provides access to cluster1's API server and than do the same for the other cluster.
Command for cluster 1:

```
istioctl x create-remote-secret \
--context="${CTX_CLUSTER1}" \
--name=cluster1 | \
kubectl apply -f
- --context="${CTX_CLUSTER2}"
```

Command for cluster 2:

```
istioctl x create-remote-secret \
--context="${CTX_CLUSTER2}" \
--name=cluster2 | \
kubectl apply -f - --context="${CTX_CLUSTER1}"
```

B.3 Multi-primary verification

As a final step we want to make sure that our multi-cluster cloud communicates properly. To do this we will create two versions, one for each cluster, of a simple Helloworld app. When it receives a request it will respond by providing its own version. To call the Helloworld service we will use another app called Sleep that will simulate traffic in the mesh network [44].
To deploy the services, we have to perform the following steps:

- Create the namespace in each cluster (in this case we use namespace "sample" but we can use whatever name we want):

  ```
  kubectl create --context="${CTX_CLUSTER1}"
  namespace sample

  kubectl create --context="${CTX_CLUSTER2}"
  namespace sample
  ```

- We need to enable sidecar injection in this way:

  ```
  kubectl label --context="${CTX_CLUSTER1}"
  namespace sample \
  istio-injection=enabled


  kubectl label --context="${CTX_CLUSTER2}"
  ```

```
namespace sample \
istio-injection=enabled
```

- Create the service in each cluster:

```
kubectl apply --context="${CTX_CLUSTER1}" \
-f samples/helloworld/helloworld.yaml \
-l service=helloworld -n sample

 kubectl apply --context="${CTX_CLUSTER2}" \
-f samples/helloworld/helloworld.yaml \
-l service=helloworld -n sample
```

- Deploy the application to cluster1 and cluster2:

```
kubectl apply --context="${CTX_CLUSTER1}" \
-f samples/helloworld/helloworld.yaml \
-l version=v1 -n sample

kubectl apply --context="${CTX_CLUSTER2}" \
-f samples/helloworld/helloworld.yaml \
-l version=v2 -n sample
```

- Deploy the Sleep application to both clusters:

```
   kubectl apply --context="${CTX_CLUSTER1}" \
  -f samples/sleep/sleep.yaml -n sample
  kubectl apply --context="${CTX_CLUSTER2}" \
 -f samples/sleep/sleep.yaml -n sample
```

- Now we have to send requests from the Sleep pod on cluster1 to the HelloWorld service:

```
kubectl exec --context="${CTX_CLUSTER1}"
-n sample -c sleep \
"$(kubectl get pod --context="${CTX_CLUSTER1}"
-n sample -l \
app=sleep -o
jsonpath='{.items[0].metadata.name}')" \
-- curl -sS helloworld.sample:5000/hello
```

- Do the same for the cluster2:

```
kubectl exec --context="${CTX_CLUSTER2}"
-n sample -c sleep \
"$(kubectl get pod --context="${CTX_CLUSTER2}"
-n sample -l \
app=sleep -o
jsonpath='{.items[0].metadata.name}')" \
-- curl -sS helloworld.sample:5000/hello
```

In both cases we have to verify that Helloworld version toggles between v1 and v2.

```
francesco@francesco-81N8:~/istio-1.18.0$ kubectl exec --context="${CTX_CLUSTER1}" -n sample
 -c sleep     "$(kubectl get pod --context="${CTX_CLUSTER1}" -n sample -l \
    app=sleep -o jsonpath='{.items[0].metadata.name}')"     -- curl -sS helloworld.sample:5
000/hello
Hello version: v1, instance: helloworld-v1-78b9f5c87f-7n5wj
francesco@francesco-81N8:~/istio-1.18.0$ kubectl exec --context="${CTX_CLUSTER1}" -n sample
 -c sleep     "$(kubectl get pod --context="${CTX_CLUSTER1}" -n sample -l \
    app=sleep -o jsonpath='{.items[0].metadata.name}')"     -- curl -sS helloworld.sample:5
000/hello
Hello version: v2, instance: helloworld-v2-7bd9f44595-47fwm
```

Figure B.2: Helloworld version toggling between v1 and v2 after sending at least two requests from the Sleep pod on cluster1

```
francesco@francesco-81N8:~/istio-1.18.0$ kubectl exec --context="${CTX_CLUSTER2}" -n sample
 -c sleep \
    "$(kubectl get pod --context="${CTX_CLUSTER2}" -n sample -l \
    app=sleep -o jsonpath='{.items[0].metadata.name}')" \
    -- curl -sS helloworld.sample:5000/hello
Hello version: v2, instance: helloworld-v2-7bd9f44595-47fwm
francesco@francesco-81N8:~/istio-1.18.0$ kubectl exec --context="${CTX_CLUSTER2}" -n sample
 -c sleep     "$(kubectl get pod --context="${CTX_CLUSTER2}" -n sample -l \
    app=sleep -o jsonpath='{.items[0].metadata.name}')"     -- curl -sS helloworld.sample:5
000/hello
Hello version: v1, instance: helloworld-v1-78b9f5c87f-7n5wj
```

Figure B.3: Helloworld version toggling between v2 and v1 after sending at least two requests from the Sleep pod on cluster2

Bibliography

[1] Muji. *A brief history of apllication development*. URL: https://hhhypergrowth.com/a-brief-history-of-application-development/.

[2] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: *Nist* (2011).

[3] IBM Market Development Insights. *Microservices in the enterprise, 2021: Real benefits, worth the challenges*. URL: https://www.ibm.com/downloads/cas/OQG4AJAM.

[4] Rachel Nizinski. *Optimizing your multicloud or hybrid environment strategy*. URL: https://blogs.oracle.com/cloud-infrastructure/post/optimize-multicloud-strategy.

[5] Yunusa Simpa Abdulsalam and Mustapha Hedabou. "Security and Privacy in Cloud Computing: Technical Review". In: *ResearchGate* (2021). DOI: 10.3390/fi14010011. URL: https://www.researchgate.net/publication/357354384_Security_and_Privacy_in_Cloud_Computing_Technical_Review.

[6] Hillary Baron et al. "Clooud security complexity: Challenges in managing security in Hybrid and Multi-cloud Environments". In: *Cloud security alliance* (2019).

[7] Baeldung. *Microservices vs. Monolithic Architectures*. URL: https://www.baeldung.com/cs/microservices-vs-monolithic-architectures.

[8] Chandler Harris. *Microservices vs. monolithic architecture.* URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[9] Wubin Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *IEEE International Conference on Service-Oriented System Engineering (SOSE)* (2019), pp. 122–127. DOI: 10.1109/SOSE.2019.00026. URL: https://ieeexplore-ieee-org.ezproxy.biblio.polito.it/document/8705911.

[10] Floyd Smith and Owen Garrett. *What is a service mesh?* URL: https://www.nginx.com/blog/what-is-a-service-mesh/.

[11] *The Istio service mesh.* URL: https://istio.io/latest/about/service-mesh/.

[12] *Architecture.* URL: https://istio.io/latest/docs/ops/deployment/architecture/.

[13] *Istio security.* URL: https://istio.io/latest/docs/concepts/security/.

[14] *What is AWS App Mesh?* URL: https://docs.aws.amazon.com/app-mesh/latest/userguide/what-is-app-mesh.html.

[15] *Transport Layer Security.* URL: https://docs.aws.amazon.com/app-mesh/latest/userguide/tls.html.

[16] *How AWS App Mesh works with IAM.* URL: https://docs.aws.amazon.com/app-mesh/latest/userguide/security-iam.html.

[17] *AWS Secrets Manager Features.* URL: https://aws.amazon.com/secrets-manager/features/?nc1=h_ls.

[18] *AWS Shield.* URL: https://docs.aws.amazon.com/waf/latest/developerguide/shield-chapter.html.

[19] *Logging with AWS CloudTrail.* URL: https://docs.aws.amazon.com/app-mesh/latest/userguide/logging-using-cloudtrail.html.

[20] *Linkerd overview*. URL: https://linkerd.io/2.14/overview/.

[21] *Linkerd architecture*. URL: https://linkerd.io/2.14/reference/architecture/.

[22] *Automatic mTLS*. URL: https://linkerd.io/2.14/features/automatic-mtls/.

[23] *Authorization Policy*. URL: https://linkerd.io/2.14/features/server-policy/.

[24] *Kong Mesh overview*. URL: https://docs.konghq.com/mesh/latest/.

[25] *Kong Mesh architecture*. URL: https://docs.konghq.com/mesh/latest/introduction/architecture/.

[26] *Mutual TLS*. URL: https://docs.konghq.com/mesh/latest/policies/mutual-tls/.

[27] *ACL*. URL: https://docs.konghq.com/hub/kong-inc/acl/#main.

[28] *What is multicloud?* URL: https://cloud.google.com/learn/what-is-multicloud.

[29] *Kubernetes documentation*. URL: https://kubernetes.io/docs/home/.

[30] *What is Azure Kubernetes Service?* URL: https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes.

[31] *GKE overview*. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview.

[32] *Deployment Models*. URL: https://istio.io/latest/docs/ops/deployment/deployment-models/.

[33] *Install multicluster*. URL: https://istio.io/latest/docs/setup/install/multicluster/.

[34] *Online boutique*. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[35] *Prometheus overview.* URL: https://prometheus.io/docs/introduction/overview/.

[36] *Monitoring Multicluster Istio with Prometheus.* URL: https://istio.io/latest/docs/ops/configuration/telemetry/monitoring-multicluster-prometheus/.

[37] *Kiali.* URL: https://istio.io/latest/docs/ops/integrations/kiali/.

[38] *Grafana.* URL: https://istio.io/latest/docs/ops/integrations/grafana/.

[39] *Deploy an Azure Kubernetes Service (AKS) cluster.* URL: https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-cluster?tabs=azure-cli.

[40] *Create Autopilot Clusters.* URL: https://cloud.google.com/kubernetes-engine/docs/how-to/creating-an-autopilot-cluster?hl=it.

[41] *Configure Access to Multiple Clusters.* URL: https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/.

[42] *Plug in CA Certificates.* URL: https://istio.io/latest/docs/tasks/security/cert-management/plugin-ca-cert/.

[43] *Install Multi-Primary on different networks.* URL: https://istio.io/latest/docs/setup/install/multicluster/multi-primary_multi-network/.

[44] *Verify the installation.* URL: https://istio.io/latest/docs/setup/install/multicluster/verify/.