POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Zero Trust Architecture in a Multi-Cloud environment

Supervisors

Prof. Riccardo SISTO

Pietro Santoro, LIQUID REPLY

Candidate

Andrea MARTIRADONNA

July 2023

To those who stayed
and those who left

Table of Contents

List of Figures

Acronyms

AI
artificial intelligence

AKS
Azure Kubernetes Service

AWS
Amazon Web Services

API
Application Programming Interface

CA
Certification Authority

DB
Database

EC2
Elastic Compute Cloud

EKS
Elastic Kubernetes Service

HTTP
Hypertext Transfer Protocol

IaaS
Infrastructure as a Service

IAM

Identity and Access Management

JSON

JavaScript Object Notation

JWKS

JSON Web Key Set

JWT

JSON Web Token

mTLS

mutual TLS

Oauth

Open standard for Authorization

OIDC

OpenID Connect

PaaS

Platform as a Service

PoC

Proof of Concept

SaaS

Software as a Service

SQL

Structured Query Language

SSO

Single Sign On

TLS

Transport Layer Security

VM

Virtual Machine

VPN

Virtual Private Network

Chapter 1

Introduction

Over the past two decades, the migration of workloads to the cloud has dominated the landscape and is now evolving towards solutions encompassing more than just a single cloud instance. Organizations are increasingly more interested in defining valid multi-cloud strategies, and with that, the need for robust and secure architectures arises. This elevates the intrinsic challenges that come with cloud infrastructures to a whole new level and introduces new challenges like automatic service discovery in different networks, consistent identity, and access management throughout the entire architecture, secure channels for communication across clusters, and a way of monitoring/logging all the workloads' interactions. This thesis' goal is to tackle those challenges by exploring a possible solution both theoretically and in practice via a proof of concept emulating the target situation. This work starts by analyzing an alternative to the traditional approach of assuming trust within a network perimeter: This kind of assumption is, in fact, deemed insufficient for the discussed genre of environments, so the concept of Zero Trust is explored, emphasizing the necessity to regard every network and communication as untrusted. As a way of implementing it, the use of a service mesh is explored as a means to ensure secure and reliable communication across different services. This is, in fact, a powerful technology consisting of a swarm of proxies, piloted by a single controller, that attach to the various different application microservices and oversee their traffic behavior. The distributed nature of this approach is perfect for environments spanning multiple clusters, given that, no matter where the deployments are, as long as they can communicate with the control plane, the proxies can effectively manage and monitor the application microservices' traffic behavior. This enables seamless coordination and control across geographically dispersed clusters, ensuring a scalable and resilient framework to monitor and manage both traffic routing and access control between and within cloud environments. While being an overall sound technology both already well-documented and used in single cluster scenarios, it is still in its 'early stages'

for multi-cloud environments. Literature on this topic is scarce and often lacks practical implementations leaving plenty of room to research and 'lateral' challenges such as design and complementary concerns to consider. Specifically, Istio is first discussed and then employed as the key technology for implementing the proposed solution in a proof of concept documented in the paper's second half. After an overview of how the product works and a presentation of the mesh-approach's to the multi-cloud, the design choices of the architecture are discussed, preceding their implementation details, covered in the 6th chapter. The PoC demonstrates how security can be achieved, using a mock-app deployed over two different clusters, emulating a multi-cloud environment, through the enforcement of different kinds of policies, resources and the use of both an external authenticator and a JWT-based fine-grained route permission check. The Istio's mesh is the keystone of the architecture, allowing the clusters, not only to communicate in the first place but also me to orchestrate the architecture's services behaviors according to the application's specific necessities and gate them behind an architectural-integrated custom authentication flow. This is a crucial point of the thesis because while Istio enables control and policy enforcement, it lacks a direct IAM solution that therefore needs to be designed and implemented. A considerable amount of effort has also been devoted to ensuring the PoC's reproducibility through thoughtful design and implementation choices, with the aim of providing any interested party with the necessary resources to establish a laboratory for additional research.

To recap and quickly aid in this work navigation, the thesis is structured into three main sections. The first section, consisting of the first four chapters, focuses on the theoretical aspects of the subject matter. The second section, comprised of chapters 5 and 6, explores the complete design of the architecture and provides a detailed account of the implementation process from the initial software and network choices to its final completion. Lastly, in the last two chapters, before conclusions are drawn, a comprehensive validation of what has been achieved is presented, covering security aspects in both intra and inter-cluster

Chapter 2

Background Concepts

In the following two chapters, I will provide a brief overview of the background concepts necessary to comprehend the objective and scope of the thesis. While the information presented here will definitely not cover all aspects comprehensively, by the end of them, readers should gain a sufficient understanding to grasp the issues and solutions discussed later in the work.

2.1 Cloud Computing

Cloud computing is defined as "The practice of using a network of remote servers hosted on the internet to store, manage, and process data, rather than a local server or a personal computer."[Oxford Languages] These types of services are usually divided in three different categories depending on how much 'control' the user has, those are:

- Infrastructure as a Service (**IaaS**) provide full control virtualized computing resources such as servers, storage and networking. Nominal examples important to us are services like Azure or AWS, which is going to be used in the later-described PoC.

- Platform as a Service (**PaaS**) provide a platform for developing, testing and application deployment, without having to deal with the underlying infrastructure. A use-case would be the deployment of a React Application on Heroku to make it available on the networking.

- Software as a Service (**SaaS**) provide just software applications over the internet. Users usually pay a subscription to access the software, probably the most famous example is Office 365.

Cloud computing is strongly based on the concept of *virtualization*, which allows one or more machines/containers to run on a single physical machine. Let's take a quick step back and define what a container is. Containers are a lightweight and portable solution for packaging, distributing, and running applications. They are essential to the Microservice approach to software architecture - which is based on the idea of structuring an application as a collection of small independent services instead of the classical approach of going for a single monolithic software. The strength of this implementation sits in how efficient containers are by design: They can in fact be started up quickly, replaced and multiplied with little to no effort. The idea of *Orchestration* is strongly tied to the latter in the fact that containers need automation in order to get the desired abstraction and easy provisioning. Orchestration is provided by management software such as Kubernetes and consists in the deployment, scaling and management of containers. Summing up, by virtualizing resources and feeding those to an orchestrator, any application can be quickly and efficiently deployed in a distributed fashion to be accessible from anywhere.

This leads to another categorization of the cloud services based on 'where' the cloud infrastructure is actually deployed and who has access to it. A cloud could be in fact hosted in a proprietary network, be it on-premise or on a third party provider (dedicated to the single organization). This solution, called '*Private Cloud*', provides more control and customization of the infrastructure, leading to enhanced overall security and performance. Another case is what is called '*Public Cloud*', in this model a third-party provider owns and operates the infrastructure on account of different clients. From the economical side, this model follows a pay-per-use type of billing, making it a cost-effective solution for variable workloads. Combining the two approaches leads to what is called '*Hybrid Cloud*'. What this thesis is going to end up focusing on is the '**Multi-Cloud**' Model, to which the following section is dedicated.

2.2 Multi-cloud Environments

A multi-cloud, as the name suggests, is an environment that encompasses more than a single cloud. This usage pattern can be attributed to organizations wanting to avoid the risk of service availability failure and generally speaking dependence on a single cloud provider, what is normally called '**vendor lock-in**', gaining the option to choose specific services from different providers to get the best available options. There are other several factors that make a multi-cloud approach important and put it as the winning horse in the future of IT technologies. The top reasons for enterprises to migrate to a multi-cloud environment are the following:

- High Availability - Redundancy for an organization's services against security

and outages is supplied by a multi-cloud architecture. An example of this is going to be provided in the PoC

- Flexibility - Choice of selecting the best of each cloud provider for specific business goals requirements
- Cost Effectiveness - Operational expenditures can be better controlled by taking advantage of competitive market prices
- Lowered Risk - The likelihood of DDoS attacks drops significantly in favor of a higher level of resiliency that wouldn't be possible with a single provider

Unfortunately with all those positives come some negatives too. Let's look at them in the next section.

2.2.1 Challenges

Figure 2.1: Multi-cloud Challenges

Setting up and maintaining the connectivity between cloud providers is difficult, starting from the fact that they are not even incentivized to make it easy to do, considering their economic interests. This makes it so that additional effort is needed not only to establish a working infrastructure but also to reproduce the kind of security and tools that comes 'out-of-the-box' with the single provider. The lack of expertise in the subject, being it as new as it is, makes **Misconfiguration** one of the biggest risk factors in the industry and scares many companies away. What usually happens is that companies need to rely on third parties and the more third parties are involved the more the **Compliance risk** is increased, where with compliance it is intended the possible failure of acting in accordance with industry law and regulations resulting in legal and financial penalties. A great deal of concern also comes from how **Identity and Access Management** is implemented. This is core to good security and becomes harder to achieve in a multi-cloud scenario. Users need to have the right permissions across the different clusters and all of their many services, and those need to be managed to be consistent and reliable all the time. Single identity management is important and without it being 'centralized' it

becomes a nightmare. A solution would be the implementation of **Single Sign On** (SSO) - a service that allows users to log into the various services with just one ID, which becomes the only thing to manage. This will be a core part of the solution adopted in the PoC and will define how the authentication flow works. Expanding on IAM, Access Management is not to be considered as a problem limited to users. In a distributed application services are entities themselves and as such they need to be regulated: Rules should be applied to allow or block connections based on privilege levels, or as we will see in a bit 'necessity'. The bigger the environment, the harder to keep track of all that is happening. **Monitoring** and producing logs is really important, ideally every action of every communication flow should be tracked. This is even more important in the architecture we are discussing, but as the reader can imagine is even harder to achieve. A centralized monitoring solution is usually implemented, in more-standard scenarios, as an additional layer - a single pane of glass - and is a must-have nowadays. In summary, many of these problems come from the **Complexity** added with this strategy that comes from the different ever-changing resources that compose it: Variable nodes in size and addresses, access controls, different cloud features and limitations.

The standard approach pre-cloud era of boundary-defined trust is unsuitable to address these concerns and therefore in the next chapter a different, newer idea of 'trust' will be presented. While some of these challenges are going to be answered intrinsically by the later adopted technologies, some will remain and will eventually be tackled in the Design chapter. The important point here was to present a general overview of the situation before going into more specific solutions.

Chapter 3

Zero Trust Principles

A Zero Trust Architecture is a security model that assumes that all users, devices and applications are untrusted until verified. In Traditional perimeter-based security models, attackers can access any data once bypassed the perimeter's defenses. This approach used to work in an era in which there was a distinct line separating the company's network from the outside. Nowadays, however, the 'scattered' nature of architectures, thanks to the cloud, makes that line of reasoning obsolete. To address this, this paradigm imposes that trust is verified at every step via the enforcement of strict access control, even inside the company's network: IP addresses are no longer valid identification and authorization cannot only rely on considerations based on them. The whole idea can be described using four basic principles that,

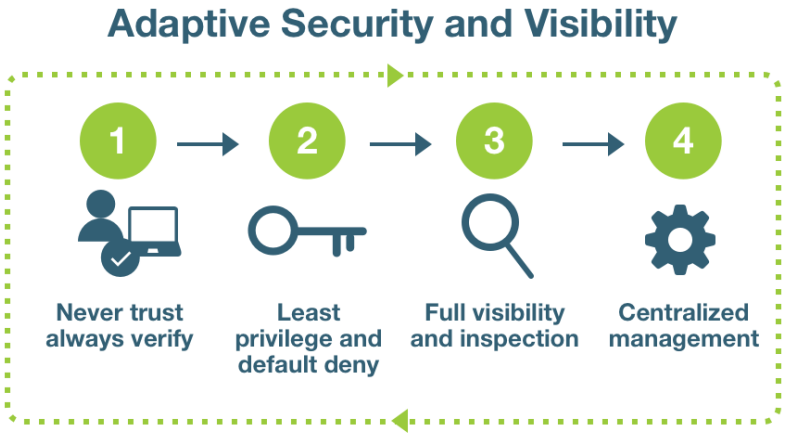


Figure 3.1: Zero Trust Principles

contextualized to our domain of interest, can be expressed as follows:

- **Verify Always**: Authentication and Authorization policies must regulate service-to-service communication.
- **Least privilege and default deny**: Services must only be able to communicate if necessary for the business logic.
- **Full visibility and inspection**: Logs about every communication inside the mesh need to be produced.
- **Centralized management**: A control center is needed to easily manage, monitor and apply policies on the scope of interest.

Least privilege is what we were discussing before when we talked about how services are to be considered entities, and they should be subjected to access control themselves. How do we 'verify always' and authenticate the service, though? We need each of them to have a defined and verifiable identity. Certificates come to mind as the solution, but they bring additional problems with them: how to deal with certificate exchange and issuing, how to share trust bundles to trust certificates from the other cluster, and who is responsible for issuing them? We will base our trust on a single root that will be issuing certificates for both clusters in the PoC, their management instead will be discussed in the next chapter.

3.1 Service Mesh

A Service mesh is a software infrastructure layer that helps manage and secure microservices in distributed systems, and from our standpoint, a perfect solution to build a Zero Trust Architecture over different clusters. It presents itself as an additional separate layer, and as such it can be applied to any application transparently providing a set of features independently of the business logic and most importantly without having any dependencies or effect on its code. It's implemented as a network of proxies that are individually attached to application pods and collectively enforce policies and manage the workloads. There are different benefits provided by its adoption aside from the security capabilities, and those can typically include load balancing features, failure recovery, monitoring and even automatic service discovery which can be a major challenge in a multi-cloud environment. An opposite approach to a mesh would be that of the API gateway, a single component that handles transactions by sitting between clients and services. What makes the mesh much better for the situation is how an API gateway needs updates every time the application changes in the form of a microservice being added or removed, while a mesh just naturally adapts to it.

The structure of a service mesh can be divided into a '**Control Plane**' and a '**Data Plane**'. The former's job is to receive the configuration resources and

rely them on the latter to carry them out. The structure is asymmetrical because whereas the control plane is made by core components of the mesh, the data plane is made by a collection of proxies called 'sidecar proxies' attached to every microservice whose job is to intercept any incoming connection and decide what to do based on how they have been instructed.

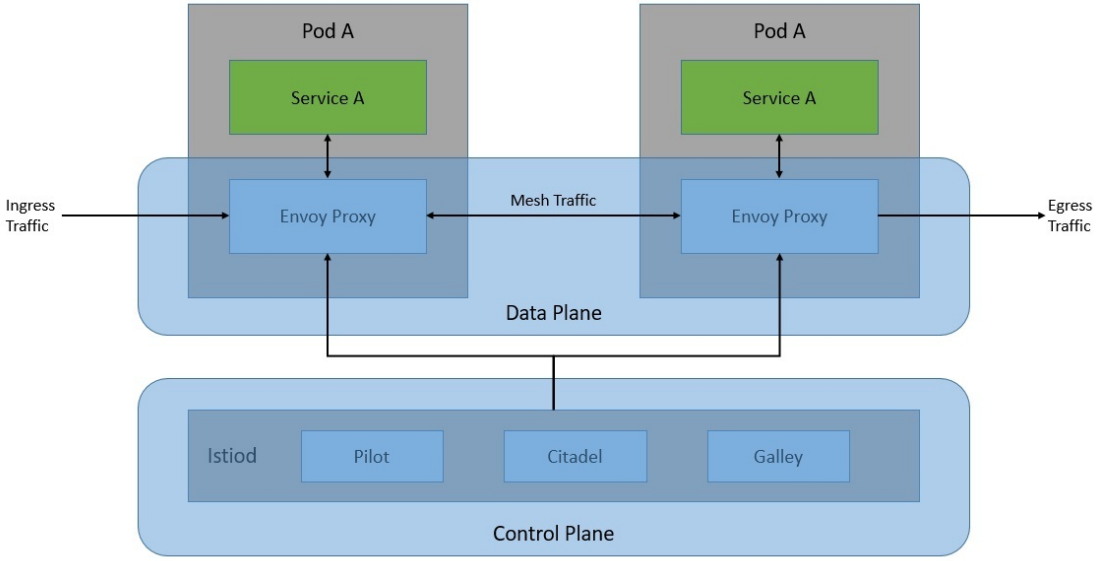


Figure 3.2: Service Mesh Structure

This type of architecture has obviously both advantages and downsides. Let's look at them: Upsides:

- Simplification of communication between services
- Easier diagnose of communication errors
- Natural predisposition to security features such as encryption between services
- Faster testing and deployment of applications

Downsides:

- Each service call must pass through proxies, which is an additional step
- Network management complexity is abstracted but not eliminated

Overall it's a good deal, but the enterprise adoption is still nascent and expertise in it is still missing for the most part. There are different viable solutions already used in production, with the most known being Istio, Linkerd and Consul. When

selecting a product to explore, several considerations have been taken into account such as supported workloads, provided features and the available documentation but ultimately, research has proven all of them to be equally valid under all of these aspects. Therefore, the choice of Istio as the chosen technology primarily relies on the existing expertise within Liquid Reply, the company collaborating on this thesis and the higher level of interest it has drawn to itself compared to the other two options.

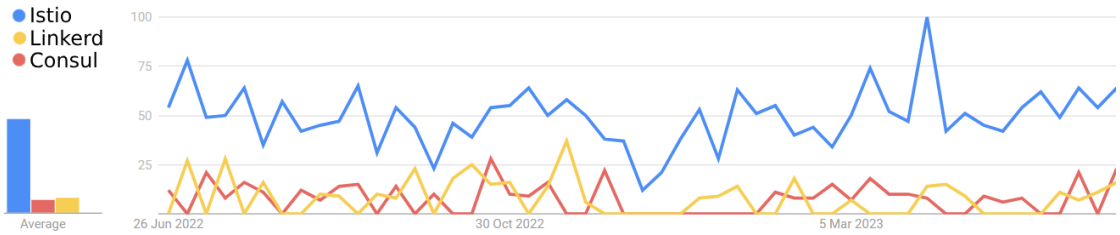


Figure 3.3: Service Mesh Search Trend

3.2 Istio

Istio is the most widely adopted open source implementation of the Service Mesh, it can be used with any cluster manager like Kubernetes and brings vital features including, half-quoting from its official documentation:

- Secure service-to-service communication with mTLS, strong identity-based authentication and authorization.
- Fine-grained control traffic.
- A policy layer supporting access controls.
- Automatic load balancing.
- Automatic metrics, logs and traces for all traffic.

Fitting the previous description of service meshes, Istio itself is logically split into a Data Plane and a Control Plane. The Data Plane is composed of a set of extended '*Envoy*' proxies deployed as sidecars to services (in the same pod) where Envoys are high performance C++ based proxies that mediates all inbound and outbound traffic. The Control Plane is called **Istiod** and is a single component that encapsulates three legacy actors: Pilot, Citadel and Gallery.

- **Pilot** is the one responsible for translating rules into proxy configurations.
- **Citadel** manages certificates, identities and authentication.

- **Galley** handles User-specified configurations and integrates them into other Istio elements.

Probably the biggest feature that immediately jumps to the eye while reading this chapter's introduction is the automatic implementation of mutual TLS between services. This is the protocol responsible for establishing an encrypted connection in which both parties authenticate each other using X.509 certificates. We mentioned previously how certificate management is such an important and hard part to manage but here Istio comes to the rescue providing within its Citadel component an automatic system for certificate issuing and rotation. Following on this, Istio offers both the possibility of having an external root CA, which is going to be the case in the PoC, and the possibility of self-signed certificates using its internal CA, effectively making for a 'plug and play' secure environment for app deployment. The ability to authenticate services and having their traffic pass a proxy leads to the possibility of easily achievable monitoring and policy enforcement, key points in the Zero Trust architecture we aspire to obtain.

3.2.1 Main Resources Explanation

Istio's features are configured using a set of k8s custom resources in the form of YAML files. This section is going to provide an explanation and some screenshots of the most important and used ones in the PoC to then use as reference for any eventual doubt that could arise, reading the implementation section.

The Peer Authentication policy

```yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default  #name of the policy
  namespace: istio-system
spec:
  mtls:
    mode: STRICT
```

Figure 3.4: Istio PeerAuth Resource

PeerAuthentication defines how traffic will be tunneled (or not) to the sidecar. It is mainly used to switch from PERMISSIVE mode which allows in-clear traffic to STRICT mode which denies all but mTLS traffic in the chosen mesh, namespace, or single workload. There are possibilities to combine different preferences to allow only certain ports or have more fine-grained control over which pod is allowed

which type of communication, but the former use is the standard and can be seen in the figure.

The Authorization Policy The Authorization Policy enables access control on workloads in the mesh. It is applied using a selector to a certain set of resources and specifies:

- Sources: The 'principal' from which the communication is coming from. Introduced by the keyword 'from'
- Operations: The HTTP methods and paths of the request. Introduced by the keyword 'to'
- Conditions: Headers and JWT claims. Introduced by the keyword 'when'.
- Action: The action to take in response. Introduced by the keyword 'action'

There are three types of action, ALLOW and DENY do as their name says. There is also CUSTOM, which offers the interesting possibility of having an external authorization system to delegate the authorization decision to. While still an alpha feature, this is part of the implementation in the PoC, and therefore it is explored in a little bit more detail later.

One important use of this policy is the deny-all policy. This is a simple rule that blocks all traffic by default. What this helps achieve is the sought 'Least Privilege and Default Deny' Zero Trust principle. After the application of this rule, in fact, a rule to allow each specific communication will be needed and no freedom will be left to the services.

```yaml
#Policy that by defaults denies all access
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "deny-all"
  namespace: istio-system
spec:
  {}
```

Figure 3.5: Deny-all Policy

The Request Authentication resource The RequestAuthentication defines what request authentication methods are supported by a workload. It is used to reject requests not compliant with the configured authentication rules, even though it will not automatically reject requests that do not carry any form of authentication, letting them in simply without any authenticated identity. This is

crucial in implementing access control and can be paired to an Authorization policy to block any non-authenticated request and even accept only those containing a valid JWT token. This type of resource can be applied to the Ingress Gateway to stop unauthorized access as early as possible. In the PoC it's applied, in combination to a specific JWT property check, to a critical HTTP DELETE route for a service, and can be seen HERE and HERE.

The Virtual Service resource The Virtual service resource is used to configure traffic routing, specifically rules to follow when a host is addressed. When a match is found, the traffic is routed to the destination service or services. Like the authorization policy, it has a set of fields important to make it work. This resource is how the traffic that passes through the Ingress gateway is routed to the right microservice. One important thing to keep in mind is that this routing only happens after the auth policies have an effect and the request is not dropped. This resource offers a number of possible configurations, some even really complicated, which is necessary to cover the more particular use cases. A comprehensive guide with examples of its use can be found in Istio's documentation. The same is valid for the other resources as well, but probably less necessary. An example of it can be seen directly from the implementation in chapter 6 or by clicking HERE.

The Ingress-Egress Gateways

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: micro-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

Figure 3.6: Istio Ingress Gateway

The ingress Gateway and the Egress gateway are two different resources that are used to influence the input and output of the cluster. When a gateway is created, traffic can be routed through it, giving the possibility to apply any control needed. The ingress is thought as an entry point of the architecture and will be a critical component in the PoC, effectively enabling the designed authentication flow to work as intended. The Egress is instead the component that, used in conjunction with DestinationRules, Virtual Services and ServiceEntries, is used to deny traffic

from the inside to the outside unless 'whitelisted'. This will be implemented for completeness, because, even not being the main focus of the work, is still such a powerful tool to have and therefore interesting to look at.

Chapter 4

Service Mesh in the Multicloud

In Chapter 1, we discussed a range of challenges associated with constructing an architecture across multiple cloud environments. Now, with the advent of service meshes as a solution, it is only natural to explore how these challenges are effectively tackled and identify any outstanding issues. Let's delve into each challenge and examine the benefits introduced by service meshes:

- **Misconfiguration** - The possibility of making configuration errors that can lead to security or performance issues. Service mesh → Offers centralized traffic management and configuration capabilities other than straight automatic service discovery, allowing for a clear and consistent definition of communication policies between distributed services within the environment.

- **Compliance** - Maintaining compliance with regulations and corporate policies becomes harder across multiple cloud environments. Service mesh → provides tools for enforcing security policies, authentication, and authorization consistently across the multi-cloud. It aids in maintaining compliance by facilitating access controls and the implementation of tailored security measures for each specific environment within the multi-cloud.

- **IAM** (Identity and Access Management) - Managing the identity and access of various distributed services in a multi-cloud environment can be daunting. Service mesh → Can integrate with existing IAM systems to provide centralized and granular access control for services. Simplifies IAM management by allowing the usage of a unified authentication and authorization framework across the multi-cloud, ensuring consistent identity management practices.

- **Monitoring** - Monitoring performance, traffic, and metrics of distributed services can be challenging and scattered. Service mesh → offers centralized monitoring capabilities, allowing for the collection and visualization of performance data and metrics from all services by collecting passively information via sidecar proxies.

- **Complexity** - Managing a multi-cloud environment involves dealing with multiple platforms, tools, and configurations, increasing overall complexity. Service mesh → provides an abstraction layer and a common interface for managing, configuring, securing, and monitoring service-to-service communication, streamlining operations and therefore reducing complexity.

Service meshes are powerful tools, but by themselves are not enough to reach a working architecture compliant with our objectives. We've seen how they tackle the major challenges of the multi-cloud and how they implement the zero trust principles, but still, some other considerations need to be done. First of all, the mesh gives the tools to enforce consistent authentication policies for the users to abide by, but the entire authentication mechanism on which to base those policies, needs to be supplemented externally, and therefore needs to be an object of design, as we will see later in Chapter 5. Compliance risk is, once again, something that meshes allow addressing but it requires a case-by-case design to be overcome, as problems related to data synchronization and eventual cloud providers' direct limitations.

The type of deployment also needs to be taken into account, and this is exactly what the rest of this chapter will be about. For utility reasons, the main focus of the remainder of this study will revolve around Istio's approach to the matter, owing to its extensive documentation and its direct relevance to constructing a functional Proof of Concept.

4.1 Istio's Deployment Models

Let's make a consideration before going into detail on possible deployment models. Each Kubernetes cluster has an API Server which is used to manage configurations, deployments and also serves as a way to provide endpoint discovery by serving service information. By this logic, a cluster can be seen as boundaries that divides the internal network from the external, therefore the issue of connectivity between different clusters. This problem is answered by Istio in different possible ways depending on the specific scenario. Istio has native support for the possibility of being deployed over a multi-cluster environment made by any number of clusters and any number of different networks, where for network it is intended a collection

of workload instances that have direct reachability like in a single cluster. There are different deployment cases and topologies, ranging from the single cluster in a single network, the easiest case, to multiple clusters in different networks, which is exactly what we are going to be discussing.

Interestingly, the network models are not the only discriminant in choosing the type of installation. There are in fact two different approaches to how the control plane can manage the multi-cluster network. In the simplest case, single cluster scenario, as we've already seen how there is a single control plane that runs the mesh acting as its 'head'. A cluster with its own control plane is referred to as **Primary Cluster.** When another cluster comes into play, there is the possibility of sharing the same control plane. A cluster that does not have its own Istiod deployment and is managed by another cluster's control plane is called **Remote Cluster.** To make any of the following two model work, it's important to allow the API Server observation from the Primary cluster. To enable this, a 'remote secret' is generated on the cluster to be observed and deployed on the observing cluster. This contains credentials and an address, granting access to the API server in the cluster.

4.1.1 The Primary-Remote Model

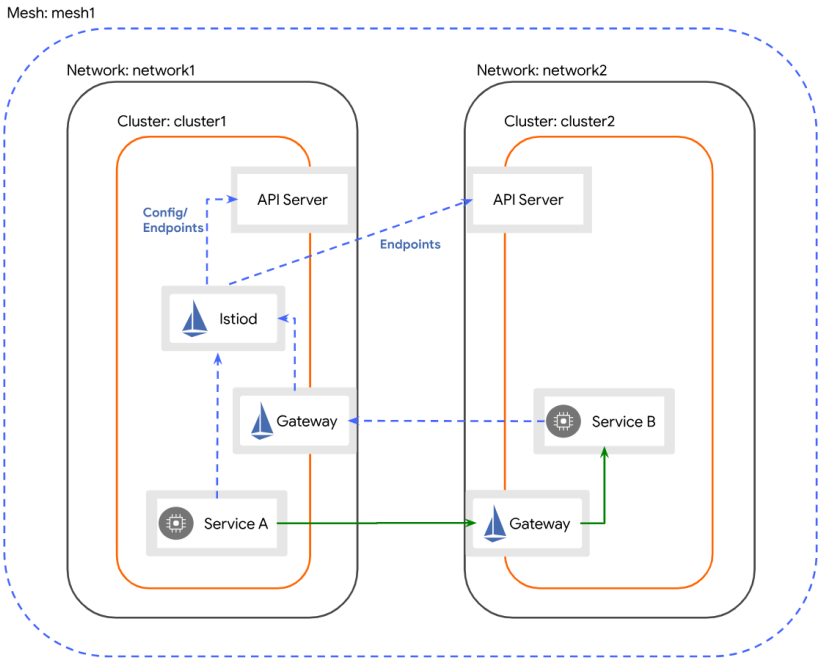


Figure 4.1: Primary and remote clusters on separate networks

In this model, the control plane is effectively deployed in one of the clusters and is accessible via a stable IP for the others. This is achieved by exposing the control plane through a dedicated Istio gateway. The control plane is going to observe both API Servers for endpoints so to be able to provide service discovery. Given that services belonging to the different clusters are still going to be in different networks, communication wouldn't be possible without the usage of dedicated gateways called east-west gateways. By default, these gateways come public on the internet and might need additional access restrictions via firewall rules in enterprise systems, nevertheless, they are configured for TLS pass-through so in actuality only traffic allowed with policies and with a trusted mTLS certificate and workload ID will pass.

4.1.2 The Multi-Primary Model

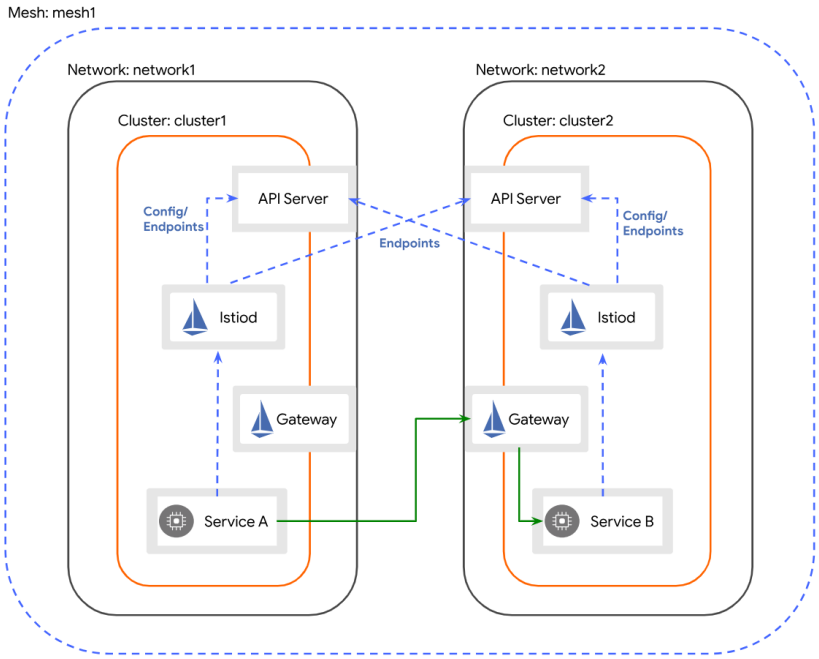


Figure 4.2: Multiple primary clusters on separate networks

In this model, there is a control plane in each cluster, effectively making them two primary clusters. Both control planes will observe the API Servers in each cluster for endpoints. Communication as previously is only possible through east-west gateways, and the same considerations as before apply. What this solution brings to the table is a higher availability and additional resilience. If one of the two control planes fails, the scope of the outage is only limited to its cluster workloads. Those

might even be deployed redundantly on the other cluster as well, and in this case, the problem would be greatly reduced if not totally eliminated. Another advantage is 'Configuration isolation', which is the possibility of changing configurations only in one cluster without impacting the other. This greater control over configurations provides a more controlled rollout, perfect for, as an example, canary configuration changes. Lastly, given how services need to be exposed to be seen by the other control plane, it's possible to restrict visibility to only part of the mesh, creating low-level isolation.

4.1.3 Considerations regarding Trust

In a single mesh, whenever a workload is created, Istio assigns it an Identity and a certificate. It's been mentioned how, by default, Istio uses an internal CA to sign those certificates. The way certificate works is that they can be verified by using the public key of the CA that created and signed the certificate. Being the CA Istio itself, the trust bundle is shared among the entire mesh. This model allows all workloads in the same mesh to authenticate each other effortlessly. To enable communication between two meshes with different CAs, the exchange of the trust bundles of both is needed. This is not directly addressed by Istio and has to be implemented in some other way. An interesting possibility to maybe explore in the future is the use of an automatic protocol like SPIFFE trust domain federation, as the documentation suggests. The way this is addressed in the PoC is by having a common external root CA with its signature on both the intermediate CAs responsible for each cluster's certificates, which is just as valid of a solution for the kind of study that needs to be conducted.

4.2 Other Available Approaches

When previously the choice of Istio was discussed, it was mentioned how implementation products for the service mesh were equally valid. This section covers for informational purposes a little bit of how the two other products approach multiple clusters.

Linkerd multi-cluster support is implemented through the "mirroring" of services via the use of a specific component - service mirror, and allows connectivity using a multi-cluster gateway. The mirror updates local services after it detects a change in a cluster it is observing. The only requirement is that Kubernetes Services match a label selector to be exported to other clusters. One thing to note is that Linkerd needs a control plane installation in each cluster with a common trust anchor, so it basically acts like a multi-primary Istio installation.

Consul proposes WAN federation in which the same services are deployed over different independent clusters that communicate over the WAN on specific ports.

Clusters are connected as a full-service mesh, in which they are able to connect via RPC and gossip protocols. All consul servers in all federated datacenters must use RPC certificates signed by a shared CA and have their datacenter name figured in the SAN certificate field. The model also has the first cluster created and designed as primary with some authority over global states and configurations. A more advanced WAN federation is also possible but goes outside this overview.

Independently of the products' specific implementation, service meshes' service discovery, traffic routing, load balancing and monitoring capabilities provide a unified control plane to manage and configure services consistently regardless of the underlying number of networks and cluster configurations, abstracting away the complexities of network topologies. So to conclude this section, while specific implementations or even models may vary across the different possible solutions, the general approach to multi-cluster environments ends up providing the same kind of useful abstractions.

Chapter 5

Designing the Architecture

This chapter aims to provide a comprehensive 'conceptual' insight of the realized work. It will introduce the PoC and highlight key 'high-level' choices made during its development. In the next section, the chosen authentication strategy will be introduced, to later be expanded upon in the next chapter, along with a discussion about its and the application's possible deployment configurations accounting problems such as Data synchronization and consistency and some consideration on Data sovereignty and compliance. Finally, the developed sample application will be presented to, by the end of the chapter, give an almost complete high-level overview of the final architecture.

5.1 The Auth Flow

When the starting chapter introduced the challenges of multi-cloud, it highlighted the difficulties of maintaining consistent user privileges across a distributed architecture. This encompasses the broader issue of IAM (Identity and Access Management). The underlying concept that this thesis follows and implements in the PoC (Proof of Concept) is the separation and independence of the authentication process from the deployed application's logic. This approach allows for the creation of a system where permissions can be tailored to specific business logic, while also serving as an adaptable framework that connects and manages security across different networks. To achieve the desired authentication flow, the SSO (Single Sign-On) solution is combined with Istio's routing and policy enforcement capabilities. By restricting access to the cluster through an Ingress Gateway, it becomes possible to verify if the user attempting to access the system has the necessary privileges.

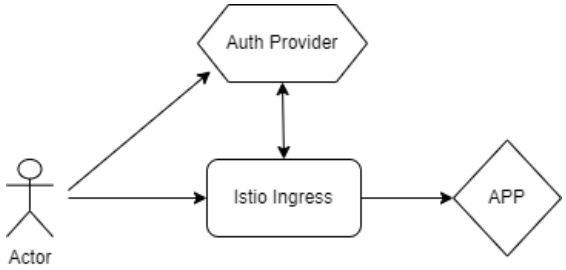


Figure 5.1: Simple Auth Schema

The chosen solution is based on the issuance of a JWT (JSON Web Token) from a configured provider, ensuring appropriate user authentication. Subsequently, the focus shifts to authorization: having identified the user interacting with the system, the next step is to grant them the appropriate level of access. Once again, Istio proves valuable in enabling consistent and manageable behavior through its Authorization policies. Furthermore, as mentioned earlier, the same JWT used for granting access can also carry additional information to accommodate more fine-grained access requirements, if necessary. These rules can be applied to any section of the multi-cluster architecture, given that it is managed conjunctively by Istio. Consequently, only one identity per user is required, effectively serving as the keystone of a secure and policy-driven strategy across all microservices. For the complete implementation of this auth flow and more low-level details, see the corresponding section or click here.

5.2 Deployments' Choices

By using this combination of resources, components, and traffic rules, we can be certain that, whoever tries to interact with the underlying application will always have its permissions checked, independently of where the service he is trying to access is deployed in the architecture. This approach to IAM results in an efficient way to solve the issue at hand but opens another interrogative. The 'authentication zone' is a critical section of the system, and having it fail would cause the entire framework to collapse and the application to stop working. Working with multiple clusters, the possibility of multiple deployments opens up: Should this zone be deployed redundantly on different clusters, or is it a bad decision? Design choices such as this one are the next logical step to take: There are pros and cons, like in every choice. Having a redundant auth workflow is extremely beneficial for resiliency and availability, but some new problems arise. We've looked at the authenticator as a single entity for now, but what happens if we deploy another version of it? As long as they behaved like static entities, probably nothing. The moment one were to get

updated, though, the other would end up being out of sync. Synchrony would be a major issue and would need to be addressed properly. When considering possible implementation of the desired architecture, one potential solution for this scenario was to adopt a shared database instance among the various Keycloak provider instances. The idea behind this approach was to ensure automatic synchronization between the providers, but it shifted the concern of resilience on the database itself, rather than on the providers. The reliance on a single database instance was itself a potential single point of failure, not improving the state of the overall system.

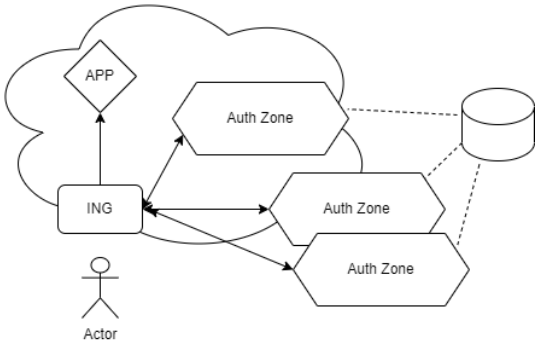


Figure 5.2: Redundancy on AuthZone

As a consequence, an alternative approach was sought to address this concern. Rather than focusing on multi-cluster redundancies for the authentication zone, the concept of dedicating an entire cluster solely to authentication was explored. This approach aimed to emulate a real-life scenario where critical elements of the system could be allocated more resources and dimensioned better.

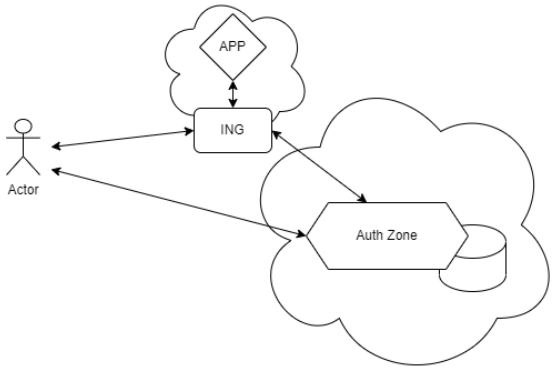


Figure 5.3: AuthZone Dedicated Cluster

The resulting architecture consists of two clusters: one dedicated to authentication and the other dedicated to the application. While the authentication cluster represents a potential single point of failure, it can also be protected and scaled as necessary to meet the system's requirements, effectively allowing it to receive focused attention ensuring availability and performance.

Regarding the deployment of the application, a similar line of reasoning can be applied: What would be the most optimal approach? The answer to this question heavily relies on the specific case at hand. However, some general considerations can still be made, especially when it comes to compliance. Data sovereignty and Compliance regulations, such as data privacy laws or industry-specific requirements, impose certain standards that need to be met. By leveraging multiple clusters, organizations could potentially benefit from enhanced compliance measures: Different clusters to handle specific types of data or to adhere to specific compliance requirements. An example of this is how a cluster could be designated for services that don't handle personal data, while another could be dedicated to sensitive operations. This separation would ensure that compliance measures implemented for data such as those imposed by the GDPR don't burden the rest of the infrastructure, also allowing for better controls and auditing procedures in the process, thereby aiding compliance efforts.

5.3 App Introduction

In order to showcase the workings of a service mesh properly, the multi-cluster infrastructure will host a sample application whose development has been driven by the objective of highlighting all the relevant use cases that have emerged during its conception. It is important to note that the application is an independent entity within the architecture and is not constrained or mandated to undergo alterations for compatibility purposes. This highlights the fact that the development choices were made solely to explore the possibilities presented by such an architecture. Since the application didn't need to abide by any special needs, given its microservice nature, my natural choice was to use JavaScript with the React framework for its main component and NodeJs for its satellite servers. This was a given for me because of the experience gained acquired during various university-related projects. The app was built gradually, starting from its 'main' body and being enriched with new external services each time a new feature was required for some policy showcase. Due to this kind of programming approach, a version of a forum came into mind, in which users can create topics to which they or others can post updates. The structure of the application ended up being made of 4 main components:

1. react-fend

2. microdb
3. microtwo
4. microusr

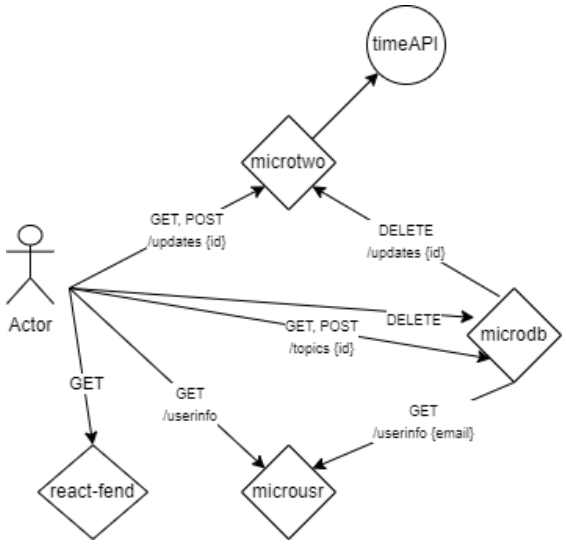


Figure 5.4: App Schema

The **react-fend** component is the one that provides the interface for the user to view gathered information from the other components and interact with them via controlled inputs. Being the application React-based, this element will be contacted once by the user to get the resources and code needed and then run inside its browser natively. Each subsequent request will come directly from its browser and won't pass through this.

The **microdb** component is responsible for different operations: It presents topics and enriches them with information regarding their owner, deletes updates on topic deletion to keep consistency, manages the insertion of new topics and offers a debug utility to have better insight on what happens underneath. Its endpoints are the following:

- /debug - prints information needed for debug. In its last iteration prints the JWT associated with the request if one is present
- /topics [GET] - allows to retrieve all the topics present in the database and enriches them with information relative to its owner by contacting the microusr service on /userinfo/owner

- /topics [POST] - allows the insertion of a new topic in the database with a unique ID and information linking it to the owner
- /topics/id [DELETE] - Deletes the topic with the specified ID, also contacts microtwo to delete relative updates to keep consistency

In order to test the authorization request feature provided with the service mesh, the application distinguishes between PRIME members and standard members. the /topics [DELETE] route is therefore protected and only accessible to PRIME privilege holders.

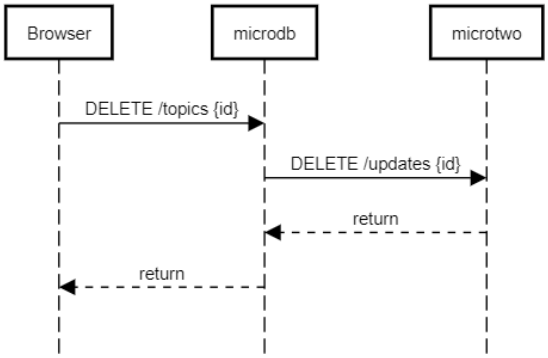


Figure 5.5: Sequence Diagram - Deleting topic

The **microtwo** component is an interface to a database whose job is to store individual updates to each post. This component is also responsible to timestamp updates and uses an external service to do so, specifically worldtimeapi.org. This was done to highlight the functionalities of the egress gateway, specifically its ability to enable only communications towards permitted hosts. It offers:

- /updates/id [GET] - Returns updates for a specific topic
- /updates/id [POST] - Saves a new update for a specific topic, timestamps it
- /updates/id [DELETE] - Deletes all the updates for a specific topic

The /updates/id [DELETE] route can only be called by the microdb component and will fail in any other case. This 'extends' the control present on the deletion route for microdb effectively only allowing PRIME members to delete reviews.

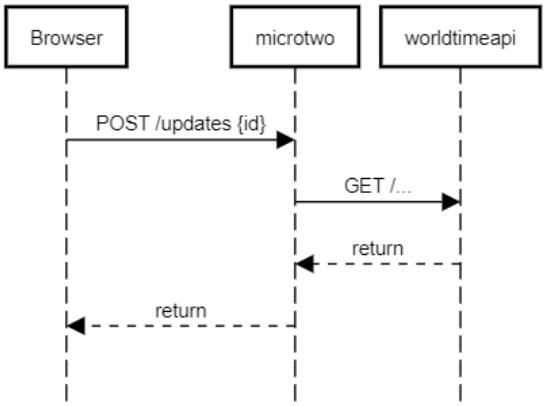


Figure 5.6: Sequence Diagram - Posting Update

Lastly, the **microusr** component serves as storage for app-related user information, which in this case is a simple color preference for their in-app name. This component offers two routes that basically do the same thing:

- /userinfo/mail - Returns information in the database for the user associated with the specified email
- /userinfo - Returns information for the 'current user'

The main difference between the two is that the route without parameter relies on information present in an associated JWT to fetch the correct data. This was done as a further initial way of testing the architecture and following a personal interest as a test to bring the JWT checking directly into the app code. Each of the micro* services just described, excluding the first one, relies on a personal database to operate. The choice was the simple SQLite option. An instance was generated for each of them and is only accessible to them being physically mounted to their respective pod. This was a simplification done to speed the development of the app, but in a real-world scenario where services are deployed over different machines by an orchestrator, a different approach should be considered for sure.

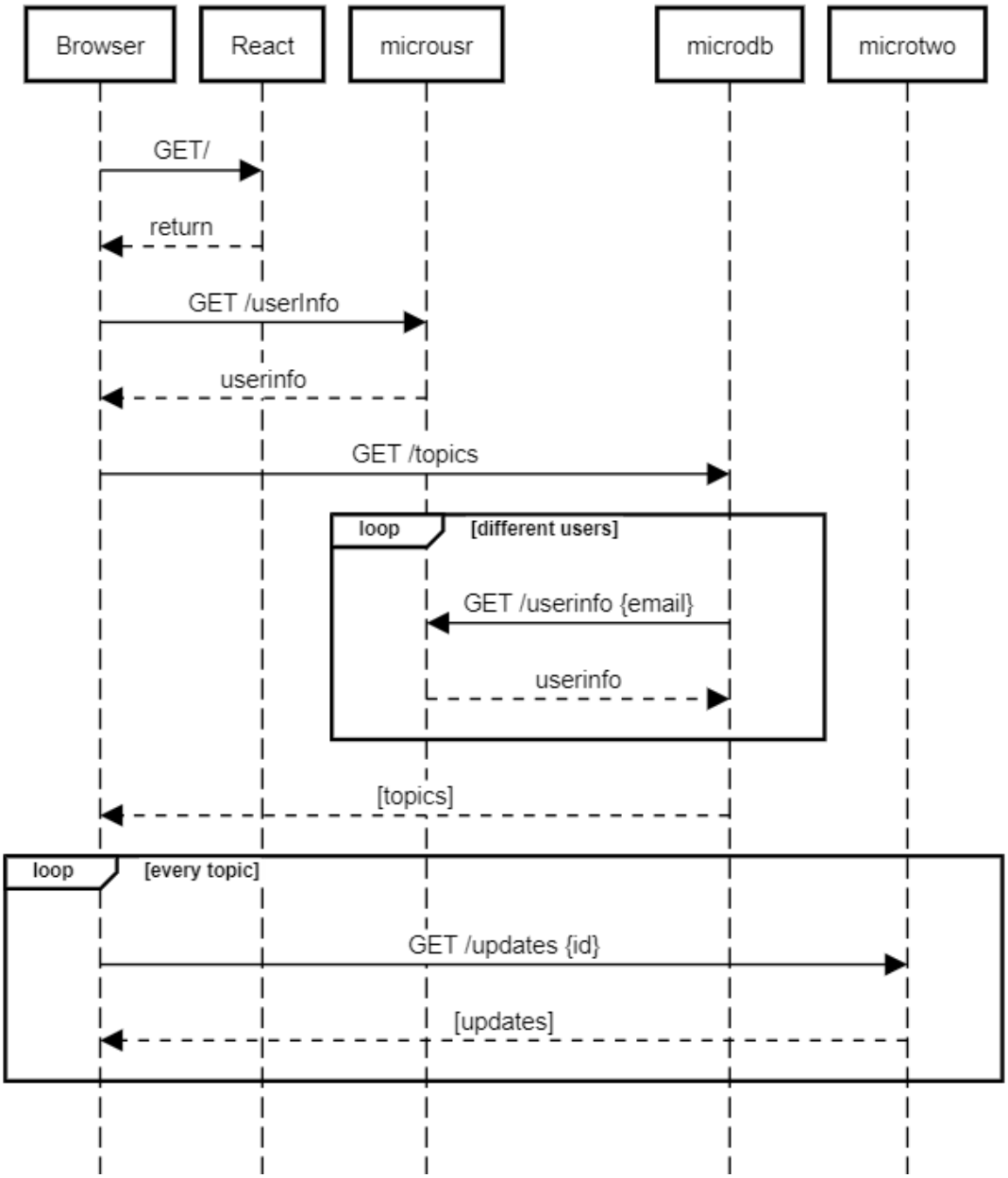


Figure 5.7: Sequence Diagram - First Access

5.4 Architecture Overview

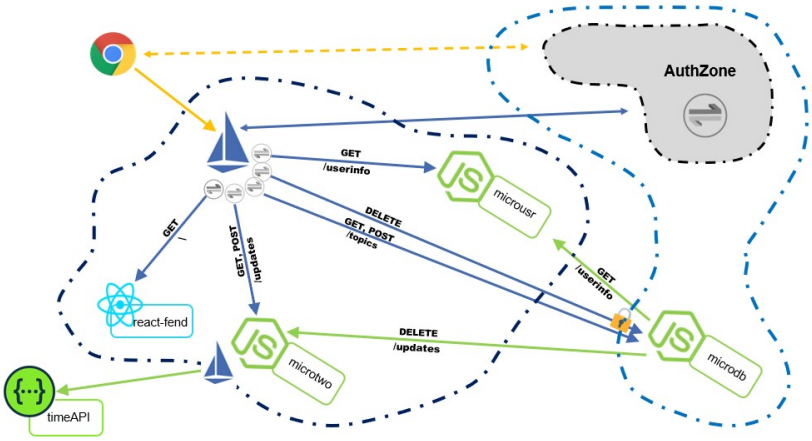


Figure 5.8: Application Overview

We have provided an overview of the app's functionality, briefly discussed the desired authentication flow, and considered the distribution of microservices. In the final architecture, a decision was made to deploy the authentication zone (authzone) to a separate cluster, distinct from the one hosting the application. For demonstration purposes, after the considerations made in Section 5.2, the application was divided, and one of its services was deployed in the same cluster as the authzone but in a different namespace. In the upcoming chapter, we will delve into the complete implementation process for this project, encompassing all the relevant technical details. We will also solidify the concept of the 'authzone' that has thus far been presented in a conceptual manner.

Chapter 6

Proof of Concept

This chapter will cover all the implementation details regarding the proof of concept that was developed for this thesis. This was thought to serve multiple objectives. Firstly, it aims to provide a first-hand experience with the theoretical concepts that have been studied so far, offering a more practical perspective on the matter and possibly soliciting more thoughts. Secondly, it aims to establish a laboratory, creating a working environment specifically designed for testing purposes related to this topic. Typically, setting up such an environment would require a significant initial effort, but this thesis endeavors to simplify the process. Lastly, the thesis aims to thoroughly examine the proposed architecture in order to validate the solution. The final section of this chapter will cover a Script written during the thesis to speed up setup and testing time, and ideally will reference the material explained in the sections before connecting it all together.

6.1 Setting up the environment

First of all, when it was time to decide how to start on the PoC, a meeting with my Liquid Reply team took place in which options were discussed. As it turned out, the setup effort and economic cost of setting up and running a 'concrete' multi-cloud Kubernetes environment using services from different providers such as AKS and EKS, would have been way too much for the demonstrative purpose and nature of the work to be conducted. A different solution was needed, something that could be easily picked up and turned into an accessible laboratory. To cover the Kubernetes aspect, a software called minikube was opted for, this will be better explained later, but it basically emulates an entire cluster on a single node. The node in question was the other concern: Initially, I started locally on my company computer. What became apparent was that it was not the optimal solution because what was required to run weighted heavily on the resources and greatly limited my

ability to multitask. This was the reason the company quickly provided me with a dedicated machine on the cloud for me to do anything I needed. The machine provided was an AWS XLarge-format EC2 instance running Ubuntu and 16 GB of RAM. The way I interacted with it was through SSH using my user 'admar' as it will possibly appear in some of the following screenshots. For what concerns security, a 'security group' was created alongside the instance and through it, it was possible to configure firewall rules to limit or grant access to the machine. During the period of development, these changed many times, reflecting the different trials done but ended up granting access from my domestic network and any of the Reply networks. As an alternative to AWS, a test with Azure'instance, free for a period of time thanks to my politecnico-granted subscription was done too. This was used by me as a second environment during the past months both as an ulterior test environment and to run a connectivity test to better understand the subject, but was later abandoned during the final stage when the PoC was ultimately completed on the AWS machine.

6.1.1 Minikube Setup

Once the machine was up, it was time to install minikube. This was simply done by following the official documentation [1]. As the reader will see in the link, minikube runs on a 'driver' which is basically any virtualization platform. As the suggested choice, I proceeded to install Docker Engine [2]. Once everything was installed and properly tested, it was time to go back to the drawing board. Different solutions were explored for setting up a multi-cluster test environment: The initial approach was the use of different instances, each running a minikube cluster. In this stage both the AWS and Azure machines were being used, and for a while this was the direction the majority of efforts went to. A number of problems in the later stages of development were enough to halt and change the approach. What was happening is that, as we will see in a bit, both machines needed a reverse proxy to connect the minikube network to the outside. Those two proxies complicated a lot of the management of connections between services from each cluster and while this was still manageable, it became nearly impossible to continue once the use of third-party programs became mandatory and the majority of time was being spent on workarounds instead than on topic-related research.

The solution that worked the best, and made the cut ultimately, was the use of two 'instances' of minikube, run on a single machine but on different docker networks. A docker network is a virtual network that comes with isolation and a 'strong' boundary, its own ID and DNS server, effectively behaving like a normal network would. This was perfect because it reproduced the real-world scenario in every aspect of interest, with the only drawback of being more resource intensive on

the host machine than it would have been otherwise. Some additional networking had to be done to allow traffic routing between the two networks, so once those were created, respectively cube1 and cube2, the appropriate rules to the iptables were added. The two instances of minikube were now ready to be run, and fortunately, an out-of-the-box feature provided by the software itself helped greatly in this: 'profiles'. By creating different profiles, two different clusters would be built, isolated from each other on the specified network and with the specified resources allocated to each. Adding on top of this, any interaction with both cluster was now possible using the same command line, just switching context (one associated with each profile). This simplified greatly the creation of a 'helper script' that will be presented in the last section of this chapter but ended up being effectively the most valuable asset in the testing stage of the thesis.

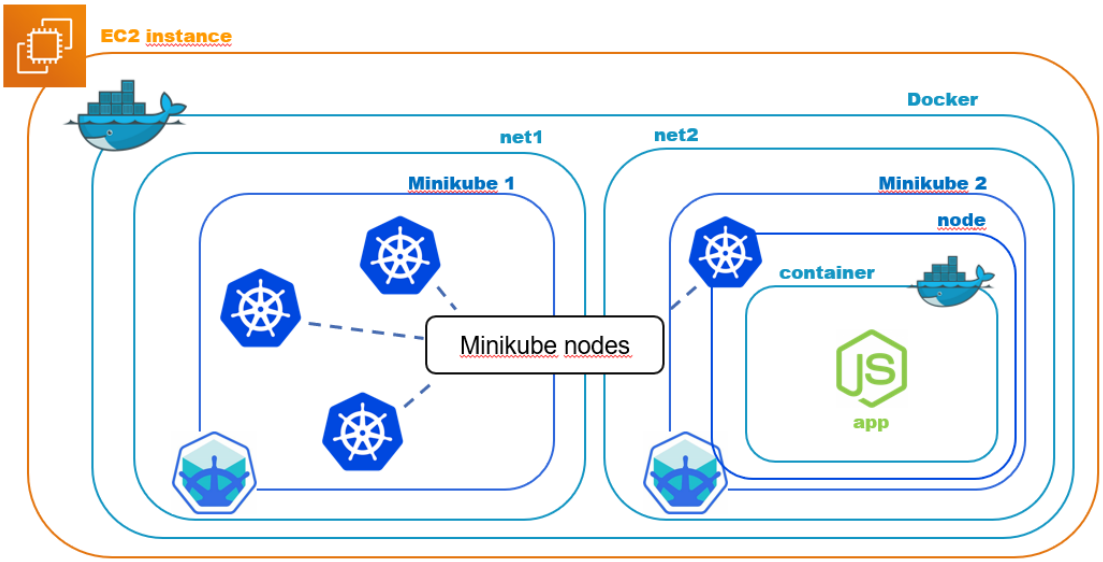


Figure 6.1: PoC Layers

6.1.2 Networking

After the two clusters were set up, it was time to study their connectivity to the 'outside' to be able to interact with the application from the machine itself and eventually from any other. This was not immediate, work had to be done to get in the desired situation. This is because of the way minikube works. Let's quickly take a step back though and go as 'inside' as possible. To connect to a pod in a cluster, a service is needed. Services can be of different types, but by default, they come with a ClusterIP. This means that they can only be reached from the cluster. To expose them to the outside, for example, to be able to curl their endpoint from a

terminal, a type of NodePort or LoadBalance is needed. I won't go into much detail on this section because it's all explained in any Kubernetes documentation, but what's important is that load balancers and nodeports need to be further exposed if running inside minikube. This is because the minikube cluster is running on a virtual machine (the single node-environment) and traffic needs redirection from the host machine to the appropriate services running inside the cluster. Fortunately, this is not hard and can be done using different command line options such as minikube tunnel and or minikube service <service name> ...

One of the major concern of the work was to make it as simple as possible to interface with it and standardize its execution. This meant that having to manually expose services for both clusters and have terminals dedicated to hosting was not ideal. Here a minikube plugin, **MetalLB** really shined. MetalLB is a software loadbalancer that automatically assign IP addresses to LoadBalancer and NodePorts services, reachable externally. Its configuration is done by using a simple YAML resource in which a range of IPs is specified. This proved to be a consistent way to have the needed services always have the same 'host-local' IP addresses and allowed to proceed to the next step.

```
server {
    listen 80;

    proxy_set_header X-Forwarded-For $proxy_protocol_addr;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header Host $host;


    location /auth {
      proxy_http_version 1.1;
      proxy_pass http://keycloak.com:8080;
    }

    location /kiali {
      proxy_http_version 1.1;
      proxy_pass http://kiali.app.cube2:20001;
    }

    location / {
      proxy_http_version 1.1;
      proxy_pass http://apptest.com;
    }
}
```

Figure 6.2: Nginx configuration

Once services were metalLB exposed, they could be reached from the host machine via the terminal without any problem. What was missing was the possibility of reaching them from another machine, to not only give the possibility to other members of the team to interact with the cluster, but also to get closer to the ultimate objective of having the application reachable from any browser. Traffic coming from another machine could only reach the EC2 instance via its IP, so a

way to forward this traffic to the right 'host-local' address was found in the use of a reverse-proxy. This is a proxy that listens on a specific port of a machine and routes requests according to configurations. **Nginx** was chosen as the specific product and installed directly on the machine via apt-get. Its configuration files are in /etc/nginx/sites-available, and its log files are in /var/logs/nginx/.

Something that can be noted is that the majority of HTTP requests are routed to the ingress Gateway of the application, /auth is redirected to the Auth provider and /kiali is used for monitoring purposes. After the deployment of Nginx, the final schema of the architecture can be reassumed as follows. All the details on the various architectural components present in the image, some of which have not yet been mentioned, will be discussed by the end of the chapter.

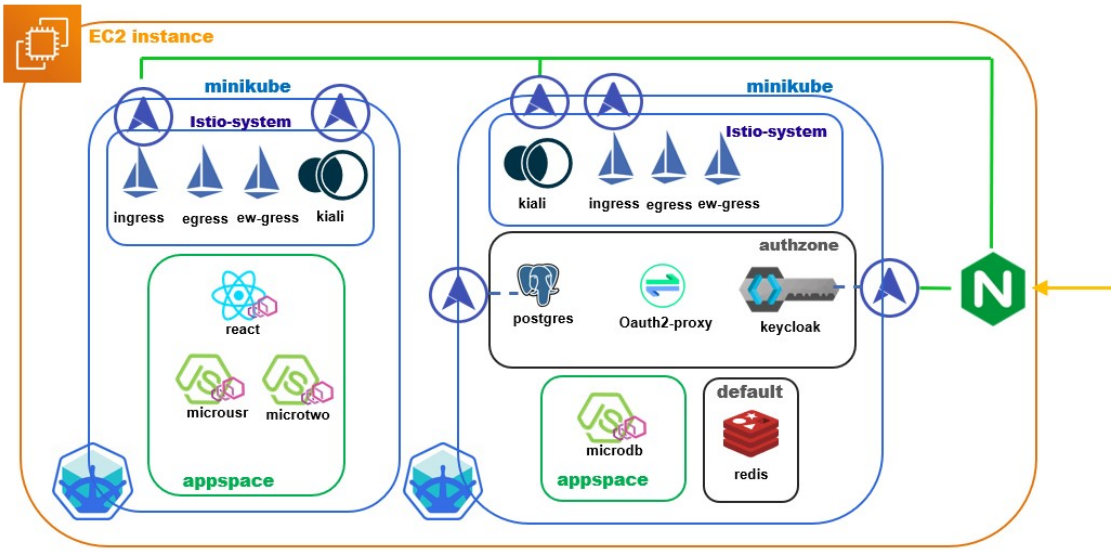


Figure 6.3: Full Architecture Schema

6.2 Architecture Implementation

6.2.1 Configuring Istio

Istio was installed initially on a single cluster following an online guide [3] to get some familiarity with its basic functions. The demo profile, suggested, already mounts Istiod (essential in primary installations), an Ingress Gateway and an Egress. This concept of modularity in the installation is something that later came back when I needed to use an 'operator', a resource specifically made to customize the installation.

```yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    accessLogFile: /dev/stdout
    extensionProviders:
    - name: "oauth2-proxy"
      envoyExtAuthzHttp:
        service: "oauth-proxy-service.authzone.svc.cluster.local"
        port: "4180" # The default port used by oauth2-proxy.
        includeHeadersInCheck: ["authorization", "cookie","x-forwarded-access-token","x-forwarded-user
        headersToUpstreamOnAllow: ["authorization", "path", "x-auth-request-user", "x-auth-request-ema
        headersToDownstreamOnDeny: ["content-type", "set-cookie"] # headers sent back to the client wh
  values:
      global:
        meshID: mesh1
        multiCluster:
          clusterName: cluster1
        network: network1
```

Figure 6.4: Istio Operator for Cluster 1

The figure shows the operator used to customize the installation of Istio in the first cluster, specifying parameters required for the multi-cluster setup and inserting in the mesh configuration an external authenticator, important for the authentication flow as explained later. For the multi-cluster installation, the official documentation [4] was used with the occasional git issue to fix different problems met along the way. One important step that was taken before the two installations was the creation of a root CA and two intermediate, one for each cluster, to later use as a secret 'cacert' for each installation. See here for more details. For different purposes, two additional parameters were set with the installation command, namely:

- values.global.proxy.privileged=true - required for analysis purposes
- meshConfig.outboundTrafficPolicy.mode=REGISTRY-ONLY - Imperative to block traffic towards the outside and use the egress gateway effectively

After the 'istio-system' namespace and its resources were automatically created (in consecutive iterations via the help of the script) on both clusters and the procedure was complete, a quick connectivity test [LINK] was run before proceeding to create the required namespaces for the application and authentication and label them as 'istio-injection=enabled', granting all their pods deployment to be injected with proxies and therefore effectively adding them to the mesh.

6.2.2 The Application

Before delving into the Istio's configurations, I'd like to spend a section on the Application's development and deployment. Even though the architecture itself can function without an application, it would not have been possible to conduct all the studies and showcase its potential without it. As mentioned in the overview chapter about it, the application is a series of microservices written in JavaScript. Its Client uses the React Framework, while the other services are all nodeJs instances. Everything was written using Visual Studio Code, connected to different target platforms, depending on the stage of the development. This is because initially the application was written with its code running inside a container, when it came to services WSL hosted it instead and finally, for the last changes, VS Code was directly connected to the EC2 Instance. The Code was kept as simple and straightforward as possible, presenting a repeating structure for each service consisting of a single main file, a SQL database and an API interface to it. The main application, aside from the auto-generated files, is composed by a file containing all the APIs and five main components:

- CreateTopic.js
- CreateUpdate.js
- UpdateList.js
- Topic.js - Includes UpdateList and CreateUpdate
- TopicList.js

Each microservice was then **dockerized**: a Dockerfile was written for them starting from a node:alpine image. In the image, the React's Dockerfile can be seen.

```
FROM node:alpine
LABEL project="mockapp"
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Figure 6.5: Dockerfile for the React Application

At this point, images could have been uploaded to Docker Hub, but they ended up being built locally directly. To deploy these services some additional resources

were needed, namely a deployment YAML, a Service and a Service Account (if needed). The Deployment resource contains the structure of the pod: its image, how many replicas, its ports and its service account. The Service resource is responsible for inter and intra-cluster connections. With the architecture in mind, the application will never be directly exposed to the outside but will always sit behind an ingress gateway, therefore all services will be of type ClusterIP. The Service Accounts are associated with specific services and are used to specify authorization policies regarding them. All three of those resources can be grouped in a single file by using separators '—' making for a cleaner way to organize files. One thing to note is that labels can be used to facilitate deploying only some of those resources contained in the same file if organized like this. This is helpful when deploying only the service that is needed, like we've seen is required in a multi-cluster environment. An example of these resources follows and is representative of all the others.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: db-acc
  labels:
    account: microdb
    svc: api-db
```

Figure 6.6: microdb ServiceAccount

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-db-service
  labels:
    svc: api-db
spec:
  selector:
    app: api-db
  ports:
    - name: http-db
      port: 3003
      targetPort: 3003
```

Figure 6.7: microdb Service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-db-deplo
  labels:
    app: api-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-db
  template:
    metadata:
      labels:
        app: api-db
    spec:
      serviceAccountName: db-acc
      containers:
      - name: api-db
        image: microdb-image
        imagePullPolicy: Never
        ports:
        - containerPort: 3003
        volumeMounts:
        - name: db-data
          mountPath: /app/db/microdb.db
      volumes:
      - name: db-data
        hostPath:
          path: /host/microdb/db/microdb.db
          type: File
```

Figure 6.8: microdb Deployment

The following is a screenshot of how the application looks in the browser and also gives a quick idea of how it works.

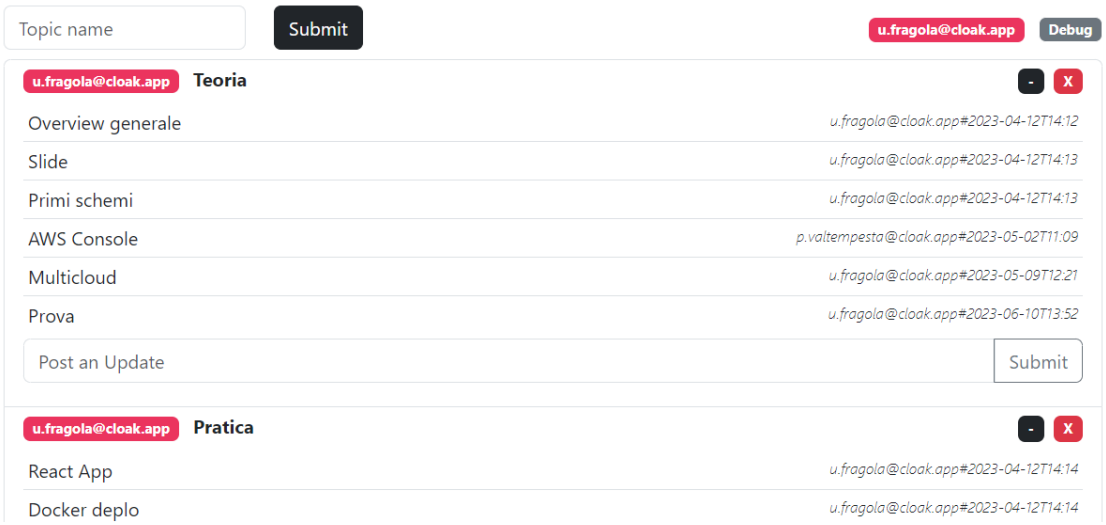


Figure 6.9: In-Browser Application Screenshot

6.2.3 Ingress Gateway

The ingress gateway is already provided with the default installation as a configurable component. It's exposed to the outside and is configurable with a Gateway Resource. It acts as an access point for the application and as such a virtual service is needed to route connection to their target services. The image shows the configuration and as it's apparent the prefix is the route on which the decision is taken and the destination specifies the target service and port.

When the authentication flow will be discussed, it will become apparent how this virtual service only acts after another type of routing has been done before. This means that when a request reaches the Ingress gateway, this will first be processed by the Auth Flow, and then it will pass the gateway.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-microvs
spec:
  hosts:
  - "*"
  gateways:
  - micro-gateway
  http:
  - match:
    - uri:
        prefix: "/topics"
    route:
    - destination:
        host: api-db-service.appspace.svc.cluster.local
        port:
          number: 3003
  - match:
    - uri:
        prefix: "/"
    route:
    - destination:
        host: react-service.appspace.svc.cluster.local
        port:
          number: 3000
  - match:
    - uri:
        prefix: "/userinfo"
    route:
    - destination:
        host: api-usr-service.appspace.svc.cluster.local
        port:
          number: 3005
  - match:
    - uri:
        prefix: "/oauth2"
    route:
    - destination:
        host: oauth-proxy-service.authzone.svc.cluster.local
        port:
          number: 4180
  - match:
    - uri:
        prefix: "/updates"
    route:
    - destination:
        host: api-two-service.appspace.svc.cluster.local
        port:
          number: 3002
```

Figure 6.10: Ingress Virtual Service

6.2.4 Intra-Service Policies

The authorization policies for inter and intra-cluster communications are implemented using the Authorization policies seen before. It's possible, as the image demonstrates, to specify the 'principal' or authenticated name to restrict access to only some other services. This implements the idea we discussed in the first chapters of services as 'entities'. The policies allow specifying, for each service, which other service can reach them, on which path and using which method. This allowed, by creating a resource for each service, to enforce the communication schema last chapter. It should also be noted that a PeerAuthentication resource was used to only allow mTLS connection on the entire application namespace and also a deny-all policy was enforced just for safety measures.

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "in-two"
  namespace: appspace
spec:
  selector:
    matchLabels:
      app: api-two
  rules:

  #Routes coming from microdb
  - from:
    - source:
        principals: ["cluster.local/ns/appspace/sa/db-acc"]
    to:
    - operation:
        paths: ["/updates/*"]
        methods: ["DELETE"]

  #Routes coming from the gate
  - from:
    - source:
        principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
    to:
    - operation:
        paths: ["/updates/*"]
        methods: ["GET", "POST"]
```

Figure 6.11: Basic Authorization Policy Example

The image shows The authorization policy applied to the microtwo service. Looking at the communication graph, we can see how microdb, its first allowed route (DELETE), is present on the second cluster. The policy works seamlessly because microdb's service is replicated in the local cluster and Istio takes care of managing cross-cluster communications on its own as mentioned in the theory section.

6.2.5 Auth Strategy Implementation

In the previous chapter, the general idea of what is the target result was introduced. In this section, we will expand on that view and detail the entire final implementation of the authentication flow. As mentioned above. To start things off, an authorization server is needed to manage identities and authenticate users for the purpose of providing a JWT Access token to use for authentication and authorization in the architecture. Two alternatives were considered in the analysis: Keycloak and PingFederate. The latter was introduced to me in a side project I helped with in my internship regarding a simplified version of this architecture on a single cluster. The former, Keycloak, was heavily suggested to me by my team for its widespread

use in the modern panorama, as Liquid confirmed based on its clients, and for the already present expertise in the company.

Keycloak is an open-source software product that enables SSO with IAM for modern applications and servers. It supports protocols such as SAML v2 and OpenID Connect (OIDC) / OAuth2. It's written in Java and supported by Red Hat. The software offers a number of features that I couldn't fully explore, as many of them weren't strictly correlated to what I needed to do. This in fact was only the creation of some users with different privileges to then use for policies' logic in the architecture. This server had to be accessible for both the user and the ingress, to one side provide the token, and the other confirm its authenticity. The deployment wasn't hard, given that a deployment resource was provided and its documentation [5] was sufficient to create users and assign them a role.

To keep consistency across multiple minikube resets so creating an actual persistence of data and allowing for eventual scalability, a **Postgres** database was used as a backing store for the server to store and fetch data. I deployed those two elements in a different namespace: the 'AuthZone' for the separation of concerns principle and to adhere to the high-level architectural view that was thought of during the design phase. One thing to note is that Keycloak interfaces with both the users and the inside of the mesh, therefore it cannot abide to a strict peer authentication policy unless the user traffic was to be routed through the ingress first. This is a possible configuration, but for the PoC it was thought sufficient to just leave it in a PERMISSIVE mode. The reason Keycloak needs to be accessible from inside the mesh is that when releasing tokens, the server signs them and therefore Istio or the interested party needs to verify the signature before considering it valid. To do so an endpoint is provided - the JWKS URI:

- http://<base-url>/realms/<realm>/protocol/<protocol>/certs

This is the location of the set of public keys used, containing also the one used to sign the JWT issued alongside others in some cases. For what concerns the user, the endpoint for getting a valid token is the following:

- http://<base-url>/realms/<realm>/protocol/<protocol>/token

A request to this endpoint requires various parameters in the body to be considered valid:

- client-id : ID of the target client inside the realm (appclient)
- grant-type: The type of access grant flow for authenticating the user (password)
- scope: the protocol used

- username
- password

The configuration effort, as anticipated was minimal: I created a client (appclient), which is a name to identify an application that Keycloak is serving, and I populated it with three different users with one of them having the PRIME role.

It was now time to use this in combination with Istio to restrict and grant access. For reasons that will become apparent in a bit, it was chosen to not directly apply the request authentication on the gateway but to instead move it directly to the service that needed protection. The two following figures show the Request Authentication and Authorization Policies that limit the DELETE route to only users with the PRIME role. One thing immediately noticeable is how the value of the jwksUri is hard-coded, while the issuer is not. This is because while Keycloak can always be reached on a cluster address by entities inside the cluster, the issuer is programmed to correspond to the user-exposed interface, reachable from the browser and therefore linked with the machine IP. Given that this IP changes every machine reset, the value cannot be hard-coded (with this configuration) and will be automatically updated by the script during the setup phase.

```yaml
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: jwt-db-req
  namespace: appspace
spec:
  selector:
    matchLabels:
      app: api-db
  jwtRules:
  - issuer: http://{{MY_SERVICE_IP}}/auth/realms/appcloak
    jwksUri: http://keycloak-service.authzone.svc.cluster.local:8080/auth/realms/appcloak/protocol/openid-connect/certs
    forwardOriginalToken: true
    fromHeaders:
      - name: x-auth-request-access-token
```

Figure 6.12: Request Authentication

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt-db
  namespace: appspace
spec:
  selector:
    matchLabels:
      app: api-db
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
    to:
    - operation:
        paths: ["/topics/*"]
        methods: ["DELETE"]
    when:
    - key: request.auth.claims[resource_access][appclient][roles]
      values: ["PRIME"]
```

Figure 6.13: Authorization Policy checking JWT field

With this setup, a working auth has been reached. The only problem is that it's not user-friendly in any way. For the target result, more was wanted: a redirect to have anyone trying to connect directly moved to the Keycloak login page and blocked from proceeding further unless this was completed. Here comes the other major part of the Auth Flow: the **oauth2-proxy**. This is another open-source project that acts as reverse proxy and static file server, providing authentication using different supported providers. In this work, this software was used with Istio's **External Auth Provider** feature serving as an entity to delegate an authorization decision to.

```yaml
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-in-gate"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: CUSTOM
  provider:
    name: oauth2-proxy
  rules:
    - to:
      - operation:
          notPaths: ["/oauth2/*", "/sockjs-node/*"]
          paths: ["/*"]
```

Figure 6.14: CUSTOM action Authorization Policy

The proxy acts as an OAuth Client, detects if incoming requests are authenticated or not, and orchestrates the authentication of the user with the configured OAuth server. Going into more details: When a request from Istio's Ingress Gateways is received, a check on its cookie is done to see if the authentication already took place. If not, an Oauth2.0 Auth flow [6] is initiated by redirecting the user to Keycloak's login endpoint. After this has been completed, Oauth2-proxy stores the access token alongside the ID token, refresh token (if enabled) and session state in its internal storage. It then responds to the Ingress Gateway with an HTTP 200 OK response including two important headers:

- Authorization header - containing the ID token
- X-Auth-Request-Access-Token header - containing the Access Token, what we are interested in and check in the authorization policy as seen above.

When a request comes with an already valid cookie to the gate, this is still forwarded to the proxy, but this time this will promptly respond with an HTTP 200OK response including the two headers mentioned above. Adding on top of this, if refresh tokens are enabled in Keycloak, the proxy will handle the automatic refreshing of access tokens at regular customizable intervals, completely eliminating the need for a user to repeat authentication for the allowed window of time. One thing to mention is that to help reduce the size of the cookie and achieve better performance, a Redis instance was deployed as support for the proxy.

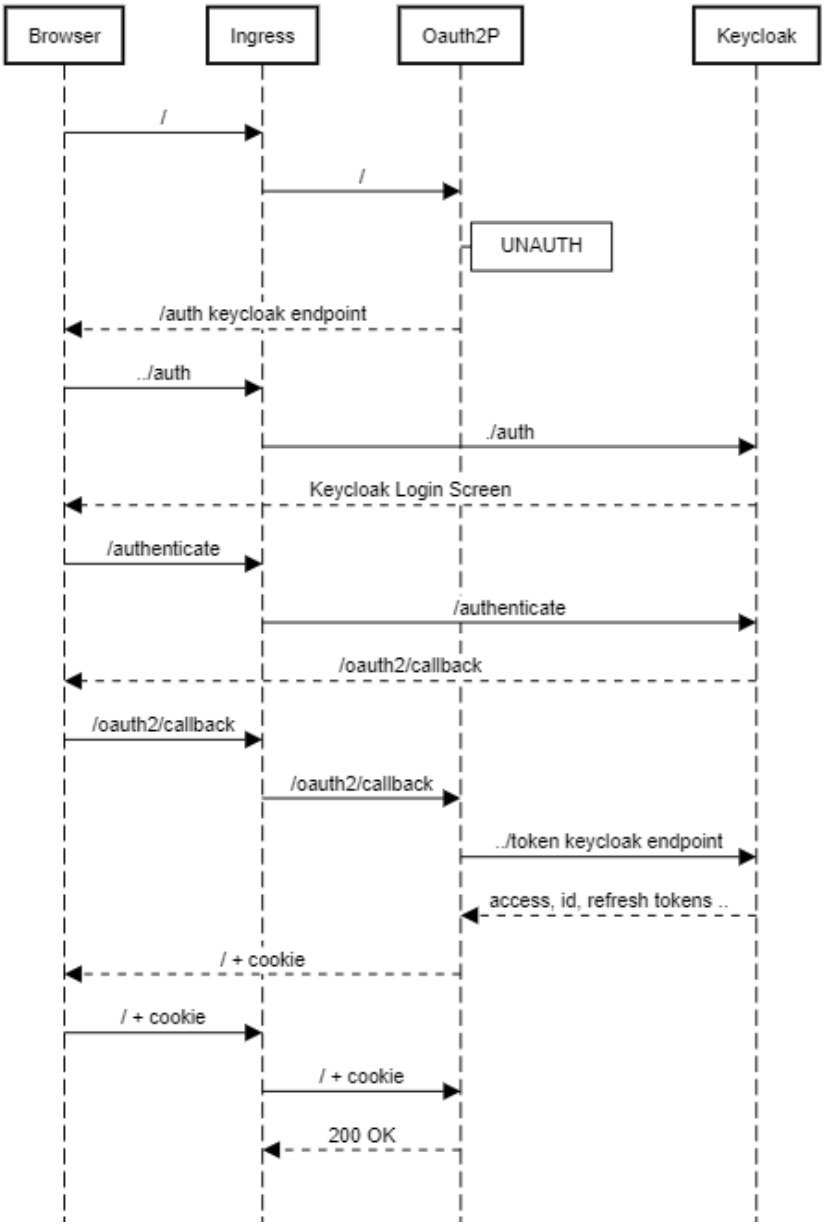


Figure 6.15: Full Oauth2-Proxy Sequence Diagram

6.2.6 The Egress Gateway

Like the Ingress Gateway, the main resource used to configure the egress gateway is a simple Gateway resource, that is very much similar to the one used to configure the ingress presented in the introduction to Istio's resources chapter. The interesting aspect of the Egress is that Both a DestinationRule and a Virtual Service

are required for it to be defined properly. The VirtualService, in particular, is responsible for two different connections: From within the mesh to the gateway and from the gateway to the external service. When Istio's installation was documented, it was outlined how the OutboundTrafficPolicy's mode was to be set to registry only. This means that only services registered with a Service Entry resource which defines host, port, protocol and how to resolve the host, are allowed. As the last piece needed, the Service Entry for timeapi.io was created.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: through-egress-gateway
  namespace: istio-system
spec:
  hosts:
  - timeapi.io
  gateways:
  - mesh
  - exit-gateway
  tls:
  - match:
    - gateways:
      - mesh
      port: 443
      sniHosts:
      - timeapi.io
    route:
    - destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
        subset: timeapi-ext
        port:
          number: 443
  - match:
    - gateways:
      - exit-gateway
      port: 443
      sniHosts:
      - timeapi.io
    route:
    - destination:
        host: timeapi.io
        port:
          number: 443
      weight: 100
```

Figure 6.16: Egress Virtual Service

6.3 The "Helper" Script

The entire work was done, mainly in its later stages, on the EC2 machine as explained in the first part of the chapter. This meant that basically each day the machine was turned on and off, sometimes even at different times during the same day. Even when not turning the machine off, many times, the entire cluster

was deleted and recreated to experiment with different installation configurations, new components, or even try out variations of the architecture. This, over the entire period of time the thesis was carried out, made for hundreds of resets that, especially near its conclusion, would have seen me repeat an enormous amount of command line code each time.

To expedite the process and allow for testing various configurations, I began developing a script alongside my work on the cloud machine. Over time, this script grew in complexity and became an invaluable tool for advancing my work. In fact, it significantly reduced the setup time, giving me more available time to focus on my research. Ultimately, this script played, I'd say, a pivotal role in propelling my thesis forward. The second-biggest advantage the script offered was the possibility of easily running different versions of the project by specifying parameters on its execution. In the early days, the script would allow, for example, the flag –multi to build a multi-cluster architecture, defaulting to a single cluster. In its current version, the script assumes the multi-cluster installation as its only viable option and offers other flags like:

- –app - to mount the sample application
- –test - to verify and test the built environment
- –kiali - to mount the add-ons (Prometheus and Kiali) necessary for monitoring purposes

One last important thing the script allowed me to do, was to dynamically adapt the code, in some of its sections, to the ever-changing machine's IP address. Let's have a look at some of its sections.

The first thing the script does is the initialization of the Docker Networks and the creation of the required Intermediate CAs. The intermediate lines modify kernel parameters related to inotify, which is responsible for monitoring file system events. This is needed to prevent crashes when running Minikube with many pods, which was happening when running the two clusters, by allowing for a larger number of file system events to be handled simultaneously.

```
...
# Calls scripts/networksetup.sh which does
docker network create -d bridge --opt \
    com.docker.network.bridge.name=cube1-net cube1 \
    --subnet=192.168.49.0/24 --gateway=192.168.49.1
docker network create -d bridge --opt \
    com.docker.network.bridge.name=cube2-net cube2 \
    --subnet=192.168.58.0/24 --gateway=192.168.58.1
sudo iptables -I DOCKER-USER -i "cube1-net" -o "cube2-net" -j ACCEPT
```

```
sudo iptables -I DOCKER-USER -i "cube2-net" -o "cube1-net" -j ACCEPT
...
sudo sysctl fs.inotify.max_user_watches=655360
sudo sysctl fs.inotify.max_user_instances=1280
...
make -f istio-1.17/tools/certs/Makefile.selfsigned.mk root-ca
make -f istio-1.17/tools/certs/Makefile.selfsigned.mk cluster1-cacerts
make -f istio-1.17/tools/certs/Makefile.selfsigned.mk cluster2-cacerts
...
```

Minikube's first profile is then created, MetalLB configured for it and dynamic entries fixed.

```
...
minikube start --mount-string=/home/admar/first/app:/host --mount \
  --service-cluster-ip-range='10.96.0.0/12' \
  --apiserver-ips 192.168.49.2 \
  --cpus 4 --memory 6000 \
  --profile cube1
...
minikube addons enable metallb --profile='cube1'
kubectl apply -f res/metal.yaml --context='cube1'
...
scripts/dynfix.sh $MY_SERVICE_IP
```

Here's a snippet of what dynfix does: like mention previously it makes it so that code that depends on the machine's IP address is 'fixed' starting from a reusable template.

dynfix.sh

```
...
cp res/deplo/oauth2-tmp.yaml res/deplo/oauth2.yaml
cp res/istio/jwtrules-tmp.yaml res/istio/jwtrules.yaml
sed -i "s/{{MY_SERVICE_IP}}/$MY_SERVICE_IP/g" res/deplo/oauth2.yaml
sed -i "s/{{MY_SERVICE_IP}}/$MY_SERVICE_IP/g" res/istio/jwtrules.yaml
...
```

What follows is the installation of Istio in the first cluster and its multi-cluster predisposition.

```
...
kubectl label namespace \
    istio-system topology.istio.io/network=network1 --context='cube1'
istioctl install -y \
  --set profile=demo \
```

```
  --set values.global.proxy.privileged=true \
  --set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY \
  --context='cube1' \
  -f res/istio/operator.yaml
...
#scripts/multicluster.sh is called which does:
scripts/istio-scripts/gen-eastwest-gateway.sh \
--mesh mesh1 --cluster cluster1 --network network1 | \
istioctl install --context='cube1' -y -f -
kubectl apply -n istio-system --context='cube1' \
-f scripts/istio-scripts/expose-services.yaml
```

This finishes the preparation work on the first cluster. For what concerns the second, a very similar command structure is followed, starting from the minikube profile creation, metalLB configuration and proceeding with both the Istio's installation and multicluster configuration like we've seen similarly done above. An important step that follows this is the secret exchange between the two clusters.

```
...
istioctl x create-remote-secret \
  --context='cube2' \
  --name=cluster2 | \
  kubectl apply -f - --context='cube1'
istioctl x create-remote-secret \
  --context='cube1' \
  --name=cluster1 | \
  kubectl apply -f - --context='cube2'
...
```

If the tag for mounting the application has been used, the app's installation will now take place in the appmount.sh script. This one, in order:

1. Sets up namaespaces and labels in both cluster required for the application and authzone deployments
2. Creates a persistent volume for postgres
3. Builds all the Docker Images
4. Applies all the YAML files and Mirrors Services on their opposite cluster if needed
5. Configures all the Istio's policies specific for the application and non.
6. Initializes Postgres with saved data

We've pretty much covered all the points presented previously aside from point 6.

```
#Sets up ENV VARIABLES {...}
...
#Waits for required Pods to be deployed and ready {...}
...
#Updates Postgres
psql -h $POSTGRESIP -p 5432 -d postgres -U p-user \
    -c "DROP DATABASE keycloak"
psql -h $POSTGRESIP -p 5432 -d postgres -U p-user \
    -c "CREATE DATABASE keycloak"
psql -h $POSTGRESIP -p 5432 -d keycloak -U p-user < res/sqlcloak.db
```

What's happening is that Postgres is first 'cleared' from previous Keycloak tables, in case the command is executed in runtime, then it's populated with a Keycloak's DB dump generated previously (A 'clean' state). If the flag –kiali is passed the script will now install those tools, likewise if –test is an argument it will start running tests on the environment. Those tests will be the focus of the next chapter, so we will not discuss them here but still present their code as a reference.

Click here to see results for the code below.

multicluster verification

```
#Initialization phase for required resources {...}
...
P1=$(kubectl get pods -n sample -l "app=sleep" \
    --context='cube1' -o jsonpath='{.items[0].metadata.name}')
P2=$(kubectl get pods -n sample -l "app=sleep" \
    --context='cube2' -o jsonpath='{.items[0].metadata.name}')
...
echo "sleep (cube1) -> helloworld service"
for i in $(seq 10); do kubectl --context='cube1' \
    -n sample exec $P1 -c sleep -- curl -s helloworld:5000/hello; done

echo "sleep (cube2) -> helloworld service"
for i in $(seq 10); do kubectl --context='cube2' \
    -n sample exec $P2 -c sleep -- curl -s helloworld:5000/hello; done
```

Click here to see results for the code below.

Visibility

```
echo "probe.default.cube1 -> microdb.appspace.cube2"
kubectl --context='cube1' -n default \
    exec $PROBE_DEF -- \
    curl api-db-service.appspace.svc.cluster.local:3003/debug
```

Click here to see results for the code snippet below.

Strict

```
echo "probe.default.cube1 -> micro-two.appspace.cube1"
kubectl --context='cube1' -n default \
    exec $PROBE_DEF -- \
    curl api-two-service.appspace.svc.cluster.local:3002/updates/7..6

echo "probe.appspace.cube1 -> micro-two.appspace.cube1"
kubectl --context='cube1' -n appspace \
    exec $PROBE_APP -- sh -c \
    "wget -qO- \
    api-two-service.appspace.svc.cluster.local:3002/updates/7..6 \
    | jq -C"
```

Click here to see results for the code snippet below.

Intra-Service Policies

```
echo "probe.appspace.cube1 -> microdb.appspace.cube2"
kubectl --context='cube1' -n appspace \
     exec $PROBE_APP -- \
     curl api-db-service.appspace.svc.cluster.local:3003/debug

echo "----------------Allowing route---------------"
kubectl apply -f res/istio/test-pol.yaml --context='cube2'
kubectl --context='cube1' -n appspace \
     exec $PROBE_APP -- \
     curl api-db-service.appspace.svc.cluster.local:3003/debug
```

Click here to see results for the code snippet below.

testdd.sh

```

test_connection() {
    source_pod=$1
    endpoint=$2
    context=$3
    method=$4
    ...
    response=$(kubectl exec -n appspace --context=$context \
    $source_pod -- curl -s -o /dev/null -w "%{http_code}" \
        -X $method http://$endpoint)
    ...
    if [ $response -eq 200 ]; then
        echo "Connection ACCEPTED"
    else
```

```
        echo "Connection DENIED"
    fi
}
# Define endpoints {...}
# Get the list of pods {...}
...
for source_pod in $pods; do
    for endpoint in "${svc_endpoints_get[@]}"; do
        test_connection $source_pod $endpoint $context 'GET'
    done
    for endpoint in "${svc_endpoints_get[@]}"; do
        test_connection $source_pod $endpoint $context 'DELETE'
    done
    ...
done
```

One last useful part of the script is the module that removes the clusters' installations and cleans everything. This is not included in the main script, but is instead another short and easy one. For completeness, a snippet of its code is reported.

mkteardown.sh

```
echo "Deleting minikube instances..."
minikube stop --profile='cube1'
minikube delete --profile='cube1'
minikube stop --profile='cube2'
minikube delete --profile='cube2'

echo "Removing docker networks..."
docker network rm cube1
docker network rm cube2
sudo systemctl reload docker
```

Chapter 7

Validation and Results

This chapter will serve the purpose of analyzing the created architecture and summing up the goals reached. Firstly, we will look at the inter-cluster connectivity and if it is working properly. Once that is done, we will test whether the Zero Trust Paradigm was applied correctly and hopefully add some additional insight into the process. In the following sections, some tests will see the use of additional components, not present in the presented architecture, as helpers, to create 'ad hoc' study situations. We are mainly talking about a 'probe' pod, a custom deployment mounting some simple analysis tools and command line utilities.

7.1 Mesh-granted Service Discovery

First of all, when we introduced the challenges of a multi-cloud environment, we outlined the issue of service discovery. As a quick recap: two services in two different clusters cannot reach each other without a third party's help. One of the service mesh's advantages is the capability to do just that, therefore, in this first section a test on that will be done. A probe pod is deployed in the default namespace of the first cluster and a request is tried to be sent to the microdb pod in the second cluster. The key thing to note here is that being in the default namespace, the probe pod, in this instance, is not associated with an envoy proxy and therefore is outside the mesh. The figure shows how the address cannot be resolved. A working scenario is not directly demonstrated here, as it can be found in any of the following test cases, for reference, using the same pods (both inside the mesh) it can be seen in the second case of Figure 7.6.

```
-----------------------------------------------
probe.default.cube1 -> microdb.appspace.cube2
-----------------------------------------------
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
curl: (7) Failed to connect to api-db-service.appspace.svc.cluster.local port 3003 after 6 ms: Couldn't connect to server
command terminated with exit code 7
```

Figure 7.1: Test n°1: Discovery

7.2 Testing Load Balancing capabilities

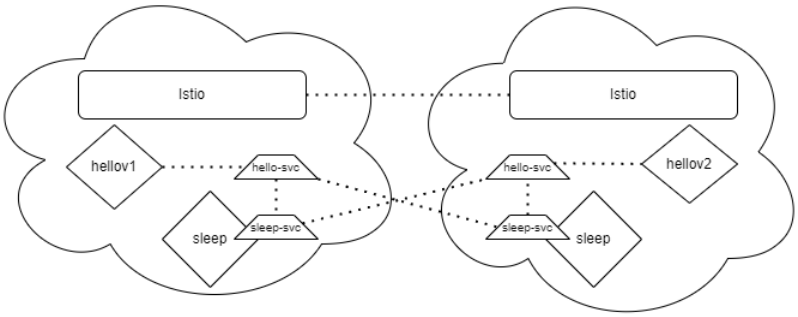


Figure 7.2: Schema of the Test

The following test is provided by Istio's documentation [7]. Its purpose is to test the multi-cluster installation to see if it's working properly. This is done by testing connections from cluster to cluster and vice versa and also, at the same time, load-balancing capabilities: A different version of an HelloWorld application is going to be deployed on both clusters in a sample namespace. Sharing the same service, upon receiving a request, one of the two is going to answer with its version, showing, ideally, how the service load-balances requests correctly. From the image it's apparent how in both sides, over 10 requests those are distributed equally between the two services, proving not only that connectivity is working as expected but also that the load-balancing capabilities advertised do in fact work.

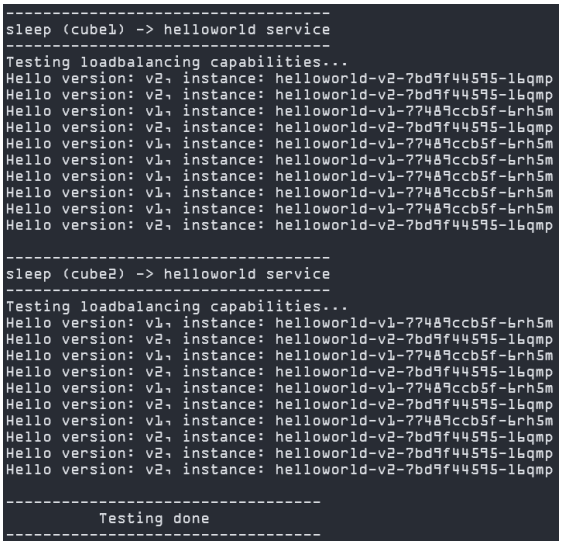


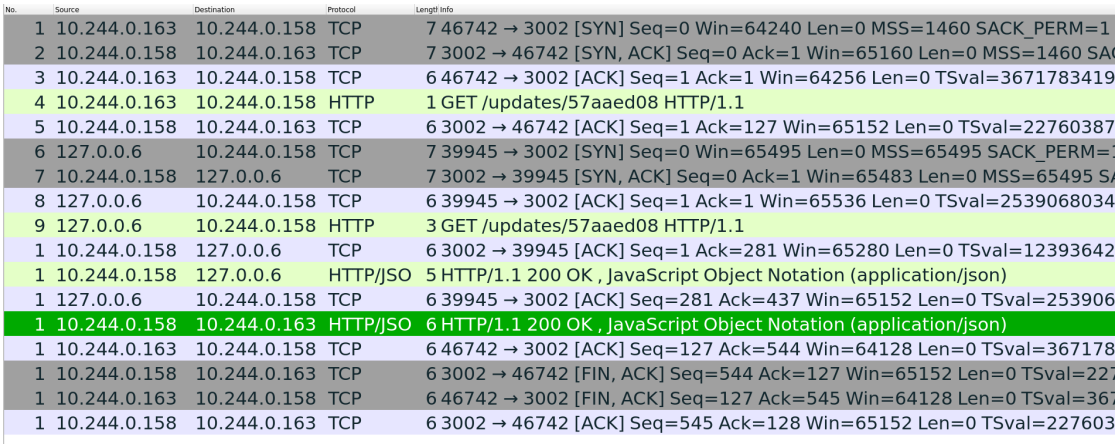
Figure 7.3: Test n°2: Multi-cluster LoadBalancing/Connectivity

7.3 Zero Trust Paradigm Correctness

When The Zero Trust Paradigm was introduced in Chapter 3, four principles were defined. We can say that we reached centralized management considering that we effectively applied configurations from a single point, the control plane, that were then used to enforce the whole infrastructural security policies. For the other three points, some tests and additional studies can be done to produce concrete results and certify our claims.

7.3.1 Verify Always: Looking at two traffic samples

Let's start with the first principle. In every communication inside the mesh, proxies are the real actors, and they verify by default the Istio's auth policies before putting their respective services in contact. What's more is that they use mutual TLS based on Istio's issued certificates called SPIFFE, basically x509 certificates. We can verify this by sniffing traffic between two services with a combination of the ksniff [8] plugin and Wireshark, and look at it closely. We'll do this for two different captures and compare results. The traffic will be captured by an additional container injected inside the microtwo pod, so the following 'perspectives' will be from that side to get as much information as possible. The first traffic capture is on a request made by a Probe deployed in a default namespace towards a Service Inside the mesh. This is allowed via a temporary Permissive policy and is expected to be readable as not TLS is being applied.



No.	Source	Destination	Protocol	Length	Info
1	10.244.0.163	10.244.0.158	TCP	74	46742 → 3002 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
2	10.244.0.158	10.244.0.163	TCP	74	3002 → 46742 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SA
3	10.244.0.163	10.244.0.158	TCP	66	46742 → 3002 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3671783419
4	10.244.0.163	10.244.0.158	HTTP	141	GET /updates/57aaed08 HTTP/1.1
5	10.244.0.158	10.244.0.163	TCP	66	3002 → 46742 [ACK] Seq=1 Ack=127 Win=65152 Len=0 TSval=22760387
6	127.0.0.6	10.244.0.158	TCP	74	39945 → 3002 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=
7	10.244.0.158	127.0.0.6	TCP	74	3002 → 39945 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SA
8	127.0.0.6	10.244.0.158	TCP	66	39945 → 3002 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2539068034
9	127.0.0.6	10.244.0.158	HTTP	355	GET /updates/57aaed08 HTTP/1.1
1	10.244.0.158	127.0.0.6	TCP	66	3002 → 39945 [ACK] Seq=1 Ack=281 Win=65280 Len=0 TSval=12393642
1	10.244.0.158	127.0.0.6	HTTP/JSO	515	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
1	127.0.0.6	10.244.0.158	TCP	66	39945 → 3002 [ACK] Seq=281 Ack=437 Win=65152 Len=0 TSval=253906
1	10.244.0.158	10.244.0.163	HTTP/JSO	566	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
1	10.244.0.163	10.244.0.158	TCP	66	46742 → 3002 [ACK] Seq=127 Ack=544 Win=64128 Len=0 TSval=367178
1	10.244.0.158	10.244.0.163	TCP	66	3002 → 46742 [FIN, ACK] Seq=544 Ack=127 Win=65152 Len=0 TSval=22
1	10.244.0.163	10.244.0.158	TCP	66	46742 → 3002 [FIN, ACK] Seq=127 Ack=545 Win=64128 Len=0 TSval=36
1	10.244.0.158	10.244.0.163	TCP	66	3002 → 46742 [ACK] Seq=545 Ack=128 Win=65152 Len=0 TSval=227603

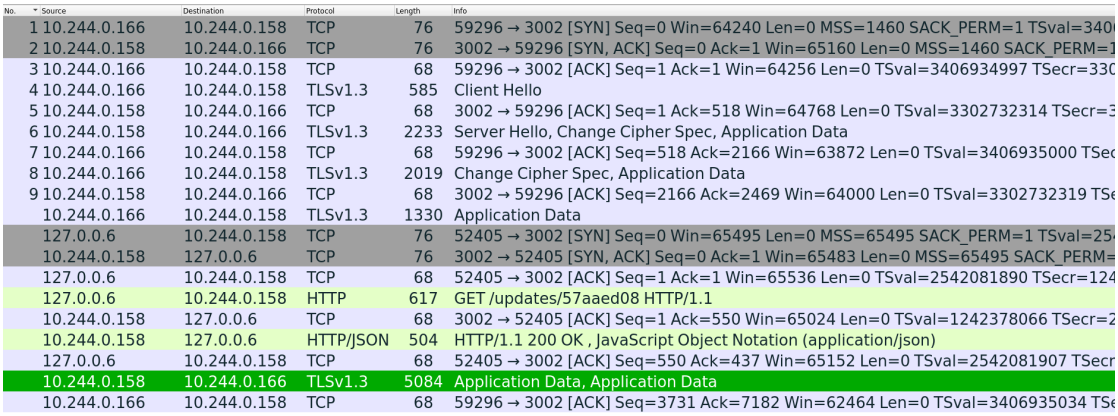
Figure 7.4: Wireshark Capture Case 1: Standard

Let's break down what's happening in the image:

1. n[1-3] Simple TCP handshake between the probe and the api-two's envoy
2. n[4-5] Clear HTTP request probe → api-two
3. n[6-8] TCP Handshake between envoy and container using loopback address
4. n[9-10] HTTP Request forward
5. n[11-12] Response container → proxy
6. n[13-14] HTTP Response forwarded by the proxy **in clear** to the probe
7. n[15-17] TCP Connection termination

Two things are important here: the first one, which we'll see happen in the other case too, is that the proxy and the container talk to each other using the loopback address. The second one, important to us for this analysis, is the answer from the proxy to the probe, highlighted in green in the image. This answer is in clear and completely readable by anyone.

This second capture shows a communication between the probe service, which has been moved inside the mesh this time, and the same micro-two service used before.



No.	Source	Destination	Protocol	Length	Info
1	10.244.0.166	10.244.0.158	TCP	76	59296 → 3002 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=340
2	10.244.0.158	10.244.0.166	TCP	76	3002 → 59296 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1
3	10.244.0.166	10.244.0.158	TCP	68	59296 → 3002 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3406934997 TSecr=330
4	10.244.0.166	10.244.0.158	TLSv1.3	585	Client Hello
5	10.244.0.158	10.244.0.166	TCP	68	3002 → 59296 [ACK] Seq=1 Ack=518 Win=64768 Len=0 TSval=3302732314 TSecr=3
6	10.244.0.158	10.244.0.166	TLSv1.3	2233	Server Hello, Change Cipher Spec, Application Data
7	10.244.0.166	10.244.0.158	TCP	68	59296 → 3002 [ACK] Seq=518 Ack=2166 Win=63872 Len=0 TSval=3406935000 TSec
8	10.244.0.166	10.244.0.158	TLSv1.3	2019	Change Cipher Spec, Application Data
9	10.244.0.158	10.244.0.166	TCP	68	3002 → 59296 [ACK] Seq=2166 Ack=2469 Win=64000 Len=0 TSval=3302732319 TSe
	10.244.0.166	10.244.0.158	TLSv1.3	1330	Application Data
	127.0.0.6	10.244.0.158	TCP	76	52405 → 3002 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=25
	10.244.0.158	127.0.0.6	TCP	76	3002 → 52405 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=
	127.0.0.6	10.244.0.158	TCP	68	52405 → 3002 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2542081890 TSecr=124
	127.0.0.6	10.244.0.158	HTTP	617	GET /updates/57aaed08 HTTP/1.1
	10.244.0.158	127.0.0.6	TCP	68	3002 → 52405 [ACK] Seq=1 Ack=550 Win=65024 Len=0 TSval=1242378066 TSecr=2
	10.244.0.158	127.0.0.6	HTTP/JSON	504	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
	127.0.0.6	10.244.0.158	TCP	68	52405 → 3002 [ACK] Seq=550 Ack=437 Win=65152 Len=0 TSval=2542081907 TSecr
	10.244.0.158	10.244.0.166	TLSv1.3	5084	Application Data, Application Data
	10.244.0.166	10.244.0.158	TCP	68	59296 → 3002 [ACK] Seq=3731 Ack=7182 Win=62464 Len=0 TSval=3406935034 TSe

Figure 7.5: Wireshark Capture Case 2: In-Mesh

Immediately the image looks way different from before, but let's break it down further before making additional considerations:

1. n[1-3] TCP handshake between the two services
2. n[4-10] TLS Setup + Request sent as Application Data in the last packet
3. n[11-13] TCP Handshake between the api-two's envoy and the container
4. n[14-15] HTTP request forward
5. n[16-17] Response container → Proxy
6. n[18-19] HTTP Response forwarded by the proxy as Application data

Firstly, the moment the communication was started, a TLS channel was created. As an outsider, the request is indecipherable as it is App Data and not in clear as before. Internally, between the proxy and the container, the situation is the same as before but, this time, the response is not sent back in clear to the probe service but is instead encrypted and understandable only to the TLS parties.

These captures show only a segment of the possible communications, but the situation is the same for any other. This proves that services inside the mesh, always verify each other before creating a connection and therefore in each communication both parties know and are sure of each other's identity as is required for the verification principle. Setting the peerAuth policy to strict makes it so that no cases like the one in the first image can happen at any time, once this is proven we can be sure this point has been properly validated. Following is a test in which a request is sent from the Probe pod to the micro-two service with a strict policy enforced. In the first case, the probe is not inside the mesh, while in the second it

is. As expected, the strict policy makes it necessary to create a mTLS channel for communication to happen. Point proven.

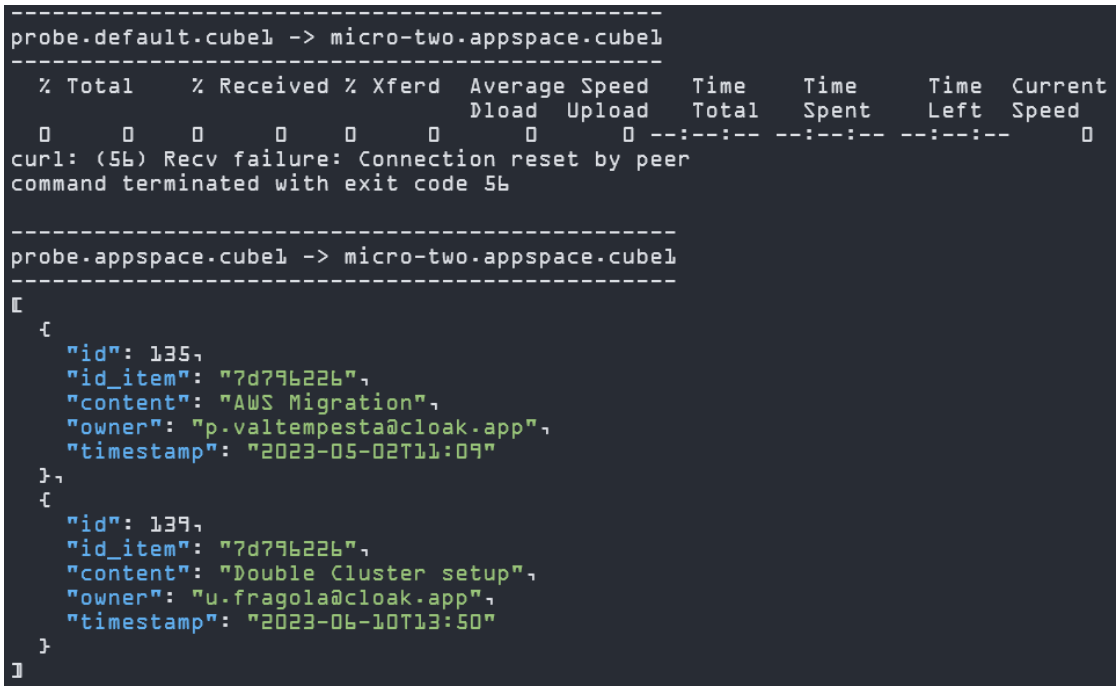


Figure 7.6: Test n°3: The Strict Policy

7.3.2 Least privilege and default deny: RBAC tests

This principle requires that for every component, only the information and resources that are necessary for its purposes should be reachable. In the Intra-Service Policies section of last chapter, we stated that every microservice is only allowed to interact with those that are required for the application structure. This implies that, for example, if we take the microdb service, it will only accept traffic from the ingress gate. Let's deploy the probe pod without changing any policy and send a request to microdb. Then let's create a policy and do that again. The results in the image show how the request is denied in the first case, while the expected response in the absence of a token is received in the second case.

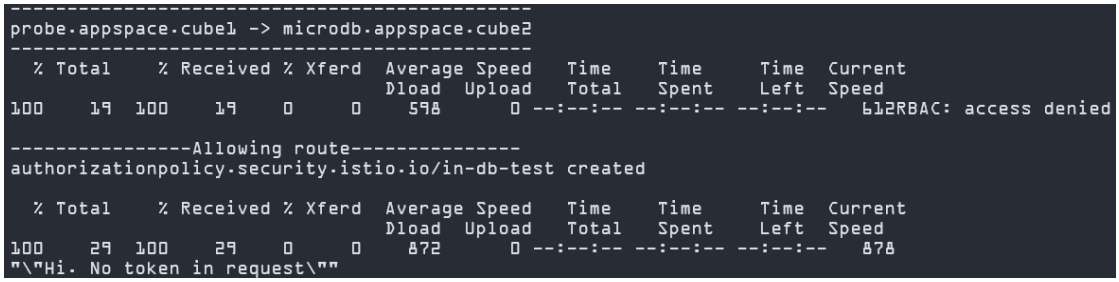


Figure 7.7: Test n°4: RBAC

As mentioned in the previous chapter, a 'deny-all' policy has been applied to both the 'appspace' namespaces. This, combined with the above, is enough to prove the desired least privilege/default deny principle has been successfully satisfied. A test on all the possible connections between the different 'appspace' services has been conducted, and a cut output can be seen here with the two allowed routes highlighted. To conclude this subsection, a test on the JWT route check is needed.

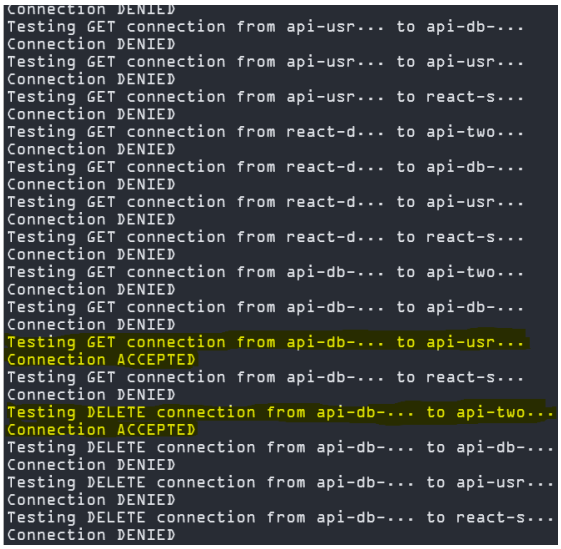


Figure 7.8: Test n°5: Default Deny

By using the wrong JWT, be they expired or with a different issuer, a simple message stating the error will be returned. The way I wanted to show this test instead is directly by showing the real use case inside the browser of a user trying to delete a topic. We know that every request that doesn't match an ALLOW policy is denied, so if someone unauthorized were to try to DELETE a topic, he should be stopped and the request denied. In the images below, two users will attempt the task, with one of them being PRIME and therefore allowed and the

other being a normal user. For commodity reasons, the debug button has been pressed in each case showing the JWT carried with the request. The test returns the expected results, concluding the verification on this principle point.

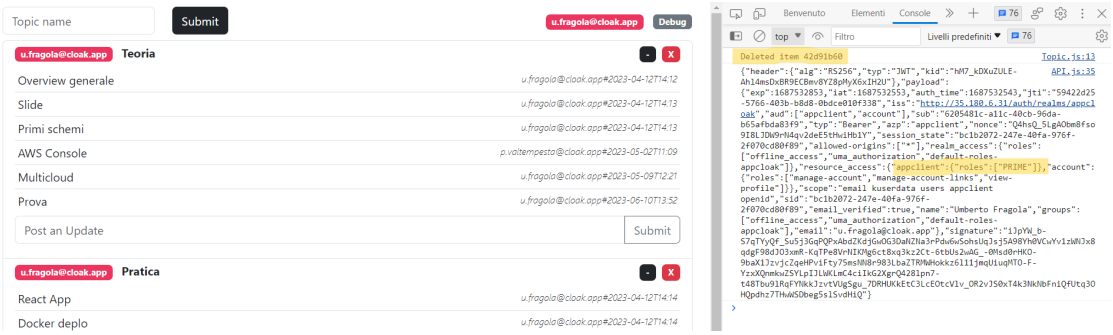


Figure 7.9: Test n°6: Case 1 - DELETE Allowed

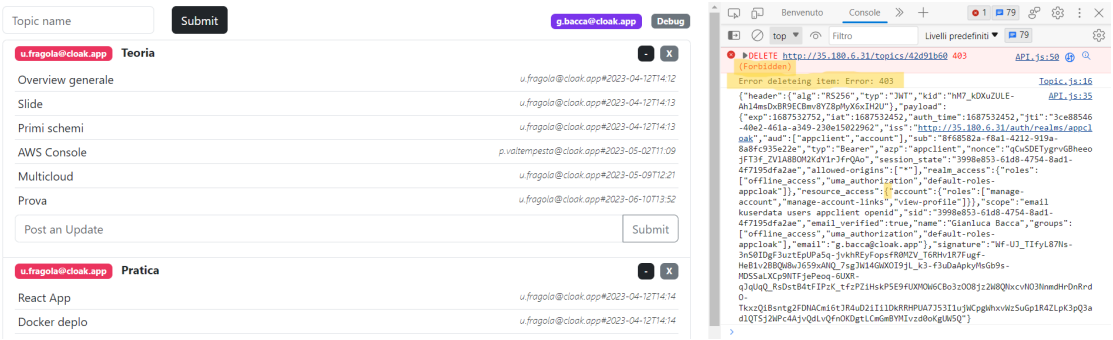


Figure 7.10: Test n°6: Case 2 - DELETE Denied

7.3.3 Visibility: A look at the kiali console

The last principle is one of the main advantages that come with the usage of service meshes. The visibility features are made possible by the proxies that intercept every request, allowing the logging and elaboration of metrics for a custom dashboard to organize and present. This is the case of Istio's suggested add-ons like Prometheus and Kiali. The main interest in this work was towards the latter, a dashboard that not only gives a very clear overview of the mesh but can also be used to investigate connections and the health status of the services. Kiali is an incredibly powerful tool but unfortunately as of the time this thesis is being written, still doesn't offer complete support towards multi-cluster solutions, with many of its features being in a beta state and definitely not ready for a production environment. The image below shows a screenshot of an instance of kiali running on the second cluster that highlights the traffic exchanged up to that point.

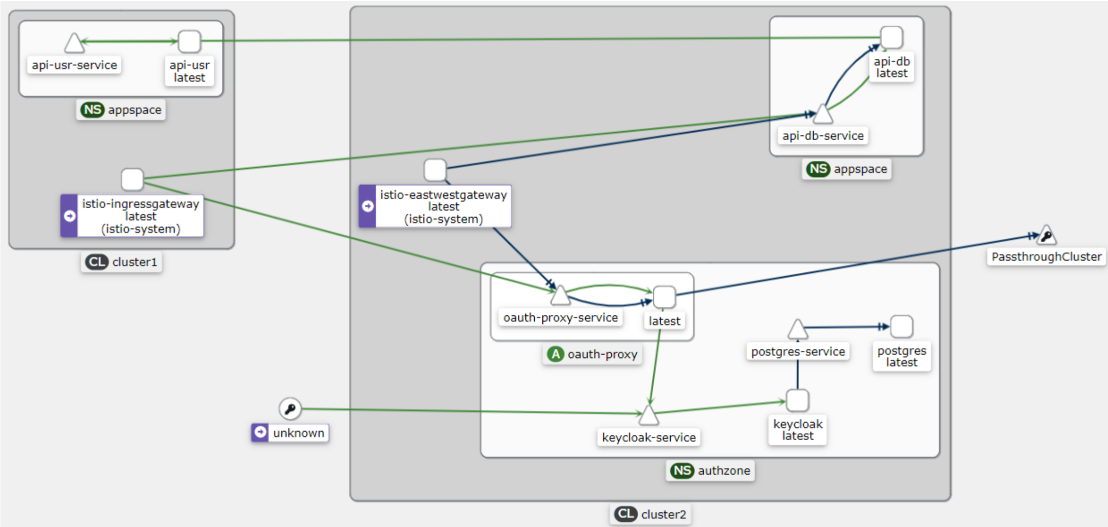


Figure 7.11: A Look at Kiali's Mesh Graph

7.4 Auth Flow Correctness

Keycloak is used as the authentication server and has been configured for three different accounts, with one of them having the PRIME role. There is not much to validate here, as only some of its basic features were used, and any mistake would have been reported by its dashboard. Regarding the JWT, we use oauth2-proxy to automatically do the redirect and manage the issued token. Before going into that, we can nonetheless test the token endpoint ourselves. In the image, a JWT response is obtained by providing the required fields. The decoded JWT is not provided, as an example of it can be seen thanks to the debug option in either Fig7.9 or Fig 7.10.

Figure 7.12: Keycloak issuing JWT

For what concerns oauth2-proxy, the authentication flow correctness is immediately evident by using the application. Once the user tries to connect to the application's page for the first time, a redirection occurs to Keycloak's login screen. This is mandatory and cannot in any way be circumvented. The first image shows the authentication screen and some requests-insight using the browser's DEV tools. The Sign In step makes sure the right credentials are inserted, otherwise will show an error and prompt the user to try again. Any following access to the page, after the first successful login, will be met with the application's page without any additional step visible to the user.

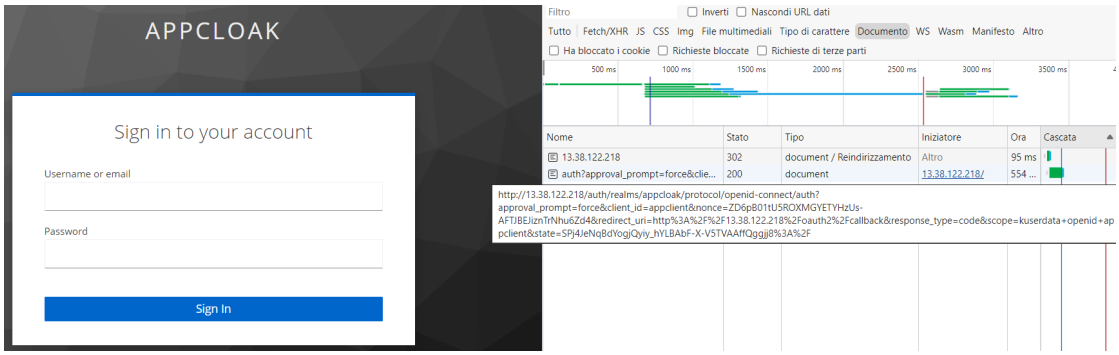


Figure 7.13: Auth Flow: First Access

Chapter 8

Conclusion and Future Works

The aim of this thesis was to examine the implementation of a Zero Trust architecture in a multi-cloud environment. The primary objective was to understand the feasibility and design considerations for achieving workload authentication in those kinds of environments.

The research comprised different steps: Initially, I studied various aspects of cloud computing and multi-cloud environments, drawing from resources such as the cloud computing exam material from Politecnico and relevant papers found on different academic portals. Next, I delved into the documentation related to Zero Trust security, Service Mesh, and specifically Istio. This provided a solid foundation for the subsequent phase.

The following stage involved designing a Proof of Concept (PoC) to demonstrate Istio's capabilities and verify the attainment of a Zero Trust architecture in a multi-cluster environment. This took the longest time as it was the heart of the thesis and many challenges had to be addressed to keep the resulting laboratory as easily accessible as possible. Rigorous testing was then conducted to validate the initial objective and generate paper-suitable material.

In conclusion, I have successfully demonstrated that the Service Mesh, particularly Istio, facilitates the establishment of a Zero Trust architecture in a multi-cloud environment with relative ease. Furthermore, this approach enhances clusters' functionality by enabling observability and traffic management capabilities, furthermore effectively addressing security and management challenges typical of those environments. However, it is essential to acknowledge that this work represents only an initial step in the dynamic landscape of multi-cloud approaches. Nonetheless, one of the core objectives of this thesis was not only to contribute to the existing knowledge base but to also provide what is needed to establish a simple laboratory

for further research.

Future extensions of this thesis could explore alternative solutions to trust issues, specifically without necessitating the sharing of a common Certificate Authority between the two clouds. Additionally, considering the growing demand for multi-cloud migrations, it would be worthwhile to investigate the possibility of connecting different service meshes (e.g., Istio, Kong) as such scenarios may arise in the near future and are currently obscure.

Bibliography

[1] *minikube start*. https://minikube.sigs.k8s.io/docs/start/. Accessed: 24-06-2023 (cit. on p. 31).

[2] *Install Docker Engine on Ubuntu*. https://docs.docker.com/engine/install/ubuntu/. Accessed: 24-06-2023 (cit. on p. 31).

[3] *Step by Step Guide to install Istio Service Mesh in Kubernetes*. https://dev.to/techworld_with_nana/step-by-step-guide-to-install-istio-service-mesh-in-kubernetes-d6d. Accessed: 24-06-2023 (cit. on p. 34).

[4] *Install Multi-Primary on different networks*. https://istio.io/latest/docs/setup/install/multicluster/multi-primary_multi-network/. Accessed: 24-06-2023 (cit. on p. 35).

[5] *Docker-Keycloak*. https://www.keycloak.org/getting-started/getting-started-docker. Accessed: 24-06-2023 (cit. on p. 42).

[6] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749 (cit. on p. 45).

[7] *Verify the installation*. https://istio.io/latest/docs/setup/install/multicluster/verify/. Accessed: 25-06-2023 (cit. on p. 55).

[8] *Github ksniff*. https://github.com/eldadru/ksniff. Accessed: 25-06-2023 (cit. on p. 56).