Secure microservice communication between heterogeneous service meshes

Zara Wajid Butt

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 29.07.2022

Supervisor

Prof. Tuomas Aura (Aalto) and
Prof. Panos Padadimitratos (KTH)

Advisor

Cihan Eryonucu (M.Sc.) and
Gabriela Limonta (M.Sc. Tech.)

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Zara Wajid Butt

Title Secure microservice communication between heterogeneous service meshes

Degree programme SECCLO

Major Security and Cloud Computing **Code of major** SCI3113

Supervisor Prof. Tuomas Aura (Aalto) and Prof. Panos Padadimitratos (KTH)

Advisor Cihan Eryonucu (M.Sc.) and Gabriela Limonta (M.Sc. Tech.)

Date 29.07.2022 **Number of pages** 62+6 **Language** English

Abstract

Microservice architecture is an emerging paradigm that has been unceasingly adopted by large organizations to develop flexible, agile, and distributed applications. This architecture involves breaking a large monolithic application into multiple services that can be deployed and scaled autonomously. Moreover, it helps to improve the resiliency and fault tolerance of a large-scale distributed application. However, this architecture is not without challenges. It increases the number of services communicating with each other, leading to an increased surface of attack. To overcome the security vulnerabilities, it is important that the communication between the services must be secured.

Service Mesh is increasingly embraced to resolve the security challenges of microservices and facilitate secure and reliable communication. It is a dedicated infrastructure layer on top of microservices responsible for their networking logic. It uses sidecar proxies to ensure secure and encrypted communication between the services. This thesis studies different deployment models of service meshes, identifies the reasons for federating heterogeneous service meshes, investigates the existing problems faced during the federation process, and proposes a solution to achieve a secure federation between heterogeneous service meshes, i.e., Istio and Consul. The security of the proposed solution was evaluated against the basic security requirements, such as Authenticity, Confidentiality, and Integrity. The evaluation results proved the solution to be secure and feasible for implementation.

Keywords Service Mesh, Istio , Consul , Federation , Kubernetes , PKI , mTLS

Acknowledgements

First and foremost, I thank almighty Allah (SWT) for his countless blessings, support, help, and generosity. He has always given me strength and encouragement throughout the challenging moments of my life.

I would express my sincere thanks to Professor Tuomas Aura from Aalto University for his valuable feedback and guidance that helped me complete the thesis. From KTH University, I would like to thank Professor Panos Papadimitratos and Cihan Eryonucu for their support. I express my deepest gratitude to manager Yoan Miche and advisor Gabriela Limonta for giving me a great opportunity to become a part of the Network Security team at Nokia Bell Labs. Gabriela has been a great support throughout the thesis process. Her constant support, feedback, guidance, and motivation kept me going. I sincerely thank all my colleagues at Nokia Bell Labs for their encouragement.

Special thanks to the SECCLO faculty and Erasmus Mundus for turning my dream of studying at the best European universities into reality. I would like to express my sincere gratitude to Dr. Hamza for encouraging me to apply for SECCLO and guiding me throughout the Erasmus journey. I am forever indebted to my wonderful parents, Wajid and Sughra, for their unconditional love, prayers, and support. I am grateful to my siblings, Sara, Shaheer, and Zoha, whose love and support kept me motivated and confident. Last but not least, a big thanks to all my friends for always being there for me in every situation and always encouraging me to do my best.

Espoo, 29.07.2022

Zara Wajid Butt

With the support of the Erasmus+ Programme of the European Union

Contents

List of Figures

Abbreviations and Acronyms

CA	Certificate Authority
PKI	Public Key Infrastructure
mTLS	Mutual Transport Layer Security
UN	United Nations
SDGs	Sustainable Development Goals
REST	Representational state transfer
API	Application programming interface
RPC	Remote Procedure Call
HPC	High Performance Computing
CSR	Certificate Signing Request
CNCF	cloud Native Computing Foundation
SAN	Subject Alternate Name
OSM	Open Service Mesh
SPIFFE	Secure Production Identity Framework for Everyone
RAFT	Reliable, Replicated, Redundant, And Fault-Tolerant
IoT	Internet of Things
AR	Augmented Reality
VR	Virtual Reality
VNF	Virtual Network Functions
SPIRE	SPIFFE Runtime Environment
VPC	Virtual Private Cloud
EKS	Amazon Elastic Kubernetes Service
IaC	Infrastructure as Code
OPA	Open Policy Agent
NIST	National Institute of Standards and Technology
CRL	Certificate Revocation List
OCSP	The Online Certificate Status Protocol

1 Introduction

In today's rapidly advancing world, many large-scale organizations are widely adopting cloud-based technologies [6]. With this broad acceptance of cloud-based technologies, microservices have also gained significant traction. Microservices allow an application to be loosely decoupled as multiple services that can be deployed and managed independently on different cloud platforms.

The microservice architecture is embraced by large enterprises, such as Amazon, Netflix, eBay, and LinkedIn [37]. This architecture enables these organizations to deploy their large-scale applications in the form of independently testable, scalable, deployable, and upgradable services on different cloud platforms [44, 33]. This architecture is an established approach for building enterprise and cloud-based applications as it permits a large-scale monolithic application to be divided into multiple services or modules, each responsible for accomplishing an independent task. This architecture offers a variety of benefits, such as streamlining the software development and delivery process as well as introducing more scalability. However, in comparison to the monolithic architecture, it is complex in terms of deployment and management. It requires multiple small-scale services to coordinate with each other constantly. Following this approach also increases the surface of attack because it leads toward more interconnections and communication links between microservices that always need to be protected [16, 14]. Furthermore, it increases the potential vulnerabilities an attacker can exploit. This architecture does provide the flexibility to implement each service using a different language or framework. However, that prompts additional issues. In a real-world scenario, hundreds of services communicate with each other, and it becomes a tedious process to implement the networking logic of each service using a different language or framework. Additionally, the networking logic requires different libraries that must be compatible with the business logic.

In recent years, there has been an incline toward the utilization of service meshes to provide secure and reliable communication and address some of the existing problems with the microservices architecture [42]. A service mesh is a dedicated infrastructure layer on top of the microservices to facilitate networking functionalities without changing the underlying implementation of the service. The fundamental features of a service mesh include service discovery, load balancing, fault tolerance, circuit breaking, traffic monitoring, authentication, authorization, and traffic management. A service mesh architecture isolates the business logic of a microservice from the networking logic by injecting intelligent sidecar proxies with the service. These proxies intercept the incoming and outgoing traffic of the service to provide a mechanism for establishing secure communication with other services [15].

A service mesh can help to solve the operational and development problems that come along with the microservice architecture. However, it may introduce a higher level of complexity when the deployment model shifts from a single mesh to multiple meshes.

1.1 Problem Statement

Service meshes (e.g., Istio[1], Consul[2], OSM[3] and Linkerd[4]) have been adopted as a solution to enable secure communication between microservices, both running in single or multi-cloud environments [15, 42]. However, establishing secure microservice communication between two services owned by separate entities that use different service mesh still remains an unsolved problem [35].

Current service meshes, such as Istio and Consul, offer some degree of federation between different clouds. However, they require using the same service mesh solution by both entities that want to expose services to each other. With the increased adoption of cloud-native technologies, DevOps, and hybrid cloud deployments, each company chooses the networking tools and cloud platform best aligned with their needs and requirements [9, 34, 45]. This may result in a very diverse deployment model, which is not fully supported by the current service meshes.

One real-world example of this complex model can be services owned by different organizations that must be securely exposed. Consider a scenario in which one service collects data and then shares it with a third-party service for analysis. As a result of the freedom offered by DevOps and the microservices architecture, there is a high possibility that each organization is using a different service mesh. The current service meshes fail to provide an efficient federation mechanism between heterogeneous service meshes.

Different service meshes have a built-in Certificate Authority (CA) component responsible for generating and rotating the certificates assigned to different workloads [19, 50, 48]. These implementations also support custom CA integration and using existing Public Key Infrastructure (PKI). Each service part of a service mesh has a unique identity, verified using X.509 certificates assigned by the CA to the service. The services use these certificates to verify each other's identity and start a secure Mutual Transport Layer Security (mTLS) communication. One problem arises when different service meshes having different root CAs need to be federated together. This introduces the need to establish a trusting relationship between the meshes to verify the identity of the services across the mesh.

The main problem with service mesh federation is establishing a trust anchor between the different service meshes that want to expose their services to each other. Especially when the two entities do not share a common root CA. In addition, there is a lack of compatibility between different service meshes due to the lack of a common interface between them and the use of different conventions for identifying and authenticating their services. Therefore, a mechanism is required for assigning consistent identities across any type of service mesh that can assist in achieving integration between different meshes. The thesis aims to solve the problems that are encountered when

[1] https://istio.io/latest/docs/
[2] https://www.consul.io/docs
[3] https://release-v1-1.docs.openservicemesh.io/
[4] https://linkerd.io/2.11/overview/

federating heterogeneous service meshes to ensure secure and reliable communication between cross-mesh services.

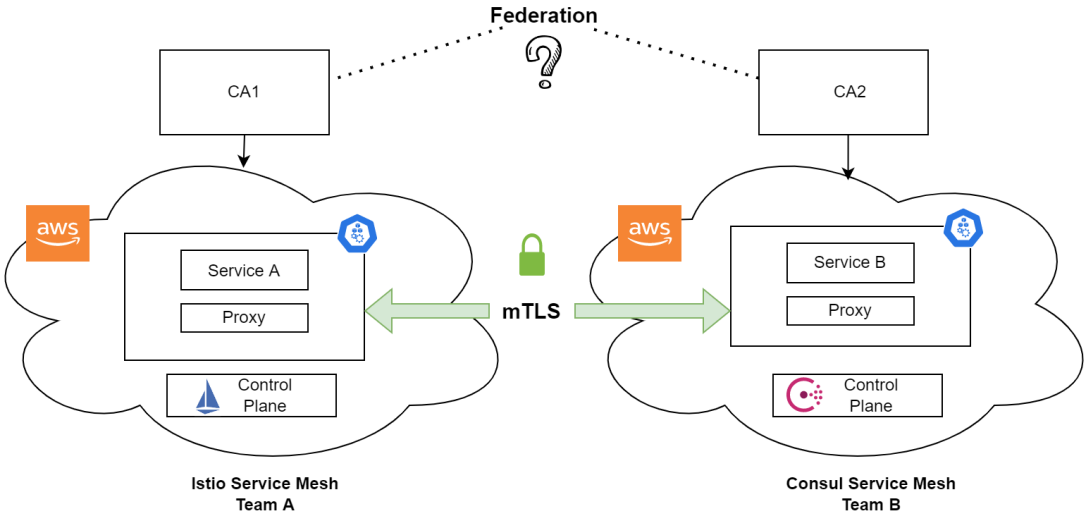

Figure 1: Limitation of Existing Service Meshes.

1.2 Main Goals

The main objective of this thesis is to propose an efficient and secure solution that resolves the problems experienced during the federation process of heterogeneous service meshes.

The goals of the thesis are as follows:

- Study the existing service meshes and their federation mechanisms
- Identity the challenges faced when federating heterogeneous service meshes
- Recognise the security properties required for establishing secure and reliable communication between heterogeneous service meshes
- Propose a solution that overcomes the problem of achieving federation between heterogeneous meshes and evaluate it against the identified security requirements.

Research questions are as follows:

- What are the primary reasons that encourage the integration of heterogeneous service meshes.
- How to achieve federation between heterogeneous service meshes so that different meshes having different trust roots expose services to each other and establish a secure mTLS connection.

- What are the drawbacks of the existing solutions, which make them inefficient for accomplishing the federation between meshes.
- What are the limitations of the proposed solution that can be explored in the future

No scientific papers describe the problems faced during the federation process of heterogeneous service meshes. Additionally, the official documentation of the service meshes does not provide sufficient information to conduct this process in an automated process. Considering the information mentioned above, it can be stated that the problem has not been extensively explored earlier.

1.3 Methodology

The research methodology of this thesis was to conduct a comprehensive literature review. The service mesh technology is novel; therefore, information was collected from different sources, such as scientific papers, official documentation, white papers, conference talks, blogs, and discussion forums. The information was useful in understanding the working of a service mesh. After the literature review was completed, it was confirmed that the problem under consideration is a novel idea, and no significant research efforts have been carried out in this direction. The precise literature review provided more profound insights into the service mesh technology and the problems encountered during the federation of heterogeneous service meshes. It helped to propose a solution to resolve the federation problem. The research methodology included the security analysis of the proposed solution. The proposed solution's limitations were also identified. The limitations can be considered as the future direction of advancements related to the thesis topic.

1.4 Delimitations

The current work can be continued in the future to implement a fully automated mechanism for conducting the trust bundle exchange between service meshes without the initial bootstrapping of trust, as discussed in Section 8. As of now, there is no significant academic research being carried out related to heterogeneous service mesh federation, Therefore, different areas were identified during the thesis that requires future research efforts. The service mesh implementations investigated in the thesis are new in the industry and lack the debugging support required for the federation scenarios. Due to this reason and the time frame, only the basic infrastructure of the proposed solution was deployed and tested on AWS. Furthermore, the microservice architecture used in the thesis as a case study is not a real-world microservice example. It is a small-scale application that consists of 3 microservices only. However, in a real-world scenario, a large-scale application consists of hundreds of services communicating with each other.

1.5 Sustainability and Ethics

United Nations (UN) presents its 17 Sustainable Development goals (SDGs) [5]. The thesis contributes toward achieving the Decent Work, and Economic Growth goal as the proposed solution aims to improve and modernize the current IT infrastructures. It incorporates advanced technologies, such as service mesh, into the existing cloud infrastructures. It also resolves the existing problem encountered while federating heterogeneous service meshes. This enhances the acceptance of this technology by diverse industries and companies. In the upcoming sections, we will explore that large-scale organizations are already migrating towards the cloud and embracing cutting-edge technologies, such as service mesh, containerized applications, microservice architecture, and Kubernetes. This adoption, together with the proposed solution, assists organizations in developing efficient and interoperable applications with a large profit margin. It also results in increased IT services export. Thus, eventually leading to higher economic growth.

The thesis aims to propose an innovative solution for the federation problem that can benefit the IT industry. Industry, Innovation, and Infrastructure is one of SDG's goals, which can also be achieved with the proposed solution. The solution proposed in the thesis resolves the microservice architecture problems and facilitates secure and reliable communication between services deployed on heterogeneous service mesh. Consequently, it discourages crime and protects the network infrastructure of various organizations using the service mesh technology. In this way, it helps to achieve the Peace, Justice, and Strong Institution goal presented by the UN. Concerning ethics, this thesis does not expose any sensitive data. Moreover, open-source service mesh implementations, such as Istio and Consul, are used in the thesis.

1.6 Thesis Outline

This thesis is structured as follows:

- Chapter 1 presents the motivation behind the research and an overview of the problem this thesis aims to solve
- Chapter 2 explains the main concepts related to the service mesh architecture. It describes the microservices architecture and compares different service meshes. Additionally, it discusses related work.
- Chapter 3 investigates the federation models of different service meshes
- Chapter 4 analyses the limitations and drawbacks of the existing solutions.
- Chapter 5 presents the security requirements for secure microservice communication between heterogeneous service meshes.
- Chapter 6 examines the proposed solution and deliberates about further areas of improvement.

[5] https://sdgs.un.org/goals

- Chapter 7 evaluates the security of the proposed solution against the requirements identified in chapter 4
- Chapter 8 outlines the future work directions related to service mesh federation
- Finally, chapter 9 concludes the thesis

2 Background

This chapter provides the background information related to the microservices architecture, Kubernetes, and PKI that are important to understand the service mesh architecture and its main concepts. Additionally, we discuss the workflow and entities of the two most popular service meshes, i.e., Consul and Istio. Finally, we review previous work done on the integration problem under investigation.

2.1 Microservice Architecture

The microservice architecture is an established approach to build large-scale cloud-based applications [21]. It allows a large monolithic application to be decoupled into multiple services that can be developed by different teams or organizations using different languages and frameworks. Moreover, the services can be deployed and managed separately on different cloud environments based on the use case.

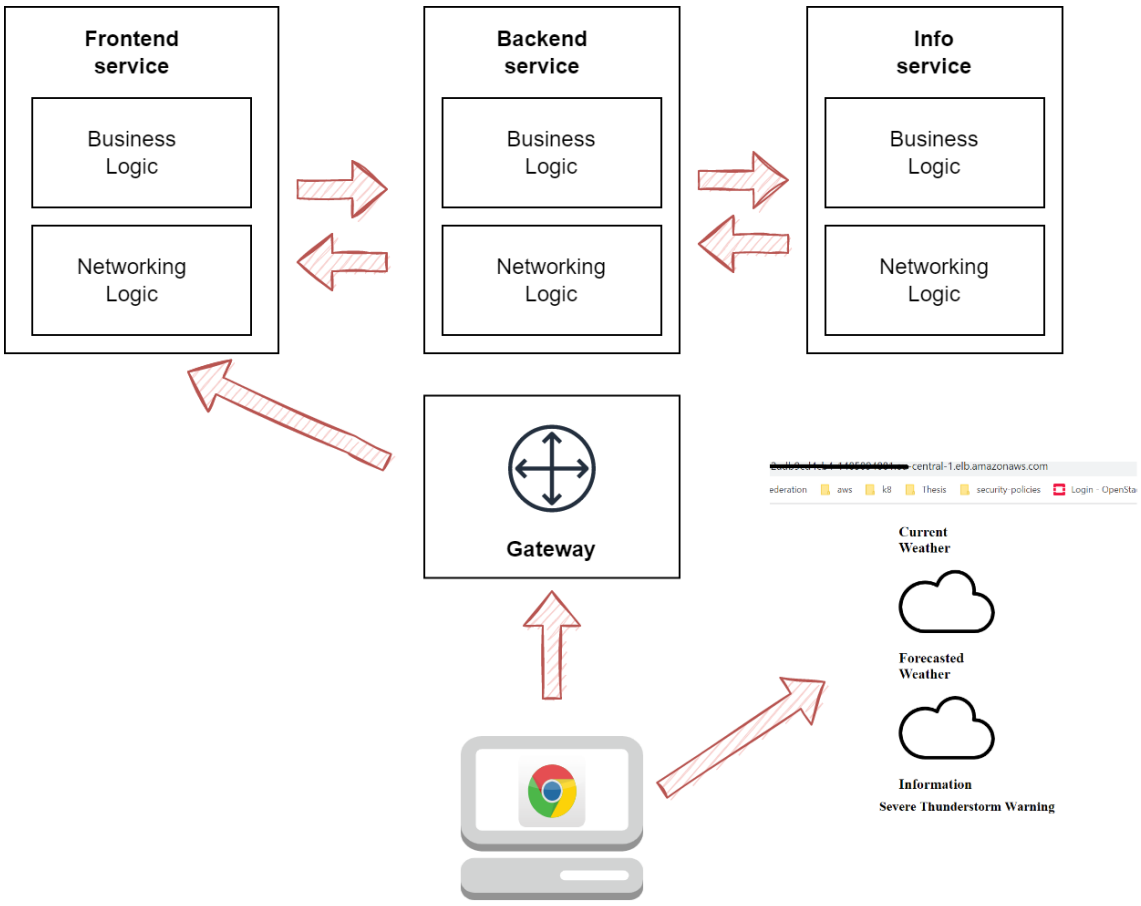

Figure 2: Weather Application Microservice Architecture

Microservices offer a plethora of advantages that lead towards faster software development and delivery [21]. The main benefits are as follows:

- **Development and deployment agility:** The decoupling of the application into multiple services enables them to be modified and deployed individually without affecting the state of the other service. Hence, the application development and deployment process become more agile.
- **Scalability:** The architecture facilitates scalability as each service can be scaled up or down independently of the others depending on the workload it receives. This helps to make the application more cost and resource-efficient.
- **Usability:** The microservices support representational state transfer (REST) application programming interface (API) and remote procedure call (RPC) for communicating with other services. Therefore, they are easy to use and integrate with other solutions.
- **Flexibility and Availability of tools:** Microservices offer flexibility to use any language or technology, and there are wide-ranging tools available to configure and deploy microservices on cloud platforms.

Microservice architecture is an efficient approach to divide a large-scale monolithic application into a suite of smaller services. Fig. 2 shows the microservice architecture that will be used as the case study in this thesis. It divides a monolithic weather application into a set of smaller services: frontend, backend, and info service. The backend service sends a request to the OpenWeather API [53] to get the current and forecasted weather in Helsinki. The backend service also connects with the info service to get additional remarks about the weather. The React-based frontend application sends a request to the backend to get the information and renders it on the browser when the client visits the URL of the application. The final outcome of the application can be reviewed in Fig. 2.

Driven by the advantages mentioned above, microservices have been adopted in different areas, e.g., big data processing [25], IoT backend[5], and high-performance computing (HPC) [17]. However, similar to any other solution, microservices also have some challenges and drawbacks listed below:

- **Communication and security overhead:** More microservices result in more interconnections between the services and more communication links to be protected. Therefore, it increases the communication and secure data transmission overhead.
- **Service discovery overhead:** Services can come and go dynamically; therefore, a mechanism is needed to discover the new services so that other services can communicate easily.
- **Monitoring overhead:** With a large number of services forming a single application, it becomes difficult to debug the errors and enable monitoring features for each service separately.
- **Authentication overhead:** All microservices must be treated as unauthenticated. A mechanism is needed to provide and verify their identities so that two

services communicating with each other know they are talking to the legitimate service. This will also help to prevent service impersonation attacks.

- **Authorization overhead:** The fine-grain nature of microservices requires fine-grained authorization at each microservice. However, this may require the security policies to be defined centrally and then distributed to different microservices to have a uniform and consistent configuration.

2.2 Kubernetes

Kubernetes is an open-source orchestration platform developed by Google [51]. It is the most used technology to orchestrate, i.e., deploy, manage and scale container-based applications. The widespread adoption of Kubernetes has made it the de-facto standard for container orchestration [41]. It makes it possible to create a cluster of servers and provide a fully automatic control to deploy a containerized application on that cluster.

The weather application presented in Fig. 2 is deployed on a Kubernetes cluster. A Kubernetes cluster is a set of nodes that run the containerized application. The cluster follows a master and worker node architecture. The master nodes combine to act as a control plane to configure the worker nodes. It is responsible for controlling the overall state of the cluster. The master coordinates the scheduling and scaling of the application on different worker nodes. It maintains the overall state of the cluster by ensuring that there are no unhealthy nodes inside the cluster. It also ensures that all the worker nodes are up-to-date. On the other hand, the worker nodes actually run the application and perform the tasks assigned by the master.

After the application is deployed to the worker nodes, it will run inside a pod. A pod is the smallest and most basic deployable object in Kubernetes. It is a single or a group of containers with shared storage and network resources. It is where an instance of an application actually runs inside containers.

The pods running in Kubernetes are ephemeral, and their IP address is inconsistent. Therefore, Kubernetes offers a service resource that is an abstraction defining a logical set of pods. It allows assigning a group of pods a specific name and a fixed IP address in the cluster network. A service resource exposes an interface to access the pods from within the cluster. An external IP is assigned to a service of type load-balancer, which can be accessed by external clients. The selector field of the service is matched with the labels of pods deployed on the cluster. A pod with a matching label as the selector of service becomes part of the service and can be accessed using the service name or IP address.

In the case-study application used in this thesis, we deploy the different parts of our weather application (frontend, backend, info) on a Kubernetes cluster and define Kubernetes service resource for each of them to be accessed. Afterward, we deploy Istio or Consul on the top of the cluster to take responsibility for secure service-to-service communication.

2.3 Secure Communication Using: PKI and mTLS

The PKI is the set of software, policies, and procedures used to assign identities to different entities over the internet in the form of digital certificates and public keys [1].

In a traditional PKI, a CA is responsible for issuing certificates to all the entities [1]. It is a common trusted third party for all the entities. For example, to receive a signed certificate from a CA, Alice first generates a private key and a public key, as illustrated in Fig. 3. Next, it sends its public key and certificate signing request (CSR) to the CA. After verification of Alice, the CA will issue a certificate by signing Alice's name and public key with its own private key. Now, when Alice receives the certificate from the CA, Alice is ready to initiate an authenticated connection with all entities on the network. Bob will follow the same process as Alice to get a signed certificate from the CA. Using the signed certificates, Alice and Bob can prove their identity to each other.

When Alice sends a request to Bob. Bob will present Alice with its certificate. Alice will verify the certificate with the CA's public key because the CA acts as a common trust for both Bob and Alice. Once the certificate is verified, Alice knows that the entity at the other end of the network is Bob and not someone impersonating to be Bob. Bob can also request Alice for the signed certificate and verify it using the CA's public key. By following this process, Alice and Bob are sure they are communicating with each other. This not only helps to achieve authentication but also assists in avoiding impersonation attacks.

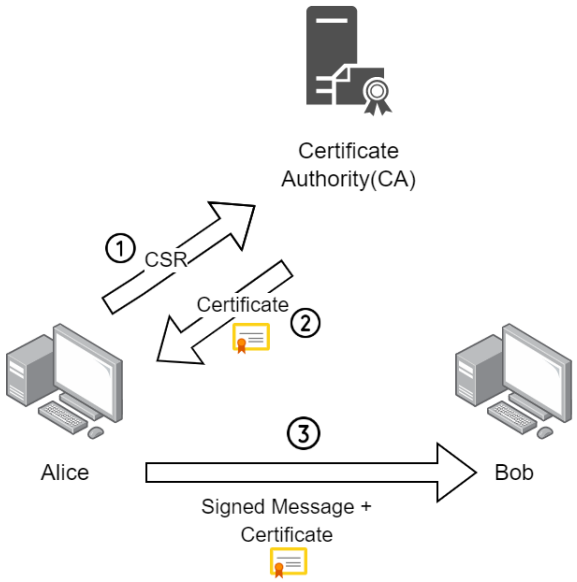

Figure 3: Public Key Infrastructure

The mTLS protocol is often used to secure and encrypt the traffic between users, devices, servers, and services. It is a common industry practice for microservice security. mTLS, along with PKI, is a method used to secure service-to-service

communication [11]. Different components of the PKI, such as CA, digital certificates, and public keys, assist in verifying the identity of the parties at each end of a network. It works in the following way.

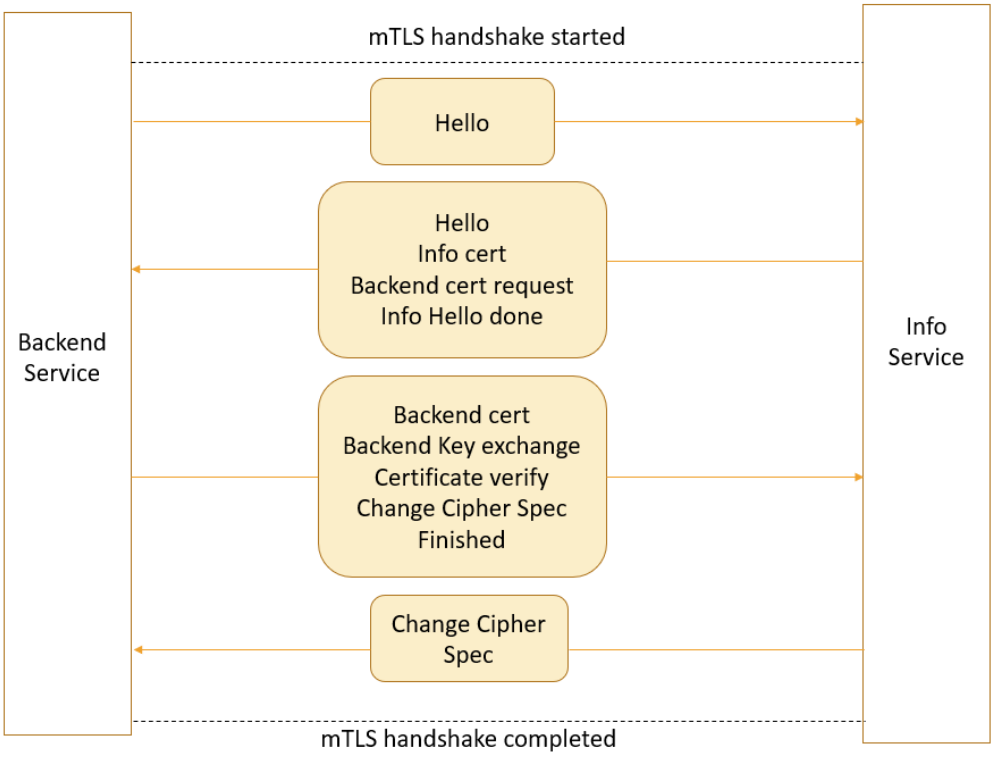

Figure 4: mTLS Connection

.

As illustrated in Fig. 4, the backend service sends a Hello message to the info service. In response to the Hello message, the info service sends Hello and its certificate. The backend service will verify the certificate using the CA's public key. In addition, the backend service will send its certificate to the info service. If both parties verify each other's certificate correctly, then a secure and encrypted mTLS connection is established between the services [48].

2.4 Service Mesh

A service mesh is a dedicated infrastructure layer for handling service-to-service communication without modifying the existing service implementation [15]. It handles the networking functionality using intelligent sidecar proxies that intercept all the traffic reaching a service. As shown in Fig. 5, the service mesh consists of two main components: the data plane and the control plane.

The data plane consists of sidecar proxies that mediate and control all network communication between the different services [15]. After each sidecar proxy gets injected into a service, it sits in front of the service and takes control of the traffic reaching the service.

The control plane is also referred to as the brain of the mesh. It is responsible for configuring and controlling the data plane (proxies) behavior across the mesh. Additionally, it enforces security and traffic policies as well as collects telemetry data [15].

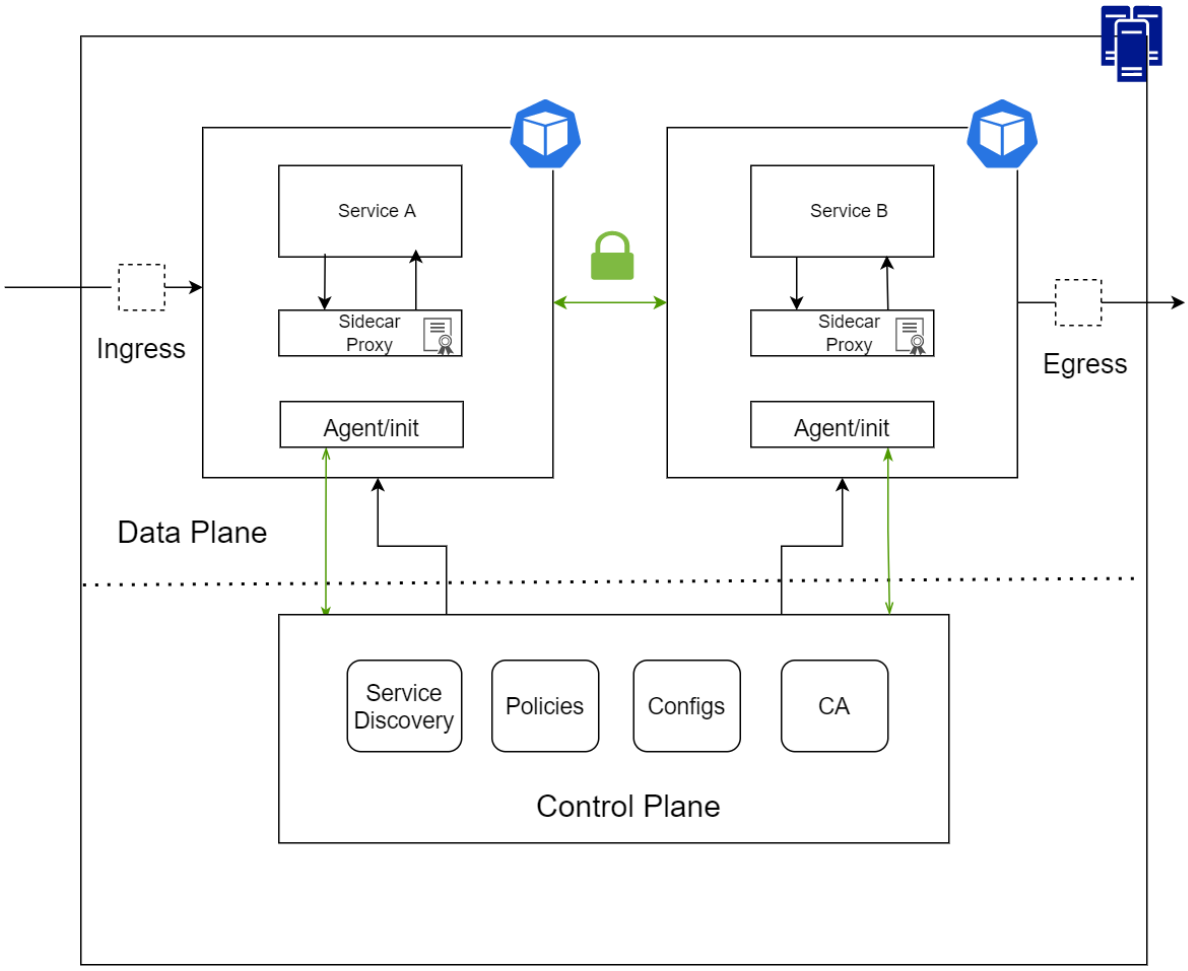

Figure 5: Service Mesh Basic Architecture.

2.4.1 Service Mesh Fundamental Features

Service meshes are widely adopted because they offer secure communication between different services and observability. Every service mesh must provide the following features:

- **Authentication and Authorization:** Facilitate certificate management, key generation, authorization policies, and token-based authentication
- **Service Discovery:** Allow the discovery of the endpoints of the services and register them to a centralized registry
- **Secure service communication:** Provide support for mutual Transport Layer Security (mTLS)
- **Resilience and Stability features**: Support features such as circuit breaking, retries, rate-limiting, fault injections, fail-overs, and timeouts
- **Observability and monitoring**: Support logging, metrics, and distributed tracing
- **Load balancing**: The ability to distribute the load between different running instances of a service for scaling and better performance

2.4.2 Service Mesh Implementations

The service mesh technology is getting widely embraced in the technology industry. There are many service meshes in the industry, such as Istio, Consul, Linkerd, and Open Service Mesh (OSM). In Table 2, we can observe that OSM is the least mature out of all four popular service meshes since it is not ready for a production environment. According to the number of Github stars, Consul and Istio are the most popular service meshes. They provide many features and functionality that allow the services to function properly. Moreover, these two service meshes support the Secure Production Identity Framework for Everyone (SPIFFE) standard for service identity [56] that is discussed in the Section 2.4.3. The SPIFFE identity can be helpful when trying to get the meshes to interoperate. Linkerd is also production-ready and supported by Cloud Native Computing Foundation (CNCF). However, its limitation is that it does not support the SPIFFE standard as the identity.

2.4.3 Service Identity

As discussed earlier in Section 2.3, PKI uses certificates to assign an identity to different entities. They act as the credentials to verify the identities of a pair of services in a transaction. Similarly, Istio and Consul provide each service a strong identity and an X.509 certificate. The identity is specified in the Subject Alternate Name (SAN) field of a certificate. The certificate is then used by services to initialize a secure and encrypted communication with another service.

The identity provided by Istio and Consul is based on the SPIFFE standard [48, 56]. According to the standard, each service is assigned a unique SPIFFE Identity. This

	Istio	Consul	Linkerd	OSM
Data Plane	Envoy Proxy	Envoy Proxy	Rust Proxy	Envoy Proxy
Advantage	Strong community and CNCF supported	Built-in service discovery mechanism	Stable and CNCF supported	CNCF supported
Limation	No service discovery mechanism	Complex architecture	No SPIFFE identity support	No multi-cluster federation support
Maturity	Production ready	Production ready	Production ready	Not Production ready
Popularity	30.5k	24.9k	8.5k	2.4k

Table 2: Service Mesh Implementation

identity is a string that represents the logical purpose of the service [24]. It has a specific and uniform format for all the services. For example, a backend service will have the identity shown in Figure 6. Both Istio and Consul support identity based on SPIFFE. This could be leveraged as the potential building block for the federated solution.

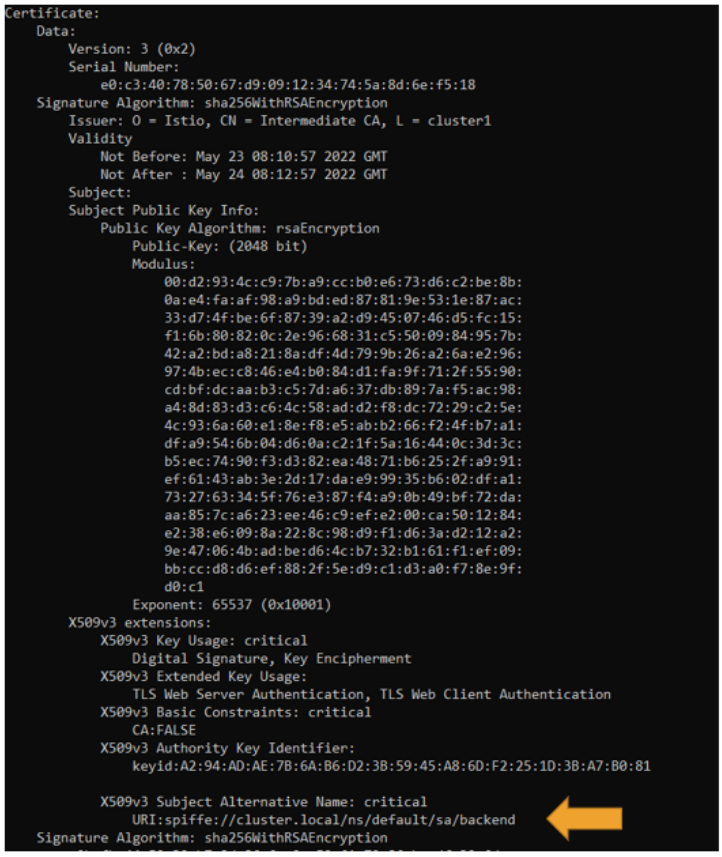

Figure 6: SPIFFE Identity

.

2.5 Istio

In this section, we will discuss the main components and workflow of Istio. It is apparent from the comparison drawn in Table 2 that Istio is currently the leading service mesh implementation.

2.5.1 Core Components

Istio is an open-source service mesh that adds a dedicated infrastructure layer underneath existing cloud-based distributed applications. With few or no code changes, Istio allows a developer to add vital features to the existing implementation. These features include secure service-to-service communication using mTLS, automatic load balancing, fine-grained authorization and authentication policies, metrics, logs, and telemetry data of all the traffic that enters the service mesh. Istio is designed to introduce visibility and extensibility for the service mesh. It also supports platforms such as Kubernetes, Openshift, and traditional deployment environments like VMs and bare metal servers [19].

As mentioned in Section 2.4, Istio is composed of two main components: the data plane and the control plane. The data plane consists of intelligent Envoy sidecar proxies that secure service-to-service communication. Envoy proxy[6] is a modern, high-performance, and lightweight service proxy based on C++ and is widely adopted for different service meshes [19]. The control plane is also referred to as Istiod, where d stands for daemon, and it manages and configures the data plane to ensure secure communication between microservices [28, 39]. Istiod has three main responsibilities: service discovery, certificate management, and configuration management [38, 39, 50].

Pilot is the component responsible for configuration management and service discovery. It is mainly responsible for the configuration of the proxies. It handles the responsibility of translating the Istio-defined APIs into Envoy proxy-specific configurations.

Citadel is the component that is responsible for certificate management. It acts as a CA and assigns certificates to all the entities of the service mesh in order to support mTLS. It also offers identity management of the services. This component plays a major role in secure service communication.

Galley handles the Istio-specific configurations. It is the interface for the underlying API with which the control plane needs to interact. It is responsible for interpreting and validating user-specified configuration into a format Istio can understand.

[6] https://www.envoyproxy.io/

2.5.2 Istio Workflow

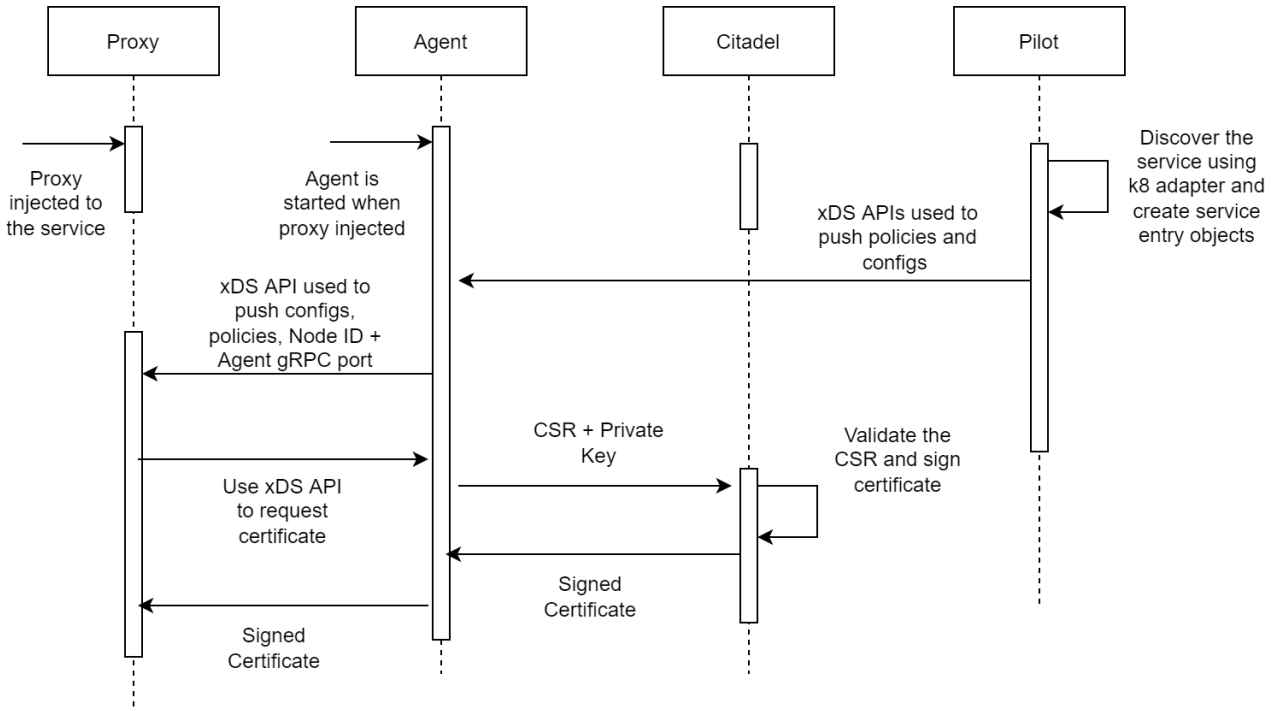

Figure 7: Istio Single Cluster Workflow.

When a service is deployed on a Kubernetes cluster, Pilot will be responsible for discovering this service. The discovery process is accomplished using the Kubernetes adapter that watches the Kubernetes API Server to obtain service information. The information is then used to create a service entry object in the internal service registry, which Istio maintains containing the set of all the discovered services. Pilot will then inject a sidecar proxy into the service. Before initiating a connection, the proxy uses the service registry to determine the endpoints of the other services [19]. Pilot implements a discovery service called xDS, which is used to configure the sidecar proxies [30].

The information for the service discovery from the internal registry, along with configuration information, is sent to the proxies with the xDS APIs[7]. These APIs are a set of aggregated discovery APIs for the proxies to discover different resources, such as Envoy Routes, Clusters, Secrets, Listeners, and Endpoints. Pilot is also responsible for propagating the changes in configuration across the service mesh. It does so by generating a model by combining Istio configurations from Galley and service information from the service registry. This model is used to generate configurations for the data plane. The model also changes when the configuration changes, which is conveyed to the relevant services via an active gRPC channel [19].

Along with the sidecar proxy, a local agent is also started. It acts as an intermediate

[7]https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol

control relay between the sidecar proxies and Istiod. All communication related to secrets and configurations between the proxy, Istiod, and agent happens using the xDS APIs. The agent handles sending a CSR to Istiod and then sending the signed certificate back to the proxy. When a proxy gets injected, Pilot will use the xDS APIs to push the policies and configurations to the agent. The agent will then push the cached policies, configurations, NodeID, and gRPC port to the proxy. The NodeID and gPRC port will be used by the proxy to connect to the agent and request an X.509 certificate [19]. The envoy proxy requests the certificate from an agent using the xDS API. In response to this request, the agent will generate a private key and CSR to the Istiod (Citadel Component acts as the default CA). Istiod will verify the CSR and issue a signed certificate to the proxy [50]. The signed certificate has the identity encoded in the certificate as SAN. This identity is based on SPIFFE. The signed certificate is then passed from the agent to the proxy. In this way, the proxy gets all the relevant configurations and policies to start mTLS connections with other proxies [19].

2.5.3 Traffic Management

Istio offers several routing options to control the traffic flow between services that are part of a service mesh. The traffic management process relies on the sidecar proxies that intercept all the incoming and outgoing traffic to the service; thus, making it easy to control the traffic around the service mesh without changing the state of any service.

Before controlling and directing the traffic to the correct service, Istio needs to populate its service registry by connecting with the underlying service discovery system. Once the services have been discovered, this information is propagated to all the other services.

Virtual Service: One of the most important traffic management API resources is the virtual service. It is used to define the traffic routing rules for services. It allows you to define routing rules that tell the envoy proxy how and where to direct the traffic. It can be used for canary roll-outs where the traffic reaching a specific version of a service gradually increases [28]. A virtual service can be used to redirect a certain percentage of the traffic to different versions of a service. Virtual services can also be used in the configuration of the traffic rules in combination with the gateway to handle ingress and egress traffic associated with the service mesh [50].

Destination Rule: Another core resource is the destination rule which grants control over what happens with the traffic that reaches a specific proxy. It provides the control of dividing a service into subsets based on version labels. Virtual services are then used to direct traffic to these different subsets. Moreover, this resource also allows setting load-balancing options and TLS settings [50].

Gateways: Gateways handle the traffic entering and leaving a service mesh. A gateway configuration can be added to a standalone Envoy proxy deployed at the edge of a mesh. Istio ingress and egress gateway deployments are configured along with the Istio basic installation. To handle routing, virtual services can be linked and deployed with the gateway to control the flow of traffic instead of adding application-layer (L7) traffic routing. This resource also lets you configure layer 4-6 load-balancing properties [50].

2.5.4 Security Management

Secure communication in a service mesh is accomplished with the following components and policies.

Istio Identity: Service identity plays a major role in initializing a secure connection between services. A pair of services that want to communicate with each other need to exchange their identity information encoded in a X.509 certificate SAN. In the case of Kubernetes, the identity is usually the service account of the service [50]. Kubernetes uses service accounts to assign an identity to services deployed inside a cluster. Along with the service, a service account is also linked and deployed with it [51]. Moreover, Istio and Consul both support SPIFFE Identity format mentioned in Section 2.4.3.

Alongside each Envoy proxy, Istio also provisions an agent with the service. This agent and istiod work together to automatically assign the proxy a certificate to prove its identity to other proxies [19].

Secure Naming: Istio secure naming maps the service account and server identity to a particular service name. It checks whether a server or service account is allowed to run the service. The control plane of Istio will watch the Kubernetes API Server and generate the secure naming information. Later the information is propagated to the proxies. The secure naming information is used during the mTLS connection between different services.

When a client service sends a request to a server service, it presents the client service with its certificate. The client service uses the service account name encoded in the certificate and secure naming information to determine whether the service account is authorized to run the service. If the verification is successful, only then the client and server services can establish a mTLS connection. The secure naming check prevents service impersonation attack. If a forged service with a different service account attempts to establish a connection with the client, the secure naming check will fail. This prevents establishing a connection with a forged service; thus, improving the overall security of the service mesh,

Authorization Policies: The authorization policies supported by Istio provide mesh, namespace, and workload-wide access control for different services. These policies can be used to control which namespace, service, or service account can

access another service. These policies dictate the behavior of proxies across the mesh. It provides options like ALLOW and DENY to control what happens to a request that reaches a service from a specific source [50, 19]. We will see examples of these policies later in Section 7.

Authentication Policies: Istio supports two types of authentication: peer authentication and request authentication. Peer authentication is used for service-to-service authentication and provides each service a strong identity in the form of a certificate. Istio distributes and updates the certificates across the mesh. The authentication policies can be used to enforce mTLS on the communication between service mesh workloads.

Request authentication is used for end-user authentication with JWT [50] The credentials attached to the request are verified before the requests are allowed to enter the mesh. In this thesis, we focus on inter-service communication in microservices which can be accomplished with peer authentication policies.

2.6 Consul

Consul is an open-source service mesh solution developed by HashiCorp [48]. Consul Connect is used interchangeably with Consul service mesh. It offers multiple features, including service discovery, L7 traffic management, health checking, and a key-value store. It provides the authentication and authorization features to ensure secure service-to-service communication with mTLS. It has a built-in proxy and also supports third-party proxy integration, such as Envoy. Envoy proxy is recommended for production environments. Consul connects automatically injects Envoy proxy to the services running in a Kubernetes cluster.

2.6.1 Main Entities

Consul has a complex architecture that consists of multiple components. Fig. 8 shows the main components of a Consul cluster. The cluster has multiple agents: clients and servers. The agents form the control plane of the service mesh. A single cluster must have 3-5 servers for fault tolerance and availability, and every node of a cluster has one client automatically running inside it [48].

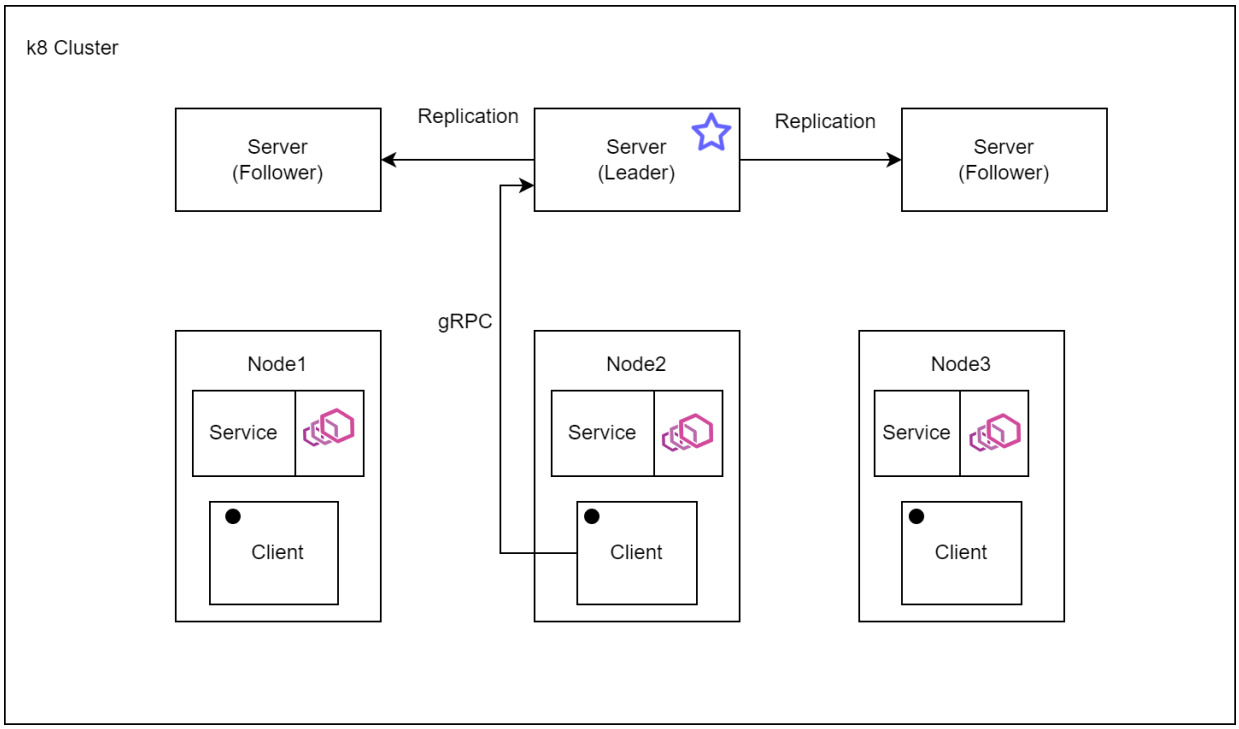

Figure 8: Consul Architecture for Kubernetes Cluster.

Consul Client: The Consul client agent runs on each node of the cluster and is responsible for generating the private key for the sidecar proxy. After the generation of the key material, it also sends a CSR to the Consul server. The server provides the client with certificates for the running services. After 75% of the current certificate lifetime elapses, the client will send a fresh CSR to the server. It also performs the health checking of the pods running on the node and configures sidecar proxy instances with the services running on the node [22].

Consul Servers: The Consul server agent handles the task of collecting information about the health of the services, service location, and availability. Furthermore, it performs the service discovery and certificate signing process. The servers are the central configuration point for policies and rules that define the behavior of the proxies in the data plane [22]. Consul uses intentions to define access-control policies across the mesh, which control which services may establish a connection [48].

2.6.2 Main Protocols

This section discusses the main protocols used in a Consul deployment.

Consensus Protocol: The Consul deployment must have at least 3-5 servers for availability in case of a failure. One of the servers is elected as the leader using the RAFT (Reliable, Replicated, Redundant, And Fault-Tolerant) consensus protocol [4]

and the others are selected as the followers. All the RPC requests and transactions are handled by the leader, and the results are replicated to the followers [48].

Gossip Protocol: It is used to manage membership and broadcast important messages to the cluster. All the agents in the data center must participate in the gossip pools [48].

LAN Gossip Pool: Each data center has a LAN gossip pool that contains all the members of the data center. The membership information provided by the LAN gossip pool helps the client automatically discover the servers.

WAN Gossip Pool: Regardless of the data center, all servers participate in the WAN gossip pool. It allows servers to initiate cross-data-center requests.

2.6.3 Consul Service Discovery

The services that are deployed on a Kubernetes node are registered to the server by a lightweight client agent running on each node. The agent is also responsible for initiating a sidecar proxy with the service. When a service is registered by the client agent, the central service registry is populated, which is a catalog of all the services that are part of the mesh. The agents also ensure consistent health-checking of the service to ensure the traffic is not redirected to an unhealthy instance of the service [22].

In a Kubernetes cluster, the services are automatically discovered and registered in the central service registry by the client agent. However, to configure an external service that is not part of the Kubernetes cluster, a manual service definition has to be provided to the agent. The upstream services with which the proxy needs to communicate can also be configured using the service definition.

2.6.4 Configuration Management

Along with other responsibilities, the client also bootstraps the proxy configurations that consist of an ID of the node the proxy is running on and the RPC port of the client agent. After starting up, the proxy will initiate a connection to the local Consul client to receive the remaining configurations.

The configurations sent from the Consul client agent to the sidecar proxy with xDS APIs include the X.509 certificate for the service identity, private key, Consul CA root certificate, L7 route configurations, and upstream services.

The Consul agent is also watching the server for any changes to the configurations. This is accomplished with the help of a streaming mechanism that ensures that all the proxies have the latest configurations and rules. The clients subscribe to a topic, and the servers publish a change to the topic that gets propagated to the relevant client agents. The updates are then passed to the proxies with xDS APIs and open RPC channels [22].

2.6.5 Certificate Management

When connect-inject is enabled for a service, Consul injects an init container to the service. The init container uses the service account token of the service to log in to the Consul server via the Kubernetes authorization method. As follows, the Consul server will validate the service account token by connecting to the Kubernetes API server. After the verification is successful, the Consul server returns a Consul ACL token that has the service identity and permissions to register the proxy and the service to the service registry. The proxy then uses the acquired token to authenticate to the client and request a certificate. The client validates the token and then sends a CSR to the Consul Server. The token and CSR are validated by the server before issuing the proxy a X.509 certificate with the SPIFFE identity encoded as the SAN of the certificate. This certificate is passed back to the client agent. Afterward, the client agent passes the certificate and the configuration bundle back to the proxy. This bundle contains the CA root certificate, intentions, and L7 route configurations [48]. Using the configurations and X.509 certificate, a proxy can initiate a secure mTLS connection with other proxies.

2.6.6 Consul Workflow

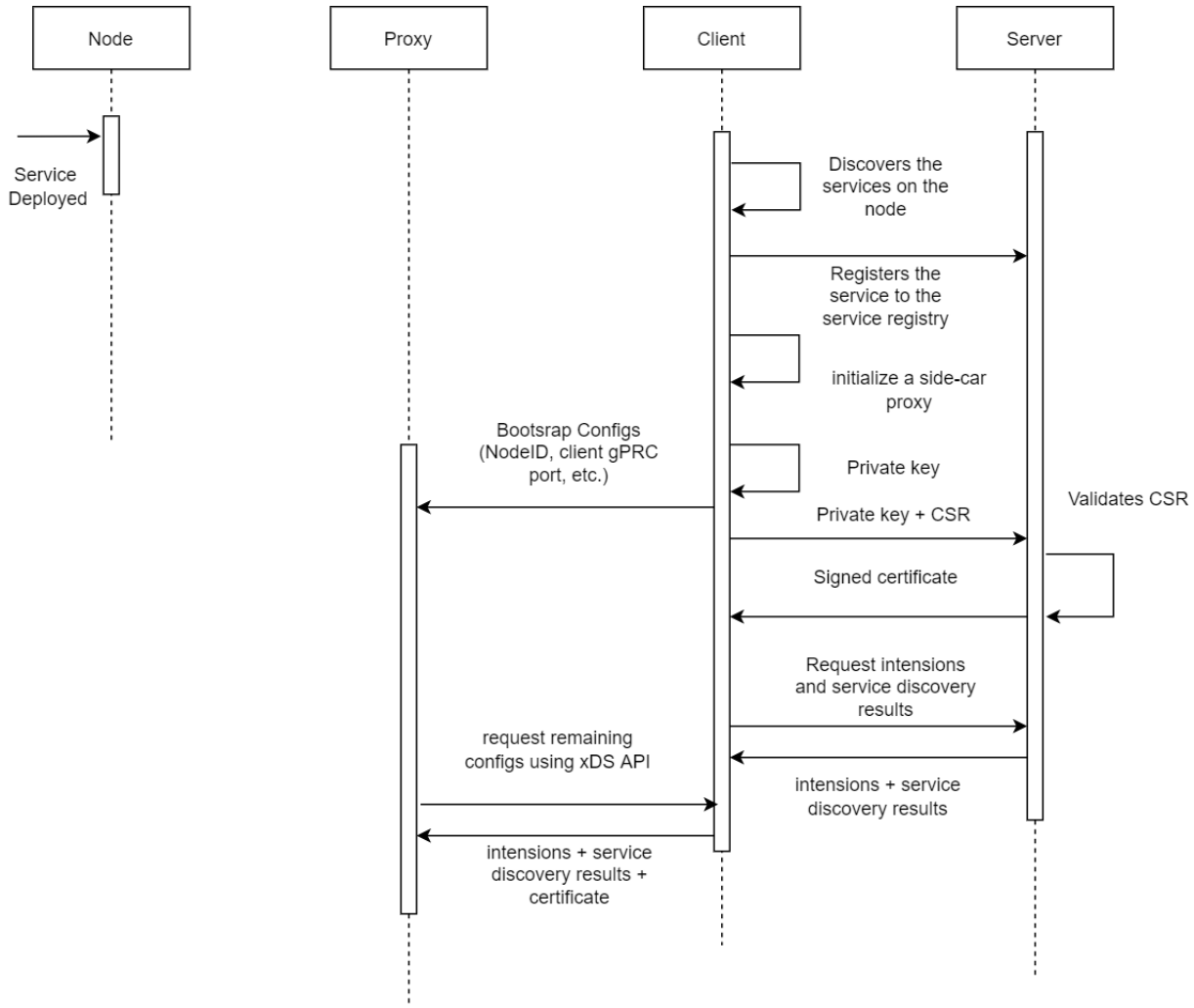

Figure 9: Consul Single Cluster Workflow.

2.7 Related Work

In this section, we explore whether there are any existing research efforts being carried out related to heterogeneous service mesh federation. This section will study the use of service mesh in different domains, such as Mobile Edge Computing, telecommunication, and the IoT. We discuss the need for heterogeneous service mesh federation in the context of these domains and investigate if there is any active work being done in this direction. Furthermore, we discuss state of the art in federation models and their limitations related to heterogeneous service mesh federation.

Service meshes are predominantly popular for cloud-native microservice applications. They have particularly gained traction for developing web-based applications hosted in a centralized cloud environment [42]. As a result of the high-level digitization of the IT business, the need for performance-driven decentralized applications has also surged. These applications mainly focus on automotive, industrial control, Augmented Reality/Virtual Reality (AR/VR), and Internet of Things (IoT) use cases with strict requirements regarding latency, bandwidth, and reliability. The high performance and low latency requirements encourage the adoption of Edge Cloud to distribute and decentralize the computational resources.

A systematic analysis was conducted to evaluate the suitability of a service mesh for a performance-demanding application hosted in a mobile edge cloud environment. The results of the experiments revealed that service meshes fail to fulfill the needs of performance-demanding mobile edge cloud workloads. Hence, more research activities are required to design a performance-efficient architecture that is able to satisfy the arising requirements [42]. It was concluded in the research [42] that the current state-of-art service meshes are only suitable for centralized cloud-native applications.

The telecommunication industry does not consider service mesh performance satisfactory because they provide less support for decentralized mobile edge computing and respond inefficiently to the diverse service requirements of 5G. The deployment model of a mobile edge cloud consists of multiple sites and tenants. It includes a hierarchically structural model comprising multiple sites, such as edge sites, distributed sites, and central sites. These sites have heterogeneous hardware and software setups and different geographical locations. Thus, making it very complex to migrate the sites towards a service mesh architecture [15]. High expertise is needed to integrate service meshes into this architecture. The level of complexity will drastically increase if different hierarchies use different service meshes. The existing scientific work does not discuss this problem of federating heterogeneous meshes. Therefore, the thesis aims to propose a solution for integrating heterogeneous meshes.

Contrarily, in another research, service mesh showed promising results when used in the deployment of an IoT platform [26]. After extensively testing the platform based on the Istio service mesh against different parameters, such as response speed, availability, transmission delay, concurrency, and throughput, it was concluded that the architectural design with the Istio service mesh had a satisfactory performance [26].

Today, there is an apparent increase in the demand for cross-domain IoT applications. This can lead to the adoption of collaborative and interoperable IoT solutions to introduce improvements in the future IoT ecosystem [12]. If the cross-domain IoT platforms are deployed with different service mesh implementations, it is important to have a smooth federation process. However, the existing research efforts do not discuss the collaboration mechanism needed to federate different IoT platforms that have heterogeneous service meshes.

Multi-tenancy is another popular deployment model in the telecommunication industry. This industry requires deploying virtual network functions (VNFs) owned by data-center tenants to incorporate the latest technologies, such as service meshes. However, achieving isolation among the tenants is a difficult task when considering a multi-tenancy model. A solution is proposed in [31] that uses multiple CAs to achieve isolation between tenants in a single Istio service mesh. The solution is also tested and proved to satisfy the performance and security requirements described in the work. However, this research does not discuss the federation process of the service mesh hosting services for multiple tenants with an external heterogeneous service mesh.

Ingress and Egress traffic of the service mesh is handled using gateways supported by service meshes. For example, Istio has in-built support for Istio ingress and egress gateways. Similarly, Consul supports several gateways, such as mesh gateway and ingress gateway, to take care of external traffic outside the service mesh. These in-built gateways allow secure access to services from outside the cluster. They support many functionalities at the edge of the service mesh that includes load balancing, authentication, authorization, rate limiting, traffic splitting, and telemetry collection. These functionalities offered by the in-built service mesh gateways overlap with the ones provided by an API gateway. The API gateway also acts as an entry point to the service mesh. An API gateway abstracts and encapsulates the internal implementation and workings of the system [13]. It acts as a mediator between the external clients and the microservices and assists in making the services lightweight by supporting different authentication options [18].

API gateways are predominantly adopted by the industry to design microservice applications [7]. Instead of service mesh gateways, they can be used to achieve federation with other meshes. However, most of the important features offered by an API gateway are already adopted by service meshes (Istio and Consul) in-built gateways. Therefore, using the in-built gateways instead of an API gateway along with the service mesh reduces the overhead of getting into a more complicated design pattern that comprises different technologies. Service mesh gateways provide most of the functionality that an API gateway provides. Thus, instead of integrating an API gateway with a service mesh, the default gateways can prove to be a more practical approach. The debugging process will be easy if a single technology is adopted. Moreover, additional expertise will not be required to develop and implement the API gateways along with the service mesh technology. Therefore, this thesis uses the in-built gateways to achieve federation between heterogeneous service meshes.

Federation between instances of the same service mesh (Istio and Istio) has been discussed [20]. Different models are discussed, such as two Istio clusters sharing the same control plane or having a dedicated control plane. Moreover, implementation details of deploying multiple Kubernetes clusters on AWS, integrated with a shared Istio control plane over OpenVPN, are discussed comprehensively [20]. The Istio documentation also explains various federation deployment models, such as deploying Istio on a single cluster, multiple clusters having the same control plane, and multiple clusters with different control planes [50]. However, these current state-of-the-art federation solutions are only limited to homogeneous service meshes.

Gloo Mesh is a service mesh and control plane that federates the configuration, operation, and visibility of different services deployed in distributed environments [55]. It supports multi-tenancy and multi-platform federation management. It also supports the federation of the trust domains to facilitate an easier and more secure multi-cluster federation. However, Gloo Mesh is specifically designed to support the Istio service mesh and does not support the federation of heterogeneous service meshes, such as Istio and Consul. The federation of two service meshes with different CAs and trust domains is challenging. The services residing in one mesh should be able to verify the identity of other mesh services to establish a secure connection. According to the Istio documentation, to start communication with two different service meshes, their trust bundles need to be exchanged, which can either be done manually or using SPIFFE [50, 56]. However, there is no concrete information available to accomplish this trust bundle exchange. Similarly, Consul documentation provides information about federating Consul meshes that are deployed on different cloud platforms(AWS and Azure) with the support of mesh gateways [48]. However, there is not sufficient information available for the federation of clusters with different root CAs. Thus, this is still an open research area that this thesis aims to explore.

2.8 Summary

In this chapter, we discussed important concepts that laid the basis for understanding the service mesh architecture and workflow. After comparing different service mesh implementations, Istio and Consul were concluded to be the most popular. Furthermore, this chapter gave us a detailed overview of the main entities and working of Consul and Istio. Lastly, we investigated the exiting research work being carried out related to heterogeneous service mesh federation.

3 Service Mesh Federation

Nowadays, many organizations prefer migrating to the cloud to deploy and manage their applications [2]. The use of multi-cloud platforms is likewise acquiring a foothold [23]. It offers several benefits, such as flexibility in using the best feature of a platform on a per-instance basis leading toward cost optimization. It also reduces the risk of failure of a single cloud platform by eliminating a single point of failure. Moreover, it allows coping with privacy issues [29].

DevOps is an emerging software development and delivery practice that is widely embraced by many technological organizations [36] . It not only accelerates the software delivery and development process but also provides the freedom to choose any language, framework, or technology stack. In addition to choosing different programming languages and software frameworks for microservices, teams can use diverse virtual networking technologies, including service meshes, for secure microservice communication.

Heterogeneous technology choices can also originate from corporate mergers and acquisitions. In a merger situation, organizations that use different technology stacks and deployment options may need to integrate their applications to communicate with each other securely. One approach for achieving this could be to make tools and technologies of the acquired organization consistent with the acquirer organization, e.g., make the acquiree shift to the same service mesh infrastructure as the acquirer. However, this is a lengthy process as it requires the developers to learn new skills and change the established ways of working. An alternative to quickly integrating these two services would be the ability for their service mesh infrastructures to federate easily.

Today, interoperability has become a pressing need for information systems [3]. A microservice deployed on service mesh A might request some data from an external service deployed on service mesh B to make the whole application work efficiently. This increasing demand for interoperable services has driven the need for federating different technologies.

The federation of service meshes across different clouds is a complex topic. Integrating different technologies and platforms to achieve a smooth workflow requires a simplified solution [10].

In the following sections, we will discuss scenarios in which service meshes can be deployed and federated. We consider use cases where different microservices that need to communicate are deployed on different Kubernetes clusters with their own service mesh on the top.

3.1 Homogeneous Service Mesh integration

Here we discuss service mesh federation in the situation where the service mesh layer is homogeneous, i.e., one service mesh implementation is used by a pair of entities.

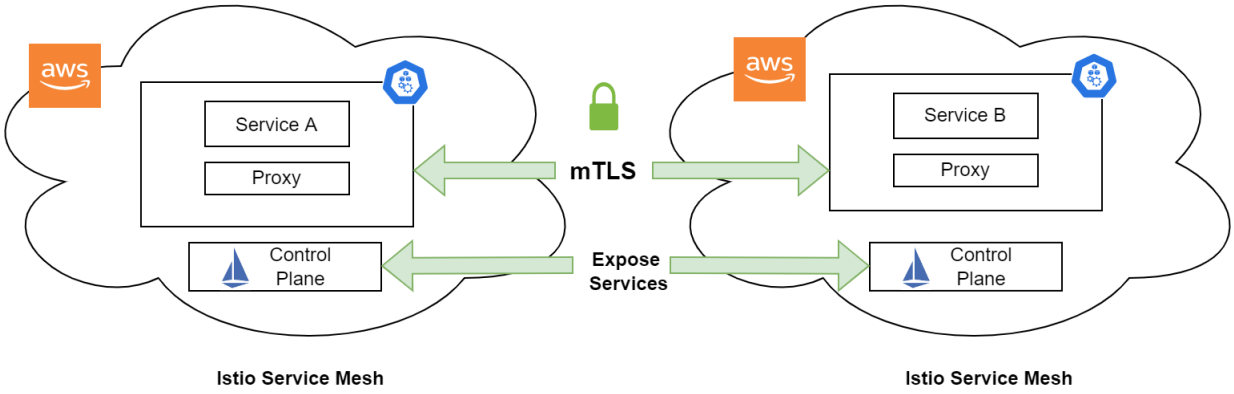

Figure 10: Istio-Istio federation.

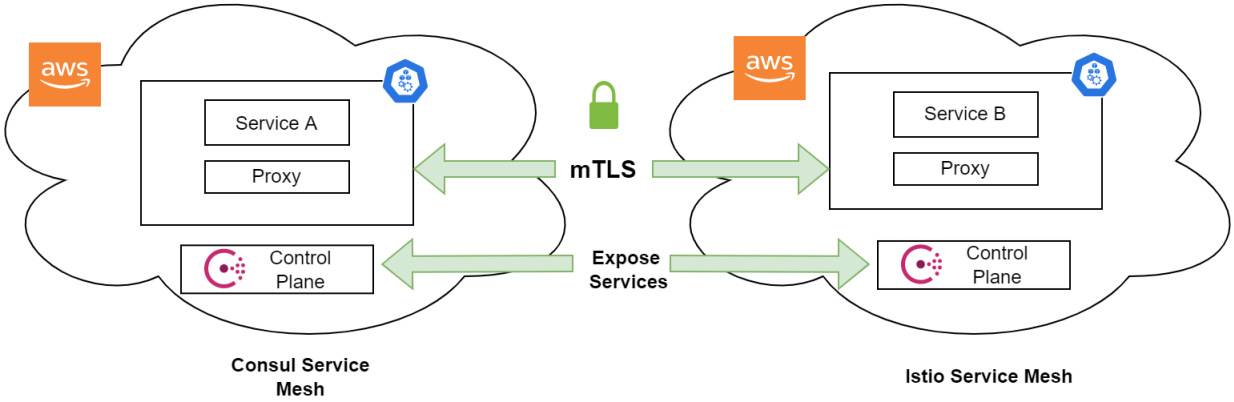

Figure 11: Consul-Consul federation.

3.1.1 Owned by the same organization

A practical example of this use case is that Company 1 has two teams (A and B) taking care of different solutions. Suppose team A service wants to consume the data produced by the service of team B. Take into account another consideration that both the teams use the same service mesh, either Istio or Consul, on both ends, as shown in Figures 10 and 11.

In this use case, achieving federation would require both teams to decide if they would prefer a shared control plane or a separate control plane. The choice will merely depend on the level of security the companies want to achieve. In the case of a shared control plane, the primary control plane will connect with the Kubernetes API server of the other service mesh. This will enable automatic service discovery. However, for a separate control plane, the Kubernetes API Server secret will be shared to connect the two control planes and discover the services [50].

The control plane of the service mesh act as the CA. Once the services are discovered by the control plane, it assigns each service an X.509 certificate as an identity. The service later uses this certificate to initiate a secure mTLS connection with other services.

3.1.2 Owned by different organizations

An example of this use case is two companies (Company 1 and Company 2), that have collaborated together. Company 2 has to expose specific services for Company 1 to consume useful data. In this example, we consider that both the companies use the same service mesh, either Istio or Consul.

In the case of different organizations, the same root CA cannot be used, and the control plane cannot be shared. Each cluster will be configured with a separate control plane and a different root CA. The services can be manually added to the service registry for discovery because sharing Kubernetes API secrets with another organization is not a recommended security practice. It leads towards an external organization gaining complete access to the local services in a mesh.

Once the service registry is populated, and services are assigned identity in the form of X.509 certificates, the service of Company 1 can initiate requests to service of Company 2. When the services are trying to establish a mTLS connection, they exchange the X.509 certificates to prove their identities to each other. However, the mTLS connection will fail as both clusters will be unable to verify the certificates signed by different CAs. To achieve federation in this use case, both the clusters require the public certificates of the other cluster to be imported to verify each other correctly and start a secure mTLS connection.

3.2 Heterogeneous Service Mesh Integration

Here we discuss service mesh federation scenarios in which the service mesh layer is heterogeneous, i.e., different service mesh implementation is used by a pair of entities.

3.2.1 Owned by the same organization

A practical example of this use case is that Company 1 has two teams (A and B) developing different solutions. Suppose a service of team A, which is using Istio service mesh, wants to consume the data produced by the service of team B, which uses Consul service mesh.

In this case, both the teams will have separate control planes specific to their service mesh. However, the root CA can be the same for both teams. If team A uses Istio and team B uses Consul, they both can be configured to use the same root CA to sign the certificates generated for each service. Also, if the service in the Consul service mesh needs to be exposed to the service in the Istio service mesh, it can be accomplished by sharing the Kubernetes secret [50]. It will allow each mesh to connect with the

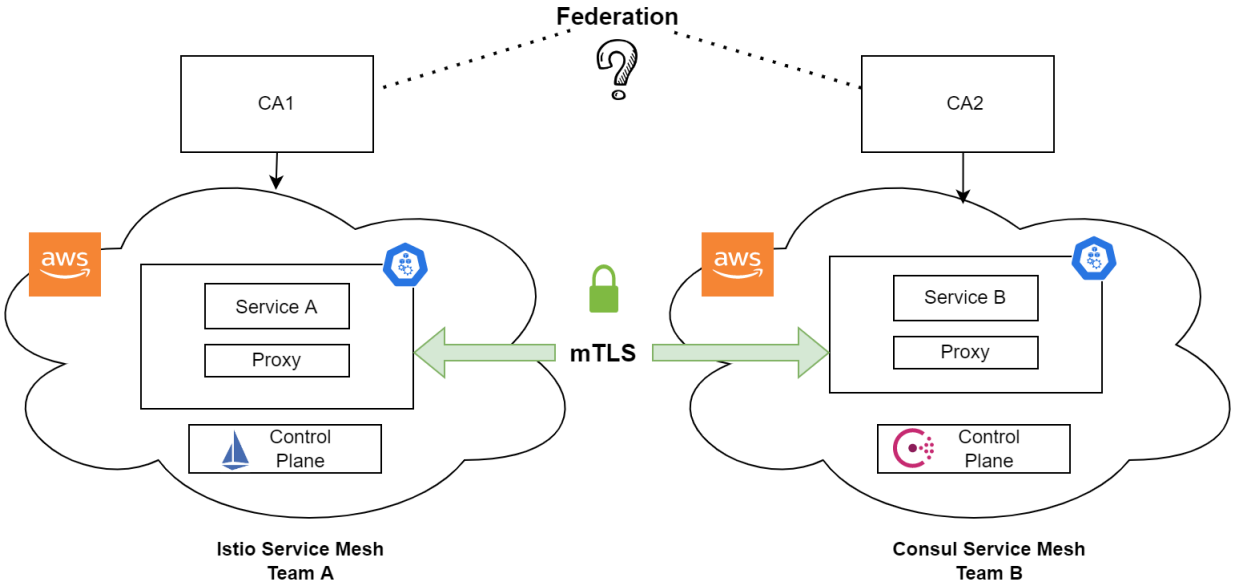

Figure 12: Istio-Consul federation.

Kubernetes API Server of the other mesh and discover the services automatically. If the same root certificates are plugged inside the control plane of both the clusters, then the service inside each cluster can verify each other's identity.

3.2.2 Owned by different organizations

An example of this use case is two large companies (Company 1 and Company 2) that have collaborated. Now, Company 2 has to expose specific services for Company 1 to consume useful data. We can see from the Fig. 12 that both the companies use a different service mesh, either Istio or Consul.

Considering the common security practices, there will be a separate control plane and different root CA configured for each service mesh. To achieve federation between these heterogeneous meshes having different root CAs, a common trust need to be established between the meshes to start a secure mTLS connection to the external service sitting in another service mesh. Establishing trust between heterogeneous service meshes is a complex and manual process with very limited help available online on the official documentation and discussion forums of Istio and Consul. This thesis aims to provide an efficient and secure solution to implement federated deployments.

3.3 Problem Areas of Service Mesh Federation

Different organizations have started adopting the service mesh technology as a next step to modernize their IT infrastructure. The service mesh technology is continuously evolving and maturing over time [47]. Homogeneous service mesh integration is

becoming a common use case, and sufficient help is available on the existing official documentation of Istio and Consul to federate the same service meshes. However, a developer faces multiple challenges while federating heterogeneous meshes.

Istio and Consul both support different access control policies. For example, Istio has authorization policies to control access [50]. On the other hand, Consul provides access control in the form of intentions [48]. Currently, there is no mechanism to unify these access control policies when distinct meshes are getting federated.

Service discovery between heterogeneous service meshes is also a difficult task. Organizations do not prefer to share Kubernetes secrets with others because they can provide complete access to the services. Istio and Consul support manual addition of services. This feature can add an external service in the mesh which is not part of the local Kubernetes cluster. However, integrating the control planes of two separate service meshes to automatically discover the service is not a straightforward task. Furthermore, integrating two service meshes, either homogeneous or heterogeneous, can result in the conflict of private IP addresses. This problem can be resolved with the inbuilt gateways supported by both Istio and Consul.

The services exposed to the external service mesh should also have a unified naming convention. For example, both Istio and Consul have the identity as the SPIFFE URL, as discussed in Section 2.4.3. However, the template of the identity can vary among deployments. If the identity naming is not unified, this can result in unsuccessful mTLS connections between services across the service meshes. Therefore, prior to exposing services, the organizations need to agree on having a unified naming convention for the services they plan on exposing.

Establishing trust between the different service meshes when the root CAs differ is still a complex problem. According to Istio's official documentation, to establish trust between meshes with different CAs, the trust bundles can be manually exchanged between the control planes. SPIFFE Trust domain federation, discussed in the next chapter, can also be utilized to automatically carry out the exchange [50]. On the other hand, Consul documentation does not provide any guidelines related to this problem. Therefore, it can be cumbersome for a new developer to resolve this problem with the limited support provided by the existing documentation.

The problem of exchanging trust bundles when having heterogeneous service meshes with incompatible control planes remains unsolved. Both service meshes, Istio and Consul, have different and incompatible control planes. Istio has Istiod as its control plane, and Consul has different servers acting as the control plane of the mesh. However, the features and functionalities offered by the control plane are almost the same. Exchanging the trust bundles between these conflicting control planes can be a time-consuming task. Therefore, an automated approach is required to establish, update and revoke trust between meshes to accomplish successful integration.

3.4 Summary

This chapter described the reasons for integrating different service mesh and utilizing the multi-cloud approach. Different use cases for federating the service meshes are explained in this chapter. These use cases assist us in understanding the ongoing problems better that are encountered when homogeneous and heterogeneous service meshes having a different foundation of trust are being federated.

4 Analyzing Existing Solutions

This chapter discusses the existing solutions that can resolve the challenges associated with heterogeneous service mesh federation. The drawbacks and limitations of the existing solutions are analyzed in this chapter. Furthermore, we discuss the need for an efficient solution that can overcome the problems of the existing solutions while meeting all the security requirements.

4.1 Cross Signing

Cross signing can be a possible solution to solve the lack of common CA problem. The root certificate of each mesh can be cross signed manually by the root certificate of the other mesh. As illustrated in Fig. 13, the root CA1 and root CA2 cross sign each other to generate cross signed certificates for CA1 and CA2, respectively. These cross signed certificates are then used to generate certificates for the services. As illustrated in Fig. 17, the services present each other certificates during the mTLS handshake. These certificates are successfully verified as they are generated with cross signed certificates. This allows the trusted root CA to extend its trust to the other CA [27]. The dynamic number of clusters in a large-scale application makes this solution non-scalable. Usually, it consists of a large number of clusters. If the number of clusters is n, then the complexity of cross signing the certificates is $O(n^2)$. The cross signing of the certificates is hard to automate. The revocation of certificates is also challenging in case of CA misbehavior or stolen keys [8, 27]. The cross signing also makes it difficult to track the revocation of a certificate, especially in the case of non-browser applications [27].

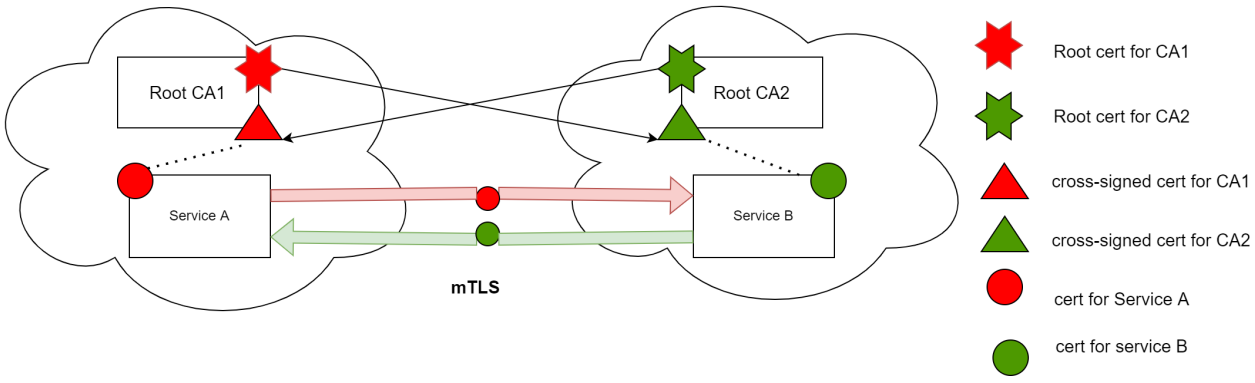

Figure 13: Cross Signing

4.2 API Gateway

API Gateway is often considered an alternative approach to service meshes for accessing external services deployed in a separate cluster [13]. As seen in Fig. 14, it acts as a reverse proxy. The gateway receives the request from the client and

forwards the request to different microservices for the response. Afterward, it sends the response received from the microservices back to the client. In this way, the client is abstracted from the details of the underlying microservice architecture and only has a single point of entry to the services.

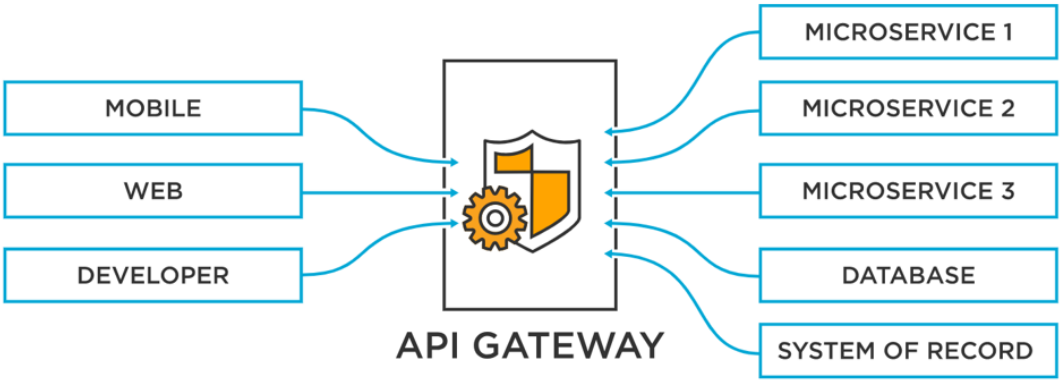

Figure 14: API Gateway [54]

An API gateway sits at the edge of a cluster and is only responsible for handling external traffic reaching the cluster. It does not take care of securing the east-west traffic between the services. The zero trust model is becoming the need of today's digitalized world. It follows the principle of never trusting and verifying all the network entities [32]. The concept of a service mesh is based on this model. Thus, the service mesh provides more security because they also ensure secure and reliable service-to-service communication along with securing external traffic. API Gateways are preferred in the case when the consumer-provider relationship is clear and when service-to-service security is not a critical requirement. Service mesh implementations, such as Istio and Consul, already support inbuilt gateways to take care of north-south traffic entering the mesh. Thus, using an API Gateway along with the service mesh technology will be an extra overhead. Additional skill sets would be required to develop and maintain an API Gateway. Due to the various benefits the service mesh technology offers, it is becoming progressively popular every passing day and is an optimal choice when considering microservices security [21].

4.3 SPIFFE

SPIFFE is an open-source set of standards for software identity [24]. SPIFFE aims to have interoperable service identities that can consistent across organizations. It automatically delivers identities to services while managing the lifecycle of the issued identity. The SPIFFE ID has a specific format as mentioned in Section 2.4.3.

The concept of having a separate trust domain is also introduced by SPIFFE specifications that can be used to manage security boundaries within and between different

organizations [24]. The trust domain is part of the SPIFFE ID and where a set of public keys is considered authoritative. Every trust domain has a trust bundle associated with it, which contains the set of public keys that are used to verify the services that claim to reside in the said trust domain.

Figure 15: SPIFFE Federation [24]

To achieve federation and allow services in a local trust domain to access services in a foreign trust domain, each local service must possess the trust bundle associated with the trust domain of the foreign service. As illustrated in Fig. 15, SPIFFE supports a bundle endpoint mechanism to exchange the trust bundles. Consequently, allowing the services in the local trust domain to validate the identity of the services in a foreign trust domain. The exposed bundle endpoint, HTTPS endpoints, are used to exchange the trust bundles with the foreign trust domains that might be owned by a different team or organization.

SPIRE (SPIFFE Runtime Environment) is a production-ready implementation of SPIFFE. As shown in Fig. 15, it has two major components: server and agent. The server is responsible for generating and managing the identities, and the agent is responsible for requesting the identities for different services that run on a node where the agent is deployed. This solution does make the overall federation process easy. However, it is expensive in terms of resources. When SPIRE is integrated with a service mesh, the SPIRE agent and server must be deployed along with the service mesh resources. Thus, increasing the overall cost of deploying and managing extra resources. Moreover, the two service meshes under consideration in this thesis already support SPIFFE identity. Thus, deploying a SPIRE Server to issue and manage identities would be an additional overhead, as the service mesh control plane already takes care of this task. This integration of SPIFFE with the service mesh will introduce more complexity and complicate the debugging process.

4.4 Summary

The three existing solutions discussed in this chapter come with operational complexity and increased cost of deployment. Therefore, a new solution is needed that provides the existing solutions' benefits and overcomes their limitations and drawbacks.

5 Proposed Solution Security Requirements

Service meshes aim to simplify the microservices security by shifting the authentication and authorization functionality to a common infrastructure layer [21]. They allow you to achieve zero-trust by following the principle of trusting nothing and verifying everything. Different features offered by service mesh, such as mTLS, authorization policies, and certificate rotation, make it possible to achieve a zero-trust network.

When it comes to federating service meshes, the primary concern is security. The federation of multi-cloud homogeneous and heterogeneous service mesh can have multiple use cases and implementation techniques; however, secure communication between different services must be prioritized in all use cases. The following security properties must be fulfilled when implementing a secure and efficient federation solution.

5.1 Authentication

A service that is part of a service mesh must prove its identity to other services [21] in order to prevent an attacker from impersonating any service over the network. The services part of the service mesh must exchange X.509 certificates to authenticate and then exchange encrypted information. If this security property is not met, then any malicious service can talk to the legitimate services and break the security of the federated system. Moreover, the certificates assigned to the services registered to a mesh must have a short lifetime. Also, they should be refreshed frequently to help limit service impersonation attacks.

Service mesh intends to provide a zero-trust network that allows building secure solutions on distrusted networks. In a zero-trust network, nothing is trusted, and every entity must prove its identity [32]. Consider a scenario where an attacker has managed to bypass the mesh and added a forged service with a certificate signed by a different CA. The legitimate services establishing a mTLS connection will fail to verify the certificate of this malicious service as it will be signed by a different CA. Consequently, the legitimate service drops the connection, thus, protecting from service impersonation attacks. Moreover, a service mesh, such as Istio, conducts a secure naming check before establishing a connection with any service. As explained in section 2.5.4, the secure naming process ensures that the client service extracts the identity of the server service and verifies if it is allowed to run a particular service. This also helps to limit the impersonation attack as a legitimate service will not be able to establish a connection with a malicious service.

When service meshes owned by different organizations are federated, there is always a need to exchange the trust bundles. The concept of trust bundles was already explained in Section 4.3. This exchange is done by establishing a connection between the control planes of the clusters and initiating an exchange. Thus, the connection between the control planes should be secured before exchanging the bundles.

5.2 Confidentiality

An attacker should not be able to access the information it is not meant to see, i.e., an attacker should not be able to decrypt encrypted network traffic being exchanged between a pair of services. A secure connection must be established between services to ensure authenticated and encrypted data exchange. An unauthorized party should never be able to access the data. The data should always be fully encrypted [43]. In the case of the federation, both service meshes that are being federated must have an encrypted exchange of data.

5.3 Limited exposure

The concept of the trust domain discussed in Section 4.3 can be used to achieve limited exposure to the services. The services that must be exposed should be part of a separate trust domain, and only the trust bundle associated with that separate domain should be shared with any external organization. This allows to only provide an external organization with access to the services that are part of the dedicated trust domain. Thus, protecting the mesh by providing limited access to an external third party. Other confidential services should be part of another trusted domain, and the external organization should not be able to access or authenticate with them until the trust bundle is explicitly exchanged. Thus, facilitating better separation of concern and limiting external access. Moreover, even in a single mesh, all the services should not have implicit access to each other. Only the required services should be able to access each other with the help of different rules and access control policies.

5.4 Integrity

The data being exchanged should not be illegally tampered with and must remain in its original form [43]. An attacker should not be able to modify a message sent over the network. Whether in a single mesh or cross-mesh, the data must remain in its original structure throughout its life cycle. When the data is successfully received by a service, it should always check if the data has not been altered in transit by an adversary sitting on the network.

5.5 Summary

In this chapter, we discussed the main security properties recommended by the National Institute of Standards and Technology (NIST) [21]. These properties must be considered before developing a solution that can resolve the challenges of the heterogeneous service mesh federation. Even minor security breaches during the federation process of meshes can lead to the whole mesh and system getting compromised. Thus, for the proposed solution to be acknowledged as secure and reliable, it is important that it fulfills all these security requirements. In Section 6, the proposed solution is evaluated against the main properties discussed in this section to examine the level of security it offers.

6 Proposed Solution

The service mesh technology is widely accepted by large organizations [42] due to several benefits, such as observability, security, reliability, and traffic management. Similar to every other new technology, service meshes also have some limitations. However, with the support of the open-source community, this technology has evolved over the years. When the security of microservices is a primary concern, the service mesh technology can be considered an efficient solution. This section proposes a potential solution that can help to resolve the lack of common CA problem discussed in Section 3.3.

6.1 Solution Infrastructure on AWS

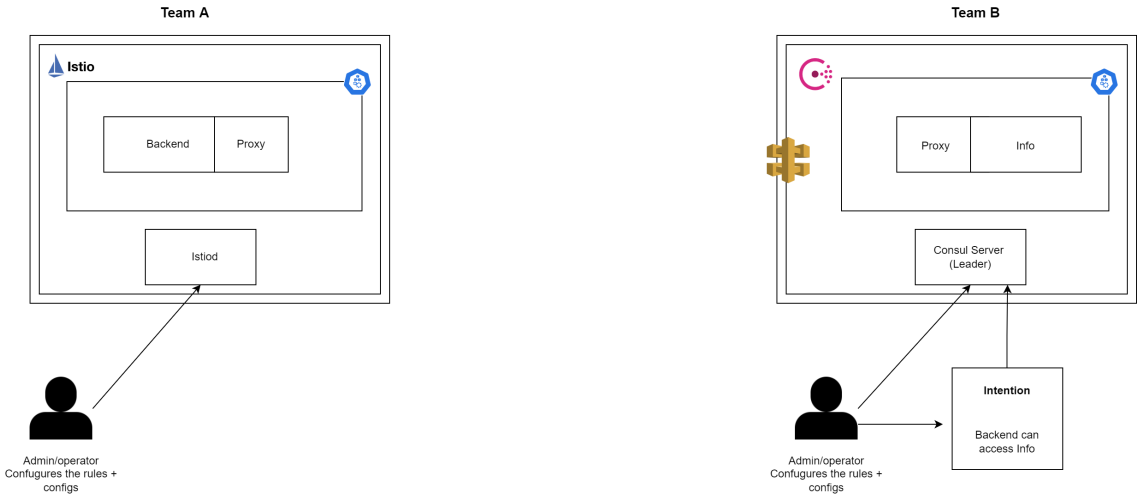

Figure 16: Proposed Solution Architecture

The infrastructure shown in Fig. 16 is deployed on Amazon Web Services (AWS) cloud platform. First, a Virtual Private Cloud (VPC) is created for the Istio service mesh. The VPC provides full access to a virtual networking environment. Different resources, such as EC2 instances, security groups, load balancers, and gateways can be launched in this logically isolated virtual network.

The next step is to create a Kubernetes cluster and attach the VPC created for Istio with it. Kubernetes is an open-source orchestration platform for managing containerized workloads and services. The main components of a Kubernetes cluster were already discussed in Section 2.2. Amazon Elastic Kubernetes Service (EKS) was used to create a Kubernetes cluster. This is a managed Kubernetes service to run and scale Kubernetes applications on AWS. EKS removes the overhead of manually deploying the Kubernetes control plane and worker nodes. EKS handles managing the Kubernetes control plane across multiple AWS availability zones. It automatically scales them based on the load. Moreover, it always runs the updated

Node group configurations	
Kubernetes version	1.20
AMI type	AL2_x86_64
AMI release version	1.20.15-20220629
Instance types	t3.medium
Disk size	100 GiB
Container runtime	docker://20.10.13
OS (Architecture)	linux (amd64)
Kubelet version	v1.20.15-eks-99076b2
OS image	Amazon Linux 2
Minimum size	2 nodes
Maximum size	3 nodes
Desired size	2 nodes
Capacity type	On-Demand

Table 3: Node group configurations

version of open-source Kubernetes software so that a developer can use all the latest tools and plugins from the Kubernetes community. The EKS control plane has a minimum of two instances of the Kubernetes API server exposed via the EKS endpoint associated with the cluster. The control plane can be accessed using kubectl tool to get information about the cluster.

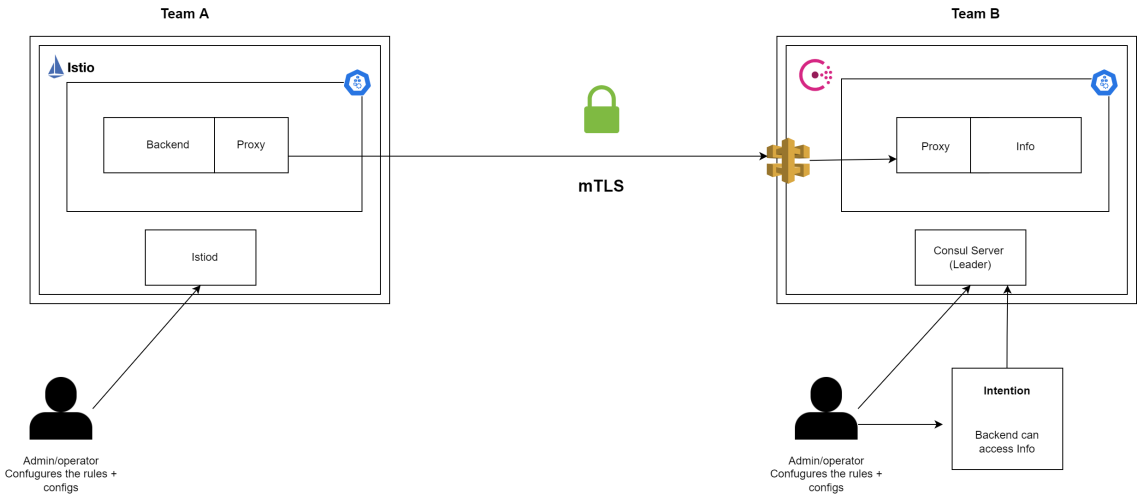

Figure 17: Heterogeneous service mesh integration

Once the cluster is successfully created, we add node groups to the cluster on which the application will be deployed. The node group contains EC2 instances on which the application will run. Table 3 describes the configurations of the node groups. Finally, when the cluster is ready, Istio is installed, and the backend service is deployed to the Istio EKS cluster. Using the same procedure discussed above, another VPC and EKS

cluster is created for the Consul service mesh. That is followed by installing Consul and deploying the info service. Later in Section 6.3, we discuss how the process of installing Istio and Consul was automated.

At the Consul side, an ingress gateway is additionally deployed to receive external requests from the backend proxy and forward them to the info proxy. When the backend service wants to request additional weather information from the info service, the proxy of the backend service will intercept the traffic. Then using the certificate assigned by the CA, it will send a mTLS request to the Consul ingress gateway. To directly send a request from the backend proxy to the gateway, a service entry must be created in the Istio service mesh. While configuring the service entry, the external IP of the Consul gateway must be specified to correctly direct the traffic to the gateway. As the control planes of Istio and Consul are not compatible, it is not possible to automatically expose the services to each other. Therefore, the service entry has to be manually configured at the Istio side to route the mTLS traffic to the external IP address of the Consul ingress gateway. The gateway must be configured with the TLS passthrough mode. This mode means the gateway will not decrypt the traffic and will simply pass it to the info service proxy. As seen in Fig. 17 the info service proxy will verify the certificate of the backend proxy and provide its own certificate to the backend proxy. If Istio and Consul mesh are configured with the same root CA, then a secure mTLS connection can be established between the services, and the certificates can be successfully verified as they are signed by the same root CA. However, if both the clusters have different root CA, then the mTLS connection will fail. A solution is needed to establish a common trust between these heterogeneous service meshes.

6.2 Exchanging Trust Bundles

Direct exchange of trust bundles is one possible solution to alleviate the problem of establishing trust between meshes with different CAs. As illustrated in Fig. 18, the direct exchange can be established by exposing HTTPS endpoints on the control planes of both Consul and Istio. Once Istio and Consul service mesh have exported the trust bundles of each other, their control planes will propagate the trust bundle to all the proxies using an active Remote Procedure Call (RPC) channel and xDS APIs. Finally, when the backend proxy sends a request to the info proxy, they will present their certificates to each other. After the trust bundle exchange process is complete, the verification of the certificates will be successful as the proxies are configured with each other's root CA certificate.

The trust bundle exchange frequency depends on the expiry time of the root CA certificates. Istio and Consul root CA usually have a long expiry time. However, it is recommended to have a short-lived certificate; in that case, the trust bundle exchange will happen more frequently depending on the expiration time.

This exchange of trust bundles allows these heterogeneous service meshes to integrate and make it possible for the backend service to request additional information from

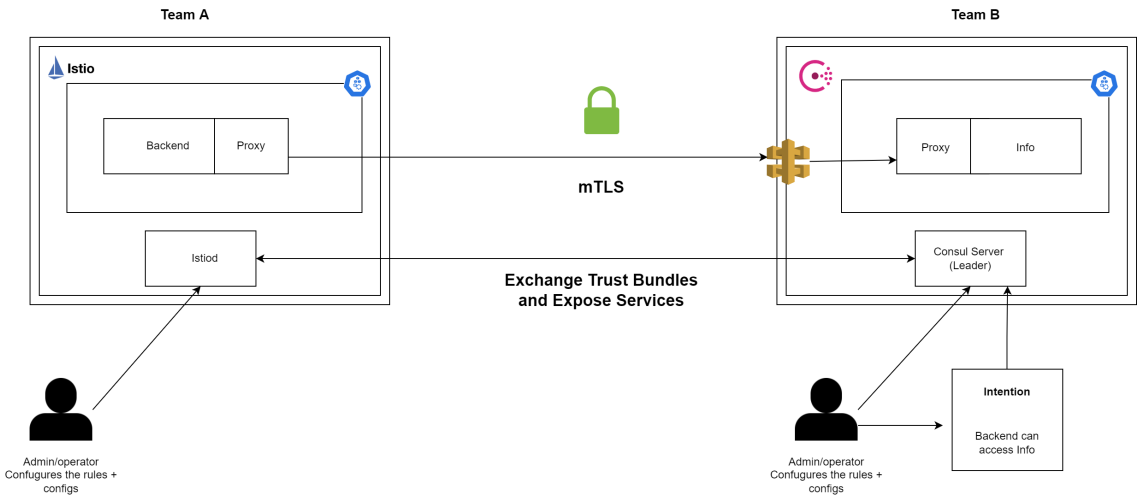

Figure 18: Trust bundle exchange

the info service. Thus, enabling access to cross-service-mesh services.

6.3 Automation Scripts

Setting up the EKS cluster and deploying worker nodes requires many manual steps. Since AWS bills the EKS service hourly, the clusters must be regularly destroyed and recreated. This work followed the infrastructure as a code (IaC) process to save time and cost.

Terraform, an open-source infrastructure as a code tool was used to create the EKS clusters and worker node groups automatically. Using tools like Terraform to achieve IaC offers the benefits of automating infrastructure management. The same resources can be created multiple times consistently without any errors caused by human intervention. Moreover, it allows recreating the same infrastructure on different cloud platforms, thus, improving multi-cloud infrastructure deployment. Using IaC also reduces the development cost as you can provision on-demand testing environments and resources.

6.4 Further Advancements For The Proposed Solution

The solution presented in the Section 6.1 still has some room for improvement and future work. Exposing the HTTPS endpoints on the control plane of Istio and Consul to exchange the trust bundle requires initial trust bootstrapping. HTTPS connection between the control planes would require them to initially prove their identity to each other with the help of certificates. First, you have to manually configure the trust bundle of the other mesh to have an initial connection. Once the initial connection is set up, the trust bundles can be updated using the exposed endpoints. Thus, this solution does not provide full automation of the trust bundle exchange process. Moreover, this leads to a bottom turtle problem where we need an additional secret

to protect an existing secret. Hence, a more automated solution is needed to resolve the bottom turtle problem that would not require the initial trust bootstrapping to exchange the trust bundles and to have a smooth and secure mTLS connection between services existing on different services meshes.

6.5 Summary

This chapter discusses the architecture of the proposed solution on AWS. The process of establishing a secure trust bundle exchange is explained. This chapter also highlights why Terraform is used to create the initial infrastructure of the proposed solution. Lastly, we discuss the limitations of the proposed solution and how they can be solved in the future to have a more efficient federation solution.

7 Solution Security Evaluation

This chapter performs the security evaluation of the solution described in the Section 6 that can help resolve the trust issues between heterogeneous service meshes. We analyze whether the solution proposed in the previous chapter fulfills the security requirements described in chapter 5. Furthermore, we discuss how the service mesh architecture can become vulnerable to different attacks when the security requirements are not met properly.

Despite the service mesh being a new technology, it is still widely adopted as a cloud-native mechanism for minimizing the operational complexity of the microservice architecture. In March 2020, the Cloud Native Computing Foundation report indicated that 42% of respondents are testing the use of service mesh on different cloud platforms, and 27% are using it for production environments [40]. The service mesh architecture is based on the zero trust model that follows the principle of never implicitly trusting any entity on the network. It recommends considering the network untrustworthy and as an adversary itself. Following this network security approach helps to achieve higher security than the traditional solutions [32]. This means that every entity on the network - whether internal or external - should always be authenticated. Without using a service mesh, it is difficult to achieve zero trust because it requires separate tools to manage certificates for the services, as well as for authentication and authorization. However, a service mesh removes this overhead by providing a component that acts as a CA and assigns identities across the mesh [49].

The service mesh provides an identity to each service that is part of the mesh. This identity is in the form of X.509 certificates assigned to the sidecar proxy injected with each service. The services utilize the certificates to authenticate with each other and prove their identity before initializing a connection. NIST strongly recommends establishing a mTLS connection for secure service-to-service communication that ensures every service proves its identity using the X.509 certificate assigned by the CA [21]. However, this exchange of certificates during the mTLS connection fails when two heterogeneous service meshes with different roots of trust are federated. Lack of common trust and different root CAs cause the certificate verification failure. The solution proposed in chapter 6 provides a mechanism to mitigate this problem by exchanging the trust bundles so that the services can verify the identity of the external services. After the service meshes export the trust bundles of the external service meshes they aim to integrate with, all the required services can communicate with each other securely by establishing a mTLS connection described in Fig. 4.

Enforcing mTLS across the meshes provides reliable and secure communication. It also offers confidentiality as the traffic between the services is always encrypted [21]. This pair of services trying to establish a connection exchange a symmetric shared key during the mTLS handshake, which is used to encrypt and decrypt the traffic. This encryption of traffic helps to prevent a man-in-middle attack. Any adversary eavesdropping between the two services will not be able to understand the encrypted

communication between the services. The adversary does not possess secret keys that can be used to decrypt the traffic. As discussed earlier, when the service mesh is deployed on a Kubernetes cluster, the control plane of the mesh takes away the responsibility of distributing certificates to the services and enforcing mTLS that encrypts all the requests and responses. Thus, preventing the services from eavesdropping attacks. It also helps to prevent attacks when a malicious service tries to impersonate a legitimate service. Since the malicious service spawned by an attacker does not have the correct X.509 certificate; therefore, it will not be able to authenticate to other legitimate services. Another strong recommendation by NIST is to have a short lifetime of the X.509 certificates assigned to services. It is preferable to have them on the order of hours. This also prevents impersonation attacks as the expired certificates can not be used by a malicious third party to impersonate a service. Moreover, when the certificates are short-lived, the overhead of checking the revocation status of the certificate using Certificate Revocation List (CRL) and Online Certificate Status Protocol (OCSP) can be avoided [19].

An infrastructure based on microservices and service mesh means an increased surface of attack and more opportunities for malicious parties to exploit; therefore, it is important to ensure that the workload only communicates using mTLS. It helps encrypt all the data exchange between the services distributed in the mesh. The use of mTLS to secure the traffic also helps to maintain the integrity of the data. During the mTLS handshake between the services, the two services decide the hashing algorithm that will be used to maintain the consistency and integrity of the data. Thus, this hash calculation that takes place during the handshake ensures that the data is not tampered with by an attacker or man-in-the-middle.

The proposed solution will solve the problems of establishing a secure mTLS connection when different service meshes, having different root CA, are federated. This ensures that services communicate securely even if they belong to another mesh. The enforcement of mTLS inside and across the mesh makes the services less vulnerable to different attacks, such as man-in-the-middle and service impersonation attacks. As a result, it makes service-to-service communication secure and reliable.

The proposed solution also ensures to utilize of the access control policies, Intentions for Consul, and Authorization policies for Istio, to provide limited exposure to the internal services of the service mesh. Meaning that only the service that has to be accessed by a third-party mesh should be exposed instead of providing full access to all the local services. By using authorization policies in Istio and Intentions in Consul, the implicit access to a service from any source can be denied, and only access from a specific source can be allowed. The developer or maintainer of the mesh should explicitly define these access control policies.

In the thesis, we consider a Backend service deployed in Istio and an Info service deployed in the Consul service mesh. We can limit the default access to these services by other services and only allow these two services to access each other in the following way:

Istio Authorization Policy:
This policy allows the info service to access the backend service only. The control plane of Istio is used to configure and propagate this authorization policy to the proxy. After the backend proxy has initialized a mTLS connection with the info proxy, the proxy will check whether it is authorized to communicate with the info proxy. In Istio, each proxy has an authorization engine that authorizes the requests at runtime [50]. The following policy will allow the backend proxy to only receive requests from the info proxy, any other request will be automatically declined.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-info"
  namespace: default
spec:
  selector:
    matchLabels:
      app: backend
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/info"]
```

Consul Intention:
The Intension configured in the Consul mesh will allow the info service to only receive a GET request from the backend proxy. All requests from any other service will be declined automatically. The consul control plane propagates these intentions to all the service proxies. When a request is received by a proxy, it will first verify these intentions.

```
apiVersion: consul.hashicorp.com/v1alpha1
kind: ServiceIntentions
metadata:
  name: allow-backend
spec:
  destination:
    name: info
  sources:
    - name: backend
      permissions:
        - action: allow
          http:
            pathPrefix: /
            methods: ['GET']
```

The access to the services can be further limited by having multiple trust domains in the mesh, as discussed in 4.3. Each trust domain will have a different trust bundle, which contains the public keys to verify the service that belongs to that specific trust domain. There should be a different trust domain for the exposed services, and only the trust bundle associated with that trust domain should be exchanged. Currently, Istio and Consul do not support multiple trust domains [48, 50]. However, other tools like SPIFFE can be used to achieve a separation of the trust bundles during the federation process.

7.1 Summary

Overall, the proposed solution meets most of the security requirements needed for achieving a secure and reliable solution. However, the proposed solution is accompanied by some drawbacks. It has the limitation of trust bootstrapping and does not support the multi-trust domain solution to limit the exposure to the services. However, it satisfies the main requirements, such as Authenticity, Confidentiality, Integrity and traffic-flow control between services.

8 Discussion

Microservice technology has revolutionized how cloud-native applications are designed and developed [37]. However, microservice technology is not without some drawbacks and limitations. Because the use of microservices fueled by DevOps increases the number of communication links to be protected and also the development complexity of the networking logic. This has led to the emergence of the service mesh, which is widely embraced as the de-facto standard for secure microservice communication and traffic management [42].

Recently, a shift has been witnessed towards using service mesh in production environments [40]. However, this emerging technology has some limitations that were discovered while working on the thesis. In this chapter, we will discuss how the limitations can be transformed into open areas for future research efforts related to heterogeneous service mesh federation.

8.1 Unified Access Control

While studying the concepts and workflows of the most popular service meshes, i,e., Istio and Consul, it was examined that these two service meshes support different methods to implement fine-grained access controls across the service mesh to limit the access to the services. Istio has authorization policies, and Consul uses Intentions to limit access to the services. However, these policies are not compatible when we talk about the integration of these meshes. The authorization policy configured for Istio is different from the Consul Intentions. Hence, unifying these access control policies for cross-mesh access and federation is necessary. Instead of manually configuring the same policies in distinct ways on the diverse meshes, there is a need for a common platform or an agent to unify these access control policies across meshes. This unification will help to achieve a more automated and efficient heterogeneous service mesh integration process that is turning out to be the need of today.

A policy engine similar to Open Policy Agent (OPA) can be evaluated as one of the approaches to achieving this unification of access controls regardless of the underlying technology. OPA is specifically designed for cloud-native environments. It acts as a unified tool-set and framework for policy across the cloud-native stack [46]. Instead of using different languages, models, and APIs to enforce access control policies across a service mesh, OPA can be used as an alternative way to unify fine-grained access policies across heterogeneous service meshes. This unification can be useful when service meshes get integrated, and access controls are needed to limit the full access to the services from different sources. As future work, it can be further researched whether OPA is compatible with the service meshes under consideration, i.e., Istio and Consul.

8.2 Local routing

When heterogeneous service meshes are deployed on the same cluster, the services deployed on separate meshes should be able to have direct connectivity with each other. They should not require an additional round trip to the gateway of the other service mesh. The workflow should be similar to a single cluster deployment model where a round trip to the gateway is unnecessary, and all the services have direct connectivity. This can help reduce the latency and make the service mesh more optimal for high-performance-driven distributed computing, where low latency can be quite beneficial. When two homogeneous service meshes deployed on the same cluster are integrated, they do not require an additional gateway to have cross-mesh access between services [50]. Therefore, more research can be conducted in the future to accomplish a similar mechanism that can enable local routing for heterogeneous service meshes deployed on the same network. This interesting area is still open for future research and advancements.

8.3 Debugging support

Debugging across service meshes becomes quite complicated when organizational boundaries exist between different service meshes. The boundary can also occur due to the freedom offered by DevOps, which may leads to DevOps team boundaries. In these cases, if any networking problem occurs, the developers or maintainers of the local mesh can not access the logs of the foreign mesh to debug and resolve the communication errors. They can only access the logs of the local mesh, and that information is inadequate to find the root cause of a problem. Therefore, a mechanism is needed to access the logs of the external meshes for easier debugging purposes. The service meshes support various telemetry and observability tools, such as Prometheus, Graffana, and inbuilt dashboards, to troubleshoot the errors. However, they are only confined to a single service mesh or homogeneous meshes owned by the same organization. We can not observe the logs of the heterogeneous meshes integrated on the dashboards supported by service meshes.

To have an efficient and smooth integration process between meshes, there is a great need for a solution that can act as a common tool or dashboard for accessing the logs of all the federated service meshes. Further research and development are needed to achieve easier access to the logs of any external service mesh. This can be achieved in several ways. For example, the external service mesh can expose a secure endpoint to access the logs, and some documentation can be provided to the developers of the local mesh, which can help them during the endpoint configuration. This endpoint configuration will allow the developers to have real-time access to the logs of the external mesh. However, it can lead to serious security concerns like leakage of confidential information from the logs. In that case, it will be important to ensure that only a specific category of logs are accessed by the other party that does not contain any sensitive information.

8.4 Summary

The service mesh is the most emerging and extensively adopted technology in today's technologically advancing world. However, it is still getting mature over time. The majority of the service meshes are open-source. This is one main reason the technology keeps evolving daily with cutting-edge features needed to implement diverse use cases, such as heterogeneous service mesh federation. Like any other emerging technology, this technology also has more room for improvement and research that can make it more optimal for implementing wide-ranging use-cases for diverse industries, such as telecommunication, finance, healthcare and education.

9 Conclusion

Even though the service mesh technology is young, it has been adopted by advanced organizations to help solve the problems of microservice architecture. According to the survey Voice of the Enterprise; DevOps, Workloads and Key projects 2022, it was observed that 16% of the companies have already shifted to service mesh technology across their entire IT infrastructure. 20% of the respondents have adopted it in their team internally, and 38% reported that they are planning to modernize their IT infrastructure by adopting the service mesh technology in the future [52]. The factors that drive this widespread adoption include secure microservices communication without making any changes in the application code and the level of observability offered by the meshes that result in improved performance. Together with the adoption of the service mesh, the technology itself is expanding quickly. As various organizations become familiar with this emerging technology, the requirements and feature requests to handle diverse use cases also escalate. An example of one intriguing yet complex use case is a heterogeneous service mesh federation.

As discussed in the thesis, integrating service meshes is not a simple task. Many challenges are faced when two organizations plan on integrating meshes to achieve interoperability between services. We proposed a solution to resolve one of the major hurdles faced while federating heterogeneous service meshes, i.e., lack of common trust between meshes. The solution was analyzed against different security parameters and was acknowledged to be secure. However, it is discussed in the thesis that the proposed solution is not 100% efficient in terms of scalability and automation because it still has the limitation of initial trust bootstrapping. Moreover, the thesis evaluates the drawbacks of the existing solutions that can be used alternatively to resolve the integration obstacles. We discussed that the drawbacks of the existing solutions are more computationally and operationally expensive than the proposed solution. Thus, performing an initial bootstrapping of trust is still a better choice than dealing with the expensive hitches of the existing solutions, such as cross signing, SPIFFE, and API gateways.

The thesis aimed to study the popular service meshes and propose a solution to make the overall federation process of heterogeneous service meshes more convenient and automated. The goal of the thesis has been reached as a secure solution is proposed that can facilitate the direct exchange of trust bundles between meshes. The solution was evaluated in terms of security. It was evaluated that it satisfies the security requirements for a secure and robust solution. As stressed several times, the service mesh technology offers many features and functionalities. However, after working on this thesis, it can be concluded that there is still more scope for improvement in debugging support and achieving a more automated federation process between heterogeneous service meshes.

References

[1] R. Perlman. "An overview of PKI trust models". In: *IEEE Network* 13.6 (1999). DOI: 10.1109/65.806987.

[2] Muhammad Ali Babar and Muhammad Aufeef Chauhan. "A Tale of Migration to Cloud Computing for Sharing Experiences and Observations". In: SECLOUD '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011. DOI: 10.1145/1985500.1985509.

[3] Laura White et al. "Understanding Interoperable Systems: Challenges for the Maintenance of SOA Applications". In: *2012 45th Hawaii International Conference on System Sciences*. 2012. DOI: 10.1109/HICSS.2012.614.

[4] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014.

[5] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. "Designing a Smart City Internet of Things Platform with Microservice Architecture". In: IEEE, Aug. 2015. DOI: 10.1109/FiCloud.2015.55.

[6] Rashmi Rai, Gadadhar Sahoo, and Shabana Mehfuz. "Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration". In: *SpringerPlus* 4 (1 Dec. 2015). DOI: 10.1186/s40064-015-0962-2.

[7] Fabrizio Montesi and Janine Weber. *Circuit Breakers, Discovery, and API Gateways in Microservices*. 2016. DOI: 10.48550/ARXIV.1609.05830.

[8] Chris Williams. *How a chunk of the web disappeared this week: GlobalSign's global HTTPS snafu explained*. Oct. 2016. URL: https://www.theregister.com/2016/10/15/globalsign_incident_report/.

[9] Dennis Gannon, Roger Barga, and Neel Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4 (5 Sept. 2017), pp. 16–21. DOI: 10.1109/MCC.2017.4250939.

[10] Uchechukwu Awada. "Hybrid Cloud Federation: A Case of Better Cloud Resource Efficiency". In: IEEE International Conference on Cloud Computing At: San Francisco, USA. July 2018.

[11] Tetiana Yarygina and Anya Helene Bagge. "Overcoming Security Challenges in Microservice Architectures". In: *IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018. DOI: 10.1109/SOSE.2018.00011.

[12] Ivana Podnar Žarko et al. "Collaboration mechanisms for IoT platform federations fostering organizational interoperability". In: *Global Internet of Things Summit (GIoTS)*. IEEE. 2018.

[13] J T Zhao, S Y Jing, and L Z Jiang. "Management of API Gateway Based on Micro-service Architecture". In: *Journal of Physics: Conference Series* 1087 (Sept. 2018). ISSN: 1742-6588. DOI: 10.1088/1742-6596/1087/3/032032.

[14] Chien-An Chen. "With Great Abstraction Comes Great Responsibility: Sealing the Microservices Attack Surface". In: *IEEE Cybersecurity Development (SecDev)*. 2019. DOI: 10.1109/SecDev.2019.00027.

[15] Wubin Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019. DOI: 10.1109/SOSE.2019.00026.

[16] Xing Li, Yan Chen, and Zhiqiang Lin. "Towards automated inter-service authorization for microservice applications". In: *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*. 2019.

[17] Nikita Yu Samokhin, Anatoly A Oreshkin, and Anton S Suprun. "Implementation of agent interaction protocol within cloud infrastructure in geographically distributed data centers". In: *Journal Scientific and Technical Of Information Technologies, Mechanics and Optics* 124.6 (2019).

[18] Rongxu Xu, Wenquan Jin, and Dohyeun Kim. "Microservice Security Agent Based On API Gateway in Edge Computing". In: *Sensors* 19.22 (2019). ISSN: 1424-8220. DOI: 10.3390/s19224905. URL: https://www.mdpi.com/1424-8220/19/22/4905.

[19] Calcote Lee Butcher ZACK. *ISTIO: Up and running*. O'REILLY Media, INC, USA, 2019.

[20] Amir Boroufar. "Software Delivery in Multi-Cloud Architecture". Master's Thesis. Politecnico di Torino, 2020.

[21] Ramaswamy Chandramouli, Zack Butcher, et al. "Building secure microservices-based applications using service-mesh architecture". In: *NIST Special Publication* 800 (2020).

[22] Consul. *The Life of a Packet Through Consul Service Mesh*. May 2020. URL: https://www.hashicorp.com/blog/the-life-of-a-packet-through-consul-service-mesh.

[23] Jie Cui et al. "Extensible Conditional Privacy Protection Authentication Scheme for Secure Vehicular Networks in a Multi-Cloud Environment". In: *IEEE Transactions on Information Forensics and Security* 15 (2020). DOI: 10.1109/TIFS.2019.2946933.

[24] D Feldman et al. "Solving the Bottom Turtle: a SPIFFE way to establish trust in your infrastructure via universal identity". In: *Sprint Lab, Nova Zelândia* (2020).

[25] Jing Gao et al. "Provisioning big data applications as services on containerised cloud: a microservices-based approach". In: *International Journal of Services Technology and Management* 26.2-3 (2020). DOI: 10.1504/IJSTM.2020.106744.

[26] Xiliu He and Fang Deng. "Research on Architecture of Internet of Things Platform Based on Service Mesh". In: *12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*. 2020. DOI: 10.1109/ICMTMA50254.2020.00164.

[27] Jens Hiller, Johanna Amann, and Oliver Hohlfeld. *The Boon and Bane of Cross-Signing: Shedding Light on a Common Practice in Public Key Infrastructures*. 2020. DOI: 10.48550/ARXIV.2009.08772.

[28] Anjali Khatri and Vikram Khatri. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[29] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks". In: *Journal of Cloud Computing* 9 (1 Dec. 2020). ISSN: 2192-113X. DOI: 10.1186/s13677-020-00194-7.

[30] Xi Ning Wang. *Architecture Analysis of Istio: The Most Popular Service Mesh Project*. Dec. 16, 2020. URL: https://www.alibabacloud.com/blog/architecture-analysis-of-istio-the-most-popular-service-mesh-project_597010.

[31] Oksana Baranova. "Multi-Tenant Isolation in a Service Mesh". Master's Thesis. Aalto University, 2021.

[32] Christoph Buck et al. "Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust". In: *Computers & Security* 110 (2021). DOI: https://doi.org/10.1016/j.cose.2021.102436.

[33] Byungkwon Choi et al. "PHPA: A Proactive Autoscaling Framework for Microservice Chain". In: *5th Asia-Pacific Workshop on Networking (APNet 2021)*. Shenzhen, China, China: Association for Computing Machinery, 2021. DOI: 10.1145/3469393.3469401.

[34] SR Dileepkumar and Juby Mathew. "Optimize Continuous Integration and Continuous Deployment in Azure DevOps for a controlled Microsoft. NET environment using different techniques and practices". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 1085. IOP Publishing. 2021.

[35] Farina Giandonato. "Enabling Service Mesh in a Multi-Cloud Environment". Master's Thesis. Politecnico di Torino, 2021.

[36] Mayank Gokarna and Raju Singh. "DevOps: A Historical Review and Future Works". In: *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*. 2021. DOI: 10.1109/ICCCIS51004.2021.9397235.

[37] Işıl Karabey Aksakalli et al. "Deployment and communication patterns in microservice architectures: A systematic literature review". In: *Journal of Systems and Software* 180 (2021). DOI: https://doi.org/10.1016/j.jss.2021.111014.

[38] Arne Koschel et al. "A Look at Service Meshes". In: *2021 12th International Conference on Information, Intelligence, Systems and Applications (IISA)*. 2021. DOI: 10.1109/IISA52424.2021.9555536.

[39] N. C. Mendonca et al. "The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture". In: *IEEE Software* 38.05 (Sept. 2021). ISSN: 1937-4194. DOI: 10.1109/MS.2021.3080335.

[40] Sagar Nangare. *Why the Service Mesh Will Be Essential for 5G Telecom Networks*. Mar. 2021. URL: https://thenewstack.io/why-the-service-mesh-will-be-essential-for-5g-telecom-networks/.

[41] Wojciechowski; ukasz et al. "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh". In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 2021. DOI: 10.1109/INFOCOM42981.2021.9488670.

[42] Aleksandra Obeso Duque et al. *A Qualitative Evaluation of Service Mesh-based Traffic Management for Mobile Edge Cloud*. 2022. DOI: 10.48550/ARXIV.2205.06057.

[43] Prateek Mishra et al. "Novel lightweight interactive IoT end device architecture with tight security intelligence confidentiality, integrity, authenticity and availability". In: *International Journal of System Assurance Engineering and Management* 13 (S1 Mar. 2022). DOI: 10.1007/s13198-021-01369-4.

[44] Guilherme Vale et al. "Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022. DOI: 10.1109/ICSA53651.2022.00015.

[45] Monica Vitali. *Towards Greener Applications: Enabling Sustainable Cloud Native Applications Design*. 2022. DOI: 10.48550/ARXIV.2201.11631.

[46] Open Policy Agent. *Policy-based control for cloud native environments*. URL: openpolicyagent.org.

[47] Vanson Bourne. *Annual APIs and Integration Report 2021*. URL: https://www.vansonbourne.com/work/14072001tc.

[48] Consul. *Consul Documentation*. URL: https://www.consul.io/docs.

[49] Google. *Service meshes in a microservices architecture*. URL: https://cloud.google.com/architecture/service-meshes-in-microservices-architecture.

[50] Istio. *Istio Documentation*. URL: https://istio.io/latest/docs/.

[51] Kubernetes. *Kubernetes Documentation*. URL: https://kubernetes.io/docs/concepts/.

[52] Aspen Mesh. *What Are Companies Using Service Mesh For?* URL: https://aspenmesh.io/what-are-companies-using-service-mesh-for/.

[53] OpenWeather. *Weather API*. URL: https://openweathermap.org/api/.

[54] Oryx. *Oryx API Gateway*. URL: http://www.oryx-gateway.net/OryxAPIGateway.html.

[55] solo.io. *GlooMesh Enterprise*. URL: https://docs.solo.io/gloo-mesh-enterprise/latest/.

[56] SPIFFE. *Secure Production Identity Framework for Everyone*. URL: https://spiffe.io/.

A Solution Infrastructure as Code

A.1 Baseline Infrastructure

The Istio EKS cluster on AWS can be created using Terraform. The following code is part of the eks-cluster-istio.tf file, which will create a separate cluster to deploy Istio service mesh.

Before applying this Terraform configuration, a VPC must be created with three subnets in three different availability zones. The ID of the newly created VPC and subnets has to be updated in the elk-cluster.tf file.

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  version         = "17.24.0"
  cluster_name    = "terraform-istio"
  cluster_version = "1.20"

  vpc_id  = "vpc1"
  subnets = ["subnet1", "subnet2", "subnet3"]

  node_groups = {
    terraform_ng = {
      min_capacity     = 2
      max_capacity     = 3
      desired_capacity = 2
      instance_types = ["t3.medium"]
    }
  }
  manage_aws_auth = false
}

data "aws_eks_cluster" "cluster" {
  name = module.eks.cluster_id
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks.cluster_id
}
```

To deploy the Consul EKS cluster on AWS, first create a new VPC with three different subnets in three different availability zones. Following code is part of eks-cluster-consul.tf file, which will create a separate cluster to deploy the Consul service mesh.

```
module "eks2" {
  source          = "terraform-aws-modules/eks/aws"
  version         = "17.24.0"
  cluster_name    = "terraform-consul"
  cluster_version = "1.20"

  vpc_id  = "vpc2"
  subnets = ["subnet1","subnet2", "subnet3"]

  node_groups = {
    terraform_ng = {
      min_capacity     = 2
      max_capacity     = 3
      desired_capacity = 2
      instance_types = ["t3.medium"]
    }
  }
  manage_aws_auth = false
}

data "aws_eks_cluster" "cluster1" {
  name = module.eks.cluster_id
}

data "aws_eks_cluster_auth" "cluster1" {
  name = module.eks.cluster_id
}
```

A.2 Install Istio

Once the Istio EKS cluster is successfully created, run the following command to update the kube-config and interact with the cluster.

```
aws eks --region region update-kubeconfig --name istio-cluster-name
```

The next step is to install Istio on the cluster. The following script named istio-install.sh can be run to achieve this.

```
#Install Istio
cd /
curl -L https://istio.io/downloadIstio | sh -
cd /istio-1.14.1
export PATH=$PWD/bin:$PATH
istioctl install -y
# Enable side-car proxy in the default namespace
kubectl label namespace default istio-injection=enabled
```

Run the script using the following commands

```
# Make the script executable
chmod +x istio-install.sh
# Run the script
./istio.sh
```

A.3 Install Consul

To install Consul on the Consul EKS cluster. Use the following consul-install.sh script. The script has the following commands

```
helm repo add hashicorp https://helm.releases.hashicorp.com
cd /
# use helm to install consul
helm install -f helm-consul-values.yaml consul hashicorp/consul \
--version "0.40.0"
```

Following is the content of the helm-consul-values.yaml file that will deploy the Consul client and servers in the cluster. It will also deploy an Ingress Gateway.

```
global:
  name: consul
  datacenter: dc1
server:
  replicas: 1
ui:
  enabled: true
  service:
    type: 'NodePort'
connectInject:
  enabled: true
controller:
  enabled: true
ingressGateways:
  enabled: true
  gateways:
    - name: ingress-gateway
      service:
        type: LoadBalancer
        ports:
          - port: 80
          - port: 443
          - port: 8080
```