POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Cloud Native Security in Service Mesh

Supervisors

Prof. FULVIO RISSO

Candidate

MARCO MANCO

Tutors

Dott.ssa BELLIO MARTINA

Dott. ABBALDO DANILO

Dott.ssa NOVELLINO MARIA

A.Y. 2021-2022 - October 2022

Summary

Over the last two decades, we have witnessed the advent of cloud computing which has led to the adoption of new design patterns. More and more companies have started developing cloud-native applications, thus loosely-coupled micro-services are deployed as separate run-time processes, instead of monolithic ones. The usage of containers allows for achieving this goal by removing the limitations of hosting physical hardware and operating system. Nevertheless, another key technology has made this possible: Kubernetes is a project that allows automating deployment, scaling, and management of containerized applications.

Nowadays, connectivity, reliability, observability, and security have become the new challenges for Dev and Ops teams. In fact, although the new design patterns allow scalability and high availability, developers are forced to consider and incorporate into their applications a whole stack of additional features. In order to solve this problem, an early approach was to provide these cross-cutting concerns in the form of a library that developers could use in their applications, but a new pattern took hold, namely the sidecar pattern. Service mesh implements the sidecar pattern and enables Dev and Ops teams to move connectivity, reliability, observability, and security at the infrastructure level and implements these capabilities in a declarative way, so application teams don't need to reinvent the wheel when it comes to cross-cutting non-functional requirements.

This thesis analyzes two of the main service mesh solutions, Istio and Kuma, acting on two levels: network security and monitoring of defined policies on latency. In particular, showing a series of possible configurations to implement a zero-trust architecture that ensures secure communications among micro-services, access control, and authorization, and finally monitoring the impact of the resources used to define policies on the response latency.

Moreover, a comparison between Kubernetes and OpenShift will show the advantages and disadvantages of both of them in terms of platform management, cost, effort, and functionalities.

Acknowledgements

Sono giunto al termine del mio percorso universitario e adesso vorrei ringraziare tutte le persone che mi hanno aiutato a raggiungere questo traguardo.

Un ringraziamento speciale lo dedico alla mia famigli, in particolare ringrazio i miei genitori che non mi hanno mai fatto mancare nulla in questi anni, incoraggiandomi a non mollare mai e a dare sempre il meglio di me per superare ogni singolo traguardo. Ringrazio i miei fratelli che sono riusciti a mantenere viva in me la passione per l'informatica, proponendomi progetti, interessandosi a ciò che facevo e incoraggiandomi sempre in tutto.

Ringrazio mia nonna che dalla sua casetta ha supportato il mio percorso con le infinite preghiere e rosari.

Ringrazio tutti i miei amici di giù con i quali sono cresciuto e maturato e con i quali spero di poter condividere le mie esperienze future. In particolare vorrei ringraziare Federico che nonostante il mio caratteraccio mi sopporta sin dalle scuole materne.

Ringrazio i miei compagni di avventura a Torino per il sostegno ricevuto in questi anni e per le mille risate e serate fatte insieme che hanno permesso di alleggerire la lontananza da casa.

Ringrazio Claudia che durante tutto il periodo di tesi mi ha supportato e sopportato e che ha corretto con molta pazienza ogni singolo paragrafo di questo lavoro.

Ringrazio tutto il team di Blue Reply che mi ha aiutato in questo studio, soprattutto Martina che è sempre stata disponibile e pazienti ad ogni mia domanda.

Ringrazio tutti i professori che hanno contributo alla mia formazione e il mio relatore prof. Fulvio Risso per i consigli e il supporto ricevuto in questi mesi.

Infine, vorrei dedicare un "grazie" a tutti coloro che non ho menzionato ma che mi sono stati vicini in questi anni.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XII

1 Introduction 1
- 1.1 Micro-services Architecture . . . 1
- 1.2 Why Service Mesh . . . 2
- 1.3 Goal of the Thesis . . . 2

2 Kubernetes 4
- 2.1 Container Orchestrators . . . 4
- 2.2 Kubernetes Architecture . . . 4
  - 2.2.1 Control Plane Components . . . 5
  - 2.2.2 Node Components . . . 6
- 2.3 Kubernetes Objects . . . 7
- 2.4 RBAC . . . 11
  - 2.4.1 Service Account . . . 12

3 OpenShift 13
- 3.1 Why OpenShift . . . 13
  - 3.1.1 OpenShift Client . . . 14
  - 3.1.2 Quality Assurance Process . . . 14
- 3.2 OpenShift Architecture . . . 15
  - 3.2.1 Deployment Models Supported . . . 15
  - 3.2.2 Prerequisites and Operating System . . . 16
  - 3.2.3 Machine Roles . . . 16
  - 3.2.4 Operators . . . 18
- 3.3 OpenShift vs Kubernetes . . . 18

4 Service Mesh 21
4.1 ZTA - Zero Trust Architecture . . . 21
4.1.1 Zero Trust Networking Principles . . . 21
4.1.2 Applying Zero Trust Principles to Kubernetes . . . 22
4.2 Issues of Micro-services Approach . . . 22
4.3 Features . . . 22
4.4 How It Works . . . 23

5 Kuma 25
5.1 The Main Idea Behind . . . 25
5.2 Architecture . . . 25
5.2.1 Control Plane . . . 26
5.2.2 Data Plane Proxies . . . 26
5.3 Networking . . . 27
5.3.1 Service Discovery . . . 28
5.3.2 Gateway . . . 28
5.4 Policies . . . 30
5.5 Security . . . 31
5.5.1 Mutual TLS . . . 31
5.6 Monitoring . . . 32
5.6.1 Traffic Metrics . . . 32

6 Red Hat OpenShift Service Mesh 35
6.1 What is in . . . 35
6.2 Differences between Istio . . . 36
6.3 The Red Hat Advantages . . . 37
6.4 Istio Architecture . . . 38
6.5 Traffic Management . . . 39
6.5.1 Istio Gateway . . . 39
6.6 Security . . . 42
6.6.1 Istio Identity . . . 43
6.6.2 Authentication . . . 44
6.6.3 Authorization . . . 45
6.7 Observability . . . 46

7 Istio vs Kuma on OpenShift Container Platform 48
7.1 Demo Application - Bookinfo . . . 48
7.2 Apply Communication Encryptions . . . 49
7.3 Apply Authentication and Authorization for Service-to-Service Communication . . . 51
7.4 Apply a Naïve Deployment Strategy . . . 54

8 Ingress Controllers 58
8.1 Istio Ingress Gateway 58
8.2 Kuma Gateway 60
8.3 Kong Ingress Controller 60
8.3.1 Running the Kong Ingress Controller with OpenShift Service Mesh 61
8.3.2 Running the Kong Ingress Controller with Kuma 61

9 Latency Evaluation 63
9.1 Load Generator 64
9.2 Tests 64
9.2.1 First test case 66
9.2.2 Second test case 67
9.2.3 Third test case 69

10 Conclusion 71

Bibliography 73

List of Tables

9.1 Users' Latency and RPS 65

List of Figures

1.1 Before and After Service Mesh. 2

2.1 Components of a Kubernetes cluster [2]. 5
2.2 Kubernetes master and worker nodes [2]. 7
2.3 Kubernetes objects. 10
2.4 Kubernetes Operator [4]. 11

3.1 OpenShift PaaS [5]. 13
3.2 Red Hat OpenShift Container Platform [5]. 15
3.3 Red Hat OpenShift Container Platform infrastructure [5]. 16

4.1 Service Mesh. 24
4.2 Service Mesh Architecture. 24

5.1 Kuma architecture [12]. 26
5.2 How Kuma works [12]. 27
5.3 Kong API Gateway [12]. 29
5.4 Service Map Dashboard. 33

6.1 Istio Architecture [19]. 38
6.2 Basic flow of a request. 41
6.3 Istio Security Architecture [19]. 43
6.4 Identity Provisioning Workflow [19]. 44
6.5 Authorization Policy Precedence [19]. 46
6.6 Kiali Topology View. 47

7.1 Bookinfo Application [21]. 49

8.1 Ingress Architecture. 60

9.1 Proxy policies and components. 63
9.2 Response time and RPS. 65
9.3 Response time first test case. 67

9.4 Response time second test case. 68
9.5 Response time third test case. 69

Acronyms

API
: Application Programming Interface

CR
: Custom Resource

CSR
: Certificate Signing Request

CRD
: Custom Resource Definition

CVO
: Cluster Version Operatorn

CNI
: Container Network Interface

DNS
: Domain Name System

gRPC
: gRPC Remote Procedure Calls

HTTP
: HyperText Transfer Protocol

IP
: Internet Protocol

K8s

Kubernetes

JWT

JSON Web Token

mTLS

mutual Transport Layer Security

OC

OpenShift client

OLM

Operator Lifecycle Manager

ODO

OpenShift DO

OTA

Over-the-air

Paas

Platform-as-a-Service

PEP

Policy Enforcement Point

RHEL

Red Hat Enterprise Linux

RHCOS

Red Hat Enterprise Linux CoreOS

RBAC

Role-based access control

RPS

Requests Per Second

ROSA

Red Hat OpenShift Service on AWS

S2I

Source-to-image

SLB

Server Load Balancer

SDS

Secret Discovery Service

SCC

Security Context Constraints

TCP

Transmission Control Protocol

TLS

Transport Layer Security

ZTA

Zero Trust Architecture

Chapter 1

Introduction

Over the years, innovation in ICT has increased significantly. From the first years of the adoption of virtualization to now, many technologies have been developed and many are being developed.

One of the hottest topics of the moment is container orchestration and the whole world around it. In fact, with the advent of the container, medium and large companies had to manage large computing clusters. From here, Kubernetes was born.

Kubernetes is a system for managing containerized applications in a clustered environment. It became open-source in 2014 and a year later, Google and Linux joined together to form the Cloud Native Computing Foundation with the aim of maintaining the Kubernetes platform.

The expansion of Kubernetes is rapidly increasing among companies, and with it, new enterprise solutions were born to facilitate its use and provide new services.

Moreover, the adoption of these new technologies has not only changed the way applications are managed, but also the way they are designed.

1.1 Micro-services Architecture

A micro-services architecture is an approach to realize a single application as a set of small services, little coupled to each other and each running in its own process.

The adoption of this type of architecture brings with it many advantages. First and foremost, since these services are developed around the domain's business capabilities, distributed independently, and in a fully automated way, they allow the use of different programming languages as well as different storage technologies. In addition, it lets you split development teams into smaller sizes on individual services and new application components can be created to adapt to changing business needs. Finally, the development and redeployment of individual services

are also faster. But these are just some of the advantages introduced by this design architectural pattern compared to a monolith approach where the whole application is composed of a single source code.

1.2 Why Service Mesh

The rise of this new trend to develop cloud-native applications based on micro-services has led to the development of new technologies such as Service Mesh.

Service mesh is a software infrastructure layer built on a micro-service architecture to provide applications observability, security, and connectivity. It ensures that communication across pods is secure, fast, and encrypted by working on a platform layer, rather than an application layer.

To do this, Service mesh groups these additional functionalities into a sidecar application called sidecar proxy, and through the help of a control plane, service mesh can automatically configure and inject these proxies into each pod.

Removing these cross-cutting concerns from the main application, Service Mesh helps developers to focus only on the application's business logic.

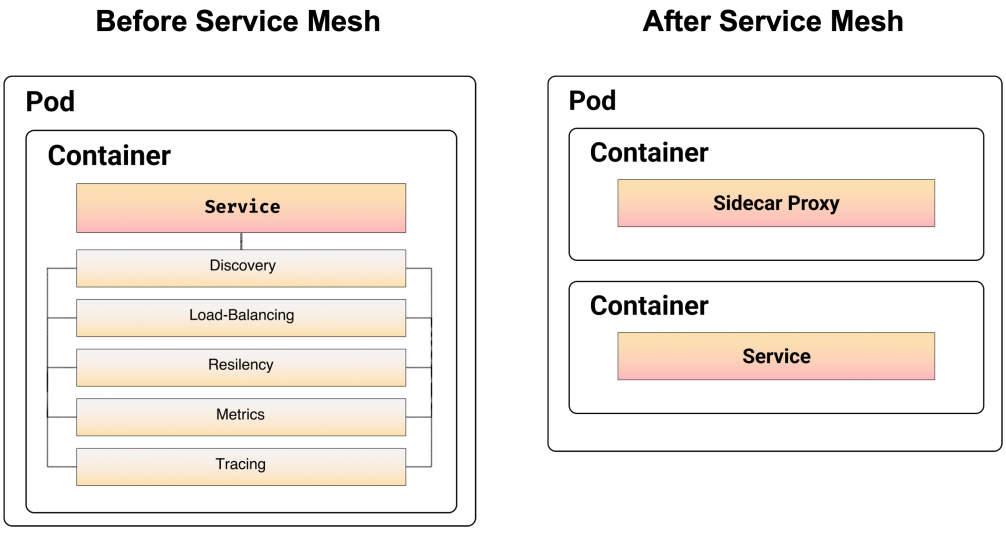

Figure 1.1: Before and After Service Mesh.

1.3 Goal of the Thesis

One of the biggest challenges in developing cloud-native applications today is speeding up the number of your deployments [1]. But more frequent releases can negatively affect application reliability. Thus, it's important that operations

and DevOps teams develop processes in order to monitor and handle internal communication between micro-services and to automate deployment strategies that minimize risk to the product and customers.

Nowadays, aspects such as security, monitoring, reliability, and load balancing are increasingly configured in a network using a Service Mesh. This new technology allows separating cross-cutting concerns from the main application by implementing a sidecar pattern.

In this thesis work, we demonstrate how an enterprise orchestrator of containers can help a company in order to increase security aspects, get a user-friendly experience, integrate commercial tools, and so on. Moreover, we understand in depth what a Service Mesh is and how it works, in order to compare two of the most popular solutions of Service Mesh, Istio and Kuma, in different use cases. In particular, we analyze how a company can use these two tools in order to:

- Deliver identity-based service-to-service access and communication: based on service identity, the mesh can authenticate and authorize service-to-service access and communication, in order to mitigate and avoid attacks such as impersonation, unauthorized access, etc.
- Deliver communication encryptions.
- Deliver a naïve deployment strategy.
- Handle the accessibility of the services from outside.

Moreover, we analyze how a company that manages its services through an Ingress Controller such as Kong Ingress Controller, can integrate these tools into its system to create a zero-trust network.

Finally, we analyze the impact on the response in terms of latency when the policies used to encrypt communication and handle access control on workloads are enabled.

Chapter 2

Kubernetes

In this chapter, we illustrate the framework Kubernetes, presenting its main concepts and components because Kubernetes technology is the one behind all the work exposed in this thesis work.

2.1 Container Orchestrators

Thanks to the rise of lightweight virtualization solutions, more and more companies had to deal with large clusters of containers, so it became essential to find a way to manage them.

Container orchestrators are born to solve this problem, in fact, they are designed to easily manage from one central place complex deployments of containers across multiple machines. One of the most popular tools for this purpose is Kubernetes. It was born from a Google's inside project (Borg) and released with open-source licenses in 2014.

Moreover, Kubernetes provides also some edge functions, such as load balancer, service discovery, storage orchestration, Role-Based Access Control, and so on.

2.2 Kubernetes Architecture

Kubernetes cluster architecture is made up of one or more nodes, each one may be a virtual or physical machine.

Each node can play the worker role or the master role (control plane). The control plane node has the task of managing the various worker nodes, whereas, the worker nodes contain all the necessary to run workloads [2].

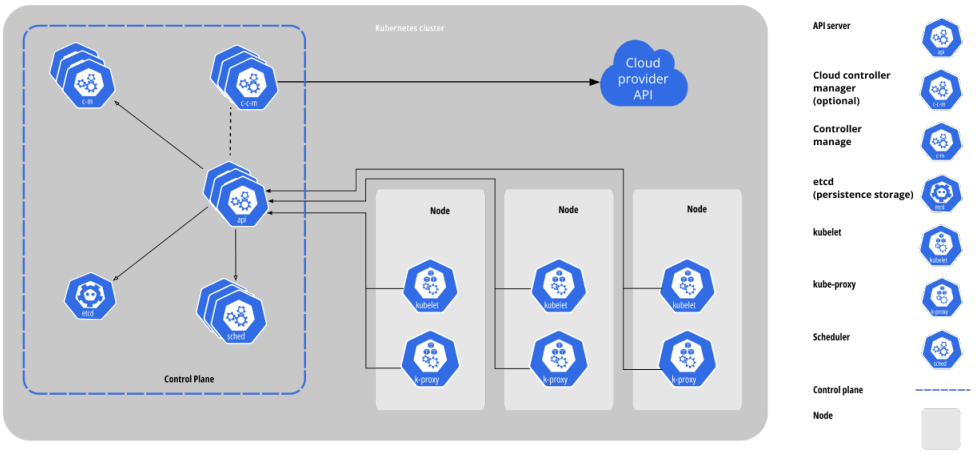

Figure 2.1: Components of a Kubernetes cluster [2].

2.2.1 Control Plane Components

As the name says, a control plane has the task of managing and making global decisions for the entire cluster. It can be run on any machine in the cluster, but generally, its components are executed on the same machine.

API Server

This component exposes the Kubernetes API in the REST format, thus it plays the front-end role for the Kubernetes control plane and it can be scaled horizontally.

Etcd

It is a storage component used by K8s in order to store all cluster data in a format of key-value.

Since etcd writes data to disk, its performance strongly depends on disk performance.

Moreover, this component is deployed in high availability.

Scheduler

When a new pod has to be created, a component of the control plane called Scheduler has the task to watch all nodes and to select one of them in order to run the pod.

In order to perform this task, the scheduler has to consider: resource requirements, hardware/software/policy constraints, affinity, etc.

Kube Controller Manager

This component is responsible for moving the current state toward the desired state. To do so, it watches the shared state of the cluster through the API Server and makes changes.

There are more types of controllers (for instance, node controller, job controller, etc), all of them compiled into a single binary and run in a single process.

Cloud Controller Manager

In order to link your cluster to your cloud provider's API, a cloud controller manager is used. It separates the components that interact with the cloud platform from components that only interact with your cluster.

2.2.2 Node Components

As for the control plan node, every node also needs some components to run and keep pods running, and provides the Kubernetes runtime environment.

Kubelet

This component makes sure that containers are running in a Pod, in fact, the kubelet takes a set of PodSpecs and ensures that the containers described in those PodSpecs are running and healthy.

Kube-proxy

Kube-proxy is a network proxy. It is used to maintain network rules on nodes which network rules allow network communication to your Pods.

Moreover, it implements part of the Kubernetes Service concept.

Container Runtime

In order to run containers, each node needs a container runtime.

To fulfill this task, any container runtime that implements the Kubernetes CRI (Container Runtime Interface) is supported, like contained, or CRI-O.

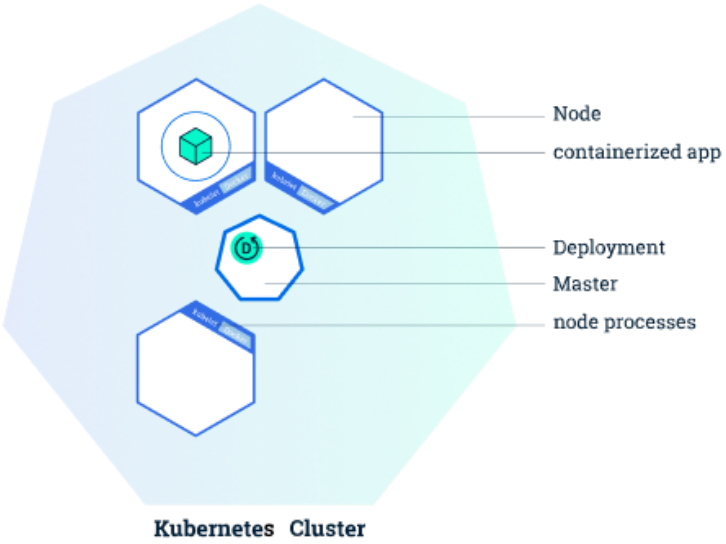

Figure 2.2: Kubernetes master and worker nodes [2].

2.3 Kubernetes Objects

Kubernetes provides many predefined build blocks (Kubernetes objects) used to represent the state of your cluster.

In order to interact with these K8s objects you need to use the Kubernetes API or a client like kubectl that makes the necessary Kubernetes API calls for you.

Each Kubernetes object is defined and described in a declarative way. Nearly all Kubernetes objects contain two nested object fields that govern object configuration:

- spec: It provides a description of the characteristics you want the resource to have.
- status: It describes the current state of the object. It is supplied and updated by the Kubernetes system.

Moreover, each object requires some additional fields:

- apiVersion: It describes the version of the Kubernetes API used to create the object.
- kind: It describes what kind of object you want to create.
- metadata: This field helps uniquely identify the object, including a name string, UID, and an optional namespace.

As we said before, Kubernetes offers many build blocks, below we present some of the most important.

Namespace

In order to isolate groups of resources within a single cluster, Kubernetes offers the namespaces resource. In fact, namespaces are used to logically divide a cluster, and them cannot be nested inside one another .

Every namespace provides a scope for names and within it each resource name needs to be unique, but not across namespaces.

Moreover, namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc) [2].

Pod

In Kubernetes, the smallest deployable unit is the pod. A Pod can group one or more containers in order to share storage and network resources, and a specification for how to run the containers [2]. Therefore, the shared context of a Pod is a set of Linux isolation mechanisms, namely the same things that isolate a Docker container (namespaces, groups, ...) but unlike Docker, Kubernetes supports more container runtimes.

In terms of Docker concepts, a Pod is similar to a Docker container group with shared file-system volumes and shared namespaces.

ReplicaSet

In order to maintain a stable group of replica Pods running all the time, Kubernetes uses the ReplicaSet resource.

When an operator specify a number of replicas required a ReplicaSet fulfills its purpose of maintaining the number of replicas constant by creating or deleting the necessary pods.

A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field [2].

Deployment

A Deployment resource is used to provide the desired state in a declarative way of Pods and ReplicaSets.

With the definition of a Deployment, it automatically creates a ReplicaSet, which as we have seen, is responsible to creates and maintaining the desired number of pods. To do so, the spec. replicas field is specified in the Deployment resource. If spec. replicas field is not defined, the ReplicaSet instantiates just a replica of the

desired application.

Therefore, when working with Kubernetes, an application is generally executed within a Deployment and not in a simple Pod.

Service

In order to expose an application running on a set of Pods, Kubernetes introduces a resource called Service. This resource groups all Pods belonging to an application and balance the load among them.

The motivation behind this design pattern is that Pods are nonpermanent resources, in fact, they are dynamically created and destroyed to match the desired state of the cluster. This leads to a problem: How do you find out and keep track of which IP address to connect to?

To do so, Kubernetes gives Pods their own IP addresses and groups all Pods that belong to an application to a single DNS name.

Therefore each Pod can be associated with a service. Services most commonly abstract access to Kubernetes Pods thanks to the selector.

The Service resource allows you to specify how the application should be exposed via`spec.type` field.

By default, each application is exposed with `ClusterIP` type, but also other types are possible:

- **ClusterIP:** It exposes the Service on a cluster-internal IP, but in this way the Service is only reachable from within the cluster.

- **NodePort:** It exposes the Service on each Node's IP at a static port. Unlike the latter method, the Service is reachable from outside, by requesting <NodeIP>:<NodePort>.

- **LoadBalancer:** Exposes the Service externally using a cloud provider's load balancer. Moreover, in this way, NodePort and ClusterIP Services are automatically created in order to route the external load to them.

- **ExternalName:** Maps the Service to the contents of the `externalName` field, by returning a CNAME record with its value.

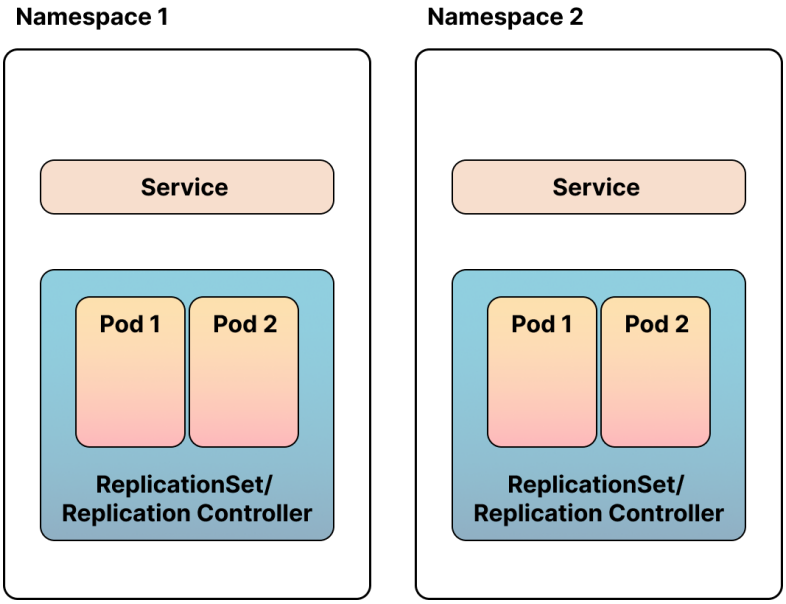

Figure 2.3: Kubernetes objects.

Label, Selector & Annotation

As we have already seen for the Services, in Kubernetes the objects are often selected and attached to other objects. To do this, each Kubernetes object has to specify identifying attributes (by using some labels) that are meaningful and relevant to the system but do not directly imply semantics to it.

Labels are key/value pairs, and each key must be unique for a given object. Labels can be used to organize and select subsets of objects. Furthermore, they can be attached to objects at creation time and subsequently added and modified at any time.

Unlike names and UIDs, labels do not allow uniqueness for objects. In general, many objects carry the same labels. In fact, via a label selector, the client can identify a set of objects.

Kubernetes supports two types of selectors: equality-based (=,==, and !=) and set-based (in, notin, and exists). Moreover, a label selector can be made of multiple requirements, each one separated by a comma that acts as a logical AND operator.

Differently from labels, Annotations are used to attach arbitrary non-identifying metadata to objects.

The metadata specified in an annotation can be of variable size, structured or unstructured, and can include characters not permitted by labels.

Operator

One of the most important objects in Kubernetes is the Operator. A Kubernetes operator is an application-specific controller that expands the functionality of the Kubernetes API to create, configure, and manage instances of elaborate applications [3].

An operator uses custom resources, which provide high-level configuration and settings, to manage applications and their components.

A custom resource definition defines a CR and lists out all of the configurations available. Therefore, the operator translates the high-level directives into the low-level actions to make the current state match the desired state in the resources. Moreover, an operator implements control loops that repeatedly compare the desired state of a resource to its actual state. If the resource's actual state does not match the desired state, then the operator takes action to fix the problem.

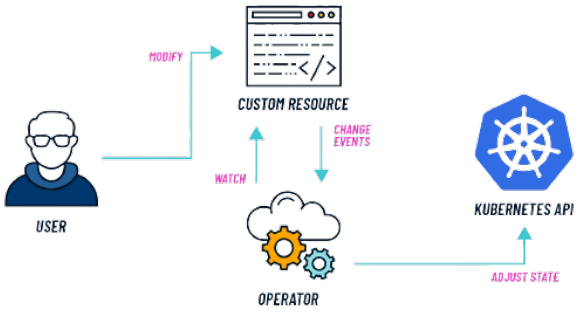

Figure 2.4: Kubernetes Operator [4].

2.4 RBAC

Kubernetes uses a role-based access control method to regulate access to resources based on the roles of individual users within the organization.

The API group defined by Kubernetes, for the management of accesses and authorization decisions, identifies four different object types to define permission:

- **Role:** It defines rules valid for a specific namespace. No DENY rules can be defined because permissions are purely additive.
- **ClusterRole:** It defines rules valid for the entire cluster. No DENY rules can be defined because permissions are purely additive.

- **RoleBinding:** It is used to link an identity to a set of rules in a specific namespace.
- **ClusterRoleBinding:** It is used to link an identity to a set of roles in all namespaces.

2.4.1 Service Account

The ServiceAccount is a Kubernetes object used to provide an identity for processes that run in Pods.

As for User Account, which is used to authenticate a user when access to the cluster; processes of the containers inside pods can be authenticated as a particular Service Account.

A new certificate is created by the API Server every time a new ServiceAccount is created. This certificate is used, next, for the authentications.

Chapter 3

OpenShift

In the previous chapter, we analyzed Kubernetes, dissecting its architecture, its resources, and how it works. In this chapter, we will instead focus on Red Hat OpenShift Container Platform, an open-source enterprise platform for container application development, deployment, and management. We will clarify the reasons for using an enterprise application and, after illustrating its architecture, we will compare the latter with Kubernetes.

3.1 Why OpenShift

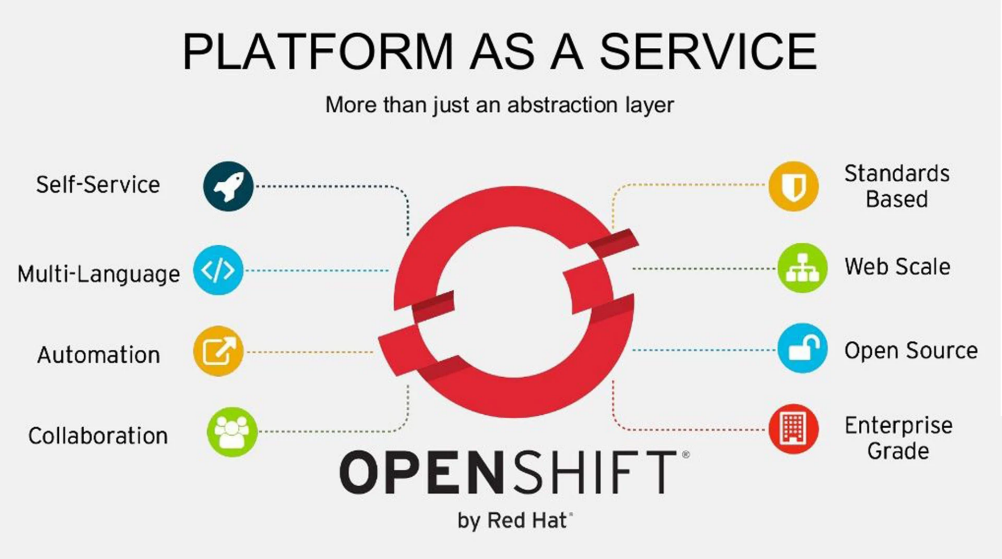

Figure 3.1: OpenShift PaaS [5].

Before entering the core of this chapter we must point out that since OpenShift is an enterprise Kubernetes application platform, the OpenShift cluster APIs are hundred percent compliant with respect to Kubernetes and OpenShift also does not apply any changes to a cluster running on K8s compared to being run on it [6].

Moreover, as of 27 April 2020, the name *Red Hat OpenShift Container Engine* was changed to *Red Hat OpenShift Kubernetes Engine* to better communicate the value of the offering product [5].

Therefore, now that we have clarified that the heart of OpenShift is Kubernetes all that remains is to see what makes it a successful enterprise PaaS.

3.1.1 OpenShift Client

OpenShift brings a set of command-line features by means of tools like oc and odo. For instance, oc is the OpenShift command-line and, just like kubectl, allows you to manage the platform, but adds several features: in fact, this includes the ability to build container images and deploy applications directly from the source code or binaries [6].

3.1.2 Quality Assurance Process

Since Red Hat supports many customers with OpenShift and many of these for business-critical scopes, the new features of Kubernetes before being integrated pass through three main stages:

- Preview phase.
- Tech Preview phase.
- General Availability Phase.

This is done purposely to get additional quality assurance within the OpenShift release cycle.

Moreover, Red Hat provides support for a much longer period of its OpenShift releases [6].

3.2 OpenShift Architecture

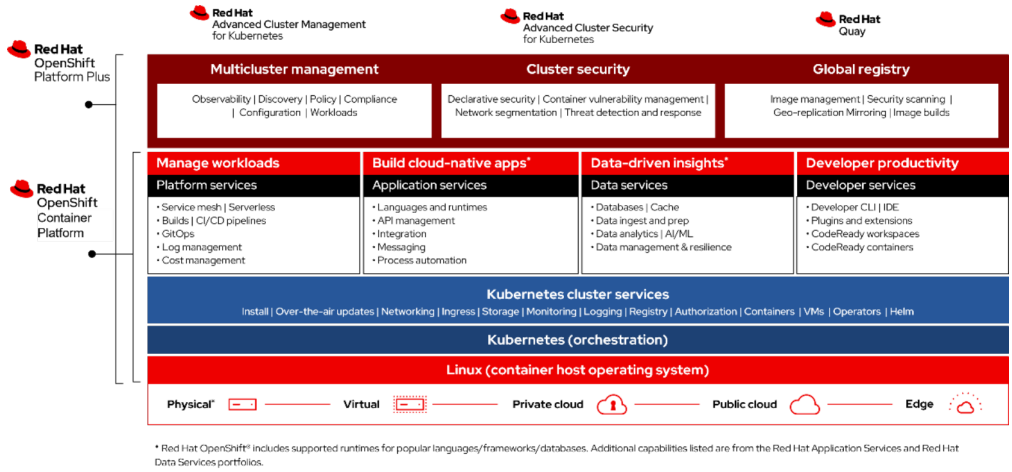

Figure 3.2: Red Hat OpenShift Container Platform [5].

Red Hat OpenShift Container Platform is based on an architecture of distributed system made of different components. Each component of the system can be installed in high reliability, and the default HAProxy load balancer can be used to create a multi-master and multi-etcd cluster environment. Since the foundation of OpenShift is based on K8s, they share the same technology. In addition, OpenShift exploits several other open-source projects and technologies so that development teams can gain:

- A more secure and performing enterprise-grade Kubernetes distribution.
- A single, integrated platform that provides monitoring, logging, and analytics solutions, along with build and deploy automation through CI/CD to supercharge productivity [7].

And many others.

3.2.1 Deployment Models Supported

Red Hat OpenShift is distributed in many different environments as PaaS, in fact, OpenShift Container Platform can be considered as the core component that offers a cloud-based Kubernetes container platform offered by Red Hat for customers with on-premise and/or hybrid infrastructure.

Among the best-known models there are:

- **OpenShift Dedicated:** it is a complete OpenShift Container Platform cluster provided as a cloud service, configured for high availability, and dedicated to a single customer. Moreover, It is managed by Red Hat and hosted on Amazon Web Services (AWS) or Google Cloud Platform (GCP).
- **Red Hat OpenShift Service on AWS:** ROSA is a fully-managed, turnkey application platform. It provides seamless integration with a wide range of AWS services.
- **Red Hat OpenShift on IBM:** it is a high-availability OpenShift cluster hosted on the IBM Cloud.

3.2.2 Prerequisites and Operating System

OpenShift is designed to work in different environments: private, public, and hybrid cloud, with physical o virtual machines. At the very bottom layer of the OpenShift, there is the operating system, which supports only RHEL, but lock-in customers to the Red Hat platform. Still, along with the subscription of the OpenShift bundled product, Red Hat provides a subscription to its operating system.

3.2.3 Machine Roles

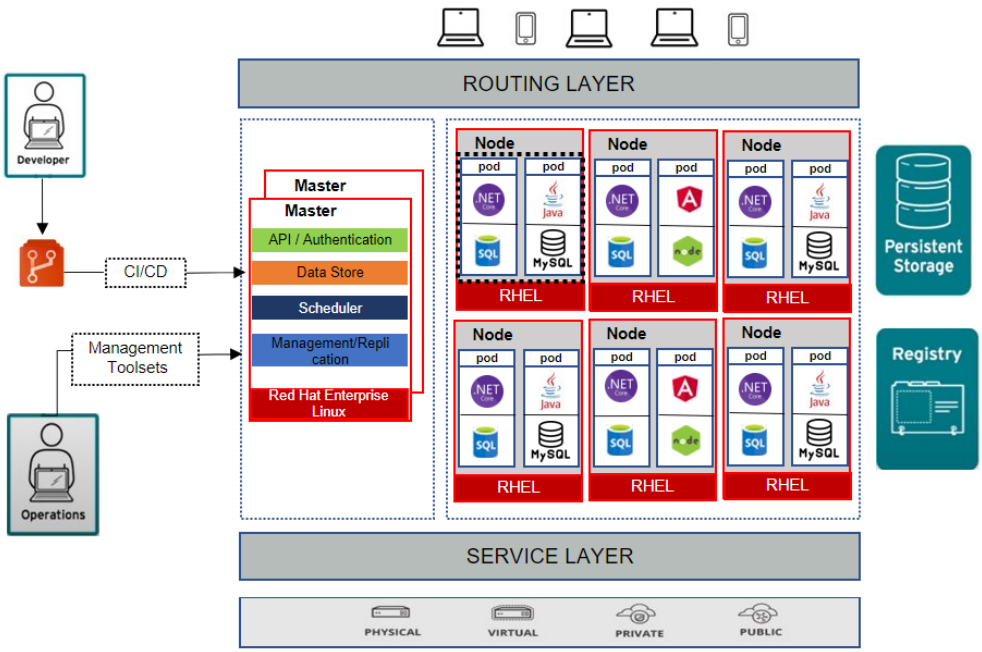

Figure 3.3: Red Hat OpenShift Container Platform infrastructure [5].

In the OpenShift Container Platform, each host has different roles in order to define its functionality within the cluster. Two main definitions are expressed in the cluster, namely for the standard master and worker role types [5].

Master Role

The master or control plane is the node responsible to manage the OpenShift Container Platform cluster; it controls and manages a set of worker nodes using the same components adopted in K8s:

- Kubernetes API server
- Data-store - ETCD
- Scheduler
- Kubernetes controller manager

Obviously, the master's version of OpenShift must match the one of its worker's nodes, and just temporary mismatches during cluster upgrades are acceptable.

Unlike Kubernetes, in the OpenShift Container Platform, the control plane machines contain more than just the Kubernetes services for managing the OpenShift Container Platform cluster. In fact, a master must contains the OpenShift services:

- OpenShift API server
- OpenShift OAuth API server
- OpenShift OAuth server
- OpenShift controller manager

And these services may run either as systemd services or as static pods [6].

Worker Role

Unlike Kubernetes, OpenShift can support many kinds of machines. For instance, the *worker machines* are classed as *compute machines*. This type of machine manages and runs the requested workload. All worker machines are controlled by a *machine set*, namely groupings of machine resources (under the machine−api namespace) designed to start new machines on a specific cloud provider.

3.2.4 Operators

One of the most important component of OpenShift is the Operator, used for packaging, deploying, and managing services on the control plane. They provide a way to manage OTA updates, monitor applications, perform health checks, and, ensure that applications remain in your specified state.

Although in K8s and OpenShift both definitions of Operator follow the same concepts and objectives, the operators of the OpenShift container platform are managed by two different systems, according to their objectives:

- **Platform Operators:** they are installed by default to perform cluster functions, and are operated by the Cluster Version Operator (CVO).
- **Optional add-on Operators:** different from the previous one, they are used by users to run in their applications and are managed by Operator Lifecycle Manager (OLM) [6].

3.3 OpenShift vs Kubernetes

As we have seen from the previous paragraphs, there are already several differences between Kubernetes and OpenShift in the architecture and in the basic concepts. In the next paragraphs, we will go in-depth on the most important differences between the two platforms.

Open Source Project vs Enterprise Product

Kubernetes is an open-source project with a large community, thus no full support is offered to users. On the other hand, OpenShift is a commercial product that supports organizations to manage private, public, and hybrid infrastructure. In fact, an OpenShift subscription enables users to get paid support [8].

Security Aspects

The security aspects are a strong point of OpenShift. In fact, in OpenShift administrators can use Security Context Constraints to govern permissions for pods in a similar way that RBAC resources control user access.

These permissions include actions that a pod can perform and what resources it can access, restricting you from running simple container images as well as many official images.

In this way, OpenShift avoids pod manifest authors altering the list of capabilities offered by docker for each pod by requesting additional capabilities or removing some of the default behaviors [5].

The authentication processes are different, as well. In fact, for Kubernetes, the setup and configuration of authentications require a lot of effort whereas OpenShift offers an integrated server for avoiding these efforts and for better authentication.

Web-UI

A Web User Interface (UI) is essential for successful cluster management. In fact, with OpenShift you can easily access a web console that allows showing, creating, and changing most resources with just a few clicks. The authentication and authorization process is speeded up with an out-of-the-box login page, as well. Unlike Kubernetes that does not provide any login page and requires an installation of a dashboard separately [8].

Deployment Approach

Among the various changes introduced by OpenShift, there is the deployment strategy. Whether with Kubernetes you can internally implement deployment objects via operators to manage and update pods; with OpenShift, the deployment is done with the DeploymentConfig command. In this way, no controllers can be used, but dedicated pod logic.

Moreover, although DeploymentConfig does not support multiple updates like Kubernetes objects, it can handle versioning and triggers in order to drive automated deployments.

Continuous Integration & Continuous Delivery

Although neither of the two provide full CI/CD support, and additional tools like a CI server are required to build a full CI/CD pipeline, OpenShift simplifies this process because it offers three CI/CD solutions:

- OpenShift Builds: It creates cloud-native apps by using a declarative build process based on the YAML file in order to create a BuildConfig object. When deployed, the BuildConfig object generally builds an executable image and drives it towards a container image registry.

- OpenShift Pipelines: It provides a Kubernetes native CI/CD framework for designing and executing each stage of the pipeline in its own container. It uses Tekton building blocks to automate deployments across multiple platforms by abstracting away the underlying implementation details [5].

- OpenShift GitOps: it is an Operator that exploits Argo CD as the declarative GitOps engine.

Moreover, OpenShift offers a Jenkins image that integrates directly with it and you can use it for the CI server.

Integrated Image Registry

Contrary to Kubernetes which offers an easy way to set up your own Docker registry, OpenShift provides an integrated image registry with a console where you can search for information about images and image streams, and that you can use with Red Hat or Docker Hub.

Updates

Both Kubernetes and OpenShift enable you to easily upgrade existing clusters instead of rebuilding them from scratch, but if K8s usually uses the kubeadm upgrade command to update to a newer version, OpenShift relies on the Red Hat Enterprise Linux package management system to update the cluster to the newest version [8].

With OpenShift Container Platform 4, Red Hat provides the possibility to update the cluster with a single operation by using the web console or the OpenShift CLI [5].

Chapter 4

Service Mesh

Service mesh is the core of this work since we want to compare two of the most popular Service Mesh. We must first understand the service mesh base concepts and features. In the following paragraphs, we will present the main issues of a micro-services approach and how a service mesh can mitigate them. Moreover, particular attention will be paid to security and internal communication among micro-services.

4.1 ZTA - Zero Trust Architecture

The year 2021 was not a good one for data security. Remote work due to COVID-19 increased the breach and the average cost, as well. In fact, we have witnessed the highest average total cost in the 17-year history [9].

The micro-services approach expands exponentially the surface area available for the attack, exposing data to greater risk. Moreover, if with a monolith application network-related problems like access control, load balancing, and monitoring are solved once, now they must be handled separately for each service within a cluster. Nevertheless, a Zero-Trust approach with a service mesh technology can help micro-service applications to configure into a transparent infrastructure layer the cross-cutting functionality like policy control, authorization, authentication, encryption and, even better, it can help organizations to increase their cyber resiliency, observability, as well as ways to implement and manage the risks.

4.1.1 Zero Trust Networking Principles

For years, network security has been based on implementing a moat-and-castle approach by contrasting attackers just at the perimeter, using strong firewall policies. This principle is used in the Kubernetes clusters, too. In fact, it denies

both ingress and egress traffic in the cluster by means of user-defined rules and provides a flat network where each service can exchange messages without any restrictions because Kubernetes considers trusted the internal network by default, and whoever is already in it [10]. Sadly, it has never been a reliably effective strategy.

In 2010, Forrester Research coined the term "Zero Trust" and the primitive perimeter-based security model was replaced with a new principle: "never trust, always verify." This means that no person or machine is trustworthy by default in or out of the network.

4.1.2 Applying Zero Trust Principles to Kubernetes

The service mesh addresses these principles by issuing secure identities to services, handling the creation and renewing certificates, and mounting the appropriate certificates to the sidecars. This offers a secure and verifiable identification of the services across the mesh using the mTLS. Additionally, requests for access to services may be authorized or blocked by intentions, namely by DENY/ALLOW policies that enable operators to define service-to-service communication authorizations by service name.

Consequently, using the Service Mesh many attacks against containerized applications can be mitigated: for instance service impersonation, unauthorized access, packet sniffing, and data ex-filtration attacks.

4.2 Issues of Micro-services Approach

Focusing on the micro-service approach, we said it brings many advantages with respect to the monolithic one. Since each service is an independent deployable entity, it can be more easily scaled and re-deployed; in addition, many versions and deployment strategies can be adopted. Nevertheless, when hundreds or more services communicate together it becomes challenging to grant security, reliability, and the observability of the entire application.

Starting from these assumptions, the first service meshes were born. In the next section we will see how it works and which features provides.

4.3 Features

As mentioned before, when a micro-service application starts to increase in terms of services and instances it is difficult to assure scalable, safe, and reliable network traffic among micro-services. In this regard, the service meshes come to the rescue. The main aspects of a service mesh are:

- **Reliability:** it increases the efficiency of internal communication among services offering strategies of automating retries and backoff for the failed request.

- **Observability:** Each service exposes its metrics and traces allowing one to observe the traffic flows of the application by means of console and tracing tools, such as Kiali, Jaeger, and many more.

- **Security:** The service mesh allow to address the security aspects like communication encryptions, authentication and authorization, and the possibility of setting up traffic policies that permit or refuse communication between certain services.

But these are not the only addressed characteristics: indeed, with a service mesh you can obtain:

- **Load Balancing:** All proxies among the mesh allow for the implementation of load-balancing policies between services. These policies are strategies used by the mesh to decide which replication will receive the original request, just as if you have small load balancers in front of each service.

- **Service Discovery:** A powerful service discovery mechanism is offered, thus a service can discover the other ones avoiding the problems of the legacy DNS services.

- **Simplify Deployment:** More sophisticated deployment strategies like rolling updates, canary release, blue-green deployments etc. are available.

4.4 How It Works

Now that we have understood the main benefits of a service mesh and what problems it addresses, we have to understand how it works.

A service mesh is a dedicated infrastructure layer that you can add to your applications. This means that a service mesh does not introduce any new serviceability to an app's run-time environment. Applications already define rules to specify how requests get from point A to point B. What is different about a service mesh is that the logic governing service-to-service communication is abstracted into this infrastructure layer [11].

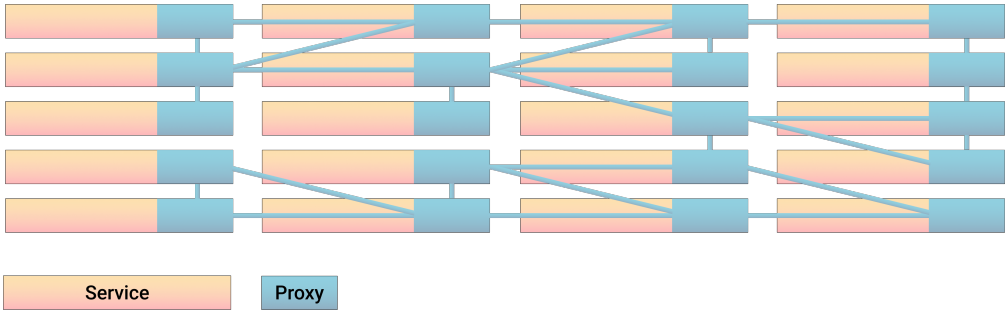

Figure 4.1: Service Mesh.

To do this, a service mesh implements the sidecar pattern, building a network of proxies; it injects a sidecar alongside each service so that requests are routed between micro-services through these proxies in their own infrastructure layer.

Nevertheless, this does not only divide the logic governing service-to-service communication from the main container but also simplifies diagnosing communication failures because the logic that controls inter-service communication is no longer hidden within each service.

The component responsible for configuring the sidecars into the service mesh is the control plane. It takes your desired configuration and dynamically programs and updates the proxy servers with the new rules changes.

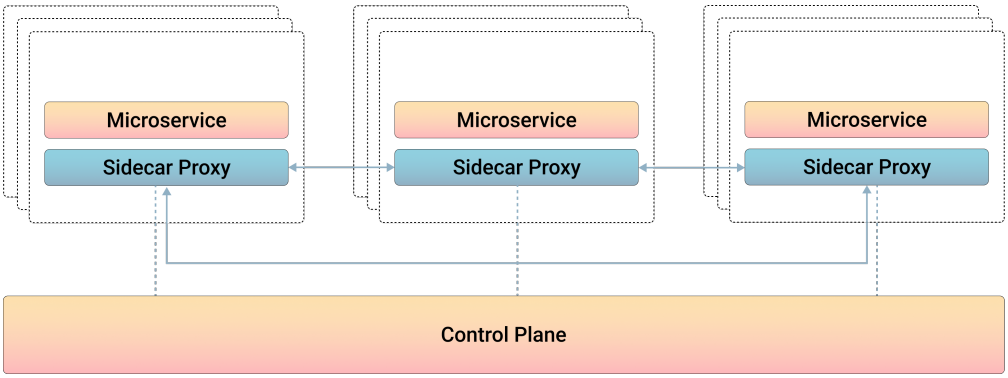

Figure 4.2: Service Mesh Architecture.

Chapter 5

Kuma

This chapter analyzes Kuma architecture, starting from the idea behind it. Then it describes the main concepts of this open-source project; how it works, how it handles security and observability aspects, and what resources it makes available to us.

5.1 The Main Idea Behind

Kuma is a universal, multi-tenant, and open-source control plane for service mesh and micro-services management. It is built on top of Envoy proxy with support for Kubernetes, VM, and bare metal environments [12].

The main idea behind Kuma was born from the need to integrate a service mesh into one of the most popular open-source API gateways called Kong because the previous existing mesh solutions were *"hard to use, hard to deploy, and hard to configure"* according to the CTO and co-founder of Kong, Marco Palladino.

Starting from this need, the production of Kuma began, paying particular attention to the ease of use and the fact that it had to support not only Kubernetes but also other environments, such as VMs [13].

5.2 Architecture

Kuma architecture is based on top of four main principles:

- **Agnostic:** Kuma is platform-agnostic and can run and operate not just on Kubernetes but also in other environments.
- **Centralized Control Plane:** Kuma control plane is a single executable written in GoLang. It allows for operating and controlling multiple independent meshes from one place.

- **Envoy-based:** Kuma data plane is built on Envoy, which features a powerful administration API for monitoring and troubleshooting the running data plan.
- **Attribute-based policies:** Kuma allows teams to apply fine-grained and intuitive L4 + L7 policies for security, traffic control, observability, routing, and other features, with any arbitrary tag selector [12].

5.2.1 Control Plane

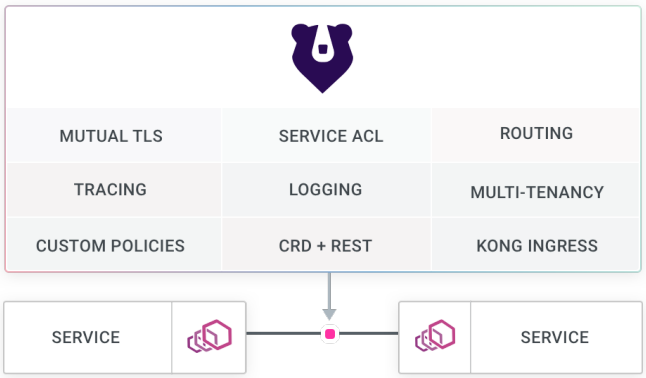

Figure 5.1: Kuma architecture [12].

Kuma control plane (kuma–cp) is used as a source of truth to dynamically configure the underlying data plane proxies.

It is both universal and simple to deploy. Kuma requires a PostgreSQL database in order to store its configuration, since it does not leverage the underlying K8s API server to do so.

Moreover, a single control plane can support multiple individual meshes and implements an out-of-the-box multi-zone connectivity thanks to the native service discovery and ingress capability in order to support multiple clouds, regions, and Kubernetes clusters.

5.2.2 Data Plane Proxies

As already described, Kuma follows a sidecar proxy model for the data plane proxies, which means we have an instance of a kuma–dp for every instance of our services. Kuma identifies four different types of the data plane so that, when an instance starts, the kuma–dp attempts to connect to the control plane in order to communicate:

- What type of data plane they are (standard, zone-ingress, zone-egress or gateway).

- what is its address/port combination.
- What services they expose. This is also called *inbound* and it is the networking configuration for the service in order to configure what port the data plane proxy will listen to and the Tags that belong to the service.
- The way the application will use the side-car to access other services (either via a transparent proxy, or by explicitly listing the services it will connect to) [12].

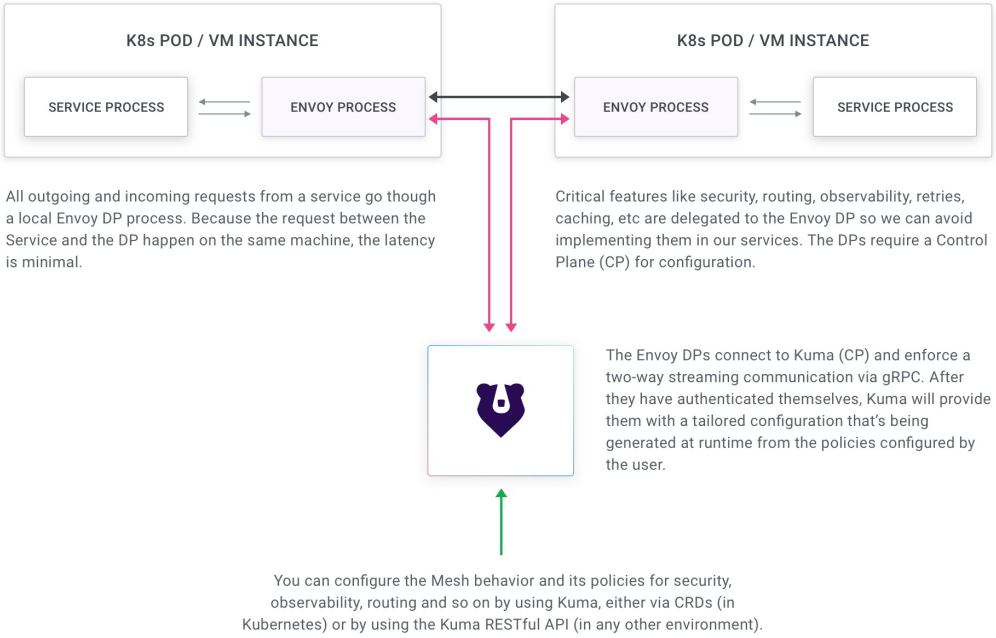

Figure 5.2: How Kuma works [12].

Tags

In Kuma, each data plane proxy is associated with one or more tags, which configure the service mesh with matching policies and identify the service.

Generally, Kuma tags are prefixed with kuma.io like kuma.io/service, used to identify the service name, or kuma.io/protocol that points the protocol exposed by the service.

5.3 Networking

In order for Kuma to enhance the underlying connectivity between services by making the underlying network more reliable, it is necessary that all incoming and

outgoing traffic, on the host performing the service, is routed through the Kuma data plane proxy. In Kuma, a transparent proxy allows you to get this [12].

Transparent Proxying

Transparent proxying makes it easier to deploy a service mesh by preserving the denomination of existing services and preventing changes to the application code.

As previously mentioned, Kuma utilizes transparent proxying using iptables. It can be installed with the kuma-init container or CNI; in this way, all traffic is automatically intercepted by kuma−dp. In addition, both these options require certain additional privileges in order to install the transparent proxy.

Moreover, the usage of Kuma CNI in an OpenShift environment provides tuned default settings with Multus [12].

5.3.1 Service Discovery

Since services typically call one another and the modern microservice-based application more and more runs in virtualized or containerized environments where the number of instances of a service and their locations changes dynamically, the automatic detection of devices and services has to be addressed.

In order to handle this issue, in Kuma, when a service starts, a gRPC streaming connection is initialized between the data plane proxy and the control plane. As we have mentioned in the previous paragraphs, this allows the data plain to retrieve the latest policy configuration and sends diagnostic information to the control plane.

While doing so, the data planes also advertise the IP address of each service, by looking at the address of the Pod or by looking at the inbound listeners that have been configured in the inbound property of the data-plane specification, depending on where Kuma runs (K8s, Universal).

In this way, the IP address advertised by every data plane to the control plane can be used to route service traffic from one kuma-dp to another kuma-dp. This means that Kuma knows at all times what are all the IP addresses associated to every replica of every service [12].

5.3.2 Gateway

In order for external services to be able to access internal services, we need a component. Generally, this component is called *gateway* or *ingress* and it may be considered to have one foot outside the mesh to receive traffic and one foot inside the mesh to deliver this external traffic to the services inside the mesh.

The gateway should be deployed as any other service within the mesh, but

however, it is also responsible for the security of the entrance since we want inbound traffic to go directly to the gateway, otherwise, customers should receive dynamically generated certificates for inter-service communication within the mesh.

Moreover, there are no differences between a Kuma data plane and a Kuma gateway; both are deployed as an instance of the kuma−dp process and manage an Envoy proxy process.

In Kuma, there are two possibilities that can deploy a gateway:

- **Delegated:** This makes it possible for users to use any existing gateway.
- **Builtin:** It sets up the data plane proxy to expose external listeners to drive traffic inside the mesh [12].

Since that the second one is still in the experimental phase and should be enabled with a special flag during installation, in the next chapters we will use the delegated gateway, namely the Kong API Gateway.

Kong API Gateway

Kong or Kong API Gateway is a cloud-native API Gateway, designed to be platform-agnostic and scalable, moreover, it can be extended using plugins.

Kong is one of the most popular API gateways and it serves as the central layer for orchestrating micro-services or conventional API traffic with ease since it offers functionality for proxying, routing, load balancing, health checking, authentication, and more [14].

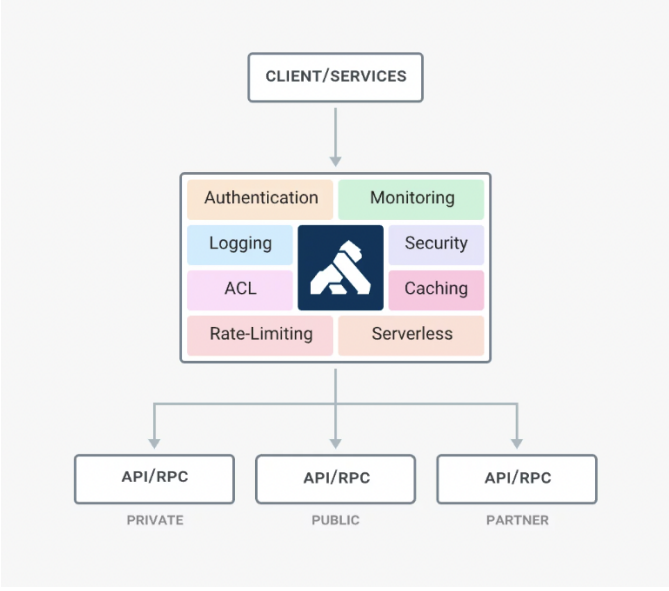

Figure 5.3: Kong API Gateway [12].

5.4 Policies

The policies in Kuma allow you to configure its behavior and build a modern and reliable Service Mesh. Once installed, you can apply policies using its client kumactl.

All policies that will be applied to the data plane proxy, generally follow the same basic structure; in fact, they are made up of:

- sources: List of selectors which specify the data plane objects which are the origin of the network traffic.
- destinations: List of selectors that specify the data plane object where source traffic is sent.
- conf: The configuration that applies to network traffic between source and destination.

Moreover, in order to match all traffic, the wildcard character (*) is supported. For instance, the customer-side policies of connecting between two data plan objects do not support arbitrary tags in the destination selector [12].

Traffic Permissions

An important policy that you can use in Kuma is traffic permission. It provides access control rules to define authorized traffic in the mesh.

It is generally used in order to specify what source services can consume which destination services or can also be used to limit traffic to services outside the mesh.

Nonetheless, the TrafficPermission resource requires Mutual TLS enabled on the Mesh since it is used in order to validate the service identity with data plane proxy certificates. Otherwise, all service traffic is allowed by Kuma [12].

At the installation time of Kuma, a default TrafficPermission policy is created into the default Mesh. This policy allows all communication between all services in the Mesh.

Traffic Route

In order to configure routing rules for the traffic in the mesh, Kuma provides the TrafficRoute resource. The features offered by this resource are:

- Weighted routing.
- Versioning across services.
- Deployment strategies (such as blue/green or canary).

Even in this case, at the installation time of Kuma, a TrafficRoute policy that enables the traffic between all the services in the mesh will be created.

5.5 Security

Kuma handles many different security aspects like the security between services belonging to the Mesh, the secure communication between the control plane and data plane, and so on. To do so, Kuma holds autogenerated certificates and other useful files in a working directory.

Secrets

In Kuma, the Secret resource plays a fundamental role. In fact, by means of secrets, Kuma implements an integrated interface to store sensitive information which can be used later on by any policy at runtime.

These Secret resources are held in the same namespace as the Control Plane. Moreover, each Secret resource has to specify a type field with the value system.kuma.io/secret and a label kuma.io/mesh with the name of the Mesh on which the resource belongs to, because a Secret resource cannot be shared across different Meshes whether it must be used for Mesh Policies like Provided CA or TLS setting in External Service. In fact, just global-scoped Secrets are not bound to a given Mesh since they are used for internal purposes [12].

Data plane proxy to control plane communication

As we have seen above, the first step for a data plane is to attempt a connection with the control plane in order to retrieve its own setting, but since certain sensitive information is exchanged during the connection, the communication needs to be encrypted by TLS.

By default, the control plane's server is secured by TLS using autogenerated certificates. When the data plane consumes this server, it has to verify the identity of the control plane; in fact, it needs to obtain the CA that was used to generate the certificate, which is different from the CA used in service-to-service communication.

In Kuma, this operation is performed automatically by the data plane proxy Injector, which provides the CA to the Kuma DP in order to confirm the control plane identity [12].

5.5.1 Mutual TLS

In Kuma, secure communication between services in a Mesh, as well as assigning an identity to every data plane proxy, is achieved through the mTLS policy.

Furthermore, Kuma supports the following CA backends:

- **builtin:** Kuma auto-generates a CA root certificate and key, storing it as a Secret, by default.

- **provided:** Kuma supports a third-party CA root certificate and keys provided by the user in the form of a Secret, as well.

Choosing the last option, you must also manage the certificate lifecycle yourself, and you must first upload the Secret resources of the certificate, and key and then reference the Secrets in the mTLS configuration.

Nevertheless, the certificate and key for a provided backend must satisfy the following requirements:

- It MUST have basic constraint CA set to true.
- It MUST have key usage extension keyCertSign set.
- It MUST NOT have key usage extension keyAgreement set.
- It SHOULD NOT set key usage extension digitalSignature and keyEncipherment to be SPIFFE compliant.

Therefore, the generated certificates, after a CA backend has been specified, are SPIFFE compatible and are used to identify all workloads in the Mesh in order to authenticate and authorize their messages.

By default, mTLS is not enabled. In order to activate it, you need to configure the mtls property in a Mesh resource.

Once this policy is enabled, all traffic is denied unless a TrafficPermission policy is being configured. In order to avoid unexpected traffic interruptions, Kuma provides at installation time a TrafficPermission policy that allows all traffic [12].

Starting from the 1.4.1 version Kuma introduced a new mTLS mode called PERMISSIVE. It avoids the previous unexpected traffic interruptions by accepting both TLS and plain-text inbound connections on the server-side.

5.6 Monitoring

Although an orchestration solution such as K8s can greatly simplify application deployment in containers and across clouds, it brings its own set of complexities. In fact, monitoring a very large, complex system has two major challenges: the high volume of components being analyzed, and the need to keep the maintenance cost reasonably low. Kuma control plane allows you to have a central place to collect data and metrics related to the meshes that belong.

5.6.1 Traffic Metrics

Kuma natively integrates with Prometheus for auto-service discovery and collection of traffic measurements.

Moreover, since metrics are included in the mesh configuration, Prometheus can automatically find each proxy in the mesh.

In addition, Kuma can also work with systems such as Splunk, Logstash, and many others.

At the installation time, proxies in the Mesh do not expose automatically metrics. To do so, you have to configure the Prometheus backend in the Mesh resource.

By default, after the basic configuration, the Kuma data plane starts to expose an HTTP endpoint with Prometheus metrics on port 5670 and URI path /metrics.

In order to avoid conflict on a particular Pod that uses the metrics port or path for other purposes, Kuma provides two different annotations to override the Mesh configuration: prometheus.metrics.kuma.io/port and prometheus.metrics.kuma.io/path.

Grafana

When metrics are installed through the Kuma client (kumactl install metrics), default Grafana dashboards are also provided. Otherwise, they can be imported from the Grafana Labs repository of *Konghq*.

These dashboards let you investigate:

- The status of a single data plane in the mesh (Kuma Dataplane Dashboard).
- The aggregated statistics of a single mesh (Kuma Mesh Dashboard).
- The aggregated statistics from data planes of specified source service to data planes of specified destination service (Kuma Service to Service Dashboard).
- The control plane statistics (Kuma CP Dashboard).
- The aggregated statistics for each service (Kuma Service Dashboard).

In addiction, a Service Map Dashboard is provided in order to show a topology view of your service traffic dependencies.

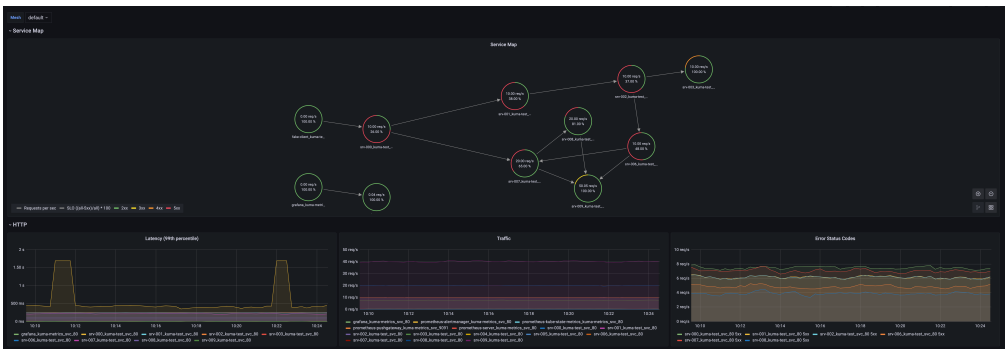

Figure 5.4: Service Map Dashboard.

Traffic Trace

In order to monitor and troubleshoot micro-service behavior, Kuma introduces the TrafficTrace resource that determines what services you want to trace. Nevertheless, Kuma requires an explicit specification of the protocol for each service and data plane proxy you want to enable tracing for, since the Mesh supports HTTP, HTTP2, and gRPC protocols [12].

A native Zipkin and Jaeger (as the Zipkin collector) integration are provided when installed with the kumactl install tracing command. Still, as for each other policy, Kuma pretends a specification of the backend as a Mesh resource property.

Likewise Kuma provides support for systems such as Datadog.

Chapter 6

Red Hat OpenShift Service Mesh

Red Hat OpenShift Service Mesh is an Istio-based project; Istio is an open platform-independent service mesh that can be applied in a transparent way to the applications.

OpenShift Service Mesh is available for Red Hat OpenShift Container Platform at no cost in order to provide a uniform way to connect, manage and observe microservices-based applications.
Some benefits provided by Red Hat OpenShift Service Mesh are:

- **Ready for production:** Red Hat OpenShift Service Mesh is a product pre-validated and deployed as an Operator for the Red Hat OpenShift platform.

- **Security-focused:** Using transparent mTLS encryption and fine-grained policies, Red Hat OpenShift Service Mesh provides an easy way to implement zero-trust networking.

- **Based on open source:** As mentioned before, Red Hat OpenShift Service Mesh is an open source Istio-based project which includes others projects like Jaeger and Kiali to provide additional functionality [15].

6.1 What is in

Focusing on the last point of the former paragraph, we can analyze the number of projects that compose Red Hat OpenShift Service Mesh.

Istio

Istio is the core of this project. It is a sidecar container implementation that provides traffic monitoring, access control, discovery, security, resiliency etc.

Istio is platform-independent, which means it has been built to enable support not just for Kubernetes-based deployments but also for other environments like VMs.

As we already saw for Kuma, Istio does not require any changes to the code of the application, since it implements the sidecar model, deploying a proxy next to each service [16].

Jaeger

The adoption of the micro-services approach moves the handling of a request from the response of one service to a response generated by many different ones, which participate in order to produce it.

Since the path of this request is a distributed transaction, a tracing tool can be involved to track the request among the different services in order to better understand the serialization, parallelism, sources of latency, and the whole chain of events in the distributed transaction.

Red Hat OpenShift Service Mesh integrates the open source Jaeger tool to implement this tracking mechanism. It gives an overview of the entire request process from beginning to end recording the execution of individual requests across the whole stack of micro-services and presenting them as traces [17].

Kiali

In order to obtain robust observability, Red Hat OpenShift Service Mesh provides by default Kiali. It is a management console for the Istio service mesh.

Kiali offers in-depth traffic topology, health grades and powerful dashboards [18], thus helping you define, validate and observe your Istio service mesh.

Moreover, since Kiali is designed for Istio, it provides the ability to validate the configurations of Istio.

Finally, if Grafana and Jaeger are available, Kiali provides distributed tracing and basic Grafana integration [17].

6.2 Differences between Istio

Red Hat OpenShift Service Mesh differs from Istio in terms of installations and provided features. Moreover, it helps to resolve issues and ease deployment on the OpenShift Container Platform.
The main differences are:

- **Multi-tenant control plane:** Unlike cluster-wide installation, the multi-tenant installation uses a different scope of privileges. The control plane components no longer use cluster-scoped RBAC resource ClusterRoleBinding, but rely on project-scoped RoleBinding [17]. In addition, each project in the members list will have a RoleBinding for every service account connected to a control plane deployment; this latter will exclusively watch these member projects. Every project is matched by a maistra.io/member−of label, in which the member-of value is the project that contains the control plane installation.

- **RBAC features:** Starting from the mechanisms added by Istio in order to control access to a service, a subject can be identified by user name or by specifying a set of properties and applying access controls accordingly. The upstream Istio community installation provides the possibility to perform exact header matches, match wildcards in headers, or check for a header containing a specific prefix or suffix. Red Hat OpenShift Service Mesh extends these mechanisms of matching by adding the opportunity to perform a match through a regular expression.

- **OpenSSL:** Since the underlying Red Hat Enterprise Linux operating system already provides the OpenSSL libraries (libssl and libcrypto), Red Hat OpenShift Service Mesh replaces BoringSSL with OpenSSL. These libraries are dynamically linked from the OS to the Red Hat OpenShift Service Mesh Proxy binary.

- **Kiali and Jaeger:** As mentioned in the paragraphs above, Red Hat OpenShift Service Mesh integrates and configures by default the Kiali and Jaeger tools in its project in order to provide tracing and visualization of the mesh.

6.3 The Red Hat Advantages

The advantages of using OpenShift Service Mesh are due to the fact that it has been engineered in order to be run in a production environment. Since it has been specifically projected by Red Hat developers to work on the OpenShift Container Platform, it requires an easier installation step and it easily integrates with the other Red Hat products.

Using OpenShift Service Mesh, developers can increase their productivity because they do not have to integrate specific libraries to manage the various cross-cutting concerns or the communication policies.

6.4 Istio Architecture

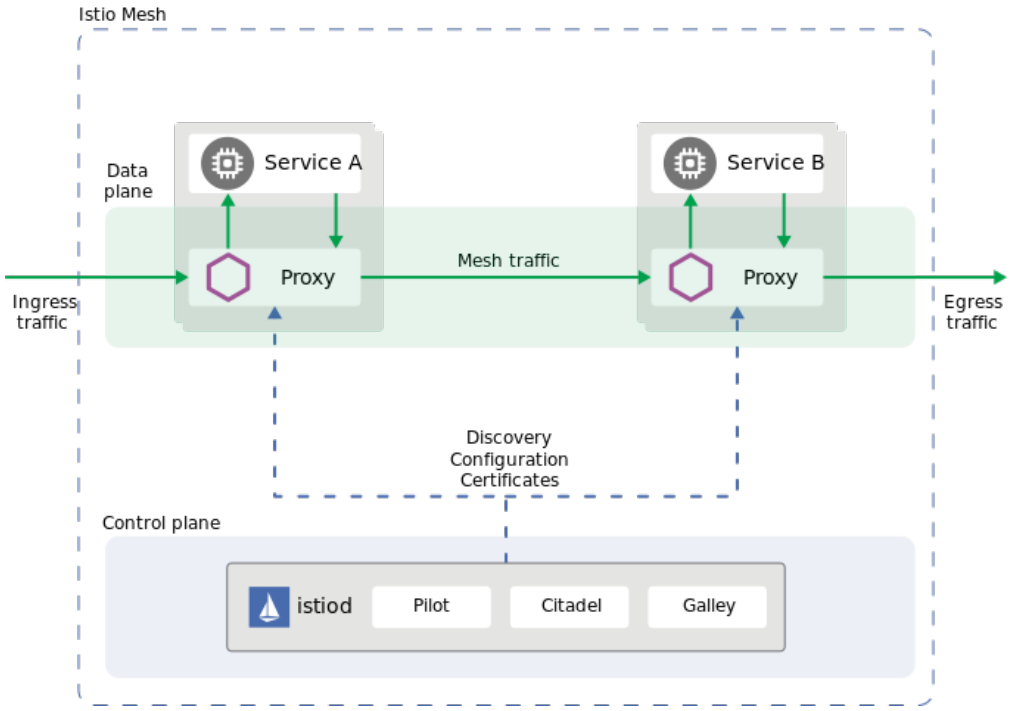

Figure 6.1: Istio Architecture [19].

Like others Service Mesh, Istio service mesh architecture is logically split into a data plane and a control plane. The data plane is made up of a network of Envoy proxies. Each one deployed as a sidecar alongside the main service. The control plane manages and configures the proxies to route traffic.

Envoy

For its proxies, Istio uses an extended version of the Envoy proxy. Envoy proxies are deployed as sidecars to services and they are the only Istio components that mediate all inbound and outbound data plane traffic.

This sidecar architecture allows you to retrieve a rich telemetry that can be sent to monitoring systems and enforce policy decisions.

Moreover, working with a sidecar approach also allows you to add Istio capabilities to an existing deployment without rearchitecting or rewriting code [19].

Istiod

Istiod is the control plane of Istio. Starting from user configurations, Istiod converts them into Envoy-specific configurations and propagates them to the sidecars at runtime.

Istiod is made up of three sub-components:

- **Pilot:** It provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing and resiliency.
- **Citadel:** It enables strong service-to-service and end-user authentication with built-in identity and credential management.
- **Galley:** It is Istio's configuration validation, ingestion, processing and distribution component.

Furthermore, in order to allow secure mTLS communication in the Mesh, Istiod generates certificates and acts as a CA [19].

6.5 Traffic Management

Using traffic routing rules Istio lets you easily control the traffic flow and API calls between services. Moreover, traffic routing rules affect performance and enable better deployment strategy.

In order to handle traffic within the mesh, Istio needs to know each service and to which endpoints it belongs. Working with K8s facilitates the step of discovering the systems; in fact, Istio automatically detects services and endpoints in the cluster and populates its own service registry.

The Envoy proxies can direct traffic to the desired services using this registry. The Envoy proxies default distribute traffic across each service's load balancing pool using a round-robin model [19].

6.5.1 Istio Gateway

Traditionally, Kubernetes uses Ingress controllers to handle traffic from the outside to the cluster. This is no longer the case while using Istio [20].

Three are the fundamental reasons why the Ingress resource is no longer used with Istio:

- The specification for HTTP loads is too simple (only traffic on ports 80 and 443 can enter).
- No rules for traffic splitting and traffic monitoring are provided.

- Implementing a custom solution based on annotations may make the code not portable.

Istio gateways use new Gateway resources and VirtualServices resources to control ingress traffic.

Gateway

Gateway is responsible to handle the first traffic inbound and/or the last that leaves the mesh. Incoming traffic to the mesh is filtered by port, the protocol used, Cert, and more. Gateway configurations are applied to stand-alone Envoy proxies that are running at the edge of the mesh.

Unlike Ingress APIs, Istio gateways let you the possibility to configure layer 4-6 load balancing properties such as ports to expose, TLS settings, and so on.

In order to work, a virtual service is bound to the gateway. This lets you basically manage gateway traffic like any other data plane traffic in an Istio mesh.

According to the type of the Istio installation, an istio−ingressgateway and istio−egressgateway deployments are provided. You can apply your own gateway configurations to these deployments or deploy and configure your own gateway proxies [19].

Virtual Service

In order to find and reach the service that will handle the incoming request, Istio provides the VirtualService resource, one of the fundamental building blocks of Istio's traffic routing functionality. Combined with the Gateway resource, it allows you to decouple where clients send their requests from the destination workloads that actually implement them.

Using this resource you can configure how requests are routed to a service. In fact, each virtual service consists of a set of routing rules that are evaluated in order, letting Istio match each given request to the virtual service to a specific real destination within the mesh.

You use routing rules within virtual service that tell Envoy how to send virtual service traffic to proper destinations. Route destinations can be versions of the same service or totally different services [19].

The basic flow of a request

Gateway and Virtual Service resources play a fundamental role; in fact, when a client sends a request on a specific port for a host, the basic steps that the request follows before reaching the destination are:

1. The Server Load Balancer listens to this port so that it can forward the request to the cluster, using the same port or another one.

2. Then, the SLB routes the request to the port, listened to by Istio IngressGateway service.

3. This service forwards the request to the corresponding IngressGateway pod.

4. Gateway and VirtualService resources are configured on the IngressGateway pod. If the port, protocol, and related security certificates are set up on the Gateway; the VirtualService routing information is instead used to find the correct service.

5. The request is routed by Istio IngressGateway pod to the corresponding application service, based on the routing information.

6. Finally, the application service forwards the request to the corresponding application pod.

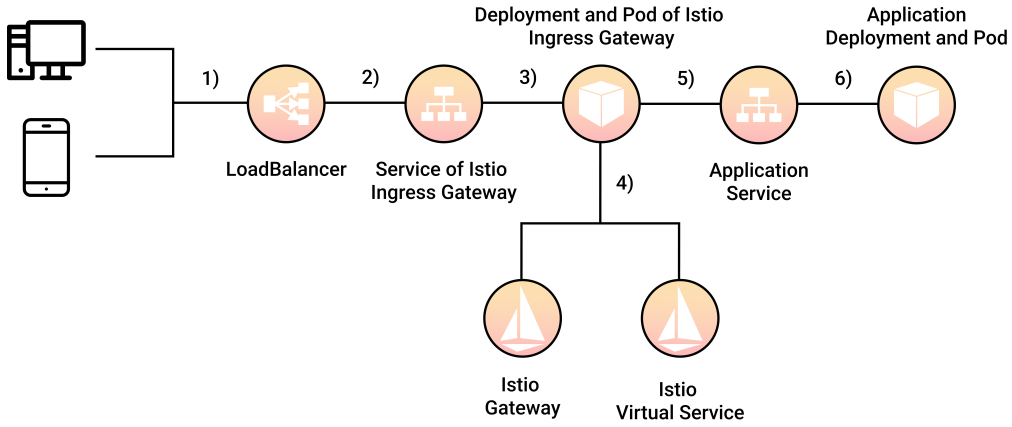

Figure 6.2: Basic flow of a request.

Destination Rules

Another key part of Istio's traffic routing functionality is the destination rules: they allow to configure what happens to traffic for a destination.

These rules are applied to the real traffic destination because they are evaluated after the virtual service routing rules.

Specifically, you use destination rules to specify named service subsets, such as grouping all instances of a given service by version.

Destination rules also let you customize Envoy's traffic policies [19] in terms of:

- Load balancing model.
- TLS security mode.
- Circuit breaker settings.

And many other destination rule options.

6.6 Security

Despite all the benefits of micro-service architecture, including better agility, better scalability and better ability to reuse services, micro-services have particular security needs.

Istio Security provides a security solution in order to address these issues and achieves the following goals:

- **Security by default:** Application code and infrastructure do not need changes.
- **Defense in depth:** Multiple layers of defense are provided because of integration with existing security systems.
- **Zero-trust network:** build security solutions on distrusted networks.

High-level Architecture

In order to provide a strong identity, powerful policy, authentication, authorization and audit (AAA) tools, and others, Istio security involves multiple components:

- A Certificate Authority.
- The configuration API server in order to distribute policies of authorization, authentication, and secure naming information.
- Policy Enforcement Points (played by sidecar and perimeter proxies) that grant secured communication between clients and servers.
- A set of Envoy proxy extensions to manage telemetry and auditing [19].

All these components are configured and handled by the control plane.

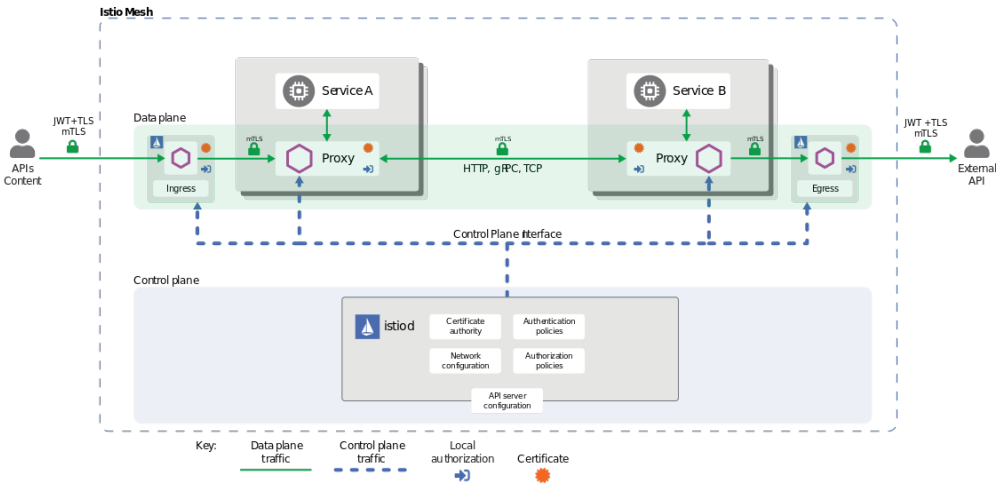

Figure 6.3: Istio Security Architecture [19].

6.6.1 Istio Identity

When two workloads start a communication, it is important to exchange credentials with their identity information for mutual authentication purposes.

Working with Istio, this process is different between client and server. On the client side, the server's identity is checked against the secure naming information to see if it is an authorized runner of the workload. On the other hand, the server can determine what information the client can access based on the authorization policies, audit who accessed what at what time, charge clients based on the workloads they used, and reject any clients who failed to pay their bill from accessing the workloads [19].

In order to provide a secure and strong identity, Istio supplies an X.509 certificate to every workload.

At the service initialization, Istio provides an istio−agent along with the Envoy proxy. Its purpose is to automate key and certificate rotation at scale working together with istiod.

When started, the Istio agent cooperates with istiod and consumes the gRPC service offered by it.
The steps performed by the Istio agent are the following ones:

1. Istio agent creates the private key and certificate signing request, and then sends the CSR with its credentials to istiod for signing.

2. The CA in istiod validates the credentials carried in the CSR. Upon successful validation, it signs the CSR to generate the certificate.

3. When a workload is started, Envoy requests the certificate and key from the Istio agent in the same container via the Envoy secret discovery service API.

4. After sending the certificates received from istiod and the private key to Envoy via the Envoy SDS API, the Istio agent monitors the expiration of the workload certificate and it repeats the process periodically for certificate and key rotation.

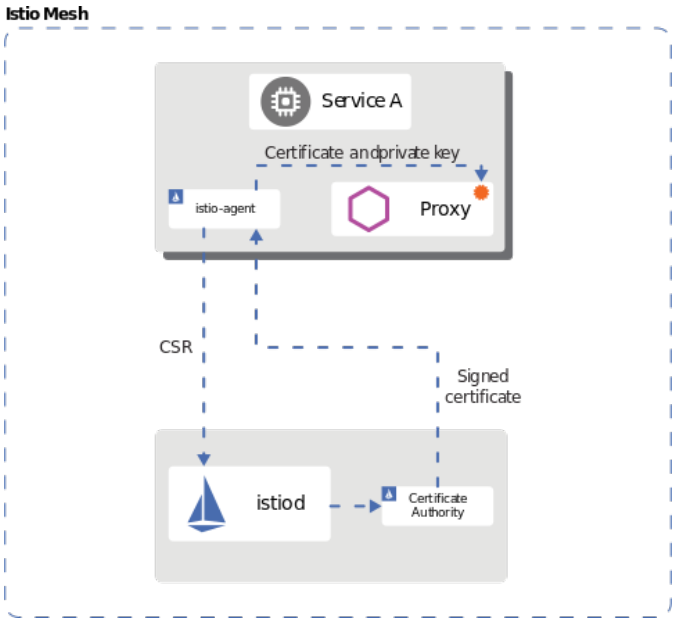

Figure 6.4: Identity Provisioning Workflow [19].

6.6.2 Authentication

Working with Istio, you can identify two types of authentication. The first one is used to authenticate service-to-service communications and it is named peer authentication. In order to support this type of communication, Istio provides mutual TLS as a full stack solution, thus using the mTLS solution ensures: a strong identity, secures service-to-service communication, and a key management system to automate key and certificate generation.

The second type of authentication is request authentication. It is used for end-user authentication to verify the credential attached to the request. Furthermore, Istio provides support request-level authentication with JWT validation either using a custom authentication provider or any OpenID Connect providers.

In all cases, Istio stores the authentication policies in the Istio config store via a custom Kubernetes API, and then Istiod will be responsible for keeping them

up-to-date for each proxy [19].

Moreover, Istio allows a fine-grain policy definition. When policies have an empty selector they are applied to all workloads in the mesh (mesh-scope policies) and are stored in the root namespace. On the other hand, policies can be applied also to particular namespaces (namespace-scope policies). This is achieved by configuring a selector field into the policy, thus authentication policies only apply to workloads matching the conditions you configured. In the last case, the policy is stored in the corresponding namespace.

Mutual TLS Authentication

When two services must communicate together, Istio tunnels this communication by means of the client and server-side PEPs.

If mTLS is enabled and a workload sends a request to another workload, before starting the communication the client side proxy must start a mutual TLS handshake with the server side Envoy. During the handshake, the client-side Envoy also does a secure naming check to verify that the service account presented in the server certificate is authorized to run the target service.

Only after server-side Envoy authorizes the request, the traffic is forwarded to the backend service through local TCP connections.

In order to enable mTLS without breaking existing communications, Istio introduces a mutual TLS with a permissive mode, which allows a service to accept both plain-text traffic and mutual TLS traffic at the same time [19].

6.6.3 Authorization

Istio introduces APIs for authorization features. it includes a single AuthorizationPolicy CRD that provides mesh, namespace, and workload-wide access control. The operators can set custom conditions on Istio attributes, and use CUSTOM, DENY and ALLOW actions that are enforced naturally on Envoy.

Moreover, Istio provides natively support for the protocols: gRPC, HTTP, HTTPS, and HTTP/2, as well as any plain TCP protocols.

Access control is enforced to the inbound traffic in the server-side Envoy proxy by means of authorization policy. To do so, each Envoy proxy runs an authorization engine which is responsible to authorize requests at runtime. When a request comes to the proxy, the authorization engine evaluates the request context against the current authorization policies, and returns the authorization result, either ALLOW or DENY. Nevertheless, for workloads that have no authorization policies, Istio allows all requests.

The precedence used by Istio for matching policies is in the order: CUSTOM, DENY, and then ALLOW.

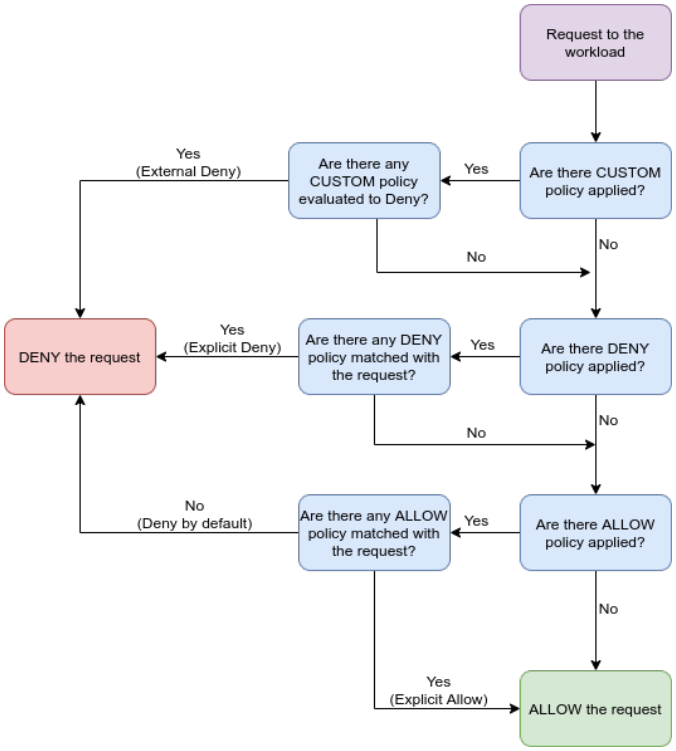

Figure 6.5: Authorization Policy Precedence [19].

6.7 Observability

Istio provides observability of services within a mesh, empowering operators to troubleshoot, maintain, and optimize their applications.

Istio generates three types of telemetry:

- **Metrics**
- **Distributed Traces**
- **Access Logs**

Metrics

In order to monitor mesh behavior, Istio generates a set of metrics for all service traffic in, out, and within an Istio service mesh.

Since it is also important to monitor the behavior of the mesh itself, Istio components export metrics on their own internal behaviors, providing detailed metrics for the mesh control plane.

Therefore, these metrics are separated into:

- **Proxy-level metrics:** Every proxy generates a large number of metrics about all traffic passing through it and it also provides detailed statistics about the administrative functions, configuration, and health information.
- **Service-level metrics:** These metrics cover the four *golden signals*: latency, traffic, errors, and saturation of a service. Operators may choose to turn off the generation and collection of these metrics, because they are entirely optional, by allowing operators to meet their individual needs.
- **Control plane metrics:** The Istio control plane also provides a collection of self-monitoring metrics. These metrics allow monitoring of the behavior of Istio itself [19].

The standard Istio metrics are exported to Prometheus by default and a default set of Grafana dashboards is provided by Istio in order to visualize the metrics collected.

Moreover, as mentioned in the paragraphs before, a Kiali tool can be installed in order to offer in-depth traffic topology, health grades, and powerful dashboards, thus helping you define, validate and observe your Istio service mesh.

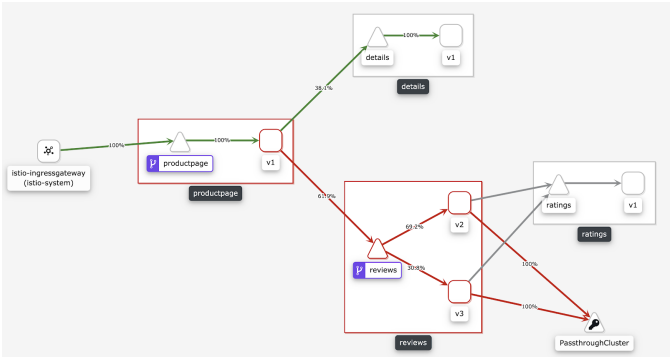

Figure 6.6: Kiali Topology View.

Distributed Traces

Istio supports distributed tracing in order to monitor individual requests as they flow through the mesh.

The proxies automatically generate trace spans on behalf of the applications they proxy, requiring only that the applications forward the appropriate request context [19].

Istio supports many tracing backends, like Zipkin, Jaeger, Lightstep, and Datadog.

When Kiali is installed and a tracing backend is present in the mesh (like Jaeger), Kiali can provide distributed tracing in its dashboards.

Chapter 7

Istio vs Kuma on OpenShift Container Platform

After presenting the technologies involved, the next step is to test them by implementing the various scenarios and analyzing their behavior in order to discover their own advantages and disadvantages. Therefore, in this chapter, we will present some use cases in order to create a zero-trust network. In particular, we will test Istio and Kuma on three different scenarios in order to:

- Deliver communication encryptions.
- Deliver authentication and authorization for service-to-service communication.
- Deliver a naïve deployment strategy.

7.1 Demo Application - Bookinfo

The environment used to test the technologies is composed of one OpenShift cluster hosted on the IBM Cloud.

On this cluster a micro-services application called Bookinfo [21] is deployed and developed by Istio for test purposes.

The Bookinfo application is made up of four micro-services:

- **productpage:** It uses the details and reviews micro-services to populate the page shown to the client.
- **details:** It contains book information.
- **ratings:** It contains book ranking information.

- **reviews:** It contains book reviews and, according to its version (there are three different versions of it), it calls the ratings micro-service in order to retrieve the book ranking information and display it as 1 to 5 colored stars.

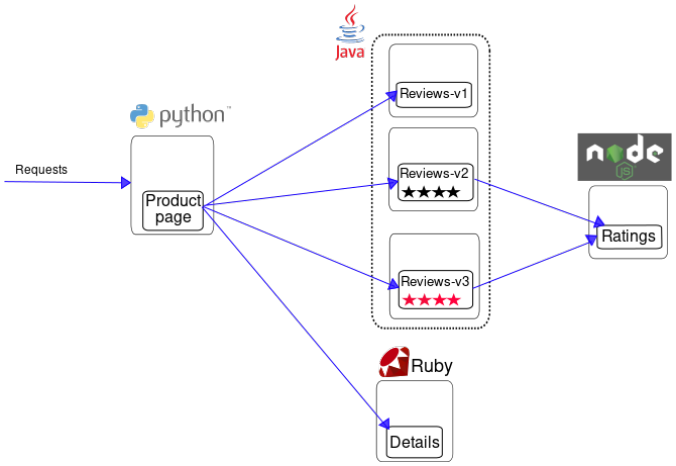

Figure 7.1: Bookinfo Application [21].

7.2 Apply Communication Encryptions

Since micro-services communicate with each other through APIs, securing communications between individual services is becoming more important than ever. Moreover, securing communications in a cloud-native application can prevent eavesdropping and man-in-the-middle attacks.

Below, we will see how both Istio and Kuma implement communication encryption.

Red Hat OpenShift Service Mesh

Using the OpenShift Service Mesh, first of all, we have to enable communication between the pod and the control plane. To do so, the OpenShift Service Mesh operator requires the creation of the ServiceMeshMemberRoll resource (with the name: default) into the istio−system project. All projects belonging to the mesh will then be listed in the field spec.members of this resource.

Next, we can enable the mTLS between services. By default, Istio tracks workloads with an Istio proxy and configures client proxies to automatically send mTLS traffic to those workloads or normal text traffic to workloads without a sidecar. This type of mode is called PERMISSIVE.

To set up communication only in mTLS we can apply a Peer Authentication

policy with STRICT mTLS mode.

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

In this way, when two micro-services that belong to the mesh need to communicate, they establish the mTLS connection between the sidecars through which encrypted traffic will flow. The sidecars exchange certificates and authenticate each other with the certificate authority.

Furthermore, Istio allows the possibility to define three different degrees of policy:

- **Workload-specific policy:** By means of the selector field, you can specify the workload to which the policy applies. An additional portLevelMtls field can be specified in order to set to which port of the workload applies the policy.
- **Namespace-wide policy:** No selector field is defined and the policy is specified for a non-root namespace.
- **Mesh-wide policy:** This policy is applied to all workloads that belong to the mesh. It is specified for the root namespace (istio –system) and without selector field.

Kuma

Unlike Istio, Kuma does not enable by default the mTLS, thus it must be done explicitly. Moreover, in Kuma, when the mTLS is enabled, all traffic is denied unless a TrafficPermission policy is configured to explicitly allow traffic between proxies.

Similarly to Istio, Kuma allows a PERMISSIVE mode which accepts both mTLS and plain-text traffic for all workloads.

In addition, Kuma gives the possibility to specify easily two useful fields for certificate generation:

- dpCert: It determines how often Kuma should automatically rotate the certificates assigned to every data plane proxy.

- caCert: It is used in order to determine some properties exploited by Kuma to auto-generate the CA root certificate.

In order to enable the mTLS, it is necessary to modify the Mesh resource by setting the mTLS fields, which indicate to Kuma what CA should be instantiated.

```
apiVersion: kuma.io/v1alpha1
kind: Mesh
metadata:
  name: default
spec:
  networking:
    outbound:
      passthrough: false
  mtls:
    enabledBackend: ca-1
    backends:
      - name: ca-1
        type: builtin
        dpCert:
          rotation:
            expiration: 1d
        conf:
          caCert:
            RSAbits: 2048
            expiration: 10y
```

7.3 Apply Authentication and Authorization for Service-to-Service Communication

Working on a K8s cluster allows you to exploit the network that it provides. Since Kubernetes considers the container network or applications running on it as trusted, every service can talk to another one without any restrictions.

Although users can create policies to deny both ingress and egress traffic in the cluster, you still need service-to-service authentication and authorization to ensure that only required resources are made available to services.

You can solve this problem by introducing a Service Mesh, which let you assign service identities to each service running on the Kubernetes cluster [22].

Red Hat OpenShift Service Mesh

As we saw in the previous paragraph, Istio provides a strong identity by means of X.509 certificates. Istio agents, running alongside each Envoy proxy, work together

with Istiod to automate key and certificate rotation at scale [19]. Therefore, Istio offers mutual TLS as a full-stack solution for transport authentication, it can be enabled without any service code changes.

After enabling and configuring mTLS (as in the paragraph before), we can define some Authorization Policies. Istio provides three different levels of access control: mesh-wide, namespace-wide, and workload-wide in order to increase security in the internal communication of the mesh.

The Authorization Policies are based on custom conditions to enforce access control to the inbound traffic in the server-side Envoy proxy, in order to deny or allow it.

Therefore we can define the Authorization Policies for the demo application; firstly denying all traffic directed to the workloads, and then, gradually defining for each host what type of request it can accept and from what source:

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-nothing
  namespace: bookinfo-istio
spec:
  {}
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "productpage-login"
  namespace: bookinfo-istio
spec:
  selector:
    matchLabels:
      app: productpage
  action: ALLOW
  rules:
  - to:
    - operation:
      methods: ["GET"]
    - operation:
      methods: ["POST"]
      paths: ["/login"]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "details-viewer"
```

```
  namespace: bookinfo-istio
spec:
  selector:
    matchLabels:
      app: details
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
        ↪  ["cluster.local/ns/bookinfo-istio/sa/bookinfo-productpage"]
    to:
    - operation:
        methods: ["GET"]
---
...
```

As we can see, Istio allows defining not only the source and destination, but in the spec.rules field you can specify the methods and paths fields, as well.

In this way, deploying gradually these policies, we can see from the website (getting an error) how sending a request is not consumed by the workload that has not an Authorization Policy that authorizes the communication.

Kuma

Similarly to Istio, Kuma allows you to define access control rules on workloads, so that you can define the authorized traffic in the Mesh and ensure that only required resources are made available to services. Moreover, also the Traffic permissions of Kuma require Mutual TLS enabled on the Mesh, because it is used by Kuma to validate the service identity with data plane proxy certificates, and if Mutual TLS is disabled, Kuma allows all service traffic.

As we saw in the Kuma chapter, at the installation time Kuma creates a TrafficPermission resource that authorizes all traffic; therefore, before creating the policies, we need to delete the allow−all−traffic TrafficPermission policy (oc delete TrafficPermission allow−all−traffic) and, as for Istio, we can gradually create the various rules:

```
...
---
kind: TrafficPermission
mesh: default
metadata:
  name: productpage-viewer
```

```
spec:
  sources:
    - match:
        kuma.io/service: '*'
  destinations:
    - match:
        kuma.io/service: productpage_bookinfo-kuma_svc_9080
---
apiVersion: kuma.io/v1alpha1
kind: TrafficPermission
mesh: default
metadata:
  name: reviews-viewer
spec:
  sources:
    - match:
        kuma.io/service: productpage_bookinfo-kuma_svc_9080
  destinations:
    - match:
        kuma.io/service: reviews_bookinfo-kuma_svc_9080
        kuma.io/protocol: http
---
...
```

As we can see from the snippet of the manifest, Kuma uses tags to select the target workload to which applies the policy. Unlike Istio, there is no possibility to indicate and filter by operation or path of the request.

Moreover, Kuma requires that all services must be annotated with the right labels in order to select and determine the right service of the policy; for instance, in this case each service is annotated with kuma.io/protocol: http and with the appProtocol : http field.

7.4 Apply a Naïve Deployment Strategy

Sometimes companies develop and deploy a new version of an existing service, but it needs to be tested using a small percentage of user traffic, before being released publicly to all. For this purpose, a Service Mesh solution can help you to implement a deployment strategy.

Since the productpage deployment adds an end−user header we can apply a naïve deployment strategy, simply by forwarding all traffic of a specific user to a particular service version.

Red Hat OpenShift Service Mesh

Before you can use Istio to control the Bookinfo version routing, you need to define the available versions, called *subsets*, in destination rules.

Therefore, for each deployment we define a DestinationRule resource by specifying the following fields:

- **host:** The name of a service from the service registry.
- **subsets:** One or more named sets that represent individual versions of a service.

```yaml
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ratings
spec:
  host: ratings
```

```
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v2-mysql
    labels:
      version: v2-mysql
  - name: v2-mysql-vm
    labels:
      version: v2-mysql-vm
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: details
spec:
  host: details
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Now we can apply the deployment strategy. To do so, we can declare into the VirtualService resource a headers field, under the match field, with the end–user header, the respective username, and the associate destination:

```
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: Marco
    route:
```

```
13      - destination:
14          host: reviews
15          subset: v2
16    - route:
17      - destination:
18          host: reviews
19          subset: v1
```

In this way, user Marco's traffic will be forwarded to version 2 of the reviews service; whereas all other requests will be routed to version 1.

Kuma

Differently from Istio, Kuma v1.6 does not support in the TrafficRoute resource the possibility of forwarding traffic to a specific version based on a field of the header HTTP yet. Therefore, this goal cannot be achieved by Kuma.

Chapter 8

Ingress Controllers

Working on Kubernetes, it is necessary for the cluster to have an ingress controller running so that the Ingress resource can work.

Unlike other types of controllers, running as part of the kube−controller−manager binary, Ingress controllers are not automatically started with a cluster [23].

In this chapter, we will analyze the ingress controllers solution offered by Istio and Kuma. Moreover, we will consider the scenario where a company using a Kong Ingress Controller wants to make its own network secure by applying the zero-trust principles using a Service Mesh technology.

8.1 Istio Ingress Gateway

Along with support for K8s Ingress resource, Istio offers the Istio Gateway configuration model. This configuration provides more extensive customization and flexibility than Ingress.

In order to make the Bookinfo application accessible from outside in Red Hat OpenShift Service Mesh, we have to define the Gateway and VirtualService resources.

As mentioned in the previous paragraphs, the Gateway resource describes a load balancer operating at the edge of the mesh that receives incoming HTTP/TCP connections and allows you to indicate which of them can enter the Mesh; while the VirtualService resource contains the routing information used to reach the correct service to forward the request to.

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: Gateway
3 metadata:
4   namespace: bookinfo
5   name: bookinfo-gateway
6 spec:
```

```
7     selector:
8       istio: ingressgateway # use istio default controller
9     servers:
10      - port:
11          number: 80
12          name: http
13          protocol: HTTP
14        hosts:
15        - "*"
16    ---
17  apiVersion: networking.istio.io/v1alpha3
18  kind: VirtualService
19  metadata:
20    namespace: bookinfo-istio
21    name: bookinfo
22  spec:
23    hosts:
24    - "*"
25    gateways:
26    - bookinfo-gateway
27    http:
28    - match:
29      - uri:
30          exact: /productpage
31      - uri:
32          prefix: /static
33      - uri:
34          exact: /login
35      - uri:
36          exact: /logout
37      - uri:
38          prefix: /api/v1/products
39      route:
40      - destination:
41          host: productpage
42          port:
43            number: 9080
```

As we can see in this manifest, the spec. selector . istio field of the Gateway resource indicates a specific set of pods/VMs on which this gateway configuration should be applied and spec.server describes the properties of the proxy on a given load balancer port.

In the VirtualService resource, the spec.gateways indicates the names of gateways and sidecars that should apply these routes. Finally we can specify an ordered list

of route rules for HTTP traffic in the spec.http field.

8.2 Kuma Gateway

Although Kuma is predisposed to manage a gateway because it provides a special type of data plane proxy, Kuma v1.6 does not offer an official built-in implementation. But, as we can see in the documentation, an experimental implementation can be tested by forcing the installation of the CRDs through the ––experimental–meshgateway flag at the installation time.

Because of its instability, due to the fact that this version is still in an experimental phase, Kuma built-in Gateway will not be discussed as a topic in this thesis.

However, in the next paragraph, we will examine a delegated approach that allows users to use any existing gateway: for example, Kong Ingress Gateway.

8.3 Kong Ingress Controller

Nowadays, if you want to expose any service running inside Kubernetes, a company can use several solutions: a very handy one is to have an Ingress.

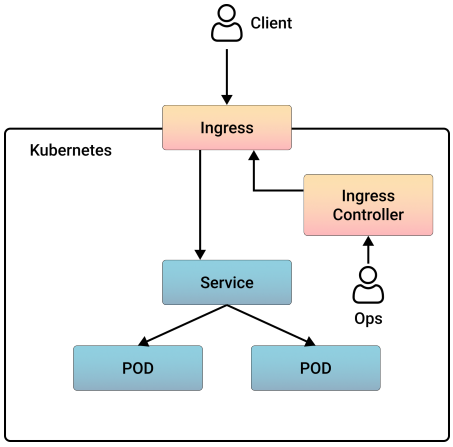

Figure 8.1: Ingress Architecture.

Kong Ingress Controller is one of the main solutions chosen by companies as it allows the management of the routing rules that control external user access to the service in a Kubernetes cluster from the same platform.

In the following paragraphs, we will see if the Service Mesh solutions analyzed above (Istio and Kuma) can be used together with the ingress controller developed by Kong in order to do a zero-trust network.

8.3.1 Running the Kong Ingress Controller with OpenShift Service Mesh

Unfortunately, after several tests and with the help of the community, we can say that Kong Ingress Controller v2.6 and OpenShift Service Mesh OSSM_2.1.3 are incompatible. This is because Kubernetes resources are mapped to Kong resources to correctly proxy all the traffic, in particular, Kong maps Pods as a Target belonging to the Upstream (the upstream corresponding to the Kubernetes Service), and when a request reaches the Kong Ingress controller, it is not directed via kube−proxy but directly to the pod [24]. This causes a mismatch between Istio and Kong, because Istio is not capable to work in this way by default.

8.3.2 Running the Kong Ingress Controller with Kuma

Unlike Istio, Kuma was designed to work with a delegate Gateway. In particular, it offers full compatibility combined together with Kong Ingress Controller, because they derive from the same company.

To be integrated with existing API Gateways, the Kuma Dataplane entity operates in gateway mode. This mode lets you skip exposing inbound listeners so it won't be intercepting ingress traffic [12].

At this point, considering the Bookinfo demo application, after having installed the Kong Ingress Controller (for instance with the helm char), we can look at how to use the Ingress resource to help route external traffic to the services within the Kuma mesh.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    namespace: bookinfo
5    name: bookinfo-ingress
6  spec:
7    ingressClassName: kong
8    rules:
9    - http:
10       paths:
11       - path: /productpage
12         pathType: ImplementationSpecific
13         backend:
14           service:
15             name: productpage
16             port:
17               number: 9080
18       - path: /static
```

```
19          pathType: Prefix
20          backend:
21            service:
22              name: productpage
23              port:
24                number: 9080
25        - path: /login
26          pathType: ImplementationSpecific
27          backend:
28            service:
29              name: productpage
30              port:
31                number: 9080
32        - path: /logout
33          pathType: ImplementationSpecific
34          backend:
35            service:
36              name: productpage
37              port:
38                number: 9080
39        - path: /api/v1/products
40          pathType: ImplementationSpecific
41          backend:
42            service:
43              name: productpage
44              port:
45                number: 9080
```

In this way, thanks to the spec.ingressClassName field, Kong Ingress Controller is able to understand the rules you defined in the Ingress resource and routes it to the productpage service.

Chapter 9

Latency Evaluation

Although Service Mesh platforms allow organizations to split cross-cutting concerns like policy control, authorization, authentication, encryption, and so on, from the main application with no changes to the application code, they create a new additional layer into the cluster.

A request before being processed by the workload passes through the sidecar proxy which is responsible for authenticating and authorizing the request.

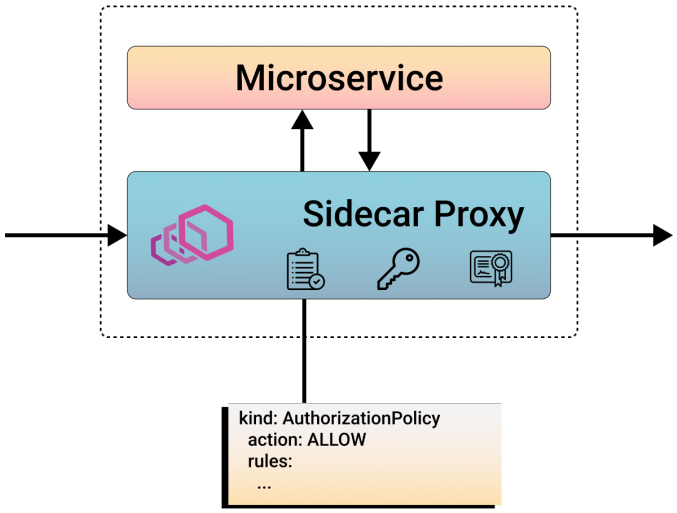

Figure 9.1: Proxy policies and components.

In this chapter, we will evaluate how the latency varies when you apply a Service Mesh to an application, namely how and if the additional step that a request makes before being processed might affects the latency.

For this purpose, we will analyze the case in which the policies seen in chapter

7 are applied to the Bookinfo demo application.

9.1 Load Generator

For the creation of the load generator, it has been used an open-source load testing tool, named Locust [25].

Locust allows defining the user behavior with simple Python scripts. Using this tool, each user implements a closed-loop model, namely that new requests are only sent after previous requests' completion. Moreover, it lets swarm your system with a large number of simultaneous users.

In order to perform the tests, it has been written a Python script that sends a new request every 0.1 seconds at the path http://productpage:9080/productpage.

After creating the image from the Dockerfile, we deployed the load generator into our cluster.

Finally, we exposed the TCP port 8089 of the load generator service so that we could run the oc port−forward service/loadgenerator 8089 command and interact with the web UI made available by Locust in order to define and start the various tests.

9.2 Tests

Using an application with few services and having few resources available for the cluster, it has not been possible to carry out a stress test at great levels. Nevertheless, from the tests carried out you can notice some differences.

In the next sections, we take a look at the performance of the configuration. We will see three different comparisons:

- How latency is affected by the presence of OpenShift Service Mesh and the presence of the policies applied to the Mesh.
- How latency is affected by the presence of Kuma and the presence of the policies applied to the Mesh.
- The last comparison will contrast the final configurations of the two platforms, OpenShift Service Mesh and Kuma. We will see the variation between the policies defined with Istio and those defined with Kuma.

Various tests were made to choose the right number of active users to use, because the ultimate goal was not to stress the demo application, not even to see how a service mesh reacts to the increase of users, but rather to calculate the latency difference between the various configurations.

Therefore, knowing that:

1. The tool used to generate the workload uses a closed-loop model.

2. Using a closed-loop model, the request generation is limited by a maximum RPS.

It has been executed a first initial test with five different configurations of users in order to find the right number that did not saturate the application too much.

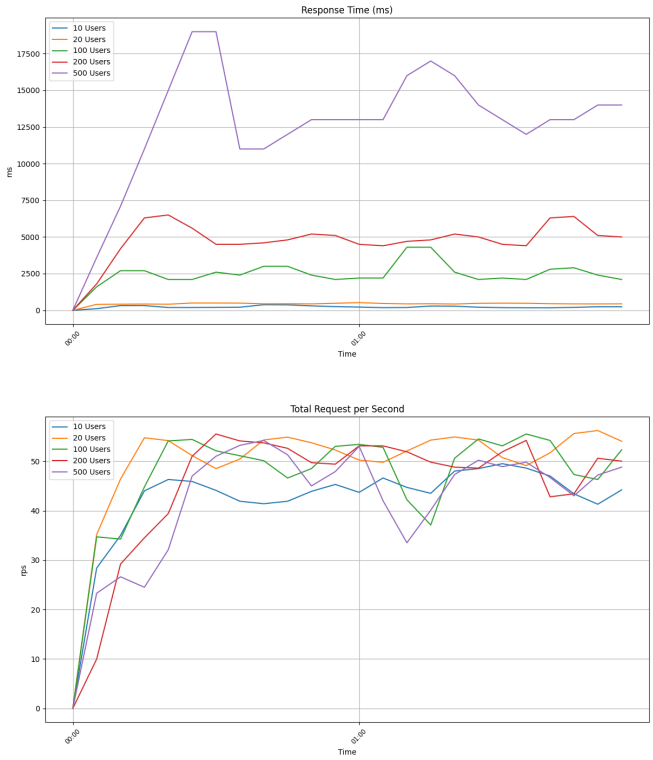

Figure 9.2: Response time and RPS.

Solution	Average RPS	Average 95%ile (ms)
10 Users	41.96	225.42
20 Users	49.52	435.56
100 Users	46.79	2454.17
200 Users	44.89	4725.0
500 Users	41.94	12612.5

Table 9.1: Users' Latency and RPS

The graph above 9.2 illustrates the latency and RPS values, in case you use 10, 20, 100, 200, or 500 users. Increasing the number of users does not bring any increase in terms of response per second, but on the contrary, tends to saturate the application which will respond at a lower rate for a while and reduce the average RPS of the testing. The average of the latency and RPS values for each configuration can be displayed in the table 9.1.

Therefore, in the next tests, we will evaluate the performance considering the P95 latency obtained using a load generator that simulates the presence of 20 active users. In addition, data reported for each scenario is derived from the average of the first 120 seconds of the value obtained by executing the same scenario three times.

9.2.1 First test case

For the first test case, we analyzed the latency difference among three environments:

1. In the first environment, we consider only the demo application without any Service Mesh.

2. Next, we applied the OpenShift Service Mesh to the demo application.

3. Finally, we also applied the policies seen in chapter 7, thus communication encryptions by means of mTLS and authorization policies.

As we can see from the chart 9.3, the behavior of the demo application and the demo application with Istio is pretty the same.

The average P95 latency added by the proxy is 50.69 ms with respect to 435.56 ms obtained with the basic configuration.

Considering also the presence of the Istio policies the difference from the basic configuration increases to 223.61 ms.

Moreover, as we can see, the last configuration (Istio with policies) in the first seconds of the test has a peak. It is probably due to a first initial configuration of the proxy because at each test the application has been reinitialized.

Obviously, the application used for this test has few micro-services and each additional micro-service in the workflow will probably add a little bit more overhead.

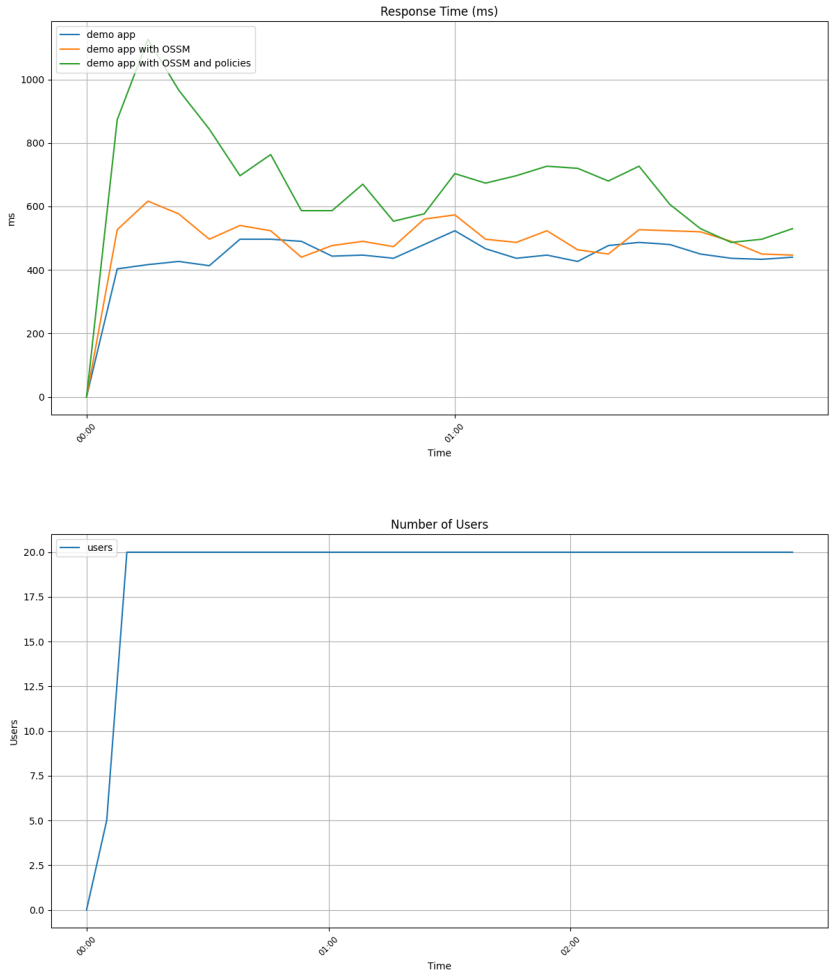

Figure 9.3: Response time first test case.

9.2.2 Second test case

As for the first test case, we analyzed the latency difference among three different configurations, but in this case, the Service Mesh used is the Kuma Mesh.

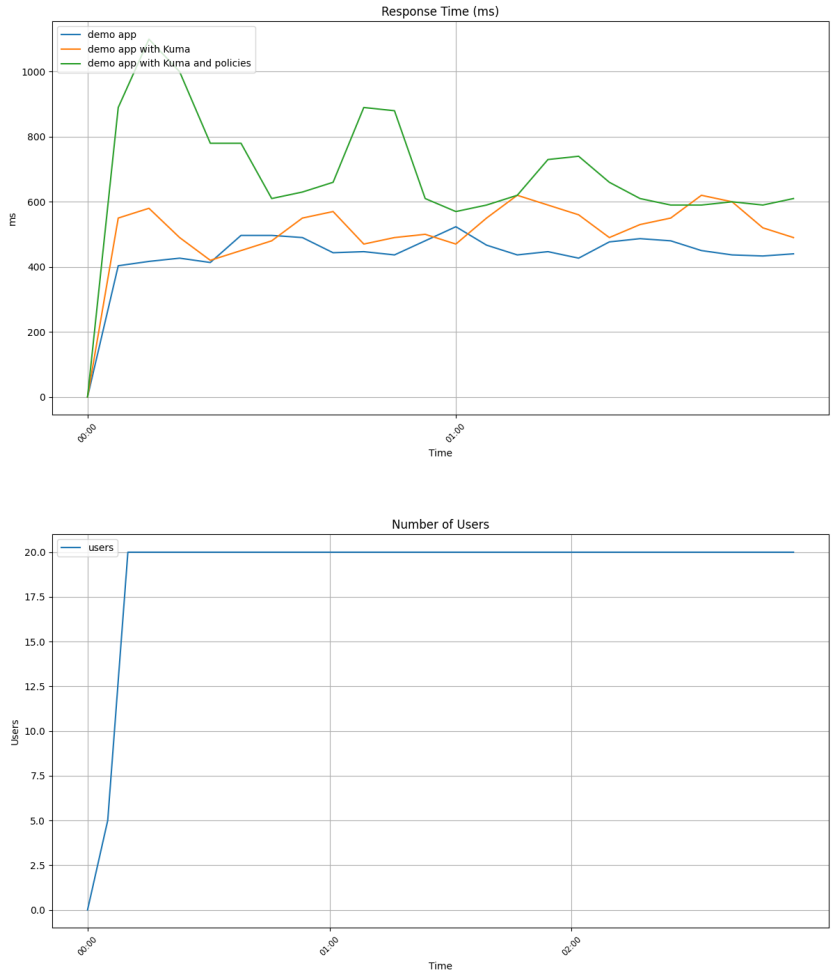

Figure 9.4: Response time second test case.

As you can see in the charts above 9.4, the behavior of the first two environments remains quite the same. The difference in the average P95 latency between the baseline solution (without Service Mesh) and the presence of Kuma is 70.27 ms, with 505.83 ms for Kuma and 435.5 ms for the basic configuration.

As for the Istio configuration, the average P95 latency for the last configuration increases, reaching an average value of 680.42 ms.

As for the previous test, increasing the depth of the call chain in the application may incur more performance penalties.

9.2.3 Third test case

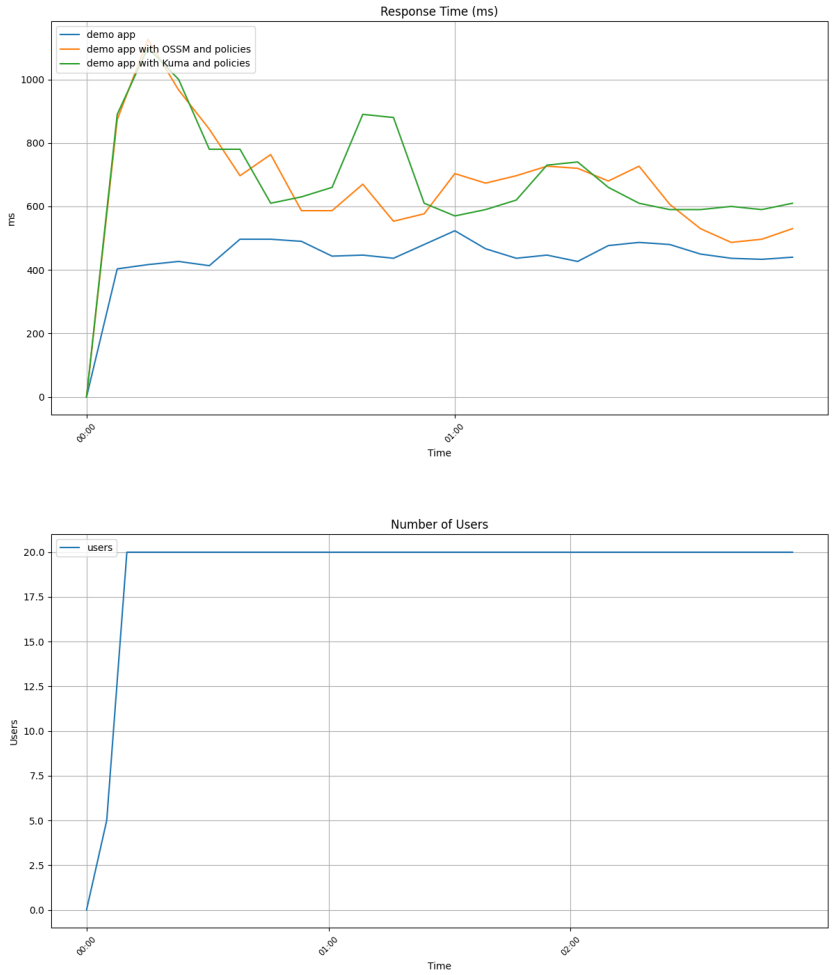

Figure 9.5: Response time third test case.

Finally, we can compare the last configurations of both Service Mesh.

We can see also in this case (fig. 9.5) as OpenShift Service Mesh performs a little bit better than Kuma, with a difference of 21,25 ms.

Therefore, as we expected, the results obtained added in both cases a response delay (nearby 200 ms from the tests performed) and considering a scenario where each ms is essential for the application scope, using a mesh solution is probably not the right choice. However, in a scenario where such an increase is tolerable, the added value of a Service Mesh as we could see in the previous chapters is remarkable because it reduces the efforts of development teams to maintain the

various cross-cutting requirements in each micro-service.

Chapter 10

Conclusion

From the comparisons shown in chapter 7, we can conclude that although both solutions allow the implementation of communication encryptions, Istio tracks workloads with an Istio proxy and configures client proxies to automatically send mTLS traffic to those workloads and regular text traffic to workloads without a sidecar (PERMISSIVE mode); moreover, a STRICT mTLS mode can be configured in order to set up communication only in mTLS. On the other hand, Kuma does not enable by default the mTLS, thus it must be done explicitly by modifying the Mesh resource and specifying the PERMISSIVE mode if you want to apply a behavior similar to Istio's initial one. In addition, Kuma gives the possibility to specify two useful fields for certificate generation.

Regarding the Authorization Policies, Istio provides fine-grain policies with three different levels of access control: mesh-wide, namespace-wide, and workload-wide. Unlike Istio, Kuma does not provide the possibility to indicate and filter by operation or path of the request. Moreover, Kuma requires that all services must be annotated with the right labels in order to select and determine the right service of the policy.

Considering the naïve deployment strategy, we have seen that Kuma v1.6 does not support the possibility of forwarding traffic to a specific version based on a field of the header HTTP yet.

Finally, we have seen that OpenShift Service Mesh (as well as Istio) does not currently allow to be integrated with Kong Ingress Controller because they operate through two different mechanisms regarding routing of requests. On the other hand, Kuma cooperates perfectly with Kong Ingress Controller, in fact, an enterprise solution called Kong Mesh [26] is also offered by Kong.

In the end, we can affirm that although both Istio and Kuma have implemented all use cases in analysis, unlike Istio, Kuma has resulted not to be a very mature technology, because it cannot apply fine-grain policies or, as we have seen, it cannot forward all traffic of a specific user to a particular service version.

Moreover, also considering the measures performed during chapter 9, we can say that Istio performed slightly better than Kuma. In fact, as we have seen, Istio response delay is about 21 ms less than Kuma.

Therefore, in a context in which the latency obtained during the tests from the two Service Meshes analyzed in this thesis turns out to be tolerable, I think it is more appropriate to use Istio as a service mesh to implement the previous use cases because it allows creating fine-grain policies.

Bibliography

[1] Anita Buehrle. *Introduction to Service Meshes on Kubernetes and Progressive Delivery*. URL: https://dzone.com/articles/introduction-to-service-meshes-on-kubernetes-and-p (cit. on p. 2).

[2] Kubernetes. *Kubernetes docs*. URL: https://kubernetes.io/docs/home/ (cit. on pp. 4, 5, 7, 8).

[3] Red Hat. *What is a Kubernetes operator?* 2022. URL: https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator (cit. on p. 11).

[4] SparkFabrik Team. *Kubernetes Operator: cosa sono*. 2022. URL: https://blog.sparkfabrik.com/it/kubernetes-operator-cosa-sono (cit. on p. 11).

[5] Red Hat. *Red Hat OpenShift Docs*. URL: https://docs.openshift.com/container-platform/4.8/ (cit. on pp. 13–20).

[6] Jaafar Chraibi. *A Guide to Enterprise Kubernetes with OpenShift*. 2020. URL: https://cloud.redhat.com/blog/enterprise-kubernetes-with-openshift-part-one (cit. on pp. 14, 17, 18).

[7] Red Hat. *An introduction to enterprise Kubernetes*. URL: https://www.redhat.com/en/engage/introduction-enterprise-kubernetes-s-202003040257 (cit. on p. 15).

[8] Gilad David Maayan. *OpenShift vs. Kubernetes: The Seven Most Critical Differences*. 2020. URL: https://www.dataversity.net/openshift-vs-kubernetes-the-seven-most-critical-differences/ (cit. on pp. 18–20).

[9] IBM. *How much does a data breach cost?* 2021. URL: https://www.ibm.com/security/data-breach (cit. on p. 21).

[10] Ashher Syed. *Applying Zero Trust Security to Kubernetes Via Service Mesh*. 2022. URL: https://thenewstack.io/applying-zero-trust-security-to-kubernetes-via-service-mesh (cit. on p. 22).

[11] Red Hat. *What's a service mesh?* 2018. URL: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (cit. on p. 23).

[12] Kuma. *Kuma Docs*. URL: https://kuma.io/docs/1.6.x/ (cit. on pp. 25–32, 34, 61).

[13] Bill Doerrfeld. *Introducing Kuma, the Service Mesh From Kong*. 2020. URL: https://containerjournal.com/topics/container-networking/introducing-kuma-the-service-mesh-from-kong/ (cit. on p. 25).

[14] Kong. *Kong API Gateway*. URL: https://github.com/Kong/kong (cit. on p. 29).

[15] Red Hat. *What is Red Hat OpenShift Service Mesh?* URL: https://www.redhat.com/en/technologies/cloud-computing/openshift/what-is-openshift-service-mesh (cit. on p. 35).

[16] Red Hat. *Introduction to Istio service mesh*. URL: https://developers.redhat.com/topics/service-mesh (cit. on p. 36).

[17] Red Hat. *Service Mesh*. URL: https://access.redhat.com/documentation/en-us/openshift_container_platform/4.1/html/service_mesh/service-mesh-architecture (cit. on pp. 36, 37).

[18] Kiali. *About Kiali*. URL: https://pre-v1-41.kiali.io/ (cit. on p. 36).

[19] Istio. *Istio Docs*. URL: https://istio.io/latest/docs/ (cit. on pp. 38–47, 52).

[20] Wang Xining. *North-South Traffic Management of Istio Gateways (with Answers from Service Mesh Experts)*. 2020. URL: https://www.alibabacloud.com/blog/north-south-traffic-management-of-istio-gateways-with-answers-from-service-mesh-experts_596658 (cit. on p. 39).

[21] Istio. *Bookinfo*. URL: https://github.com/istio/istio/tree/master/samples/bookinfo (cit. on pp. 48, 49).

[22] Ashher Syed. *Applying Zero Trust Security to Kubernetes Via Service Mesh*. 2022. URL: https://thenewstack.io/applying-zero-trust-security-to-kubernetes-via-service-mesh (cit. on p. 51).

[23] Kubernetes. *Ingress Controllers*. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (cit. on p. 58).

[24] Kong Ingress Controller. *Concepts overview*. URL: https://github.com/Kong/docs.konghq.com/blob/main/src/kubernetes-ingress-controller/concepts/design.md#translation (cit. on p. 61).

[25] *Locust*. URL: https://locust.io/ (cit. on p. 64).

[26] Kong. *Kong Mesh*. URL: https://docs.konghq.com/mesh/latest/ (cit. on p. 71).