

POLITECNICO DI TORINO

## Service Mesh

---



April 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Kubernetes</b>	<b>3</b>
2.1	What's Kubernetes . . . . .	3
2.2	Kubernetes architecture . . . . .	3
2.2.1	Cluster Setup . . . . .	5
2.3	Container Network Interface (CNI) . . . . .	6
2.3.1	Cilium & Setup . . . . .	6
2.4	Kubectl . . . . .	8
2.5	Pods . . . . .	8
2.6	Namespace . . . . .	9
2.7	Deployment . . . . .	10
2.8	Service . . . . .	12
<b>3</b>	<b>Service Mesh</b>	<b>15</b>
<b>4</b>	<b>Istio</b>	<b>16</b>
4.1	Istio Setup . . . . .	16
4.1.1	Mutual TLS (mTLS) . . . . .	17
4.1.2	AuthorizationPolicy . . . . .	18
4.2	Hands-On with Istio . . . . .	20
4.2.1	Applying Istio mTLS . . . . .	27
4.2.2	Applying Istio Authorization Policies . . . . .	27
4.2.3	Testing the Security Policies . . . . .	30
<b>5</b>	<b>Linkerd</b>	<b>32</b>
5.1	Linkerd Setup . . . . .	32
5.2	Securing micro-services with Linkerd . . . . .	34
5.2.1	Deploying the workloads . . . . .	34
5.2.2	Exercise A - Retry budget . . . . .	40
5.2.3	Exercise B – DoS route Breaking . . . . .	42
5.2.4	Final Considerations . . . . .	44

# 1 Introduction

The purpose of this laboratory is to acquire hands-on experience in Kubernetes and Service Mesh technologies. Unlike low-level virtualization, Kubernetes orchestrates the deployment, scaling, and management of containerized applications by abstracting the complexity of an underlying infrastructure.

A Service Mesh further enhances Kubernetes capabilities by providing advanced networking and security features without modifying the application code. During this laboratory, first it will be explained Kubernetes basics, where you will learn how to set up a Kubernetes cluster and deploy a simple application. Then, you will move on to more complex and security-relevant scenarios, where you will experiment with the security features provided by Service Mesh solutions such as Istio and Linkerd.

All exercises related to Kubernetes and Service Meshes will be conducted in a controlled virtual environment using virtual machines provided by Crownlabs <https://crownlabs.polito.it/>.

## 2 Kubernetes

### 2.1 What's Kubernetes

Kubernetes is an open source container orchestration platform for automating deployment, management, and scaling of containerized applications [1]. It offers several significant advantages that simplify the workload behind the use of containers (here are reported the main ones):

- **Automated deployment and scalability:** Automatically manages the deployment, scaling, and updating of applications, enabling efficient management of large-scale systems.
- **High availability and reliability:** Ensures application resiliency by continuously monitoring application status and automatically restarting failed containers.
- **Load balancing and service discovery:** Provides built-in load balancing and dynamic discovery services, allowing applications to communicate across the cluster.
- **Resource optimization:** Schedules containers on cluster nodes based on available resources, optimizing infrastructure utilization.
- **Declarative configuration management:** Supports infrastructure-as-code through declarative YAML manifests, making it easy to manage complex systems.

These capabilities make Kubernetes an essential tool for managing modern cloud-native applications, allowing developers and administrators to achieve greater operational agility and efficiency.

### 2.2 Kubernetes architecture

The Kubernetes architecture is based on the concept of **cluster**, which is a group of nodes (physical or virtual machines) that work together to run containerized

applications. Typically, a cluster is composed of a **master node** and multiple **worker nodes**.

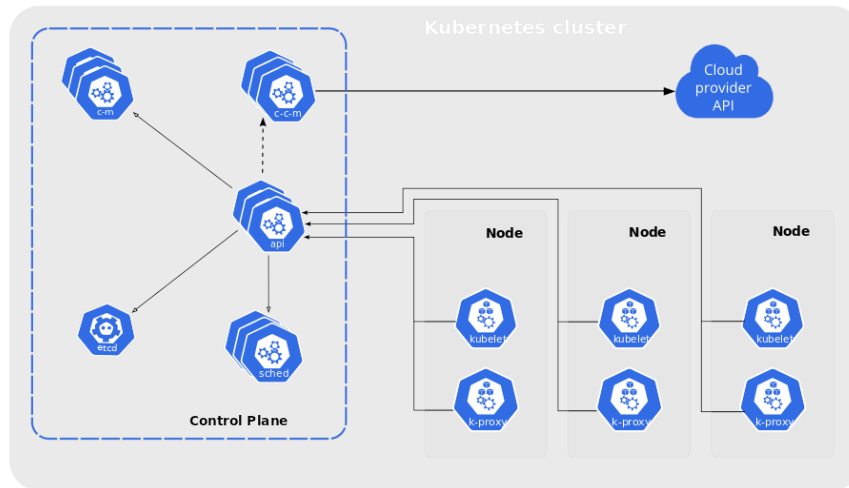


Figure 1: Kubernetes Cluster

As shown in Figure 1, the main components of a Kubernetes cluster are:

- **Control Plane:** Responsible for managing the clusters overall state and making global decisions. It processes API requests and orchestrates workloads. The control plane is typically composed of:
  - **API Server:** The entry point for all cluster interactions. It exposes the Kubernetes API and handles REST requests.
  - **etcd:** A distributed key-value store that holds the entire cluster configuration and state.
  - **Scheduler:** Assigns Pods to nodes based on resource availability and policy constraints.
  - **Controller Manager:** Monitors the clusters desired state and makes adjustments (e.g., restarting failed Pods).
  - **Cloud Controller Manager:** Interfaces with cloud providers to manage infrastructure-specific resources.

- **Worker Nodes:** These are the machines where containers run. Each worker node includes:
  - **Container Runtime:** The software that pulls and runs containers (e.g., Docker).
  - **Kubelet:** An agent that ensures containers are running as defined and reports node status to the control plane.
  - **kube-proxy:** Manages networking rules to allow communication between Pods and supports load balancing through Services.

### 2.2.1 Cluster Setup

In order to set up a Kubernetes cluster, you need to create two instances of the Kubernetes VM (one for the master and one for the worker node).

Run the following commands on the master node:

#### Cluster Setup

```
$ sudo kubeadm init --pod-network-cidr=10.245.0.0/16
--service-cidr=10.255.0.0/16
$ mkdir -p $HOME/.kube
$ sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

These commands initialize the Kubernetes control plane with custom network CIDRs, then configure local access to the cluster using kubectl by setting up the kubeconfig file in the user's home directory.

After the first command, if you read carefully the output on the terminal, you can see a command like:

#### Cluster Setup

```
$ kubeadm join <ip>:<port> --token <token>
--discovery-token-ca-cert-hash <hash>
```

Copy that and paste it into the worker node terminal using root privileges. Now, to check that everything is properly installed, run:

#### Cluster Setup

```
$ kubectl get nodes
```

You should see that the two nodes are in a **NotReady** state. This is expected: Kubernetes marks a node as **Ready** only when the network plugin (CNI) is also running. In the next Section it will be more clear.

## 2.3 Container Network Interface (CNI)

A **Container Network Interface (CNI)** is a plugin system that Kubernetes uses to provide network connectivity to Pods. CNIs are responsible for assigning IP addresses to Pods, configuring routing tables, and enforcing network policies, ensuring that containers can communicate both within and across nodes in a cluster. Without a functioning CNI, Pods would not be able to reach each other. Kubernetes itself does not provide native networking capabilities in fact, relies on CNI plugins for these tasks. There are several CNI options available, each offering different features and levels of complexity. A complete list of supported plugins can be found at: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.

### 2.3.1 Cilium & Setup

In this lab, you will use the **Cilium** CNI plugin. Cilium is a modern and powerful CNI built on top of **eBPF** (Extended Berkeley Packet Filter), a Linux kernel technology that allows safe and efficient execution of user-defined programs within the kernel [2].

Unlike traditional CNIs, Cilium enables high-performance networking by dynamically processing traffic at the kernel level. This makes it particularly well-suited for modern microservices environments where security, observability, and performance are essential.

For this laboratory the version used is the v0.16.24 available at: <https://github.com/cilium/cilium/releases>, and it can be installed using the following commands:

#### Installing Cilium

```
$ CILIUM_CLI_VERSION=v0.16.24
$ curl -L --remote-name-all \
  https://github.com/cilium/cilium-cli/releases/ \
  download/${CILIUM_CLI_VERSION}/cilium-linux-amd64.tar.gz

# Uncompress the TAR archive in the /usr/local/bin folder
$ sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin

# Install Cilium as the CNI
$ cilium install

# Verify the status of the installation
$ cilium status
```

**Note:** After installing Cilium, it may take some minutes before that the installation is fully completed. To check the status of the installation run:

#### Installing Cilium

```
# Verify the status of the installation
$ cilium status
```

Run it until all components are listed as healthy and no errors are returned. Once Cilium is working properly, run:

#### Installing Cilium

```
$ kubectl get nodes
```



Finally, you should see that the two nodes are in **Ready** state, which confirms that the Cilium installation was successful and the cluster is now fully operational. At this point, any new Pod created will be connected to the cluster via Cilium.

## 2.4 Kubectl

Kubectl is the Kubernetes command-line interface (CLI) used to interact with clusters. It enables users to perform operations such as deploying applications, inspecting cluster resources, managing Pods, Services, and Deployments. Kubectl works by communicating directly with the Kubernetes API Server, making it an essential tool for developers to control Kubernetes environments.

In Kubernetes, whenever you want to create or update every kind of resource, such as a Pod, deployment, service, or namespace, you need to use the `kubectl apply` command. This command takes a YAML or JSON manifest file that defines the desired state of your resource and applies it ensuring that the current configuration in the cluster matches what is defined in your file. If the resource does not exist, it is created, otherwise is updated accordingly. So, every time you need to apply or delete resources you have to use the following commands:

### Command to Work with resources

```
$ kubectl apply -f <file.yaml>
$ kubectl delete -f <file.yaml>
```

Many other `kubectl` operations will be introduced in the next sections (e.g., inspecting Pods, creating Namespaces, exposing Services, etc.).

For a complete list of available commands, see the official Kubernetes documentation at: <https://kubernetes.io/docs/reference/kubectl/quick-reference/>.

## 2.5 Pods

In Kubernetes, a Pod represents the simplest deployable unit that can be managed. A Pod encapsulates one or more related containers (such as Docker containers) sharing storage, network resources and configuration details on how they should

be executed.

Key characteristics of Pods include:

- **Atomic unit of deployment:** Containers within a Pod are always scheduled together on the same node.
- **Shared context:** Containers in a Pod share an IP address and ports, enabling direct communication via localhost.
- **Ephemeral nature:** Pods are designed to be transient; Kubernetes automatically replaces Pods in case of failure.

Commands to work with pods:

#### Pods

```
# Retrieve pods
$ kubectl get pods

# Show more detail about a specific pod
$ kubectl describe pod <pod-name>

# Delete a specific pod
$ kubectl delete <pod-name>
```

## 2.6 Namespace

In Kubernetes environments, another crucial aspect is the Namespace. It is used to provide a mechanism for isolating groups of resources within a single cluster. You can think of namespaces as separate folders within your cluster, each isolated from the others.

Using namespaces is particularly recommended for managing multiple environments such as development, testing, and production, avoiding resource conflicts. Namespaces are also valuable in collaborative settings where different teams or users operate within the same cluster, as each group can work in its own isolated

space. In addition, they make cleanup much easier, as deleting a namespace removes all its resources in one command.

A custom namespace, however, is not necessary in cases like quick local tests, minimal setups where the default namespace is sufficient, or when you are simply learning the basic behavior of Kubernetes components such as Pods or Services.

In our labs, we will perform all operations within the default namespace.

Here are presented some commands to work with namespaces.

#### Namespace Creation

```
$ kubectl create namespace <namespace_name>
```

#### Namespace Deletion

```
$ kubectl delete namespace <namespace_name>
```

#### Show Existing Namespaces

```
$ kubectl get namespace
```

## 2.7 Deployment

A **Deployment** in Kubernetes is a high-level abstraction used to manage the lifecycle of application instances encapsulated in Pods. It defines the desired state (number of replicas, container image, ports, etc.), and Kubernetes continuously ensures that the actual state matches this configuration. This approach is the so-called **declarative management**: instead of tell to the system what to do, you describe, in a YAML manifest, your desired outcome. Kubernetes, then, will take care of maintaining that state.

Below is presented an example of a simple Deployment YAML manifest:

#### backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: hashicorp/http-echo
          args: ["-text=Hello from backend"]
          ports:
            - containerPort: 3000
```

Here is a breakdown of the main fields:

- **apiVersion:** Defines the API version to be used (`apps/v1` is the standard one for Deployments).
- **kind:** Specifies the type of Kubernetes object in this case, a `Deployment`.
- **metadata.name:** The name assigned to the Deployment.
- **spec.replicas:** Number of identical Pods to be created and maintained.

- **spec.selector.matchLabels:** Ensures that the Deployment targets the correct set of Pods based on labels.
- **containers:** Defines the containers inside each Pod. Here we use the image `hashicorp/http-echo`, which is a lightweight web server used to return a fixed string.
- **args:** Defines the arguments passed to the container.
- **containerPort:** Declares the port that the container will listen on internally.

## 2.8 Service

A Kubernetes **Service** provides a stable endpoint to expose a set of Pods over the network. This abstraction allows applications to discover and communicate with each other without needing to track IP address changes caused by Pod rescheduling.

There are different types of Services:

- **ClusterIP (default):** Only accessible within the cluster.
- **NodePort:** Maps a port on each node to the service used for basic external access.
- **LoadBalancer:** Integrates with cloud providers to expose services to the public Internet.

In our case, we will create a **ClusterIP** Service to expose the HTTP endpoint of our backend Deployment. The Service will forward traffic from port 80 to the backend container listening on port 3000.

#### backend-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 3000
```

#### Explanation of the fields:

- `ports.port`: Exposes port 80 on the service (virtual IP).
- `ports.targetPort`: Maps port 80 to port 3000 inside the Pod container.

1. Create a new folder (e.g., `testService`) and add both YAML files: `backend-deployment.yaml` and `backend-service.yaml`.
2. Apply the YAML files using:

#### Apply YAML

```
$ kubectl apply -f backend-deployment.yaml
$ kubectl apply -f backend-service.yaml
```

3. Verify that the Pod is running:

```
$ kubectl get pods
```

4. Create a temporary Pod to test the service:

#### Create Test Client

```
$ kubectl run client --rm -it --restart=Never  
↪ --image=radial/busyboxplus:curl -- sh
```

5. Inside the client shell, call the service:

#### Call the Service

```
$ curl http://backend-service
```

6. You should see:

```
Hello from backend
```

**Why does this work?** Because the Service you created (of type `ClusterIP`) assigns a stable internal DNS name to the backend Pods: `backend-service`. This name is automatically resolved by Kubernetes into the IP address of one of the available backend Pods.

Even if Pods restart, change IP, or scale up/down, clients can continue to access the application by referencing the Service name, not the individual Pod IPs. This mechanism eliminates the need to manually track or hardcode Pod addresses a major advantage in real-world, dynamic systems.

7. Type `exit` to leave the client pod.

### 3 Service Mesh

Nowadays, applications adopt microservice architectures, and with this the number of independent services that need to communicate with each other grows fast. This introduces several challenges, especially around communication, security, and observability. Kubernetes does not natively provide excellent control over these aspects.

A **Service Mesh** is an infrastructure layer that runs with the application code to obtain secure, observable, and reliable communication. These features are offered to microservices without requiring changes to the application itself (here are reported the main ones):

- **Mutual TLS (mTLS)**: Automatically encrypts traffic between services and authenticates their identity.
- **Fine-grained traffic policies**: Allows control over which services can communicate with each other, and under which conditions (methods, paths, etc.).
- **Telemetry and observability**: Collects detailed metrics, logs, and traces of service-to-service interactions.
- **Resiliency features**: Supports retries, timeouts, and circuit breakers to improve system reliability.
- **Zero-trust security**: Implements least-privilege communication by default.

Service meshes typically follow the **sidecar proxy model**, where a lightweight network proxy is deployed alongside each service instance. This proxy intercepts all inbound and outbound traffic, enabling the mesh to:

- Secure traffic transparently
- Enforce access control policies
- Collect telemetry data (e.g., request latency, error rates)
- Route traffic intelligently (e.g., canary deployments)

In this lab, you will use **Istio** which uses the **Envoy** proxy as its data plane, and **Linkerd** which uses **Linkerd2-Proxy**, a micro proxy developed in Rust.



## 4 Istio

Istio is an open source service mesh that layers transparently onto existing distributed applications. Istio's powerful features provide a uniform and more efficient way to secure, connect, and monitor services. Istio is the path to load balancing, service-to-service authentication, and monitoring with few or no service code changes [3]. It gives you:

- **Mutual TLS (mTLS):** Automatically encrypts traffic between services and verifies the identity of each party through certificates.
- **Envoy sidecars:** Deployed with every Pod, Envoy proxies intercept and control all traffic entering and leaving the service.
- **Authorization and Access Control:** Istio supports access control policies that can allow or deny requests based on source identity, request path, method, and more.

In this lab, you will focus primarily on Istio's security capabilities: enabling mTLS encryption and enforcing access policies using AuthorizationPolicy resources. These features will be applied to a set of microservices running in a Kubernetes cluster.

### 4.1 Istio Setup

To begin using Istio, it must be installed in the Kubernetes cluster. The version used in this lab is v1.16.7, available at <https://github.com/istio/istio/releases>.

Before installing Istio, you must update the Cilium configuration to ensure compatibility between the CNI and the service mesh. Run the following:

#### Edit Cilium Config

```
$ kubectl -n kube-system edit configmap cilium-config
```

Set the following values:

- `cni-exclusive:` `"true"`

- `bpf-lb-sock-hostns-only`: `"true"` (add it if not already present)

Now, proceed with the Istio installation:

#### Install Istio

```
$ curl -L https://istio.io/downloadIstio |  
ISTIO_VERSION=1.16.7 sh -  
$ cd istio-1.16.7/  
$ export PATH=$PWD/bin:$PATH  
$ istioctl install --set profile=demo -y
```

Once the installation is complete, verify that all Istio pods are running:

#### Check Istio

```
$ istioctl version  
$ kubectl get pods -n istio-system
```

Finally, enable automatic sidecar injection for your working namespace (in this case, the default namespace):

#### Enable Sidecar Injection

```
$ kubectl label namespace default istio-injection=enabled
```

This label ensures that all future Pods deployed in that namespace will automatically receive the Envoy sidecar. Without it, Istio features like mTLS or traffic authorization would not apply to those Pods.

#### 4.1.1 Mutual TLS (mTLS)

Mutual TLS is a powerful security feature that encrypts communication and ensures strong identity verification between services.

To enable mTLS in the entire namespace, apply the following PeerAuthentication manifest:

#### peer-auth.yaml

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

This file enforces mutual TLS for every pod in the default namespace by setting a `PeerAuthentication` resource with the mode set to `STRICT`. Once the manifest is applied, all traffic between sidecars inside that namespace is authenticated and encrypted. With `STRICT` mode, each Envoy proxy (Istio sidecar) automatically uses its own certificate when sending requests to other services. At the same time, the Istio sidecar verifies the certificates of incoming requests to confirm that they are issued by Istio and are valid. If a request arrives without a valid certificate, the proxy automatically rejects it, enforcing only secure communications within the mesh.

In the next lab section, you will apply this manifest before deploying the microservices. This guarantees that every call between services is protected by mutual TLS.

#### 4.1.2 AuthorizationPolicy

`AuthorizationPolicy` allows developers to define who can access what, and under what conditions. It enables service-level access control using rules based on identity.

Here is a generic example:

#### authorization.yaml

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-specific-access
  namespace: default
spec:
  selector:
    matchLabels:
      app: target-service
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/client-a"]
      to:
    - operation:
        methods: ["POST"]
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/client-b"]
      to:
    - operation:
        methods: ["GET"]
```

#### How it works:

- **selector:** selects the target workload (e.g., the service receiving requests).
- **from.source.principals:** defines which ServiceAccounts are allowed to initiate the request.
- **to.operation.methods:** limits access to specific HTTP methods (e.g., GET, POST).

In this lab, you will apply three AuthorizationPolicies, Once deployed, you will test access from the various services to verify that unauthorized requests are rejected with "RBAC: access denied" and authorized requests go through.

## 4.2 Hands-On with Istio

In this part of the laboratory, you are going to create a three-microservices application that implements the security features previously introduced. The application is used to handle the aspects related to the exam booking process. Here, the microservices are reported:

- **exam-scheduler:** A platform to handle exam registration.
- **student-portal:** Where students can register for the exam call.
- **professor-tools:** Where professors can get the list of registered students.

In the following table are summarized the key aspects of this application:

Source → Destination	Method	Access	Reason
student-portal → exam-scheduler	POST	✓	exam registration
student-portal → professor-tools	*	✗	access denied
professor-tools → exam-scheduler	GET	✓	read registered students
* → student-portal	*	✗	deny all to prevent personal data scraping

Now, for each microservice, you have to create a yaml manifest that contains the Deployment that runs the container, the ServiceAccount to identify the service within the mesh and the Service to expose the application within the cluster. Create a file named `exam-scheduler.yaml` with the following content:

`exam-scheduler.yaml`

```
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
```

```

metadata:
  name: exam-scheduler-sa
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: exam-scheduler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: exam-scheduler
  template:
    metadata:
      labels:
        app: exam-scheduler
    spec:
      serviceAccountName: exam-scheduler-sa
      containers:
        - name: exam-scheduler
          image: alepisp/exam-scheduler:latest
          ports:
            - containerPort: 8080
# Service
apiVersion: v1
kind: Service
metadata:
  name: exam-scheduler
spec:
  selector:
    app: exam-scheduler

```

```
ports:
  - port: 80
    targetPort: 8080
```

Create a file named `professor-tools.yaml` with the following content:

#### professor-tools.yaml

```
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
metadata:
  name: professor-tools-sa
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: professor-tools
spec:
  replicas: 1
  selector:
    matchLabels:
      app: professor-tools
  template:
    metadata:
      labels:
        app: professor-tools
    spec:
      serviceAccountName: professor-tools-sa
      containers:
        - name: professor-tools
          image: curlimages/curl
```

```

        command: [ "sleep", "3600" ]
# Service
apiVersion: v1
kind: Service
metadata:
  name: professor-tools
spec:
  selector:
    app: professor-tools
  ports:
    - port: 80
      targetPort: 8080

```

Create a file named `student-portal.yaml` with the following content:

```

student-portal.yaml
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
metadata:
  name: student-portal-sa
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: student-portal
spec:
  replicas: 1
  selector:
    matchLabels:
      app: student-portal

```



```

template:
  metadata:
    labels:
      app: student-portal
  spec:
    serviceAccountName: student-portal-sa
    containers:
    - name: student-portal
      image: curlimages/curl
      command: [ "sleep", "3600" ]
# Service
apiVersion: v1
kind: Service
metadata:
  name: student-portal
spec:
  selector:
    app: student-portal
  ports:
  - port: 80
    targetPort: 8080

```

Now run these commands to apply the manifests:

#### Apply Resources

```

$ kubectl apply -f exam-scheduler.yaml
$ kubectl apply -f professor-tools.yaml
$ kubectl apply -f student-portal.yaml

```

After applying these files, you can verify that all Pods are running correctly with this command, this could require around one minute.

### Check Pods

```
$ kubectl get pods
```

Once the pods are running you can start sending all requests you want. To do so, you have to open, for example, the student-portal terminal with:

### Open Terminal

```
$ kubectl exec -it deploy/student-portal -c student-portal \
-- sh
```

After that, you are in the student-portal shell and there you can start to POST an exam registration to the exam-scheduler by running:

### Exam Registration

```
# Command POST an exam registration
$ curl -X POST exam-scheduler -H \
  "Content-Type:application/json" -d '{"matricola": "s123456", \
  "exam": "NCS"}'
# if you want to exit student-portal shell
$ exit
```

If everything works, you should see something like the following image:

```
crownlabs@instance-j5x1h:~$ kubectl exec -it deploy/student-portal -c student-portal -- sh
~ $ curl -X POST exam-scheduler -H "Content-Type:application/json" -d '{"mat
ricola": "s123456","exam": "NCS"}'
✅ Exam received: {'matricola': 's123456', 'exam': 'NCS'}~ $ exit
```

Figure 2: POST Succeeded

At this point you didn't establish any authorizationPolicy or Authentication method like TLS. As well as this, if you try to run a pod that doesn't belong to the Istio Service Mesh, you can still send requests without problems. Run this command:

### Temporary pod

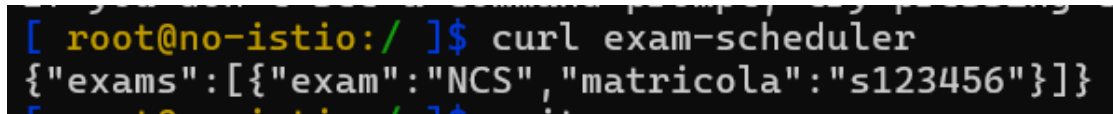
```
$ kubectl run no-istio \  
  --image=radial/busyboxplus:curl \  
  -it --rm --restart=Never \  
  --labels='sidecar.istio.io/inject=false' \  
  --command -- sh
```

This command starts a temporary Kubernetes pod named `no-istio` using the `radial/busyboxplus:curl`. The `-it` flag starts a shell, where the user can run commands inside the container. The `--rm` flag ensures that the pod is deleted once the session ends and `--restart=Never` prevents Kubernetes from restarting it. The label `sidecar.istio.io/inject=false` explicitly tells Istio to not inject the Envoy sidecar into this pod. If you run the following command inside the newly shell:

### Showing Pods

```
$ curl exam-scheduler
```

You should see the list of exams that are registered in the `exam-scheduler` like in the image:

A terminal window with a dark background. The prompt is `[ root@no-istio:/ ]$`. The command `curl exam-scheduler` has been executed. The output is a JSON object: `{"exams": [{"exam": "NCS", "matricola": "s123456"}]}`.

```
[ root@no-istio:/ ]$ curl exam-scheduler  
{  
  "exams": [  
    {  
      "exam": "NCS",  
      "matricola": "s123456"  
    }  
  ]  
}
```

Figure 3: GET Succeeded

However, this behavior reveals a critical issue: the request succeeds even though it was sent from a pod that is not part of the Istio service mesh. This means that there are no security mechanisms in place to restrict or authenticate incoming traffic.

It is now time to enable the security features discussed above.

### 4.2.1 Applying Istio mTLS

In order to use mTLS, you have to apply the yaml file described in Section: 4.1.1

#### Apply Authorization Policies

```
$ kubectl apply -f peer-authn.yaml
```

To prove that mTLS works properly, run again the command that creates a no-istio pod 4.2. Then, inside the shell, try to access the exam-scheduler service again using:

#### Showing Pods

```
$ curl exam-scheduler
```

You should see:

```
[ root@no-istio:/ ]$ curl exam-scheduler
curl: (56) Recv failure: Connection reset by peer
```

Figure 4: GET Failed

This confirms that mTLS is working properly: it blocks any requests from clients do not have valid identity certificate issued by Istio.

At this point, try performing GET and POST requests to the exam-scheduler service both as a student and as a professor. You will notice that all requests succeed, even though not all of them should be allowed. This happens because no authorization policy has been defined yet.

### 4.2.2 Applying Istio Authorization Policies

In order to secure the communication of thr three microservices you can use `AuthorizationPolicy` resources. These policies define which services are allowed to interact, using which HTTP methods.

Create a file named `authz-scheduler.yaml` with the following content:

`authz-scheduler.yaml`

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-scheduler-access
  namespace: default
spec:
  selector:
    matchLabels:
      app: exam-scheduler
  rules:
  - from:
    - source:
        principals:
          ["cluster.local/ns/default/sa/student-portal-sa"]
      to:
      - operation:
          methods: ["POST"]
  - from:
    - source:
        principals:
          ["cluster.local/ns/default/sa/professor-tools-sa"]
      to:
      - operation:
          methods: ["GET"]
```

This policy allows only:

- `student-portal-sa` to make POST requests to `exam-scheduler`
- `professor-tools-sa` to make GET requests to `exam-scheduler`

All other requests are implicitly denied.

Create a file named `authz-professor.yaml` with the following content:

`authz-professor.yaml`

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-professor-inbound
  namespace: default
spec:
  selector:
    matchLabels:
      app: professor-tools
  rules: []
```

This policy blocks all the traffic to `professor-tools`, since no rules are defined.

Create a file named `authz-student.yaml` with the following content:

`authz-student.yaml`

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-student-inbound
  namespace: default
spec:
  selector:
    matchLabels:
      app: student-portal
  rules: []
```

This policy blocks all the traffic to `student-portal`, since no rules are defined.

Now run these commands to apply the policies.

### Apply Authorization Policies

```
$ kubectl apply -f authz-scheduler.yaml
$ kubectl apply -f authz-professor.yaml
$ kubectl apply -f authz-student.yaml
```

#### 4.2.3 Testing the Security Policies

Once the authorization policies are applied, you can now test the behavior of the service mesh by simulating interactions between the services.

Use the following command to open a shell in the `student-portal` Pod and send a POST request to the `exam-scheduler` service:

### POST from student-portal

```
$ kubectl exec -it deploy/student-portal -c student-portal
-- sh

# Inside the Pod:
$ curl -X POST exam-scheduler -H "Content-Type: \
application/json" -d '{"matricola": "s654321", "exam": "ISS"}'
```

If the policies are correctly applied, you should see something like this image:

```
~ $ curl -X POST exam-scheduler -H "Content-Type: application/json" -d '{"matricola": "s654321", "exam": "ISS"}'
✅ Exam received: {'matricola': 's654321', 'exam': 'ISS'}~ $ exit
```

Figure 5: POST Succeeded

**Note:** Any other method such as a GET from the `student-portal` should fail like:

```
crownlabs@instance-6tflb:~$ kubectl exec -it deploy/student-portal -c student-portal -- sh
~ $ curl http://exam-scheduler
RBAC: access denied~ $ |
```

Figure 6: GET Failed

Now you can make request from `professor-tools` to `exam-scheduler`. Open a shell in the `professor-tools` Pod and send a GET request:

```
GET from professor-tools

$ kubectl exec -it deploy/professor-tools -c professor-tools
-- sh
# Inside the Pod:
$ curl http://exam-scheduler
```

The response should include the list of exams previously submitted, like this image:

```
crownlabs@instance-dvnbh:~$ kubectl exec -it deploy/professor-tools -c professor-tools -- sh
~ $ curl http://exam-scheduler
{"exams":[{"exam":"NCS","matricola":"s123456"}, {"exam":"ISS","matricola":"s654321"}]}
```

Figure 7: GET Succeeded

**Note:** Any other method such as a POST from `professor-tools` should be denied. In this Section, you have experienced how easily a Service Mesh like Istio can achieve Confidentiality, Authentication and Authorization.



## 5 Linkerd

Linkerd is a service mesh for Kubernetes. It makes running services easier and safer by giving you runtime debugging, observability, reliability and security.

Linkerd has two basic components: a control plane and a data plane. Once Linkerd's control plane has been installed on your Kubernetes cluster, you add the data plane to your workloads and the Service mesh is active [4].

Here are reported the main security features:

- **HTTP, HTTP/2, and gRPC Proxying:** Linkerd will automatically enable advanced features (including metrics, load balancing, retries, and more) for HTTP, HTTP/2, and gRPC connections.
- **Retries and Timeouts:** Linkerd can retry and timeout HTTP and gRPC requests.
- **Automatic mTLS:** Linkerd automatically enables mutual Transport Layer Security (TLS) for all communication between meshed applications.
- **Authorization Policy:** Linkerd can restrict which types of traffic are allowed between meshed services.
- **Service Profiles:** Linkerd's service profiles enable per-route metrics as well as retries and timeouts.

For a more detailed description of the features it offers, visit <https://linkerd.io/2.18/features/>.

### 5.1 Linkerd Setup

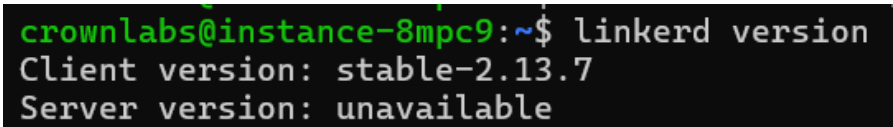
In this section you will set up the Linkerd service mesh to experiment and test the available security features. The Linkerd version proposed for this lab activity is v2.13 available at <https://linkerd.io/2.13/overview/index.html>.

Now, in order to set up Linkerd run the following commands:

### Linkerd Setup

```
$ curl -sL \
https://github.com/linkerd/linkerd2/releases/download \
/stable-2.13.7/linkerd2-cli-stable-2.13.7-linux-amd64 -o
linkerd
$ chmod +x linkerd
$ sudo mv linkerd /usr/local/bin
$ linkerd version
```

After the last command, you should see something like the following image.



```
crownlabs@instance-8mpc9:~$ linkerd version
Client version: stable-2.13.7
Server version: unavailable
```

Figure 8: Linkerd Version

Don't worry if you see *Server version: unavailable*, once you install Linkerd inside the cluster it will be fixed.

With:

### Linkerd Setup

```
$ linkerd check --pre
```

you should see something like *Status check results are ✓*. If so, everything is fine and you can continue. After that you can proceed with the installation of Linkerd inside the cluster:

### Linkerd Setup

```
$ linkerd install --crds | kubectl apply -f -
$ linkerd install --set proxyInit.runAsRoot=true | kubectl \
apply -f -
```

To verify that everything is working properly run:

#### Linkerd Setup

```
$ linkerd check
```

After that you can proceed with the installation of Viz extension:

#### Linkerd Setup

```
$ linkerd viz install | kubectl apply -f -
```

To verify that everything is working properly run:

#### Linkerd Setup

```
$ linkerd viz check
```

## 5.2 Securing micro-services with Linkerd

In this part of the lab you will deploy two pods (`aura-api` and `web-frontend`), attach them to the Linkerd service mesh, and experiment with three resilience & security mechanisms already available in version **2.13.7**:

- **Budgeted retries:** Protect the backend while improving success-rate.
- **Failure-accrual circuit breaking:** Automatically eject unhealthy pods from load-balancing.
- **Automatic mTLS:** Encrypt & authenticate all traffic inside the mesh with zero configuration.

### 5.2.1 Deploying the workloads

First, you have to create the three manifests. Here, is reported the code for each of them:

## aura-api.yaml

```
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aura-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: aura-api
  template:
    metadata:
      labels:
        app: aura-api
    spec:
      containers:
      - name: aura-api
        image: servicemeshnics/aura-api:latest
        ports:
        - containerPort: 8080
        env:
        - name: ERROR_RATE
          value: "0.4"
        - name: MIN_DELAY_S
          value: "0.2"
        - name: MAX_DELAY_S
          value: "3.0"

# Service
apiVersion: v1
kind: Service
```

```
metadata:
  name: aura-api
spec:
  selector:
    app: aura-api
  ports:
    - port: 80
      targetPort: 8080
```

## web-frontend.yaml

```
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web-frontend
  template:
    metadata:
      labels:
        app: web-frontend
    spec:
      containers:
        - name: curl
          image: curlimages/curl
          command: ["sleep", "3600"]

# Service
apiVersion: v1
kind: Service
metadata:
  name: web-frontend
spec:
  selector:
    app: web-frontend
  ports:
    - port: 80
      targetPort: 8080
```

#### aura-profile.yaml

```
# Service Profile
apiVersion: linkerd.io/v1alpha2
kind: ServiceProfile
metadata:
  name: aura-api.default.svc.cluster.local
  namespace: default
spec:
  routes:
    - name: GET /aura
      condition:
        method: GET
        pathRegex: "^/aura$"
        isRetryable: true
    - name: GET /slow
      condition:
        method: GET
        pathRegex: "^/slow$"
        isRetryable: true
      condition:
        method: GET
        pathRegex: "^/chaos$"
        isRetryable: false
  retryBudget:
    retryRatio: 0.2
    minRetriesPerSecond: 10
    ttl: 10s
```

The key aspects of the Service Profile manifest are:

- **isRetryable:** tells Linkerd to retry eligible failures for this route; non-retryable routes will propagate the error immediately.

- **retryBudget:** defines the *maximum* volume of retries across all retryable routes. This prevents retries from a denial-of-service if the backend is down.

Now, as always, you have to apply the resources with:

#### Apply resources

```
$ linkerd inject aura-api.yaml | kubectl apply -f -
$ linkerd inject web-frontend.yaml | kubectl apply -f -
$ kubectl apply -f aura-profile.yaml

# Verify all pods are injected & Running
$ kubectl get pods
```

You should see each pod with 2/2 containers (*app + Linkerd proxy*) like in the following image:

```
crownlabs@instance-6ssbk:~$ kubectl get pods
[NAME                                READY   STATUS    RESTARTS   AGE
aura-api-7bddf4cb6b-5twv8            2/2     Running   0           106s
aura-api-7bddf4cb6b-fjg5w            2/2     Running   0           106s
aura-api-7bddf4cb6b-p9cz7            2/2     Running   0           107s
web-frontend-5d6db568d-8275w         2/2     Running   0           34s
```

Figure 9: Running Pods

Unlike Istio, Linkerd supports by default mTLS protocol. In order to check this feature run:

#### mTLS Feature

```
$ linkerd viz edges deploy -n default
```

You should see something like the following image:



```

crownlabs@instance-6ssbk:~$ linkerd viz edges deploy -n default
SRC          DST          SRC_NS      DST_NS      SECURED
[prometheus  aura-api      linkerd-viz default      ✓
prometheus   web-frontend linkerd-viz default      ✓

```

Figure 10: Active mTLS Linkerd

**Note:** The SECURED flag in viz edges indicates encrypted communication between proxies via mTLS. However, Linkerd 2.13 does not block unauthenticated traffic from non-meshed workloads unless you use external mechanisms.

### 5.2.2 Exercise A - Retry budget

In this part, you will use two different endpoints to see how Linkerd behaves:

- **/aura:** happy path, 40 % random 500 errors
- **/slow:** always success but 2–4s latency

#### Part A.1–Normal traffic

First, open the web-frontend shell and generate normal traffic with:

```

web-frontend shell
$ kubectl exec -it deploy/web-frontend -c curl -- sh
$ while true; do curl -s http://aura-api/aura; sleep 1; done

```

While the last command is running, open another terminal and run first the tap command, then the routes, while the web-frontend is generating traffic:

### Route stats (aura)

```
$ linkerd viz tap deploy/web-frontend --to svc/aura-api \
-n default

$ linkerd viz routes deploy/web-frontend --to svc/aura-api \
-n default -o wide
```

You can observe the outputs and notice that:

- The **latency** column (e.g. 300–3000ms) matches the two random delays configured in the aura-api.yaml manifest: `MIN_DELAY_S=0.2`, `MAX_DELAY_S=3.0`.
- A `status=500` followed by a new request that ends with code 200 occurs because: `ERROR_RATE=0.4` makes the app fail ~40% of the time and the ServiceProfile marks /aura as retryable. Linkerd retries once on another pod, usually succeeding.
- `EFFECTIVE_SUCCESS = 100%` but `ACTUAL_SUCCESS ≈ 60%`: Effective counts only the first try, which ends OK thanks to the retry, while Actual includes the failed attempt too.

Now, try to increase the error rate on-the-fly to drain the budget:

### Drain the retry budget

```
$ kubectl set env deploy/aura-api ERROR_RATE=0.9
```

After a few seconds, the output of the `linkerd viz routes` command reveals an increase in `ACTUAL_RPS` (actual requests per second), while `ACTUAL_SUCCESS` drops significantly, often to values below 20%. This behavior indicates that the service is failing frequently and Linkerd is attempting retries until the retry budget is exhausted.

You can also observe this behavior in real time with the tap command explained before.

Initially, you'll see sequences of `status=500` followed by `status=200`, meaning Linkerd is retrying successfully. When the budget is over you will see only 500 responses.

This mechanism protects the backend from being overwhelmed, demonstrating how Linkerd manages resilience with a controlled retry budget.

## Part A.2—Slow traffic

Start generate traffic in the `/slow` route, by opening the web-frontend shell and running inside it:

### Continuous `/slow` calls

```
$ kubectl exec -it deploy/web-frontend -c curl -- sh
$ while true; do curl -s http://aura-api/slow; sleep 1; done
```

Observe with `tap` that the latency column now shows values around 2s-4s; no 504 will appear because Linkerd 2.13 has no per-route timeout yet. If you were using Linkerd v2.14 you would see the timeout effect: in the serviceProfile you could set timeouts: 1s and in the tap command output you would see 504 errors. The timeout feature isn't yet available in this version of Linkerd, as a result, the `/slow` route will have always `status=200` OK, with latency around 2–4 seconds, but never interrupted by proxy. You do not see any difference between ACTUAL e EFFECTIVE success-rate: both will remain at 100% because there are not errors or retry. This occurs because in the ServiceProfile you set `isRetryable: true`, but Linkerd will retry only on 5xx errors. In this case the backend responds always 200, no retry is counted and budget stays full, in fact `ACTUAL_RPS` is pretty small.

### 5.2.3 Exercise B – DoS route Breaking

The goal of this part is to understand how Linkerd eject a pod that always fails. First delete the Service Profile:

### Disable ServiceProfile

```
$ kubectl delete -f aura-profile.yaml
```

Linkerd disables the failure accrual mechanism when a **ServiceProfile** is present. In order to activate the circuit breaking mechanism, via annotations, the ServiceProfile must be removed first.

Then, enable circuit breaking by annotating the Service:

### Service Annotating

```
$ kubectl annotate svc aura-api \
balancer.linkerd.io/failure-accrual=consecutive \
balancer.linkerd.io/failure-accrual-consecutive-max-failures=4\
balancer.linkerd.io/failure-accrual-consecutive-min-penalty=30s
```

Scale to one replica and restart:

### Scale to one replica

```
$ kubectl scale deploy aura-api --replicas=1
$ kubectl rollout restart deploy aura-api
```

Please wait at least one minute before making the next steps. You will execute a GET request to the chaos route to obtain 500 responses.

### Continuous /chaos calls

```
$ kubectl exec -it deploy/web-frontend -c curl -- sh
$ while true; do curl -s http://aura-api/chaos; sleep 1; done
```

Open a new terminal and stream the outbound requests:

### Tap the traffic

```
$ linkerd viz tap deploy/web-frontend --to svc/aura-api \
  -n default | grep 'dst='
```

You should see that after 4 consecutive 500 errors the proxy ejects the endpoint. As a result, the `tap` stops showing `dst=...`

Passed 30 s (value set with `-min-penalty=30s`), the proxy **will restart** the "banned" endpoint following this logic:

1. **Forwarding just one request**

2. **Evaluating the outcome:**

- **Success:** Any status < 500 (200, 204, 404...) is considered good. So, the endpoint will be *reinserted* in the pool of load-balancing and the traffic will return flooding.
- **Still error:** Status 5xx. So, the proxy expels it again and doubles the penalty according to a *exponential back-off* :

$$30\text{ s} \rightarrow 60\text{ s} \rightarrow 120\text{ s} \rightarrow \dots (\text{max } 5\text{ min})$$

This mechanism ensures that:

- the failed pod does not receive traffic until it actually becomes healthy again;
- the proxy does not queue requests or overload still-functioning pods;
- "re-admission" occurs in a gradual and controlled manner.

#### 5.2.4 Final Considerations

Even on an older **Linkerd 2.13.7** you already get:

- Resilience with budgeted retries and circuit breaking
- Confidentiality & authentication with automatic mTLS

- Real-time observability via `tap`, `top`, and `stat`

With **Linkerd**  $\geq$  **2.14** you could additionally:

- Enforce per-route timeout (`auto-504`).
- Apply `ServerPolicy` / `ClientPolicy` resources for fine-grained zero-trust authorisation.
- Inject HTTP fault scenarios (`faultInjection`) for chaos engineering.

Overall, you have learned how a service mesh lets you introduce strong security controls and operational safeguards outside the application code, enabling strong isolation and resilience even when the applications themselves are fragile.

## References

- [1] Kubernetes Authors. *What is Kubernetes?* Available at: <https://kubernetes.io/>.
- [2] Cilium Authors. *What is Cilium?* Available at: <https://docs.cilium.io/en/stable/overview/intro/#what-is-cilium>.
- [3] Istio Authors. *What is Cilium?* Available at: <https://istio.io/latest/docs/overview/what-is-istio/>.
- [4] Linkerd Authors. *What is Linkerd?* Available at: <https://linkerd.io/2.18/overview/>.