Politecnico di Torino

Kubernetes Network Policies and Service Meshes

Alessandro Genova
s330893

August 2024

Contents

1 Introduction

This lab provides aims at practicing with two essential technologies for improving network security and service communication in Kubernetes: Network Policies and Service Meshes. Network Policies control pod-to-pod communication by acting as a firewall, while Service Meshes, such as Istio, manage service-to-service communication, offering features like traffic management, security and observability.

By the end of this lab, students will:

- Implement and test Kubernetes Network Policies to control pod communication.
- Deploy and configure Istio as a Service Mesh.
- Apply traffic management using Istio's Virtual Services and Destination Rules.
- Implement mutual TLS (mTLS) between services.
- Use observability tools like Kiali, Grafana, and Jaeger in Istio.
- Understand the different use cases for Network Policies and Service Meshes.

2 Lab Environment Setup

2.1 Kubernetes Cluster Setup

Let's start by setting up the necessary tools and environment.

Steps:

1. **Install Docker**, instructions at: `https://docs.docker.com/engine/install/ubuntu/`.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.\
    asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https\
      ://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
# Install docker
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker\
    -compose-plugin

# Add to group to use docker as normal user
sudo usermod -aG docker $USER && newgrp docker

# Verify docker works
docker run hello-world
```

2. **Install Minikube + Cilium**, instructions at:

   - `https://minikube.sigs.k8s.io/docs/start/?arch=%2Flinux%2Fx86-64%2Fstable%2Fbinary+download`
   - `https://docs.cilium.io/en/stable/gettingstarted/k8s-install-default/`
   - `https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-using-native-package-management`

   For this lab, we'll be using Minikube to set up our Kubernetes cluster. Minikube was chosen for several reasons:

   - **Simplicity and Ease of Use**: Minikube is designed to be easy to install and configure, making it ideal for who is new to Kubernetes or for testing and developing applications in a local environment.
   - **Minimal Resources**: Unlike other solutions that require multiple nodes and significant resources, Minikube can run on a single machine with minimal resource requirements (requires only 2GB of free RAM). This makes it perfect for small projects.

   In this lab, for simplicity, we will have only one control node and only one worker node.

   Important Note: It's not sufficient to just install Minikube. To enable Network Policies in your Kubernetes cluster, you need to initialize the cluster with a CNI (Container Network Interface) that supports Network Policies. To enable Network Policies in your Kubernetes cluster, you need to initialize the cluster with a CNI (Container Network Interface) that supports Network Policies. A CNI plugin configures network interfaces for containers and sets up network bridges (which link network segments and manage traffic) on the host. By default, Kubernetes does not support Network Policies unless a CNI is installed.

   After some testing, I found out that **Cilium** is the CNI that works better in this case (also according to Cloud Computing Polito labs, Cilium is the recommended choice).

You can use the default Kubernetes NetworkPolicy resources with Cilium, or optionally take advantage of the more advanced policies specific for Cilium. We'll concentrate on default Kubernetes ones.

Follow the instructions below to set up Minikube with Cilium as the CNI:

```
sudo apt update && sudo apt install -y curl
# Install minikube
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64

# If you haven't done it before, add to group to use docker as normal user
sudo usermod -aG docker $USER && newgrp docker

minikube config set driver docker # could be useful to set this, because if we forget to \
    manually specify the driver, docker will be automatically considered

# Start Minikube with Cilium CNI:
minikube delete && minikube start --driver=docker --cni=cilium

# Install kubectl if not already installed:
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl gnupg && sudo mkdir -p /\
    etc/apt/keyrings/
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -\
    o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
sudo chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg # allow unprivileged APT \
    programs to read this keyring
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/\
    core:/stable:/v1.31/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list
sudo chmod 644 /etc/apt/sources.list.d/kubernetes.list # helps tools such as command-not-\
    found to work correctly
sudo apt-get update
sudo apt-get install -y kubectl
```

3. **Verify Internet Connectivity:** After starting Minikube, verify that the cluster has internet connectivity by running the following commands:

```
minikube ssh
ping google.com
```

If the ping fails, modify the DNS settings with these steps:

```
sudo vi -c "set nocompatible" /etc/resolv.conf # set nocompatible is to make vi similar \
    to vim
```

In the 'vi' editor:

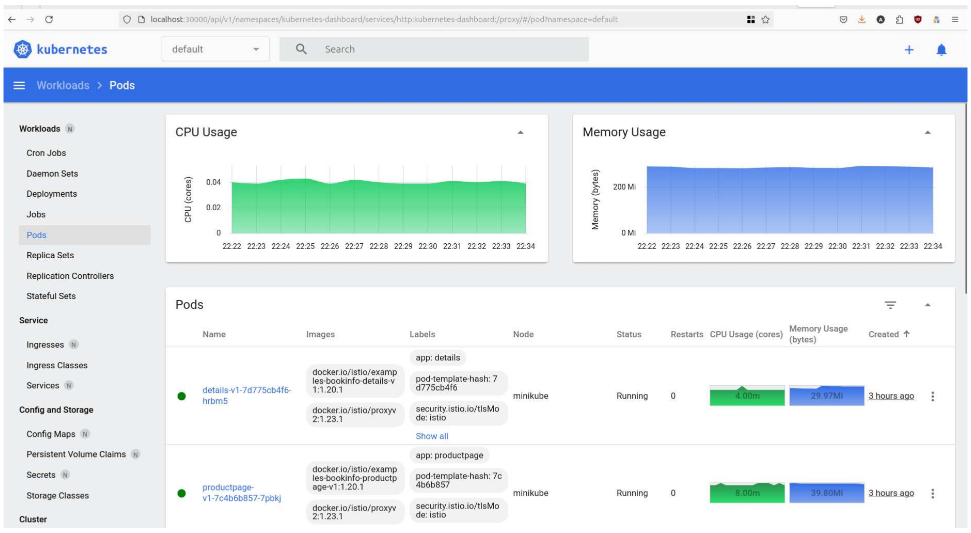
- Press 'i' to enter insert mode.
- Locate the 'nameserver' line, delete the existing IP, and replace it with '8.8.8.8'.
- Press 'Esc', then type ':wq' and press 'Enter' to save and exit.

Try pinging again. If successful, exit the Minikube node by typing 'exit' or by pressing 'Ctrl+D'.

4. **Minikube Dashboard:** To open the Minikube dashboard in your browser, use the following command:

```
minikube dashboard &     # where '&' is to open in background and be able to continue to \
    use the same terminal
```

Wait some time and it will output the url to reach we the web dashboard. The dashboard provides a graphical interface to monitor and manage your Kubernetes resources. However, for learning purposes, we recommend using terminal commands instead of relying on the dashboard GUI.



Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
● details-v1-7d775cb4f6-hrbm5	docker.io/istio/examples-bookinfo-details-v1:1.20.1 docker.io/istio/proxyv2:1.23.1	app: details pod-template-hash: 7d775cb4f6 security.istio.io/tlsMode: istio Show all	minikube	Running	0	4.00m	29.97Mi	3 hours ago
● productpage-v1-7c4b6b857-7pbkj	docker.io/istio/examples-bookinfo-productpage-v1:1.20.1 docker.io/istio/proxyv2:1.23.1	app: productpage pod-template-hash: 7c4b6b857 security.istio.io/tlsMode: istio	minikube	Running	0	8.00m	39.80Mi	3 hours ago

Figure 1: Kubernetes Web Dashboard

To also view CPU and memory usage of your pods (like in the image), you can enable the metrics-server addon. This addon is also necessary to display these metrics via the terminal with ‘kubectl top pods‘:

```
minikube addons enable metrics-server
```

2.2 Remote Access and SSH Tunneling with CrownLabs

If you are using the CrownLabs service offered by Politecnico di Torino, follow these instructions to access dashboards or applications, running on the remote machine.

Assuming you have connected to the CrownLabs machine using the following SSH command, where `CROWNLABS_REMOTE_IP` is the IP address of the CrownLabs machine you have access:

```
ssh -J bastion@ssh.crownlabs.polito.it crownlabs@CROWNLABS_REMOTE_IP
```

Setting Up an SSH Tunnel:

To access web-based services on a remote machine, you can establish an SSH tunnel that forwards traffic from a local port to a service on the CrownLabs machine. Use the following command:

```
  ssh -J bastion@ssh.crownlabs.polito.it crownlabs@CROWNLABS_REMOTE_IP \
     -L LOCAL_PORT:LOCAL_CROWNLABS_MACHINE_ADDRESS:REMOTE_PORT -N &
```

- `LOCAL_PORT`: The port on your local machine that you wish to use.
- `LOCAL_CROWNLABS_MACHINE_ADDRESS`: The local address of the CrownLabs machine (commonly `localhost`, or a cluster IP for services exposed via the cluster).
- `REMOTE_PORT`: The port on the CrownLabs machine where the service is running.

Note:

- The `-L` option specifies the local port forwarding configuration. It maps a port on your local machine to a remote port on the CrownLabs machine, allowing you to access the remote service as though it were local.
- The `-N` option prevents SSH from executing commands on the remote machine, while `&` runs the tunnel process in the background, enabling continued use of the terminal.

Examples:

- To access the Minikube dashboard (assuming it is exposed on port 53522 on the CrownLabs machine) and forward it to local port 1337, run:

  ```
  ssh -J bastion@ssh.crownlabs.polito.it crownlabs@CROWNLABS_REMOTE_IP -L 1337:localhost\
      :53522 -N &
  ```

  You can then access the dashboard via your browser at `http://localhost:1337/URLOFDASHBOARD` (where `localhost` refers to your local machine).

- To access a service running on IP `10.244.0.94` on port `80` and forward it to local port `9000`, use:

  ```
  ssh -J bastion@ssh.crownlabs.polito.it crownlabs@CROWNLABS_REMOTE_IP -L \
      9000:10.244.0.94:80 -N &
  ```

  You can access the remote service by navigating to `localhost:9000`.

2.3 Sample Application Deployment

Deploy a sample application to test network policies and service mesh features. For a tutorial, visit: `https://kubernetes.io/docs/tutorials/hello-minikube/`

Steps:

1. **Create a Deployment:**

   ```
   kubectl create deployment hello-node \
   --image=registry.k8s.io/e2e-test-images/agnhost:2.39 -- /agnhost netexec --http-port=8080
   ```

   This command creates a deployment named ‘hello-node‘ using the specified image and configuration.

2. **Monitor the Deployment:** Use the following commands to check the status of your deployment:

   ```
   kubectl get pods
   kubectl get deployments
   kubectl get service
   kubectl get events
   kubectl config view # Not very meaningful for us
   kubectl logs hello-node-5f76cf6ccf-br9b5 # Use the pod name from 'kubectl get pods'
   ```

   These commands will operate in the 'default' namespace as no specific namespace is provided.

3. **Understand Key Concepts:**

   - **Pod**: The basic unit in Kubernetes, representing a single instance of a running process. Pods are managed by controllers such as Deployments.
   - **Deployment**: Manages Pods, ensuring the desired number of replicas are running and handling rolling updates with zero downtime. The pod information (image, ports, volumes, etc.) is typically defined here.
   - **Service**: Provides a single IP address and DNS name to access a set of Pods, balancing traffic (LoadBalancer) and exposing applications externally.
   - **Events**: Show the state of resources in the cluster, useful for debugging (look for 'Warnings').
   - **Config View**: Displays the current Kubernetes configuration, including cluster, user, and context details.
   - **Logs**: Displays logs from a specific Pod, useful for troubleshooting and monitoring. Containers write logs to stdout and stderr. The container runtime (e.g., Docker, containerd) collects these logs and stores them in log files on the Kubernetes worker node's filesystem. You can access these logs using kubectl logs, which retrieves them from the worker node where the Pod is running.

4. **Expose the Deployment as a Service:** Create a service to expose the deployment:

```
kubectl expose deployment hello-node --type=LoadBalancer --port=8080
```

This command creates a Service named ‘hello-node‘ (on an internal port 8080, but that’s not important).

5. **Find the address and access the service:** To access the service externally:

```
minikube service hello-node # The URL column shows external IP and external port. It also\
     opens the service in browser if you have a GUI.
# Other ways to find the external IP (without port) that represent our cluster in our \
    machine are the commands 'minikube ip' or 'kubectl cluster-info'
```

So we obtained the EXTERNAL cluster IP and the exposed port given by the LoadBalancer.

To find the INTERNAL IP, you can use:

```
kubectl get svc -n default
```

or check the Service page in the web dashboard. (Note that the internal port is 8080 as configured in `kubectl expose deployment` command). Actually, it’s not so important to know the IP, because we can just reach the service typing the name of it with its port (in our case, ’http://hello-node:8080’), as Kubernetes resolves it automatically.

If needed, you can forward a specific local port:

```
kubectl port-forward service/hello-node 7080:8080 &
```

This allows you to reach the service at ‘http://localhost:7080‘.

3 Part 1: Kubernetes Network Policies

3.1 Overview

Network Policies are Kubernetes resources designed to manage the flow of network traffic between pods, acting like a firewall to define rules for ingress and egress traffic, operating at the IP address or port level (OSI layers 3 and 4). Like we said some page before, Network Policies require a network plugin that supports them: in our case, it's Cilium.

For further informations and resources about Network Policies, I recommended to check out: `https://kubernetes.io/docs/concepts/services-networking/network-policies/`.

Key Features:

- **Traffic Control**: Specify rules for TCP and UDP protocols.
- **Entity Communication Rules**: Define which pods, namespaces, and IP blocks can communicate with each other.
- **Policy Application**: Use selectors to apply policies to specific pods or namespaces, controlling connections and interactions.

3.2 Default Behavior

By default, Kubernetes allows all pod-to-pod communication. To verify this, follow these steps:

1. **Run a Test Pod:** Execute the following command to run a test pod. This pod will be created in the default namespace:

   ```
   kubectl run test-pod --rm -it --image=alpine -- sh
   ```

2. **Test Connectivity:** Once inside the test pod, use the following command to check if you can access the ‘hello-node‘ service:

   ```
   wget -qO- http://hello-node:8080
   ```

   In this command:

   - ‘hello-node‘ correctly refers to our service. It's very cool because it avoids us to deal with IPs. Kubernetes resolves the service name to the appropriate IP address automatically.
   - Alternatively, you can use the IP and port you found previously (either the externalIP:externalPort or internalIP:internalPort).

3. **Expected Outcome:** You should be able to access the ‘hello-node‘ service from within the test pod, confirming that pod-to-pod communication is working as expected.

3.3 Exercise 1: Basic Ingress Deny-All

In this exercise, you will create a Network Policy that denies all ingress traffic to pods in a specific namespace.

Steps:

1. **Create a New Namespace:** Create a new namespace for this exercise:

   ```
   kubectl create namespace network-policy-demo
   ```

2. **Define the Network Policy:** Create a file named ‘`deny-all-ingress.yaml`‘ (it's in ’`yamls`’ folder, take it from there, Latex does not respect indentation) with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: network-policy-demo
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

In this configuration:

- ‘apiVersion‘ specifies the version of the NetworkPolicy resource, which can vary in features and syntax.
- An empty ‘podSelector‘ means the policy applies to all pods in the ‘network-policy-demo‘ namespace.
- The ‘policyTypes‘ field is set to ‘Ingress‘, which implicitly denies all incoming traffic since no specific ingress rules are defined.

3. **Apply the Network Policy:** Apply the Network Policy with the following command:

```
kubectl apply -f deny-all-ingress.yaml
```

4. **Test the Policy:** Create two pods in the namespace:

```
kubectl run pod1 --image=nginx --namespace=network-policy-demo
kubectl run pod2 --image=busybox --namespace=network-policy-demo --rm -it -- /bin/sh
```

5. **Access Test:** From ‘pod2‘, try to access ‘pod1‘:

```
wget -O- --timeout=5 http://POD1_IP # you have to put the pod1 IP
```

Note: Since ‘pod1‘ is NOT a service, you can’t use the name but we must use the IP address. Find ’pod1’ IP via:

```
kubectl describe pod pod1 --namespace=network-policy-demo # in ’IP:’ line
# Or
kubectl get pod --namespace=network-policy-demo -o wide
```

or check the IP in the web dashboard under the Pod section of the ‘network-policy-demo‘ namespace.

Anyway, the request should timeout, indicating that the ingress traffic is blocked.

6. **Verify Policy Removal:** To verify that the policy is correctly blocking traffic, delete the Network Policy and retry the access test:

```
kubectl delete -f deny-all-ingress.yaml
kubectl run pod2 --image=busybox --namespace=network-policy-demo --rm -it -- /bin/sh
```

Running ‘wget‘ again should succeed, confirming that the ingress traffic is no longer blocked.

3.4 Exercise 2: Allowing Ingress Traffic from Specific Pods

In this exercise, you’ll modify the previous Network Policy to allow traffic from pods with a specific label.

Steps:

1. **Define the Network Policy:** Create a file named ‘`allow-specific-ingress.yaml`‘ (it’s in ’`yamls`’ folder, so take it from there instead copying from here otherwise indentation will not be respected) with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-specific-ingress
  namespace: network-policy-demo
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: allowed
```

This time we are selecting pod with label 'app: web' and we are setting a specific ingress rule: we allow the traffic coming from pods with the specific label 'access: allowed'.

2. **Apply the new Network Policy:**

```
kubectl apply -f allow-specific-ingress.yaml
```

3. **Label pod1 with ‘app: web‘:**

```
kubectl label pod pod1 app=web -n network-policy-demo
```

4. **Create a new pod with the ‘access: allowed‘ label:**

```
kubectl run pod3 --image=busybox --namespace=network-policy-demo \
--labels="access=allowed" --rm -it -- /bin/sh
```

5. **Test Access from the New Pod:**
   From ‘pod3‘, try to access ‘pod1‘:

```
wget -O- --timeout=5 http://POD1_IP # you have to put the pod1 IP, like we did before
```

   This request should succeed because ‘pod3‘ has the required label to access ‘pod1‘. If we insted put '`access-denied`' (or anything that isn't '`access-allowed`') as labels when we run the pod3, request would timeout since we are not allowed to reach pod1.

6. **Remove the rule**, like we did for the previous rule we can do:

```
kubectl delete -f allow-specific-ingress.yaml
```

3.5 Exercise 3: Egress Policies

In this exercise, you'll create a Network Policy that restricts outgoing traffic from pods to specific IP ranges or DNS names.

Steps:

1. **Define the Network Policy:** Create a file named ‘`restrict-egress.yaml`‘ (you can find it in '`yamls`' folder) with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-egress
  namespace: network-policy-demo
```

```
spec:
  podSelector:
    matchLabels:
      app: testRestrictEgress
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 192.168.1.0/24
  - to:
    - namespaceSelector: {}
    ports:
    - protocol: UDP
      port: 53
```

This policy is for pods labeled with 'testRestrictEgress', and allows egress traffic only to the 192.168.1.0/24 IP range (which could be our host OS LAN subnet, but you can allow whatever you want) and to DNS servers (port 53). It blocks all other kind of traffic, including the traffic towards other pods of cluster, since 'namespaceSelector:{}' inside the 'egress' rule means that we don't permit to contact any pod of any namespace.

2. **Apply the egress policy:**

```
kubectl apply -f restrict-egress.yaml
```

3. **Create the new pod with the correct label:**

```
kubectl run pod4 --image=busybox --namespace=network-policy-demo \
--labels=app=testRestrictEgress --rm -it -- /bin/sh
```

4. **Test DNS resolution** from inside 'pod4':

```
nslookup google.com
```

This should succeed, as DNS traffic is allowed.

5. **Test access to external website** from 'pod4':

```
wget http://www.google.com
```

This should fail, as the egress traffic to external websites is blocked.

6. **Test access to other pods** from 'pod4':

```
wget http://POD1_IP   # replace like before with the pod1 ip
```

This should fail too, as the egress traffic to all other pods of all the namespaces is blocked by the 'namespaceSelector:{}' inside the 'egress' rule.

7. **But if you ping your Host OS** (assuming IP is inside 192.168.1.0/24 subnet):

```
ping 192.168.x.y # replace with your Host OS IP
```

This should work, as you have allowed that subnet.

8. **Remove the rule:**

```
kubectl delete -f restrict-egress.yaml
```

4 Part 2: Service Mesh with Bookinfo Application

4.1 Overview

A Service Mesh is a dedicated infrastructure layer for managing service-to-service communication in microservices architectures. It enhances Kubernetes' native capabilities for complex inter-service interactions. This section covers both the theory of Service Mesh and its practical implementation using Istio with the Bookinfo sample application.

Key Features:

- **Security**: Implements mutual TLS (mTLS) to secure communications without modifying application code.
- **Traffic Management**: Provides fine-grained traffic routing control.
- **Observability**: Offers insights into service interactions, including distributed tracing and metrics.
- **Resilience**: Supports fault tolerance through circuit breaking, timeouts, and retries.

Architecture:
More info here: `https://istio.io/latest/docs/ops/deployment/architecture/`
and here `https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy`

- **Sidecar Proxy**: Each microservice is paired with an Envoy sidecar proxy, which transparently intercepts all traffic to and from the containers. Envoy, operating at Layer 7, manages traffic routing, client-side load balancing (which is more effective and scalable than k8s one), circuit-breaking (to prevent application overload and handle failures more gracefully), and encryption (though encryption is limited to communication between sidecars, not between the sidecar and its container, as this would be too resource-intensive). This architecture enables advanced out-of-the-box observability, monitoring, and tracing without modifying application code.
- **Control Plane**: A centralized control plane, called istiod, is responsible for enforcing all policies within the service mesh and gathering telemetry data from each Envoy sidecar.

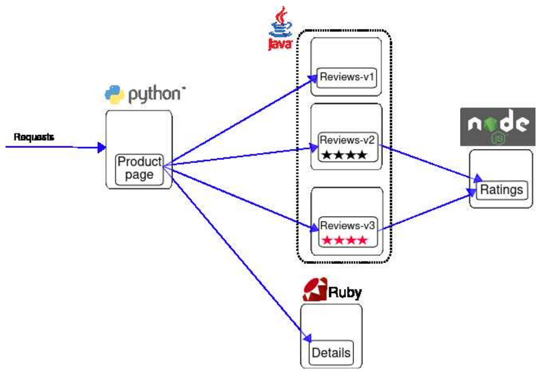
Considerations Though Istio offers numerous benefits, it introduces significant overhead. Simply enabling Istio can sometimes increase CPU usage by 20-30% (because of the added layer of complexity and Envoy being a user space (L7) application). This raises the question of whether the trade-off is always worth it.

4.2 Bookinfo Application

Bookinfo is a sample application that we will use to make practice with service mesh.

Bookinfo services:

- **productpage:** Fetches information from the details and reviews services, and displays it via a webpage.
- **details:** Provides book information.
- **reviews:** Contains book reviews, with three versions:
  - **v1:** Does not call the ratings service.
  - **v2:** Calls the ratings service and displays ratings as black stars.
  - **v3:** Calls the ratings service and displays ratings as red stars.
- **ratings:** Provides book ranking information.



4.3 Installing Istio

In this exercise, you'll install Istio in your Kubernetes cluster. Installation and configuration are from this installation guide: `https://istio.io/latest/docs/setup/getting-started/`.

Important Note: While Minikube can run with as little as 2GB of RAM, Istio is resource-intensive and typically requires at least 16GB of RAM for optimal performance. Due to these high resource requirements, you may experience unexpected behavior or performance issues when running Istio on a system with limited resources. Keep this in mind as you proceed with the exercises.

Steps:

1. **Install Istio CLI:**

```
curl -L https://istio.io/downloadIstio | sh -
export PATH="$PATH:/path/to/istio/bin/folder" # REPLACE the path with the path you \
    downloaded to, specifically the istio bin folder.
```

2. **Install Istio in the Cluster:**

```
istioctl install --set profile=demo -y # if something fails, re run the command
```

   This command installs Istio with the default configurations. Make sure that nothing fails, otherwise you have to rerun the command.

3. **Label the default namespace for Istio injection:**

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

   This command adds add a namespace label to instruct Istio to automatically inject Envoy sidecar proxies when we will deploy the various applications later.

4.4 Exercise 1: Deploying and Configuring the Bookinfo App

In this exercise, you will deploy the Bookinfo sample application to your Istio-enabled Kubernetes cluster and configure it for external access and traffic management. You can refer to the detailed guide at `https://istio.io/latest/docs/examples/bookinfo/`.

Steps:

1. **Deploy the Bookinfo application:**

```
cd /path/to/istio
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```

   This command deploys all four services and their respective versions used in the Bookinfo application.

2. **Verify that all services and pods are running:**

```
kubectl get services
watch kubectl get pods # Watch as the pods initialize (Ctrl+C to exit)
```

Ensure that all services (details, productpage, ratings, reviews) are listed and their respective pods are in the Running state. **Note**: It might take a few minutes for all the pods to be up!!

3. **Confirm the Application is Running:**

```
# Execute a command in the 'ratings' container of the pod with the label 'app=ratings'
kubectl exec "$(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}')" \
    -c ratings -- \
   curl -sS productpage:9080/productpage | grep -o "<title>.*</title>"

# Note: If copying from here, replace ‘ with normal quotes. It’s advisable to paste this \
    command into a text editor for any necessary adjustments.
```

This will perform a request starting from ratings pod to productpage pod. You should see output indicating that the application is running:

```
<title>Simple Bookstore App</title>
```

4. **Create a Gateway for external access:**

To inspect the Istio ingress gateway's configuration, use the following:

```
kubectl describe svc -n istio-system istio-ingressgateway
```

If you scroll down, you will see (among all the others) output very similar to this:

```
Port:                   http2  80/TCP
TargetPort:             8080/TCP
NodePort:               http2  30710/TCP
Endpoints:              10.244.0.160:8080
```

Here, the external port 80 (internally it's port 30710 in this case, it can vary, but this port is not important) is redirected to port 8080.

Now, we need to map port 8080 to the Bookinfo Gateway:
in `samples/bookinfo/networking/bookinfo-gateway.yaml`, under the 'Gateway' resource type, there is the configuration which defines the port used by the Bookinfo Gateway:

```
port:
  number: 8080
  name: http
```

In the same file, the ‘VirtualService‘ resource configuration specifies that traffic matching some URL paths should be routed to port 9080 on the ‘productpage‘ service (otherwise, it will be a 404 error):

```
http:
- match:
  - uri:
      exact: /productpage
  - uri:
      prefix: /static
  - uri:
      exact: /login

% ... etc etc

  route:
  - destination:
      host: productpage
      port:
        number: 9080
```

So finally, let's apply the rules:

```
kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

So to be clear: this command will configure the Istio ingress gateway to handle external ingress traffic directed to port `80` and to route it internally to port `8080`. The traffic is then managed by the Bookinfo Gateway, which routes it to the `productpage` service on port `9080`.
The `samples/bookinfo/networking/bookinfo-gateway.yaml` configuration has 'Gateway' and 'VirtualService' rules which specify these settings of ports and URLs for routing.

5. **Expose the Ingress Gateway:**

```
minikube tunnel # You could also put '&' at the end to run in the background and keep \
    using the same terminal
```

This command will expose the Istio ingress gateway, assigning it an external IP. In a test lab environment like Minikube, this IP will be the internal IP of the Istio ingress gateway itself. In a cloud environment, it would be mapped to a real external load balancer.

6. **Determine the Ingress IP and Port:**

```
# Define the name and namespace of the Ingress service
export INGRESS_NAME=istio-ingressgateway
export INGRESS_NS=istio-system

# Retrieve the external IP address of the Ingress service
export INGRESS_HOST=$(kubectl -n "$INGRESS_NS" get service "$INGRESS_NAME" \
-o jsonpath='{.status.loadBalancer.ingress[0].ip}')
# Note: If copying from here, replace ' with normal quotes. It's best to paste this \
    command into a text editor for editing.

# Retrieve the port number for the HTTP2 service
export INGRESS_PORT=$(kubectl -n "$INGRESS_NS" get service "$INGRESS_NAME" \
-o jsonpath='{.spec.ports[?(@.name=="http2")].port}')
# Note: If copying from here, replace ' with normal quotes. It's best to paste this \
    command into a text editor for editing.

# Combine the IP address and port to form the gateway URL
export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT

# Display the full URL for the Ingress gateway
echo $GATEWAY_URL # This is the IP address and port of the Istio Ingress gateway pointing\
     to our Bookinfo application.

# List all services in the 'istio-system' namespace to verify the Ingress gateway details
kubectl get service -n istio-system # This will display all Istio services, including the\
     Ingress gateway with its IP and exposed ports.
```

This retrieves the IP and port of the ingress gateway for the Bookinfo application.

Note: if you are using a Crownlabs machine, to be able to access it via your local browser, you have to create an SSH tunnel with these IP and port, following the mini-guide of some page ago.

7. **Test external access to the Bookinfo application:**

```
curl -s "http://${GATEWAY_URL}/productpage" | grep -o "<title>.*</title>"
```

The expected output is:

```
<title>Simple Bookstore App</title>
```

You can also access the application from your browser at: `http://$GATEWAY_URL/productpage`.

4.5 Exercise 2: Traffic Management

In this exercise, you'll use Istio's Virtual Services and Destination Rules to implement traffic routing and load balancing.

If you want further information about Traffic Management in Istio,
you can check out: `https://istio.io/latest/docs/concepts/traffic-management/`.

Steps:

1. **Define available versions with Destination Rules:**

   Before you can use Istio to control the routing of different versions of the Bookinfo application, you need to define these versions using Destination Rules. Run the following command to create default destination rules for all Bookinfo services:

   ```
   kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml
   ```

   Allow a few moments for the Destination Rules to propagate. In Istio, `DestinationRules` manage traffic to specific versions of a service based on pod labels.

   For example, the part related to 'reviews' service:

   ```
   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
   ```

   Here, the 'subsets' field defines three versions (v1, v2, v3) of the 'reviews' service based on the labels 'version: v1', 'version: v2', and 'version: v3', respectively.

   These version labels are defined in the Kubernetes Deployment manifests:
   `samples/bookinfo/platform/kube/bookinfo.yaml` that we deployed at the beginning.

2. **Verify Destination Rules:** To inspect the Destination Rules that have been applied, use:

   ```
   kubectl describe destinationrules
   ```

   You can visit with browser the `http://$GATEWAY_URL/productpage` and if you refresh several times you'll see at the bottom that can appear different kind of reviews: no stars (v1), black stars (v2) and red stars (v3).

3. **Route all traffic to version v1:** Create a Virtual Service to route all traffic to v1 of each microservice by running:

   ```
   kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml
   ```

   A 'VirtualService' in Istio defines routing rules for traffic within the mesh.

   In particular, the 'hosts' and 'http' fields in the configuration file specify which traffic the routing rules apply to; the 'route' section determines where the traffic is directed, in this case to the 'v1' subset of the specified service, that as we said, were defined thanks the Destination Rule file `samples/bookinfo/networking/destination-rule-all.yaml`.

4. **Display defined routes:** To view the current Virtual Services and their configurations, use:

```
kubectl describe virtualservices
```

Verify the routing by accessing the application multiple times. You should only see reviews with no stars (v1).

If you see also the stars, this means that something is off in Istio. I advice you to delete the cluster and reinstall istio so restart from the Part 2 of this lab. Unfortunately it can happen (expecially in low spec environments) that something does not work well.

5. **Remove the previous Virtual Service rule:** Delete the Virtual Service rule that routes all traffic to v1 with:

```
kubectl delete -f samples/bookinfo/networking/virtual-service-all-v1.yaml
```

6. **Implement weighted routing:** Now simulate a more realistic scanario, we can think about v1 as the most lightweight service, while v3 the heaviest, so we want to forward the most of the traffic to v1, and smaller parts to v2 and v3. Let's route 50% of the traffic to v1 review service, 30% to v2 and 20% to v1. Apply this configuration (don't copy-paste from PDF, but instead copy from the file in '`yamls`' folder called '`istio-weighted-routing.yaml`'):

```
# Here instead of applying from a file, we do it via stdin. Just to change.
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 50
    - destination:
        host: reviews
        subset: v2
      weight: 30
    - destination:
        host: reviews
        subset: v3
      weight: 20
EOF
```

7. **Observe traffic distribution:** Access the application multiple times. You should now see that the most of the time there are no stars (so version 1 reviews).

8. **Delete the Virtual Service rule:** Another way to delete a resource, is to delete it by type of resource (virtualservice) and name of rule ('reviews', as indicated by the metadata-name above).

```
kubectl delete virtualservice reviews
```

4.6 Exercise 3: mTLS

In this exercise, you'll implement mutual TLS authentication between services using Istio's security features.

For further information, you can check the official documentation:
`https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/`.

By default, Istio operates in 'PERMISSIVE' mode. In this mode, an Istio-injected service (a service with an Envoy sidecar) can accept both plaintext traffic from services that are not injected and mutual

TLS traffic from other services that are injected. To force services in the service mesh to only communicate with Istio-injected services, you need to apply 'STRICT' mode.

Steps:

1. **Enable mutual TLS for the entire mesh:** This will apply to any namespace with Istio-injected pods. The file is as usual in '`yamls`' folder, named '`mTLS-strict.yaml`'... copy from there.

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

2. **Verify that mutual TLS is enabled** looking for 'PeerAuthentication' resources (`STRICT` means having mutual TLS for all the workloads of the namespace istio-system, so basically all your istio enabled mesh):

```
kubectl describe peerauthentication --all-namespaces
```

   Next, retrieve the IP address of the 'productpage' service or pod using the following commands:

```
# Retrieve the service cluster IP of the 'productpage' service
kubectl get svc -n default

# Or:

# Retrieve the pod IP of the 'productpage' pod
kubectl get pod -n default -o wide
```

   Alternatively, you can use a web dashboard to find the IPs. Remember, the IP you are looking for is the internal cluster IP, **NOT** the external 'GATEWAY_URL' used for ingress. We are testing mutual TLS (mTLS) **within** the Istio-injected cluster.

   Once you have the IP, try to access the service:

```
# Using the service cluster IP
wget -qO- http://PRODUCTPAGE_SERVICE_CLUSTER_IP:9080/productpage

# Using the pod IP
wget -qO- http://PRODUCTPAGE_POD_IP:9080/productpage
```

   If mutual TLS is correctly configured, the request from a pod without the Istio sidecar should fail, as it lacks the required certificates for mTLS. To verify the mTLS configuration, you can also delete the previous 'PeerAuthentication' rule with an appropriate 'kubectl' command.

```
kubectl delete peerauthentication default -n istio-system
```

   If needed, delete the pod and create it again:

```
kubectl delete pod legacy-app -n legacy
kubectl run legacy-app --image=busybox -n legacy --rm -it -- /bin/sh
```

   After recreating the pod, retry accessing the service. It should succeed if mutual TLS is not enforced.

4.7 Exercise 4: Observability

In this exercise, you'll explore Istio's observability features, including graphical visualization, monitoring and tracing.

For more information I suggest to check out the official doc:

- `https://istio.io/latest/docs/tasks/observability/kiali/`
- `https://istio.io/latest/docs/tasks/observability/metrics/using-istio-dashboard/`
- `https://istio.io/latest/docs/tasks/observability/distributed-tracing/jaeger/`

Steps: We are following this: `https://istio.io/latest/docs/setup/getting-started/#dashboard`

1. **Install Kiali, Grafana, and Jaeger:**

```
kubectl apply -f samples/addons
kubectl rollout status deployment/kiali -n istio-system
# Note: In low-spec environments, you may encounter issues during rollout. If you really \
    cannot solve, you can delete minikube cluster and reinstall Istio.
```

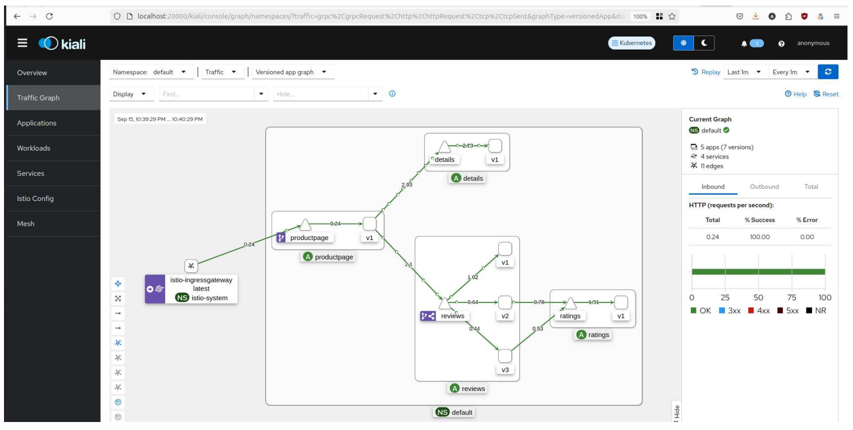
2. **Access the Kiali dashboard** to view the service mesh topology, traffic flow, and metrics.

```
istioctl dashboard kiali &
```

3. **Generate some traffic and visualize it via Kiali**.

```
for i in $(seq 1 10000); do curl -s -o /dev/null http://$GATEWAY_URL/productpage; done
```

Now visualize the traffic via the Traffic Graph in Kiali dashboard. Select the appropriate namespace and display options (Traffic animation is visually appealing, while Traffic rate is more concrete for understanding traffic distribution). You can manually refresh and adjust the refresh interval. You can also check error and success rates. Also, if you apply the previous discussed routing rules you'll are able to have a visual representation of the traffic splitting as you wanted.



Total	% Success	% Error
0.24	100.00	0.00

Figure 2: Kiali Web Dashboard

4. **Access Grafana dashboard** for metrics visualization:

```
istioctl dashboard grafana &
```

In Grafana, view the pre-built dashboards metrics for Istio, which provide insights into CPU, memory usage, and network traffic within the service mesh. Navigate to the dashboard labeled `istio` and try them. Generate traffic as mentioned above to see these metrics in action.

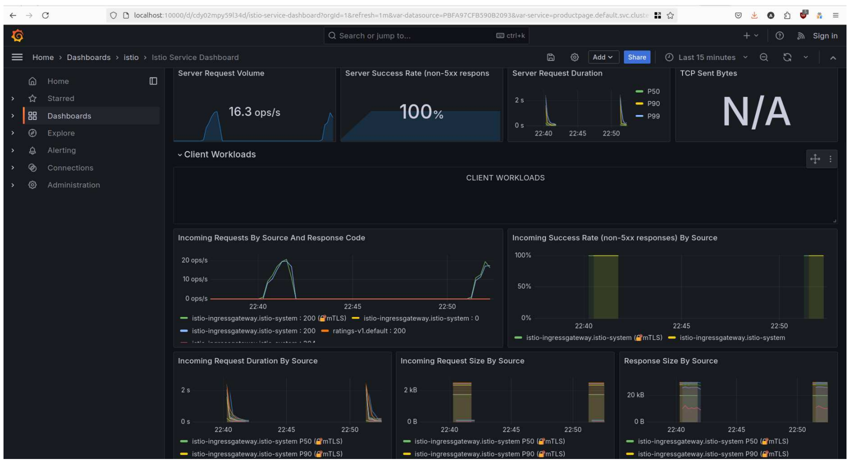

Server Request Volume
16.3 ops/s
Server Success Rate (non-5xx respons
100%
Server Request Duration
TCP Sent Bytes
N/A
Client Workloads
CLIENT WORKLOADS
Incoming Requests By Source And Response Code
Incoming Success Rate (non-5xx responses) By Source
Incoming Request Duration By Source
Incoming Request Size By Source
Response Size By Source

Figure 3: Grafana Web Dashboard

5. **Access Jaeger dashboard** for distributed tracing:

```
istioctl dashboard jaeger &
```

   In the Jaeger UI, select a service to view traces. Traces are made up of spans, which represent individual operations within a ”full” request. This allows you to understand the request flow and latency/faults across different microservices. The Istio service mesh integrates tracing via the Envoy sidecar proxy, although the application must include specific headers to correlate spans into a single trace.

   Note: if you don’t see any traces, it might be a bug: `https://github.com/orgs/jaegertracing/discussions/4687` `https://github.com/istio/istio/issues/8709`

5 Comparison of Network Policies and Service Meshes

5.1 Network Policies vs. Service Meshes

Network Policies:

- **Pros:**
  - Native Kubernetes resource, lightweight, and simple to implement.
  - Provides basic traffic control at L3/L4, then efficient.
- **Cons:**
  - Limited to L3/L4 traffic control; lacks advanced features such as mTLS, fine-grained traffic management, and observability.
  - Does not handle client-side load balancing or circuit breaking.

Service Meshes:

- **Pros:**
  - Advanced traffic management, client-side load balancing (which is more effective and scalable than k8s one), and circuit breaking (to prevent application overload and handle failures more gracefully).
  - Security improved with mTLS for secure service-to-service communication, and it's very easy to enable it, without manually handle certificates etc.
  - Observability and tracing out-of-the-box, without modifying application code or make application code heavier.
- **Cons:**
  - Adds complexity and overhead (requires deployment and managing of sidecar proxies (e.g., Envoy which is L7) in each pod), potentially increasing CPU consumption by 20-30%.

5.2 Best Practices and Considerations

- Use Network Policies as a first line of defense for controlling basic pod communication.
- Implement a Service Mesh for bigger architectures requiring more resilience and advanced features like mTLS, fine-grained traffic management, observability.

6 Troubleshooting

When working with Kubernetes, Network Policies, and Service Meshes, you may encounter issues. Here are some useful commands and strategies for troubleshooting:

6.1 Useful Commands

1. **View events** across all namespaces, sorted by creation timestamp, excluding normal events:

```
kubectl get events --all-namespaces --sort-by=.metadata.creationTimestamp | grep -v '\
    Normal' # where -v Normal excludes normal events and only shows other useful one, \
    like warning
```

   This command helps identify any warnings or errors that have occurred in your cluster.

2. **View all pods** across all namespaces with additional details:

```
kubectl get pods --all-namespaces -o wide
```

This provides a comprehensive view of all pods, including their status and the nodes they're running on.

3. **View logs** for a specific pod:

```
kubectl logs <pod-name> -n <namespace>
```

Replace `<pod-name>` with the name of the pod and `<namespace>` with the namespace it's in. This can help diagnose issues specific to a particular pod.

6.2 NetworkPolicy and Service Mesh Troubleshooting Strategies

- **Verify Network Policies:** Check policies with `kubectl describe networkpolicy --all-namespaces`. To test if a Network Policy is causing issues, you can temporarily delete it using `kubectl delete networkpolicy <policy-name> -n <namespace>`.
- **Istio Troubleshooting:** Run `istioctl proxy-status` to check for stale states that indicate communication issues. For broader configuration checks, use `istioctl analyze` optionally specifying a namespace with `-n`.
- **Service Mesh Observability:** Use Kiali, Grafana and Jaeger to diagnose issues in your service mesh.