NIST Technical Note

-

Blueprint for Deploying Secure Cloud-Native Environments for 5G/O-RAN Testbeds:

Multi-Cluster Kubernetes and Service Mesh

Draft Stage

Kye Hwan Lee

This publication is available free of charge from:

-
June 2025

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

NIST Technical Series Policies
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History

Contact Information
kyehwan.lee@nist.gov

-
June 2025

Abstract

This document explains the installation of Kubernetes multi cluster with Service mesh using Istio and envoy proxy.

Keywords

Istio, Envoy, IngressGateway, EastWestGateway, Service Mesh, Cilium

Table of Contents

List of Tables

List of Figures

1. System Requirements

This section introduces the system environment and prerequisites for deploying the Zero Trust Architecture (ZTA) framework in a cloud-native setting. Prior to the installation and integration of the ZTA components, the host system must be properly configured with the required operating system and essential tools.

The ZTA framework includes service mesh components, policy enforcement modules, and secure service communication via mTLS (mutual Transport Layer Security) and JWT (JSON Web Token)-based identity authentication. To enable proper deployment, the following system requirements and installation steps must be satisfied.

1.1. Source Download

The ZTA components can be downloaded directly from the official Git repository:

- git clone https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed

This command retrieves the latest version of the ZTA source code and associated configuration files necessary for deployment.

1.2. Online Installation Guide

A comprehensive installation guide, including step-by-step instructions, sample manifests, and environment variable configurations, is available at:

- https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed/README.md

This guide also provides troubleshooting tips and validation procedures to ensure all components are correctly deployed and operational.

1.3. Pre-requisite

Before proceeding with the installation, verify that the system meets the following baseline requirements:

1.3.1. Operating System

- Ubuntu 20.04 LTS (recommended)
- Ubuntu 22.04 LTS (supported)

The framework is tested and verified on the above Linux distributions to ensure compatibility with key open-source tools such as Kubernetes, Istio, and Cilium.

1.3.2. Software Dependencies

The following packages must be installed and available in the system path:

- git – for cloning source repositories
- curl – for downloading scripts or validating services
- make – for building deployment assets (optional)
- docker – for building custom container images (if required)
- kubectl – for managing Kubernetes clusters
- istioctl – for Istio control plane configuration
- helm – for deploying optional services via Helm charts

You can install the required software using the following command (Ubuntu example):

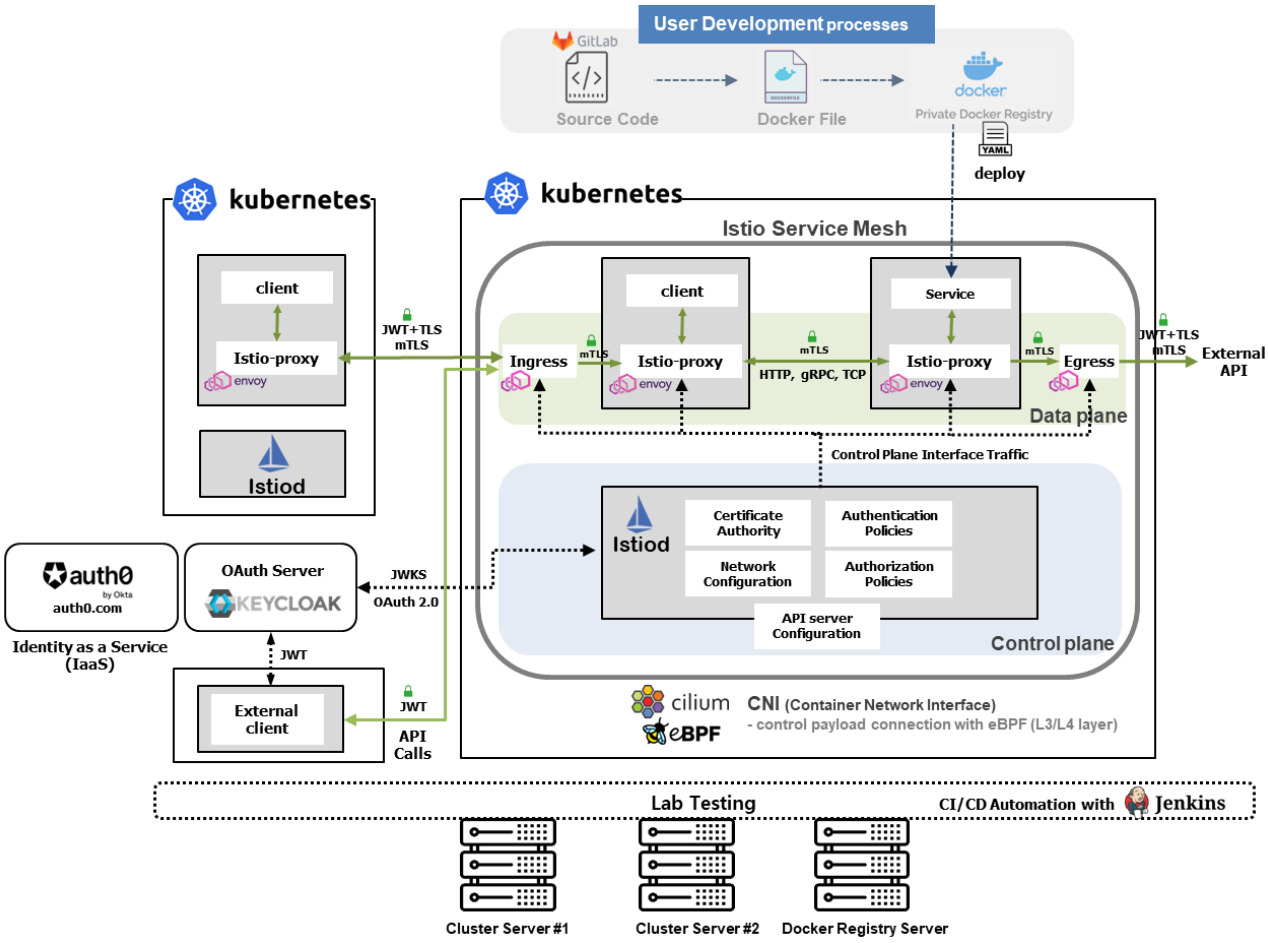
```
sudo apt update && sudo apt install -y git curl make
```

The Docker installation procedure will be revisited in Section 3.3 and the Kubernetes installation also will be handled in Section 4. The rest of various tools should be installed according to the cloud provider or on-premises cluster configuration.

2. Testbed Specification

This testbed is designed to evaluate a multi-component microservice architecture deployed across multiple Kubernetes clusters. It leverages high-performance Dell PowerEdge servers and integrates core cloud-native tools such as Istio, Cilium, and monitoring stacks for end-to-end observability and security. In this testbed, high availability is achieved through a multi-cluster, multi-control plane Istio deployment. Each Kubernetes cluster hosts its own Istio control plane (istiod) that manages local proxies while maintaining trust and policy consistency across clusters.

2.1. Architecture and function of Zero Trust Architecture (ZTA) testbed



2.1.1. User Development Processes (Top Section)

This area represents the application development pipeline:

- **GitLab** is used for version control and managing source code.
- Developers define Dockerfiles and containerize their applications.

- **Private Docker Registry** securely stores the application images, which are later pulled by Kubernetes nodes for deployment.

2.1.2. Kubernetes Clusters

There are **two Kubernetes clusters** depicted:

- **Left Cluster**: Contains a client workload, an Istio sidecar proxy, and its own Istiod control plane.
- **Right Cluster**: Hosts both client and service workloads, each with sidecar Istio-proxy (Envoy) and a dedicated Istiod control plane.

These clusters form the infrastructure for service mesh deployment and ZTA enforcement.

2.1.3. Istio Service Mesh

2.1.3.1. Data Plane

The **data plane** is made up of **Envoy proxies** injected alongside workloads:

- **Client ↔ Service traffic** is transparently handled by Envoy sidecars.
- All intra-mesh communication (HTTP, gRPC, TCP) is encrypted via **mTLS** (mutual TLS).
- Traffic entering the mesh from the outside goes through **Ingress** and exiting traffic uses **Egress** gateways.

This ensures end-to-end encryption and observability across services.

2.1.3.2. Control Plane

The **control plane** (powered by Istiod) manages:

- **Certificate Authority**: Issues and rotates certificates for mTLS.
- **Network Configuration**: Handles routing rules, service discovery, and traffic management.
- **Authentication/Authorization Policies**: Enforces **JWT-based authentication**, RBAC policies, and Zero Trust principles.

2.1.3.3. Ingress Gateway

- Acts as the entry point for external traffic.
- Enforces authentication policies, such as validating **JWT tokens**.
- Terminates TLS and forwards the request into the mesh securely.

2.1.3.4. External Client + Keycloak

- External clients (e.g., from outside the mesh) must authenticate via **Keycloak** (OAuth/OIDC provider).
- Clients receive JWT tokens which are verified at the **Ingress Gateway** by Istio.

2.1.3.5. Istiod

- Each cluster has its own **Istiod** instance managing local workloads.
- It syncs configuration, manages workload identities, and pushes XDS configs to Envoy proxies

2.1.3.6. Cilium as the CNI

- **Cilium** replaces the default Kubernetes CNI to provide **secure and observable networking** using **eBPF**.
- It enables **fine-grained policies**, **deep packet inspection**, and **performance monitoring** at L3–L7 layers.
- It ensures that the datapath between pods is highly secure and auditable.

2.1.3.7. mTLS and Service Identity

- All communication within the mesh uses **mTLS**, ensuring both encryption and **identity-based authorization**.
- The X-Forwarded-Client-Cert header (not shown but often used in mTLS) allows tracing which client service made the request.

2.2. Dell PowerEdge R640 Server specification

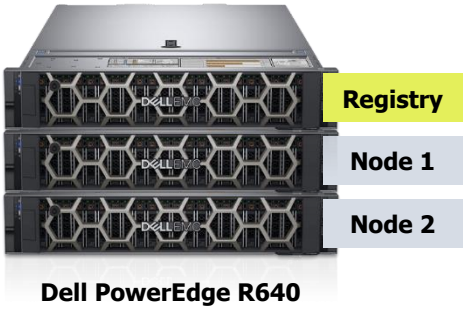
Each server used in this testbed shares the following hardware configuration, ensuring consistent compute and network capabilities across the nodes:

- Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz
- 64GB system Memory
- 32GB DDR4 DIMM, speed up to 3200 MT/s
- SCSI Disk 931GiB (999GB)
- Embedded NIC: 4× 1GbE ports

These specifications provide sufficient processing power, memory bandwidth, and disk space to support multiple containerized workloads and service mesh functionalities in a production-like testing environment.

2.3. Interfaces on each server

The roles of the three physical servers are logically partitioned to separate infrastructure services, control plane, and workload functions. The following outlines their hostname configurations and designated roles:



Spec:
Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz
64GB system Memory
32GB DDR4 DIMM, speed up to 3200 MT/s
SCSI Disk 931GiB (999GB)
Embedded NIC: 4× 1GbE ports

Registry

- **Docker Registry**
- **OAuth server**
- **Storage & ETC**

Node 1

- **Kubernetes 1.32**
- **Istio v1.24 + Cilium v1.17**
- **dec-SMO application**
- **Grafana, Kiali for monitoring**

Node 2

- **Kubernetes 1.32**
- **Istio v1.24 + Cilium v1.17**
- **External Clusters**
- **External Client app.**
- **ETC**

2.3.1. Server 1

- Hostname: 5g1-comp1.antd.nist.gov
- Role: Docker Registry v2, OAuth v2.0, Storage and ETC

This node hosts auxiliary services critical to the infrastructure, such as the private Docker registry for container image management, an OAuth authorization server, and persistent storage for stateful services.

2.3.2. Server 2

- Hostname: 5g1-comp2.antd.nist.gov
- Role: Kubernetes v1.32, Istio v1.24, Cilium v1.17, Grafana, Kiali, dec-SMO application

This server operates as the main control plane for service deployment and monitoring. It runs Kubernetes with Cilium as the CNI, and Istio for service mesh capabilities. Grafana and Kiali are used for observability of metrics and mesh visualization, while the dec-SMO application represents a core service under test.

2.3.3. Server 3

- Hostname: 5g1-comp3.antd.nist.gov
- Role: Kubernetes v1.32, Istio v1.24, Cilium v1.17, external client applications

This server acts as a secondary Kubernetes cluster and simulates external client interactions. It also runs Istio and Cilium, enabling secure and observable communication between clusters through the Istio multi-mesh gateway.

2.3.4. IP Addresses and account information for testbeds

The following table summarizes the IP address assignments and administrative access credentials for each physical testbed server in the ZTA test environment. Each server is configured with dual interfaces (eno3 and eno4) to support management and service/data plane separation. Consistent login credentials are applied across the testbed to streamline automated configuration and deployment tasks.

```
Server 1 (5g1-comp1)
  IP address for eno3: 10.5.0.2 - Used for control-plane and management traffic
  IP address for eno4: 10.5.1.2 - Used for service plane or application-specific
communication
  Account:  onfadmin
  Password: 5Gtb@ctl

Server 2 (5g1-comp2)
  IP address for eno3: 10.5.0.3
  IP address for eno4: 10.5.1.3
  Account:  onfadmin
  Password: 5Gtb@ctl

Server 3 (5g1-comp3)
  IP address for eno3: 10.5.0.4
  IP address for eno4: 10.5.1.4
  Account:  onfadmin
  Password: 5Gtb@ctl
```

- Server 1 hosts key infrastructure components such as the private Docker registry, OAuth server, persistent storage services, and other supporting utilities.
- Server 2 serves as the main Kubernetes cluster control-plane host, with Istio, Cilium CNI, observability tools (e.g., Grafana, Kiali), and key ZTA applications like dec-SMO deployed.

- Server 3 acts as a secondary Kubernetes environment and is primarily used to simulate external clients and to validate cross-cluster service behavior under ZTA policies.

3. Installation Docker Registry

3.1. Why Use a Docker Private Registry

In a production-grade Kubernetes environment, especially one leveraging **Istio service mesh** and **Cilium CNI**, a **Docker private registry** enhances security, control, and performance in managing container images.

3.2. Benefits of Using a Private Container Registry

- **Security and Access Control**
  A private registry allows you to enforce strict access controls, ensuring that only authorized users or CI/CD pipelines can push or pull images. This reduces the risk of image tampering or exposure to public vulnerabilities.
- **Faster Deployment and Lower Latency**
  Hosting the registry within the local or on-premises network minimizes image pull time across clusters. This is especially valuable in **multi-cluster setups** where clusters may reside on isolated networks or have limited external bandwidth.
- **Version and Provenance Control**
  With a private registry, you can tightly manage image versions, audit image changes, and ensure deployments are using trusted, verified artifacts built by CI pipeline.
- **Compliance and Governance**
  For regulated industries, a private registry supports compliance with internal policies and external regulations by ensuring that only approved images are used and retained securely.
- **Offline or Air-Gapped Support**
  In disconnected environments (e.g., edge deployments or secure networks), a private registry enables deployments without reliance on external public registries like Docker Hub or Quay.
- **Avoiding Pull Limits and Downtime Risks**
  Public registries such as Docker Hub enforce rate limits and are subject to outages. A private registry avoids these issues and ensures operational independence.

3.3. Docker CE (Community Edition) installation on Ubuntu

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install -y 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 latest version
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
compose-plugin
```

3.4. Running Docker Registry

3.4.1. Preparing Docker Registry Certificate

For having domain certificate represent SAN (Subject Alt Name: IP address), use the CSR described as an example below.

```
[req]
distinguished_name = private_registry_cert_req
x509_extensions = v3_req
prompt = no

[private_registry_cert_req]
OU = NIST 5G Testbed docker repository
CN = 10.5.0.2

[v3_req]
keyUsage = keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[alt_names]
IP.0 = 10.5.0.2
```

To request TLS certificate for the docker registry, the openssl command is used with “-config” option.

```
$ openssl req -x509 -config $PWD/tls.csr \
              -nodes -newkey rsa:4096 \
              -keyout tls.key -out tls.crt \
              -days 3650 -extensions v3_req
```

The certs generated as a result will be located in /home/onfadmin/docker-registry on the 5g1-comp1 server, and they are used for running the Docker registry.

3.4.2. Running command for Docker Registry

```
docker run -d --restart=always --name registry \
              -v ./docker/certs:/docker-in-certs:ro \
              -v ./registry_images:/var/lib/registry \
              -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
              -e REGISTRY_HTTP_TLS_CERTIFICATE=/docker-in-certs/tls.crt \
```

```
               -e REGISTRY_HTTP_TLS_KEY=/docker-in-certs/tls.key \
               -p 8443:443 registry:2
```

Check the docker registry status with ‘docker ps’ command.

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND    CREATED         STATUS                  PORTS
NAMES
f7ebdc789796   registry:2 "/entrypoint.sh /etc…"   9 seconds ago   Up 4 seconds
5000/tcp, 0.0.0.0:8443->443/tcp, :::8443->443/tcp      registry
```

3.5. Test example for using docker image in private docker repository

3.5.1. Pre-requisite – docker registry TLS certs handling on the client host

3.5.1.1. For docker push/pull

In order to access the docker registry running on 5g1-comp1 server (10.5.0.2) from another host, it needs to have TLS certificate which was made in 3.4.1 on the directory where docker can find (as a default setting, it is /etc/docker/certs.d/).

- For your information, tls.cert is located at /home/onfadmin/docker-registry/docker/certs/tls.crt on the docker registry

Next script shows the process how the certs is located and copied.

```
$ sudo mkdir -p /etc/docker/certs.d/10.5.0.2:8443
$ scp onfadmin@5g1-comp1.antd.nist.gov:/home/onfadmin/docker-registry/docker/certs/tls.crt
/etc/docker/certs.d/10.5.0.2:8443
```

3.5.1.2. For Kubernetes push/pull

To access the docker registry with Kubernetes, it needs to update ca certificate with the following command so that the system may recognize and update its ca certs pool.

```
sudo cp /etc/docker/certs.d/10.5.0.2:8443/certs/tls.crt /usr/local/share/ca-
certificates/ca.crt  (ca.crt or any name of crt, target: host running k8s)

sudo update-ca-certificates
sudo service docker restart   (if docker-ce is not installed, this part doens't need to be
executed)
sudo systemctl restart containerd
```

Otherwise, it may display "tls: failed to verify certificate: x509: certificate signed by unknown authority" from 'kubectl get events -w'

3.5.2. Docker upload (push), download (pull) to/from the private docker repository

This section demonstrates building a sample Docker image, pushing it to a private Docker registry, and verifying the repository contents. The private registry runs on 5g1-comp1 (10.5.0.2:8443)**.**

Change to “6-example” directory to test docker push/pull example for Kubernetes cluster

Step 1: Prepare docker image, Dockerfile

```
# Dockerfile
FROM python:3.9-slim

# work directory
WORKDIR /app

# copy source
COPY app.py .

# Flask install
RUN pip install flask

# Flask start when container runs
CMD ["python", "app.py"]
```

Step 2: Docker build

Run the following command in the 6-example directory:

```
$ docker build -t flask-hello ./
```

Step 3: Docker push to the Docker private repository (5g1-comp1 server: 10.5.0.2:8443)

Tag the image with the private registry address and push it:

```
$ docker tag flask-hello:latest 10.5.0.2:8443/flask-hello
$ docker push 10.5.0.2:8443/flask-hello
```

The push output confirms that the image has been uploaded, showing the digest and size of the image layers.

Step 4: Verify the Repository Contents

Check whether the image is stored in the registry:

```
$ curl -k https://10.5.0.2:8443/v2/_catalog
{"repositories":["flask-hello"]}
```

Expected output:

```
{"repositories":["flask-hello"]}
```

- The private Docker registry must be running and accessible at 10.5.0.2:8443.
- Using -k with curl bypasses certificate verification, since the registry is secured with a self-signed certificate.
- Images pushed to the private registry can now be pulled by Kubernetes nodes for deployment.

3.5.3. upload to Kubernetes cluster

To upload the flask-hello image into the Kubernetes cluster, it needs to have an image address where the docker images located, in this case, 10.5.0.2:8443/flask-hello:latest. This file was prepared in previous step for adding repository server's IP address and port number information (10.5.0.2:8443).

```
< istio-flask-hello-service-deploy.yaml >
apiVersion: v1
kind: Service
metadata:
  name: flask-hello
  namespace: sample
spec:
  selector:
    app: flask-hello
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-hello
  namespace: sample
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-hello
  template:
    metadata:
      labels:
        app: flask-hello
    spec:
      containers:
      - name: flask-hello
        image: 10.5.0.2:8443/flask-hello:latest
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: registry-ca
```

Check the deployment status

```
$ kubectl get po -n sample -l app=flask-hello
NAME                           READY   STATUS    RESTARTS   AGE
flask-hello-64576d5bf-x9g5w   2/2     Running   0          28d
```

3.5.4. To check the application running,

```
$ kubectl exec -it -n sample curl-5b549b49b8-5jfmf -- curl -v flask-hello.sample/hello
* Host flask-hello.sample:80 was resolved.
* IPv6: (none)
* IPv4: 10.108.18.22
*   Trying 10.108.18.22:80...
* Connected to flask-hello.sample (10.108.18.22) port 80
* using HTTP/1.x
> GET /hello HTTP/1.1
> Host: flask-hello.sample
> User-Agent: curl/8.13.0
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< server: envoy
< date: Wed, 04 Jun 2025 19:46:46 GMT
< content-type: text/html; charset=utf-8
< content-length: 11
< x-envoy-upstream-service-time: 14
<
* Connection #0 to host flask-hello.sample left intact
hello world
```

The test output above demonstrates a successful end-to-end communication flow in an Istio-enabled Kubernetes environment using Envoy as the service proxy.

In the following test, a curl pod running in the sample namespace issues an HTTP GET request to the flask-hello service at path /hello:

```
kubectl exec -it -n sample curl-5b549b49b8-5jfmf -- curl -v flask-hello.sample/hello
```

Key highlights from the output:

- * Connected to flask-hello.sample (10.108.18.22) port 80: confirms that the DNS resolution and connection to the internal Kubernetes service succeeded.
- < HTTP/1.1 200 OK: indicates that the request was handled successfully, with an HTTP 200 OK status returned.
- < server: envoy: **this is critical** — it shows that Envoy, the sidecar proxy injected by Istio, is actively handling the traffic. Envoy is responsible for managing traffic routing, enforcing policies, and collecting telemetry.
- < x-envoy-upstream-service-time: 14: confirms that Envoy forwarded the request upstream and returned the response with a measured processing time.

- hello world: this is the actual payload returned by the flask-hello application, confirming that the REST API executed successfully, and that the response was correctly routed back through Envoy to the calling curl client.

Note:
This result confirms that Envoy has successfully handled and routed the HTTP request to the flask-hello service and returned the correct response. The presence of "server: envoy" and the expected hello world output validates that Istio's service mesh is functioning as intended for REST API traffic.

4. Install Kubernetes

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. In modern cloud-native architectures, Kubernetes serves as the foundational layer for ensuring reliable, efficient, and scalable application delivery.

4.1. Using an installation script on GitLab repository,

The installation scripts for setting up the testbed are provided in a public GitLab repository. Begin by cloning the repository:

```
$ git clone https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed
```

This repository contains the scripts required to install Kubernetes, Istio, and supporting components.

4.2. First, install all the required software with "1-install_istio-test-setup.sh" script on both 5g1-comp2 and 5g1-comp3 servers

On both 5g1-comp2 and 5g1-comp3 servers, execute the 1-install_istio-test-setup.sh script. This script installs all dependencies needed to bootstrap the environment:

```
$ sh -c -x 1-install_istio-test-setup.sh

install_kubectl.sh
install_istioctl.sh
install_cilium.sh
install_k9s_webi.sh
```

- install_kubectl.sh – installs the Kubernetes command-line tool.
- install_istioctl.sh – installs the Istio CLI tool.
- install_cilium.sh – installs Cilium as the CNI plugin.
- install_k9s_webui.sh – installs K9s for Kubernetes monitoring.

4.3. Use "2-bootstrap-kube-install-istio-setup-comp2.sh" script to install on both 5g1-comp2 and 5g1-comp3 servers

Next, run the bootstrap script 2-bootstrap-kube-install-istio-setup-comp2.sh on both 5g1-comp2 and 5g1-comp3:

```
$ sh -c 2-bootstrap-kube-install-istio-setup-comp2.sh
```

This script installs the core pods at the kube-system namespace

```
kube-system    coredns-7c65d6cfc9-blwmc  1/1  Running   192.168.0.205   5g1-comp3
kube-system    coredns-7c65d6cfc9-f9qsn  1/1  Running   192.168.0.186   5g1-comp3
kube-system    etcd-5g1-comp3            1/1  Running   10.5.0.4        5g1-comp3
kube-system    kube-apiserver-5g1-comp3  1/1  Running   10.5.0.4        5g1-comp3
kube-system    kube-controller-manager-5g1-comp3 1/1    Running  10.5.0.4 5g1-comp3
kube-system    kube-proxy-tqsvn          1/1  Running   10.5.0.4        5g1-comp3
kube-system    kube-scheduler-5g1-comp3  1/1  Running   10.5.0.4        5g1-comp3
```

This script installs the core Kubernetes pods into the kube-system namespace, including:

- **CoreDNS** – cluster DNS service
- **etcd** – key-value store for Kubernetes
- **kube-apiserver** – Kubernetes API server
- **kube-controller-manager** – manages cluster controllers
- **kube-scheduler** – schedules Pods to nodes
- **kube-proxy** – manages network rules on each node

5. Cilium CNI installation

Cilium is used as the **Container Network Interface (CNI)** in this Kubernetes setup to provide advanced, secure, and observability-rich networking for workloads. It leverages **eBPF (extended Berkeley Packet Filter)** in the Linux kernel to deliver high-performance packet processing, fine-grained security policies, and deep visibility without relying on iptables.

5.1. Key Benefits of Using Cilium:

- **eBPF-Powered Performance**: Cilium avoids iptables scaling limitations by implementing network and security logic at the kernel level with eBPF. This results in faster connection setup, lower latency, and more efficient CPU usage.
- **L7-Aware Network Policies**: Unlike traditional CNIs, Cilium can enforce Kubernetes network policies at Layer 7 (e.g., HTTP/gRPC), which complements Istio's service mesh security model and provides defense-in-depth.
- **Deep Observability**: Cilium provides built-in flow visibility and network tracing through Hubble, which is especially valuable when debugging traffic in complex Istio and multi-cluster deployments.
- **Seamless Integration with Istio**: Cilium supports transparent proxy redirection with Istio's sidecar model, ensuring that Envoy traffic redirection is handled efficiently and securely.
- **Compatibility with MetalLB**: Cilium interoperates cleanly with MetalLB in Layer 2 mode, allowing external IPs to be assigned and routed correctly even in bare-metal environments.

5.2. Why Cilium in this Architecture

In this architecture—Istio-based service mesh, with MetalLB exposing ingress and east-west Envoy proxies—Cilium ensures reliable pod connectivity, enforces security policies between services, and gives clear traffic visibility. It enables scalable, secure networking that aligns well with service mesh goals while offering kernel-level efficiency.

5.3. Cilium installation onto two clusters – both 5g1-comp2 and 5g1-comp3.

Use "3-helm_install_cilium_mod-comp2.sh" script to handle CNI installation with helm chart.

```
$ sh -c 3-helm_install_cilium_mod-comp2.sh
```

CLUSTER1_CTX macro can be designated for 5g1-comp2 and CLUSTER2_CTX macro is for 5g1-comp3 server in the installation script.

```
$ echo "Installing cilium in $CLUSTER1_NAME..."
$helm upgrade --install cilium cilium/cilium --version 1.14.1 --kube-context $CLUSTER1_CTX \
   --namespace kube-system \
```

```
   --set cluster.name=cluster1 \
   --set cluster.id=1 \
   --set operator.replicas=1 \
   --set image.pullPolicy=IfNotPresent \
   --set ipam.mode=kubernetes \
   --set bgpControlPlane.enabled=true


$echo "Installing cilium in $CLUSTER2_NAME..."
$helm upgrade --install cilium cilium/cilium --version 1.14.1 --kube-context $CLUSTER2_CTX \
   --namespace kube-system \
   --set cluster.name=cluster2 \
   --set cluster.id=2 \
   --set operator.replicas=1 \
   --set image.pullPolicy=IfNotPresent \
   --set ipam.mode=kubernetes \
   --set bgpControlPlane.enabled=true
```

6. Metallb Load Balancer

In bare-metal Kubernetes clusters or environments without a cloud load balancer, **MetalLB** provides the essential LoadBalancer service type support required by Istio's ingress and east-west gateways. Istio deploys its **Envoy proxy** via Kubernetes `Service` resources of type `LoadBalancer` to expose ingress traffic or enable multi-cluster communication.

Since native cloud load balancers are not available in such environments, MetalLB acts as a software load balancer that assigns **external IPs** to these services. This enables seamless integration of Istio gateways with the cluster's networking layer, allowing external clients or other clusters to reliably access the Envoy proxies.

Without MetalLB, services like `istio-ingressgateway` or `istio-eastwestgateway` would lack externally reachable IPs, making it impossible to route traffic into the mesh.

To install, use the provided shell script,

```
$ sh -c 4-install_metallb_istio-mod-comp2.sh
```

Or manually, it will be installed step by step to achieve MetalLB installation and configuration of IPAddressPool and L2Advertisement.

```
$ kubectl apply --context="${CLUSTER1_CTX}" -f
https://raw.githubusercontent.com/metallb/metallb/v0.13.10/config/manifests/metallb-
native.yaml

$ kubectl apply --context="${CLUSTER2_CTX}" -f
https://raw.githubusercontent.com/metallb/metallb/v0.13.10/config/manifests/metallb-
native.yaml


$ kubectl apply --context="${CLUSTER1_CTX}" -f - <<EOF
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: lb-pool
  namespace: metallb-system
spec:
  addresses:
  - 10.5.0.100-10.5.0.120
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: lb-adv
  namespace: metallb-system
EOF


$ kubectl apply --context="${CLUSTER2_CTX}" -f - <<EOF
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
```

```
metadata:
  name: kind-pool
  namespace: metallb-system
spec:
  addresses:
  - 10.5.0.121-10.5.0.139
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: all-pools
  namespace: metallb-system
EOF
```

7. Istio Installation

This section describes the process of installing Istio service mesh into a Kubernetes-based testbed environment. Istio provides powerful traffic management, security, observability, and policy control across microservices. For our testbed, Istio was installed in **multi-primary, multi-network** mode to support service communication across independent Kubernetes clusters.

7.1. Installation Prerequisites

Before installing Istio, the following components must be prepared on each Kubernetes cluster:

- Kubernetes (v1.26 or later) installed and running
- istioctl CLI installed (istioctl version 1.24.3 or later)
- MetalLB installed and configured for LoadBalancer support
- Calico or Cilium installed as CNI with IP forwarding enabled
- Kubernetes control-plane node has internet access or pre-loaded Istio images

7.2. Istio Deployment Steps and Options

This section describes the steps taken to install and configure Istio in a dual-cluster environment (`cluster1` and `cluster2`) with mutual TLS (mTLS), east-west gateway configuration, and endpoint discovery for service communication across clusters.

To deploy Istio across multiple Kubernetes clusters, two installation options are provided: **manual step-by-step configuration** and **automated deployment using a script**.

Note: Before running the script, ensure that the environment variables and required files (e.g., env-istio.sh, certificate files, cluster context configuration) are correctly set up as outlined in the repository documentation.

This approach significantly reduces human error and accelerates deployment in reproducible testbed environments.

7.3. Manual Installation

For users who prefer greater control over each installation step—or for educational, debugging, or fine-tuning purposes—the following section outlines the **manual installation procedure** for Istio across multiple Kubernetes clusters.

This method involves explicitly executing commands to:

- Install Istio control planes in each cluster using istioctl
- Deploy certificates for mutual TLS
- Set up east-west gateways for cross-network communication

- Expose gateway services using Istio Gateway resources
- Exchange remote secrets between clusters to enable endpoint discovery

Manual installation is particularly useful for understanding the internal structure and behavior of Istio in multi-cluster environments. It also serves as a transparent reference that complements the automated script found in the GitLab repository.

7.3.1. Environment Setup

Before beginning the Istio installation, environment variables are loaded from a predefined script:

```
. ../env-istio.sh
LOC="$PWD"
```

7.3.2. Certificate Installation for mTLS

Istio can be configured with custom root and intermediate certificates to enable mTLS authentication between services. Once a common root certificate is generated, intermediate certificates can be created and signed by the root. The same CA certificate may also be reused for CSR generation in each cluster.

For deployment, the required certificates and keys are provided under the repository folders `certs/cluster1` and `certs/cluster2`. Using these files, Kubernetes secrets are created in each cluster's `istio-system` namespace. These secrets combine the cluster-specific certificates and keys and are applied with the `kubectl create secret generic cacerts …` command.

For Cluster 1:

```
$ kubectl create namespace istio-system --context=${CLUSTER1_CTX}
$ kubectl create secret generic cacerts -n istio-system \
  --from-file=${LOC}/certs/cluster1/ca-cert.pem \
  --from-file=${LOC}/certs/cluster1/ca-key.pem \
  --from-file=${LOC}/certs/cluster1/root-cert.pem \
  --from-file=${LOC}/certs/cluster1/cert-chain.pem \
  --context=${CLUSTER1_CTX}
```

For Cluster 2:

```
$ kubectl create namespace istio-system --context=${CLUSTER2_CTX}
$ kubectl create secret generic cacerts -n istio-system \
  --from-file=${LOC}/certs/cluster2/ca-cert.pem \
  --from-file=${LOC}/certs/cluster2/ca-key.pem \
  --from-file=${LOC}/certs/cluster2/root-cert.pem \
  --from-file=${LOC}/certs/cluster2/cert-chain.pem \
  --context=${CLUSTER2_CTX}
```

7.3.3. Network Labeling

In a multi-network Istio deployment, each cluster's Istio namespace must be labeled to indicate its logical network name. This labeling allows Istio to differentiate service endpoints that reside on different networks and to configure the east-west gateway routing accordingly.

- The label topology.istio.io/network=<network-name> must be set in the istio-system namespace of every cluster.
- Each cluster should use a distinct network name (e.g., network1, network2).
- These labels are critical for enabling Istio's multi-network service discovery and traffic routing.

```
$ kubectl --context="${CTX_CLUSTER1}" label namespace istio-system
topology.istio.io/network=network1
$ kubectl --context="${CTX_CLUSTER2}" label namespace istio-system
topology.istio.io/network=network2
```

7.3.4. Istio Control Plane Installation

Istio control planes are installed separately in each cluster using the IstioOperator resource. The installation specifies the mesh ID, cluster name, and network name, ensuring that Istio recognizes each cluster as part of the same mesh while keeping network identities distinct.

For Cluster1:

```
$ cat <<EOF > cluster1.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster1
      network: network1

EOF

$ istioctl --context="${CLUSTER1_CTX}" install -f ${LOC}/cluster1.yaml --skip-confirmation
```

For Cluster2:

```
$ cat <<EOF > cluster2.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
```

```
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster2
      network: network2

EOF

$ istioctl --context="${CLUSTER2_CTX}" install -f ${LOC}/cluster2.yaml --skip-confirmation
```

7.3.5. East-West Gateway Deployment

To enable service discovery and communication between clusters, an Istio east-west gateway needs to be deployed in each cluster. In this east-west gateway options, the name of cluster1 ‘s network identification, network1, will be used in the spec.ingressGateway.label and values.global.network. This esst-west gateway specifies port numbers to accept the request which comes from the outside cluster, then it handles according to the port number to send the traffic to the designated port number which belongs to the target pod.

Below is the east-west gateway configuration requesting Istio to generate the gateway with the service port for external input. In case of multi-cluster environment, especially multiple-primary, different network environment, each cluster has to have the east-west gateway to control ingress/egress traffic through the east-west gateway.

Cluster 1 Gateway Installation:

```
$ istioctl --context="${CTX_CLUSTER1}" install -y -f - <<EOF
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: eastwest
spec:
  revision: ""
  profile: empty
  components:
    ingressGateways:
      - name: istio-eastwestgateway
        label:
          istio: eastwestgateway
          app: istio-eastwestgateway
          topology.istio.io/network: network1
        enabled: true
        k8s:
          env:
            # traffic through this gateway should be routed inside the network
            - name: ISTIO_META_REQUESTED_NETWORK_VIEW
              value: network1
          service:
            ports:
              - name: status-port
                port: 15021
                targetPort: 15021
              - name: tls
                port: 15443
```

```
                targetPort: 15443
              - name: tls-istiod
                port: 15012
                targetPort: 15012
              - name: tls-webhook
                port: 15017
                targetPort: 15017
  values:
    gateways:
      istio-ingressgateway:
        injectionTemplate: gateway
    global:
      network: network1
EOF
```

Cluster 2 Gateway Installation:

```
$ istioctl --context="${CTX_CLUSTER2}" install -y -f - <<EOF
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: eastwest
spec:
  revision: ""
  profile: empty
  components:
    ingressGateways:
      - name: istio-eastwestgateway
        label:
          istio: eastwestgateway
          app: istio-eastwestgateway
          topology.istio.io/network: network2
        enabled: true
        k8s:
          env:
            # traffic through this gateway should be routed inside the network
            - name: ISTIO_META_REQUESTED_NETWORK_VIEW
              value: network2
          service:
            ports:
              - name: status-port
                port: 15021
                targetPort: 15021
              - name: tls
                port: 15443
                targetPort: 15443
              - name: tls-istiod
                port: 15012
                targetPort: 15012
              - name: tls-webhook
                port: 15017
                targetPort: 15017
  values:
    gateways:
      istio-ingressgateway:
        injectionTemplate: gateway
    global:
      network: network2
```

```
EOF
```

7.3.6. Gateway Resource Configuration

In a multi-cluster Istio deployment, the east-west gateway is used to expose services across clusters. This allows workloads in one cluster to communicate securely with workloads in another cluster.

- The gateway is deployed in the istio-system namespace.
- Port 15443 is used for cross-cluster TLS traffic.
- AUTO_PASSTHROUGH mode allows TLS connections to be forwarded without terminating them at the gateway.
- The same configuration must be applied to both Cluster1 and Cluster2.
- This ensures that services behind each east-west gateway are discoverable and accessible by workloads in the other cluster.

Expose services via the istio-eastwestgateway in both clusters:

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: cross-network-gateway
  namespace: istio-system
spec:
  selector:
    istio: eastwestgateway
  servers:
    - port:
        number: 15443
        name: tls
        protocol: TLS
      tls:
        mode: AUTO_PASSTHROUGH
      hosts:
        - "*.local"
```

Repeat the same configuration for cluster2

7.3.7. Endpoint Discovery Setup

To enable Istio control planes to discover services running in remote clusters, Istio requires exchanging remote Kubernetes secrets between the clusters. This exchange provides each cluster's control plane with credentials to access the other cluster's API server.

- istioctl create-remote-secret generates a secret that contains the credentials of one cluster's API server.
- This secret is applied to the other cluster, enabling bidirectional endpoint discovery.

- Without this step, each Istio control plane can only discover and manage services within its own cluster.
- Proper naming (--name=cluster1, --name=cluster2) is critical to avoid conflicts and ensure clarity.

```
# For cluster1 → cluster2
$ istioctl create-remote-secret --context="${CLUSTER1_CTX}" --name=cluster1 \
  | kubectl apply -f - --context="${CLUSTER2_CTX}"

# For cluster2 → cluster1
$ istioctl create-remote-secret --context="${CLUSTER2_CTX}" --name=cluster2 \
  | kubectl apply -f - --context="${CLUSTER1_CTX}"
```

7.4. Auto-install via GitLab Repo (Alternative Option)

While the previous sections detail the manual Istio installation process—including control plane setup, certificate provisioning, east-west gateway configuration, and cross-cluster endpoint discovery—the same configuration can be automated using the provided script.

GitLab repository (zta-testbed/5-istio/install-istio.sh) includes an automated script to install Istio across both clusters. This effectively sets up a multi-cluster Istio mesh with east-west gateways and mutual mTLS.

The automation script is located at: https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed.git, under the script 5-istio/install-istio.sh automates the full installation workflow, including:

- Creating necessary namespaces and secrets for mTLS
- Labeling clusters for multi-network awareness
- Installing the Istio control plane in each cluster
- Deploying east-west gateways
- Exposing gateway services and configuring cross-cluster communication
- Exchanging remote secrets for endpoint discovery

7.5. Istio's Add-ons & JWT Authentication Example

7.5.1. Kiali & Grafana

Kiali
Kiali is a management console specifically designed for Istio service mesh. It provides observability into the mesh by visualizing the service topology, traffic flows, and configuration status. Kiali enables operators to quickly identify communication patterns, detect issues such as failed requests or misconfigurations, and validate Istio's configuration. Its graphical interface helps reduce the complexity of managing a large-scale service mesh, making it easier to monitor health, security policies, and performance across microservices.

Grafana
Grafana is an open-source analytics and visualization platform that integrates seamlessly with Prometheus, Istio's primary metrics collection system. Within the Istio environment, Grafana is used to present performance and monitoring dashboards that display metrics such as request volume, latency, error rates, and resource usage. By providing interactive dashboards, Grafana helps operators and developers understand the behavior of services over time, diagnose performance bottlenecks, and support capacity planning.

Note:
Together, Kiali and Grafana enhance the observability and manageability of Istio. Kiali focuses on service mesh topology and configuration insights, while Grafana delivers powerful time-series monitoring and visualization. These add-ons are critical for maintaining reliability, diagnosing problems efficiently, and ensuring that the service mesh operates in a secure and performant manner.

Add-on configurations are included in repo and typically installed via:

```
$ kubectl apply -f samples/addons/kiali.yaml
$ kubectl apply -f samples/addons/grafana.yaml
```

Label the namespace for mesh-injection:

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

7.5.2. JWT Authentication at Ingress

The following configuration enforces JWT-based authentication at the Istio ingress gateway.

7.5.2.1. RequestAuthentication (vs-jwt-httpbin.yaml)

- This resource defines how JWT tokens should be validated.
- It specifies the **issuer** (testing@secure.istio.io) and the **JWKS endpoint** containing the public keys used to verify the token's signature.
- The configuration applies to workloads labeled as istio: ingressgateway within the istio-system namespace.

By applying this resource, the ingress gateway is able to validate the authenticity of incoming JWT tokens. However, validation alone does not enforce token usage—it only allows Istio to recognize and accept valid tokens.

vs-jwt-httpbin.yaml :

```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: ingress-jwt
  namespace: istio-system
```

```
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.26/security/tools/jwt/samples/jwks.json"
```

7.5.2.2. AuthorizationPolicy (ingress-authz-jwt)

- This policy enforces that only requests with a valid JWT token are allowed through the ingress.
- The rules section ensures that traffic is permitted only if the request originates from a principal (requestPrincipals) that matches the configured issuer.
- Without this policy, unauthenticated requests could still be forwarded, even if the JWT is validated.

To require tokens at ingress, also apply this AuthorizationPolicy:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ingress-authz-jwt
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: ALLOW
  rules:
    - from:
        - source:
            requestPrincipals: ["testing@secure.istio.io/*"]
```

This ensures only JWT-authenticated requests reach resources behind the ingress

7.6. Sample App Install & Verification

Referencing repo's scripts:

- install-apps-sample-helloworld.sh deploys a hello server + curl client
- install-apps-sleep_helloworld.sh deploys sleep and hello world apps
- verify-* scripts run basic connectivity tests using curl from one pod to another

Integration steps:

1. **Inject sidecars** and deploy apps in sample namespace

2. **Ensure network connectivity**: curl → hello and httpbin
3. **Verify using verify scripts**:

```
$ ./verify-sample-helloworld.sh
$ ./verify-sleep_helloworld.sh
```

7.7. End-to-End Verification

1. **Deploy** JWT ingress config and authz policy
2. **Apply** downstream HTTP and gRPC VirtualServices pointing to httpbin, hello
3. **Generate a JWT** using the Istio sample keys
4. **Make calls** from cluster2 or external client:

```
$ curl -v \
  --resolve httpbin.sample.svc.cluster.local:80:<INGRESS_IP> \
  -H "Host: httpbin.sample.svc.cluster.local" \
  -H "Authorization: Bearer $JWT" \
  http://httpbin.sample.svc.cluster.local/get
```

Check that:

- 200 OK with valid JWT
- 401 without or with invalid JWT

That wraps up for the testbed: multi-cluster Istio install, gateway setup, monitoring, JWT auth, and sample apps—all tested via integration scripts.

8. Deploying SPIRE on Top of Istio

8.1. Introduction

Istio provides a powerful service mesh that secures service-to-service communication using **mutual TLS (mTLS)** and **SPIFFE identities**. By default, Istio issues its own certificates through Istiod. However, in environments that require a stronger identity management framework or interoperability across heterogeneous systems, integrating **SPIRE (SPIFFE Runtime Environment)** as the workload identity provider offers several advantages:

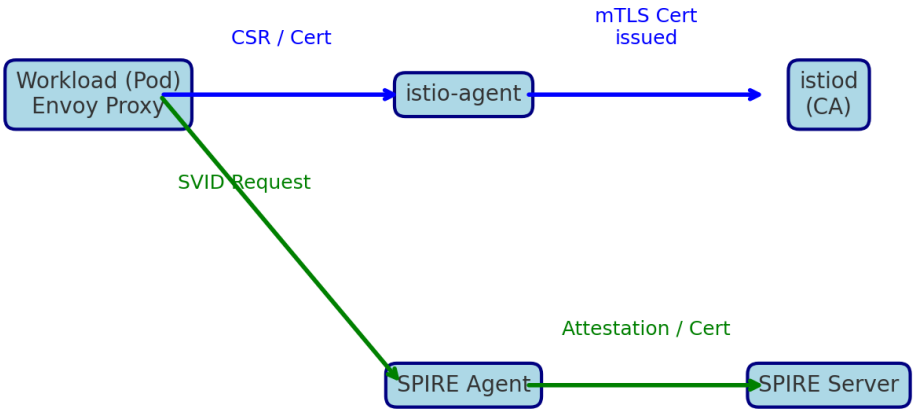
- **Standards Compliance**: SPIRE issues identities following the SPIFFE ID standard, enabling interoperability across meshes, clusters, and external systems.
- **Granular Control**: SPIRE supports flexible attestation mechanisms (e.g., Kubernetes, node, workload attestation).
- **Separation of Concerns**: Decoupling certificate authority (SPIRE) from Istio's control plane allows independent lifecycle management and tighter security boundaries.
- **Federation**: SPIRE supports identity federation between clusters and external domains, which is useful in multi-cluster or hybrid cloud deployments.

By running SPIRE alongside Istio, the Istio proxies (Envoy sidecars and gateways) can consume **SPIFFE IDs** and **X.509 SVIDs (SPIFFE Verifiable Identity Documents)** issued by SPIRE instead of relying solely on Istio's built-in certificate authority.

8.2. Prerequisites

- A running **Kubernetes cluster** with Istio installed.
- kubectl and istioctl configured for the cluster.
- SPIRE Helm charts or deployment manifests.
- Familiarity with Istio's security configuration (RequestAuthentication, AuthorizationPolicy, PeerAuthentication).

8.3. Workload Identity Issuance in Istio: Istiod vs. SPIRE



Workload Identity Issuance: Istiod vs SPIRE

8.3.1. Default Istio Process (via Istiod)

In the default Istio configuration, workload identity and certificates are issued through **istiod** acting as the Certificate Authority (CA).

8.3.2. Steps

1. **Certificate Signing Request (CSR)**
    - The Envoy proxy inside a Pod generates a CSR.
    - The CSR is sent to the **istio-agent** sidecar.
2. **Forwarding to Istiod**
    - The **istio-agent** forwards the CSR to **istiod**.
    - Istiod verifies the workload's Kubernetes service account and namespace to determine its SPIFFE identity (usually in the form spiffe://<trust-domain>/ns/<namespace>/sa/<service-account>).
3. **Certificate Issuance**

- Istiod signs the CSR and returns an **X.509 certificate** (mTLS workload identity certificate) back through the istio-agent.
- The Envoy proxy installs the certificate and private key, enabling mTLS authentication with other workloads.

Key Characteristics

- Identity is tightly coupled to Kubernetes service accounts.
- Istiod acts as both the control plane and the CA.
- Suitable for most Istio deployments, but limited in flexibility and interoperability.

8.3.3. SPIRE-based Process (direct SVID issuance)

When integrating Istio with **SPIRE**, workload identities are issued by the **SPIRE Server** and **SPIRE Agent**, following the SPIFFE standard. In this model, the Envoy proxy bypasses the istio-agent for identity issuance and instead communicates directly with SPIRE components.

8.3.4. Steps

1. **SVID Request**
   - The Envoy proxy requests an **SVID (SPIFFE Verifiable Identity Document)** from the local **SPIRE Agent** running on the same node.
2. **Workload Attestation**
   - The SPIRE Agent performs **workload attestation** to verify the identity of the Pod (e.g., based on Kubernetes service account, labels, or other selectors).
   - The attestation information is sent to the **SPIRE Server**.
3. **SVID Issuance**
   - The SPIRE Server validates the attestation data against its registration entries.
   - If valid, it issues an **X.509 SVID** (certificate + private key) back to the SPIRE Agent.
   - The SPIRE Agent delivers the SVID to the Envoy proxy inside the Pod.
4. **Usage in mTLS**
   - The Envoy proxy now presents the SPIRE-issued SVID during mTLS handshakes with other workloads.
   - Since identities are SPIFFE-compliant, this enables interoperability across clusters and external systems that also trust SPIRE.

Key Characteristics

- SPIRE Server/Agent fully manage workload identities.

- Istiod is no longer the certificate issuer; instead, it consumes SPIFFE IDs for policy enforcement.
- Provides stronger interoperability, federation, and flexibility in defining identities.

8.4. Install SPIRE

Following the instruction on Istio official website, there are detailed procedure descriptions for installing SPIRE and for SPIRE in production environment using helm charts to integrate SPIRE and Istio.

```
# helm upgrade --install -n spire-server spire-crds spire-crds --repo
https://spiffe.github.io/helm-charts-hardened/ --create-namespace

# helm upgrade --install -n spire-server spire spire --repo https://spiffe.github.io/helm-
charts-hardened/ --wait --set global.spire.trustDomain="example.org"
```

This helm charts commands are to install SPIFFE CSI driver, which is used to Envoy-compatible SDS socket into proxies, and install SPIRE Controller Manager, which eases the creation of SPIFFE registrations for workloads.

8.5. Auto registration for gateways

In Istio service mesh, we are using the ingress gateway and the eastwest gateway for inter-communication between clusters, so we need to have a ClusterSPIFFEID manifest to have the SPIRE's Contoller Manager generate SPIFFE id based on the following resources. There are two options to achieve a spffieid registration, in this document the auto-registration is preferred to manipulate the SPIFFE id registration.

The automation is to Eliminate the need to manually create registration entries for each workload. This is especially critical in large-scale Kubernetes clusters where workloads are dynamic and frequently redeployed.

Since Istio proxies (Envoy) rely on SPIFFE IDs for mTLS authentication, automatically managing these IDs through ClusterSPIFFEID ensures seamless integration with Istio's security model.

```
$ Kubectl apply -f - <<EOF
apiVersion: spire.spiffe.io/v1alpha1
kind: ClusterSPIFFEID
metadata:
  name: istio-gateways
spec:
  className: spire-server-spire
  spiffeIDTemplate:
"spiffe://{{ .TrustDomain }}/ns/{{ .PodMeta.Namespace }}/sa/{{ .PodSpec.ServiceAccountName }
}"
  namespaceSelector:
```

```
    matchLabels:
      kubernetes.io/metadata.name: istio-system
  podSelector:
    matchExpressions:
    - key: app
      operator: In
      values: ["istio-ingressgateway","istio-eastwestgateway"]
  workloadSelectorTemplates:
    - "k8s:ns:{{ .PodMeta.Namespace }}"
    - "k8s:sa:{{ .PodSpec.ServiceAccountName }}"
EOF
```

The following manifest defines a **ClusterSPIFFEID** resource, which automatically generates **SPIFFE IDs** for workloads in Kubernetes. Unlike a standard SPIFFEID bound to a specific namespace or workload, ClusterSPIFFEID applies cluster-wide rules.

The next code will create a ClusterSPIFFEID, which is supposed to generate SPIFFEID for any pod in the default namespace.

```
$ kubectl apply -f - <<EOF
apiVersion: spire.spiffe.io/v1alpha1
kind: ClusterSPIFFEID
metadata:
  name: istio-sidecar-reg
spec:
  spiffeIDTemplate:
"spiffe://{{ .TrustDomain }}/ns/{{ .PodMeta.Namespace }}/sa/{{ .PodSpec.ServiceAccountName }
}"
  podSelector:
    matchLabels:
      spiffe.io/spire-managed-identity: "true"
  workloadSelectorTemplates:
    - "k8s:ns:default"
EOF
```

When applied, this resource allows SPIRE to manage SPIFFE ID assignment in a scalable and automated manner. Pods that meet the selector conditions will automatically receive SPIFFE IDs without requiring individual registration entries.

8.6. Update or Re-Install Istio

Create the Istio configuration with custom patches for the Ingress Gateway and istio-proxy. The Ingress Gateway component includes the spiffe.io/spire-managed-identity: "true" label. The following code is the recommendation from the Istio's official website explaining to update Istio deployment over SPIRE.

8.6.1. Update for Ingress Gateway

The Istio ingress gateway must be updated to integrate with **SPIRE** for workload identity management. This ensures that the gateway itself obtains an SVID (SPIFFE Verifiable Identity Document) directly through the SPIRE Agent using the CSI driver.

8.6.2. Key Updates in IstioOperator Manifest

1. **Mesh Configuration**
    - trustDomain: example.org
    - Sets the trust domain for SPIFFE identities across the mesh.
2. **Sidecar Injector Webhook Customization**
    - Adds a label spiffe.io/spire-managed-identity: "true" so that SPIRE manages the pod identity.
    - Configures CSI driver volume mounts under /run/secrets/workload-spiffe-uds for Envoy (istio-proxy) to retrieve SVIDs.
3. **Ingress Gateway Component Update**
    - Modifies the istio-ingressgateway deployment template to:
        - Include CSI volume mounts for the SPIRE socket.
        - Add an init container to delay startup until the CSI driver has mounted the socket.
    - Ensures that Envoy within the ingress gateway can fetch certificates directly from SPIRE instead of Istiod.

```
cat <<EOF > ./iop-ingress-fromOfficial.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  namespace: istio-system
spec:
  profile: default
  meshConfig:
    trustDomain: example.org
  values:
    # This is used to customize the sidecar template.
    # It adds both the label to indicate that SPIRE should manage the
    # identity of this pod, as well as the CSI driver mounts.
    sidecarInjectorWebhook:
      templates:
        spire: |
          labels:
            spiffe.io/spire-managed-identity: "true"
          spec:
            containers:
            - name: istio-proxy
```

```
              volumeMounts:
              - name: workload-socket
                mountPath: /run/secrets/workload-spiffe-uds
                readOnly: true
            volumes:
              - name: workload-socket
                csi:
                  driver: "csi.spiffe.io"
                  readOnly: true
  components:
    ingressGateways:
      - name: istio-ingressgateway
        enabled: true
        label:
          istio: ingressgateway
        k8s:
          overlays:
            # This is used to customize the ingress gateway template.
            # It adds the CSI driver mounts, as well as an init container
            # to stall gateway startup until the CSI driver mounts the socket.
            - apiVersion: apps/v1
              kind: Deployment
              name: istio-ingressgateway
              patches:
                - path: spec.template.spec.volumes.[name:workload-socket]
                  value:
                    name: workload-socket
                    csi:
                      driver: "csi.spiffe.io"
                      readOnly: true
                - path: spec.template.spec.containers.[name:istio-
proxy].volumeMounts.[name:workload-socket]
                  value:
                    name: workload-socket
                    mountPath: "/run/secrets/workload-spiffe-uds"
                    readOnly: true
                - path: spec.template.spec.initContainers
                  value:
                    - name: wait-for-spire-socket
                      image: busybox:1.36
                      volumeMounts:
                        - name: workload-socket
                          mountPath: /run/secrets/workload-spiffe-uds
                          readOnly: true
                      env:
                        - name: CHECK_FILE
                          value: /run/secrets/workload-spiffe-uds/socket
                      command:
                        - sh
                        - "-c"
                        - |-
                          echo "$(date -Iseconds)" Waiting for: ${CHECK_FILE}
                          while [[ ! -e ${CHECK_FILE} ]] ; do
                            echo "$(date -Iseconds)" File does not exist: ${CHECK_FILE}
                            sleep 15
                          done
                          ls -l ${CHECK_FILE}
EOF
```

IstioOperator spec from iop-ingress-fromOfficial.yaml to the cluster, reconciling Istio components defined in the file (here: the ingress gateway and related mesh settings).

The following command Performs an in-place install/upgrade: if Istio (or the gateway) already exists, it updates it to match the manifest; if not, it creates it.

```
$ istioctl install --skip-confirmation -f ./iop-ingress-fromOfficial.yaml
```

8.6.3. Update for EastWest Gateway on Cluster1

The East-West Gateway on Cluster1 must be updated to align with the **SPIRE-managed trust domain** and ensure proper service discovery across multiple clusters. This configuration enables secure cross-cluster communication by ensuring that certificates and trust domains are consistent.

```
cat <<EOF > ./iop-ew-cluster1.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: eastwest
  namespace: istio-system
spec:
  meshConfig:
    trustDomain: example.org
    trustDomainAliases:
      - cluster.local
  profile: empty
  components:
    ingressGateways:
    - name: istio-eastwestgateway
      enabled: true
      label:
        istio: eastwestgateway
        app: istio-eastwestgateway
        topology.istio.io/network: network1
      k8s:
        env:
        - name: ISTIO_META_REQUESTED_NETWORK_VIEW
          value: network1
        service:
          type: LoadBalancer
          ports:
          - name: status-port
            port: 15021
            targetPort: 15021
          - name: tls
            port: 15443
            targetPort: 15443
          - name: tls-istiod
            port: 15012
            targetPort: 15012
          - name: tls-webhook
            port: 15017
            targetPort: 15017
        overlays:
```

```
        - apiVersion: apps/v1
          kind: Deployment
          name: istio-eastwestgateway
          patches:
          # 1) SPIRE UDS volume added
          - path: spec.template.spec.volumes.[name:workload-socket]
            value:
              name: workload-socket
              csi:
                driver: "csi.spiffe.io"
                readOnly: true
          # 2) Envoy(istio-proxy) mount
          - path: spec.template.spec.containers.[name:istio-
proxy].volumeMounts.[name:workload-socket]
            value:
              name: workload-socket
              mountPath: "/run/secrets/workload-spiffe-uds"
              readOnly: true
          # 3) (Optinoal) waiting for socket
          - path: spec.template.spec.initContainers
            value:
            - name: wait-for-spire-socket
              image: busybox:1.36
              volumeMounts:
              - name: workload-socket
                mountPath: /run/secrets/workload-spiffe-uds
                readOnly: true
              env:
              - name: CHECK_FILE
                value: /run/secrets/workload-spiffe-uds/socket
              command:
              - sh
              - "-c"
              - |
                echo "$(date -Iseconds)" Waiting for: ${CHECK_FILE}
                while [[ ! -S ${CHECK_FILE} ]]; do
                  echo "$(date -Iseconds)" File does not exist: ${CHECK_FILE}
                  sleep 15
                done
                ls -l ${CHECK_FILE}

  values:
    global:
      network: network1
```

```
$ istioctl install --skip-confirmation -f ./iop-ew-cluster1.yaml
```

8.7. Application Test for SVID issued by SPIRE

This section demonstrates how to deploy a simple curl application in the SPIRE-integrated Istio environment to verify that the Pod is automatically issued an **SVID (SPIFFE Verifiable Identity Document)**.

8.7.1. Deployment Steps

1. **Service Account**
    - A dedicated Kubernetes ServiceAccount named curl is created.
    - This account will be associated with the curl Pod to bind identity management.
2. **Service**
    - A Kubernetes Service is defined to expose the curl Pod on port 80.
    - This enables basic service discovery and allows Istio sidecar injection to work seamlessly.
3. **Deployment**
    - A Deployment named curl is created with the following key configurations:
        - **Labels and MatchLabels**: Ensure proper Pod selection and Istio injection.
        - **SPIRE Identity Label**: Signals SPIRE to manage the Pod identity.
        - **Sidecar Injection Annotation**: Ensures the Envoy sidecar is customized to retrieve SVID from SPIRE instead of Istiod.
        - **ServiceAccountName**: Links the Pod with the curl service account.
        - **Container**: Runs the curlimages/curl image with a simple infinite sleep loop, so that the Pod remains active and available for interactive testing.

Install curl on the SPIRE environment

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: curl
---
apiVersion: v1
kind: Service
metadata:
  name: curl
  labels:
    app: curl
    service: curl
spec:
  ports:
  - port: 80
    name: http
  selector:
    app: curl
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: curl
  template:
    metadata:
      labels:
        app: curl
        spiffe.io/spire-managed-identity: "true"
      # Injects custom sidecar template
      annotations:
        inject.istio.io/templates: "sidecar,spire"
    spec:
      terminationGracePeriodSeconds: 0
      serviceAccountName: curl
      containers:
      - name: curl
        image: curlimages/curl
        command: ["/bin/sleep", "infinity"]
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        securityContext:
          runAsUser: 1000
      volumes:
      - name: tmp
        emptyDir: {}
```

Install httpbin on the SPIRE environment

```
apiVersion: v1
kind: Namespace
metadata:
  name: sample
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
  namespace: sample
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  namespace: sample
  labels:
    app: httpbin
spec:
  selector:
    app: httpbin
  ports:
  - name: http
    port: 80
    targetPort: 8080
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
```

```
  namespace: sample
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
  template:
    metadata:
      labels:
        app: httpbin
        spiffe.io/spire-managed-identity: "true"     # SPIRE management ID
      annotations:
        inject.istio.io/templates: "sidecar,spire"   # SPIRE template
    spec:
      serviceAccountName: httpbin
      containers:
      - name: httpbin
        image: mccutchen/go-httpbin:v2.10.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        securityContext:
          runAsUser: 1000
      volumes:
      - name: tmp
        emptyDir: {}
```

8.7.2. Check certs and SPIFFE ID

After deploying the httpbin workload with SPIRE integration, we can verify that the sidecar proxy (istio-proxy) has successfully obtained an **SVID**.

1. Verification Command

```
$ kubectl -n sample exec deploy/httpbin -c istio-proxy -- \
  curl -s http://127.0.0.1:15000/certs \
| jq -r '.certificates[].cert_chain[]?.subject_alt_names[]?.uri // empty'
```

2. Expected Output

```
spiffe://example.org/ns/sample/sa/httpbin
```

This confirms that:

- The **SPIRE Agent** mounted via the CSI driver has issued a valid certificate to the Envoy proxy.
- The certificate contains a **SPIFFE URI** in the Subject Alternative Name (SAN) field.

- The identity matches the namespace (sample) and ServiceAccount (httpbin) of the workload.

3. Pod Status Overview

Running the following command shows the state of all Pods in the cluster:

- Verifies that workloads (httpbin, curl) have been issued SPIFFE IDs via SPIRE.
- Confirms that control plane components (Istio + SPIRE) are healthy and functional.
- Ensures the environment is ready for mTLS communication between workloads using SPIFFE identities.

```
$ kubectl get po -A
NAMESPACE        NAME                                                READY   STATUS
default          curl-6d6946c997-2bsgd                               2/2     Running
istio-system     istio-eastwestgateway-d78d7bc65-h5jjh               1/1     Running
istio-system     istio-ingressgateway-79d5d5f8c9-7v4bl               1/1     Running
istio-system     istiod-6cc57f9897-gjh7c                             1/1     Running
kube-system      cilium-operator-ddb9b866-h4nmd                      1/1     Running
kube-system      cilium-w5sxn                                        1/1     Running
kube-system      coredns-7c65d6cfc9-h5gc7                            1/1     Running
kube-system      coredns-7c65d6cfc9-m7vqk                            1/1     Running
kube-system      etcd-clu1                                           1/1     Running
kube-system      kube-apiserver-clu1                                 1/1     Running
kube-system      kube-controller-manager-clu1                        1/1     Running
kube-system      kube-proxy-787mt                                    1/1     Running
kube-system      kube-scheduler-clu1                                 1/1     Running
metallb-system   controller-5456bd6d98-rc525                         1/1     Running
metallb-system   speaker-zc52g                                       1/1     Running
sample           httpbin-7c79fc6cb5-dqhmq                            2/2     Running
spire-server     spire-agent-mvtmf                                   1/1     Running
spire-server     spire-server-0                                      2/2     Running
spire-server     spire-spiffe-csi-driver-z2r26                       2/2     Running
spire-server     spire-spiffe-oidc-discovery-provider-85c8b97bf4-glv55   2/2     Running
```

8.7.3. Result: Application-to-Application Request with SPIRE-Issued SVID

A request was made from the curl Pod to the httpbin service inside the mesh:

```
$ kubectl exec deploy/curl -- curl -sS http://httpbin.sample.svc.cluster.local:80/get
```

8.7.4. Key Observations from the Response

1. Headers Section
    - Host: httpbin.sample.svc.cluster.local → Confirms request routing within the service mesh.

- User-Agent: curl/8.16.0 → Shows the client application (curl) initiating the request.
- X-Envoy-Attempt-Count: 1 → Indicates that the Istio sidecar proxy (Envoy) handled the request successfully on the first attempt.

2. mTLS Identity Evidence
   - In the field X-Forwarded-Client-Cert, the certificate chain is shown:

     ```
     By=spiffe://example.org/ns/sample/sa/httpbin;
     URI=spiffe://example.org/ns/default/sa/curl
     ```
   - This demonstrates that both workloads presented valid SPIRE-issued SVIDs during the TLS handshake.
3. Additional Metadata
   - X-Forwarded-Proto: http → Traffic was forwarded using HTTP over the sidecar.
   - X-Request-Id: d32f4a3e-2323-4b3c-8b40-87dcb4e6fb5e → Unique identifier assigned to the request for tracing.
   - origin: 127.0.0.6:42199 → Source of the request from inside the Pod.

```
$ kubectl exec deploy/curl -- curl -sS http://httpbin.sample.svc.cluster.local:80/get
{
  "args": {},
  "headers": {
    "Accept": [
      "*/*"
    ],
    "Host": [
      "httpbin.sample.svc.cluster.local"
    ],
    "User-Agent": [
      "curl/8.16.0"
    ],
    "X-Envoy-Attempt-Count": [
      "1"
    ],
    "X-Forwarded-Client-Cert": [

"By=spiffe://example.org/ns/sample/sa/httpbin;Hash=e9f8de7bbbfd6017fbae402daab3fb17ef8f93c69
6ce840dd636d1919e80395a;Subject=\"O=SPIRE,C=US\";URI=spiffe://example.org/ns/default/sa/curl
"
    ],
    "X-Forwarded-Proto": [
      "http"
    ],
    "X-Request-Id": [
      "d32fa43e-2323-4b3c-8b40-87dcb4e6fb5e"
    ]
  },
  "method": "GET",
  "origin": "127.0.0.6:42199",
  "url": "http://httpbin.sample.svc.cluster.local/get"
}
```

-
June 2025

References

[1] https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed/-/tree/main?ref_type=heads
[2] https://gitlab.nist.gov/kyehwanl/ric-servicemesh-poc/-/tree/master?ref_type=heads
[3] https://istio.io/latest/docs/tasks/security/authentication/jwt-route/
[4] https://istio.io/latest/docs/tasks/observability/kiali/
[5] https://docs.o-ran-sc.org/projects/o-ran-sc-ric-app-hw-go/en/latest/onboard-and-deploy.html