NIST Technical Note

-

ZTA Testbed Deployment

Kubernetes Clusters with 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 and JWT-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):

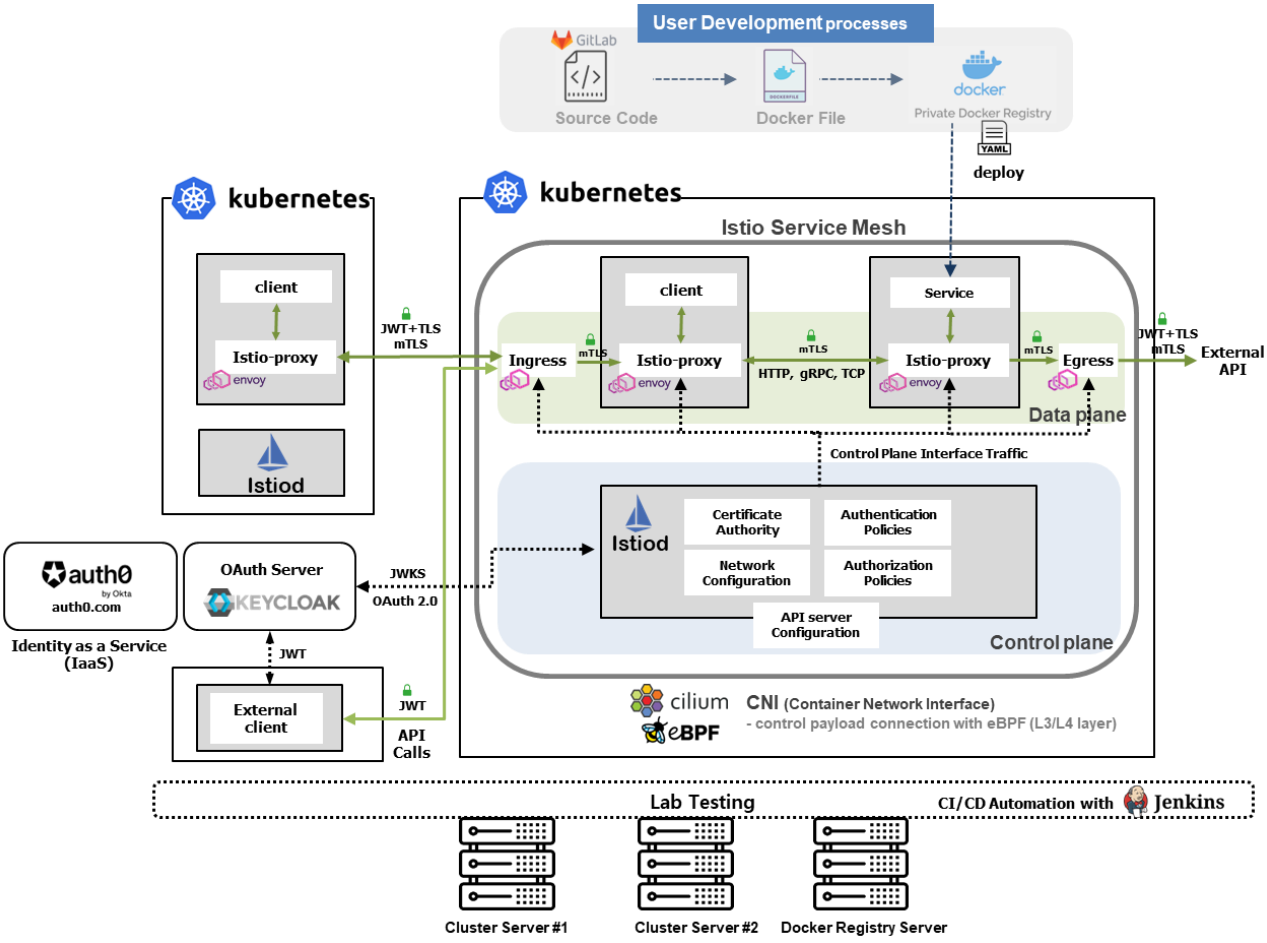
```
sudo apt update && sudo apt install -y git curl make
```

Docker and Kubernetes tools should be installed according to the cloud provider or on-premise 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.

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

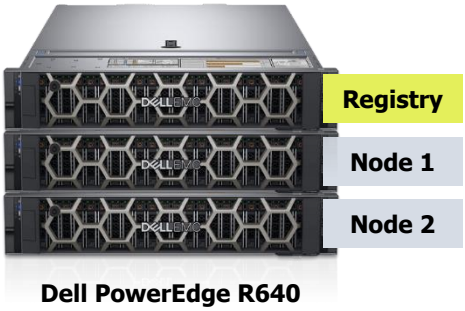
Each server used in this testbed shares the following hardware configuration, ensuring consistent compute and network capabilities across the nodes:

- 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 CSR (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
```

openssl command 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
```

Certs are located in /home/onfadmin/docker-registry on 5g1-comp1 server. Those certs are used for running 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. Docker upload (push), download (pull) to/from the private docker repository

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

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"]
```

Docker build

```
onfadmin@5g1-comp2:~/$ docker build -t flask-hello ./
…
Step 5/5 : CMD ["python", "app.py"]
 ---> Running in 9923e41b1563
 ---> Removed intermediate container 9923e41b1563
 ---> d903f11664d6
Successfully built d903f11664d6
Successfully tagged flask-hello:latest
```

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

```
onfadmin@5g1-comp2:~/$ docker tag flask-hello:latest 10.5.0.2:8443/flask-hello
onfadmin@5g1-comp2:~/$ docker push 10.5.0.2:8443/flask-hello
Using default tag: latest
The push refers to repository [10.5.0.2:8443/flask-hello]
…
ace34d1d784c: Pushed
```

```
latest: digest: sha256:a6540fe78ac6ef198c7ecebea63378aea0ef12709aa0db7084c45ba5fb8b2e89
size: 1784
```

To check the status,

```
onfadmin@5g1-comp2:~$ curl -k https://10.5.0.2:8443/v2/_catalog
{"repositories":["flask-hello"]}
```

3.5.2. 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

```
onfadmin@5g1-comp2:~/$ kubectl get po -n sample -l app=flask-hello
NAME                           READY   STATUS    RESTARTS   AGE
flask-hello-64576d5bf-x9g5w   2/2     Running   0          28d
```

3.5.3. To check the application running,

```
onfadmin@5g1-comp2:$ 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.

Conclusion:
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.

Troubleshoot

if there is "tls: failed to verify certificate: x509: certificate signed by unknown authority" from 'kubectl get events -w'

```
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
```

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,

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

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

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

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

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

```
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
```

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,

```
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 used)
- 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**.

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.

The automation script is located at: https://gitlab.nist.gov/gitlab/kyehwanl/zta-testbed.git

Specifically, 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

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. Auto-install via GitLab Repo

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.

7.4. Manual Installation (Alternative Option)

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.

7.4.1. Environment Setup

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

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

7.4.2. Certificate Installation for mTLS

Istio is configured with custom root and intermediate certificates to support mutual TLS (mTLS) authentication between services.

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}
```

7.4.3. Network Labeling

Each cluster's Istio namespace is labeled to indicate its logical network name for multi-network configuration.

```
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
```

7.4.4. Istio Control Plane Installation

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
```

7.4.5. East-West Gateway Deployment

To enable service discovery and communication between clusters, an Istio east-west gateway is deployed in each cluster.

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
```

7.4.6. Gateway Resource Configuration

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

7.4.7. Endpoint Discovery Setup

To enable Istio control planes to discover services in remote clusters, remote Kubernetes secrets are exchanged:

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

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

7.5. Add-ons & JWT Authentication Example

7.5.1. Kiali & Grafana

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

Using this file, **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"
```

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 helloworld 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.

-
June 2025

8. Jenkins Automation (TBD)

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