

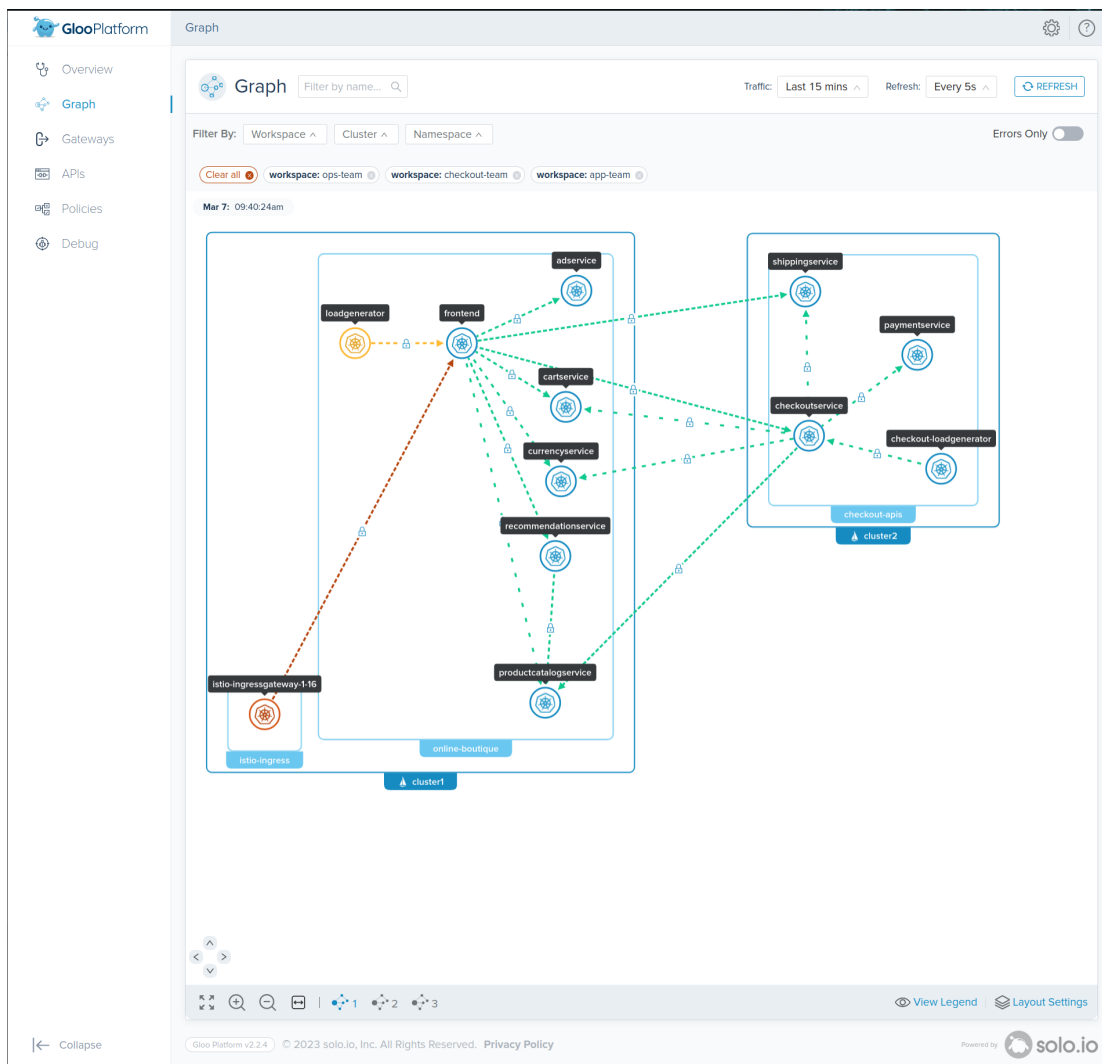
Proof of Concept

Prepared for Texas Capital Bank

Table of Contents

- [Introduction](#)
- [Lab 1 - Deploy EKS clusters](#)
- [Lab 2 - Deploy Gloo Platform](#)
- [Lab 3 - Deploy Istio](#)
- [Lab 4 - Deploy Gloo Platform Addons](#)
- [Lab 5 - Certificate Management](#)
- [Lab 6 - Deploy Online Boutique](#)
- [Lab 7 - Configure Gloo Platform](#)
- [Lab 8 - Ingress](#)
- [Lab 9 - Zero Trust Communication](#)
- [Lab 10 - Multi Cluster Secure Communication](#)
- [Lab 11 - Observability](#)
- [Lab 12 - Expose APIs](#)
- [Lab 13 - Circuit Breaking and Failover](#)
- [Lab 14 - Rate Limiting](#)
- [Lab 15 - Authentication / JWT + JWKS](#)
- [Lab 16 - Gloo Platform OPA Integration](#)
- [Lab 17 - Calling External Services](#)
- [Lab 18 - Day 2 Certificates](#)
- [Lab 19 - Zero Downtime Istio Upgrades](#)
- [Lab 20 - POC Clean Up](#)

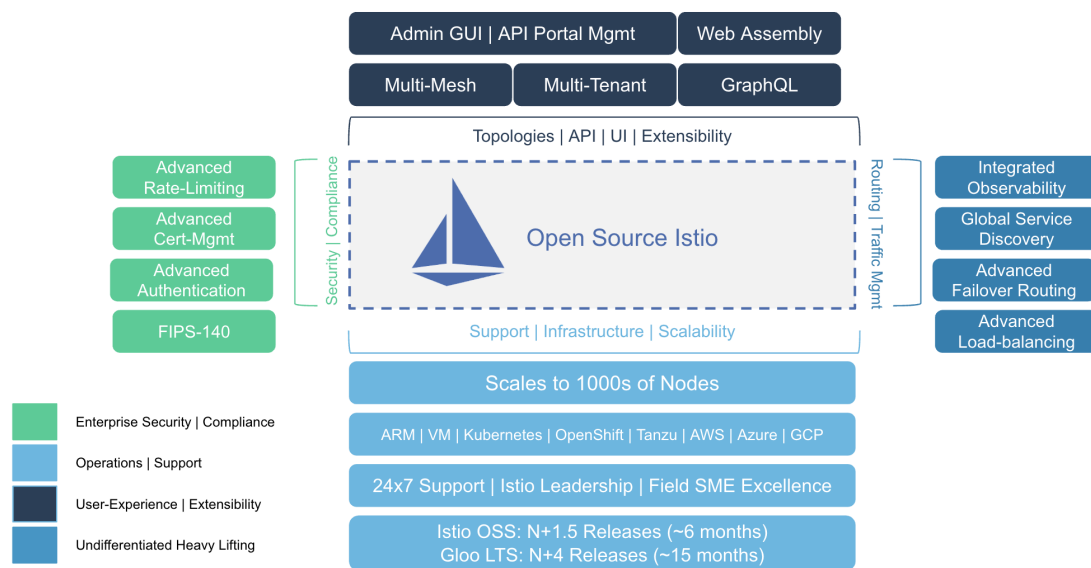
Introduction



[Gloo Platform](#) integrates API gateway, API management, Istio service mesh and cloud-native networking into a unified application networking platform. It allows you to manage gateway and service mesh together across single or multiple clusters and multiple teams.

Gloo Platform can help solve some of these challenges:

- Zero trust security architecture for APIs and microservices
- Combine north-south and east-west traffic management (API gateway + service mesh)
- Unified failover and security policy across gateway and mesh
- Aligning with the leading approaches to software development
- Scale across multiple dimensions (multi-cluster, multi-tenant, multi-cloud)



Before You Start

Before starting this POC workshop, it is important that you setup with the right components and tooling in place to ensure success.

Supporting Tools

The below tools are designed to help you understand and debug your environment.

- istioctl - Istio CLI `curl -L https://istio.io/downloadIstio | sh -`
- helm v3 - [Helm CLI](#)
- curl - <https://everything.curl.dev/get>
- docker - [Docker CLI](#)

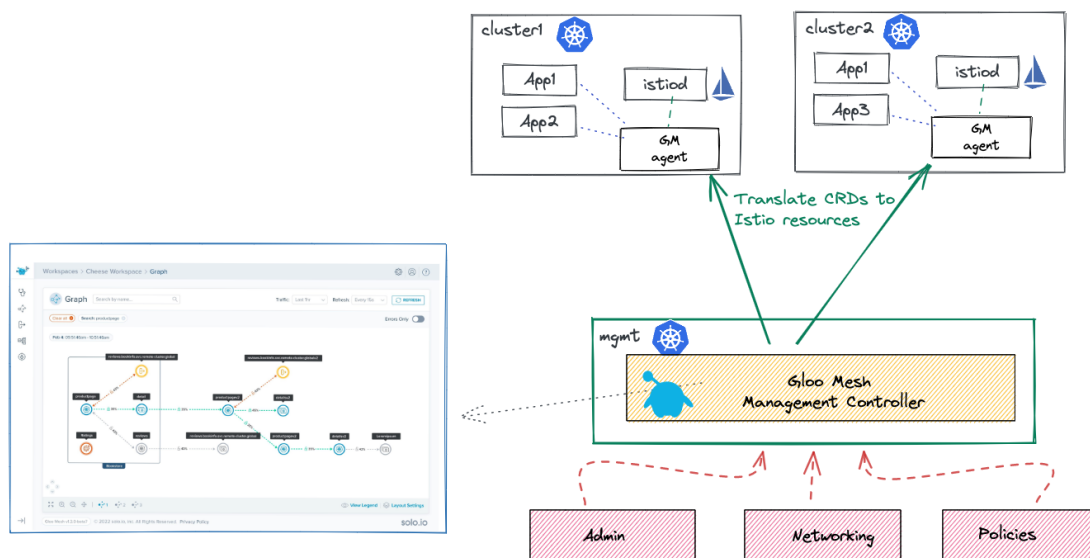
Documentation and Examples

- Gloo Platform Docs - <https://docs.solo.io/gloo-mesh-enterprise/>

Solo.io documentation is great for getting to know the Gloo Platform. From examples to API documentation, you can quickly search and find help. It is recommended that you read through the Gloo Platform concepts before getting started.

- Gloo Platform Concepts - <https://docs.solo.io/gloo-mesh-enterprise/latest/concepts/>
- API Reference - <https://docs.solo.io/gloo-mesh-enterprise/latest/reference/api/>

Cluster Setup



This POC depends on our typical 3 *cluster environment* where one cluster will be used for the administration (management) of the other two clusters (cluster-1 and cluster-2), which run the gateway, service mesh and your workloads.

The best way to facilitate the POC is by creating 3 new Kubernetes clusters in your environment so that you can test Gloo Platform without impacting other teams and workloads. Although, most prospects will want to test that Gloo Platform works in their "real" environments, its important for users to understand Gloo Platform first. This not only gives the prospect time to learn Gloo Platform but also helps the Solo.io Architects understand the requirements of your environment. Prospects are welcome to deploy Gloo Platform to their "real" environments once they have a good grasp of the operational and architectural impacts it will have on their environment.

If the POC has to be performed on existing shared clusters there are some things to be aware of.

Networking

- The 3 clusters MUST have network connectivity with each other (internally or externally) via TCP with TLS.
- The management cluster must be allowed to accept mTLS traffic without termination on ports 9900 and 4317. The preferred way to expose these ports is via **TCP Passthrough** load balancers.
 - Optionally if `NodePorts` are required due to the lack of load balancer support, the prospect must be aware of the Node IP addresses it gives each Gloo Agent. There is a chance for the Node to be recreated with a different IP address and thus would drop the connection from the Gloo Agent.

Sizing

- Each cluster should be able to connect to the LoadBalancer address attached to the other clusters by either internal or external networking.
- Minimum Cluster Resource Sizing
 - Management Cluster (mgmt):
 - Nodes: 2
 - CPU Per Node: 4vCPU

- Memory Per Node: 16Gi
- Each Workload Cluster (cluster-1 and cluster-2):
 - Nodes: 2
 - CPU Per Node: 4vCPU
 - Memory Per Node: 16Gi

Private Image Repository

Some organizations do not allow public docker repository access and need to download the images and upload them to a private repository.

- To view the images that are used in this POC, run `cat images.txt`

The easiest way to get the images in your repository is to pull the listed images, retag them to your internal registry and push.

Helm charts

The following helm charts are used in this POC or may be needed depending on your use cases.

```
# required
helm repo add gloo-platform https://storage.googleapis.com/gloo-platform/helm-charts
helm pull oci://us-central1-docker.pkg.dev/solo-test-236622/solo-demos/onlineboutique

# Optional addons
helm repo add istio https://istio-release.storage.googleapis.com/charts
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

Lab 01 - Deploy EKS clusters

Deploying 3 EKS clusters using eksctl



[eksctl](#) is a useful tool in creating EKS clusters. This guide will show you how to create 3 working clusters to use for this demo.

- Deploy the first cluster

```
eksctl create cluster -f data/cluster-1.yaml
```

- Deploy the second cluster

```
eksctl create cluster -f data/cluster-2.yaml
```

- Deploy the third and final cluster

```
eksctl create cluster -f data/management.yaml
```

- Update the kubernetes context names

```
export AWS_USER=<user>
kubectl config rename-context $AWS_USER@cluster-1.us-east-2.eksctl.io cluster-1
kubectl config rename-context $AWS_USER@cluster-2.us-west-2.eksctl.io cluster-2
kubectl config rename-context $AWS_USER@management.us-east-2.eksctl.io management
```

Install AWS [Load Balancer Controller](#)

The AWS Load Balancer Controller manages AWS Elastic Load Balancers for a Kubernetes cluster. The controller provisions the following resources:

- An AWS Network Load Balancer (NLB) when you create a Kubernetes service of type LoadBalancer.
- Create IAM Policy to allow service accounts to create aws load balancers

```
curl -o iam_policy.json https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.4.7/docs/install/iam_policy.json
```

```
aws iam create-policy \
  --policy-name AWSLoadBalancerControllerIAMPolicy \
  --policy-document file://iam_policy.json
```

- Enable OIDC provider for service accounts

```
eksctl utils associate-iam-oidc-provider --region=us-east-2 --cluster=cluster-1 --approve
eksctl utils associate-iam-oidc-provider --region=us-west-2 --cluster=cluster-2 --approve
eksctl utils associate-iam-oidc-provider --region=us-east-2 --cluster=management --approve
```

- Create a Kubernetes service account named aws-load-balancer-controller in the kube-system namespace for the AWS Load Balancer Controller and annotate the Kubernetes service account with the name of the IAM role.

```
export AWS_ACCOUNT_ID=<account_id>
eksctl create iamserviceaccount \
  --cluster=cluster-1 \
  --region us-east-2 \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --attach-policy-arn=arn:aws:iam::$AWS_ACCOUNT_ID:policy/AWSLoadBalancerControllerIAMPolicy \
  --override-existing-serviceaccounts \
  --approve

eksctl create iamserviceaccount \
  --cluster=cluster-2 \
  --region us-west-2 \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --attach-policy-arn=arn:aws:iam::$AWS_ACCOUNT_ID:policy/AWSLoadBalancerControllerIAMPolicy \
  --override-existing-serviceaccounts \
  --approve
```

```
eksctl create iamserviceaccount \
  --cluster=management \
  --region us-east-2 \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --attach-policy-arn=arn:aws:iam::$AWS_ACCOUNT_ID:policy/AWSLoadBalancerControllerIAMPolicy \
  --override-existing-serviceaccounts \
  --approve
```

- Install load balancer controller to each cluster

```
helm repo add eks https://aws.github.io/eks-charts
helm repo update
```

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --kube-context cluster-1 \
  --set clusterName=cluster-1 \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --kube-context cluster-2 \
  --set clusterName=cluster-2 \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --kube-context management \
  --set clusterName=management \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

Lab 02 - Deploy Gloo Platform

Reference Links:

- [Gloo Platform Setup Docs](#)
- [Gloo Platform Best practices for production](#)
- [Gloo Platform Architecture](#)

Gloo Platform can be installed using in many ways (helm, terraform, etc). We recommend Helm for the majority of the use cases.

Architecture

At its core, the Gloo Platform is a simple deployment of a management server and an agent on each workload cluster. Optionally, you can deploy additional components such as external authorization, rate limiting, and Istio lifecycle manager. This workshop will utilize these add-ons to highlight the features they provide on top of the Gloo Platform.



Gloo Platform Components

[Low-level architecture](#) - More in-depth diagram showing the communication between the components running in the clusters.

Setup

- A licence key is needed to install the Gloo Platform. If you do not have a license key, contact an account representative.

```
export GLOO_PLATFORM_LICENSE_KEY=<licence_key>
```

- Add helm repository got the Gloo Platform

```
helm repo add gloo-platform https://storage.googleapis.com/gloo-platform/helm-charts
helm repo update
```

Install/Configure the Management Plane

Gloo Platform can be installed a number of ways but the most common is via `helm`. In this workshop `helm` will be the primary way to deploy assets. In production it is recommended that Gloo Platform is deployed via your CI/CD pipeline. Some popular methods include ArgoCD or Flux.

- Create the `gloo-mesh` namespace in the management cluster.

```
kubectl create namespace gloo-mesh --context management
```

- Install Gloo Platform. This command installs the management plane components, such as the management server, UI and Prometheus server.

```
# helm show values gloo-platform/gloo-platform --version 2.3.9 > gloo-platform-values.yaml
helm upgrade -i gloo-platform-crds gloo-platform/gloo-platform-crds \
  --version=2.3.9 \
  --kube-context management \
  --namespace=gloo-mesh
```

```
helm upgrade -i gloo-platform gloo-platform/gloo-platform \
  --version=2.3.9 \
  --namespace=gloo-mesh \
  --kube-context management \
  --set licensing.glooMeshLicenseKey=$GLOO_PLATFORM_LICENSE_KEY \
  --set licensing.glooTrialLicenseKey=$GLOO_PLATFORM_LICENSE_KEY \
  --set licensing.glooGatewayLicenseKey=$GLOO_PLATFORM_LICENSE_KEY \
  -f data/gloo-mgmt-values.yaml
```

- Confirm that Deployments are Ready (1/1) in the gloo-mesh namespace:

```
kubectl get deploy --context management -n gloo-mesh
```

- Verify that the management server is up and running and the logs look good

```
kubectl logs deploy/gloo-mesh-mgmt-server --context management -n gloo-mesh
```


- Now grab the external endpoint of the management server so that the agents can connect to it.
We also will grab a token and root TLS certificate for agents to connect to it with.

```
# wait for the load balancer to be provisioned
until kubectl get service/gloo-mesh-mgmt-server --output=jsonpath='{.status.loadBalancer}' --
context management -n gloo-mesh | grep "ingress"; do : ; done
until kubectl get service/gloo-telemetry-gateway --output=jsonpath='{.status.loadBalancer}' --
context management -n gloo-mesh | grep "ingress"; do : ; done
GLOO_PLATFORM_SERVER_DOMAIN=$(kubectl get svc gloo-mesh-mgmt-server --context management -n gloo-
mesh -o jsonpath='{.status.loadBalancer.ingress[0].*}')
GLOO_PLATFORM_SERVER_ADDRESS=$(GLOO_PLATFORM_SERVER_DOMAIN):$(kubectl get svc gloo-mesh-mgmt-server
--context management -n gloo-mesh -o jsonpath='{.spec.ports[?(@.name=="grpc")].port}')
GLOO_TELEMETRY_GATEWAY=$(kubectl get svc gloo-telemetry-gateway --context management -n gloo-mesh -
o jsonpath='{.status.loadBalancer.ingress[0].*}')$(kubectl get svc gloo-telemetry-gateway --
context management -n gloo-mesh -o jsonpath='{.spec.ports[?(@.port==4317)].port}')

echo "Mgmt Plane Address: $GLOO_PLATFORM_SERVER_ADDRESS"
echo "Metrics Gateway Address: $GLOO_TELEMETRY_GATEWAY"
```

- Before installing Gloo agents on the workload clusters trust needs to be established. The following command copies the public TLS root certificate and a token used for the Gloo agents to authenticate with the management plane.

```
kubectl get secret relay-root-tls-secret --context management -n gloo-mesh -o
jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
kubectl get secret relay-identity-token-secret --context management -n gloo-mesh -o
jsonpath='{.data.token}' | base64 -d > token
```

- To allow a Gloo agent to connect the management plane must be made aware of its existence. Run the following command to notify the management plane that the remote clusters will connect soon.

```
kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: KubernetesCluster
metadata:
  name: cluster-1
  namespace: gloo-mesh
spec:
  clusterDomain: cluster.local
---
apiVersion: admin.gloo.solo.io/v2
kind: KubernetesCluster
metadata:
  name: cluster-2
  namespace: gloo-mesh
spec:
  clusterDomain: cluster.local
EOF
```

Install Gloo Agent on Cluster: cluster-1

By simply installing the Gloo Platform Agent on a remote cluster you gain the ability to manage it with Gloo Platform. Initially, the Gloo Agent is non-invasive and simply relays service discovery information to the Management Plane.

- Create the `gloo-mesh` namespace in cluster cluster-1

```
kubectl create namespace gloo-mesh --context cluster-1
```

- Add the authentication token and root TLS certificate

```
kubectl create secret generic relay-root-tls-secret --from-file ca.crt=ca.crt --context cluster-1 -n gloo-mesh
kubectl create secret generic relay-identity-token-secret --from-file token=token --context cluster-1 -n gloo-mesh
```

- Install the Gloo Agent in cluster-1

```
helm upgrade -i gloo-platform-crds gloo-platform/gloo-platform-crds \
  --version=2.3.9 \
  --namespace=gloo-mesh \
  --kube-context cluster-1

helm upgrade -i gloo-agent gloo-platform/gloo-platform \
  --version=2.3.9 \
  --namespace gloo-mesh \
  --kube-context cluster-1 \
  --set glooAgent.relay.serverAddress=$GLOO_PLATFORM_SERVER_ADDRESS \
  --set common.cluster=cluster-1 \
  --set telemetryCollector.config.exporters.otlp.endpoint=$GLOO_TELEMETRY_GATEWAY \
  -f data/gloo-agent-values.yaml
```

- Verify the `gloo-mesh-agent` logs

```
kubectl logs deploy/gloo-mesh-agent --context cluster-1 -n gloo-mesh
```

- Verify the `gloo-telemetry-collector` logs

```
kubectl logs ds/gloo-telemetry-collector-agent --context cluster-1 -n gloo-mesh
```

Install Gloo Agent on Cluster: cluster-2

- Create the `gloo-mesh` namespace in cluster cluster-2

```
kubectl create namespace gloo-mesh --context cluster-2
```

- Add the authentication token and root TLS certificate

```
kubectl create secret generic relay-root-tls-secret --from-file ca.crt=ca.crt --context cluster-2 -n gloo-mesh
kubectl create secret generic relay-identity-token-secret --from-file token=token --context cluster-2 -n gloo-mesh
```

- Install the Gloo Agent in cluster-2

```
helm upgrade -i gloo-platform-crds gloo-platform/gloo-platform-crds \
  --version=2.3.9 \
  --namespace=gloo-mesh \
```

```
--kube-context cluster-2
```

```
helm upgrade -i gloo-agent gloo-platform/gloo-platform \
  --version=2.3.9 \
  --namespace gloo-mesh \
  --kube-context cluster-2 \
  --set glooAgent.relay.serverAddress=$GLOO_PLATFORM_SERVER_ADDRESS \
  --set common.cluster=cluster-2 \
  --set telemetryCollector.config.exporters.otlp.endpoint=$GLOO_TELEMETRY_GATEWAY \
  -f data/gloo-agent-values.yaml
```

- Verify the `gloo-mesh-agent` logs

```
kubectl logs deploy/gloo-mesh-agent --context cluster-2 -n gloo-mesh
```

- Verify the `gloo-telemetry-collector` logs

```
kubectl logs ds/gloo-telemetry-collector-agent --context cluster-2 -n gloo-mesh
```

Verify connectivity in the Gloo Platform UI

- Open the Gloo UI and observe the agents are connected and service discovery is working

```
kubectl port-forward svc/gloo-mesh-ui 8090:8090 --context management -n gloo-mesh
echo "Gloo UI: http://localhost:8090"
```

Lab 03 - Deploy Istio



Gloo Platform works with Open Source Istio distributions but Solo.io offers a number of different distributions of Istio for different types of environments and use cases such as FIPS, Arm, and distroless. To learn more about the different distributions view [Gloo Istio Distributions](#).

Links:

- [Gloo Istio](#)
- [Supported Istio Versions](#)
- [Gloo Platform Managed Istio](#)
- [Manual Istio Installs](#)

Installing Istio via `helm` is the preferred method for installing Istio manually. If you prefer to not manage you Istio installations you can trust that responsibility to Gloo Platform via its Istio [Istio Lifecycle Manager](#) Istio will be installed using `revisions` which is the optimal way to deploy Istio for production environments. It allows for the ability to **canary** Istio components that is safe and will prevent outages.

- First add the Istio helm repositories

```
helm repo add istio https://istio-release.storage.googleapis.com/charts
helm repo update
```

Install Istio on Cluster: cluster-1

Istio Components cluster-1

- Create `istio-system`, `istio-eastwest`, `istio-ingress` namespaces

```
kubectl apply --context cluster-1 -f data/namespaces.yaml
```

- Before installing Istio or upgrading the istio/base must be run to install or reconcile the CRDs within the kubernetes cluster.

```
helm upgrade -i istio-base istio/base \
  -n istio-system \
  --version 1.16.4 \
  --set defaultRevision=1-16 \
  --kube-context=cluster-1
```

- Install the `istiod` control plane

```
helm upgrade -i istiod-1-16 istio/istiod \
  --set revision=1-16 \
  --version 1.16.4 \
  --namespace istio-system \
  --kube-context=cluster-1 \
  --set "global.multiCluster.clusterName=cluster-1" \
  --set "meshConfig.trustDomain=cluster-1" \
  -f data/istiod-values.yaml
```

- Install the Istio eastwest gateway which is used for multi-cluster communication between clusters using mTLS.

```
helm upgrade -i istio-eastwestgateway istio/gateway \
  --set revision=1-16 \
  --version 1.16.4 \
  --namespace istio-eastwest \
  --kube-context=cluster-1 \
  -f data/eastwest-values.yaml
```

- Install the Istio ingress gateway with a ClusterIP service type. For best production practices and to support multiple revisions a standalone Service will be created to allow easy migration of traffic.

```
helm upgrade -i istio-ingressgateway-1-16 istio/gateway \
  --set revision=1-16 \
  --version 1.16.4 \
  --namespace istio-ingress \
  --kube-context=cluster-1 \
  -f data/ingress-values.yaml
```

- Create the standalone Kubernetes service to sit in front of the Istio ingressgateway.

```
kubectl apply --context cluster-1 -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: istio-ingressgateway
  namespace: istio-ingress
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "external"
    service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: "instance"
    service.beta.kubernetes.io/aws-load-balancer-scheme: "internet-facing"
  labels:
    istio: ingressgateway
    app: gloo-gateway
spec:
  type: LoadBalancer
  selector:
    istio: ingressgateway-1-16
  ports:
    # Port for health checks on path /healthz/ready.
    # For AWS ELBs, this port must be listed first.
    - name: status-port
      port: 15021
      targetPort: 15021
    # main http ingress port
    - port: 80
      targetPort: 8080
      name: http2
    # main https ingress port
    - port: 443
      targetPort: 8443
      name: https
EOF
```

- Verify pods are running

```
kubectl get pods --context cluster-1 -n istio-system
kubectl get pods --no-headers --context cluster-1 -n istio-ingress
kubectl get pods --no-headers --context cluster-1 -n istio-eastwest
```

- Verify the load balancer is created`

```
kubectl get service --context cluster-1 -n istio-ingress
kubectl get service --context cluster-1 -n istio-eastwest
```

Install Istio on Cluster: cluster-2



Istio Components cluster-2

- Create `istio-system`, `istio-eastwest`, `istio-ingress` namespaces

```
kubectl apply --context cluster-2 -f data/namespaces.yaml
```

- Install the Istio specific CRDs

```
helm upgrade -i istio-base istio/base \
  -n istio-system \
  --version 1.16.4 \
  --set defaultRevision=1-16 \
  --kube-context=cluster-2
```

- Install the `istiod` control plane

```
helm upgrade -i istiod-1-16 istio/istiod \
  --set revision=1-16 \
  --version 1.16.4 \
  --namespace istio-system \
  --kube-context=cluster-2 \
  --set "global.multiCluster.clusterName=cluster-2" \
  --set "meshConfig.trustDomain=cluster-2" \
  -f data/istiod-values.yaml
```

- Install istio eastwest gateway

```
helm upgrade -i istio-eastwestgateway istio/gateway \
  --set revision=1-16 \
  --version 1.16.4 \
  --namespace istio-eastwest \
  --kube-context=cluster-2 \
  -f data/eastwest-values.yaml
```

- Verify pods are running

```
kubectl get pods --context cluster-2 -n istio-system
kubectl get pods --no-headers --context cluster-2 -n istio-ingress
kubectl get pods --no-headers --context cluster-2 -n istio-eastwest
```

- Verify the load balancer is created`

```
kubectl get service --context cluster-2 -n istio-ingress
kubectl get service --context cluster-2 -n istio-eastwest
```

Lab 04 - Deploy Gloo Platform Addons

The Gloo Platform Addons are extensions that helm enable certain features that are offered within. The Gloo Platform addons contain a set of applications and cache to enable rate limiting.

 Gloo Platform Addon Components

- Create the Gloo Platform Addons namespace

```
kubectl apply --context cluster-1 -f data/namespaces.yaml
```

- Install Gloo Platform Addon applications in cluster-1

```
helm upgrade -i gloo-platform-addons gloo-platform/gloo-platform \
  --namespace gloo-platform-addons \
  --kube-context=cluster-1 \
```

```
--version 2.3.9 \  
-f data/gloo-platform-addons.yaml
```

- Verify pods are up and running

```
kubectl get pods -n gloo-platform-addons --context cluster-1
```

- Register the external authorization server with Gloo Platform

```
kubectl create namespace ops-team --context management  
kubectl apply --context management -f - <<EOF  
apiVersion: admin.gloo.solo.io/v2  
kind: ExtAuthServer  
metadata:  
  name: ext-auth-server  
  namespace: ops-team  
spec:  
  destinationServer:  
    kind: VIRTUAL_DESTINATION  
    port:  
      number: 8083  
    ref:  
      cluster: management  
      name: ext-auth-server-vd  
      namespace: ops-team  
---  
apiVersion: networking.gloo.solo.io/v2  
kind: VirtualDestination  
metadata:  
  name: ext-auth-server-vd  
  namespace: ops-team  
spec:  
  hosts:  
    - extauth.vdest.solo.io  
  ports:  
    - number: 8083  
      protocol: TCP  
  services:  
    - cluster: "cluster-1"  
      name: ext-auth-service  
      namespace: gloo-platform-addons  
EOF
```

Lab 05 - Certificate Management

Links:

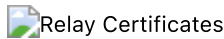
- <https://docs.solo.io/gloo-mesh-enterprise/latest/setup/prod/certs/>

Gloo Relay

To secure communication between the management and data planes, relay agents (gloo-mesh-agent) in workload clusters use server/client mTLS certificates to secure communication with the relay server (gloo-mesh-mgmt-server) in the management cluster.

By default, Gloo Mesh Enterprise autogenerates its own root CA certificate and intermediate signing CA for issuing the server and client certificates.

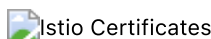
- The relay-server-tls-secret secret for the root CA certificate is stored in the gloo-mesh namespace of the management cluster.
- The root-trust.gloo-mesh secret for the relay server certificate is stored in the gloo-mesh namespace of the management cluster.
- The relay-client-tls-secret secret for the relay agent certificate is stored in the gloo-mesh namespace of each workload cluster.



Istio

The Istio deployment in each workload cluster requires a certificate authority (CA) certificate in the cacerts Kubernetes secret in the istio-system namespace.

Gloo Platform uses a root trust policy to configure the relay server (gloo-mesh-mgmt-server) to generate and self-sign a root CA certificate. This root CA certificate can sign Istio intermediate CA certificates whenever an Istio deployment in a workload cluster must create a certificate for a workload pod in its service mesh. Gloo Mesh stores the signed intermediate CA certificate in the cacerts Kubernetes secret in the istio-system namespace of the workload cluster.



Gloo Platform Managed

- Fully managed option - Gloo Platform will generate a new Root CA and Intermediates for each clusters Istio deployments

```
kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: RootTrustPolicy
metadata:
  name: root-trust-policy
  namespace: gloo-mesh
spec:
  config:
    mgmtServerCa:
      generated: {}
      autoRestartPods: true
EOF
```

In a later section, this workshop will show you how to swap out these autogenerated certificates with a managed Private Key Infrastructure (PKI)

Lab 06 - Deploy Online Boutique

The Online Boutique applicaion is a set of microservices that make up an online shopping website. There is a UI application that reaches out to many APIs to retrieve its data to populate the UI. This workshop will incrementally add features to this website in the coming labs.



- Create the Online Boutique namespaces

```
kubectl apply --context cluster-1 -f data/namespaces.yaml
```

- Deploy online boutique into cluster-1

```
helm upgrade -i online-boutique --version "5.0.3" oci://us-central1-docker.pkg.dev/field-engineering-us/helm-charts/onlineboutique \
  --namespace online-boutique \
  --kube-context cluster-1 \
  -f data/values.yaml
```

- Verify pods are running

```
kubectl get pods -n online-boutique --context cluster-1
```

Lab 07 - Configure Gloo Platform

Links:

- [Gloo Platform Multi-Tenancy](#)
- [Workspace API](#)
- [WorkspaceSettings API](#)

Setup Operations Team configuration namespace

- Create administrative namespace for ops-team

```
kubectl create namespace ops-team --context management
```

- Apply Gloo Platform configuration for ops-team

```
kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: Workspace
metadata:
  name: ops-team
  namespace: gloo-mesh
spec:
  workloadClusters:
    # administrative namespace
    - name: management
      namespaces:
        - name: ops-team
    # remote cluster namespaces
    - name: '*'
      namespaces:
        - name: istio-ingress
        - name: istio-eastwest
        - name: gloo-platform-addons
```

```

---
# workspace configuration
apiVersion: admin.gloo.solo.io/v2
kind: WorkspaceSettings
metadata:
  name: ops-team
  # placed in the administrative namespace
  namespace: ops-team
spec:
  # import service discovery from app-team
  importFrom:
  - workspaces:
    - name: app-team
  # export service discovery to any workspace that needs ingress
  exportTo:
  - workspaces:
    - name: "*"
  # for mutli cluster routing
  options:
    eastWestGateways:
    - selector:
        labels:
          app: gloo-internal-gateway
EOF

```

Setup Application Team configuration namespace

- Create administrative namespace for app-team

```
kubectl create namespace app-team --context management
```

- Apply Gloo Platform configuration for app-team

```

kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: Workspace
metadata:
  name: app-team
  namespace: gloo-mesh
spec:
  workloadClusters:
  # administrative namespace
  - name: management
    namespaces:
    - name: app-team
  # workload cluster namespace
  - name: '*'
    namespaces:
    - name: online-boutique
---
apiVersion: admin.gloo.solo.io/v2
kind: WorkspaceSettings
metadata:
  name: app-team
  namespace: app-team
spec:
  # import gateway service for ingress
  importFrom:
  - workspaces:

```

```

- name: ops-team
# share service discovery and routing to ingress
exportTo:
- workspaces:
  - name: ops-team
EOF

```

View Service Discovery in Gloo Platform UI

- Bring up the Gloo Platform Dashboard


```

kubectl port-forward svc/gloo-mesh-ui 8090:8090 --context management -n gloo-mesh
echo "Gloo UI: http://localhost:8090"


```

- View workspaces
- View app-team service discovery

Lab 08 - Ingress

 Online Boutique Frontend Links:

- [Gloo Platform Routing](#)
- [VirtualGateway API](#)
- [RouteTable API](#)
- [Route Delegation](#)

 Ingress Traffic

Operations team gateway setup

The operations team is responsible for setting the ports and protocols of Gloo Gateway. In a later section they will also secure the gateway with TLS.

- Configure Gloo Gateway ports and protocols using the Gloo `VirtualGateway` API. Delegate routing decisions to themselves, the `ops-team`

```

kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: VirtualGateway
metadata:
  name: ingress
  namespace: ops-team
spec:
  workloads:
    - selector:
        labels:
          app: gloo-gateway
          cluster: cluster-1
          namespace: istio-ingress
  listeners:
    # HTTP port

```

```

- http: {}
  port:
    number: 80
  # allow application team to make routing decisions
  allowedRouteTables:
    - host: '*'
      selector:
        workspace: ops-team
EOF

```

Note The Operations team has delegates the TOP level routing decisions to themselves. This not only gives them fine grained routing and delegation decision capabilities, it allows them to apply policies at the top level before traffic will reach an application.

- Create a `RouteTable` to delegate traffic decisions to the application team. They will decide where the traffic ultimately flows.

```

kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: RouteTable
metadata:
  name: ingress
  namespace: ops-team
spec:
  hosts:
    - '*'
  virtualGateways:
    - name: ingress
      namespace: ops-team
  workloadSelectors: []
  http:
    - name: application-ingress
      labels:
        ingress: all
      delegate:
        routeTables:
          - namespace: app-team
EOF

```

Application Team Routing

Due to the Ops team delegating routing decisions to the App team, the App team now needs to configure where traffic should flow.

- Configure a `RouteTable` object to route to online-boutique frontend

```

kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: RouteTable
metadata:
  name: frontend
  namespace: app-team
spec:
  workloadSelectors: []
  http:
    - name: frontend
      forwardTo:
        destinations:

```

```

- ref:
  name: frontend
  namespace: online-boutique
  cluster: cluster-1
port:
  number: 80
EOF

```

- Access online-boutique

```

export GLOO_GATEWAY=$(kubectl --context cluster-1 -n istio-ingress get svc -l istio=ingressgateway
-o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}') :80

echo "Online Boutique available at http://$GLOO_GATEWAY"

```

Secure Ingress with HTTPS

Most users need to secure traffic coming from outside their Kubernetes cluster. To do this you need to create a certificate that can be used by Gloo Gateway in the namespace the gateway resides.

- Create example certificate and upload to gloo gateway namespace

```

openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout tls.key -out tls.crt -subj "/CN=*"

kubectl create secret generic tls-secret --from-file=tls.key=tls.key --from-file=tls.crt=tls.crt --
context cluster-1 -n istio-ingress

```

- Using the `VirtualGateway` API we can update the current configuration to expose traffic on port 443 using TLS.

```

kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: VirtualGateway
metadata:
  name: ingress
  namespace: ops-team
spec:
  workloads:
    - selector:
        labels:
          app: gloo-gateway
          cluster: cluster-1
          namespace: istio-ingress
  listeners:
    # HTTP port
    - http: {}
      port:
        number: 80
      allowedRouteTables:
        - host: '*'
          selector:
            workspace: ops-team
    # HTTPS port
    - http: {}
      port:
        number: 443

```

```

    tls:
      mode: SIMPLE
      secretName: tls-secret # NOTE
    allowedRouteTables:
      - host: '*'
        selector:
          workspace: ops-team
EOF

```

- Access online-boutique with HTTPS

```

export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l
istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}') :443

echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"

```

- Optional curl

```

curl -k --write-out '%{http_code}' https://$GLOO_GATEWAY_HTTPS


```

Lab 09 - Zero Trust Communication

Links:

- [Zero Trust Whitepaper](#)
- [Access Policy Docs](#)
- [AccessPolicy API](#)

Deploy Zero Trust Policy

 Zero trust app team

- Disable all traffic by default for Application Team

```

kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: allow-nothing
  namespace: app-team
spec:
  applyToWorkloads:
    - {}
  config:
    authn:
      tlsMode: STRICT
    authz: {}
EOF

```

- Test traffic to online-boutique

```

https://$GLOO_GATEWAY_HTTPS


```

- Optional curl

```
curl -k --write-out '%{http_code}' https://$GLOO_GATEWAY_HTTPS
```

Allow Fine Grain Access

Now that all traffic is denied by default, traffic needs to be allowed between the microservices for the Online Boutique to function.

 Zero trust app team

- Allow access to the frontend from the ingress gateway

```
kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: frontend-ingress-access
  namespace: app-team
spec:
  applyToDestinations:
  - selector:
      workspace: app-team
  config:
    authz:
      allowedClients:
      - serviceAccountSelector:
          workspace: ops-team
EOF
```

- Open online-boutique

```
https://$GLOO_GATEWAY_HTTPS
```

- Allow frontend to reach apis in same namespace

```
kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: in-namespace-access
  namespace: app-team
spec:
  applyToDestinations:
  - selector:
      workspace: app-team
  config:
    authz:
      allowedClients:
      - serviceAccountSelector:
          workspace: app-team
EOF
```

- Refresh page

Lab 10 - Multi Cluster Secure Communication

To show the ease at which Gloo Platform can extend application routing beyond a single cluster, this lab will deploy a portion of the online boutique feature to another cluster. In order to checkout and buy your items, the frontend application needs to reach the checkout APIs. By using the Gloo Platform `VirtualDestination` API, multi-cluster applications can be represented by a "Global" hostname that can be reached by any other application that has Gloo Platform installed.



Multi Cluster Checkout Feature Links:

- [Multi-Cluster Routing Docs](#)
- [Virtual Destination API](#)

Deploy Checkout to cluster-2



Checkout APIs

- Create `checkout-apis` namespace in cluster-2

```
kubectl apply --context cluster-2 -f data/namespaces.yaml
```

- Deploy checkout APIs to cluster-2

```
helm upgrade -i checkout-apis --version "5.0.3" oci://us-central1-docker.pkg.dev/field-engineering-us/helm-charts/onlineboutique \
  --namespace checkout-apis \
  --kube-context cluster-2 \
  -f data/checkout-values.yaml
```

Configure Gloo Platform for Checkout Team

The checkout APIs will be managed by the `checkout-team` in cluster-2. To represent this, a new Gloo `Workspace` will be created for this team and its services will be exported to the `app-team`. The `app-team` workspace will need to be updated to import the `checkout-team` services.

- Create administration namespace for checkout-team

```
kubectl create namespace checkout-team --context management
```

- Create workspace for checkout-team

```
kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: Workspace
metadata:
  name: checkout-team
  namespace: gloo-mesh
spec:
  workloadClusters:
```



```

# administrative namespace
- name: 'management'
  namespaces:
    - name: checkout-team
# workload cluster namespace
- name: '*'
  namespaces:
    - name: checkout-apis
---
apiVersion: admin.gloo.solo.io/v2
kind: WorkspaceSettings
metadata:
  name: checkout-team
  namespace: checkout-team
spec:
  # share service discovery with app-team
  exportTo:
    - workspaces:
        - name: app-team
        - name: ops-team
  # import apis from app team
  importFrom:
    - workspaces:
        - name: app-team
EOF

```

- Update app-team Workspace to import checkout apis

```

kubectl apply --context management -f - <<EOF
apiVersion: admin.gloo.solo.io/v2
kind: WorkspaceSettings
metadata:
  name: app-team
  namespace: app-team
spec:
  # import gateway service for ingress
  importFrom:
    - workspaces:
        - name: ops-team
    - workspaces:
        - name: checkout-team
  # share service discovery and routing to ingress
  exportTo:
    - workspaces:
        - name: ops-team
    - workspaces:
        - name: checkout-team
EOF

```

- View workspace in Gloo Mesh UI

```

kubectl port-forward svc/gloo-mesh-ui 8090:8090 --context management -n gloo-mesh
echo "Gloo UI: http://localhost:8090"

```

Setup Global Services

In order to facilitate multi-cluster routing, Gloo Platform `VirtualDestinations` need to be created for applications being called in other clusters. By creating `VirtualDestinations` and assigning them a unique

global hostname, applications running in other clusters can seamlessly make requests to services in other clusters.

- Create Global Services for checkout-apis

```
kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: VirtualDestination
metadata:
  name: checkout
  namespace: checkout-team
spec:
  hosts:
  - checkout.checkout-team.demo.example.com
  services:
  - labels:
      app: checkoutservice
  ports:
  - number: 80
    protocol: GRPC
    targetPort:
      name: grpc
---
apiVersion: networking.gloo.solo.io/v2
kind: VirtualDestination
metadata:
  name: shipping
  namespace: checkout-team
spec:
  hosts:
  - shipping.checkout-team.demo.example.com
  services:
  - labels:
      app: shippingservice
  ports:
  - number: 81
    protocol: HTTP
    targetPort:
      name: http
  - number: 80
    protocol: GRPC
    targetPort:
      name: grpc
EOF
```

- Create Global Services for app team apis

```
kubectl apply --context management -n app-team -f data/app-team-global-services.yaml
```

- Update frontend to call checkout global services.

```
helm upgrade -i online-boutique --version "5.0.3" oci://us-central1-docker.pkg.dev/field-
engineering-us/helm-charts/onlineboutique \
  --namespace online-boutique \
  --kube-context cluster-1 \
  -f data/frontend-values.yaml
```

- Test multi cluster routing

Update Access Policies

Due to the `app-team` and `checkout-team` employing zero trust architectures, they both need to create policies that allow services to communicate between workspaces.

- Allow `app-team` to use checkout apis

```
kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: allow-app-team-frontend
  namespace: checkout-team
spec:
  applyToDestinations:
  - selector:
      workspace: checkout-team
      name: checkoutservice
  - selector:
      workspace: checkout-team
      name: shippingservice
  config:
    authz:
      allowedClients:
      - serviceAccountSelector:
          workspace: app-team
          name: frontend
          cluster: 'cluster-1'
      - serviceAccountSelector:
          workspace: app-team
          name: frontend
          cluster: 'cluster-2'
---
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: in-namespace-access
  namespace: checkout-team
spec:
  applyToDestinations:
  - selector:
      workspace: checkout-team
  config:
    authz:
      allowedClients:
      - serviceAccountSelector:
          workspace: checkout-team
---
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: ingress-gateway-access
  namespace: checkout-team
spec:
  applyToDestinations:
  - selector:
      workspace: checkout-team
      name: checkoutservice
  - selector:
```

```

        workspace: checkout-team
        name: shipping-service
    config:
        authz:
            allowedClients:
            - serviceAccountSelector:
                workspace: ops-team
EOF

```

- Allow checkout team to use app team apis

```

kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: AccessPolicy
metadata:
  name: allow-checkout-apis
  namespace: app-team
spec:
  applyToDestinations:
  - selector:
      workspace: app-team
      name: product-catalog-service
  - selector:
      workspace: app-team
      name: cart-service
  - selector:
      workspace: app-team
      name: currency-service
  - selector:
      workspace: app-team
      name: email-service
  config:
    authz:
      allowedClients:
      # only allow checkout api access
      - serviceAccountSelector:
          namespace: checkout-apis
          name: checkout-service
          cluster: 'cluster-2'
EOF

```

Lab 11 - Observability

Gloo Platform offers the best in class observability for communication between your services. By aggregating logging, tracing, and metrics, Gloo Platform combines telemetry from all environments to give you a complete picture of how your platform is performing.



Gloo Platform Graph Links:

- [Exploiting the Gloo UI](#)
- [Prometheus](#)
- [Gloo Platform Tracing/Metrics/Logs](#)

Gloo Platform UI

The Gloo UI is automatically installed in the Gloo management cluster. Let's explore some of the key features that you have access to when using the Gloo UI:

- **Gloo Platform overview: With the Gloo UI:** you can view information about your Gloo Platform environment, such as the number of clusters that are registered with the Gloo management server and the Istio version that is deployed to them. You can also review your workspace settings and which Gloo resources you import and export to other workspaces.
- **Verify service mesh configurations:** The Gloo Mesh UI lets you quickly find important information about your service mesh setup, the participating clusters, and the Gloo Mesh resources that you applied to the service mesh. For example, you can review your workspace settings and which Gloo Mesh resources you import and export to other workspaces. You can also view your gateway and listener configurations and traffic policies that you applied to your service mesh routes.
- **Drill into apps and services:** Review what services can communicate with other services in the mesh, the policies that are applied before traffic is sent to a service, and how traffic between services is secured.
- **Visualize and monitor service mesh health metrics:** With the built-in Prometheus integration, the Gloo Mesh UI has access to detailed Kubernetes cluster and service mesh metrics, such as the node's CPU and memory capacity, unresponsive nodes or nodes with degraded traffic, and mTLS settings for services in the mesh. For more information about the Prometheus integration in Gloo Mesh, see Prometheus.
- **OIDC authentication and authorization support:** Set up authentication and authorization (AuthN/AuthZ) for the Gloo Mesh UI by using OpenID Connect (OIDC) and Kubernetes role-based access control (RBAC). The Gloo Mesh UI supports OpenID Connect (OIDC) authentication from common providers such as Google, Okta, and Auth0.

For a detailed overview of what information you can find in the Gloo UI, see [Explore the Gloo UI](#).

- Navigate to Gloo Mesh Graph



Gloo Platform Graph

```
kubectl port-forward svc/gloo-mesh-ui 8090:8090 --context management -n gloo-mesh
echo "Gloo UI: http://localhost:8090"
```

- Click `Workspaces` and select all workspaces
- Click `Layout Settings` in the bottom right of the screen and update `Group By:` to `CLUSTER`



Gloo Platform Layout

Telemetry

Gloo Platform Telemetry is based on the [Open Telemetry](#) standards. Metrics, logs and traces are gathered in cluster by the `Gloo Telemetry Collector`. Where it is shipped from there is up the user. By default, telemetry data will be shipped to the `Gloo Telemetry Gateway` in the `Management Plane` and aggregated with all other telemetry data from other clusters.



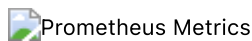
Gloo Telemetry Arch **Third Party Support**

Gloo Platform can also ship these metrics to you third party telemetry provider like `Datadog`, `Splunk`, `New Relic`, or `Dynatrace`.



Metrics

Metrics provide important information about the health of the apps in your service mesh. You can use these measures to detect failures, troubleshoot bottlenecks, and to find ways to improve the performance and reliability of your service mesh.



- To view the raw metrics

```
kubectl port-forward svc/prometheus-server --context management -n gloo-mesh 9080:80
echo "Prometheus available at http://localhost:9080"
```

- Search for metrics prefixed with `istio_`, Example `istio_requests_total`
- Here is an example query that lists non 200 codes for each workload

```
sum(rate(istio_requests_total{response_code!="200"}[1m])) by (workload_id,response_code)
```

Tracing

Sample request traces to monitor the traffic and health of your service mesh.

Distributed tracing helps you track requests across multiple services in your distributed service mesh. Sampling the requests in your mesh can give you insight into request latency, serialization, and parallelism



- Deploy Jaeger to the `management` cluster

```
helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
helm repo update
```

```
helm upgrade -i jaeger jaegertracing/jaeger \
  --kube-context management \
  --version 0.71.7 \
  --namespace gloo-mesh \
  -f data/jaeger-values.yaml
```

- Create a service for Istio to send metrics

```
kubectl apply --context cluster-1 -f data/tracing-service.yaml
kubectl apply --context cluster-2 -f data/tracing-service.yaml
```

- Enable Tracing within each cluster

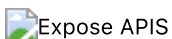
```
kubectl apply --context cluster-1 -f - <<EOF
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: default
  namespace: istio-system
spec:
  tracing:
    - providers:
        - name: zipkincustom
          randomSamplingPercentage: 100
          disableSpanReporting: false
      accessLogging:
        - providers:
            - name: envoyOtelAls
EOF
kubectl apply --context cluster-2 -f - <<EOF
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: default
  namespace: istio-system
spec:
  tracing:
    - providers:
        - name: zipkincustom
          randomSamplingPercentage: 100
          disableSpanReporting: false
      accessLogging:
        - providers:
            - name: envoyOtelAls
EOF
```

- Open Jaeger and observe tracing

```
kubectl port-forward svc/jaeger-query --context management -n gloo-mesh 9081:16686
echo "Jaeger available at http://localhost:9081"
```

- Open browser at <http://localhost:9081>
- Select service `frontend.online-boutique` and then click `Find Traces`

Lab 12 - Expose APIs



Next, let's see how easy it is to expose multiple applications. The Online Boutique frontend relies on a number of APIs to populate the UI. The product catalog API is responsible for displaying the available products and the currency API converts the cost of each product into the required denomination. To expose these APIs, we will match on URI `prefix: /currencies` and send to the currency service and `prefix: /products` to the product catalog service.

- Reminder to set the `GLOO_GATEWAY_HTTPS` environment variable

```
export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}'):443

echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"
```

Note: you may notice the weight: 100 configuration. This tells Gloo gateway to place this RouteTable before the frontend RouteTable with the / prefix route which doesn't have a weight. Higher integer values are considered higher priority. The default value is 0.

Expose APIS

- Expose the currency API

```
kubectl apply --context management -f data/currency-route-table.yaml
```

- Test the currency API

```
# get the available currencies
curl -k https://$GLOO_GATEWAY_HTTPS/currencies

# convert a currency
curl -k https://$GLOO_GATEWAY_HTTPS/currencies/convert \
--header 'Content-Type: application/json' \
--data '{
  "from": {
    "currency_code": "USD",
    "nanos": 0,
    "units": 8
  },
  "to_code": "EUR"
}'
```

- Expose the product catalog API

```
kubectl apply --context management -f data/products-route-table.yaml
```

- Test requests to the product catalog API

```
curl -k https://$GLOO_GATEWAY_HTTPS/products
```

Expose API In Another Cluster

The Gloo Gateway can also expose applications that do not reside in its own cluster using VirtualDestinations. The following lab exposes the `shipping` API in cluster-2.

- Since the API is owned by the Checkout Team, the Ops team needs to import their service to make it routable.

```
kubectl apply --context management -f data/ops-team.yaml
```

- Create the RouteTable pointing to the Shipping VirtualDestination


```
kubectl apply --context management -f data/shipping-route-table.yaml
```

- Test that the Gloo Gateway can reach the shipping service

```
curl -k -X POST https://$GLOO_GATEWAY_HTTPS/shipping/quote \
  --header 'Content-Type: application/json' \
  --header 'Accept: application/json' \
  --data '{
    "address": {
      "city": "Cambridge",
      "country": "US",
      "state": "MA",
      "street_address": "222 Third Street #3300",
      "zip_code": 12142
    },
    "items": [
      {
        "product_id": "OLJCESPC7Z",
        "quantity": 5
      }
    ]
  }'
```

Lab 13 - Circuit Breaking and Failover


Gloo Platform can enable your applications to be highly available by allowing you to deploy them across multiple clusters and localities. Gloo Platform can automatically detect these new deployments and add them as available endpoints for routing. In many cases it would be inefficient to route to all available endpoints across localities. By adding outlier detection and failover policies, you can create a circuit breaking strategy that is efficient and effective.

Links:

- [Failover](#)
- [Outlier Detection](#)
- [FailoverPolicy API](#)
- [OutlierDetectionPolicy API](#)

Deploy HA frontend

This lab will deploy the frontend application to both cluster-1 and cluster-2 and update the ingress routing to route to both available endpoints. You will then tune the routing to prefer the local frontend

and failover to the other when something bad happens.  High Availability Frontend

- Create online-boutique namespace in cluster-2

```
kubectl apply --context cluster-2 -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
  labels:
    istio.io/rev: 1-16
  name: online-boutique
EOF
```

- Deploy frontend in cluster-2

```
helm upgrade -i ha-frontend --version "5.0.3" oci://us-central1-docker.pkg.dev/field-engineering-us/helm-charts/onlineboutique \
  --kube-context cluster-2 \
  --namespace online-boutique \
  --set clusterName=cluster-2 \
  -f data/web-ui-values.yaml
```

- Wait until the frontend in cluster-2 is ready.
- Create a VirtualDestination to represent both frontend applications.

```
kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: VirtualDestination
metadata:
  name: frontend
  namespace: app-team
spec:
  hosts:
    - frontend.app-team.demo.example.com
  services:
    - labels:
        app: frontend
  ports:
    - number: 80
      protocol: HTTP
      targetPort:
        name: http
EOF
```

- Then update the ingress routing to route to the VirtualDestination

```
kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: RouteTable
metadata:
  name: frontend
  namespace: app-team
spec:
  workloadSelectors: []
  http:
    - name: frontend
      labels:
        virtual-destination: frontend
      forwardTo:
        destinations:
          - ref:
              name: frontend
              namespace: app-team
              kind: VIRTUAL_DESTINATION
            port:
              number: 80
EOF
```

- By default the ingress gateway will now route to both frontend applications in a round robin

pattern.

Locality Based Routing

In order to enable locality based routing, you need to define some parameters to organize and prioritize the available endpoints. First configuring a FailoverPolicy will set the rules for ordering endpoints based on their locality to the client. Secondly, parameters need to be configured to tell the client when it should prefer another set of endpoints (ie. when to circuit break). The OutlierDetection policy sets the conditions for when an endpoint is deemed unhealthy and will be removed from routing.



Locality Load Balancing

- Create a FailoverPolicy to setup an ordered list of endpoints.

```
kubectl apply --context management -f - <<EOF
apiVersion: resilience.policy.gloo.solo.io/v2
kind: FailoverPolicy
metadata:
  name: failover
  namespace: app-team
spec:
  applyToDestinations:
  - kind: VIRTUAL_DESTINATION
    selector:
      namespace: app-team
  config:
    # enable default locality based load balancing
    localityMappings: []
EOF
```


- Add an OutlierDetectionPolicy to set the conditions of when an endpoint is considered unhealthy.

```
kubectl apply --context management -f - <<EOF
apiVersion: resilience.policy.gloo.solo.io/v2
kind: OutlierDetectionPolicy
metadata:
  name: outlier-detection
  namespace: app-team
spec:
  applyToDestinations:
  - kind: VIRTUAL_DESTINATION
    selector:
      namespace: app-team
  config:
    consecutiveErrors: 2
    interval: 5s
    baseEjectionTime: 15s
    maxEjectionPercent: 100
EOF
```

- Refresh online boutique to observe locality based routing

Perform Failover

With the configurations in place, Gloo Platform will configure the ingress and proxy sidecars to automatically failover when the OutlierDetection parameters are met. To simulate this, the frontend in cluster-1 will be configured to no longer respond to traffic. The ingress gateway will observe this behavior and automatically remove cluster-1 frontend from the list of available endpoints. It will then

prefer the endpoint in cluster-2 until cluster-1 frontend is healthy again.  Failover

- Break frontend in cluster-1 so that it can no longer respond to traffic

```
kubectl patch deploy frontend --patch '{"spec":{"template":{"spec":{"containers":[{"name":"server","command":["sleep","20h"],"readinessProbe":null,"livenessProbe":null}]}}}}' --context cluster-1 -n online-boutique
```

NOTE: You will see 2 errors before being failed over. These could have been avoided using a retry policy.

Failover Routing

- Refresh the Online Boutique and observe the error and failover to cluster-2
- Fix frontend in cluster-1

```
kubectl patch deploy frontend --patch '{"spec":{"template":{"spec":{"containers":[{"name":"server","command":[],"readinessProbe":null,"livenessProbe":null}]}}}}' --context cluster-1 -n online-boutique
```

- Refresh the Online Boutique and observe the traffic be redirected back to the cluster-1 frontend once it's healthy.

Lab 14 - Rate Limiting

The Gloo Platform Rate Limiting API allows for very fine grained configuration to meet the needs of the end user. First a user must decide the type of rate limiting they want to enforce. Below is some common types of rate limiting.

Types of rate limiting

- Global inbound - rate limit x number of requests per second
- User based - User can make x number of requests per second
- Per backend - Prevent too many requests from overwhelming a given endpoint

Links:

- [Rate Limit Docs](#)
- [RateLimitServerConfig API](#)
- [RateLimitServerSettings API](#)
- [RateLimit API](#)
- [RateLimitClientConfig API](#)
- [RateLimitPolicy API](#)
- To better view the ratelimiting, we are going to disable circuit breaking if its enabled

```
kubectl delete outlierdetectionpolicy outlier-detection --context management -n app-team
```

- Apply the RateLimitPolicy to enable rate limiting

```
kubectl apply --context management -f data/rate-limit-policy.yaml
```

- Refresh UI a few times should see `x-envoy-ratelimited: [true]`


```
Trailers:
  content-type: [application/grpc]
  date: [Wed, 08 Mar 2023 14:42:05 GMT]
  server: [envoy]
  x-envoy-ratelimited: [true]
  x-envoy-upstream-service-time: [2]
```

- Remove the RateLimitPolicy

```
kubectl delete --context management -f data/rate-limit-policy.yaml
```

Lab 15 - Authentication / JWT + JWKS

One way for users to authenticate themselves is via JSON Web Tokens or JWT for short. Gloo Platform can validate JWT tokens against an external JSON Web Key Set (JWKS). Below is a lab that validates JWT tokens generated from an external provider Auth0.

 JWT Enforcement Links:

- [External Authorization Docs](#)
- [JWTPolicy API](#)
- Reminder to set the `GLOO_GATEWAY_HTTPS` environment variable

```
export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}'):443
```

```
echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"
```

Enforcing JWT Authentication

- Since JWT authentication is best viewed while accessing an API, first verify that the currency application is still available.

```
# get the available currencies
curl -k https://$GLOO_GATEWAY_HTTPS/currencies

# convert a currency
curl -k "https://$GLOO_GATEWAY_HTTPS/currencies/convert" \
--header 'Content-Type: application/json' \
--data '{
  "from": {
    "currency_code": "USD",
    "nanos": 0,
    "units": 8
  },
```

```
    "to_code": "EUR"
  }
}
```

- Since the Auth0 servers are not known to Gloo Platform, create an ExternalService reference.

```
kubectl apply --context management -f - <<EOF
apiVersion: networking.gloo.solo.io/v2
kind: ExternalService
metadata:
  name: auth0
  namespace: app-team
spec:
  hosts:
  - dev-64ktibmv.us.auth0.com
  ports:
  - name: https
    number: 443
    protocol: HTTPS
    clientsideTls: {}
EOF
```

- Enable JWT Authentication for the currency routes

```
kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: JWTPolicy
metadata:
  name: currency
  namespace: app-team
spec:
  applyToRoutes:
  - route:
      labels:
        route: currency
  config:
    providers:
      auth0:
        issuer: "https://dev-64ktibmv.us.auth0.com/"
        audiences:
        - "https://currency/api"
        remote:
          url: "https://dev-64ktibmv.us.auth0.com/.well-known/jwks.json"
          destinationRef:
            ref:
              name: auth0
              namespace: app-team
              cluster: management
              kind: EXTERNAL_SERVICE
              port:
                number: 443
              enableAsyncFetch: true
              keepToken: true
EOF
```

- Call the currency API and get denied

```
curl -vk https://$GLOO_GATEWAY_HTTPS/currencies
```

Expected output

```
Jwt is missing
```

- Generate JWT from Auth0 using the below command

```
ACCESS_TOKEN=$(curl -sS --request POST \
  --url https://dev-64ktibmv.us.auth0.com/oauth/token \
  --header 'content-type: application/json' \
  --data
'{"client_id":"lQEVhZ2ERqZOpTQnHChK1TUSKRBduO72","client_secret":"J_vl_qgu0pvudTfGppm_PJcQjkgy-
kmy5KRCQDj5XHZbo5eFtxmSbpmqYT5ITv2h","audience":"https://currency/api","grant_type":"client_credentia:
| jq -r '.access_token')

printf "\n\n Access Token: $ACCESS_TOKEN\n"
```

- Call currency API using the Access token to authenticate

```
curl -vk -H "Authorization: Bearer $ACCESS_TOKEN" https://$GLOO_GATEWAY_HTTPS/currencies
```

- Cleanup JWT policy

```
kubectl delete jwtpolicy currency --context management -n app-team
```

Lab 16 - Gloo Platform OPA Integration

[OPA](#) is an open source, general-purpose policy engine that you can use to enforce versatile policies in a uniform way across your organization. Compared to a role-based access control (RBAC) authorization system, OPA allows you to create more fine-grained policies. For more information, see the [OPA docs](#).

OPA policies are written in [Rego](#). Based on the older query languages Prolog and Datalog, Rego extends support to more modern document models such as JSON.

- Reminder to set the `GLOO_GATEWAY_HTTPS` environment variable

```
export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l
istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}') :443

echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"
```

Native OPA Integration

Gloo Mesh's OPA integration populates an `input` document to use in your OPA policies. The structure of the `input` document depends on the context of the incoming request, described in the following table.

OPA input structure	Description
<code>input.check_request</code>	By default, all OPA policies contain an Envoy Auth Service CheckRequest . This object has all the information that Envoy gathers about the request being processed. You can view the structure of this object in the attributes section of the linked Envoy doc.

input.http_request	When processing an HTTP request, Envoy populates this field for convenience. For the structure of this object, see the Envoy HttpRequest docs and proto files .
input.state.jwt	If you use OAuth, the token retrieved during the OIDC flow is placed into this field.

- Create an OPA policy to be ready by Gloo Platform

```
cat <<EOF > policy.rego
package test

default allow = false
allow {
    startswith(input.http_request.path, "/currencies")
    input.http_request.method == "GET"
}
EOF
```

- Create configmap for the policy

```
kubectl create configmap allow-currency-admin --from-file=policy.rego --context cluster-1 -n
online-boutique
```

- Create an `ExtAuthPolicy` that validates incoming requests against the OPA policy

```
kubectl apply --context management -f - <<EOF
apiVersion: security.policy.gloo.solo.io/v2
kind: ExtAuthPolicy
metadata:
  name: api-auth
  namespace: app-team
spec:
  applyToDestinations:
  - selector:
      labels:
        app: currency
  config:
    server:
      name: ext-auth-server
      namespace: ops-team
      cluster: management
    glooAuth:
      configs:
      - opaAuth:
          modules:
            - name: allow-currency-admin
              namespace: online-boutique
              query: "data.test.allow == true"
EOF
```

- Test requests to the currency service

```
# get the available currencies NO API Key
curl -vk https://$GLOO_GATEWAY_HTTPS/currencies
```



```
# get the available currencies with API Key
curl -vk -H "x-api-key: developer" https://$GLOO_GATEWAY_HTTPS/currencies
curl -vk -H "x-api-key: admin" https://$GLOO_GATEWAY_HTTPS/currencies

# convert a currency with developer key
curl -k -H "x-api-key: developer" https://$GLOO_GATEWAY_HTTPS/currencies/convert \
--header 'Content-Type: application/json' \
--data '{
  "from": {
    "currency_code": "USD",
    "nanos": 0,
    "units": 8
  },
  "to_code": "EUR"
}'

# convert a currency with admin key
curl -k -H "x-api-key: admin" https://$GLOO_GATEWAY_HTTPS/currencies/convert \
--header 'Content-Type: application/json' \
--data '{
  "from": {
    "currency_code": "USD",
    "nanos": 0,
    "units": 8
  },
  "to_code": "EUR"
}'
```

Lab 17 - Calling External Services

Many times users will need to communicate with applications outside of the service mesh. Gloo Platform represents these applications as `External Services`. By telling Gloo Platform about an External Service you are allowing that endpoint to be discovered and routable from applications running inside the service mesh. This concept becomes powerful due to the ability to also apply Gloo Platform policies to it. For example in the below lab you will be securing the `httpbin.org` application with the JWT authentication in the previous lab.



External API Links:

- [External Service Doc](#)
- [ExternalService API](#)
- Reminder to set the `GLOO_GATEWAY_HTTPS` environment variable

```
export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l
istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}')
echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"
```

Create an ExternalService

- Create an ExternalService reference to the external API.

```
kubectl apply --context management -n app-team -f data/external-service.yaml
```

- Update the RouteTable to route to the ExternalService

```
kubectl apply --context management -n app-team -f data/external-route-table.yaml
```

- Curl and see the response from the external service

```
curl -vk https://$GLOO_GATEWAY_HTTPS/get
```

Lab 18 - Day 2 Certificates

Gloo and Istio heavily rely on TLS certificates to facilitate safe and secure communication. Gloo Platform uses mutual tls authentication for communication between the Server and the Agents. Istio requires an Intermediate Signing CA so that it can issue workload certificates to each of the mesh enabled services. These workload certificates encrypt and authenticate traffic between each of your microservices.

It is important to design and implement a secure and reliable Public Key Infrastructure (PKI) that Gloo and Istio can rely on which is why we chose cert-manager as the PKI due to its versatility and reliability for managing certificates.

Cert Manager

Not only is cert-manager the most widely used Kubernetes based solution, it natively integrates with a number of different issuing systems such as [AWS Private CA](#), [Google Cloud CA](#) and [Vault](#). Finally, cert-manager also creates certificates in the form of kubernetes secrets which are compatible with both Istio and Gloo Platform. It also has the ability to automatically rotate them when they are nearing their end of life.

Cert Manager Backends

- Add the cert-manager helm chart

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

- Install cert-manager using helm

```
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.12.2 \
  --kube-context management \
  -f data/cert-manager-values.yaml
```

```
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.12.2 \
  --kube-context cluster-1 \
  -f data/cert-manager-values.yaml
```

```
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
```

```
--version v1.12.2 \
--kube-context cluster-2 \
-f data/cert-manager-values.yaml
```

- Wait for deployments to become healthy

```
kubectl wait deployment --for condition=Available=True --all --context management -n cert-manager
kubectl wait deployment --for condition=Available=True --all --context cluster-1 -n cert-manager
kubectl wait deployment --for condition=Available=True --all --context cluster-2 -n cert-manager
```

Self Signed Root Cert

This lab will use a self signed root certificate for all relay and workload certificates. This is not recommended in production as the Root CA key is stored in a kubernetes secret. Instead an external 3rd party PKI is recommended like Vault or Venafi.

- Create the self-signed secret

```
kubectl create secret generic issuer-ca --from-file=tls.key=data/root-key.pem --from-
file=tls.crt=data/root-cert.pem --context management -n cert-manager
kubectl create secret generic issuer-ca --from-file=tls.key=data/root-key.pem --from-
file=tls.crt=data/root-cert.pem --context cluster-1 -n cert-manager
kubectl create secret generic issuer-ca --from-file=tls.key=data/root-key.pem --from-
file=tls.crt=data/root-cert.pem --context cluster-2 -n cert-manager
```

- Create a ClusterIssuer for the root secret

```
kubectl apply --context management -n cert-manager -f data/secret-issuer.yaml
kubectl apply --context cluster-1 -n cert-manager -f data/secret-issuer.yaml
kubectl apply --context cluster-2 -n cert-manager -f data/secret-issuer.yaml
```

- Verify the issuers

```
kubectl get clusterissuer self-signed-issuer -o jsonpath='{.status}' --context management -n cert-
manager
kubectl get clusterissuer self-signed-issuer -o jsonpath='{.status}' --context cluster-1 -n cert-
manager
kubectl get clusterissuer self-signed-issuer -o jsonpath='{.status}' --context cluster-2 -n cert-
manager
```

Cluster: management Configuration

The Gloo Platform server and Telemetry Gateway will need mTLS server certificates. The following commands generate the two certificates to allow the applications to receive connections from the workload clusters. * Create certificates for Gloo Management Server and Telemetry Gateway ``shell kubectl apply --context management -f - <<EOF apiVersion: cert-manager.io/v1 kind: Certificate metadata: name: gloo-mgmt-server namespace: gloo-mesh spec:

commonName: gloo-mgmt-server dnsNames: - "*.gloo-mesh"
duration: 8760h0m0s ### 1 year life renewBefore: 8736h0m0s
issuerRef: kind: ClusterIssuer name: self-signed-issuer secretName:
gloo-mgmt-server-tls usages: - server auth - client auth privateKey:
algorithm: "RSA" size: 4096

apiVersion: cert-manager.io/v1 kind: Certificate metadata: name: gloo-telemetry-gateway namespace:
gloo-mesh spec: commonName: gloo-mgmt-server dnsNames: - "*.gloo-mesh" duration: 8760h0m0s
1 year life renewBefore: 8736h0m0s issuerRef: kind: ClusterIssuer name: self-signed-issuer
secretName: gloo-telemetry-gateway usages: - server auth - client auth privateKey: algorithm: "RSA"
size: 4096 EOF

```
* Verify certificates were created
```shell
kubectl get certificates --context management -n gloo-mesh
```

**Note** if certificates were not generated it may be beneficial to look at the cert manager logs.

```
kubectl logs deploy/cert-manager --context management -n cert-manager
```

- Cleanup old Gloo certificates and allow cert-manager to replace them

```
kubectl delete secret relay-server-tls-secret --context management -n gloo-mesh
kubectl delete secret relay-tls-signing-secret --context management -n gloo-mesh
kubectl delete secret relay-root-tls-secret --context management -n gloo-mesh
```

- Update the Gloo Platform to use the new certificates

```
helm upgrade --install gloo-platform gloo-platform/gloo-platform \
 --version=2.3.9 \
 --namespace=gloo-mesh \
 --kube-context management \
 --reuse-values \
 -f data/gloo-mgmt-values.yaml
```

## Cluster: cluster-1 Configuration

The workload clusters will need 2-3 certificates depending on your environment. The Gloo Platform Agent will require a client mTLS certificate for communicating with the Gloo Platform Server. Likewise the Telemetry Collector will also require an mTLS certificate to communicate with the Telemetry Gateway.

If you are relying on Istio's CA issuer functionality, you will also need to issue Istio a CA certificate to issue workload certificates to the dataplane. If your security posture does not allow for CA certificates to be stored on the workload clusters, ask your Solo.io Representative about `istio-csr`.

- Verify issuers is correctly setup

```
kubectl get clusterissuer --context cluster-1 -n cert-manager
```

- Create certificates for Gloo Agent, Telemetry Gateway and Istio if needed

```
kubectl apply --context cluster-1 -f - <<EOF
kind: Certificate
apiVersion: cert-manager.io/v1
metadata:
 name: gloo-agent
 namespace: gloo-mesh
spec:
 commonName: gloo-agent
 dnsNames:
 # Must match the cluster name used in the install
 - "cluster-1"
 duration: 8760h0m0s ### 1 year life
 renewBefore: 8736h0m0s
 issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
 secretName: relay-client-tls-secret
 usages:
 - digital signature
 - key encipherment
 - client auth
 - server auth
 privateKey:
 algorithm: "RSA"
 size: 4096

```

```
kind: Certificate
apiVersion: cert-manager.io/v1
metadata:
 name: gloo-telemetry-collector
 namespace: gloo-mesh
spec:
 commonName: gloo-telemetry-collector
 dnsNames:
 - "cluster-1-gloo-telemetry-collector"
 duration: 8760h0m0s ### 1 year life
 renewBefore: 8736h0m0s
 issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
 secretName: gloo-telemetry-collector
 usages:
 - digital signature
 - key encipherment
 - client auth
 - server auth
 privateKey:
 algorithm: "RSA"
 size: 4096

```

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
 name: istio-cacerts
 namespace: istio-system
spec:
 secretName: cacerts
 duration: 720h # 30d
 renewBefore: 360h # 15d
 commonName: cluster-1.demo.example.com
 isCA: true
 usages:
```

```

 - digital signature
 - key encipherment
 - cert sign
 dnsNames:
 - cluster-1.demo.example.com
 issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
EOF

```

- Verify certificates were created

```

kubectl get certificates --context cluster-1 -n gloo-mesh
kubectl get certificates --context cluster-1 -n istio-system

```

**Note** if certificates were not generated it may be beneficial to look at the cert manager logs.

```

kubectl logs deploy/cert-manager --context cluster-1 -n cert-manager
kubectl logs deploy/cert-manager-istio-csr --context cluster-1 -n cert-manager

```

- Cleanup old Gloo certificates and allow cert-manager to replace them

```

kubectl delete secret relay-client-tls-secret --context cluster-1 -n gloo-mesh
kubectl delete secret relay-root-tls-secret --context cluster-1 -n gloo-mesh
kubectl delete secret relay-identity-token-secret --context cluster-1 -n gloo-mesh

```

- Update the Gloo Platform to use the new certificates

```

helm upgrade --install gloo-agent gloo-platform/gloo-platform \
 --version=2.3.9 \
 --namespace gloo-mesh \
 --kube-context cluster-1 \
 --reuse-values \
 -f data/gloo-agent-values.yaml

```

- Verify Gloo Agent connectivity

```

kubectl logs deploy/gloo-mesh-agent --context cluster-1 -n gloo-mesh

```

- Update Istio to use new CA certificate

```

kubectl delete secret cacerts --context cluster-1 -n istio-system

```

- Verify new cacerts is generated

```

kubectl get secret cacerts --context cluster-1 -n istio-system

```

- Restart Istiod to pick up new certificate

```

kubectl rollout restart deploy --context cluster-1 -n istio-system

```

- Verify new certificate is picked up by istiod

```
kubectl logs -l app=istiod --tail 500 --context cluster-1 -n istio-system| grep x509
```

- Restart Gateways

```
kubectl rollout restart deploy --context cluster-1 -n istio-ingress
kubectl rollout restart deploy --context cluster-1 -n istio-eastwest
kubectl rollout restart deploy --context cluster-1 -n gloo-platform-addons
```

- Restart workloads

```
kubectl rollout restart deploy -n online-boutique --context cluster-1
```

## Cluster: cluster-2 Configuration

- Verify issuers is correctly setup

```
kubectl get clusterissuer --context cluster-2 -n cert-manager
```

- Create certificates for Gloo Agent, Telemetry Gateway and Istio if needed

```
kubectl apply --context cluster-2 -f - <<EOF
kind: Certificate
apiVersion: cert-manager.io/v1
metadata:
 name: gloo-agent
 namespace: gloo-mesh
spec:
 commonName: gloo-agent
 dnsNames:
 # Must match the cluster name used in the install
 - "cluster-2"
 duration: 8760h0m0s ### 1 year life
 renewBefore: 8736h0m0s
 issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
 secretName: relay-client-tls-secret
 usages:
 - digital signature
 - key encipherment
 - client auth
 - server auth
 privateKey:
 algorithm: "RSA"
 size: 4096

kind: Certificate
apiVersion: cert-manager.io/v1
metadata:
 name: gloo-telemetry-collector
 namespace: gloo-mesh
spec:
 commonName: gloo-telemetry-collector
 dnsNames:
 - "cluster-2-gloo-telemetry-collector"
 duration: 8760h0m0s ### 1 year life
```

```

renewBefore: 8736h0m0s
issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
secretName: gloo-telemetry-collector
usages:
 - digital signature
 - key encipherment
 - client auth
 - server auth
privateKey:
 algorithm: "RSA"
 size: 4096

apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
 name: istio-cacerts
 namespace: istio-system
spec:
 secretName: cacerts
 duration: 720h # 30d
 renewBefore: 360h # 15d
 commonName: cluster-2.demo.example.com
 isCA: true
 usages:
 - digital signature
 - key encipherment
 - cert sign
 dnsNames:
 - cluster-2.demo.example.com
 issuerRef:
 kind: ClusterIssuer
 name: self-signed-issuer
EOF

```

- Verify certificates were created

```

kubectl get certificates --context cluster-2 -n gloo-mesh
kubectl get certificates --context cluster-2 -n istio-system

```

**Note** if certificates were not generated it may be beneficial to look at the cert manager logs.

```

kubectl logs deploy/cert-manager --context cluster-2 -n cert-manager
kubectl logs deploy/cert-manager-istio-csr --context cluster-2 -n cert-manager

```

- Cleanup old Gloo certificates and allow cert-manager to replace them

```

kubectl delete secret relay-client-tls-secret --context cluster-2 -n gloo-mesh
kubectl delete secret relay-root-tls-secret --context cluster-2 -n gloo-mesh
kubectl delete secret relay-identity-token-secret --context cluster-2 -n gloo-mesh

```

- Update the Gloo Platform to use the new certificates

```

helm upgrade --install gloo-agent gloo-platform/gloo-platform \
 --version=2.3.9 \
 --namespace gloo-mesh \
 --kube-context cluster-2 \

```



```
--reuse-values \
-f data/gloo-agent-values.yaml
```

- Verify Gloo Agent connectivity

```
kubectl logs deploy/gloo-mesh-agent --context cluster-2 -n gloo-mesh
```

- Update Istio to use new CA certificate

```
kubectl delete secret cacerts --context cluster-1 -n istio-system
```

- Verify new cacerts is generated

```
kubectl get secret cacerts --context cluster-2 -n istio-system
```

- Restart Istiod to pick up new certificate

```
kubectl rollout restart deploy --context cluster-2 -n istio-system
```

- Verify new certificate is picked up by istiod

```
kubectl logs -l app=istiod --tail 500 --context cluster-2 -n istio-system | grep x509
```

- Restart Gateways

```
kubectl rollout restart deploy --context cluster-2 -n istio-ingress
kubectl rollout restart deploy --context cluster-2 -n istio-eastwest
```

- Restart workloads

```
kubectl rollout restart deploy -n online-boutique --context cluster-2
kubectl rollout restart deploy -n checkout-apis --context cluster-2
```

- Verify that the Gloo UI appears to be healthy
- Open the Gloo UI and observe the agents are connected and service discovery is working

```
kubectl port-forward svc/gloo-mesh-ui 8090:8090 --context management -n gloo-mesh
echo "Gloo UI: http://localhost:8090"
```

- Verify Online Boutique is functioning as expected


```
export GLOO_GATEWAY_HTTPS=$(kubectl --context cluster-1 -n istio-ingress get svc -l
istio=ingressgateway -o jsonpath='{.items[0].status.loadBalancer.ingress[0].*}')443
```

```
echo "SECURE Online Boutique available at https://$GLOO_GATEWAY_HTTPS"
```

- Optional curl

```
curl -k --write-out '%{http_code}' https://$GLOO_GATEWAY_HTTPS
```

## Lab 19 - Zero Downtime Istio Upgrades

 Upgrade Istio Historically, upgrading Istio without downtime has been a complicated ordeal. Today,

Gloo Platform can easily facilitate zero downtime upgrades using its Istio Lifecycle Manager. Gloo Platform allows you to deploy multiple versions of Istio side-by-side and transition from one version to the other. This lab will show you how to upgrade Istio quickly and safely without impacting client traffic. This is done through the use of Istio revisions.

Links:

- [Istio Production Best Practices](#)
- [Gloo Platform Managed Istio](#)
- [GatewayLifecycleManager API](#)
- [IstioLifecycleManager API](#)

### Upgrade Istio using Helm in Cluster: cluster-1

- To upgrade Istio we will deploy a whole new canary version beside it. We will also deploy new gateways and migrate traffic to them.

```
helm upgrade --install istio-base istio/base \
 -n istio-system \
 --kube-context=cluster-1 \
 --version 1.17.2 \
 --set defaultRevision=1-17
```

```
helm upgrade -i istiod-1-17 istio/istiod \
 --set revision=1-17 \
 --version 1.17.2 \
 --namespace istio-system \
 --kube-context=cluster-1 \
 --set "global.multiCluster.clusterName=cluster-1" \
 --set "meshConfig.trustDomain=cluster-1" \
 -f data/istiod-values.yaml
```

- In this lab we will show you two options for upgrading gateways, in-place and via revisions. Depending on your needs and requirements you can choose the best option that will work for you.
- Upgrade the Istio eastwest gateway in place

```
helm upgrade -i istio-eastwestgateway istio/gateway \
 --set revision=1-17 \
 --version 1.17.2 \
 --namespace istio-eastwest \
 --kube-context=cluster-1 \
 -f data/eastwest-values.yaml
```

- Deploy new Istio 1-17 ingress gateway using revisions

```
helm upgrade -i istio-ingressgateway-1-17 istio/gateway \
 --set revision=1-17 \
 --version 1.17.2 \
```

```
--namespace istio-ingress \
--kube-context=cluster-1 \
-f data/ingress-values.yaml
```

- Update the standalone Kubernetes service send traffic to the new Istio ingressgateway.

```
kubectl apply --context cluster-1 -f - <<EOF
apiVersion: v1
kind: Service
metadata:
 name: istio-ingressgateway
 namespace: istio-ingress
 labels:
 app: gloo-gateway
 annotations:
 service.beta.kubernetes.io/aws-load-balancer-type: "external"
 service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: "instance"
 service.beta.kubernetes.io/aws-load-balancer-scheme: "internet-facing"
spec:
 type: LoadBalancer
 selector:
 istio: ingressgateway-1-17
 ports:
 # Port for health checks on path /healthz/ready.
 # For AWS ELBs, this port must be listed first.
 - name: status-port
 port: 15021
 targetPort: 15021
 # main http ingress port
 - port: 80
 targetPort: 8080
 name: http2
 # main https ingress port
 - port: 443
 targetPort: 8443
 name: https
EOF
```

- Finally update the application namespaces to the new revision and perform a rolling restart.

```
kubectl label namespace online-boutique --overwrite istio.io/rev=1-17 --context cluster-1 -n
online-boutique
kubectl rollout restart deploy --context cluster-1 -n online-boutique
kubectl label namespace gloo-platform-addons --overwrite istio.io/rev=1-17 --context cluster-1 -n
online-boutique
kubectl rollout restart deploy --context cluster-1 -n gloo-platform-addons
```

- Remove Istio

```
helm uninstall istio-ingressgateway-1-16 \
--namespace istio-ingress \
--kube-context=cluster-1

helm uninstall istiod-1-16 \
--namespace istio-system \
--kube-context=cluster-1
```

- Verify only Istio 1-17 is running

```
istioctl proxy-status --context cluster-1
```

## Upgrade Istio using Helm in Cluster: cluster-2

- Upgrade Istiod to 1-17 components

```
helm upgrade --install istio-base istio/base \
 --kube-context=cluster-2 \
 -n istio-system \
 --version 1.17.2 \
 --set defaultRevision=1-17

helm upgrade -i istiod-1-17 istio/istiod \
 --set revision=1-17 \
 --version 1.17.2 \
 --namespace istio-system \
 --kube-context=cluster-2 \
 --set "global.multiCluster.clusterName=cluster-2" \
 --set "meshConfig.trustDomain=cluster-2" \
 -f data/istiod-values.yaml
```

- Upgrade the Istio eastwest gateway in place

```
helm upgrade -i istio-eastwestgateway istio/gateway \
 --set revision=1-17 \
 --version 1.17.2 \
 --namespace istio-eastwest \
 --kube-context=cluster-2 \
 -f data/eastwest-values.yaml
```

- Finally update the application namespaces to the new revision and perform a rolling restart.

```
kubectl label namespace online-boutique --overwrite istio.io/rev=1-17 --context cluster-2 -n
online-boutique
kubectl rollout restart deploy --context cluster-2 -n online-boutique

kubectl label namespace checkout-apis --overwrite istio.io/rev=1-17 --context cluster-2 -n
checkout-apis
kubectl rollout restart deploy --context cluster-2 -n checkout-apis
```

- Remove Istio

```
helm uninstall istiod-1-16 \
 --namespace istio-system \
 --kube-context=cluster-2
```

- Verify only Istio 1-17 is running

```
istioctl proxy-status --context cluster-2
```

## Lab 20 - POC Clean Up

You have completed the POC! The final step is to clean up the deployed assets and reset the environments back to their original state.

*Some of the commands here may try and cleanup things that dont exist. This is just to make sure all POC assets were accounted for.*

## Clean Up Applications

- Remove the Online Boutique Applications in cluster-1

```
helm uninstall online-boutique \
 --namespace online-boutique \
 --kube-context cluster-1
```

```
helm uninstall toys-catalog \
 --namespace online-boutique \
 --kube-context cluster-1
```

- Remove the Online Boutique Applications in cluster-2

```
helm uninstall ha-frontend \
 --namespace online-boutique \
 --kube-context cluster-2
```

```
helm uninstall checkout-apis \
 --namespace checkout-apis \
 --kube-context cluster-2
```

- Delete the namespaces

```
kubectl delete namespace online-boutique --context cluster-1
```

```
kubectl delete namespace online-boutique --context cluster-2
```

```
kubectl delete namespace checkout-apis --context cluster-2
```

## Clean up Gloo Addons

- Remove the Gloo Addons

```
helm uninstall gloo-platform-addons \
 --namespace gloo-platform-addons \
 --kube-context cluster-1
```

- Delete the Gloo Addons namespace

```
kubectl delete namespace gloo-platform-addons --context cluster-1
```

## Clean Up Istio

- Depending on if Istio was upgraded or not, the following commands may need to be modified.

```
helm ls -n istio-system --kube-context cluster-1
helm ls -n istio-ingress --kube-context cluster-1
helm ls -n istio-eastwest --kube-context cluster-1
```

```
helm ls -n istio-system --kube-context cluster-2
helm ls -n istio-eastwest --kube-context cluster-2
```

- Uninstall the gateways in cluster-1

```
helm uninstall istio-eastwestgateway \
 --namespace istio-eastwest \
 --kube-context cluster-1

helm uninstall istio-ingressgateway-1-16 \
 --namespace istio-ingress \
 --kube-context cluster-1

helm uninstall istio-ingressgateway-1-17 \
 --namespace istio-ingress \
 --kube-context cluster-1
```

- Uninstall the control plane in cluster-1

```
helm uninstall istiod-1-16 \
 --namespace istio-system \
 --kube-context cluster-1

helm uninstall istiod-1-17 \
 --namespace istio-system \
 --kube-context cluster-1
```

- Uninstall the gateways in cluster-2

```
helm uninstall istio-eastwestgateway \
 --namespace istio-eastwest \
 --kube-context cluster-2
```

- Uninstall the control plane in cluster-1

```
helm uninstall istiod-1-16 \
 --namespace istio-system \
 --kube-context cluster-2

helm uninstall istiod-1-17 \
 --namespace istio-system \
 --kube-context cluster-2
```

- Remove the CRDS **NOTE** Make sure other teams are not using Istio on the cluster

```
helm template istio-base istio/base --namespace istio-system --include-crds | kubectl delete --
context cluster-1 -f -

helm template istio-base istio/base --namespace istio-system --include-crds | kubectl delete --
context cluster-2 -f -
```

- Cleanup the namespaces

```
kubectl delete namespace istio-system --context cluster-1
```

```
kubectl delete namespace istio-system --context cluster-2
```

## Clean Up Gloo Platform

- Remove Agents from workload clusters

```
helm uninstall gloo-agent \
 --namespace gloo-mesh \
 --kube-context cluster-1
```

```
helm uninstall gloo-agent \
 --namespace gloo-mesh \
 --kube-context cluster-2
```

- Remove Gloo CRDs

```
helm uninstall gloo-platform-crds \
 --namespace gloo-mesh \
 --kube-context cluster-1
```

```
helm uninstall gloo-platform-crds \
 --namespace gloo-mesh \
 --kube-context cluster-2
```

- Delete gloo-mesh namespaces

```
kubectl delete namespace gloo-mesh --context cluster-1
```

```
kubectl delete namespace gloo-mesh --context cluster-2
```

- Remove the Workspace namespaces in management cluster

```
kubectl delete namespace ops-team --context management
kubectl delete namespace app-team --context management
kubectl delete namespace checkout-team --context management
```

- Cleanup management cluster

```
helm uninstall jaeger \
 --namespace gloo-mesh \
 --kube-context management
```

```
helm uninstall gloo-platform \
 --namespace gloo-mesh \
 --kube-context management
```

```
helm uninstall gloo-platform-crds \
 --namespace gloo-mesh \
 --kube-context management
```

- Remove namespace

```
kubectl delete namespace gloo-mesh --context management
```

## Optional Deployments

- Cleanup Keycloak

```
kubectl delete namespace keycloak --context cluster-1
```

- Cleanup cert-manager

```
helm uninstall cert-manager \
 --namespace cert-manager \
 --kube-context management
```

```
helm uninstall cert-manager \
 --namespace cert-manager \
 --kube-context cluster-1
```

```
helm uninstall cert-manager \
 --namespace cert-manager \
 --kube-context cluster-2
```

- Cleanup cert-manager namespace

```
kubectl delete namespace cert-manager --context management
```

- Cleanup Vault

```
helm uninstall vault \
 --namespace vault \
 --kube-context management
```

- Cleanup vault namespace

```
kubectl delete namespace vault --context management
```