White paper

A Modern Approach to Service Mesh. Migrating from Sidecars to Sidecarless Ambient Mesh

Author: Peter Jausovec, Principal Technical Marketing Engineer @ Solo.io

Introduction

The service mesh landscape is undergoing a significant transformation with the introduction of sidecarless, or ambient mode. This shift represents a fundamental change in how service meshes are implemented and managed, moving from the traditional sidecar model to a more flexible and resource-efficient approach.

Traditionally, Istio has relied on a sidecar architecture, where each application pod is paired with a proxy container. While effective, this model can lead to increased resource consumption and operational complexity as the number of services grows. The ambient architecture introduces a new paradigm that separates Layer 4 (L4) and Layer 7 (L7) processing.

The L4 processing is handled by a node-level component called **ztunnel**. The L7 processing is managed by optional **waypoint proxies**, which can be deployed at various levels of granularity (namespace, service, or multi-namespace).

This architectural shift allows for more efficient resource utilization and flexibility in service mesh deployment and management.

Why migrate to ambient mesh?

Migrating to ambient mesh offers several significant benefits:

- **Reduced Resource Overhead:**
  Especially beneficial in large-scale deployments.
- **Improved Scalability and Performance:**
  More efficient handling of service communication.
- **Greater Flexibility:**
  In mesh configuration and policy enforcement.
- **Simplified Operations:**
  Easier maintenance and management of the service mesh.

However, it's important to note that migration also presents challenges, including:

- The complexity of migrating large, existing sidecar-based deployments
- Potential changes required in security and traffic management policies
- The learning curve associated with new concepts and deployment models

Key Takeaway

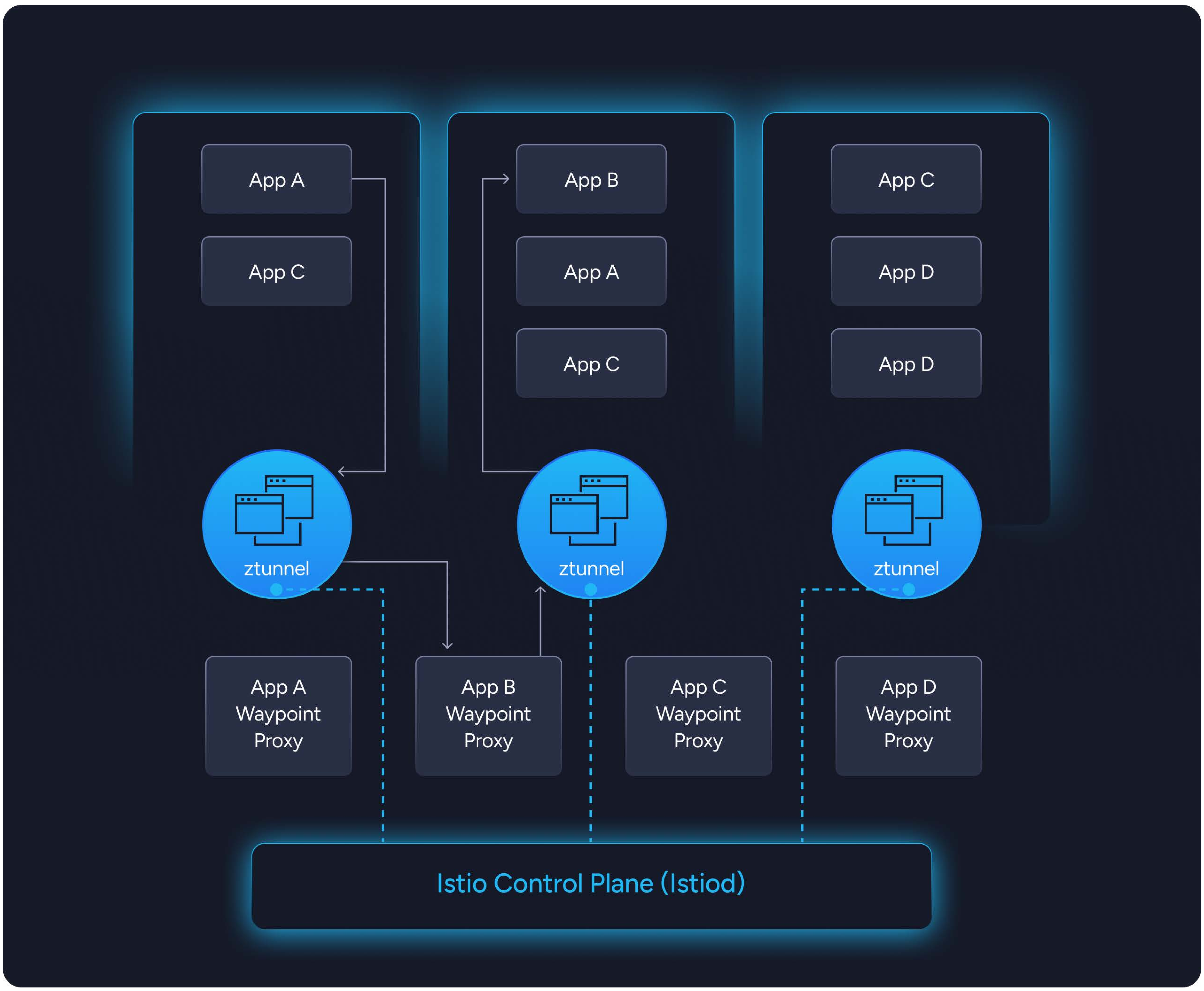
Sidecar and ambient modes can work together, allowing for a gradual migration strategy.

Understanding ambient mesh architecture

Ambient mesh introduces two key components:

1. ztunnel:
   A node-level component handling Layer 4 (L4) processing.

2. Waypoint Proxies:
   Optional components managing Layer 7 (L7) processing, deployable at various levels of granularity (namespace, service, or multi-namespace).



Ztunnels are designed to be fast, secure, and lightweight. They are deployed per node on a cluster and enable the basic service mesh configurations for L4 networking features such as mutual TLS (mTLS), telemetry, authentication, and L4 authorization.

Waypoint proxies provide L7 networking features such as any routing done in Istio's VirtualService, L7 telemetry, and L7 authorization policies.

Migration strategy and considerations

Due to the different architecture of ambient mode, there are a couple of prerequisites you must consider before deciding to migrate to ambient mode.

Ambient mesh depends on a CNI plugin called istio-cni. If you're using Istio without the istio-cni installed, you'll have to install it for ambient mesh.

1. Migrate to Kubernetes Gateway API (optional)
   The waypoint proxies in the ambient mesh use the Kubernetes Gateway API resources. You should migrate to the Kubernetes Gateway API and switch from the VirtualService resources to the HTTPRoute/TCPRoute resources. Additionally, if you're using authorization policies, make sure you're using the `targetRefs` selector in your resources as that makes the migration to ambient much easier.

2. Migrate sidecar workloads to ambient
   The last step in the migration process is removing the sidecar injection label from the namespaces and workloads in the mesh and replacing it with the ambient mode label, then doing a `rollout restart` to remove the injected sidecars.

With the split between L4 and L7 processing in ambient mesh, it's important to understand and inventory all resources and workloads where L7 processing is used. This will help you determine if you need waypoint proxies.

Security

If you're only using network-based authorization and identity-based policies, you're only doing L4 processing and you don't need a waypoint proxy. However, if you're using a full authorization policy, for example, anything in the `to` or `when` field in the AuthorizationPolicy or JWT authentication or OAuth and OIDC flows you will require a waypoint proxy.

Traffic Control

Traffic control features include load balancing, traffic shifting, and traffic mirroring. Any workload in your mesh that uses a VirtualService will require you to deploy a waypoint proxy to handle it in the ambient mesh.

Observability

The observability features include logging, tracing, and metrics. The ztunnel in upstream Istio only offers basic network logs and TCP metrics (bytes sent/received) and it doesn't support tracing.

You will need a waypoint proxy if you require L7 RED metrics (rate of requests, rate of errors, request duration), tracing, or full request metadata logging.

Note that with **Gloo Mesh core**, you get logging, tracing, and metrics without a waypoint proxy.

Resilience

The features falling under the resilience category are circuit breaking and outlier detection (defined in DestinationRule), rate limiting (EnvoyFilter and external service), timeouts, retries, and fault injection (defined in VirtualService). For these features to work you'll have to deploy a waypoint proxy.

Determining waypoint deployment granularity

Once you determine you require waypoint proxies, you'll have to decide the level of granularity for the waypoint. This can be either namespace, service, or multi-namespace. The decision on the granularity of the waypoint depends on the following:

1. Resource Utilization and Performance
   High-traffic services or services with resource intensive Istio policies may benefit from dedicated waypoint proxies. This prevents resource contention, allows for fine-tuned resource allocation, and can help optimize performance for latency-sensitive services

2. Security and Policy Management
   Services that require finer-grained policy enforcement might need separate waypoint proxies. In this case, you can deploy a dedicated waypoint proxy for a service or a group of services that require stricter isolation or have specific security policies. Conversely, if you have groups of services that share similar policies, it might be more efficient to group them under a shared waypoint proxy.

3. Organizational Structure and Operational Complexity
   Team structure and service management practices can influence waypoint deployment. If different teams manage different namespaces or services, it might make sense to deploy waypoint proxies at the namespace level to give teams more autonomy.

4. Resource Efficiency and Gradual Migration
   Start with a coarse-grained approach (e.g. one waypoint proxy per namespace) and refine based on needs and insights.

Challenges and mitigation strategies

Challenge 1

Complexity in migrating large, existing deployments

Mitigation:
Implement a phased migration approach, starting with non-critical services. Consider a hybrid model where sidecar and ambient workloads work together.

Challenge 2

Changes in security and traffic management policies

Mitigation:
Conduct an audit of existing policies and plan for necessary adjustments, migrate to Kubernetes Gateway API.

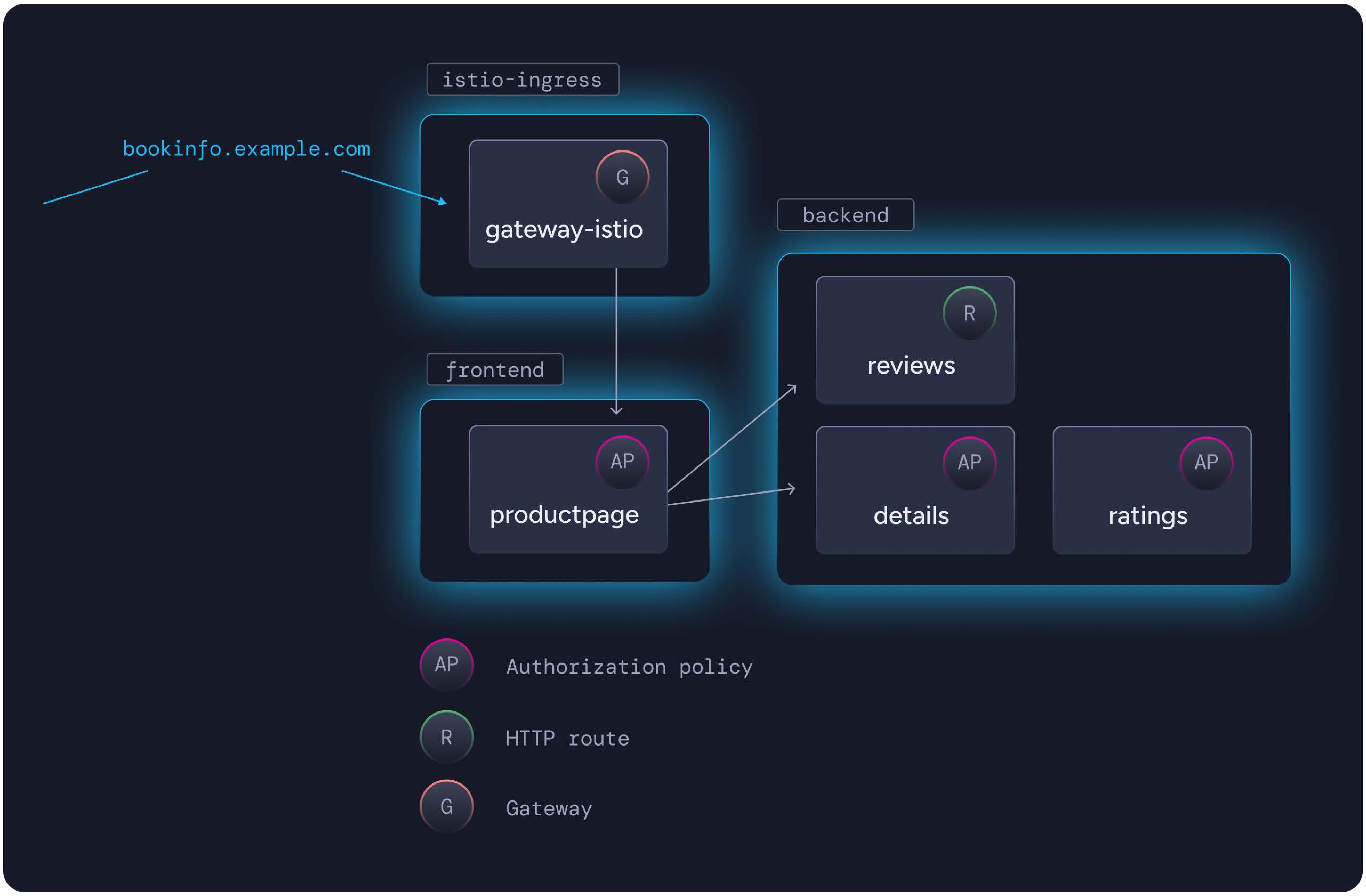
Challenge 3

Learning curve for new concepts

Mitigation:
Solo.io's expert support in ambient mesh empowers users to secure, control, and manage workloads with maximum efficiency. Our dedicated Istio specialists provide tailored guidance, helping you navigate new Ambient mesh concepts and overcome challenges with best-practice strategies.

Migration example

In this migration example, we'll use the [Bookinfo sample application] and deploy the services between two namespaces.



We'll use a Kind cluster for this test and install the latest version of Istio using Helm.

You can get all the files from this [GitHub repository].

We'll start by installing Istio and deploying the Bookinfo application in the sidecar mode, so we can showcase how the migration to ambient mesh might look like.

1. Install Istio

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

kubectl create ns istio-system

helm install istio-base istio/base -n istio-system --set
defaultRevision=default --wait
helm install istio-cni istio/cni -n istio-system --wait
helm install istiod istio/istiod -n istio-system --wait
```

2. Enable access logging:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.istio.io/v1
kind: Telemetry
metadata:
  name: mesh-default
  namespace: istio-system
spec:
  accessLogging:
    - providers:
      - name: envoy
EOF
```

3. Install Kubernetes Gateway API:

```
kubectl get crd gateways.gateway.networking.k8s.io &> /dev/null || \
  { kubectl apply -f https://github.com/kubernetes-sigs/gateway-api/
releases/download/v1.1.0/standard-install.yaml }
```

4. Install ingress gateway:

```
kubectl create ns istio-ingress

kubectl apply -f - <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: gateway
  namespace: istio-ingress
spec:
  gatewayClassName: istio
  listeners:
  - name: http
    hostname: "bookinfo.example.com"
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
        from: Selector
        selector:
            matchLabels:
              kubernetes.io/metadata.name: frontend
EOF
```

You can check the pod and LB service was created in `istio-ingress` by running `kubectl get svc,po -n istio-ingress`.

4. Install bookinfo

```
kubectl create ns frontend
kubectl label namespace frontend istio-injection=enabled
kubectl apply -f bookinfo/frontend.yaml -n frontend

kubectl create ns backend
kubectl label namespace backend istio-injection=enabled
kubectl apply -f bookinfo/backend.yaml -n backend
```

5. Create a routing rule to route traffic from the ingress gateway to the productpage service:

```
kubectl apply -f - <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: productpage
  namespace: frontend
spec:
  parentRefs:
  - name: gateway
    namespace: istio-ingress
  hostnames: ["bookinfo.example.com"]
  rules:
  - matches:
    - path:
        type: Exact
        value: /productpage
    - path:
        type: PathPrefix
        value: /static
    - path:
        type: Exact
        value: /login
```

```
    - path:
        type: Exact
        value: /logout
    - path:
        type: PathPrefix
        value: /api/v1/products
    backendRefs:
    - name: productpage
      port: 9080
EOF
```

6. Make sure you can access the product page via `http://bookinfo.example.com/productpage` (or LB IP + Host header).

```
curl -s -o /dev/null -w "%{http_code}\n" -H "Host: bookinfo.example.com"
172.18.255.200/productpage
```

The response should be an HTTP 200 OK

```
200
```

Authorization policies

We'll deploy a couple of authorization policies. First, an authorization policy that's enforced on the product page service and only allows requests to be sent from the ingress gateway:

```
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: productpage-viewer
  namespace: frontend
spec:
  selector:
    matchLabels:
      app: productpage
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
        - cluster.local/ns/istio-ingress/sa/gateway-istio
EOF
```

Sending the same request as before (through the ingress gateway) should still work, however, if we deploy a sleep pod in the `frontend` namespace and try to access the product page, it should fail:

```
kubectl run -n frontend sleep --image=curlimages/curl --command -- /bin/
sleep infinity
kubectl exec -n frontend -it sleep -- curl -s -o /dev/null -w "%{http_
code}\n" -H "Host: bookinfo.example.com" productpage:9080/productpage
```

The response should be an "HTTP 403 Forbidden".

```
403
```

The second authorization policy will be applied on the `ratings` service. We'll only allow GET and POST requests to be sent from the `reviews-v3` service:

```
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: ratings-policy
  namespace: backend
spec:
  selector:
    matchLabels:
      app: ratings
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
        - cluster.local/ns/backend/sa/bookinfo-reviews
    to:
    - operation:
        methods: ["GET", "POST"]
EOF
```

We can test this by deploying a sleep pod in the `backend` namespace and trying to access the ratings service:

```
kubectl run -n backend sleep --image=curlimages/curl --command -- /bin/
sleep infinity
kubectl exec -n backend -it sleep -- curl -s -o /dev/null -w "%{http_
code}\n" ratings.backend:9080/ratings/1
```

Confirm that the response is a "403 - Forbidden"

```
403
```

This second policy shouldn't affect the product page, as it's only applied to the ratings service.

The last authorization policy we'll deploy is for the details service and it will only allow GET requests from the product page service:

```
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: details-policy
  namespace: backend
spec:
  selector:
    matchLabels:
      app: details
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
```

```
        - cluster.local/ns/frontend/sa/bookinfo-productpage
    to:
    - operation:
        methods: [“GET”]
EOF
```

We can test this policy is applied by sending a request from a non-product page service:

```
kubectl exec -n backend -it sleep -- curl -s -o /dev/null -w “%{http_
code}\n” details.backend:9080/details/1
```

```
403
```

Traffic policies

Let’s also configure a traffic routing policy that will route all traffic to the `reviews-v3` service:

```
kubectl apply -f - <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: reviews
  namespace: backend
spec:
  parentRefs:
  - group: ""
    kind: Service
    name: reviews
    port: 9080
  rules:
  - backendRefs:
    - name: reviews-v3
      port: 9080
EOF
```

Make sure you can access the product page via `http://bookinfo.example.com/productpage` (or LB IP + Host header) and notice the reviews are only served by the `reviews-v3` service:

```
export GATEWAY_IP=$(kubectl get svc -n istio-ingress gateway-istio -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
curl -s -H "Host: bookinfo.example.com" $GATEWAY_IP/productpage | grep
"reviews-"
```

```
reviews-v3-6f5b775685-sxv4d
reviews-v3-6f5b775685-sxv4d
```

Let’s also scale up all deployments to 2 replicas:

```
kubectl scale deploy -n frontend --replicas=2 --all
kubectl scale deploy -n backend --replicas=2 --all
```

Installing Istio ambient mode

The first step is to upgrade Istio charts with ambient mode enabled and install ztunnel:

```
helm install istio-cni istio/cni -n istio-system --set profile=ambient
--wait

# Upgrade (reinstall istiod) with ambient profile
helm upgrade istiod istio/istiod --namespace istio-system --set
profile=ambient --wait

# Install ztunnel
helm install ztunnel istio/ztunnel -n istio-system --wait
```

Make sure everything is installed:

```
helm ls -n istio-system
```

```
NAME            NAMESPACE       REVISION        UPDATED
STATUS          CHART           APP VERSION
istio-base      istio-system    1               2024-10-03
16:10:25.118083 -0700 PDT    deployed        base-1.23.2     1.23.2
istio-cni       istio-system    2               2024-10-03 15:47:31.63797
-0700 PDT     deployed        cni-1.23.2      1.23.2
istiod          istio-system    2               2024-10-03
16:12:31.274487 -0700 PDT    deployed        istiod-1.23.2   1.23.2
ztunnel         istio-system    1               2024-10-03
16:14:06.883523 -0700 PDT    deployed        ztunnel-1.23.2  1.23.2
```

Migration process

Let's remove the sidecar injection label from the `frontend` and `backend` namespace - this is to ensure that any new pods that are created or restarted won't have the sidecar proxy injected:

```
kubectl label namespace frontend istio-injection-
kubectl label namespace backend istio-injection-
```

And we need to label the namespaces to tell Istio we want to add the pods to the ambient mode, once we restart them:

```
kubectl label namespace frontend istio.io/dataplane-mode=ambient
kubectl label namespace backend istio.io/dataplane-mode=ambient
```

In the sidecar mode, any routing or authorization policies are applied at the client side, so we need to determine whether we need waypoint proxies that will enforce the policies once we remove the sidecar proxies.

1. `productpage-viewer` authorization policy: this policy is applied on the product page service and only allows requests to be sent from the ingress gateway. In this case, because we're not using any L7 concepts, even if we remove the sidecar proxy from the productpage, the ztunnel will automatically enforce the policy.

2. `details-policy` authorization policy: this policy is applied to the details service and allows only product page service to send GET requests. Because we're using an L7 concept (the GET method), ztunnel won't be able to enforce this policy (it will automatically deny it), so we'll need a waypoint proxy to handle this as well.

3. `ratings-policy` authorization policy: applied to the ratings service and only allows requests from the reviews service with GET or POST methods. Since we're using HTTP method, we'll need a waypoint proxy to enforce this policy.

4. `productpage` HTTP route: this HTTP route configures the ingress gateway to route the traffic to the specific paths on the productpage. Since the routing rules are applied and enforced on the ingress gateway, we don't need to deploy a waypoint proxy for this.

5. `reviews` HTTP route: the route on the reviews service that routes all traffic to the reviews-v3 service. In this case, productpage is the client, so if we remove the sidecar proxy the client will not be able to enforce the route. We need to deploy a waypoint proxy for the reviews service to handle the traffic routing.

We'll need a waypoint proxy, so let's deploy one in the `backend` namespace and enroll the backend namespace (this means that all pods in the backend namespace will use this instance of the waypoint if needed). Later, you can decide to deploy more waypoint proxies in the backend namespace if needed.

```
istioctl waypoint apply -n backend --enroll-namespace --wait
```

```
waypoint backend/waypoint applied
namespace backend labeled with "istio.io/use-waypoint: waypoint"
```

You can check the waypoint is ready by running:

```
kubectl get gtw -n backend
```

```
NAME       CLASS           ADDRESS        PROGRAMMED   AGE
waypoint   istio-waypoint  10.96.54.173   True         25s
```

Now that we have the waypoint deployed, we can take the existing L7 policies and create an ambient version of them that uses the `targetRef` field. In ambient, the `targetRef` field is the one supported by the waypoints and it tells the waypoint to enforce the policy. We don't want to directly modify the existing policies, because we want to keep them enforced until we restart the workloads and remove the sidecar proxies. If we'd update the existing policies, the sidecar proxies wouldn't know how to enforce them until we restarted the workloads.

The routing policies will be automatically enforced by the waypoint proxies, because we're already using the HTTPRoute resources. If you weren't using that and you were using VirtualServices, you'd have to create an HTTPRoute resource for each VirtualService that you have before you remove the sidecar proxies.

Let's deploy the waypoint version of existing policies:

```
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: details-policy-waypoint
  namespace: backend
spec:
  targetRefs:
  - kind: Service
    group: “”
    name: details
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
        - cluster.local/ns/frontend/sa/bookinfo-productpage
    to:
    - operation:
        methods: [“GET”]
---
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: ratings-policy-waypoint
  namespace: backend
spec:
  targetRefs:
  - kind: Service
    group: “”
    name: ratings
  action: ALLOW
  rules:
  - from:
    - source:
        principals:
        - cluster.local/ns/backend/sa/bookinfo-reviews
    to:
    - operation:
        methods: [“GET”, “POST”]
EOF
```

We're at the point now where we have the waypoint proxy deployed and the policies are in place. The next step is to restart the pods in the `frontend` and `backend` namespace to remove the sidecar proxies and enroll them in the ambient mode:

```
kubectl rollout restart deploy -n frontend
kubectl rollout restart deploy -n backend

kubectl delete po sleep -n frontend
kubectl delete po sleep -n backend

kubectl run -n frontend sleep --image=curlimages/curl --command -- /bin/
sleep infinity
kubectl run -n backend sleep --image=curlimages/curl --command -- /bin/
sleep infinity
```

We can now run the `istioctl zc workload` and `istioctl zc service` command to verify that all pods were moved to ambient and that the pods in the backend namespace are using the waypoint proxy:

```
istioctl zc workload
```

```
NAMESPACE          POD NAME                                     IP           NODE
WAYPOINT PROTOCOL
backend            details-v1-558d6b8747-fd6nx                  10.244.0.44
kind-control-plane None     HBONE
backend            details-v1-558d6b8747-tjjwc                  10.244.0.39
kind-control-plane None     HBONE
backend            ratings-v1-78d7884947-br5hw                  10.244.0.38
```

```
kind-control-plane None     HBONE
backend             ratings-v1-78d7884947-mg2p6          10.244.0.46
kind-control-plane None     HBONE
backend             reviews-v1-cdd45ff95-h7nx9           10.244.0.45
kind-control-plane None     HBONE
backend             reviews-v1-cdd45ff95-lxhkb           10.244.0.40
kind-control-plane None     HBONE
backend             reviews-v2-78978699df-9454z          10.244.0.48
kind-control-plane None     HBONE
backend             reviews-v2-78978699df-lv99d          10.244.0.41
kind-control-plane None     HBONE
backend             reviews-v3-79ff749955-c597d          10.244.0.42
kind-control-plane None     HBONE
backend             reviews-v3-79ff749955-fm5lf          10.244.0.47
kind-control-plane None     HBONE
backend             sleep                                10.244.0.16
kind-control-plane None     TCP
backend             waypoint-69bbfbdfcb-9qqfg            10.244.0.43
kind-control-plane None     TCP
default             kubernetes                           172.18.0.2
None     TCP
frontend            productpage-v1-55586884d5-kz8tn      10.244.0.36
kind-control-plane None     HBONE
frontend            productpage-v1-55586884d5-mjntp      10.244.0.37
kind-control-plane None     HBONE
frontend            sleep                                10.244.0.15
kind-control-plane None     TCP
```

```
istioctl zc service
```

```
NAMESPACE      SERVICE NAME             SERVICE VIP   WAYPOINT ENDPOINTS
backend        details                  10.96.152.221 waypoint 2/2
backend        details-v1               10.96.175.236 waypoint 2/2
backend        ratings                  10.96.18.227  waypoint 2/2
backend        ratings-v1               10.96.171.69  waypoint 2/2
backend        reviews                  10.96.152.102 waypoint 6/6
backend        reviews-v1               10.96.94.80   waypoint 2/2
backend        reviews-v2               10.96.15.16   waypoint 2/2
backend        reviews-v3               10.96.119.175 waypoint 2/2
backend        waypoint                 10.96.54.173  None     1/1
default        kubernetes               10.96.0.1     None     1/1
frontend       productpage              10.96.239.190 None     2/2
istio-ingress  gateway-istio            10.96.186.15  None     1/1
istio-system   istiod                   10.96.240.238 None     1/1
kube-system    kube-dns                 10.96.0.10    None     2/2
metallb-system metallb-webhook-service 10.96.5.191   None     1/1
```

Similarly, if you run `istioctl zc cert`, you'll see that ztunnel issued a certificates for all workloads in the ambient mesh:

```
CERTIFICATE NAME                                               TYPE
STATUS         VALID CERT     SERIAL NUMBER                       NOT
AFTER                    NOT BEFORE
spiffe://cluster.local/ns/backend/sa/bookinfo-details          Leaf
Available      true           3a4ddbbaac79c9d8c62fc338ad608311
2024-10-10T23:23:29Z     2024-10-09T23:21:29Z
spiffe://cluster.local/ns/backend/sa/bookinfo-details          Root
Available      true           52e56cc8b52f41ff75649f759d702741
2034-10-07T22:48:53Z     2024-10-09T22:48:53Z
spiffe://cluster.local/ns/backend/sa/bookinfo-ratings          Leaf
Available      true           dddf6a026d8349ea12bce024b6a1de08
2024-10-10T23:23:29Z     2024-10-09T23:21:29Z
spiffe://cluster.local/ns/backend/sa/bookinfo-ratings          Root
```

```
Available     true           52e56cc8b52f41ff75649f759d702741
2034-10-07T22:48:53Z     2024-10-09T22:48:53Z
spiffe://cluster.local/ns/backend/sa/bookinfo-reviews          Leaf
Available     true           d65c6c34dbbf48ea9b6265b8ffd3f07c
2024-10-10T23:23:29Z     2024-10-09T23:21:29Z
spiffe://cluster.local/ns/backend/sa/bookinfo-reviews          Root
Available     true           52e56cc8b52f41ff75649f759d702741
2034-10-07T22:48:53Z     2024-10-09T22:48:53Z
spiffe://cluster.local/ns/frontend/sa/bookinfo-productpage     Leaf
Available     true           507dcf851d9c3624915e11c4a96aad32
2024-10-10T22:54:58Z     2024-10-09T22:52:58Z
spiffe://cluster.local/ns/frontend/sa/bookinfo-productpage     Root
Available     true           52e56cc8b52f41ff75649f759d702741
2034-10-07T22:48:53Z     2024-10-09T22:48:53Z
(base)
```

Testing

The last step is to validate all policies and routes are enforced correctly. We can start by testing the `productpage-viewer` policy:

```
kubectl exec -n frontend -it sleep -- curl -s -o /dev/null -w "%{http_
code}\n" -H "Host: bookinfo.example.com" productpage:9080/productpage
```

```
000
command terminated with exit code 56
```

```
From ztunnel:
...
2024-10-14T20:14:49.578208Z     error   access  connection
complete     src.addr=10.244.0.42:55494 src.workload="sleep" src.
namespace="frontend" src.identity="spiffe://cluster.local/ns/frontend/
sa/default" dst.addr=10.244.0.36:15008 dst.hbone_addr=10.244.0.36:9080
dst.service="productpage.frontend.svc.cluster.local" dst.
workload="productpage-v1-6c65c9f656-wl9c8" dst.namespace="frontend" dst.
identity="spiffe://cluster.local/ns/frontend/sa/bookinfo-productpage"
direction="outbound" bytes_sent=0 bytes_recv=0 duration="0ms" error="http
status: 401 Unauthorized"
...
```

The policy was enforced correctly and the request was denied. We can also test the `details-policy` policy:

```
kubectl exec -n frontend -it sleep -- curl -s -o /dev/null -w "%{http_
code}\n" -H "Host: bookinfo.example.com" details.backend:9080/details/1
```

```
403
```

```
From waypoint proxy:

[2024-10-14T20:15:48.344Z] "GET /details/1 HTTP/1.1" 403 - rbac_access_
denied_matched_policy[none] - "-" 0 19 0 - "-" "curl/8.10.1" "83c73503-
607e-4590-9de6-d8c243970763" "bookinfo.example.com" "-" inbound-
vip|9080|http|details.backend.svc.cluster.local - 10.96.104.243:9080
10.244.0.42:48646 - default
```

The request was denied as expected. We can also test the `ratings-policy` policy:

```
kubectl exec -n backend -it sleep -- curl -s -o /dev/null -w “%{http_
code}\n” -H “Host: bookinfo.example.com” ratings.backend:9080/ratings/1
```

```
403
```

```
From waypoint:
[2024-10-14T20:16:26.191Z] “GET /ratings/1 HTTP/1.1” 403 - rbac_access_
denied_matched_policy[none] - “-” 0 19 0 - “-” “curl/8.10.1” “f974c98a-
ce24-4d36-8619-800df3890f04” “bookinfo.example.com” “-” inbound-
vip|9080|http|ratings.backend.svc.cluster.local - 10.96.165.224:9080
10.244.0.41:49922 - default
```

Once we verified all policies are enforced correctly, we can safely remove the authorization policies that were enforced by the sidecar proxies:

```
kubectl delete authorizationpolicy -n backend details-policy
kubectl delete authorizationpolicy -n backend ratings-policy
```

Get Started with Ambient Mesh today

You can learn more about Ambient mesh at ambientmesh.io.

Want to start testing Ambient mesh? Learn how to implement it in your environment with our free on-demand labs here.

About Solo.io

Solo.io, the leading application networking company, delivers a service mesh and API platform for Kubernetes, zero trust, and microservices. The components of Gloo Gateway and Gloo Mesh enable enterprise companies to rapidly adopt microservice applications as part of their cloud journey and digital transformation. Solo.io delivers open source solutions, and is a community leader in building the technologies of the future.

SOLO.IO

contact@solo.io

www.solo.io

Learn more