Using Istio Traffic Management on Amazon EKS to Enhance User Experience

by Praseeda Sathaye, Sourav Paul, and Tiago Reichert | on 11 JAN 2024 | in [Amazon Elastic Kubernetes Service](https://aws.amazon.com/blogs/opensource/category/compute/amazon-kubernetes-service/), [Open Source](https://aws.amazon.com/blogs/opensource/category/open-source/), [Technical How-to](https://aws.amazon.com/blogs/opensource/category/post-types/technical-how-to/) | [Permalink](https://aws.amazon.com/blogs/opensource/using-istio-traffic-management-to-enhance-user-experience/) | [Comments](https://aws.amazon.com/blogs/opensource/using-istio-traffic-management-to-enhance-user-experience/#Comments) | [Share](https://aws.amazon.com/blogs/opensource/using-istio-traffic-management-to-enhance-user-experience/)

This is the second blog post in our series “Istio on EKS”, where we uncover Istio’s transformative benefits for microservices in [Amazon Elastic Kubernetes Service (Amazon EKS)](https://aws.amazon.com/eks/). In our previous blog, [Getting started with Istio on EKS](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/), we explained how to set up Istio on Amazon EKS. We covered core aspects such as Istio Gateway, Istio VirtualService, and observability with open source Kiali and Grafana.

In this blog, we’ll unlock the true potential of Istio as a service mesh by mastering Istio’s most powerful features for traffic management, the communication among microservices that is key to maintain the scalability and reliability of applications. From facilitating A/B testing and gradual rollouts to ensuring efficient load balancing, Istio routing offers indispensable capabilities. This blog delves into traffic management strategies to accomplish sophisticated testing and deployment strategies, downtime reduction, and user experience enhancement.

**Traffic Management with Istio on EKS**

In this blog post, we’ll dive into practical traffic management strategies such as path and weight-based routing, illuminate the concept of traffic mirroring, and venture into advanced scenarios like zone-aware routing.

1. Traffic Routing using Destination Rules: Destination rules for traffic routing allow you to control how traffic is directed to different versions or subsets of your microservices.
2. Weight-Based Routing: Weight-based routing in Istio controls and distributes traffic intelligently among different versions or subsets of your microservices.
3. Path-Based Routing: This creates rules that match specific URL paths and routes them to the appropriate service or subset of services.
4. Header-Based Routing: This lets you make routing decisions based on the content of HTTP headers in incoming requests.
5. Traffic Mirroring: With traffic mirroring, you will see how to duplicate incoming requests and send a copy to a designated destination for analysis, testing, or monitoring purposes.
6. Availability Zone Aware Routing: Last, you will learn about how to enable routing traffic to the pods or services within the same availability zone using the destination rule’s traffic policy.

**Deployment Architecture**

We’ll leverage the same microservices-based Product Catalog Application that we used in our previous blog [Getting Started with Istio on EKS](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/) that will serve as our practical playground. This will allow us to explore Istio’s capabilities in a hands-on manner. The application is composed of three types of microservices: [Frontend](https://github.com/aws-samples/istio-on-eks/tree/main/apps/frontend_node), [Product Catalog](https://github.com/aws-samples/istio-on-eks/tree/main/apps/product_catalog), and [Catalog Detail](https://github.com/aws-samples/istio-on-eks/tree/main/apps/catalog_detail) as shown in the Istio Data Plane in this diagram.

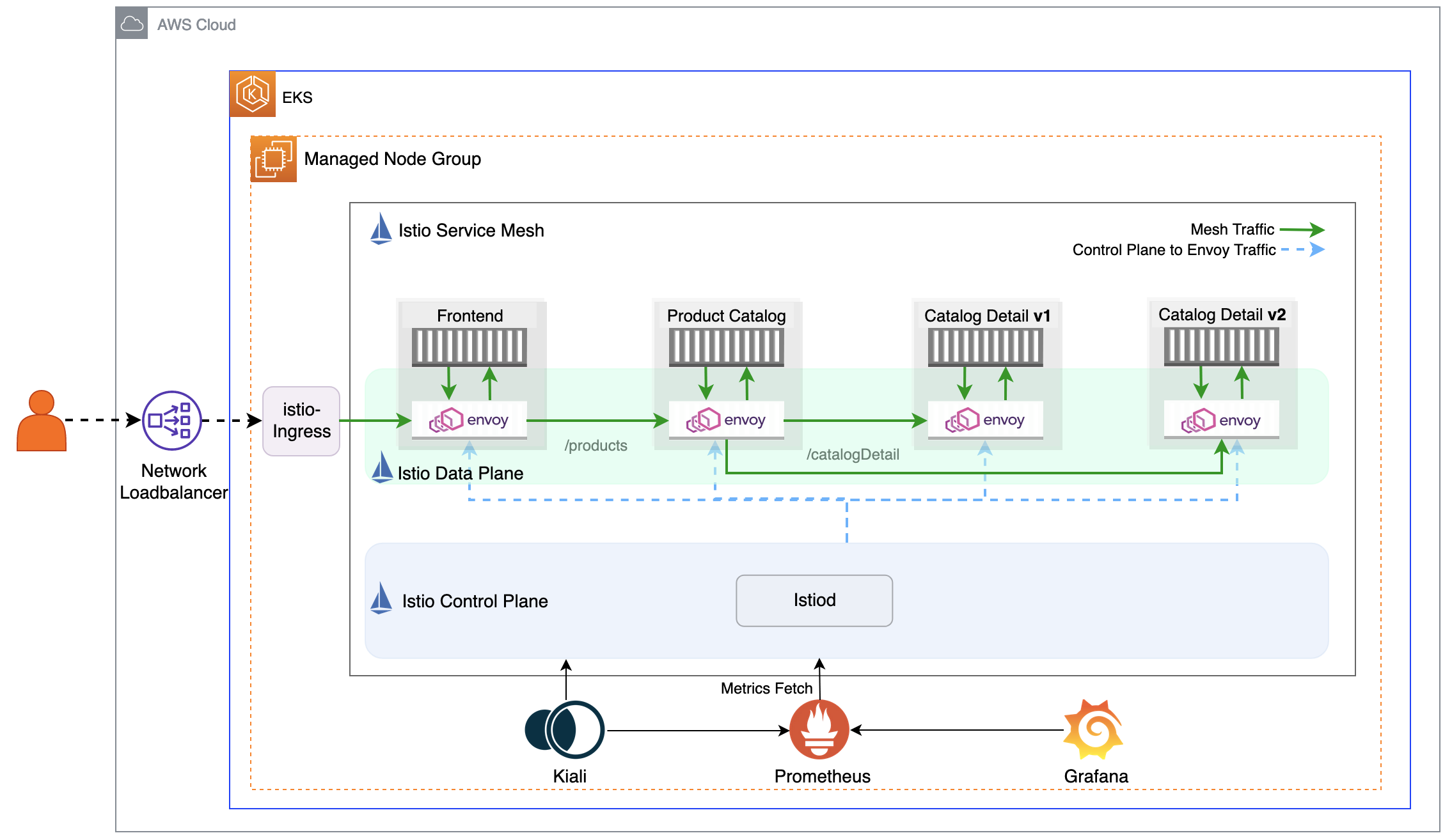


Image 1: Istio architectural overview

**Key Istio Components**

In our previous blog [Getting Started with Istio on EKS](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/), we learned about Istio VirtualService and Gateway. Now let’s dive deeper into another concept called [destination rules](https://istio.io/latest/docs/reference/config/networking/destination-rule/).

By defining destination rules, you can implement various routing strategies, such as canary deployments, A/B testing, and blue-green deployments, while also ensuring traffic reliability and fault tolerance. In Istio, destination rules work with virtual services to shape traffic behavior based on the criteria you specify, such as HTTP headers, request paths, or weighted traffic distribution. Think of a virtual service as a specific way that you route your traffic to a destination, and then you use destination rules to configure what happens to traffic for that destination. In particular, you use destination rules to specify named service subsets, such as grouping service instances by version. You can then use these service subsets in the routing rules of virtual services to control the traffic to different instances of your services. In destination rules, you can also specify a configuration for load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool. This image depicts the addition of virtual services between the ingress gateway, the frontend microservice, the productcatalog service, and the two catalogdetail services. A destination rule before the traffic reaches catalogdetail determines two groupings of microservices and to which it will route based on the defined rules.

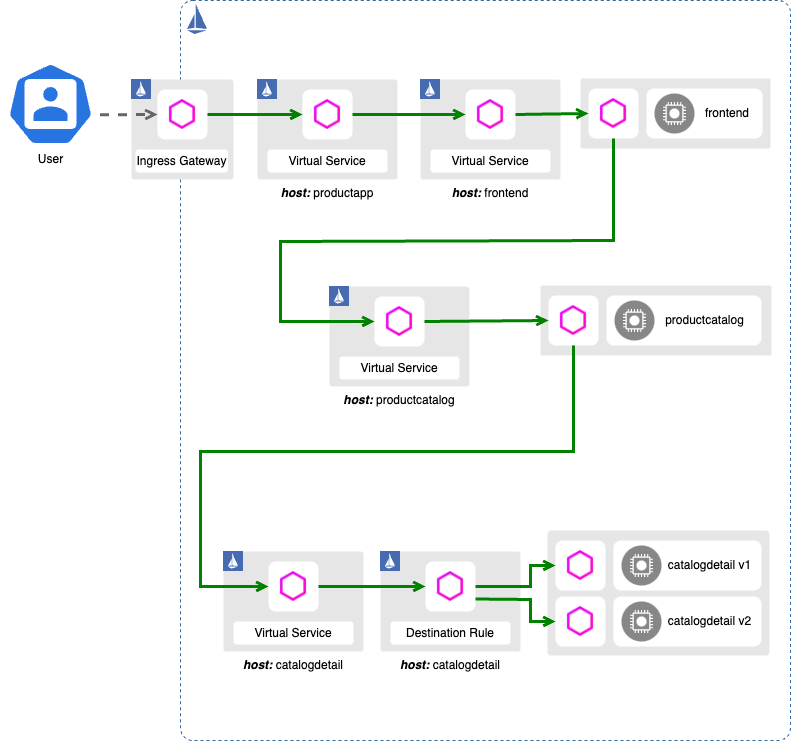


Image 2: Istio routing strategies using Virtual Service and Destination Rule

**Prerequisites and Initial Setup**

Before we proceed to the rest of this post, we need to make sure that the prerequisites are correctly installed. When complete, we will have the Amazon EKS cluster with Istio and the sample application configured.

First, clone the blog example repository:

git clone https://github.com/aws-samples/istio-on-eks.git

Then we need make sure to complete all the below steps. Note: These steps are from [Module 1 – Getting Started](https://github.com/aws-samples/istio-on-eks/tree/main/modules/01-getting-started) that was used in the first blog [Getting started with Istio on EKS](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/).

1. [Prerequisites](https://github.com/aws-samples/istio-on-eks/blob/main/modules/01-getting-started/README.md#prerequisites) – Install tools, set up Amazon EKS and Istio, configure istio-ingress and install Kiali using the same [Amazon EKS Istio Blueprints](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/) for Terraform that we used in the first blog. We will be using the Siege utility for testing throughout this blog and this tool needs to be installed as part the [Prerequisites](https://github.com/aws-samples/istio-on-eks/blob/main/modules/01-getting-started/README.md#prerequisites).
2. [Deploy](https://github.com/aws-samples/istio-on-eks/tree/main/modules/01-getting-started#deploy) – Deploy Product Catalog application resources and basic Istio resources using Helm.
3. [Configure Kiali](https://github.com/aws-samples/istio-on-eks/tree/main/modules/01-getting-started#configure-kiali) – Port forward to Kiali dashboard and Customize the view on Kiali Graph.

NOTE: Do not proceed if you don’t get the below result using the following command:

kubectl get pods -n workshop

NAME READY STATUS RESTARTS AGE

catalogdetail-658d6dbc98-q544p 2/2 Running 0 7m19s

catalogdetail2-549877454d-kqk9b 2/2 Running 0 7m19s

frontend-7cc46889c8-qdhht 2/2 Running 0 7m19s

productcatalog-5b79cb8dbb-t9dfl 2/2 Running 0 7m19s

**Traffic Routing Use Cases**

**Traffic Routing using Destination Rules**

Let’s create the VirtualService and DestinationRule for catalogdetail service as well as VirtualService for both productcatalog and frontend services in our product catalog application. We can checkout the details on these manifests [here](https://github.com/aws-samples/istio-on-eks/tree/main/modules/02-traffic-management/setup-mesh-resources).

# This assumes that we are currently in "istio-on-eks/modules/01-getting-started" folder

cd ../02-traffic-management

kubectl apply -f ./setup-mesh-resources/

# See DestinationRule and VirtualService we just created

kubectl get DestinationRule catalogdetail -n workshop -o yaml

kubectl get VirtualService catalogdetail -n workshop -o yaml

Looking at the DestinationRule described here, we notice that two subsets (v1 and v2) are created that represent individual versions of a service. Those subsets are not being used in the VirtualService. By default, Istio will route a similar amount of traffic to both versions.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: catalogdetail

namespace: workshop

spec:

host: catalogdetail.workshop.svc.cluster.local

subsets:

- name: v1

labels:

version: v1

- name: v2

labels:

version: v2

--

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: catalogdetail

namespace: workshop

spec:

hosts:

- catalogdetail

http:

- route:

- destination:

host: catalogdetail

port:

number: 3000

Let’s generate some traffic by running the following command in a separate terminal session:

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

While the load is being generated, access the Kiali console that we previously configured. In this Kiali dashboard diagram (Image 3), we will notice the traffic being distributed for catalogdetail service. The traffic is split evenly: 50% of the traffic goes to v1 and 50% to v2 of the catalogdetail microservice. The amount of requests per second (rps) is also similar for both versions.

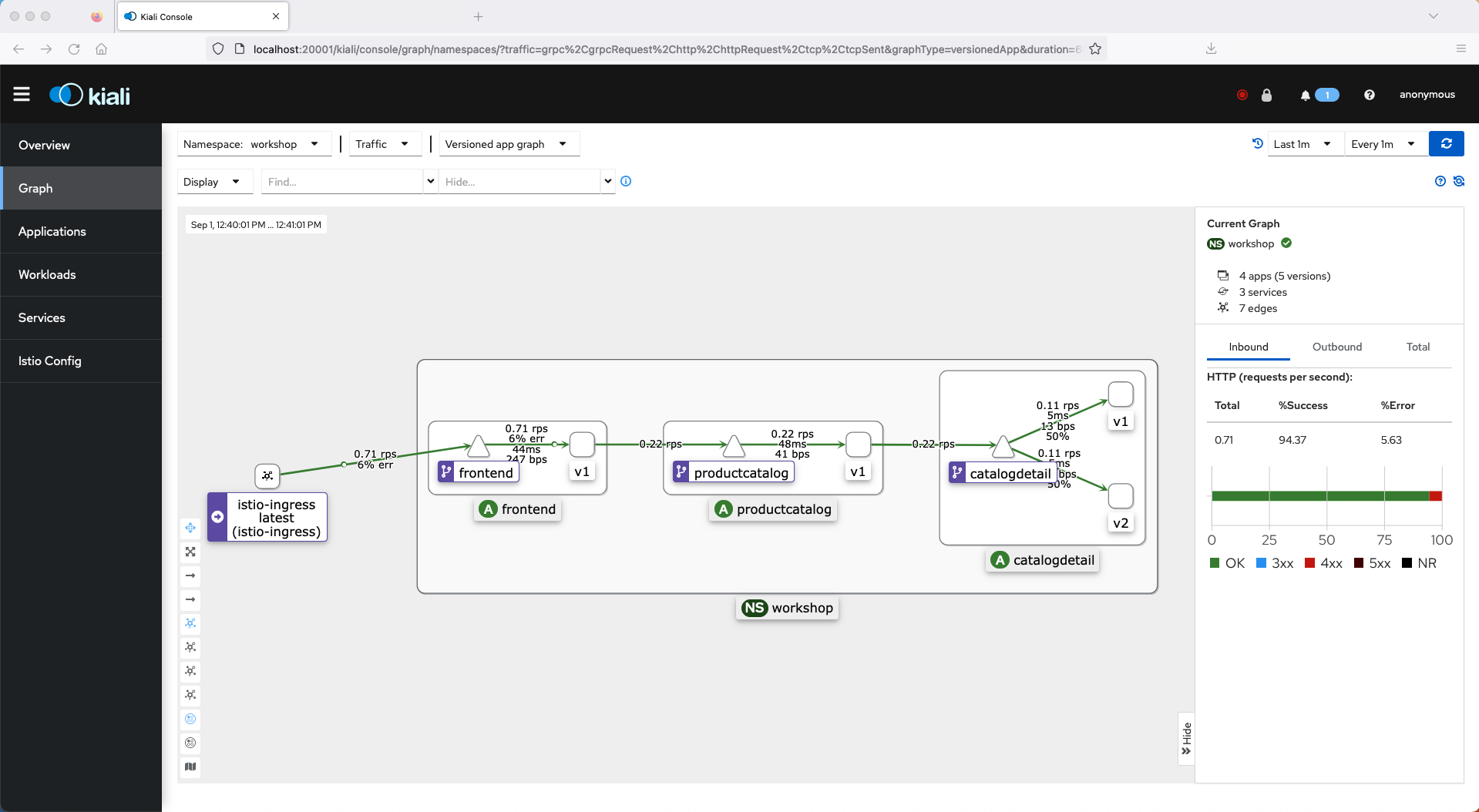


Image 3: Istio destination rule with even traffic distribution

**Route traffic to a specific app version using Specific Subset**

Now let’s change the destination for the VirtualService for the catalogdetail service to use only the subset v1 which is catalogdetail version V1

kubectl apply -f ./route-traffic-to-version-v1/catalogdetail-virtualservice.yaml

When we review the VirtualService with kubectl:

kubectl get VirtualService catalogdetail -n workshop -o yaml

The output will be similar to:

apiVersion: networking.istio.io/v1beta1

kind: VirtualService

metadata:

name: catalogdetail

namespace: workshop

spec:

hosts:

- catalogdetail

http:

- route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v1

If we run siege again, the Kiali dashboard (Image 4) shows that now the traffic is going to v1. It also shows v2 grayed out highlighting that no traffic is going to it.

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

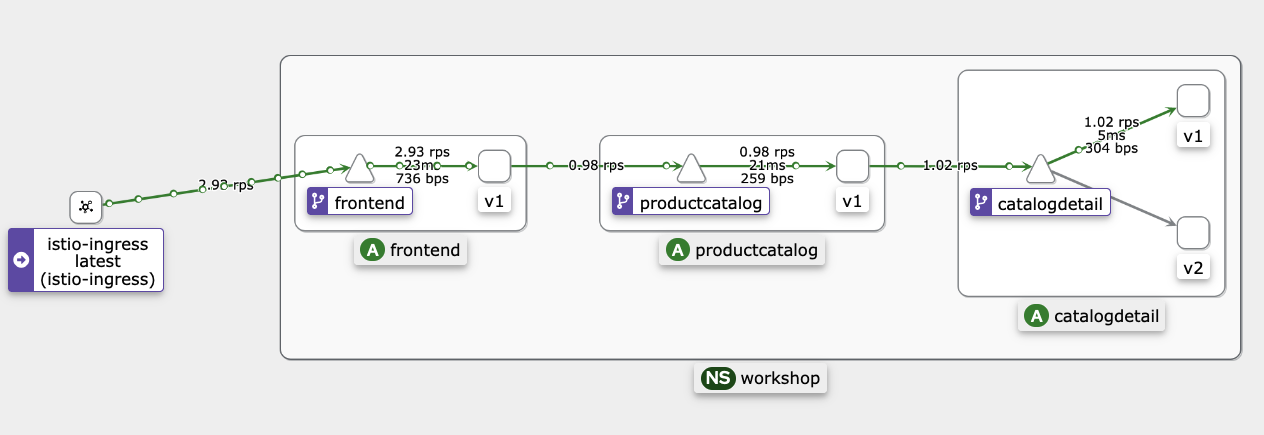


Image 4: Using weight based routing to send 100% of the traffic to catalogdetail subset v1

**Weight Based Routing**

A common scenario for [weight-based traffic management](https://istio.io/latest/docs/concepts/traffic-management/#more-about-routing-rules) is when you deploy a new version of your microservice and you want to test the new version before releasing all your customers to it, following a [canary release strategy](https://martinfowler.com/bliki/CanaryRelease.html). With weight-based routing, you can assign proportional weights to each subset, allowing you to gradually shift traffic from one version to another, perform A/B testing, or conduct canary deployments. By simply adjusting the weights, you can easily reroute traffic, monitor the performance of different subsets, and make data-driven decisions to optimize your microservices’ behavior.

In this scenario, we want to shift approximately 10% of the traffic sent to the catalogdetail VirtualService to version v2 and the rest to version v1. We achieve this by defining a destination for each of the subsets in the route definition in the VirtualService and setting their corresponding weights for the two subsets. The VirtualService manifest is shown here, note that the weight for v1 is set to 90, while the weight for v2 is set to 10.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: catalogdetail

namespace: workshop

spec:

hosts:

- catalogdetail

http:

- route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v1

weight: 90

- destination:

host: catalogdetail

port:

number: 3000

subset: v2

weight: 10

Let’s apply the VirtualService changes:

kubectl apply -f ./weight-based-routing/catalogdetail-virtualservice.yaml

Now, we review the VirtualService we applied with kubectl:

kubectl describe VirtualService catalogdetail -n workshop

The output should reflect the new weights:

Name: catalogdetail

Namespace: workshop

Labels: <none>

Annotations: <none>

API Version: networking.istio.io/v1beta1

Kind: VirtualService

Spec:

Hosts:

catalogdetail

Http:

Route:

Destination:

Host: catalogdetail

Port:

Number: 3000

Subset: v1

Weight: 90

Destination:

Host: catalogdetail

Port:

Number: 3000

Subset: v2

Weight: 10

Events: <none>

Use the siege command-line tool to generate traffic by running the following command in a separate terminal session.

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

Now the Kiali dashboard (Image 5) shows that the traffic is being routed to v1 and v2 based on the weights we configured on the VirtualService.

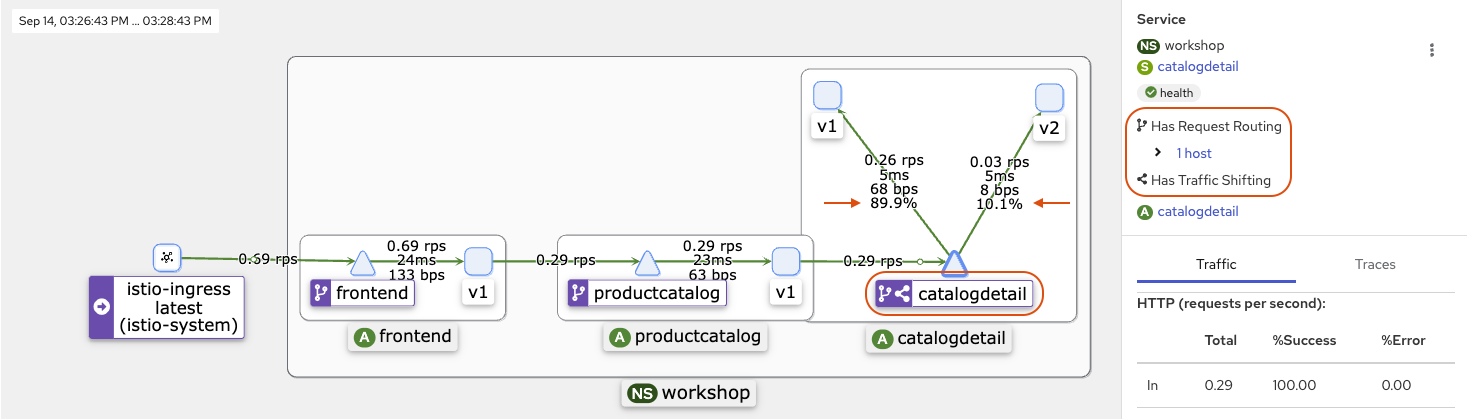


Image 5: Using weight based routing to send 90% of the traffic to subset v1 and 10% to subset v2

**Path Based Routing**

Path based routing on Istio is a versatile feature that allows you to direct incoming traffic to different microservices based on the request path. You might want to keep both versions of your microservice running for a longer period. This is commonly used when you have breaking changes to your API contract and want to enable consumers to gradually upgrade to the new version of your application. In this example, we will set up the application to use both v1 and v2 versions of the catalogdetail microservice.

To start, we need to apply the VirtualService to send requests for the path /v2/catalogDetail to the subset v2 and /v1/catalogDetail to subset v1.

kubectl apply -f ./path-based-routing/catalogdetail-virtualservice.yaml

Run kubectl to review the VirtualService:

# Describe the VirtualService

kubectl describe VirtualService catalogdetail -n workshop

The version of the catalogdetail VirtualService for this scenario defines a route each for both v1 and v2 subsets. The first route defines a match on request uri path and does an exact equality check for /v2/catalogDetail. It then sends all matching traffic to the v2 subset. The second route defines another exact uri match for /v1/catalogDetail and sends all matching traffic to the v1 subset. The productcatalog service is updated to point to either http://<host>:<port>/v2/catalogDetail or http://<host>:<port>/v1/catalogDetail by changing the AGG\_APP\_URL environment variable. Note that the /v1 and /v2 path prefixes are logical path components introduced to test path based routing by the VirtualService.

The destination catalogdetail service versions have not been updated to recognize these path prefixes. Hence if a matched request from productcatalog is forwarded unchanged to the corresponding destination subset, then it will cause a HTTP 404 Not Found error. To avoid this, we leverage the rewrite block to override the destination request URI path for both route definitions to /catalogDetail. This also showcases how Istio makes it easy to implement logical path based routing with minimal application changes.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: catalogdetail

namespace: workshop

spec:

hosts:

- catalogdetail

http:

- match:

- uri:

exact: /v2/catalogDetail

rewrite:

uri: /catalogDetail

route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v2

- match:

- uri:

exact: /v1/catalogDetail

rewrite:

uri: /catalogDetail

route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v1

To test the routing changes, let’s first configure the productcatalog microservice to use the URI path /v2/catalogDetail to invoke the catalogdetail microservice.

kubectl set env deployment/productcatalog -n workshop AGG\_APP\_URL=http://catalogdetail.workshop.svc.cluster.local:3000/v2/catalogDetail

The output will be similar to:

deployment.apps/productcatalog env updated

If we run siege, the dashboard (Image 6) shows that now the traffic is going to v2:

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

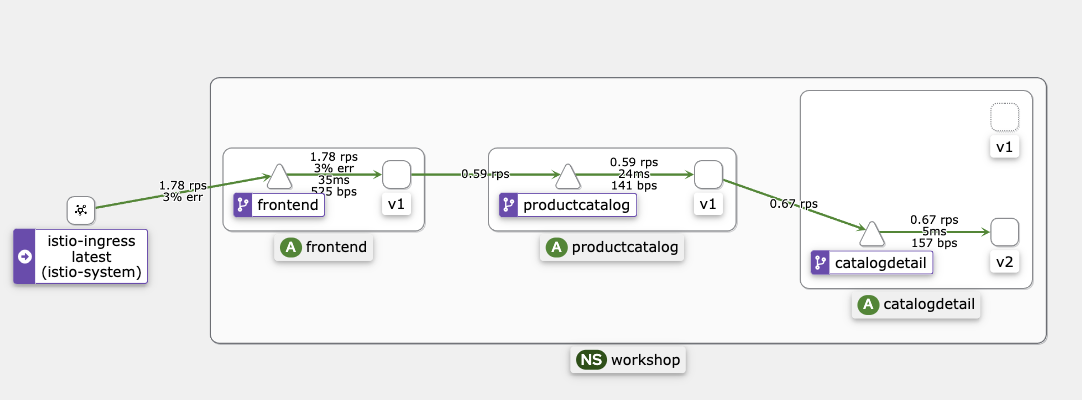


Image 6: Using path based routing to send traffic to subset v2 using AGG\_APP\_URL variable

If we switch to the URI for /v1/catalogDetail and run siege again, we will observe that now the traffic is going to v1. Set the environment variable to point to v1 URL.

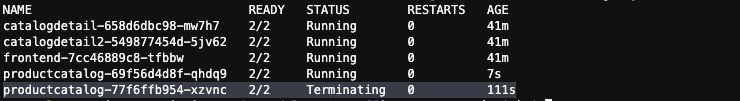
kubectl set env deployment/productcatalog -n workshop AGG\_APP\_URL=http://catalogdetail.workshop.svc.cluster.local:3000/v1/catalogDetail

The output will look similar to:

deployment.apps/productcatalog env updated

Ensure that the old productcatalog pod is terminated before running siege to avoid any traffic getting routed to the old pod.

kubectl get pods -n workshop



Once the old pod is terminated, run siege again.

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

The new siege run will cause a similar traffic pattern as shown in the diagram (Image 7) and we can observe that now the traffic is going to v1.

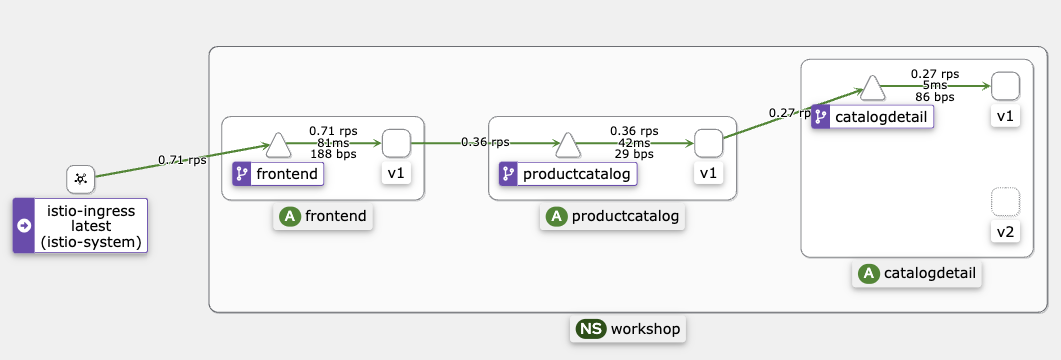


Image 7: Using path based routing to send traffic to catalogdetail subset v1 using AGG\_APP\_URL variable

At this point, we will revert to the initial configuration:

# Revert the environment variable for productcatalog

kubectl set env deployment/productcatalog -n workshop AGG\_APP\_URL=http://catalogdetail.workshop.svc.cluster.local:3000/catalogDetail

**Header Based Routing**

The same strategy we discussed for path-based routing can also be implemented with header based routing. With header-based routing, you can direct traffic to specific microservices or service versions by examining headers like user-agent, content-type, or custom headers. This capability is valuable for implementing complex routing scenarios where you need to cater to different clients, devices, or application versions.

To start, we’ll apply this change to move to header based routing:

kubectl apply -f ./header-based-routing/catalogdetail-virtualservice.yaml

Next, review the VirtualService to see the change:

# Describe the VirtualService

kubectl describe VirtualService catalogdetail -n workshop

Here we introduce a custom header called user-type for the catalogdetail VirtualService to distinguish traffic coming from internal users vs. external users. We want to route all requests from internal users to v2 version of catalogdetail. Other requests from external users will continue to flow to the v1 subset.

The updated catalogdetail VirtualService manifest is shown here. Note that the VirtualService now defines a route with a match on request headers that tests the user-type header value for an exact match on the string literal value internal. If an incoming request passes the test then it will be routed to the v2 subset of catalogdetail. The default route at the end points to subset v1. This means that any other value of the request header traffic will be forwarded to the v1 subset.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: catalogdetail

namespace: workshop

spec:

hosts:

- catalogdetail

http:

- match:

- headers:

user-type:

exact: internal

route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v2

- route:

- destination:

host: catalogdetail

port:

number: 3000

subset: v1

For demonstration purposes, we will use an [EnvoyFilter](https://istio.io/latest/docs/reference/config/networking/envoy-filter/) with the productcatalog service to add the custom request header user-type to all upstream requests to catalogdetail service. We will also set its value to internal for approximately 30% of the traffic and to external for the remaining 70% traffic.

The manifest for the EnvoyFilter is shown here. It implements a simple inline [Lua script](https://www.lua.org/pil/1.html) to add a USER-TYPE request header and randomly sets the value to internal 30% of the time. The rest of the time it sets the header value to external. The filter applies to the envoy sidecar of the productcatalog pod using the spec.workloadSelector.labels field and intercepts all outbound HTTP requests to the upstream service which in this case is catalogdetail. This also showcases how Istio makes it easy to perform header manipulation with minimal application changes.

apiVersion: networking.istio.io/v1alpha3

kind: EnvoyFilter

metadata:

name: productcatalog

namespace: workshop

spec:

workloadSelector:

labels:

app: productcatalog

configPatches:

- applyTo: HTTP\_FILTER

match:

context: SIDECAR\_OUTBOUND

listener:

filterChain:

filter:

name: "envoy.filters.network.http\_connection\_manager"

subFilter:

name: "envoy.filters.http.router"

patch:

operation: INSERT\_BEFORE

value:

name: envoy.filters.http.lua

typed\_config:

"@type": "type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua"

defaultSourceCode:

inlineString: |-

function envoy\_on\_request(request\_handle)

math.randomseed(os.clock()\*100000000000);

local r = math.random(1, 100);

if r <= 30 then

request\_handle:headers():add("USER-TYPE", "internal");

else

request\_handle:headers():add("USER-TYPE", "external");

end

end

Now, we’ll apply the EnvoyFilter to the productcatalog sidecars.

kubectl apply -f ./header-based-routing/productcatalog-envoyfilter.yaml

# Describe the EnvoyFilter

kubectl describe EnvoyFilter productcatalog -n workshop

If we run siege, we will see that the approximately 70% of the traffic is going to v1 for the request with a header of USER-TYPE “external” and 30% going to v2 for the request with a header of USER-TYPE “internal”, as shown in Image 8.

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

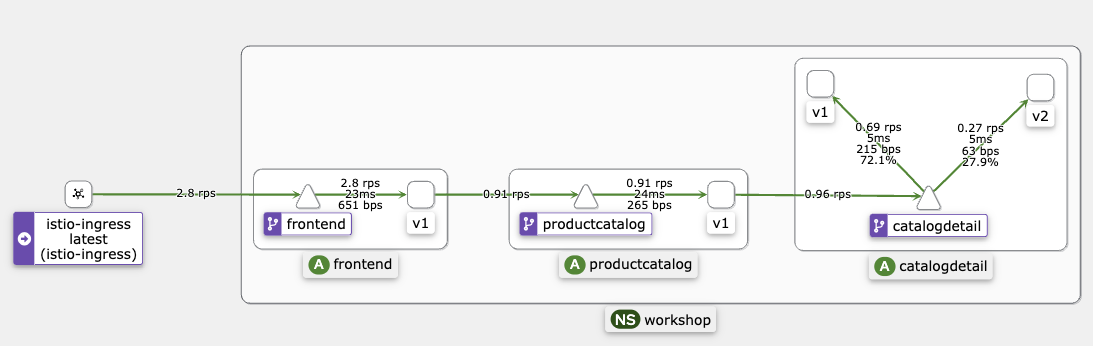


Image 8: Using header based routing to send 70% of the traffic to catalogdetail subset v1 and 30% to subset v2

Revert to the initial configuration to reset for the next option:

# Delete EnvoyFilter

kubectl delete -f ./header-based-routing/productcatalog-envoyfilter.yaml

**Traffic Mirroring**

Another option to test a new release is by using traffic mirroring, also called shadowing. It sends a copy of live traffic to a mirrored service. The mirrored traffic does not affect the response of the primary service. Using this strategy, you can test how the new version is behaving with real traffic without affecting your customers if the new release is not responding correctly. This feature is useful for scenarios like debugging, security analysis, or evaluating the performance of a new service version. By mirroring traffic, you can observe how changes or updates might affect your application without exposing users to potential issues.

We have two versions of catalogdetail service v1 and v2. To demonstrate traffic mirroring, we’ll configure the VirtualService to route 100% of the traffic to v1 and also add a rule to specify that we want to mirror (i.e., also send) 50% of the same traffic to the v2 service.

kubectl apply -f ./traffic-mirroring/catalogdetail-virtualservice.yaml

# Describe the VirtualService

kubectl describe VirtualService catalogdetail -n workshop

The output should look like this. Note the Mirror section, along with the full weight on v1.

Name: catalogdetail

Namespace: workshop

Labels: <none>

Annotations: <none>

API Version: networking.istio.io/v1beta1

Kind: VirtualService

Spec:

Hosts:

catalogdetail

Http:

Mirror:

Host: catalogdetail

Port:

Number: 3000

Subset: v2

Mirror Percentage:

Value: 50

Route:

Destination:

Host: catalogdetail

Port:

Number: 3000

Subset: v1

Weight: 100

We use siege again to generate traffic in a separate terminal session.

ISTIO\_INGRESS\_URL=$(kubectl get svc istio-ingress -n istio-ingress -o jsonpath='{.status.loadBalancer.ingress[\*].hostname}')

siege http://$ISTIO\_INGRESS\_URL -c 5 -d 10 -t 2M

Now Kiali (Image 9) shows that the traffic is being routed to v1 and 50% of the traffic is also being mirrored to v2 (the responses from v2 are discarded.)

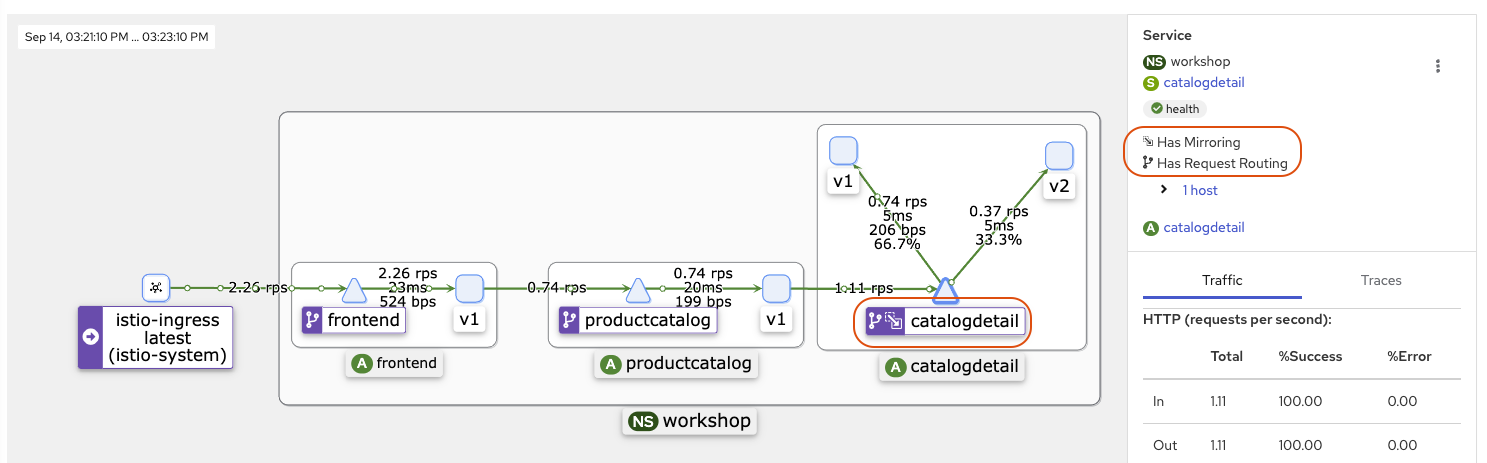


Image 9: Using traffic mirroring routing to send a copy of 50% of the traffic to catalogdetail subset v2

**Locality Load Balancing**

Istio’s [Locality Load Balancing](https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/) is a feature that optimizes traffic routing based on the geographic proximity of services, including Availability Zones and regions. Istio is configured with knowledge of the cluster’s topology, including the locations of Availability Zones. It knows which services or instances are in different zones. When routing requests, Istio prioritizes directing traffic to services that are in the same or nearby Availability Zones. This minimizes latency and optimizes network performance.

In the context of [Locality Failover](https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/failover/#configure-locality-failover), Istio recognizes that services within a locality (Region, Zone, Sub-zone) can sometimes become unavailable due to various issues. To maintain service availability and reliability, Istio can automatically perform failover to services in other localities while still ensuring that traffic prioritizes the nearest endpoints.

Implementing Locality Failover on Istio involves configuring [traffic policy](https://istio.io/latest/docs/reference/config/networking/destination-rule/#TrafficPolicy) and [outlier detection](https://istio.io/latest/docs/reference/config/networking/destination-rule/#OutlierDetection) in the destination rule to ensure that traffic is intelligently redirected to alternate endpoints when services in a specific locality (e.g., Availability Zone) experience issues. For an in-depth guide to optimize your routing decisions based on the Availability Zone, check out our blog [Addressing latency and data transfer costs on EKS using Istio](https://aws.amazon.com/blogs/containers/addressing-latency-and-data-transfer-costs-on-eks-using-istio/).

**Cleanup**

To clean up the Amazon EKS environment and remove the services we deployed, please run the following commands:

kubectl delete namespace workshop

To further remove the Amazon EKS cluster with deployed Istio that you might have created in the prerequisite step, go to the terraform Istio folder ($YOUR-PATH/terraform-aws-eks-blueprints/patterns/istio) and run the commands provided [here](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/#destroy).

**Conclusion**

In this post, we covered traffic management strategies that can be used with Istio to help implement sophisticated deployment strategies, reduce downtime, and improve the user experience. We hope you’ve gained valuable insights into the potential of Istio within Amazon EKS. But hold on to that excitement because the adventure is far from over. In our next post of this series, we’ll delve deeper into Istio’s advanced features, exploring topics of security, resiliency, and the art of fine-tuned observability. Get ready to level up your microservices game with Istio and stay tuned for the next thrilling chapter!