Enhancing Network Resilience with Istio on Amazon EKS

<https://aws.amazon.com/blogs/opensource/enhancing-network-resilience-with-istio-on-amazon-eks/>

by Praseeda Sathaye, Ravi Thakur, Sagar Das, and Sreedevi Velagala | on 06 MAY 2024 | in [Amazon Elastic Kubernetes Service](https://aws.amazon.com/blogs/opensource/category/compute/amazon-kubernetes-service/), [Best Practices](https://aws.amazon.com/blogs/opensource/category/post-types/best-practices/), [Open Source](https://aws.amazon.com/blogs/opensource/category/open-source/) | [Permalink](https://aws.amazon.com/blogs/opensource/enhancing-network-resilience-with-istio-on-amazon-eks/) | [Comments](https://aws.amazon.com/blogs/opensource/enhancing-network-resilience-with-istio-on-amazon-eks/#Comments) | [Share](https://aws.amazon.com/blogs/opensource/enhancing-network-resilience-with-istio-on-amazon-eks/)

This is the third blog post of our “Istio on EKS” series, where we will continue to explore Istio’s network resilience capabilities and demonstrate how to set up and configure these features on Amazon Elastic Kubernetes Service (Amazon EKS). Istio equips microservices with a robust set of features designed to maintain responsive and fault-tolerant communication, even in the face of unexpected events. From timeouts and retries to circuit breakers, rate limiting, and fault injection, Istio’s network resilience features provide a comprehensive safety net, ensuring that your applications continue to operate smoothly, minimizing downtime and enhancing the overall user experience. In this blog, we’ll explore the concept of network resilience with Istio and demonstrate how to set up and configure these vital features on Amazon EKS.

In our first 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 the second blog [Using Istio Traffic Management on Amazon EKS to Enhance User Experience](https://aws.amazon.com/blogs/opensource/using-istio-traffic-management-to-enhance-user-experience/), we explained traffic management strategies to accomplish sophisticated testing and deployment strategies, downtime reduction, and user experience enhancement for the communication among microservices.

Resilience refers to the ability of the service mesh to maintain stable and responsive communication between microservices, even in the face of failures, disruptions, or degraded network conditions. Istio provides several features and mechanisms to enhance network resiliency within a microservices architecture. These include timeouts, retries, circuit breaking, fault injection and rate limiting that collectively contribute to ensuring the reliability and robustness of the communication between services. The goal is to prevent cascading failures, improve fault tolerance, and maintain overall system performance in the presence of various network challenges.

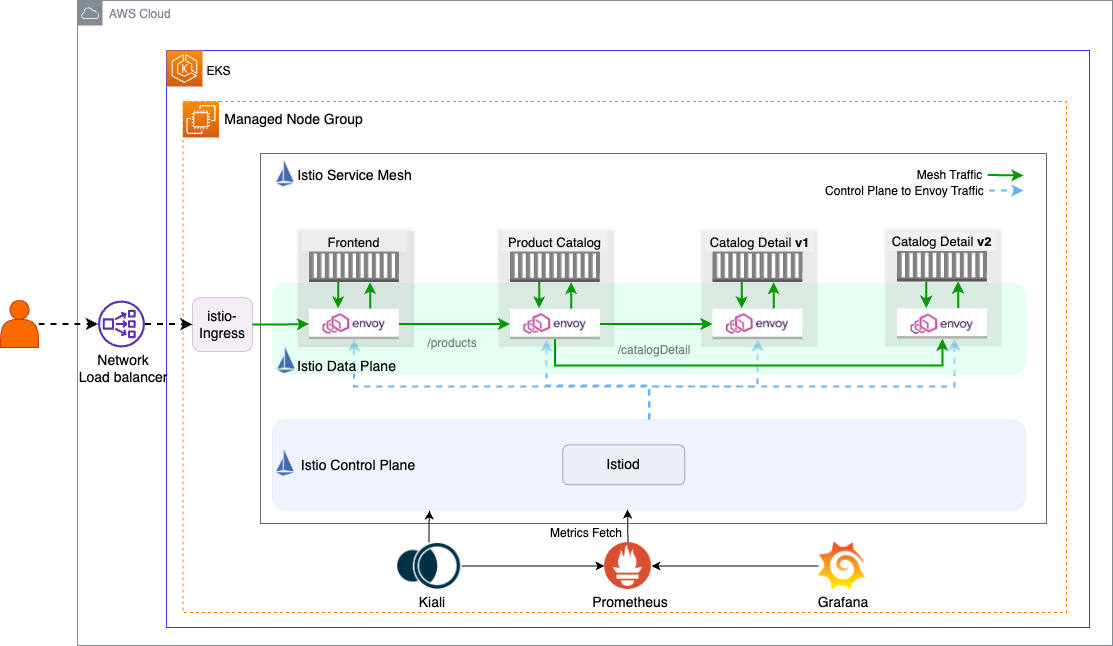
**Network Resilience with Istio on Amazon EKS**

In this part of the blog series, we’ll focus on Istio’s practical network resilience features that prevent localized failures from spreading to other nodes. Resilience ensures the overall reliability of applications remains high.

1. Fault Injection: [Fault injection](https://istio.io/latest/docs/tasks/traffic-management/fault-injection/) helps test failure scenarios and service-to-service communication by intentionally introducing errors in a controlled way. This helps find and fix problems before they cause outages in production, such as network outages, hardware failures, software failures, and human error.
2. Timeouts: [Timeouts](https://istio.io/latest/docs/tasks/traffic-management/request-timeouts/) are an important component of network resilience. They allow you to set a time limit for an operation or request to complete. If the operation or request does not complete within the specified time limit, it is considered a failure.
3. Retries: [Retries](https://istio.io/latest/docs/concepts/traffic-management/#retries) in Istio involves the automatic reattempt of a failed request to improve the resilience and availability of microservices-based applications.
4. Circuit breaker: [Circuit breaker](https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/) serves as a resilience mechanism for microservices. It will prevent continuous retries to a failing service, preventing overload and facilitating graceful degradation.
5. Rate Limiting: [Rate limiting](https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/) gives flexibility to apply customized throttling rules to avoid overloading services and restricting usage where necessary. This helps improve application stability and availability.

**Deployment Architecture**

We’ll leverage the same microservices-based product catalog application that we used in our first 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.



**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 an 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 to complete all the following 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 Istio 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 [Amazon EKS Istio Blueprints](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/) for Terraform that we used in the first blog.
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.
4. Install istioctl and add it to the $PATH

*NOTE: Do not proceed if you don’t get the result here 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

*Note: Do not execute the*[*Destroy*](https://github.com/sridevi1209/istio-on-eks/blob/network-resiliency/modules/01-getting-started/README.md#destroy)*section in Module 1!*

**Initialize the Istio service mesh resources**

The following command will create all the Istio resources required for the product catalog application. You can check the “Key Istio Components” explained in our previous blog [Using Istio Traffic Management on Amazon EKS to Enhance User Experience](https://aws.amazon.com/blogs/opensource/using-istio-traffic-management-to-enhance-user-experience/)

cd ../03-network-resiliency

kubectl apply -f ../00-setup-mesh-resources/

**Network Resilience Use cases**

Now we are going to explore various features in Istio that help manage network resilience. Throughout each section, we’ll provide step-by-step instructions on how to apply these features to your Istio service mesh. At the end of each section, we’ll also show you how to reset the environment, ensuring a clean slate before moving on to the next feature. By the end of this process, you’ll have a comprehensive overview of how Istio helps manage network resilience by providing features like fault injection, timeouts, retries, circuit breakers, and rate limiting.

**Fault Injection**

Fault injection in Istio is a powerful tool to validate and enhance the resiliency of your applications, helping prevent costly outages and disruptions by proactively identifying and addressing problems. This technique utilizes two types of faults, *Delays* and *Aborts*, both configured through a Virtual Service. Unlike other approaches that primarily target network-level disruptions, Istio allows fault injection at the application layer, enabling precise testing of specific HTTP error codes to yield more meaningful insights.

In practical usage, fault injection serves to assess applications under various conditions. This includes testing responses during sudden increases in demand by introducing delays and aborts, evaluating how applications handle database disruptions through simulated connection failures and slow responses, and assessing network connectivity in scenarios involving packet loss or network disconnections via delays and aborts.

**Injecting Delay Fault into HTTP Requests**

*Delays* are essential for fault injection, closely simulating timing failures akin to network latency increases or upstream service overload. This aids in assessing application responses to slow communication or response times, strengthening system resilience. In this section, we’ll provide a step-by-step guide to injecting delay faults into HTTP requests.

We will be updating the catalogdetail VirtualService for this use case as shown here:

..

fault:

delay:

percentage:

value: 100

fixedDelay: 15s

route:

- destination:

host: catalogdetail

port:

number: 3000

...

In VirtualService, the fault field injects delays for traffic to catalogdetail service for an internal user. Delays can be fixed or percentage-based. The above config applies a 15-second delay to 100% of requests to catalogdetail.

Create this fault injection rule for delaying traffic to catalogdetail:

cd fault-injection/

kubectl apply -f ./delay/catalogdetail-virtualservice.yaml

Test the delay by running a curl command against the catalogdetail service for the users named internal and external.

export FE\_POD\_NAME=$(kubectl get pods -n workshop -l app=frontend -o jsonpath='{.items[].metadata.name}')

kubectl exec -it ${FE\_POD\_NAME} -n workshop -c frontend -- bash

curl http://catalogdetail:3000/catalogdetail/ -s -H "user: internal" -o /dev/null \

-w "Time taken to start transfer: %{time\_starttransfer}\n"

The output should be similar to what’s shown here. A 15-second delay is introduced for the ‘internal’ user as per the catalogdetail virtual service’s delay fault config.

Time taken to start transfer: 15.009529

Run the curl command for the user named external (could be any user other than internal.)

curl http://catalogdetail:3000/catalogdetail/ -s -H "user: external" -o /dev/null \

-w "Time taken to start transfer: %{time\_starttransfer}\n"

The output should be similar to what’s shown here. There should be no delay for ‘external’ user as the catalogdetail virtual service’s delay fault config only targets ‘internal’ users.

Time taken to start transfer: 0.006548

Now, to exit from the shell inside the frontend container first press Enter, then type ‘exit’ and press Enter once more.

By introducing a delay for catalogdetail service, delay is injected into the traffic before it reaches the service, affecting a specific user or set of users. Through intentional testing, we can effectively assess the application’s resiliency, conducting experiments within a defined user scope.

**Injecting Abort Fault into HTTP Requests**

*Aborts* are vital for fault injection, mimicking upstream service crash failures often seen as HTTP error codes or TCP connection issues. By introducing aborts into HTTP requests, we can replicate these scenarios, enabling comprehensive testing to bolster application resilience. In this section, we’ll provide a step-by-step guide to injecting abort faults into HTTP requests.

We will be updating the catalogdetail VirtualService for this use case as shown here:

...

fault:

abort:

percentage:

value: 100

httpStatus: 500

route:

- destination:

host: catalogdetail

port:

number: 3000

...

In VirtualService, the fault field injects an abort for catalogdetail service traffic to the internal user. The httpStatus 500 means clients get a 500 Internal Server Error response.

Create this fault injection rule to abort traffic directed towards catalogdetail.

kubectl apply -f ./abort/catalogdetail-virtualservice.yaml

Test the abort by running a curl command against the catalogdetail service for the users named internal and external.

kubectl exec -it ${FE\_POD\_NAME} -n workshop -c frontend -- bash

curl http://catalogdetail:3000/catalogdetail/ -s -H "user: internal" -o /dev/null \

-w "HTTP Response: %{http\_code}\n"

The output should be similar to what’s shown here. You should see HTTP 500 error (Abort) for the ‘internal’ user per catalogdetail virtual service’s abort fault config.

HTTP Response: 500

Run the curl command for the user named external (could be any user other than internal.)

curl http://catalogdetail:3000/catalogdetail/ -s -H "user: external" -o /dev/null \

-w "HTTP Response: %{http\_code}\n"

The output should be similar to what’s shown here. You should see HTTP 200 success for the ‘external’ user as catalogdetail virtual service’s abort fault config targets only ‘internal’ users.

HTTP Response: 200

Now, to exit from the shell inside the frontend container first press Enter, then type ‘exit’ and press Enter once more.

By introducing an abort for the catalogdetail service, an HTTP abort fault is injected into the traffic before it reaches the service for a specific user or set of users. Through intentional testing, we can effectively assess the application’s resiliency, conducting experiments within a defined user scope.

**Reset the environment**

Reset the configuration to the initial state by running the following command from the fault-injection directory:

kubectl apply -f ../../00-setup-mesh-resources/

**Timeouts**

In Istio, timeouts refer to the maximum duration a service or proxy waits for a response from another service before marking the communication as failed. The timeouts help to manage and control the duration of requests, contributing to the overall reliability and performance of the microservices communication within the Istio service mesh.

To test the timeout functionality we will make a call from the productcatalog service to the catalogdetail service. We will be using the fault injection approach that we explored before to introduce delay in thecatalogdetail service. Then we will add a timeout to the productcatalog service.

1. Let’s apply the 5 second delay to the catalogdetail virtual service.

...

spec:

...

- fault:

...

fixedDelay: 5s

route:

...

# This assumes that you are currently in "istio-on-eks/modules/03-network-resiliency" folder

cd timeouts-retries-circuitbreaking

kubectl apply -f ./timeouts/catalogdetail-virtualservice.yaml

Test the delay by running a curl command against the productcatalog service from within the mesh.

# Set the FE\_POD\_NAME variable to the name of the frontend pod in the workshop namespace

export FE\_POD\_NAME=$(kubectl get pods -n workshop -l app=frontend -o jsonpath='{.items[].metadata.name}')

# Access the frontend container in the workshop namespace interactively

kubectl exec -it ${FE\_POD\_NAME} -n workshop -c frontend -- bash

root@frontend-container-id:/app#

# Allows accessing the shell inside the frontend container for executing commands

curl http://productcatalog:5000/products/ -s -o /dev/null -w "Time taken to start transfer: %{time\_starttransfer}\n"

If the delay configuration is applied correctly, the output should be similar to this:

Time taken to start transfer: 5.024133

2. Apply the timeout of 2 seconds to the productcatalog virtual service.

...

spec:

...

route:

- destination:

...

timeout: 2s

# This assumes that you are currently in "istio-on-eks/modules/03-network-resiliency/timeouts-retries-circuitbreaking" folder

kubectl apply -f ./timeouts/productcatalog-virtualservice.yaml

With a 2-second timeout set for the productcatalog service, any calls made to the catalogdetail service, which typically has a response time of approximately 5 seconds, will inevitably exceed the designated timeout threshold, resulting in triggered timeouts for those calls.

Test the timeout by running a curl command against the productcatalog service from within the mesh.

# Access the frontend container in the workshop namespace interactively

kubectl exec -it ${FE\_POD\_NAME} -n workshop -c frontend -- bash

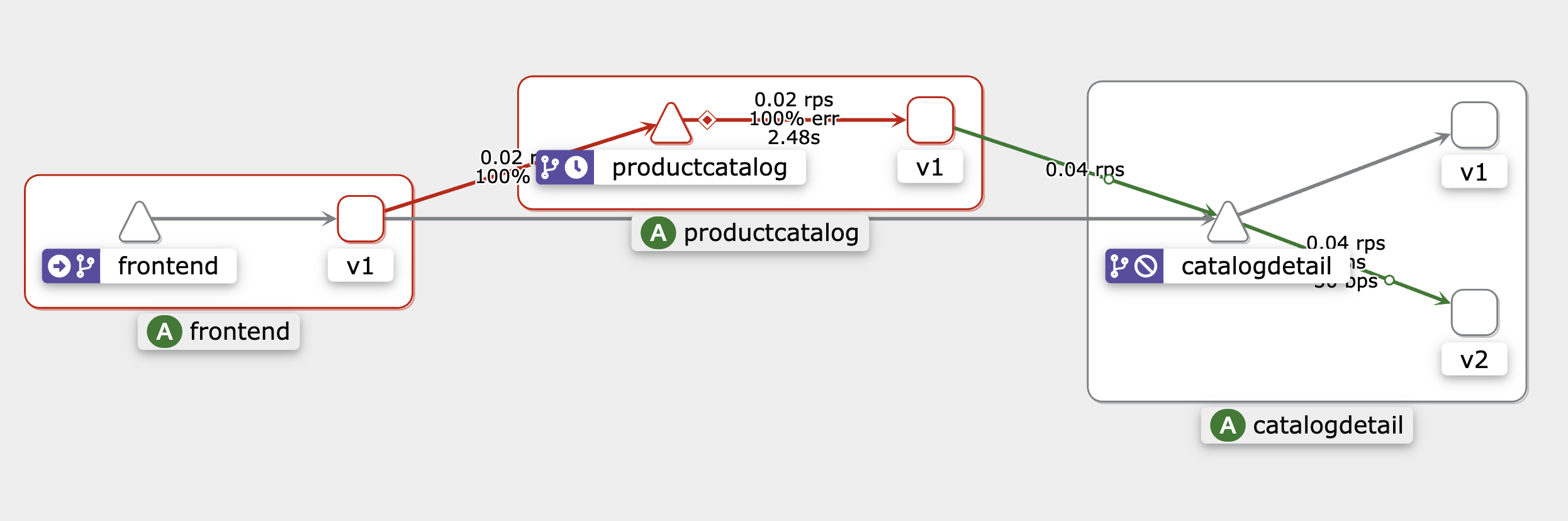
root@frontend-container-id:/app#

# Allows accessing the shell inside the frontend container for executing commands

curl http://productcatalog:5000/products/ -s -o /dev/null -w "Time taken to start transfer: %{time\_starttransfer}\n"

The output should be similar to this:

Time taken to start transfer: 2.006172



In conclusion, Istio’s timeouts feature enhances service resilience by managing the timing of requests between services in a microservices architecture. It prevents resource exhaustion by ensuring that resources are not held indefinitely while waiting for responses from downstream services. Timeout settings help isolate failures by quickly detecting and handling unresponsive downstream services, improving system health and availability.

**Reset the environment**

Reset the configuration to the initial state by running the following command from the timeouts-retries-circuitbreaking directory:

kubectl apply -f ../../00-setup-mesh-resources/

**Retries**

Retries in Istio involve the automatic re-attempt of a failed request to improve the resilience of the system. A retry setting specifies the maximum number of times an Envoy proxy attempts to connect to a service if the initial call fails. This helps handle transient failures, such as network glitches or temporary service unavailability.

To test the retries functionality we will make the following changes to the productcatalog VirtualService:

1. Add configuration for retries with 2 attempts to the productcatalog VirtualService.

...

spec:

...

http:

...

retries:

attempts: 2

..

2. Edit the productcatalog deployment to run a container that does nothing other than to sleep for 1 hour. To achieve this we make the following changes to the deployment:

* + Change the readinessProbe to run a simple command echo hello. Since the command always succeeds, the container would immediately be ready.
  + Change the livenessProbe to run a simple command echo hello. Since the command always succeeds, the container is immediately marked to be live.
  + Add a command to the container that will cause the main process to sleep for 1 hour.

To apply these changes, run the following command:

This assumes that you are currently in “istio-on-eks/modules/03-network-resiliency/timeouts-retries-circuitbreaking” folder

# This assumes that you are currently in "istio-on-eks/modules/03-network-resiliency/timeouts-retries-circuitbreaking" folder

kubectl apply -f ./retries/

kubectl get deployment -n workshop productcatalog -o json |

jq '.spec.template.spec.containers[0].readinessProbe={exec:{command:["sh","-c","echo hello"]}}

| .spec.template.spec.containers[0].livenessProbe={exec:{command:["sh","-c","echo hello"]}}

| .spec.template.spec.containers[0]+={command:["sh","-c","sleep 1h"]}' |

kubectl apply --force=true -f -

Enable debug mode for the envoy logs of the productcatalog service with the following command:

*Note: If you get an error, then open another new terminal to execute the following command. Make sure istioctl is in the path and you are in this folder “istio-on-eks/modules/03-network-resiliency/timeouts-retries-circuitbreaking”*

istioctl pc log --level debug -n workshop deploy/productcatalog

To see the retries functionality from the logs, execute the following command:

kubectl -n workshop logs -l app=productcatalog -c istio-proxy -f |

grep "x-envoy-attempt-count"

Open a new terminal and run the curl command against the productcatalog service from within the mesh by using the command here:

export FE\_POD\_NAME=$(kubectl get pods -n workshop -l app=frontend -o jsonpath='{.items[].metadata.name}')

kubectl exec -it ${FE\_POD\_NAME} -n workshop -c frontend -- bash

curl http://productcatalog:5000/products/ -s -o /dev/null

Now, you should see the retry attempts in the logs on your first terminal:

'x-envoy-attempt-count', '1'

'x-envoy-attempt-count', '1'

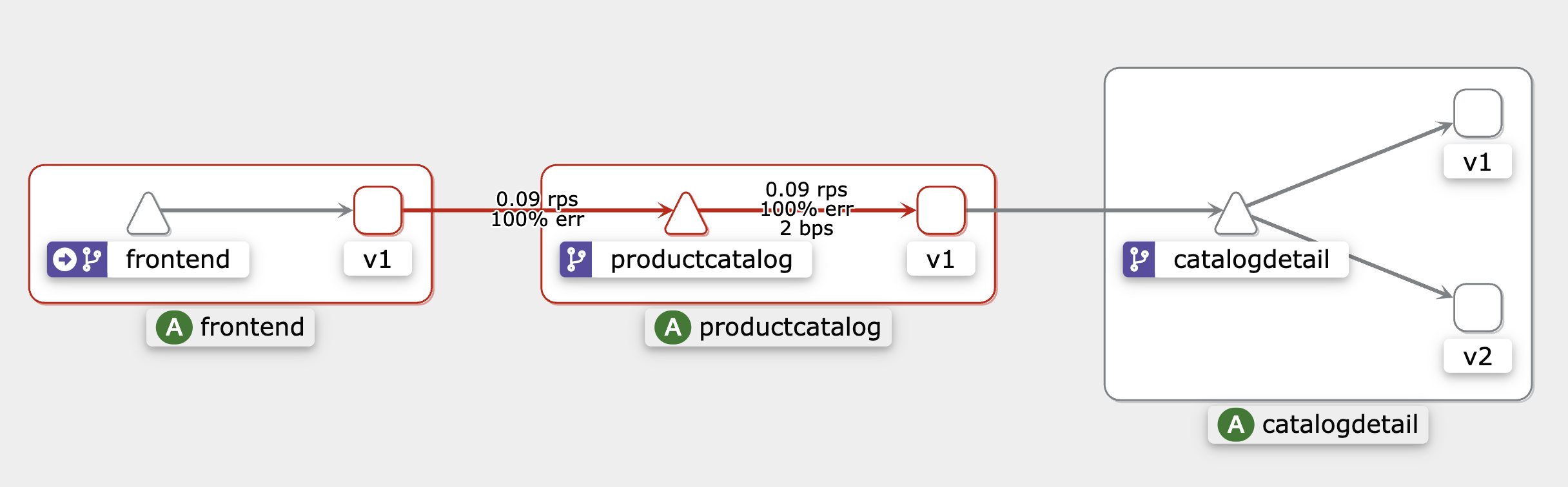
'x-envoy-attempt-count', '2'

'x-envoy-attempt-count', '2'

'x-envoy-attempt-count', '3'

'x-envoy-attempt-count', '3'

This diagram shows the Frontend envoy proxy trying to connect to the productcatalog VirtualService with 2 retry attempts after the initial call fails. If the request is unsuccessful after the retry attempts, this will be treated as an error and and it’s sent back to the Frontend service.



To conclude, the retry functionality operates as intended. The recorded ‘x-envoy-attempt-count’ of three includes the initial connection attempt to the service, followed by the two additional retry attempts, as defined in the configuration added to the productcatalog VirtualService.

**Reset the environment**

Reset the configuration to the initial state by running the following command from the timeouts-retries-circuitbreaking directory.

kubectl apply -f ../../00-setup-mesh-resources/

**Circuit Breakers**

Circuit breakers are another resilience feature provided by Istio’s Envoy proxies. Circuit breaker is a mechanism that boosts the resilience of a microservices based system by preventing continuous retries to a failing service. It helps avoid overwhelming a struggling service and enables the system to gracefully degrade when issues arise. When a predefined threshold of failures is reached, the circuit breaker activates and temporarily halts the request to the failing service to recover. Once a specified timeout elapses or the failure rate decreases then the circuit breaker resets, allowing the normal requests flow to resume.

Update the existing catalogdetail destination rule to apply circuit breaking configuration.

...

spec:

...

trafficPolicy:

connectionPool:

tcp:

maxConnections: 1

http:

http1MaxPendingRequests: 1

maxRequestsPerConnection: 1

outlierDetection:

consecutive5xxErrors: 1

interval: 1s

baseEjectionTime: 3m

maxEjectionPercent: 100

# This assumes that you are currently in the "istio-on-eks/modules/03-network-resiliency/timeouts-retries-circuitbreaking" folder

kubectl apply -f ./circuitbreaking/

As part of the [connectionPool settings](https://istio.io/latest/docs/reference/config/networking/destination-rule/" \l "ConnectionPoolSettings), the service has a maximum limit on the number of connections, and Istio will queue any surplus connections beyond this limit. You can adjust this limit by modifying the value in the maxConnections field, which is set to 1 in the above configuration. Additionally, there is a maximum for pending requests to the service, and any exceeding pending requests will be declined. You have the flexibility to modify this limit by adjusting the value in the http1MaxPendingRequests field, which is set to 1 in the above configuration.

As part of the [outlierDetection settings](https://istio.io/latest/docs/reference/config/networking/destination-rule/" \l "OutlierDetection), Istio will detect any host that triggers a server error (5XX code) in the catalogdetail Envoy and eject the pod out of the load balancing pool for 3 minutes.

To test the circuit breaker feature we will use a load testing application called fortio. To do this we will run a pod with a single container based on the fortio image. Run the command here to create a fortio pod in the workshop namespace:

kubectl run fortio --image=fortio/fortio:latest\_release -n workshop --annotations='proxy.istio.io/config=proxyStatsMatcher:

inclusionPrefixes:

- "cluster.outbound"

- "cluster\_manager"

- "listener\_manager"

- "server"

- "cluster.xds-grpc"'

Now from within the fortio pod test out a single curl request to the catalogdetail service:

kubectl exec fortio -n workshop -c fortio -- /usr/bin/fortio \

curl http://catalogdetail.workshop.svc.cluster.local:3000/catalogDetail

You can see the request succeed, as shown here:

{"ts":1704746733.172561,"level":"info","r":1,"file":"scli.go","line":123,"msg":"Starting","command":"Φορτίο","version":"1.60.3 h1:adR0uf/69M5xxKaMLAautVf9FIVkEpMwuEWyMaaSnI0= go1.20.10 amd64 linux"}

HTTP/1.1 200 OK

....

x-envoy-upstream-service-time: 8

server: envoy

{"version":"1","vendors":["ABC.com"]}%

**Tripping the circuit breaker**

We can start testing the circuit breaking functionality by generating traffic to the catalogdetail service with two concurrent connections (-c 2) and by sending a total of 20 requests (-n 20):

kubectl exec fortio -n workshop -c fortio -- \

/usr/bin/fortio load -c 2 -qps 0 -n 20 -loglevel Warning \

http://catalogdetail.workshop.svc.cluster.local:3000/catalogDetail

The output should be similar to this:

...

[0] 3 socket used, resolved to 172.20.233.48:3000, connection timing : count 3 avg 8.8641667e-05 +/- 1.719e-05 min 6.4519e-05 max 0.000103303 sum 0.000265925

[1] 3 socket used, resolved to 172.20.233.48:3000, connection timing : count 3 avg 0.000145408 +/- 7.262e-05 min 8.4711e-05 max 0.000247501 sum 0.000436224

Connection time (s) : count 6 avg 0.00011702483 +/- 5.992e-05 min 6.4519e-05 max 0.000247501 sum 0.000702149

Sockets used: 6 (for perfect keepalive, would be 2)

Uniform: false, Jitter: false, Catchup allowed: true

IP addresses distribution:

172.20.233.48:3000: 6

Code 200 : 15 (75.0 %)

Code 503 : 5 (25.0 %)

Response Header Sizes : count 20 avg 177.75 +/- 102.6 min 0 max 237 sum 3555

Response Body/Total Sizes : count 20 avg 268.9 +/- 16.57 min 241 max 283 sum 5378

All done 20 calls (plus 0 warmup) 6.391 ms avg, 306.0 qps

We notice that the majority of requests have been successful, with only a few exceptions. The istio-proxy does allow for some leeway.

Code 200 : 17 (85.0 %)

Code 503 : 3 (15.0 %)

Now re-run the same command by increasing the number of concurrent connections to 3 and number of calls to 30

kubectl exec fortio -n workshop -c fortio -- \

/usr/bin/fortio load -c 3 -qps 0 -n 30 -loglevel Warning \

http://catalogdetail.workshop.svc.cluster.local:3000/catalogDetail

The output should be similar to this:

...

[0] 5 socket used, resolved to 172.20.233.48:3000, connection timing : count 5 avg 0.0001612534 +/- 8.84e-05 min 8.515e-05 max 0.000330062 sum 0.000806267

[1] 5 socket used, resolved to 172.20.233.48:3000, connection timing : count 5 avg 0.000214197 +/- 0.0002477 min 5.6614e-05 max 0.00070467 sum 0.001070985

[2] 9 socket used, resolved to 172.20.233.48:3000, connection timing : count 9 avg 0.00011278589 +/- 5.424e-05 min 6.4739e-05 max 0.000243749 sum 0.001015073

Connection time (s) : count 19 avg 0.00015222763 +/- 0.0001462 min 5.6614e-05 max 0.00070467 sum 0.002892325

Sockets used: 19 (for perfect keepalive, would be 3)

Uniform: false, Jitter: false, Catchup allowed: true

IP addresses distribution:

172.20.233.48:3000: 19

Code 200 : 12 (40.0 %)

Code 503 : 18 (60.0 %)

Response Header Sizes : count 30 avg 94.8 +/- 116.1 min 0 max 237 sum 2844

Response Body/Total Sizes : count 30 avg 256 +/- 18.59 min 241 max 283 sum 7680

All done 30 calls (plus 0 warmup) 3.914 ms avg, 596.6 qps

As we increase the traffic towards the catalogdetail microservice we start to notice the circuit breaking functionality kicking in. We now notice that only 40% of the requests succeeded and the other 60%, as expected, were trapped by the circuit breaker.

Code 200 : 12 (40.0 %)

Code 503 : 18 (60.0 %)

Now, query the istio-proxy to see the statistics of the requests flagged for circuit breaking.

kubectl exec fortio -n workshop -c istio-proxy -- pilot-agent request GET stats | grep catalogdetail | grep pending

The output should be similar to this:

...

cluster.outbound|3000|v2|catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_active: 0

cluster.outbound|3000|v2|catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_failure\_eject: 0

cluster.outbound|3000|v2|catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_overflow: 0

cluster.outbound|3000|v2|catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_total: 0

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.circuit\_breakers.default.remaining\_pending: 1

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.circuit\_breakers.default.rq\_pending\_open: 0

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.circuit\_breakers.high.rq\_pending\_open: 0

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_active: 0

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_failure\_eject: 0

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_overflow: 17

cluster.outbound|3000||catalogdetail.workshop.svc.cluster.local.upstream\_rq\_pending\_total: 34

In summary, the output above reveals that the upstream\_rq\_pending\_overflow parameter holds a value of 17. This indicates that 17 calls so far have been flagged for circuit breaking, providing clear evidence that our circuit breaker configuration for the catalogdetail DestinationRule has effectively functioned as intended.

**Reset the environment**

Delete the fortio pod using the following command and then run the same steps as in the Initial state setup to reset the environment.

kubectl delete pod fortio -n workshop

kubectl apply -f ../../00-setup-mesh-resources/

**Rate Limit**

[Local Rate Limiting](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/local_rate_limit_filter) in Istio allows you to control the rate of traffic for individual services or service versions within your cluster. This allows you to control the rate of requests for specific services or endpoints. Global Rate Limiting in Istio allows you to enforce rate limits across the entire mesh and/or for a specific gateway.

[Global rate limiting](https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/global_rate_limiting) uses a global gRPC rate limiting service, shared by all the services in the cluster, to enforce the rate limiting for the entire mesh. The rate-limiting service requires an external component, typically a Redis database. Local rate limiting can be used in conjunction with global rate limiting to reduce load on the global rate limiting service.

*NOTE: This sub-module expects you to be in the*[*rate-limit*](https://github.com/aws-samples/istio-on-eks/tree/network-resiliency/modules/03-network-resiliency/rate-limiting)*sub module under the*[*03-network-resiliency*](https://github.com/aws-samples/istio-on-eks/tree/network-resiliency/modules/03-network-resiliency)*module of the git repo.*

# This assumes that you are currently in "istio-on-eks/modules/03-network-resiliency" folder

cd ../rate-limiting

**Local Rate Limiting**

Applying the local rate limit is done by applying [EnvoyFilter](https://istio.io/latest/docs/reference/config/networking/envoy-filter/) to an individual service within the application. In our example, we are going to be applying the limit to the [prodcatalog](http://modules/00-setup-mesh-resources/productcatalog-virtualservice.yaml) service.

Looking into the contents of the [local-ratelimit.yaml](https://github.com/aws-samples/istio-on-eks/blob/network-resiliency/modules/03-network-resiliency/rate-limiting/local-ratelimit/local-ratelimit.yaml) file:

1. The HTTP\_FILTER patch inserts the envoy.filters.http.local\_ratelimit [local envoy filter](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/local_rate_limit_filter#config-http-filters-local-rate-limit) into the HTTP connection manager filter chain.
2. The local rate limit filter’s [token bucket](https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/http/local_ratelimit/v3/local_rate_limit.proto#envoy-v3-api-field-extensions-filters-http-local-ratelimit-v3-localratelimit-token-bucket) is configured to allow 10 requests/min.
3. The filter is also configured to add an x-local-rate-limit response header to requests that are blocked.

Apply Local Rate Limiting to the prodcatalog service:

kubectl apply -f ./local-ratelimit/local-ratelimit.yaml

To test the rate limiter in action, exec into the frontend pod and send requests to the prodcatalog service to trigger the rate limiter.

POD\_NAME=$(kubectl get pod -l app=frontend -o jsonpath='{.items[0].metadata.name}' -n workshop)

kubectl exec $POD\_NAME -n workshop -c frontend -- \

bash -c "for i in {1..20}; do curl -sI http://productcatalog:5000/products/; done"

Since the 20 requests are sent in less than a minute, after the first 10 requests are accepted by the service you’ll start seeing HTTP 429 response codes from the service.

Successful requests will return the following output:

HTTP/1.1 200 OK

content-type: application/json

content-length: 124

x-amzn-trace-id: Root=1-6502273f-8dd970ab66ed073ccd2519c7

access-control-allow-origin: \*

server: envoy

date: Wed, 13 Sep 2023 21:18:55 GMT

x-envoy-upstream-service-time: 15

x-ratelimit-limit: 10

x-ratelimit-remaining: 9

x-ratelimit-reset: 45

While rate limited requests will return the following output:

HTTP/1.1 429 Too Many Requests

x-local-rate-limit: true

content-length: 18

content-type: text/plain

x-ratelimit-limit: 10

x-ratelimit-remaining: 0

x-ratelimit-reset: 45

date: Wed, 13 Sep 2023 21:18:55 GMT

server: envoy

x-envoy-upstream-service-time: 0

Similarly, if you run the same shell command without the -I flag, you’ll start seeing local\_rate\_limited responses for the requests that are rate limited. These rate limited requests will look something like this:

{

"products": {},

"details": {

"version": "2", <---------- Successful response to the request

"vendors": [

"ABC.com, XYZ.com"

]

}

}

local\_rate\_limited <---------- Rate limited requests

**Global Rate Limiting**

Applying Global Rate limiting to the Istio Service Mesh can be done by performing the following steps:

Step 1 – Setting up Global Rate Limit Service

To use the Global Rate Limit in our Istio service mesh we need a central global rate limit service that implements Envoy’s rate limit service protocol. This can be achieved by:

* Applying the Global Rate Limiting configuration for the Global Rate Limit service via the [global-ratelimit-config.yaml](https://github.com/aws-samples/istio-on-eks/blob/network-resiliency/modules/03-network-resiliency/rate-limiting/global-ratelimit/global-ratelimit-config.yaml) ConfigMap:

kubectl apply -f ./global-ratelimit/global-ratelimit-config.yaml

In this file, rate limit requests to the / path are set to 5 requests/minute and all other requests are set to 100 requests/minute.

* Deploying the Global Rate Limit service with Redis via the [global-ratelimit-service.yaml](https://github.com/aws-samples/istio-on-eks/blob/network-resiliency/modules/03-network-resiliency/rate-limiting/global-ratelimit/global-ratelimit-service.yaml) file.

kubectl apply -f ./global-ratelimit/global-ratelimit-service.yaml

This file has Deployment and Service definitions for:

* Central Rate Limit Service; and
* Redis

Step 2 – Enable the Global Rate Limit

Once the central rate limit service and redis is configured and deployed, Global Rate Limiting can be enabled and configured by applying two [EnvoyFilters](https://istio.io/latest/docs/reference/config/networking/envoy-filter/) to the [IngressGateway](https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/):

* The first EnvoyFilter defined in the [filter-ratelimit.yaml](https://github.com/aws-samples/istio-on-eks/blob/network-resiliency/modules/03-network-resiliency/rate-limiting/global-ratelimit/filter-ratelimit.yaml) file enables global rate limiting using Envoy’s global rate limit filter.

kubectl apply -f ./global-ratelimit/filter-ratelimit.yaml

* The second EnvoyFilter defines the route configuration on which rate limiting will be applied. Looking at the [filter-ratelimit-svc.yaml](https://github.com/aws-samples/istio-on-eks/blob/network-resiliency/modules/03-network-resiliency/rate-limiting/global-ratelimit/filter-ratelimit-svc.yaml) file, the configuration adds rate limit actions for any route from a virtual host.

kubectl apply -f ./global-ratelimit/filter-ratelimit-svc.yaml

To test the global rate limit in action, run the following command in a terminal session:

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

for i in {1..6}; do curl -Is $ISTIO\_INGRESS\_URL; done

In the output you should notice that the first five requests will generate output similar to the one here:

HTTP/1.1 200 OK

x-powered-by: Express

content-type: text/html; charset=utf-8

content-length: 1203

etag: W/"4b3-KO/ZeBhhZHNNKPbDwPiV/CU2EDU"

date: Wed, 17 Jan 2024 16:53:23 GMT

x-envoy-upstream-service-time: 34

server: istio-envoy

And the last request should generate output similar to:

HTTP/1.1 429 Too Many Requests

x-envoy-ratelimited: true

date: Wed, 17 Jan 2024 16:53:35 GMT

server: istio-envoy

transfer-encoding: chunked

We see this behavior because of the global rate limiting that is in effect. It is only allowing a maximum of 5 requests/minute when the context-path is /.

**Reset the environment**

Execute the following command to remove all rate-limiting configurations and services:

Delete all rate limiting configurations and services

# Delete all rate limiting configurations and services

kubectl delete -f ./local-ratelimit

kubectl delete -f ./global-ratelimit

**Clean up**

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

kubectl delete namespace workshop

*NOTE: 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 blog post, we explored how Istio on Amazon EKS can enhance network resilience for microservices. By providing critical features like timeouts, retries, circuit breakers, rate limiting, and fault injection, Istio enables microservices to maintain responsive communication even when facing disruptions. As a result, Istio helps prevent localized failures from cascading and bringing down entire applications. Overall, Istio gives developers powerful tools to build reliability and robustness into microservices architectures. The service mesh creates a layer of infrastructure that adds resilience without changing application code.

As we conclude this post on network resilience, stay tuned for our next blog where we will dive deep into security with Istio in Amazon EKS. We have much more to explore when it comes to hardening and protecting microservices. Join us on this ongoing journey as we leverage Istio and Amazon EKS to create robust cloud native applications.