Getting Started with Istio on Amazon EKS

by Praseeda Sathaye and Vijay Chintalapati | on 05 DEC 2023 | 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/getting-started-with-istio-on-amazon-eks/) | [Comments](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/#Comments) | [Share](https://aws.amazon.com/blogs/opensource/getting-started-with-istio-on-amazon-eks/)

In the dynamic landscape of modern architecture, making microservices work seamlessly in the cloud can be a puzzle. The transition to microservices often brings complexities related to traffic management, security, and observability. This is where [Istio](https://istio.io/latest/) steps in, offering a comprehensive service mesh solution that streamlines these challenges. Istio is an open source service mesh platform designed to enhance the management, security, policy enforcement, and observability of microservices-based applications. It’s typically used in complex, distributed architectures where multiple microservices need to communicate with each other. Istio augments Kubernetes by introducing a versatile, application-aware network fabric powered by the robust open source Envoy proxy as a “sidecar” configuration. Deploying Istio in [Amazon Elastic Kubernetes Service](https://aws.amazon.com/eks/) (Amazon EKS) takes this advantage a step further. It seamlessly enhances the capabilities of Amazon EKS by providing advanced traffic management, security with mutual TLS encryption, and a uniform approach to observability through other open source tools like Prometheus and Grafana.

Lets embark on an exciting journey into the world of Istio with our blog series! We’ll dive into the multiple ways Istio empowers a modern architecture when integrated with Amazon EKS, making it the go-to choice for organizations striving for excellence in their cloud native journey. In this first blog post, we’ll walk you through the process of getting Istio up and running on Amazon EKS. We’ll cover everything from deploying microservices, to testing the communication between these services. Plus, we’ll show you how to visualize all that traffic in action by enabling Istio’s telemetry add-ons for enhanced observability.

We’re thrilled to announce that we’ve recently updated the [Istio Platform Setup documentation](https://istio.io/latest/docs/setup/platform-setup/) to include Amazon EKS. This setup uses Terraform-based [EKS Blueprints for Istio](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/) and we will be using this blueprint in our blog.

**Istio Architecture**

Service mesh architecture tackles the complexities developers and operators encounter in distributed or microservices-based systems. [Istio’s architecture](https://istio.io/latest/docs/ops/deployment/architecture/) consists of two essential components: the data plane and the control plane.

The data plane manages communication between services, enhancing the network’s understanding of traffic by intercepting it. Istio employs a proxy, an extended version of the [Envoy Proxy](https://www.envoyproxy.io/) which intercepts all network traffic, enabling a wide range of application-aware features based on configurable rules. Envoy proxies are deployed alongside each service within your Amazon EKS cluster, ensuring they can manage traffic effectively.

The control plane, on the other hand, takes your predefined configurations and applies its understanding of the services to dynamically program the envoy proxy. It continuously updates them to adapt to rule changes or evolving environments, ensuring seamless and secure service-to-service communication within your microservices architecture.

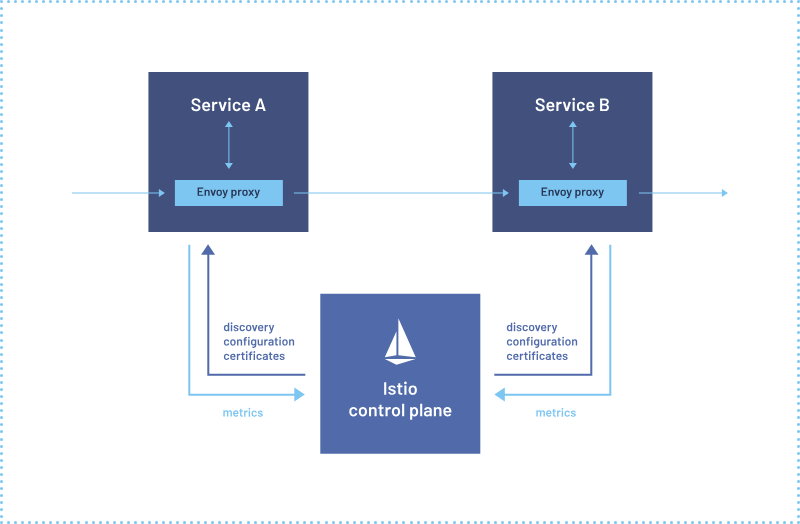


Image courtesy of: https://istio.io/latest/about/service-mesh/

**Getting Started with Istio on Amazon EKS**

In this blog post, we’ll guide you through a series of essential steps to supercharge your microservices architecture:

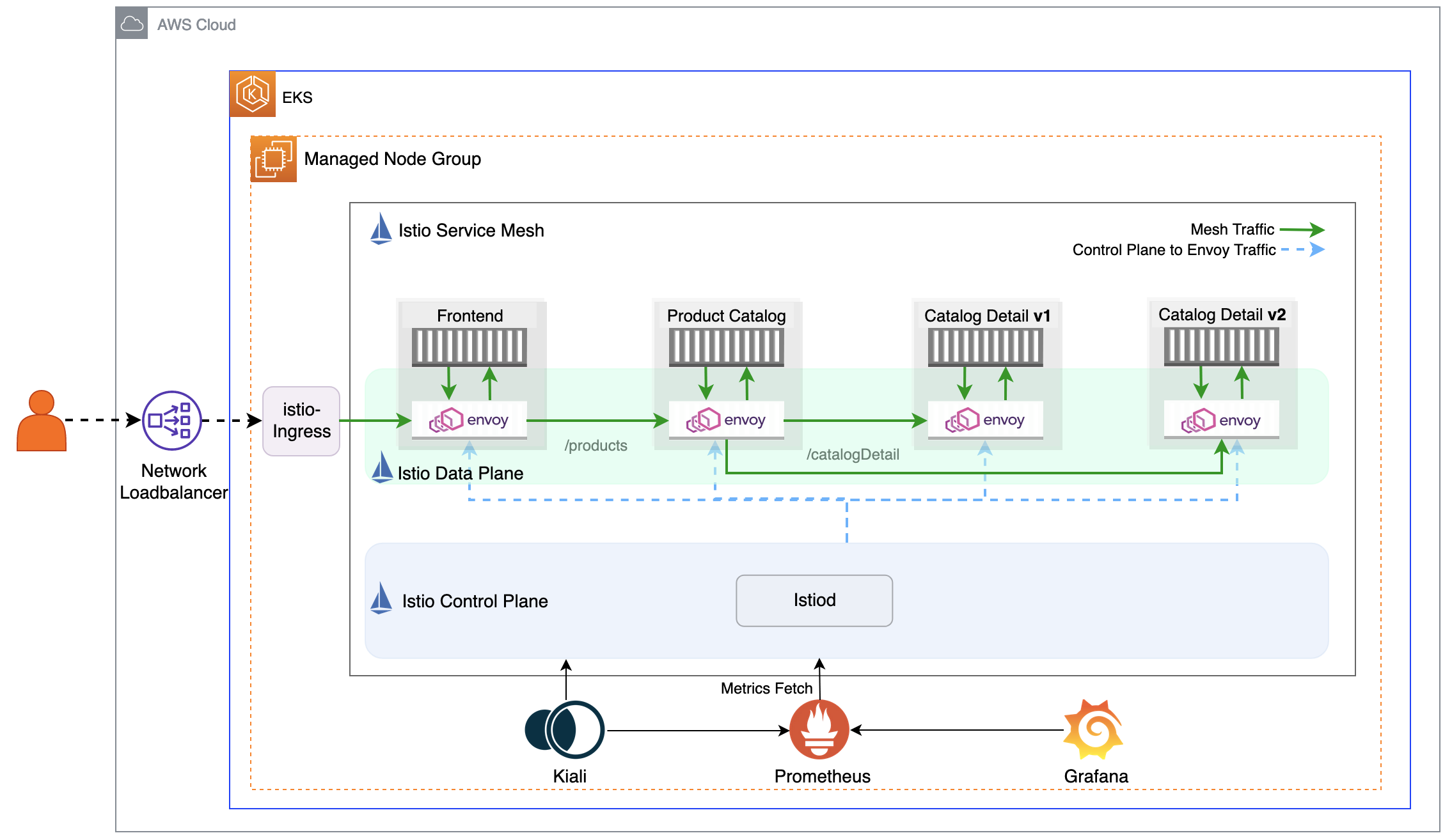
1. Deploying applications on Amazon EKS: We’ll kick things off by deploying the applications onto Amazon EKS, setting the stage for a resilient and scalable infrastructure.
2. Istio Integration: Next, we’ll seamlessly integrate these applications into Istio’s service mesh, unlocking a host of powerful capabilities for traffic management, security, and observability.
3. Edge Traffic Management: Then, we will also dive into the intricacies of handling ingress traffic at the edge of your service mesh using Istio Ingress Gateway along with AWS Network Load Balancer, ensuring secure and controlled access to your services.
4. Routing with VirtualService: We’ll show you how to finely control ingress HTTP traffic, matching it to the appropriate service within the mesh using Istio’s VirtualService[Custom Resource Definition](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#adding-custom-resources) (CRD).
5. Visualizing the Magic: Finally we will harness the power of visualization with tools like [Kiali](https://istio.io/latest/docs/ops/integrations/kiali/), [Prometheus](https://istio.io/latest/docs/ops/integrations/prometheus/), and [Grafana](https://istio.io/latest/docs/ops/integrations/grafana/).

**Deployment Architecture**

Throughout this blog, as well as for the upcoming Istio on Amazon EKS blogs, we’ll be leveraging a microservices-based Product Catalog Application as our real-world example. This application will serve as our practical playground, allowing us to explore Istio’s capabilities in a hands-on manner. The application is composed of three 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).

When a user accesses the application using its URL, the traffic first reaches the Istio Ingress Gateway via AWS Network Load Balancer and then reaches the frontend service (developed in NodeJS). The frontend service then calls the backend service productcatalog (developed in Python) to get the products and the productcatalog in turn calls catalogdetail (developed in NodeJS) backend service to get vendor information.

Kiali is used to gain insights into the versioned application graph, enabling us to visualize and comprehend the intricate relationships between these services. Istio’s metrics will be captured with Prometheus and presented using Grafana, empowering comprehensive observability into these microservices.



**EKS Cluster setup**

In this blog we will be using this [EKS Blueprints for Istio](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/) to spin up an Amazon EKS cluster along with the Istio setup and other observability add-ons. Using Terraform-based EKS Blueprints for Istio greatly simplifies the setup and management of Istio in your Kubernetes environment. This blueprint provides a structured and efficient approach to deploying Istio on Amazon EKS, eliminating many of the manual configurations and potential pitfalls. This blueprint will do the following:

* Deploy an Amazon EKS Cluster with one managed node group in a VPC
* Add node\_security\_group rules for port access required for Istio communication
* Install Istio using Helm resources in Terraform
* Install Istio Ingress Gateway using Helm resources in Terraform (This step deploys a Service of type LoadBalancer that creates an AWS Network Load Balancer).

**Clone the repository**

git clone https://github.com/aws-ia/terraform-aws-eks-blueprints.git

cd terraform-aws-eks-blueprints/patterns/istio

**Setup EKS and Istio**

Using the [prerequisites](https://aws-ia.github.io/terraform-aws-eks-blueprints/getting-started/#prerequisites) section, install the necessary tools for this setup. Run the following command to update information of available charts locally from chart repositories.

helm repo update

Execute the following sequence of commands to set up the infrastructure for EKS cluster with Istio. You can also refer to these steps in the [Deploy](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/#deploy) section of the [EKS Blueprint](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/#deploy).

terraform init

terraform apply -auto-approve

kubectl rollout restart deployment istio-ingress -n istio-ingress

Now lets add the Istio Observability Add-ons to the above EKS cluster. Replace the “XXX” in the below command with the Istio release version based on [Observability Add-ons](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/#observability-add-ons) section in the EKS Blueprint.

for ADDON in kiali jaeger prometheus grafana

do

ADDON\_URL="https://raw.githubusercontent.com/istio/istio/release-XXX/samples/addons/$ADDON.yaml"

kubectl apply -f $ADDON\_URL

done

Note: Proceed to the next step once the Amazon EKS cluster and Istio setup is completed.

**Deploying the microservices to Istio Service Mesh**

We will be following the steps from the [01-getting-started module](https://github.com/aws-samples/istio-on-eks/tree/main/modules/01-getting-started) of the istio-on-eks Git repository.

**Clone the repository**

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

cd istio-on-eks/modules/01-getting-started

**Setup Namespace for Istio Mesh**

To be able to deploy the microservices to the Istio Service Mesh automatically, the chosen namespace must be labeled with the label istio-injection=enabled. This will [inject the sidecar envoy proxy](https://istio.io/latest/docs/setup/additional-setup/sidecar-injection/#automatic-sidecar-injection) into the microservices that are part of the “workshop” namespace.

# Create workshop namespace and label it for use with Istio Service Mesh

kubectl create namespace workshop

kubectl label namespace workshop istio-injection=enabled

**Deploy Helm Chart**

Now deploy the provided [mesh-basic](https://github.com/aws-samples/istio-on-eks/blob/main/modules/01-getting-started/Chart.yaml) Helm Chart. This helm chart is packaged with a deployment manifest for:

* All the three microservices (frontend,prodcatalog, and catalogdetail)
* Istio Gateway and a VirtualService.

# Install all the microservices in one go

helm install mesh-basic . -n workshop

Output should be similar to:

namespace/workshop created

namespace/workshop labeled

NAME: mesh-basic

LAST DEPLOYED: Mon Aug 21 18:08:29 2023

......

......

Note: It may take a few minutes for the istio-ingress Network LoadBalancer to associate to the instance-mode targetGroup after the application is deployed.

**Validation**

Confirm the installation of microservices in the workshop namespace by running this command:

kubectl get pods -n workshop

Output should be similar to what’s shown here, indicating that each of the application pods is running two containers, an application container and an Istio envoy proxy as side-car container.

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

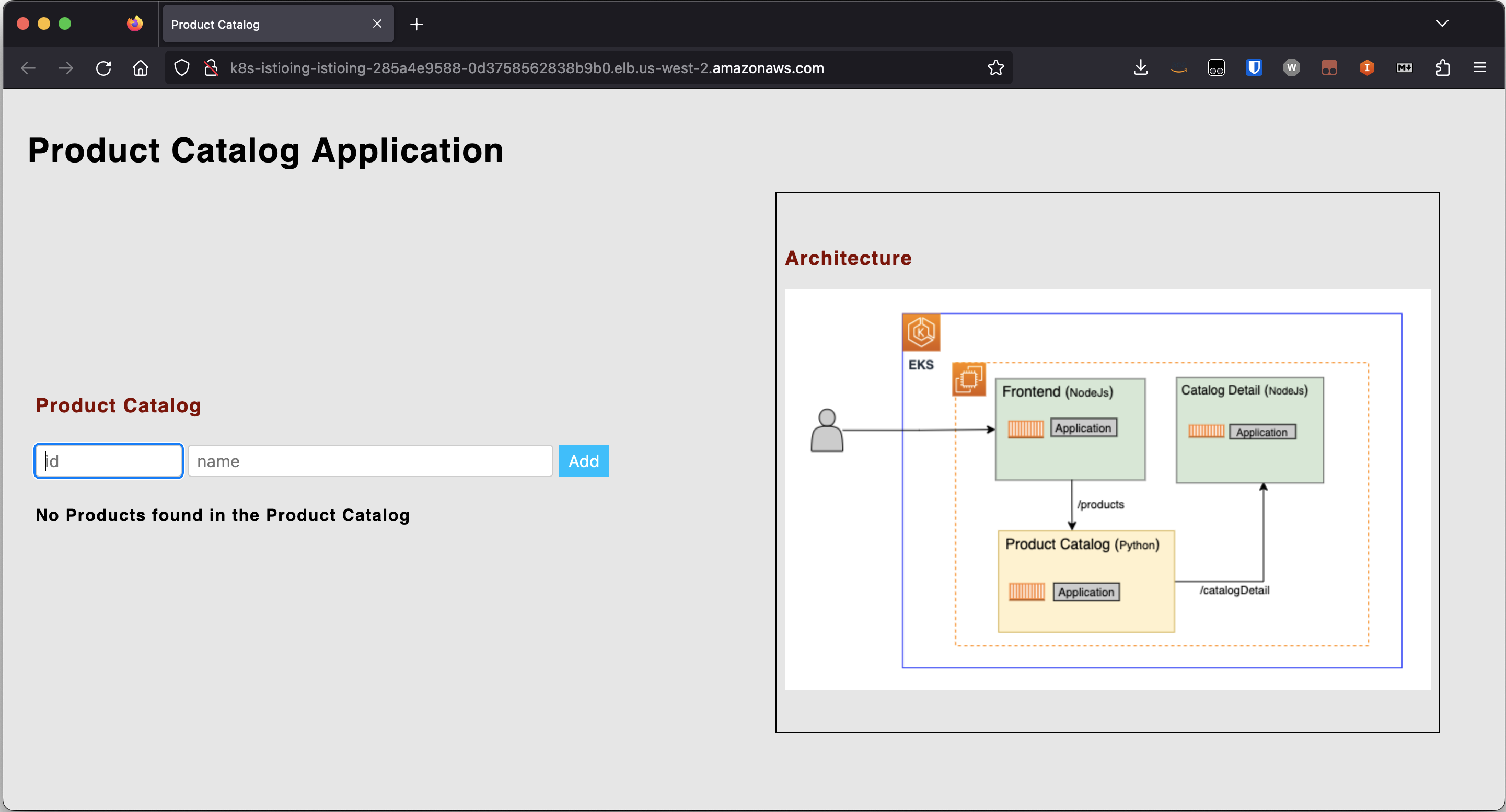
**Application User Interface**

The application’s (user interface) URL can be retrieved using the following command:

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

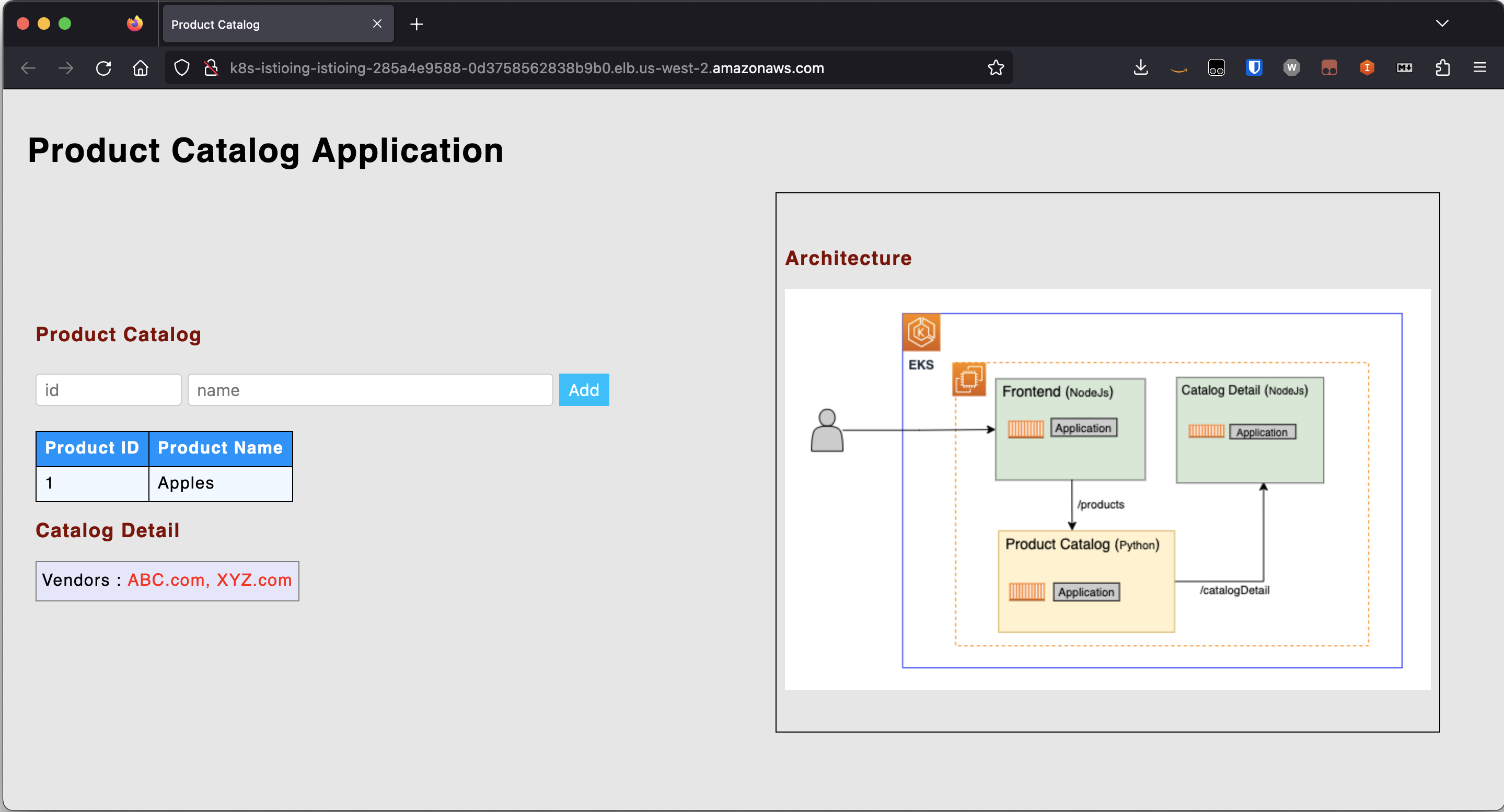
echo "http://$ISTIO\_INGRESS\_URL"

Accessing this URL in the browser will lead you to the Product Catalog application as shown here:

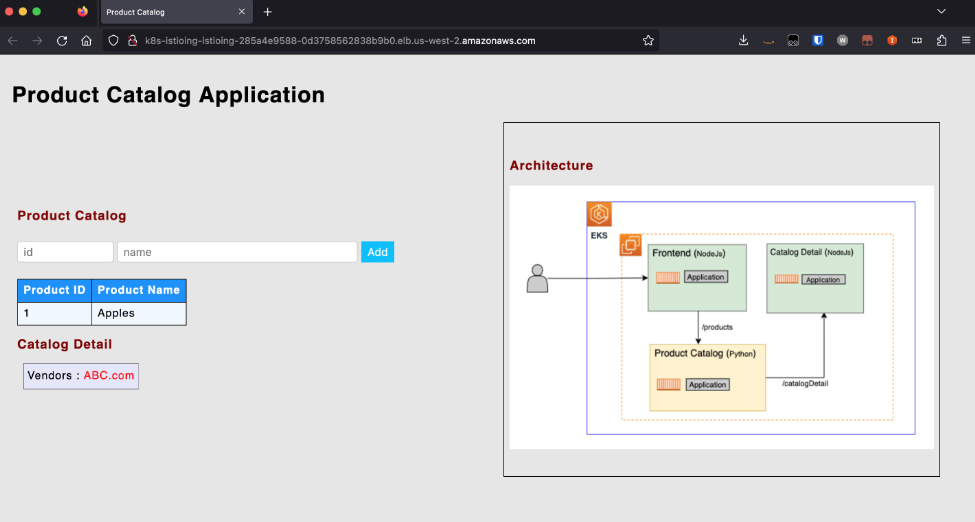


Using the input form, we can add sample products to the catalog as demonstrated in the following screenshot. When the catalog is refreshed after addition of a product Apples, we see that the catalog detail is retrieved from two vendors: ABC.com and XYZ.com

Note: The page output, under the Catalog Detail , might differ when adding products compared to the images shown here



Upon refreshing the page few times, you should see the Catalog Detail section toggling between Vendors: ABC.com and Vendors: ABC.com, XYZ.com. This is because the catalogdetail service has two versions v1 and v2 and this vendor information is coming from the respective versions. Depending on the [kube-proxy proxy modes](https://kubernetes.io/docs/reference/networking/virtual-ips/#proxy-modes), the call to this service can land on either version.



**Key Istio Components**

In this blog and the upcoming series, we will be gradually introducing all of the Istio core components. For this particular blog, we are focusing on two key Istio elements: Istio Ingress Gateway and VirtualService that are deployed via the Helm chart in the previous step.

A screenshot of a computer

Description automatically generated

Run the following command to list all the Istio resources created:

kubectl get Gateway,VirtualService -n workshop

Output should be similar to:

NAME AGE

gateway.networking.istio.io/productapp-gateway 7m50s

NAME GATEWAYS HOSTS AGE

virtualservice.networking.istio.io/productapp ["productapp-gateway"] ["\*"] 7m50s

**Istio Ingress Gateway**

[Istio Ingress Gateway](https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/) describes a network load balancer operating at the edge of the mesh receiving incoming HTTP/TCP connections. The specification describes a set of ports that should be exposed, the type of protocol to use, and configuration for the load balancer.

In our example, the productapp-gateway Gateway is responsible for defining which hostnames the ingress traffic allows through this gateway, its kind (protocol), and the port at which it is accepted.

Let’s look at the productapp-gateway definition in more detail, by running this kubectl command:

kubectl get gateway productapp-gateway -n workshop -o yaml

You should see output similar to what’s shown here:

apiVersion: networking.istio.io/v1beta1

kind: Gateway

metadata:

annotations:

meta.helm.sh/release-name: mesh-basic

meta.helm.sh/release-namespace: workshop

creationTimestamp: "2023-08-28T22:13:32Z"

generation: 1

labels:

app.kubernetes.io/managed-by: Helm

name: productapp-gateway

namespace: workshop

resourceVersion: "2436"

uid: 8a2e97eb-6879-4005-9864-2990db03b009

spec:

selector:

istio: ingressgateway

servers:

- hosts:

- '\*'

port:

name: http

number: 80

protocol: HTTP

YAML

Based on this YAML definition of the Gateway, we can conclude that the productapp-gateway Gateway:

* Is applied to the Envoy proxy of the pod with the label istio: ingressgateway
* Accepts traffic for all hosts (\*)
* Accepts traffic on port 80 for protocol HTTP

**VirtualService**

A [VirtualService](https://istio.io/latest/docs/concepts/traffic-management/#virtual-services) defines a set of traffic routing rules to apply when a hostname is addressed. Each routing rule defines matching criteria for traffic of a specific protocol. If the traffic is matched, then it is sent to a named destination service (or subset/version of it) defined in the registry. Without virtual services, Envoy distributes traffic using round-robin load balancing between all service instances mapped to the hostname. With a virtual service, you can specify routing rules that tell Envoy how to send the virtual service’s traffic to appropriate destinations.

In our example, the productapp VirtualService handles all the web traffic to the Istio Ingress endpoint by forwarding it to the frontend destination service.

kubectl get VirtualService productapp -n workshop -o yaml

You should see output similar to what’s shown here:

kind: VirtualService

metadata:

annotations:

meta.helm.sh/release-name: mesh-basic

meta.helm.sh/release-namespace: workshop

creationTimestamp: "2023-08-28T22:13:32Z"

generation: 1

labels:

app.kubernetes.io/managed-by: Helm

name: productapp

namespace: workshop

resourceVersion: "2440"

uid: 3ebc2a6f-0ef8-40b9-9fc3-dba326039577

spec:

gateways:

- productapp-gateway

hosts:

- '\*'

http:

- match:

- uri:

prefix: /

route:

- destination:

host: frontend

port:

number: 9000

YAML

Based on this YAML definition of the Gateway, we can conclude that the productapp VirtualService :

* Is associated specifically with productapp-gateway Gateway and any ingress traffic through it
* Matches any host name (\*) of the HTTP traffic
* When matched with no specific context path (/) in the request URI, routes the traffic to the frontend destination service.

**Visualization**

Now that we have demonstrated how to deploy services into Istio Service Mesh, let’s get into how you can visualize the service mesh with Kiali and its metrics in Grafana.

**Kiali**

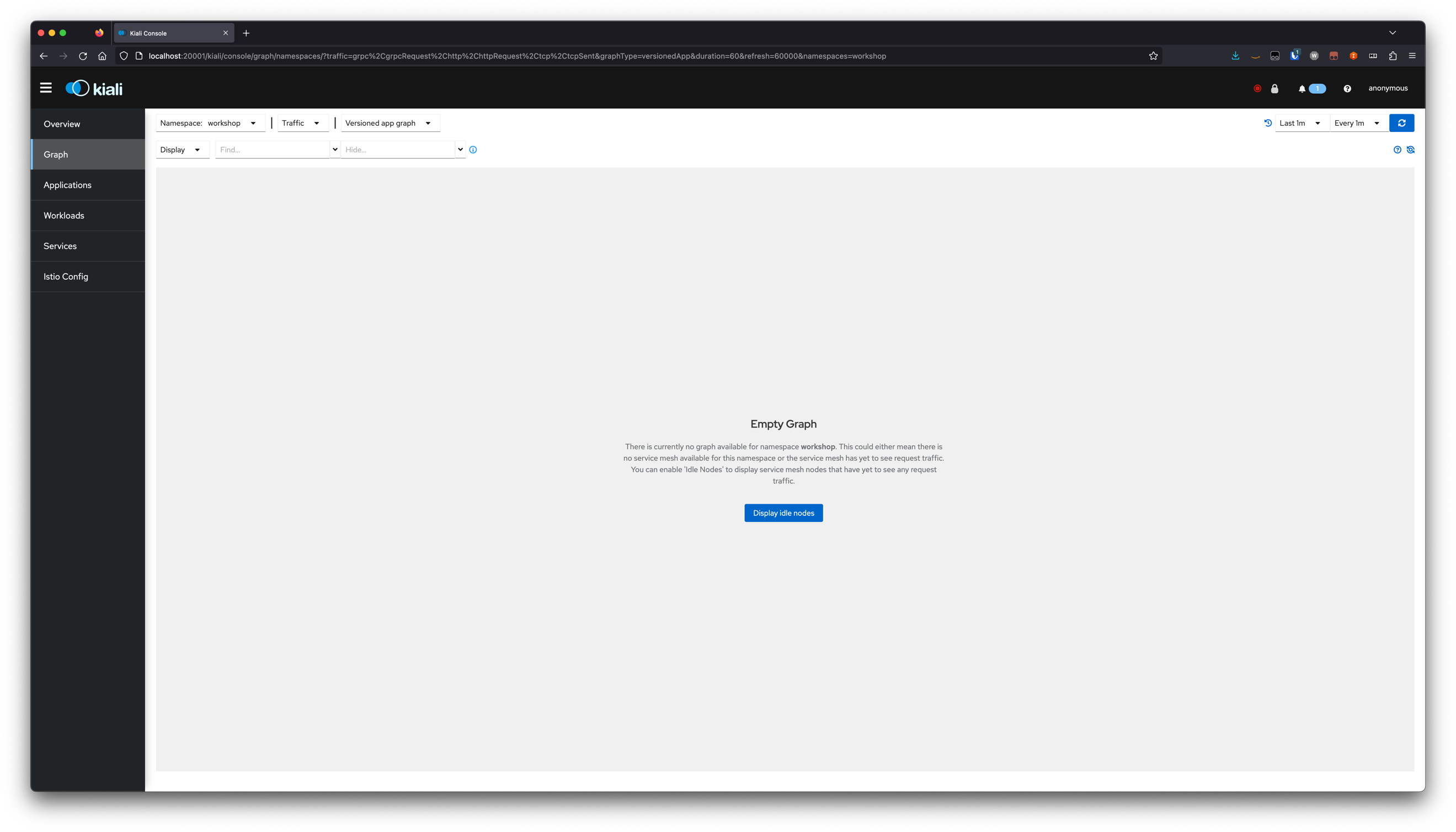
[Kiali](https://kiali.io/) is a console for Istio service mesh and we will be using Kiali to validate our setup. Kiali should already be available as a deployment in the istio-system namespace if you have setup Istio using the [EKS Istio blueprint](https://aws-ia.github.io/terraform-aws-eks-blueprints/patterns/istio/) we shared before.

Note: A complete overview of Kiali is beyond the scope of this blog post

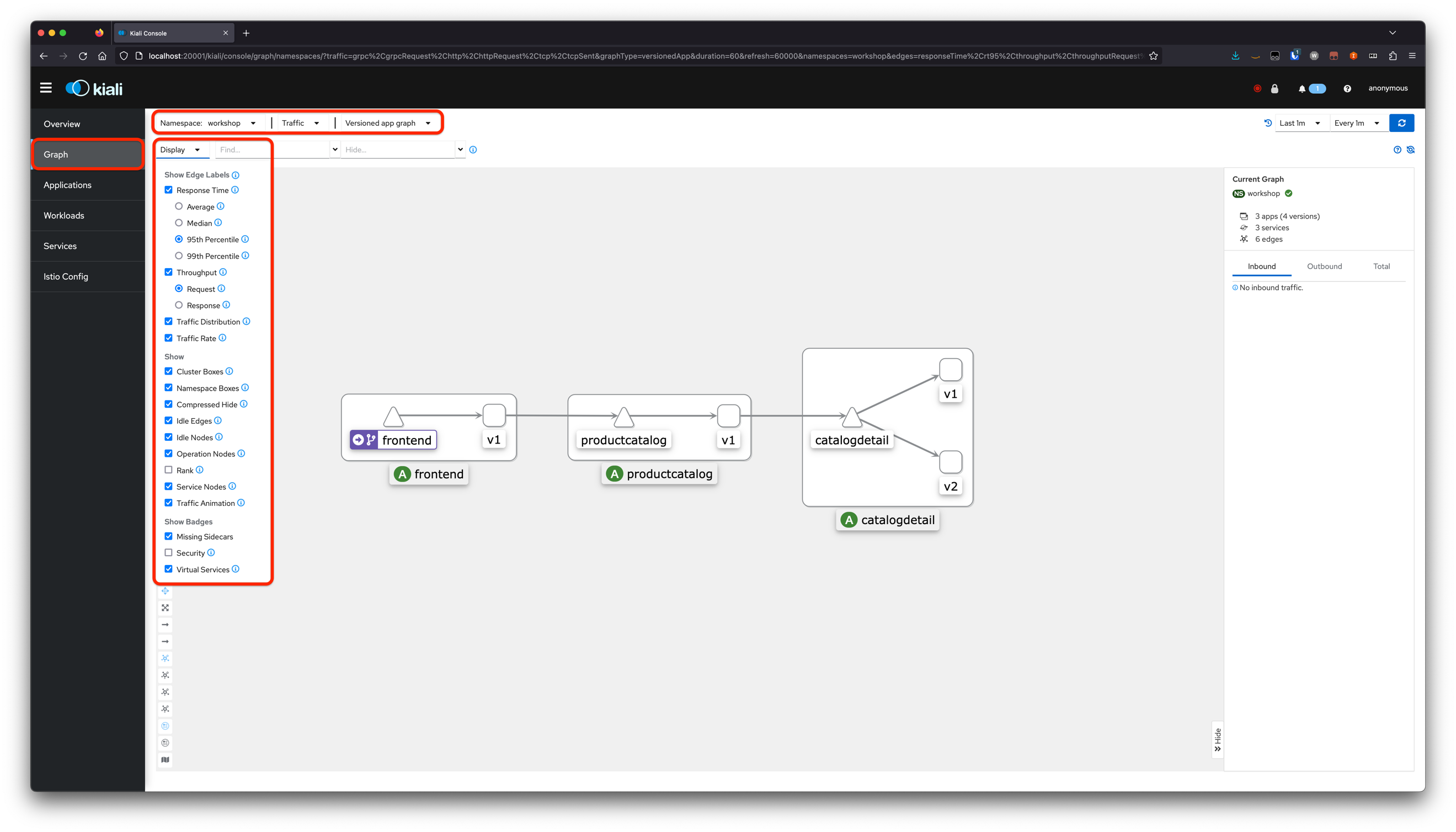
To be able to access the Kiali console, port-forward the Kiali service endpoint onto your local machine with the following command:

kubectl port-forward svc/kiali 20001:20001 -n istio-system

Use your browser to navigate to http://localhost:20001. In the Kiali console one can visualize all the microservices deployed in the Istio Mesh and how they interact with one other in the form of a directed graph. To see this in action, click Graph on the menu on the left and when presented with an Empty Graph as shown here, click on the Display Idle Nodes button.



When the view with the application node loads, customize the view further by observing the highlighted portions of the screenshot presented here and replicating the same in your environment. To see a similar graph to what we have displayed here in our screenshot, change the time on the right corner to “Last 10m.”



We note the following in this screenshot:

* The △ represents the Kubernetes service
* The ▢ represents the versioned application workloads. In our example, we have
  + v1 versioned workloads for frontend and productcatalog
  + v1 and v2 versioned workloads for catalogdetail
* The boxes encompassing a service and the versioned application workloads, collectively represent an Ⓐ application.
* The icon like the one we have shown here represents a VirtualService that points to an actual Kubernetes service such as frontend in this example.

frontend image

**Grafana**

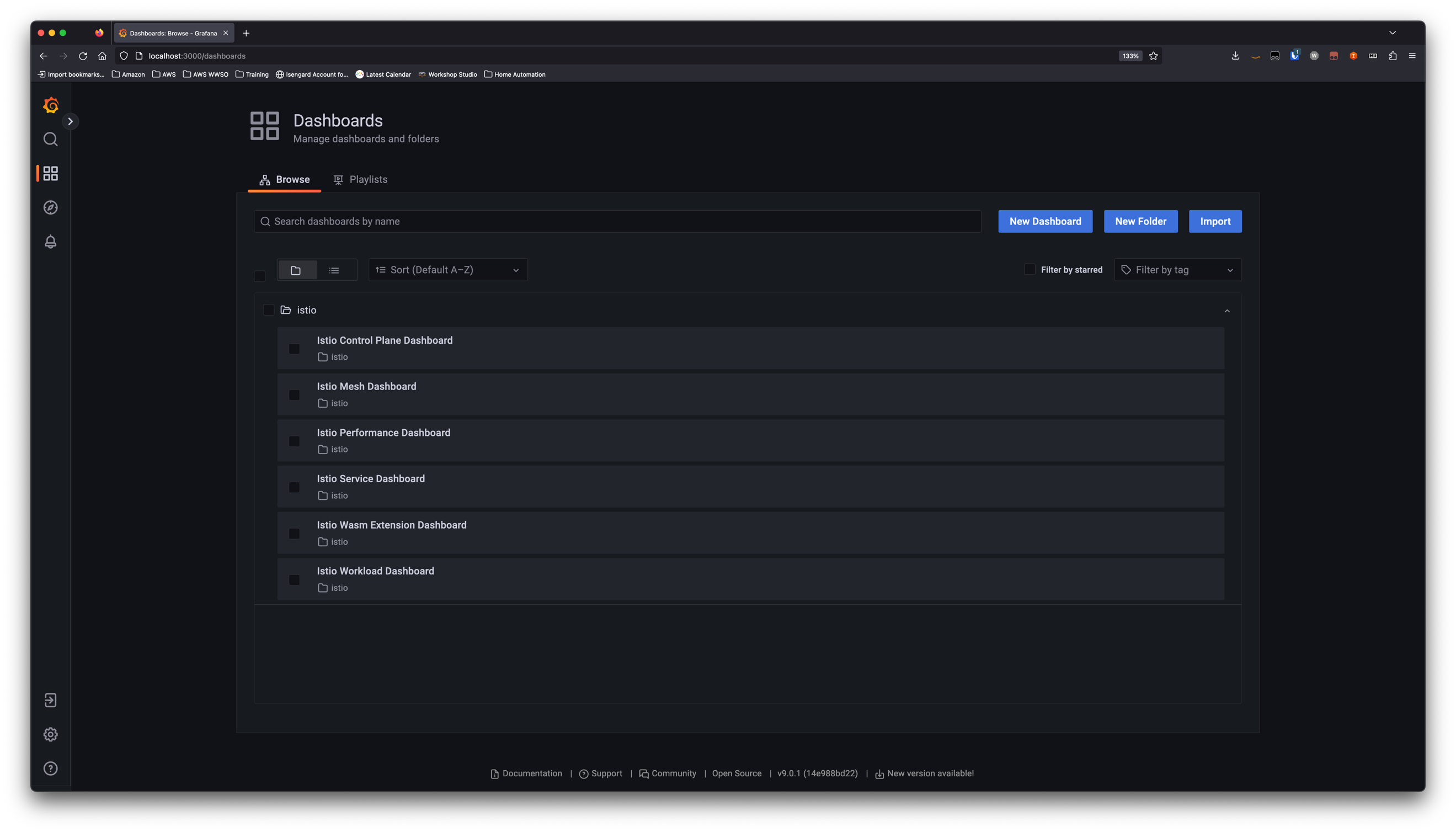
[Grafana](https://grafana.com/docs/grafana/latest/getting-started/) allows you to query, visualize, alert on and understand your metrics no matter where they are stored. As with Kiali, Grafana should also be already available as a deployment in the istio-system namespace if you have setup Istio on EKS with the blueprint provided.

Note: A complete overview of the Grafana is beyond the scope of this blog post

To access the Grafana console, port-forward the Grafana service endpoint onto your local machine with the following command:

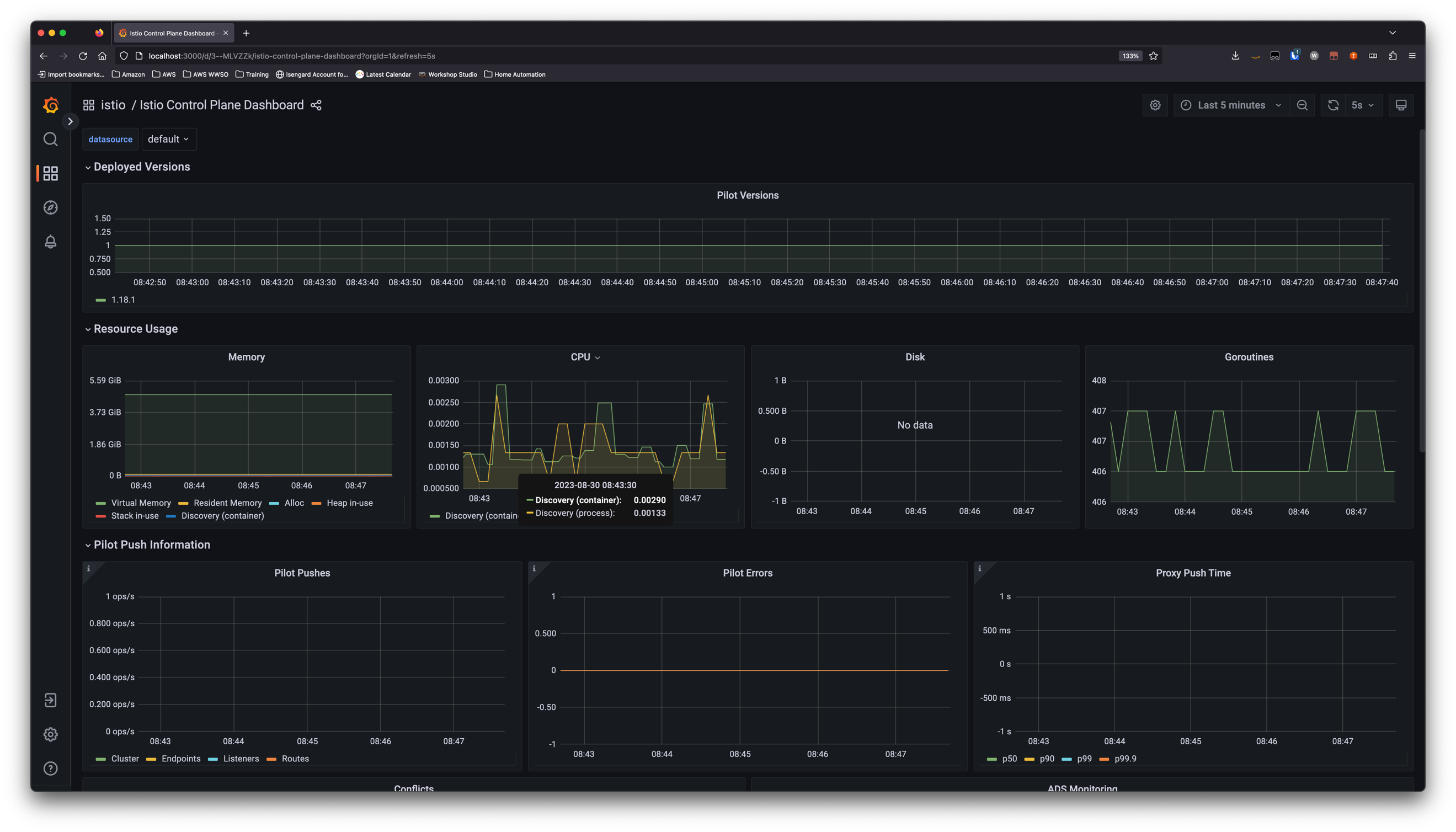
kubectl port-forward svc/grafana 3000:3000 -n istio-system

Use your browser to navigate to http://localhost:3000/dashboards.



Navigate into each of the provided Dashboards by clicking on the names as shown in this image. For example, when you navigate to the Istio Control Plane Dashboard you should see a dashboard similar to the image shown here.

Control Plane Dashboard monitors the health and performance of the control plane. Use this link to get details on all dashboards available out of the box: <https://istio.io/latest/docs/ops/integrations/grafana/>



**Testing**

We have so far visualized how the services link to one another in Kiali when there wasn’t any traffic. In this section we will go a step further and generate traffic to visualize in even more detail the following aspects and more:

* Traffic rate
* Traffic distribution
* Throughput
* Response time
* Traffic animation between services

**Generating Traffic**

Let’s generate some traffic for our application. Use the siege command line tool to generate traffic to the application’s (user interface) URL by running the following commands in a separate terminal session.

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

# Generate load for 2 minute, with 5 concurrent threads and with a delay of 10s between successive requests

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

While the load is being generated, access the kiali console you previously configured and you should notice that the traffic is flowing in the manner shown in this screenshot. The traffic originates, from the left at the istio-ingress, hops through frontend and productcatalog, ending finally with catalogdetail service which then forwards/round-robin (depends on [kube-proxy proxy modes](https://kubernetes.io/docs/reference/networking/virtual-ips/#proxy-modes)) the traffic to either of its versioned workloads.

A screenshot of a computer

Description automatically generated

**Observations**

Based on traffic animation captured in Kiali as a result of our load test, we can conclude that:

* The Ingress traffic directed towards the istio-ingress is captured by the Gateway productapp-gateway as it handles traffic for all hosts (\*)
* Traffic is then directed towards productapp VirtualService as its host definition matches all hosts (\*)
* From productapp VirtualService, the traffic reaches frontend microservice as the context-path matches /, from there moves to productcatalog and then finally to catalogdetail.
* The catalogdetail service, as expected, randomly splits the traffic between v1 and v2 versions.

**Cleanup**

To clean up your EKS environment and remove the services that were deployed, please run the following commands:

helm uninstall mesh-basic -n workshop

kubectl delete namespace workshop

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

**Conclusion**

As we wrap up this inaugural blog in our Istio on Amazon EKS series, we hope you’ve gained valuable insights into the potential of Istio within Amazon EKS. From deploying microservices to mastering Istio’s Gateway and VirtualService, you’ve embarked on a journey towards a more resilient, secure, and observable microservices architecture. But hold onto that excitement because the adventure is far from over. In our next installment, we’ll delve deeper into Istio’s advanced features, exploring topics like traffic management, mTLS, 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!