Meshery - The Service Mesh Management Plane Case Study

Krzysztof Kwiecień, Łukasz Sochacki, Szymon Sumara, Patryk Studziński
Group: Meshery-20

June 15, 2023

Contents

1 Introduction

Meshery is an open-source service mesh management platform that simplifies the deployment and management of multiple service meshes in Kubernetes and other containerized environments. It provides built-in performance testing and observability tools, allowing users to monitor and optimize their application's performance across multiple meshes and clusters.

Meshery helps users achieve greater visibility, control, and scalability in their complex multi-mesh environment while also simplifying their operations and reducing the risk of service disruptions.

More information about the project can be found on the official Meshery website [Mes] and webinar about Meshery itself [Web].

2 Theoretical background/technology stack

As the adoption of microservices architectures continues to grow, managing and monitoring these services becomes increasingly complex. Service meshes like Istio, Linkerd, and Consul provide a solution to this problem, but their configuration and management can be time-consuming and complex. This is particularly true when organizations use multiple cloud providers and platforms, which require different configurations and setups for service meshes.

Meshery simplifies service mesh management by providing a unified platform that supports multiple service mesh providers and cloud platforms. The platform offers a simple and intuitive interface for service mesh deployment, configuration, and testing, making it easy for developers and operators to manage service meshes across different environments.

Meshery also provides a range of testing and observability features, including performance testing, chaos engineering, and distributed tracing, which enable organizations to ensure the reliability and resilience of their microservices architectures, even as they scale and evolve.

We plan to deploy our system, consisting of several microservices, using an AWS Elastic Kubernetes Cluster. We will use Istio as the service mesh. We will manage the entire system using Meshery.

Below are brief descriptions of the technologies we have selected in addition to Meshery which has been described in previous section:

- **AWS EKS** [Eks] (Elastic Kubernetes Service) is a managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications using Kubernetes in the AWS Cloud.
- **Istio** [Ist] is an open-source service mesh platform that provides common networking, security, and observability features for managing communication between microservices in a distributed application. It is built on top of Envoy and provides powerful features such as traffic management, service discovery, load balancing, circuit breaking, security, and observability.
- **Docker** [Doc] open-source containerization platform used to create, deploy, and run applications in isolated containers. We will be using Docker to containerize our applications for their deployment in EKS.

3 Case study concept description

3.1 Architecture

As our case study we plan to simulate a liquid factory system that consists of multiple microservices communicating with each other using different protocols. Every service will be responsible for fulfilling different role and communicate with at least one other microservice. Those services can be divided into specialized groups:

- **Order handler** Microservice will collect orders and return packed liquids for collected orders. Prepared liquids are received from **bottling plant**.
- **Bottling plant** Produces packed liquids requested by handler and returns then when order conditions are met. In order to produce a packed beverage, all **producers** responsible for creating required components have to provide liquid and package for it.
- **Producers** These are microservices responsible for production of basic components that are later combined into packed liquids. Each producer only produces one type of product.

3.2 Scenarios

To fully utilize Meshery's and Istio's observability features and utilize the performance testing, we plan to launch multiple scenarios which will be used to monitor different parts of systems through provided features. For that goal in mind, we would like to prepare three basic scenarios:

- **Fault detection** We want to show the possibility of detecting poorly performing microservices. There will be several microservices in different versions. Some of them will produce defective products or have lower performance. The demo will consist in creating a configuration with different versions of microservices, uploading it using Meshery and performing tests. Based on the observation of statistics, detecting incorrect versions and replacing them with better ones.
- **Canary Release** After creating a new version of the microservice, instead of replacing all versions with the latest version, only some of the queries are routed to the updated version, which allows users to test it, and the impact of any errors will be smaller. We prepare a configuration in which we specify what part of the traffic is routed to the new version. Upload it with Meshery and do the tests. Based on statistics, showing that only some of the queries go to the new services.
- **Content-based routing** We will also show the possibility of routing depending on the role. It will consist in creating a configuration in which developers/testers will have access to different versions of microservices than classic users.

3.3 Communication monitoring

Additionally, Istio allows for monitoring communication between microservices where collected data can be used to calculate all kinds of metrics. We would like to use that feature for:

- **Request tracing** Monitor how requests travel between each microservice and the amount of incoming and outgoing requests for each service.
- **Load balancing** Observe how requests are distributed between subproducers attached to one master producer.
- **Delay calculation** Calculate the overall delay in communication depending on which microservices are used.

4 Solution architecture

The microservices will be built as Docker images, ready to deploy on EKS. Meshery with Istio will also be deployed and Grafana will be configured for metric analysis.

To achieve all of our goals and perform tests, we will build the following microservices:

- **WebServer** - Will provide a front-end GUI to send requests to the system. It will be possible to order a specified amount of drinks. The requests will be passed to the BottlingPlant using REST. It will be written in JavaScript.
- **BottlingPlant** - The central microservice that communicates will all other microservices via different protocols. It receives orders via REST API. It communicates with two other backend microservices (DrinkProducer, BottleProducer) via gRPC. It requests a certain amount of components from them and joins them together. It then sends the combined product back to the client as a response to the original request. This microservice will be written in Java.
- **DrinkProducer** - Microservice that produces drinks to order, communicates with the BottlingPlant via gRPC. This microservice is intentionally not fully reliable and can sometimes fail in order to simulate a more complex and realistic system. The frequency of failures can be configured at runtime. This service will have multiple versions, each one with a different failure rate. This behavior will allow us to test canary releases. The microservice will be written in JavaScript.
- **BottleProducer** - Similar to DrinkProducer, it will communicate with the BottlingPlant via gRPC and produce one of the components needing to complete the orders. It will also be prone to failures. It will be written in Java.

We will utilize Istio and Meshery to study the following topics:

Istio:

- Request tracing
- Communication load balancing
- Canary release
- Metrics collection

Meshery:

- Environment setup
- Configuration updates
- Connection with Istio and Grafana
- Performance testing

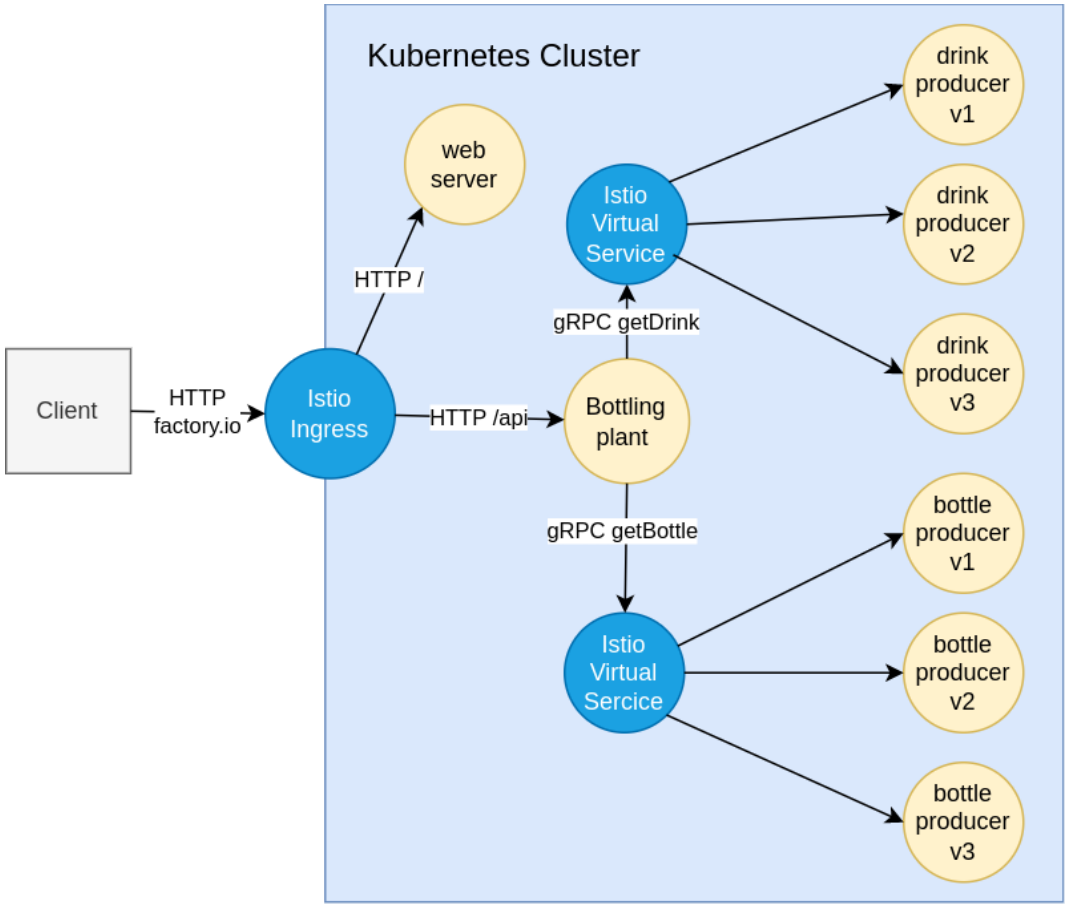


Figure 1: Architecture overview

5 Environment configuration description

For the study we are using a Kubernetes cluster running in AWS EKS. In this section we present our configuration of the cluster, in which we will deploy our applications and install Meshery and other tools used for the study. The EKS cluster was created using AWS Console.

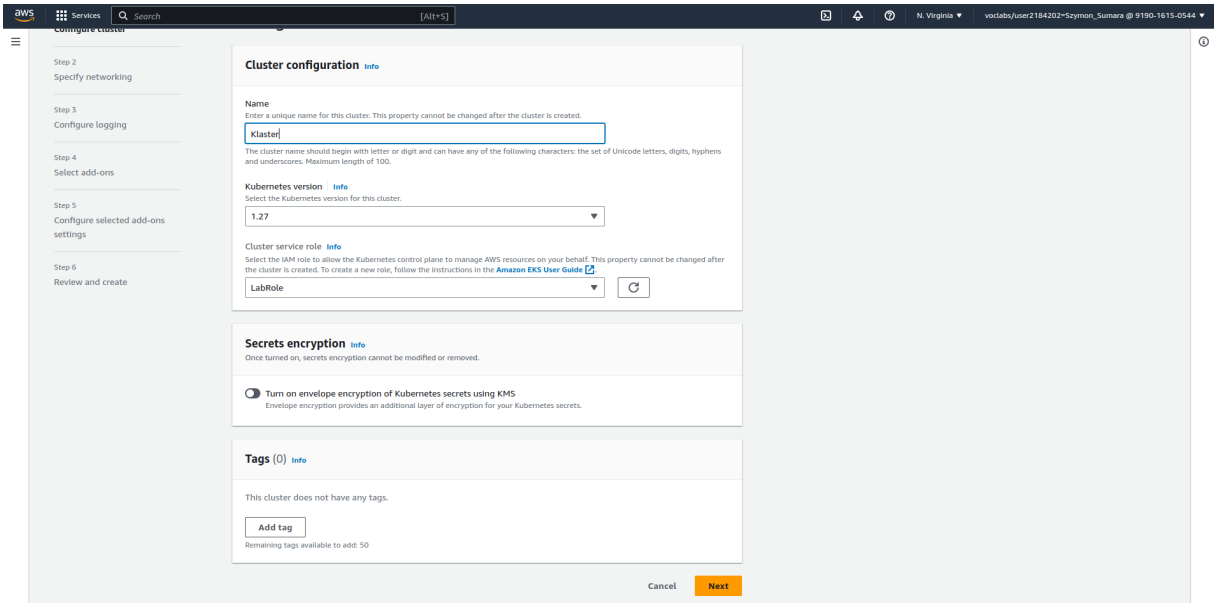


Figure 2: Cluster creation process

The cluster is running on a node group, consisting of three t3.medium instances, each providing 4GBs of memory and 2 vCPUs. The cluster was deployed to 4 subnets, with public access for easy remote configuration.

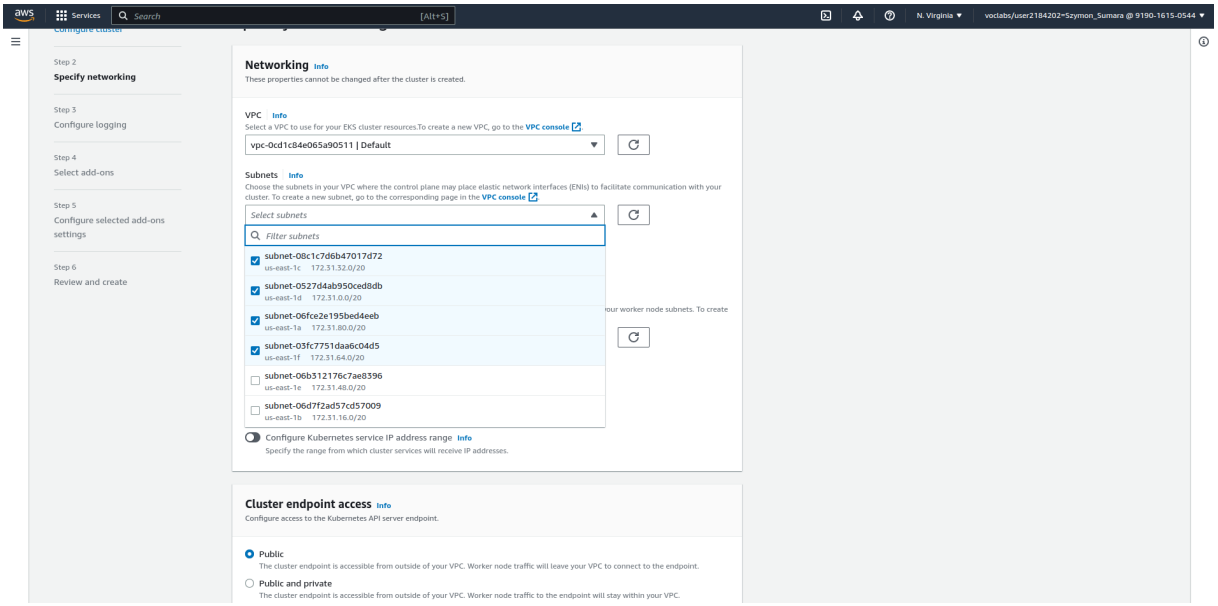
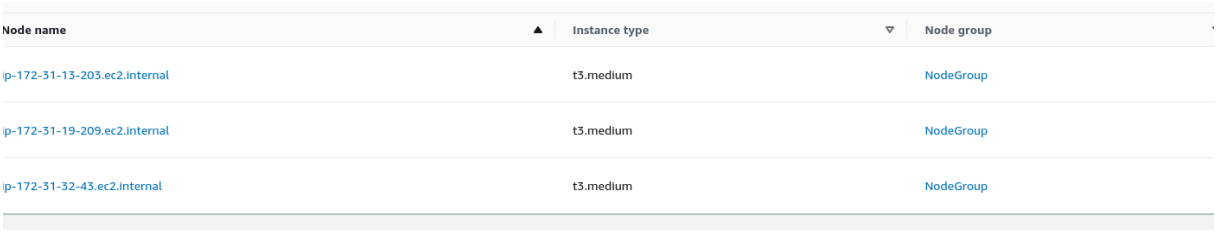


Figure 3: Node group subnets selection



Node name	Instance type	Node group
ip-172-31-13-203.ec2.internal	t3.medium	NodeGroup
ip-172-31-19-209.ec2.internal	t3.medium	NodeGroup
ip-172-31-32-43.ec2.internal	t3.medium	NodeGroup

Figure 4: Node group summarize

The configuration of the cluster was done using CLI tools: AWS CLI and kubectl. The following steps allows for remote access and configuration of the cluster:

1. Install AWS CLI
2. Install *kubectl*
3. Authenticate with AWS by running *aws configure* command and providing AWS credentials
4. Authenticate *kubectl* to AWS EKS cluster using command *aws eks –region us-east-1 update-kubeconfig –name Klaster*

Now it's possible to use kubectl locally to configure the remote cluster on EKS

6 Installation method

6.1 Overall installation process

First, we install Meshery in the Kubernetes cluster, followed by the installation of Istio in the same cluster. We then turn on injection of the Istio proxy in the namespace we intend to use. Finally, we are ready to install our application in the cluster.

6.2 Meshery installation

Installation process for Meshery can easily be described as below.

1. Install mesheryctl (How to install it is described on official **Meshery Website)**
2. Meshery has to be installed on kubernetes cluster. It can be done by using ***mesheryctl system context create eks -p kubernetes -s*** command
3. Start Meshery in the cluster with ***mmesheryctl system start***
4. Enable *automatic sidecar injection* in order to allow istio sidecars to be automatically injected into deployed pods

6.3 Istio installation

Additionally, an Istio can to be manually added in order to use it with Meshery. Installation process lokks as follows:

1. Install istioctl (Installation guide can be found on official **Istio Website)**
2. Install Istio inside Meshery (it can be done by manually adding Istio through Meshery dashboard)

6.4 Plugins

6.4.1 Kiali

Kiali [Kia] is a graphical observability console for Istio. It provides an intuitive and visual representation of your service mesh topology and allows you to monitor the traffic flow between services. Kiali also offers features like service dependency graphs, traffic tracing, and distributed tracing. It helps you understand the behavior of your microservices and diagnose issues related to traffic management and service communication within the Istio mesh.

6.4.2 Prometheus

Prometheus [Pro] is a monitoring and alerting toolkit widely used in the Kubernetes ecosystem. In the context of Istio, Prometheus collects metrics from the Istio components, such as proxies, and provides a time-series database for storing and querying these metrics. It enables you to monitor the health, performance, and behavior of your Istio service mesh.

6.4.3 Grafana

Grafana [Gra] is a popular open-source platform for data visualization and monitoring. It integrates well with Prometheus and allows you to create customizable dashboards to visualize the metrics collected by Prometheus. With Grafana, you can build real-time charts, graphs, and alerts based on the Istio and application-level metrics gathered by Prometheus.

6.4.4 Installation process

At last, we need to add plugins such as Kiali ,Prometheus and Grafana for metrics calculation and performance tracking. It can be easily done through Meshery dashboards by enabling plugins we would like to add. They will be installed automatically after selecting them. With plugins installed, we can deploy our applications and will be able to use Meshery.

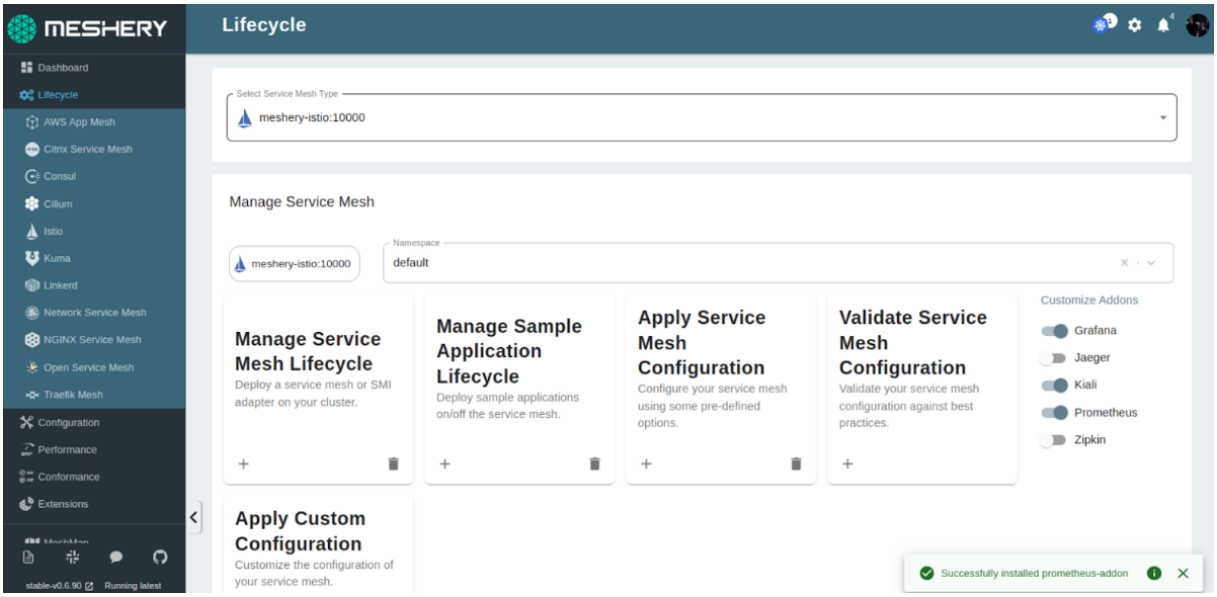


Figure 5: Prometheus successful installation

7 How to reproduce - step by step

As the first step we apply start Kubernetes manifests by using '*kubectl apply -f k8s/start/*' command. Executing it will grant us access to our application.

Next, in order to receive address of our ingress gateway, we have to execute '*kubectl -n istio-system get svc istio-ingressgateway*' command. The address gives access directly to our application.

Services are up and running after performing mentioned step. To enter our application, received address has to be pasted into a browser which will then redirect user to application's frontend. A table with summary of requests sent and their details can be seen as a result.

7.1 Infrastructure as Code approach

Our services and their deployment are defined and organized using helm charts. By storing all of the setup and configuration as code we can make sure that everything gets deployed properly and we can reliably recreate the environment.

Using helm charts we have configured the deployment, destination rules, load-balancers, virtual services and a gateway. By preparing separate configurations as code, we can easily change the behavior of the system, such as routing.

We can also easily prepare a canary release, which is easy to check, verify and deploy, thanks to it being defined as code.

8 Demo deployment steps

8.1 Deploying our app with Meshery dashboard

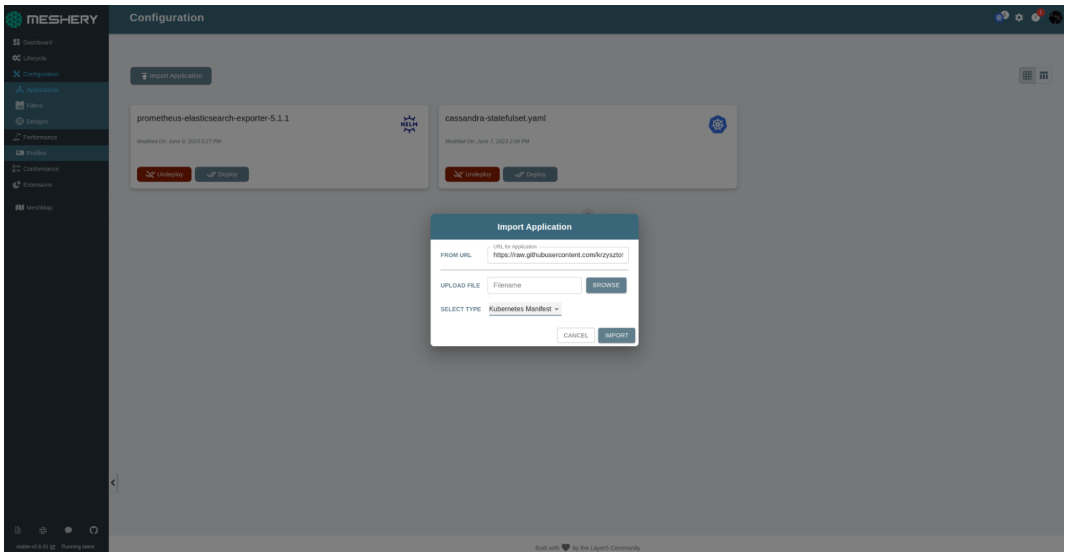


Figure 6: Configuration section in Meshery dashboard

In the first step, we need to enter the Configuration section in the Meshery dashboard. Then, we can import the Kubernetes manifest to deploy our application. All of the Kubernetes manifests and files for our presentation project are available in our repository. After importing the application, we should see it in the panel. We simply need to click the deploy button, and our application will be available in a couple of minutes.

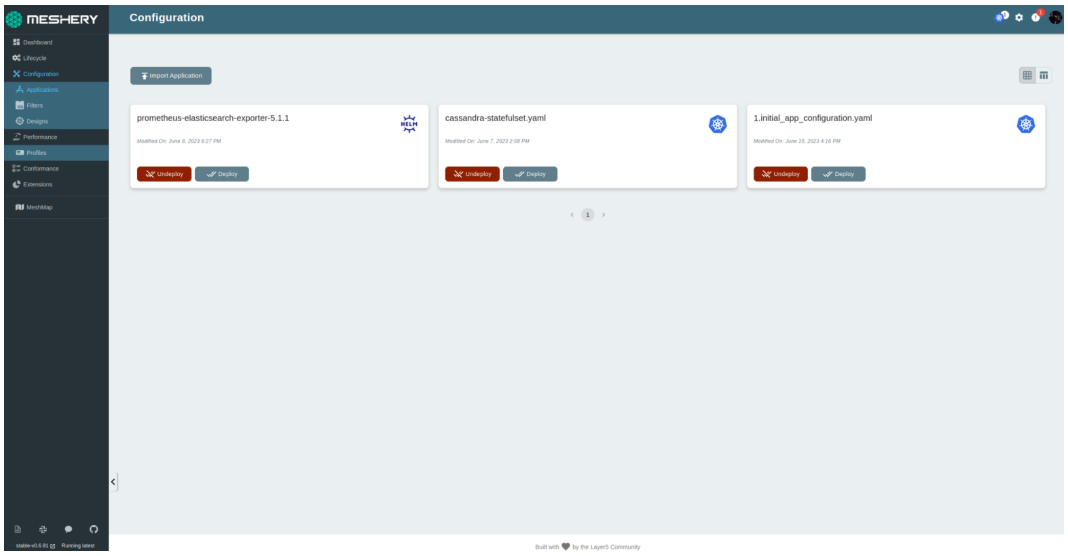


Figure 7: Imported configuration

To get the URL of our app, we need to type the console command 'kubectl -n istio-system get svc istio-ingressgateway' and we should receive a similar output as shown in image 8. When we copy the value from the EXTERNAL-IP field and paste it into a browser, we should see our app as depicted in picture 9.

```
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP
istio-ingressgateway    LoadBalancer   10.100.59.18   a0a5a9152e2f44d95a658c0e9d239a98-1659466792.us-east-1.elb.amazonaws.com
```

Figure 8: Example command output

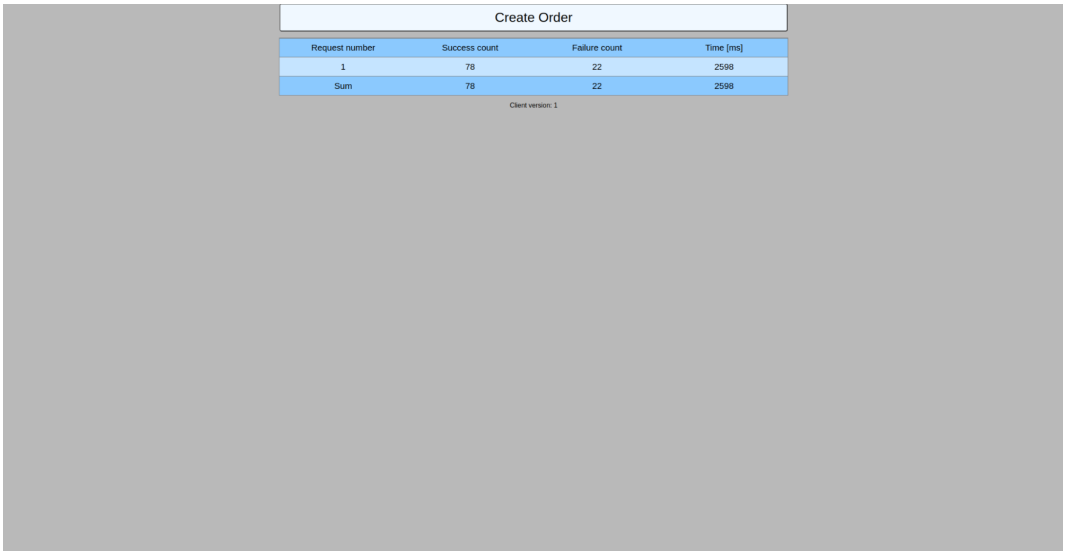


Figure 9: Frontend app

8.2 Reduce query errors

Now, our queries are returning many errors and taking a lot of time. In the next steps, we will try to fix this. Firstly, we can create a performance test to generate some traffic. Go to the performance section and create a test with parameters similar to those shown in picture 10.

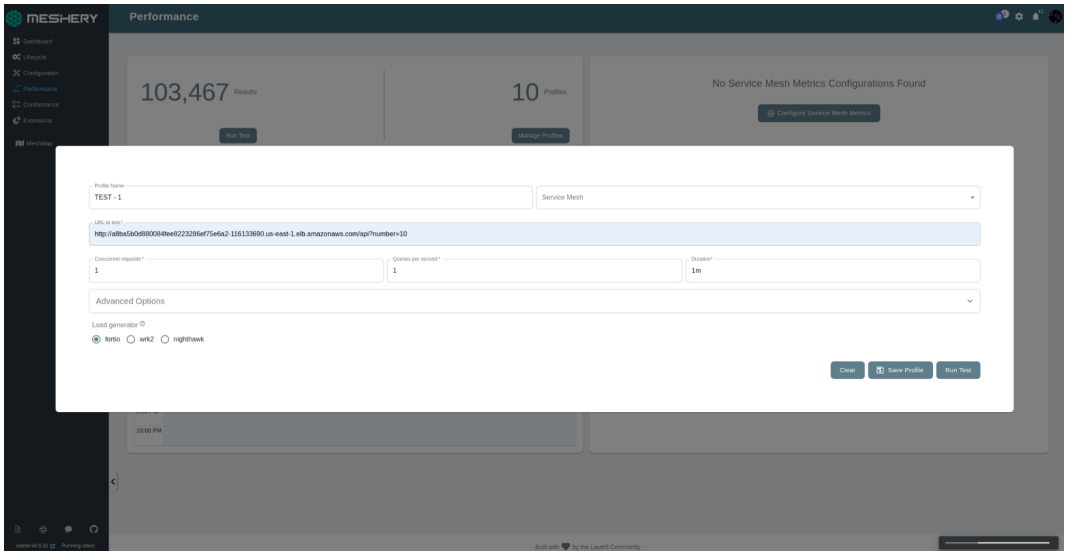


Figure 10: Performance test properties

When the performance test is running, we can enter the Kiali dashboard by running the command 'istioctl dashboard kiali'. In the graph section, we should see something similar to picture 11.

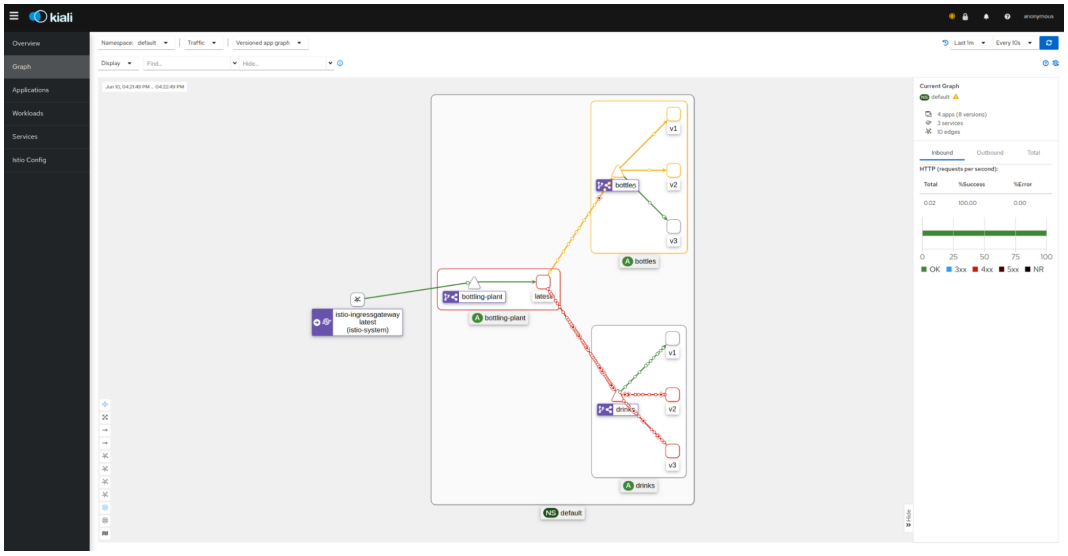


Figure 11: Performance test properties

In Kiali, we see that most of the errors are caused by the drinks services in versions v2 and v3. To address this, we will delete these services in the next step. To do this, we need to apply the following Kubernetes manifest located at the link [Kubb]. After completing this step, our app's structure should resemble picture 12

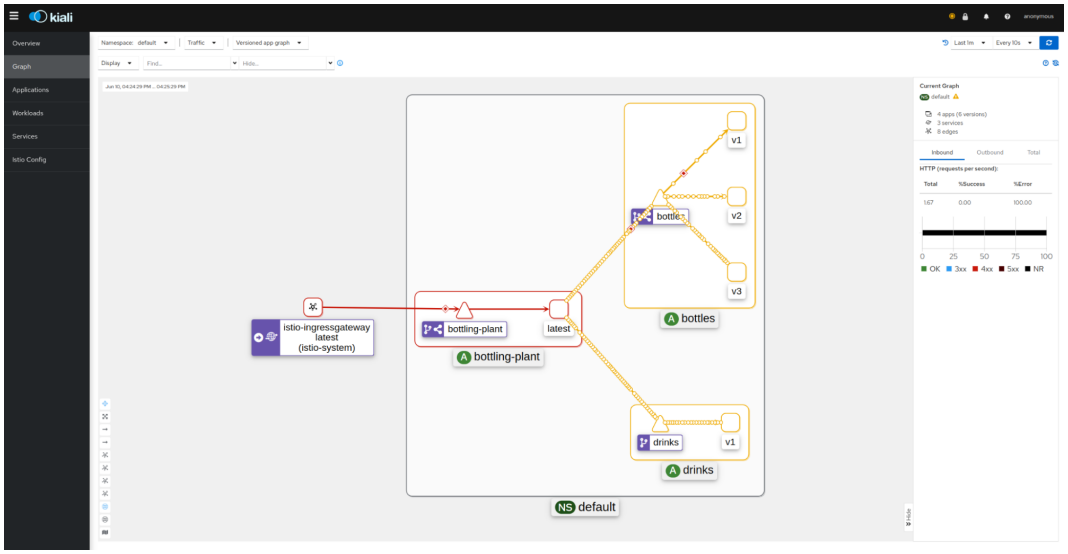


Figure 12: Application structure after applying second Kubernetes manifest

Now, we can make some requests from the app's frontend. As we can see, the number of errors is noticeably lower (image 13).

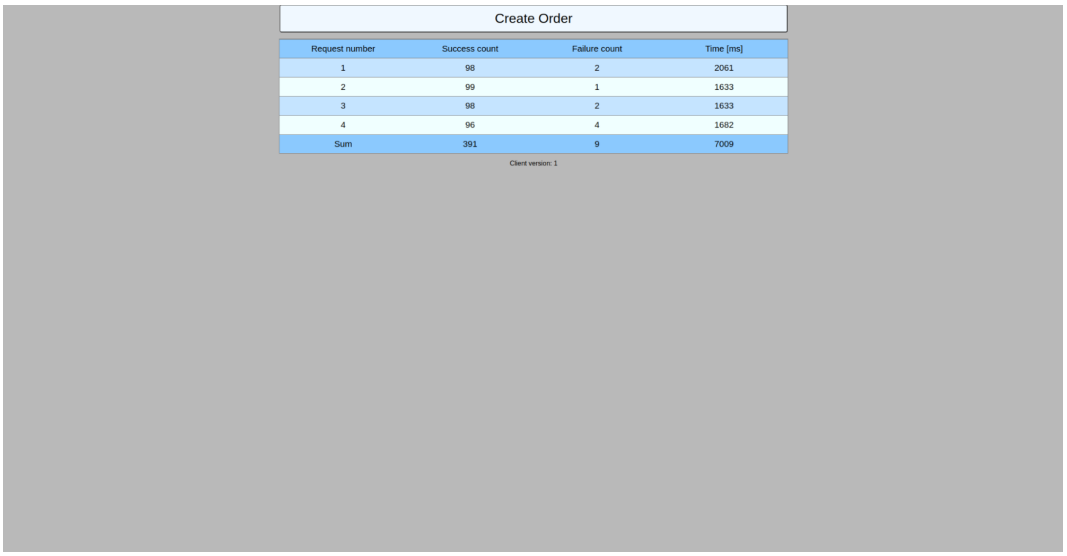


Figure 13: Errors number after applying second Kubernetes manifest

8.3 Reduce query time

In this step, we will try to reduce the query time. To view the query time of each service, we can open Grafana. In the console, we need to type 'istioctl dashboard grafana', and Grafana will automatically open in the browser. Choose 'Istio mesh dashboard', and then we can see (image 16) that the services 'bottles v1' and 'bottles v2' are much slower compared to the 'bottles' service in version 3. Therefore, we will redirect most of the traffic to the service in version v3 and create a new instance with the same version. These changes are stored in the Kubernetes manifest at the following link [Kubc]. After applying this operation, the request time is reduced by 50% (image 15), and the app structure in Kiali has changed (image 17).

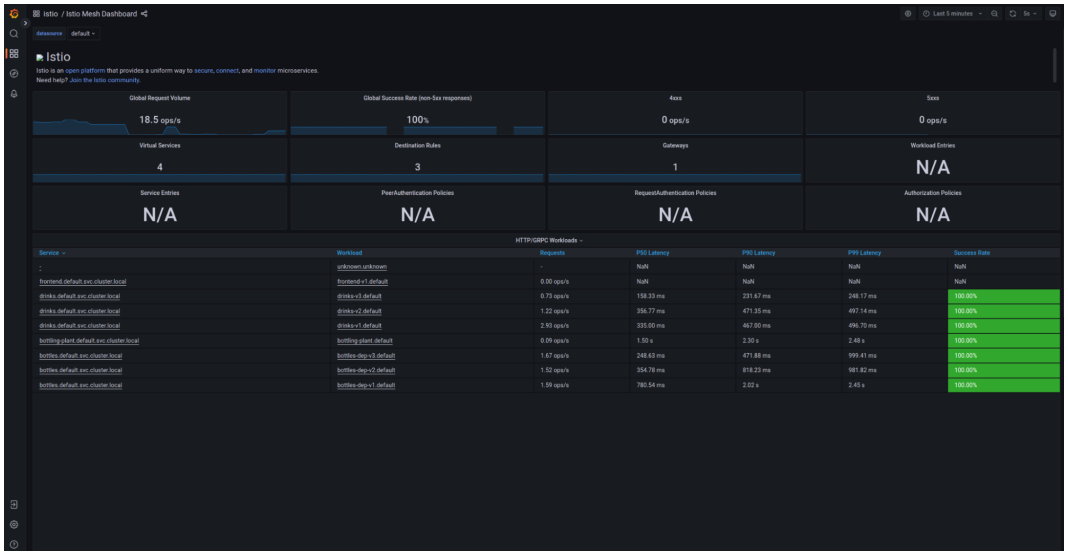
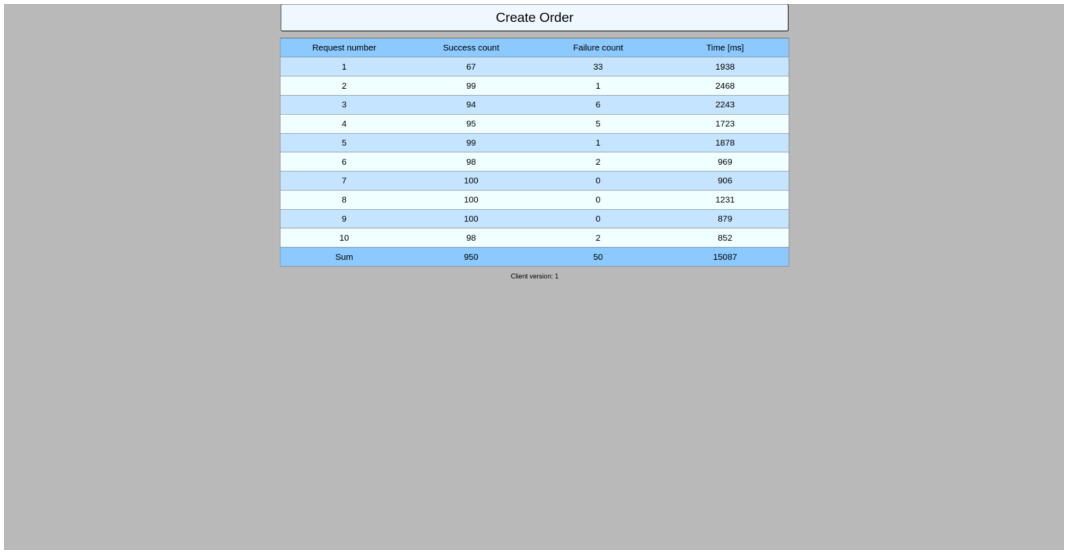


Figure 14: Grafana dashboard



Create Order

Request number	Success count	Failure count	Time [ms]
1	67	33	1938
2	99	1	2468
3	94	6	2243
4	95	5	1723
5	99	1	1878
6	98	2	969
7	100	0	906
8	100	0	1231
9	100	0	879
10	98	2	852
Sum	950	50	15087

Client version: 1

Figure 15: Times after applying third manifest

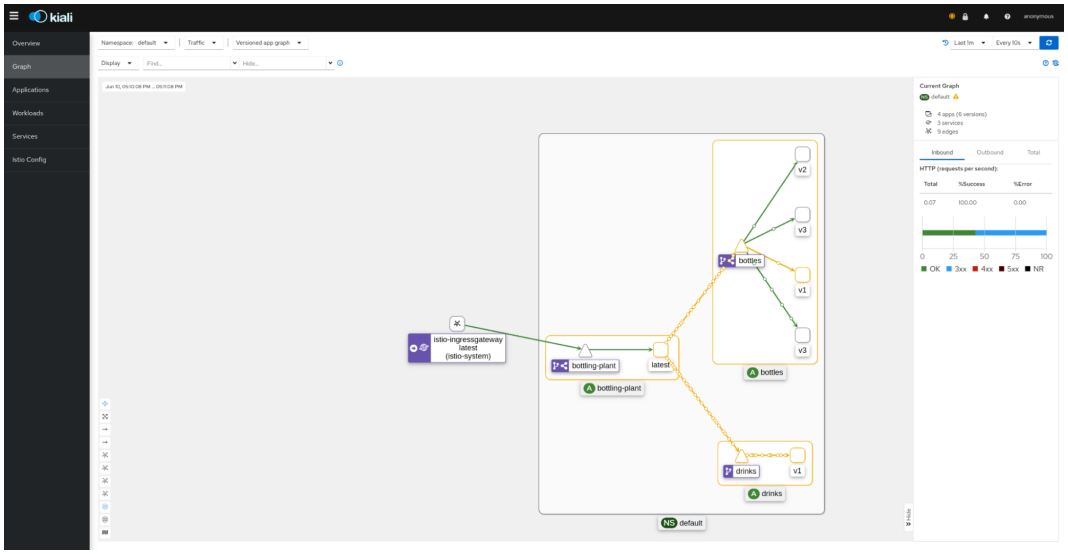


Figure 16: Kiali graph structure after applying third manifest

8.4 Canary release

In this section we show how fast canary release can be realized with Istio. For our app we have prepared Kubernetes manifest that changed weights for services and added new one. To do this step we need to apply Kubernetes manifest located at [Kuba]. We showed changes at Kiali graph (images 17 and 18). Before our changes most of traffic went to version 1 and some to version 2. After, most of traffic went to version 2 and some to version 3.

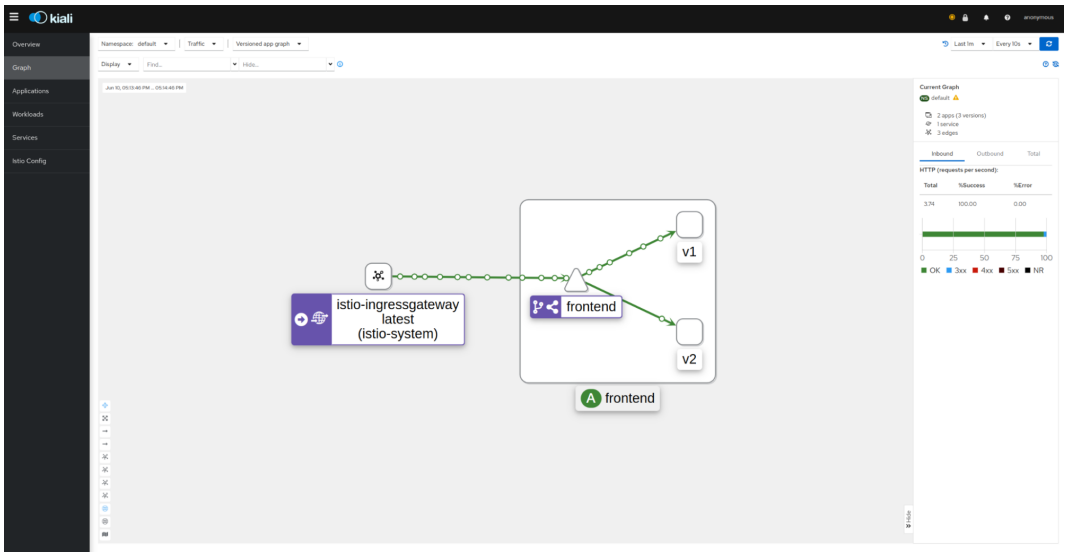


Figure 17: Kiali graph shows frontend services before applying Kubernetes manifest

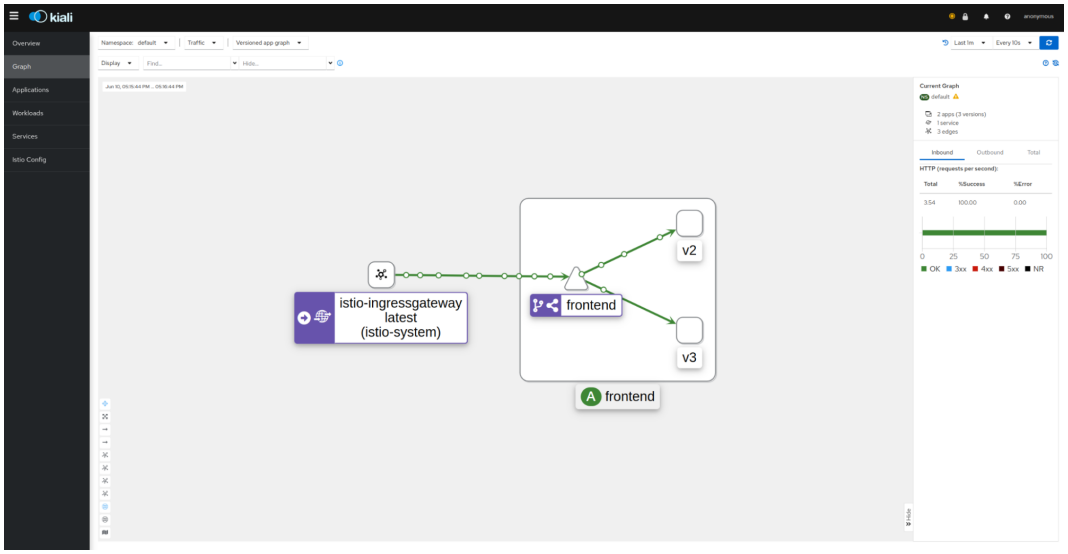


Figure 18: Kiali graph shows frontend services after applying Kubernetes manifest

9 Summary – conclusions

In our case study, we showed some of the many possible uses of Meshery.

Using Meshery itself is very simple and requires only a few installation commands. Having an available dashboard, we can easily install additional tools such as Istio, Grafana, Prometheus. Another big advantage of Meshery is the ease of deploying applications. In our project, having prepared configurations, we could configure our application with a few clicks, which was very convenient and fast.

The tool also allows you to test the application, which is very useful, but unfortunately has some imperfections. Among other things, the interface is not very intuitive and there is no possibility to stop the tests. The results obtained in the tests were much less detailed compared to other tools such as Kiali or Grafana.

Considering our experience with using Meshery and the amount of all the possibilities offered by Meshery, it is certainly a very helpful and useful tool for managing and monitoring applications.

References

[Doc] Docker. https://www.docker.com/.

[Eks] Aws elastic kubernetes service. https://aws.amazon.com/eks/.

[Gra] Grafana. https://grafana.com/.

[Ist] Istio. https://istio.io/.

[Kia] Kiali. https://kiali.io/.

[Kuba] Kubernetes-manifest-canary-release. https://raw.githubusercontent.com/krzysztofxkwiecien/meshery-case-study/main/k8s/dep/4.cannaryrelase.yaml.

[Kubb] Kubernetes-manifest-errors. https://raw.githubusercontent.com/krzysztofxkwiecien/meshery-case-study/main/k8s/dep/2.reducedrinksserviceerrors.yaml.

[Kubc] Kubernetes-manifest-time. https://raw.githubusercontent.com/krzysztofxkwiecien/meshery-case-study/main/k8s/dep/3.reducebottleservicestime.

[Mes] Meshery. https://meshery.io/.

[Pro] Prometheus. https://prometheus.io/.

[Web] Meshery webinar. https://www.cncf.io/online-programs/cncf-on-demand-webinar-meshery-the-service-mesh-manager.