Meshery - The Service Mesh Management Plane Case Study

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

April 28, 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.

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** [1] (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** [2] 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** [3] open-source containerization platform used to create, deploy, and run applications in isolated containers. We will be using Docker to conternaize 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:

[1] https://aws.amazon.com/eks/
[2] https://istio.io/
[3] https://www.docker.com/

- **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:

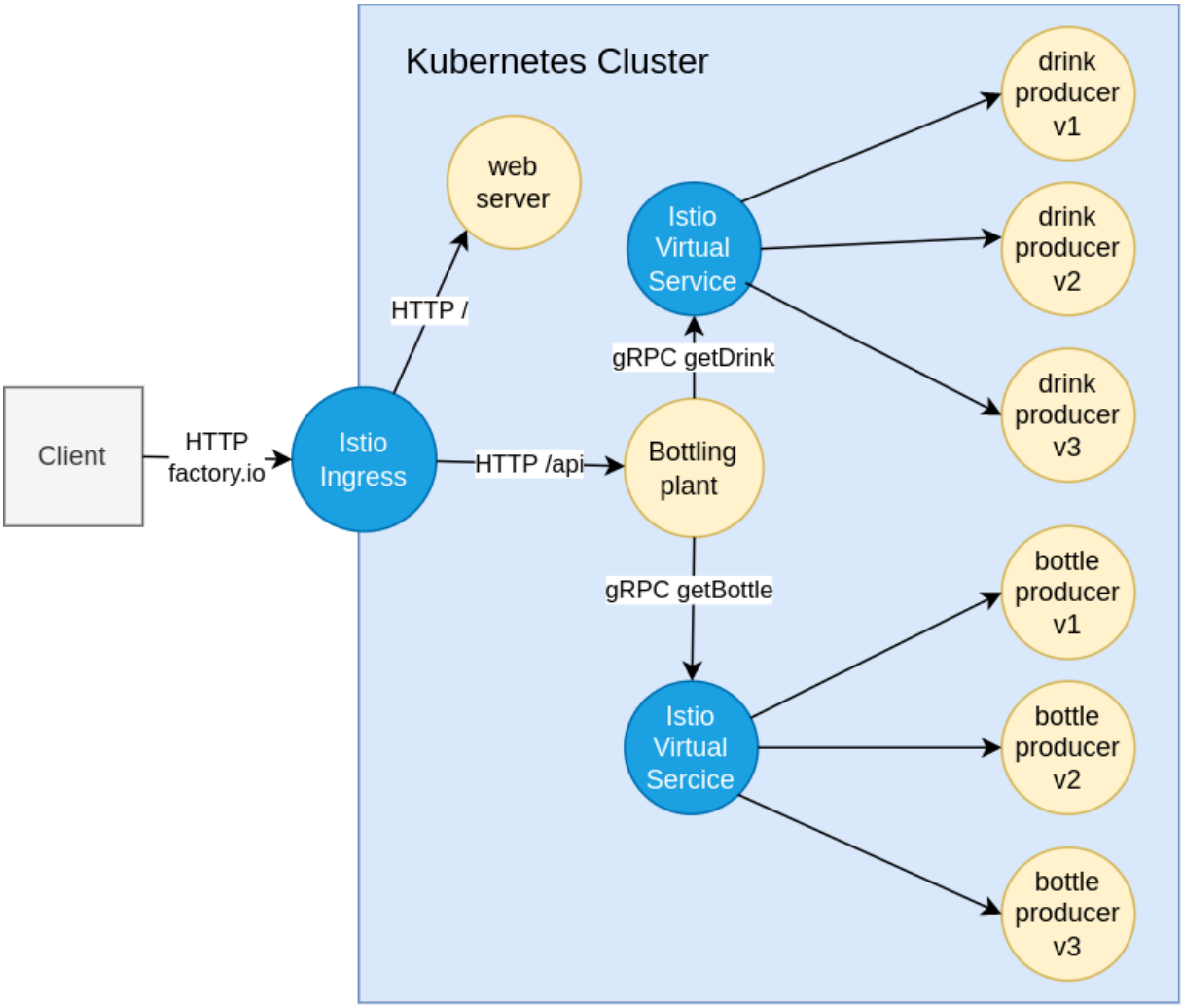
- **Request tracing** Monitor how requests travel between each microservice and the amount of incoming and outgoing requests for each service.
- **Load balancing** Observe how requests are distributed between subproducers attached to one master producer.
- **Delay calculation** Calculate the overall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

4 Solution architecture

The microservices will be built as Docker images, ready to deplon 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

5 Environment configuration description

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 Installation method

7 How to reproduce - step by step

7.1 Infrastructure as Code approach

8 Demo deployment steps

8.1 Configuration set-up

8.2 Data preparation

8.3 Execution procedure

8.4 Results presentation

9 Summary – conclusions

References

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